UNIVERSITY OF CALIFORNIA,
IRVINE

Computational State Transfer: An Architectural Style for Decentralized Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Michael Martin Gorlick

Dissertation Committee:
Professor Richard N. Taylor, Chair
Associate Professor James A. Jones
Professor Crista V. Lopes

2016

DEDICATION

To the memory of

my mother
Doreen Viola Gorlick (née Rose)
December 1927 – February 2015

and

my brother
Patrick David Gorlick
February 1953 – December 2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

If it takes a village to raise a child it takes a small mob to usher a doctoral student through advancement to candidacy and from there to final defense.

In the spring of 2004 I approached my colleague and friend Dick Taylor with an unusual request: would he consider, given that our professional relationship stretched back to 1976 when we were both graduate students at the University of British Columbia, taking me on as a doctoral student? I will forever treasure his expression of shock and surprise when I popped that question. He bravely and recklessly assented; following the customary tedium of exams, forms and fees, I officially enrolled in the doctoral program at the University of California, Irvine on October 1, 2004 at the tender age of 53. I am deeply indebted to Dick for his willingness to take a flyer on me when any other rational man of letters would have dismissed me as an old fool.

It has been a challenging slog from that first day to my thesis defense more than 11 years later. There was a running joke in the department as to which would come first, my doctorate or a Social Security check. I'm pleased to report that those of you in the betting pool who put your money on the Social Security check lost. Along the way I benefited greatly from the kindness and support of friends, many of them my fellow doctoral students. My first office mate was Eric Dashofy, who made the mistake of seriously entertaining my speculations on software architecture and then responding with articulate, well-informed arguments and observations. I repaid his kindness, upon his defense and graduation, by recommending him for a position at The Aerospace Corporation, my employer then, and now. His first office at Aerospace was next door to mine. Eric has risen quickly through the Aerospace ranks; I'm not sure that he will ever forgive me for entrapping him.

My second office mate was Justin Erenkrantz, former president of the Apache Foundation and a respected figure of the open-source community. Justin became both a friend and colleague and I enjoyed our time together immensely. We quickly discovered our common interest in internet-scale architectures and began, almost immediately, vigorously debating the architectural future of the web. Those conversations were among the most spirited and enlightening of my career. A series of intense whiteboard discussions over the course of two or three days led us to reconsider the foundations of Representational State Transfer (REST) and, in its place, posit Computational REST as a successor architectural style to REST. Computational REST become the topic of Justin's doctoral thesis and is the intellectual precursor of my work. Without Justin's contributions this thesis would simply not exist, and for that alone, I owe Justin a tremendous debt.

I also owe thanks to several of Dick's other graduate students whose spirited questions and discussions were an ongoing, ever unfolding pleasure: Alegria Bacquero, Matias Giorgio, Kyle Strasser, and Michael Wolfe. Each, whether they realize it or not, made material contributions to the work presented here.

Alegria Bacquero took an early formulation of Computational State Transfer (COAST) and applied it to electronic medical records, a wholly new problem area for me. Matias Giogio and I spent many hours discussing the fine details of, and justification for, the internal mechanisms of the COAST reference implementation. His master's thesis was a substantial contribution to the evaluation of the efficacy of the architectural style. Kyle Strasser worked tirelessly on the implementation of COASTCAST, the first major demonstration of the style. The techniques that he developed for the replication and transfer of video streams within COASTCAST inspired the protocols for live update with hot backup that appear in this thesis as an evaluation study. Michael Wolfe, in his master's thesis, demonstrated that the COAST style was compatible with the demands of well-known standards for entity authentication, an important question for me at the time.

Debra Brodbeck, Assistant Director of the Institute for Software Research, provided invaluable guidance to a bewildering university bureaucracy and was a center of calm. Kari Nies, a staff programmer/analyst at the Institute, was an early "adopter" of the COAST reference implementation and worked closely with Alegria Bacquero and Matias Giorigio in developing the first websites devoted to COAST. To Debra and Kari I offer heartfelt thanks; I will miss you both.

I would be remiss if I did not give thanks to the other two members of my doctoral committee, Associate Professor James Jones and Professor Crista Lopes. Jim Jones' interest never flagged over the years and Crista Lopes stepped up at a critical juncture. Thanks again for the signatures; I intend to have a set framed. In addition, Professor Gillian Hayes, the Vice Chair for Graduate Affairs of the School for Informatics and Computer Science, staunchly defended me against powerful bureaucratic forces over which I had no control and even less understanding. Without her efforts I would not be writing these acknowledgments and I owe her much thanks.

Over the course of my doctoral studies I continued working at The Aerospace Corporation. My colleagues and supervisors there showed me far more patience and understanding than I had any right to expect. To my succession of section and department heads: Stuart Kerr, Dr. Scott Michel, Jorge Seidel, and Dr. David Wangerin I offer thanks for your flexibility and encouragement as I bent my work schedule into pretzel shapes to meet the demands of my doctoral studies.

My Aerospace colleagues, Drs. Kirstie Bellman, Eric Coe, Nehal Desai, Sam Gasster, Jim Gillis, Henry Helvajian, Dave Kunkee, Chris Landauer and Larry Miller, (you can't toss a stone ten feet on the Aerospace campus without hitting a Ph.D. in some discipline) frequently encouraged and advised me (often along the lines of "just finish the damn thing"). They'll be glad to know that I finally took them at their word. Larry Miller deserves special thanks; he often served as a sounding board as I described the Swiss watch intricacy of the reference implementation and was always a sympathetic ear.[1]

My wife, Wendy Hansen, has been a stalwart friend and partner from beginning to end. She is a professional technical editor and if my thesis is error-free and properly punctuated it is all her doing. Over the past year she has read and critiqued countless drafts and rewrites and at the end of it probably knew the subject area as well as I and could have defended the thesis as my second. It was a long haul for both of us and though I may hold the degree she deserves much of the credit. It would have been impossible without her support and, for that, words of thanks and gratitude are frankly inadequate.

This material is based upon work supported by the National Science Foundation under Grant Numbers 0438996[2], 0820222[3], 0917129[4] and 1449159[5]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

[1] Though it came with a price. If I hear another golf story I'll simply go mad.

[2] NSF Designing Architectures for Networked Applications: A REST-ful Approach (2004/10-2009/1).

[3] NSF Collaborative Research: Recombinant Services Recasting the Web for Continuously Evolving Systems (2008/9–2012/8).

[4] NSF Making and Tracing: Architecture-centric Information Integration (2009/9–2012/8).

[5] NSF EAGER: Accountability Through Architecture for Decentralized Systems (2014/9–2016/8).

CURRICULUM VITAE

Michael Martin Gorlick

Education

Doctor of Philosophy (June 2016)

University of Callifornia, Irvine
Donald Bren School of Information and Computer Sciences
Department of Informatics
Advisor: Richard N. Taylor
Dissertation: *Computational State Transfer: An Architectural Style for Decentralized Systems*

Master of Science (October 1978)

University of British Columbia, Vancouver, British Columbia, Canada
Department of Computer Science
Advisor: John L. Baker
Thesis: *Computation by One-Way Multihead Marker Automata*

Professional Experience

2003/01 –	Senior Engineering Specialist, The Aerospace Corporation, El Segundo, California
2001/05 – 2002/12	Research Investigator, Molecular Biology Institute, University of California, Los Angeles
2000/06 – 2001/05	Chief Software Architect, Endeavors Technology, Irvine, California
1995 – 1997	Founder and Vice President, Interstitial Magic Limited, Gardena, California
1989 – 1994	Lecturer, Institute for Advanced Professional Studies, Cambridge, Massachusetts
1984/08 – 2000/05	Research Scientist, The Aerospace Corporation, El Segundo, California
1979/08 – 1984/08	Researcher & Senior Unix Systems Programmer, TRW, Redondo Beach, California
1978/10 – 1979/07	Consultant, British Columbia Telephone Company, Burnaby, British Columbia, Canada
1971 – 1975	Research Assistant, University of British Columbia, Vancouver, British Columbia, Canada

Publications

1. Michael M. Gorlick and Richard N. Taylor, Communication and Capability URLs in COAST-based Decentralized Services, In REST: Advanced Research Topics and Practical Applications, C. Pautasso, E. Wilde, and R. Alarcon, Eds. Springer, December 2013.
2. Michael M. Gorlick, Kyle Strasser and Richard N. Taylor, COAST: An Architectural Style for Decentralized On-Demand Tailored Services, WICSA/ECSA 2012 Helsinki, Finland, August 2012.
3. Larry Miller, Michael M. Gorlick, David Wangerin and Christopher Landauer, What's Coming on Spacecraft: Next-Generation Distributed Satellite Bus Information Systems, Ground System Architecture Workshop, Los Angeles, February 2012.
4. Michael M. Gorlick, Kyle Strasser, Alegria Baquero and Richard N. Taylor, CREST: Principled Foundations for Decentralized Systems, SPLASH 2011, Portland, Oregon, October 2011.
5. Larry Miller, Michael M. Gorlick, David Wangerin and Christopher Landauer, Next-Generation Distributed Satellite Bus Information Systems, Flight Software Workshop, Johns Hopkins, October, 2011.
6. Justin Erenkrantz, Michael M. Gorlick, Girish Suryanarayana and Richard N. Taylor, From Representations to Computations: The Evolution of Web Architectures, International Conference on Foundations of Software Engineering, Dubronvik, Croatia, September 2007.

7. Justin Erenkrantz, Michael M. Gorlick and Richard N. Taylor, Rethinking Web Services from First Principles, International Conference on Design Science Research in Information Systems and Technology (DESRIST), Pasadena, California, May 2007.
8. Michael M. Gorlick, Justin Erenkrantz, and Richard N. Taylor, CREST: When REST Just Isn't Enough, ISR Connector Issue 7, Fall/Winter 2007.
9. Michael M. Gorlick et al., Flow Webs: Mechanism and Architecture for Sensor Webs, Ground Systems Architecture Workshop, Manhattan Beach, California, March 2007.
10. Michael M. Gorlick and Samuel Gasster, Future Trends in Ground Systems: Rethinking Systems Engineering in the Light of Hyperexponential Change, Crosslink, Spring 2006.
11. Eric Dashofy, John Georgas, Justin Erenkrantz, Michael M. Gorlick and Rohit Khare, Designing in a RESTful World, ISR Connector, Institute for Software Research, University of California, Irvine, Spring/Summer 2005, pp. 7-8.
12. John Georgas, Michael M. Gorlick, and Richard N. Taylor, Raging Incrementalism: Harnessing Change with Open-Source Software, Proceedings of the Workshop on Open-Source Software Engineering, St. Louis, Missouri, May 2005.
13. Michael M. Gorlick and John Georgas, A Scalable Digital Video System for Launch Range Operations, Proceedings of the 2005 Ground Systems Architecture Workshop, Manhattan Beach, California, March 2005.
14. Michael M. Gorlick, Raging Incrementalism — System Engineering for Continuous Change, Proceedings of the 2004 Ground Systems Architecture Workshop, Manhattan Beach, California, April 2004.
15. Alissa Resch, Michael M. Gorlick, et al., Assessing the Impact of Alternative Splicing on Domain Interactions in the Human Proteome, Journal of Proteome Research, November 2003.
16. D. Stott Parker, Michael M. Gorlick and Christopher Lee, Evolving from Bioinformatics in the Small to Bioinformatics in the Large, OMICS, 7, 34-48, 2003.
17. Michael M. Gorlick, Geek Chic and the Future of Space Systems, Workshop on Evaluating Software Architectural Solutions, University of California Irvine, Irvine, California, October 2000.
18. Michael M. Gorlick, Location Awareness is Where It's At, The Workshop on Internet-Scale Technologies: Internet- Scale Namespaces, University of California Irvine, Irvine, California, August 1999.
19. E. James Whitehead Jr., Rohit Khare, Richard N. Taylor, David S. Rosenblum and Michael M. Gorlick, Architecture, Protocols, and Trust for Info-Immersed Active Networks, Proceedings of the Interagency Workshop on Smart Environments, Atlanta, Georgia, July 1999.
20. Michael M. Gorlick, Staying in Touch: Using Physical Contact as a Control and Communication Modality, Proceedings of the Interagency Workshop on Smart Environments, Atlanta, Georgia, July 1999.
21. Peyman Oreizy, Micheal M. Gorlick, et al., An Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems, 14(3), May/June 1999.
22. Michael M. Gorlick, Nanonetworks: Shrinking the Internet Down to Size, Proceedings of the Second International Conference on Integrated Micro/Nanotechnology for Space Applications, Pasadena, California, April 1999.
23. Michael M. Gorlick, Software for Nanosatellites: Report of the Software Panel, Proceedings of the Second International Conference on Integrated Micro/Nanotechnology for Space Applications, Pasadena, California, April 1999.
24. J. V. Osborn, E. Y. Robinson, D. Hinkley, N. Ho, Michael M. Gorlick, B. Carlson, D. Gilmore, J. Swenson, L. Gurevich, L. Berenberg, C. Chien, J. Yao, V. Panov, B. Schiffer, R. Au and N. Romanov, A

PicoSAT Network and MEMS Space Platform, Proceedings of the Second International Conference on Integrated Micro/Nanotechnology for Space Applications, Pasadena, California, April 1999.

25. Michael M. Gorlick, Electric Suspenders: A Fabric Power Bus and Data Network for Wearable Digital Devices, Proceedings of the International Symposium on Wearable Computers, San Francisco, California, October 1999.
26. Michael M. Gorlick, The Role of Satellite Services in Internet-Scale Event Notification, Workshop on Internet- Scale Event Notification, University of California Irvine, Irvine, California, July 1998.
27. Samuel D. Gasster, Andrew Bustillos, David Kunkee, Michael M. Gorlick and April Gillam, Development of a microwave radiative transfer simulation tool for the NPOESS program, The 1998 IEEE International Geoscience and Remote Sensing Symposium, IGARSS, Seattle, Washington, July 1998, pp. 686-688.
28. Michael M. Gorlick, Distributed Debugging on $5 a Day, Proceedings of the California Software Symposium, University of California Irvine, Irvine, California, November 1997.
29. Michael M. Gorlick, Weaves as an Interconnection Fabric for ASIMS and Nanosatellites, Proceedings of the International Conference on Integrated Micro/Nanotechnology for Space Applications, Houston, Texas, November 1995.
30. Michael M. Gorlick and Alex Quilici, Visual Programming-in-the-Large versus Visual Programming-in-the-Small, IEEE Symposium on Visual Languages, St. Louis, Missouri, October 1994.
31. Michael M. Gorlick and Alex Quilici, A Visual Platform for the Synthesis of Complex Systems, Proceedings of the Complex Systems Engineering Synthesis and Assessment Technology Workshop, Beltville, Maryland, July 1994.
32. Michael M. Gorlick. Cricket: A Domain-Based Message Bus for Tool Integration, Proceedings of the Second Irvine Software Symposium, University of California at Irvine, March 1992.
33. Michael M. Gorlick and Rami Razouk, Using Weaves for Software Construction and Analysis, Proceedings of the 13th International Conference on Software Engineering, Austin, Texas, May 1991.
34. Michael M. Gorlick, The Flight Recorder: An Architectural Aid for System Monitoring, Proceedings of the ACM/ ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, California, May 1991.
35. Michael M. Gorlick, Carl Kesselman, Daniel Marotta and D. Stott Parker, Mockingbird: a Logical Methodology for Testing, Journal of Logic Programming 8(2), 1990, pp. 95–119.
36. Michael M. Gorlick and Rami Razouk, A Model-Theoretic Semantics for Real-Time Interval Logic, Proceedings of the Berkeley Workshop on Temporal and Real-Time Specification, International Computer Science Institute, Berkeley, California, December 1990, pp. 47–61.
37. Michael M. Gorlick and Carl Kesselman, Gauge: A Workbench for the Performance Analysis of Logic Programs, Logic Programming: Proceedings of the Fifth International Conference, Seattle, Washington, August 1988.
38. Michael M. Gorlick and Carl Kesselman, Timing Prolog Programs Without Clocks, Proceedings of the Fourth IEEE Symposium on Logic Programming, San Francisco, California, September 1987.
39. Harvey Abramson, Mark Fox, Michael M. Gorlick, Vince Manis and John Peck, The Pica-B Computer: An Abstract Target Machine for a Single-User Environment, Proceedings of the ACM Annual Conference, December 1978, pp. 301–309.
40. Michael M. Gorlick, Vince Manis, Tom Rushworth, Peter Van Den Bosch and Ted Venema, Texture: A Document Processor, Proceedings of the CIPS/CSA Canadian Computer Conference, May 1976.

Technical Reports

1. Michael M. Gorlick, Justin R. Erenkrantz and Richard N. Taylor, The Infrastructure of a Computational Web, Technical Report UCI-ISR-10-3, Institute for Software Research, University of California,

Irvine, May 2010.
2. J. Erenkrantz, Michael M. Gorlick, Girish Suryanarayana, and Richard N. Taylor, Harmonizing Architectural Dissonance in REST-based Architectures, Technical Report UCI-ISR-06-18, Institute for Software Research, University of California, Irvine, December 2006.
3. Michael M. Gorlick, Streaming State Kinematics and Flow Engineering, Technical Report UCI-ISR-06-03, Institute for Software Research, University of California, Irvine, March 2006.
4. Greg A. Bolcer, Michael M. Gorlick, et al., Peer-to-Peer Architectures and the Magi Open-Source Infrastructure, Endeavors Technology, 2001.
5. David Notkin, Michael M. Gorlick and Mary Shaw, An Assessment of Software Engineering Body of Knowledge Efforts, A Report to ACM Council, May 2000.
6. Michael M. Gorlick, Workshop Report: Software for Nanosatellites, Proceedings of the International Conference on Integrated Micro/Nanotechnology for Space Applications, Houston, Texas, November 1995.
7. Michael M. Gorlick, Carl Kesselman and D. Stott Parker, Gauge: Its Philosophy and Design, TR-0086A (2920-05)-2, The Aerospace Corporation, El Segundo, California, May 22 1989.
8. Carl Kesselman, Michael M. Gorlick and Joseph Bannister, Integrated Evaluation of Parallel Systems, Aerospace Technical Report SD-TR-88-109, The Aerospace Corporation, El Segundo, California, 1988.
9. Michael M. Gorlick, Vince Manis, Tom Rushworth, Peter Van Den Bosch and Ted Venema, Texture User's Manual, Technical Manual 75-08, Department of Computer Science, University of British Columbia, December 1975.

Patents

1. Michael M. Gorlick, Brian M. Morrow and Arthur H. Muir, Systems and Methods for Indexing Data in a Network Environment, European Union Patent #WO03042871, May 22, 2003.
2. Michael M. Gorlick, Wearable Electronics Conductive Garment Strap and System, U.S. Patent #6,350,129, February 26, 2002.

ABSTRACT OF THE DISSERTATION

Computational State Transfer: An Architectural Style for Decentralized Systems

By

Michael Martin Gorlick

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2016

Professor Richard N. Taylor, Chair

A decentralized system is a distributed system that operates under multiple, distinct spheres of authority in which collaboration among the principals is characterized by mutual distrust. Now commonplace, decentralized systems appear in a number of disparate domains: commerce, logistics, medicine, software development, manufacturing, and financial trading to name but a few. These systems of systems face two overlapping demands: security and safety to protect against errors, omissions and threats; and ease of adaptation in response to attack, faults, regulatory requirements, or market demands.

We consider decentralized systems in the context of architectural style, a domain-specific set of rules or constraints that confer benefits to the system in question. COmputAtional State Transfer (COAST) is an architectural style for decentralized systems where mobile code (more specifically, the exchange of live computations) is the principal interaction among peers. COAST exchanges rely on communication by introduction, meaning that a peer x can communicate with a peer y only if peer x holds a Capability URL (CURL) for y. CURLs are cryptographic structures; they are tamper-proof and cannot be guessed or forged. Live computations received by peers via CURLs are evaluated in the context of execution sites, flexible sandboxes that confine the functional and communication capability of visiting computations.

These four fundamentals: communication by introduction, mobile code, execution sites and CURLs, are sufficient to protect against many common security threats including unwanted intrusion, resource theft, or gross abuse of capability. These same four concepts also account for a considerable degree of adaptation and flexibility. More broadly, the COAST architectural style embeds computation exchange in the object-capability model of security; both computation exchange and object-capability contribute in equal measure to security and adaptation.

To validate the twin claims of security and adaptation for COAST we constructed a COAST-compliant reference implementation comprising *Motile*, a language for mobile code exchange and execution, and *Island*, a peering infrastructure for decentralized systems. We performed four studies with the *Motile* and *Island* platform: two each directed at adaptation and security.

The first adaptation study analyzed the problem of cooperative live update of a simple server, a gold standard for adaptive systems. Modeling an individual server as an endless service loop reading and responding to service requests we constructed *Motile*/*Island* protocols for three forms of server update: live update in place, live update with hot backup in a single address space, and live remote update with hot backup in which the update is instantiated in a remote address space. This evaluation demonstrates that COAST is capable of fail-safe cooperative live update for individual services in a decentralized system.

The second adaptation study examined the problem of evolving web service APIs, in particular client-driven API evolution. Using a web bookmark service as a test case, we demonstrate a minimalist service API that is extended per-client by client-developed mobile code delivered provider-side. The client code examples illustrate that service API extension and adaptation can be shifted from a service provider to the service clients, thereby easing the burdens on a service provider and speeding the pace of service evolution. We conclude that COAST is well-suited for client-driven service extension and customization.

The first security study offers a proof, by case analysis, that COAST is authority-safe and that *Motile* is a capability-safe language, hence authority-safe. These results place COAST and *Motile*/*Island* squarely within the growing body of work on the security and safety of capability-safe languages and confirms that the idioms and patterns of object-capability security are available to COAST.

In the second security study we evaluated COAST with respect to architectural accountability and obtained outcomes that indicate COAST is well-suited for capability accounting, a system accounting practice in which capability is the basic unit of exchange within and among systems. Our results include a communication capability analysis of a model financial transaction system, mapping points of capability creation, exchange, and exercise to system actions, and a sample analysis of capability traces for verifying system behavior and detecting process and security faults. This preliminary experiment hints that capability accounting may be useful for debugging, behavioral analysis, or early warning of threatening security events.

INTRODUCTION

Copious network bandwidth, combined with almost ubiquitous computation, permits decentralized systems — those that span multiple distinct spheres of authority — at unprecedented low cost and large scale. However, decentralization exacts a price. As decentralized services grow in scale and complexity, security becomes ever more challenging, suggesting that future systems cannot be designed and constructed as they have in the past. To this end I consider a counterintuitive notion: mobile code can improve the adaptation and security of decentralized systems.

The Problem

A *decentralized system* is a distributed system that operates under multiple, distinct spheres of authority in which collaboration among the principals is characterized by mutual distrust. Decentralized systems appear in numerous domains, including disaster response, coalition military command, commerce, finance, education, and research. These systems of systems face two overlapping demands: security and safety to protect against errors, omissions and threats; and ease of adaptation in response to attack, faults, regulatory requirements, or market demands. By their very nature they offer unique and conflicting challenges.

Many of the underlying subsystems of decentralized systems are *platforms* that offer service interfaces by which other (sub)systems can invoke specific service functions. With the proliferation of platforms it is possible for a small team of developers to quickly assemble a decentralized application that, in the past, a single, isolated but resource-rich agency would be hard-pressed to complete and deploy. But such decentralization exacts a price—a service-consuming agency exposes itself to risk as services come and go, as the performance of decentralized resources fluctuates, or as service interfaces evolve.

Further, absent a single overarching authority, and since the operating environment of each agency may vary with respect to market and customer demands, regulatory requirements, finances, and other factors, not to mention organizational agility and rate of change, a set of interdependent organizations may not synchronize or even roughly coordinate the evolution and deployment of their services. Thus service adaptability is a prized goal for both service providers and service consumers. Consumers must adapt to changes to external service interfaces and semantics while maintaining continuity of both their internal services and the outward-facing services they may offer in turn to their downstream clients. From

this perspective a vibrant service economy is simultaneously a powerful enabler and a source of risk and uncertainty.

Reliance on another requires trust, but trust is rarely unconditional. The role of an escrow agent in a real-estate transaction is a common example of a trusted intermediary employed by two or more mutually suspicious parties. Military coalition partners may severely limit the scope and specific content of information they share, as do suppliers and customers in a business relationship. Elaborate social, legal, financial, commercial, and technical structures have evolved over time to standardize, enforce, execute, and guarantee trust relations [210]. Both decentralized services and applications, like their precursors, have similar requirements.

Consequently, creating a well-engineered decentralized service poses multiple challenges: responsibly exercising local authority while participating as a "good citizen"' in larger applications; managing security, safety, and privacy; and meeting the customary demands of good software engineering. These issues become even more difficult when the services must evolve over time to reflect varying circumstances: clients demand modifications to existing services, another party engages in behavior that raises trust questions, or a new party with unanticipated demands is added to the mix, to name but a few.

As decentralized services grow in scale and complexity, security becomes a predominant concern. Attackers enjoy massive asymmetric technical advantages, including the large and vulnerable attack surfaces of many critical systems; patchwork security; enormous and complex legacy systems ill-suited for comprehensive protection and defense; low-cost development tools for malware; and a resource-rich underground economy of exploits, attack tools, and botnets. While the size, complexity, and costs of defensive security mechanisms increase rapidly, the size, complexity, and costs of malware attacks remain both small and constant [140], suggesting that future systems must be designed and constructed in ways dramatically different from present systems.

Here is the problem in a nutshell. We've long since arrived at the point at which services are decentralized; rarely do web server and web browser share the same sphere of authority. Once it became common practice for web servers to dynamically inject JavaScript for browser-side execution into the pages they delivered to browsers, the security risks increased dramatically. Mashups accelerated these trends, while layering web servers over existing legacy services (a natural and sound separation of presentation logic from application logic) was the final blow to the integrity of the classic, firewalled security perimeter.

Simultaneously, rapidly adapting enterprise services became an express enterprise strategy, leaving us with the confluence of two irresistible, destabilizing forces — decentralization on one hand and accelerated ongoing change in the architectures and construction of services on the other — while security and safety suffered greatly as a consequence. How do we embrace both decentralization and adaptation while maintaining a high level of security and safety?

Trends, Influences, and Insights

In this thesis I address the pressing problems of adaptive and secure decentralized services from the perspective of architectural styles [163]. Valued system attributes — modularity, performance, encapsulation, scaling, and adaptivity, to name but a few — arise from software architecture [238]. Security is cross-cutting, intruding upon many levels simultaneously. From an architectural perspective this is the natural consequence of the necessary and repeated application of a system or architectural constraint at multiple levels: an architectural order and sensibility directed to the concerns of security. Here I define and elaborate on a particular set of system principles that encompass security at many levels and in many ways.

My approach relies on well-known and established security principles and technologies, but combines or applies them in novel ways to achieve higher-order security consistent with the subject architecture. Object-capability security, in the seminal work of Miller [170], provides the scaffolding by which an architectural style can be repeatedly applied, at multiple system levels, to ensure system security. The COAST style and the mechanisms that it introduces are fractal; the mechanisms remain the same but may be repeatedly applied at ever finer levels of detail.

First and foremost, COAST relies on computation exchange: the exchange of live computations *among* computations. The exchange of live computations can be used to implement service requests and their responses, protocols based on active messages, asset[6] exploitation, search, discovery, analysis, and reflection. By combining computation exchange with the twin mechanisms of communication confinement and functional confinement we can introduce: *highly differentiated services*, *fractalware*, *continuation transfer*, *service innovation*, and *service customization*.

Communication confinement is the mechanism and practice of modulating communications among

[6] Databases, sensors, actuators, instruments and the like.

system computations. COAST cleaves communication from computation; that is, communication is an explicit capability and is always governed by contract. In fact, for a computation x there is no guarantee that it will be granted the capability to communicate or, if granted, that it will ever satisfy the contract that accompanies the capability. COAST reduces communication among all system elements to a single mechanism — unidirectional, asynchronous, immutable messaging — and defines the capability URL (CURL), a cryptographic structure, as the only means by which one computation may communicate (interact) with another.

Every message sent by a computation x to a computation y must be accompanied by a CURL u for y. A capability URL both contains and references contracts (predicates) that confine the capability to communicate. Contracts include, but are not limited to: when a computation may communicate (including time locks and expiration dates), rate limits on the frequency of communication or bandwidth consumption, bounds on the total number of messages, the origin of the communication, the contents of a message, the workload of the recipient, arbitrary predicates on the state of the recipient, and boolean combinators. Each individual communication from x to y may be governed by a separate and distinct CURL, each CURL with its own contracts. CURLs are revocable and can be non-delegable. Just because computation x communicated with computation y in the past does not guarantee that it will be allowed to communicate with y in the future.

Functional confinement dictates what a computation may do. All computations x execute within the confines of an x-specific execution site, a sandbox that confines both resource capability (the fungible resources granted to x such as processor cycles or memory) and functional capability, the functions that x may execute within the execution site. The functional capability of an execution site can be customized in numerous ways: limiting the arguments that a computation may use when calling a function, the specific behavior of a function, when a function can be called, the rate at which a function can be called, bounds on the total number of times a function can be called, or invocations governed by arbitrary predicates. A specific function f may be expunged from an execution site altogether; for example, unless the communication primitive `send` is present in the execution site of x, computation x can never send a message to computation y even if x holds a CURL for y.

The combination of communication and functional confinement eliminates *ambient authority*, capability that a computation acquires "out of thin air" from an implicit global context. Under COAST all

capability is explicit and its origin can always be traced back to an explicit grant of capability, the transfer of capability from one computation to another. This property, *capability safety*, is a critical security property and is instrumental in ensuring safety and adaptability. Since COAST is architecturally recursive by construction the same architectural and security principles apply at all levels of service (computation) instantiation; in particular each and every COAST computation is capability-safe. Embedding computation exchange in the model of object-capability security directly addresses several security concerns. To the extent that computation exchange can be confined by object-capability then the risks of computation exchange (where confinement is a proxy for acceptable risk), are modulated and limited by object-capability.

Communication and functional confinement lead to *highly differentiated services* in which the same service provider (computation) can offer a distinct service facet to each of its service clients (themselves computations). The individual service facets are distinguished one from the other by the CURLs that the provider generates and distributes to service clients. Because CURLs can carry arbitrary metadata, including snapshots of live computations (mobile code), the provider that issues the CURL u can include parameters that differentiate the service for a particular client. Alternatively, the provider can embed a live computation of the customized service itself in the metadata, creating a "fire and forget" CURL u in which the provider need not retain the implementation of the service facet. To request the u-specific service the client dispatches its request via CURL u and the provider can simply execute the implementation that it included in the metadata of u. In general, CURL metadata can contain arbitrary values, structures, and snapshots of live computations for the sake of one or more service facets. Here object-capability and computation exchange contribute in equal measure to both security and adaptation.

Under COAST every computation is both service provider and service consumer simultaneously — a service peer, in other words. *Fractalware* is an architecture of self-similar, recursive COAST service peers organized as a forest of trees that expand and contract in response to the live computations that they receive from other service peers [110]. Because every CREST computation is a service provider in miniature the fluidity of service within a fractalware infrastructure belies the traditional notion of middleware altogether. The notion of a layer in a peering collective is computation-centric and its fractal structure suggests that "middleware service" is wherever one happens to find it: layers are a matter of convenience, perspective, and scope, not static construction and restriction.

Continuation transfer allows a computation x to snapshot itself at a safe point in its execution and migrate the continuation elsewhere, where it resumes execution at the point the snapshot was taken. Continuation transfer can be used to implement service migration, service replication, and service branching; the latter when the service, post-replication, is commanded to alter its service profile or undertakes executing a new stream of service requests. This technique can be used in collaborative service development, service testing, and service exploration.

Service branching is closely related to *live update* where an executing service (computation) is modified, without service interruptions, as it executes. A variant of continuation transfer can be used to implement live update with hot backup.[7] In this scenario computation x performs a form of continuation capture, suspends execution at the point of the snapshot, and awaits confirmation that the live update was successful — the continuation is migrated elsewhere and restarted, but with code updates in place. If the live update fails for any reason then computation x resumes execution exactly where it left off without loss of service. If the live update is successful then computation x performs whatever housekeeping is necessary for a safe shutdown and halts cleanly.

Computation exchange transforms the relationship between service provider and service consumer by pushing *service innovation* and *service customization* out to service clients. If a service client can dispatch a live computation to a service provider for execution in a service-specific execution site then the client is free, within the capability strictures of the execution site, to craft the service offerings to suit its own needs. This may simplify the design, implementation, and deployment of services provider-side; the provider need provide only a handful of essential service primitives and stand back as clients shape those primitives into customized services.

Service innovation may become crowdsourced, as service providers, observing how clients harness the service primitives, add additional service primitives or expand their semantics. Fewer service primitives may also reduce the lifecycle costs of service offerings; consequently, providers can introduce additional services more frequently or offer a family of variations that reflect client-driven customization. There are other advantages. Clients need not wait for providers to deploy bug fixes if a flaw can be remedied by a workaround in a client-dispatched computation. Clients may also host higher-order services atop the provider service primitives, erecting an ecology of client-formulated service platforms

[7] See Chapter 8 for a full discussion of live update.

that feeds a virtuous cycle of service experimentation and innovation. Computation exchange has the potential to reshape service platforms by reducing barriers to entry and expanding the scope of service design, development, and deployment.

COAST: Introduction, Import, and Evaluation

To introduce COAST I put forward a notional decentralized system and suggest how mistrust may arise among its principals. Using this example as an exemplar I sketch COAST, the benefits that it confers upon decentralized systems, and briefly discuss the primary COAST mechanisms. I close with a discussion of each of the four evaluations and briefly describe a video streaming service constructed with *Motile*/*Island*, the reference implementation of COAST.

A decentralized system G is a distributed system, but with two distinctive characteristics: first it operates under multiple, distinct, independent spheres of authority and second, its principals collaborate from a posture of mutual distrust.[8]

The first means that not all subsystems of G answer to the same authority; for example, half of system G may be owned and operated by Alice and the other half owned and operated by Bob. Alice and Bob are collaborating to build and supply widgets to a third party, Carol — but each is contributing proprietary design and technology to the fabrication of the widgets.

The second means that Alice must grant Bob access to computational elements of her automated widget design system and Bob must grant Alice limited access to his state-of-the-art widget performance models, but neither trusts the other. Both must ensure that their counterpart has access to only the services (computations) that are required for their half of the joint project and nothing more. How then do Alice and Bob cooperate with one another while safeguarding their proprietary services from unwanted exploitation by their partner?

To answer such questions I turn to architectural style, a set of rules or constraints that confer specific benefits on those systems whose architecture conforms to the rules and constraints. Architecture can induce valuable system attributes such as security, adaptation, scaling, and resilience. These attributes do not arise by accident. COmputAtional State Transfer (COAST) is an architectural style for decentralized systems, that induces decoupling, isolation, confinement, and stateless interaction to yield systems for

[8] Or equivalently, a posture of *limited trust*. For my purposes these two characterizations are equivalent.

which security and adaptation are both natural outcomes. The thesis, *adaptive and secure decentralized services are feasible under COAST*, has two attendant claims:

- COAST is sufficiently expressive to implement adaptive decentralized services.
- COAST is sufficiently secure to ensure that service providers can adequately protect themselves against erroneous or malicious visiting computations.

Four distinct but interrelated COAST mechanisms — mobile code, execution sites, communication by introduction, and capability URLs — act in concert to guard assets, forestall sabotage, and hinder interference within decentralized systems. These four mechanisms play dual roles, simultaneously affording both adaptation and security. In other words, under COAST adaptation and security need not be mutually exclusive; in fact, identical mechanisms can serve both.

Of particular note, the capability URL (CURL) in COAST ensures isolation among computations in COAST-compliant architectures. Computation x can communicate with computation y only if it has been introduced to y beforehand — that "introduction" is a CURL for y. Unless x holds a CURL for y it is impossible for x to directly send a message to y.[9] Further, a CURL only grants unidirectional communication capability. Even though x may send messages to y using a CURL for y, absent an introduction to x (that is, a CURL for x) computation y cannot directly send a message to x.

Communication by introduction maximizes isolation, ensuring that a computation in a decentralized system never communicates outside of a small circle of known, legitimate computations. Communication by introduction can guarantee that a computation never communicates at all or communicates with at most one other computation. Guarantees of this ilk are essential for preserving the integrity, and assuring the correctness, of decentralized systems.

To evaluate the efficacy of COAST a COAST-compliant reference implementation was developed and deployed. It comprises *Motile*, a programming language whose semantics are grounded in the transmission and evaluation of mobile code, and *Island*, a peering infrastructure for decentralized systems and for which the serialization, encryption, decryption, deserialization, and confinement of mobile code are first-order concerns.

The reference implementation was used in four studies to validate the claims: two each for adaptation

[9] Without a CURL for y computation x might, via a proxy z, pass a message to y but that is only because x holds a CURL for z and z in turn holds a CURL for y. In other words, at some point in the past, x was introduced to z and z was introduced to y.

and security.

The first adaptation study analyzed the problem of cooperative live update of a simple server, a gold standard for adaptive systems. Modeling an individual server as an endless service loop reading and responding to service requests I constructed *Motile*/*Island* protocols for three forms of server update: live update in place, live update with hot backup in a single address space, and live remote update with hot backup in which the update is instantiated in a remote address space. This evaluation demonstrated that COAST is capable of fail-safe cooperative live update for individual services in a decentralized system.

The second adaptation study examined the problem of evolving web service APIs, in particular client-driven API evolution. Using a web bookmark service as a test case, I demonstrate a minimalist service API that is extended per-client by client-developed mobile code delivered provider-side. The client code examples illustrate that service API extension and adaptation can be shifted from a service provider to the service clients, thereby easing the burdens on a service provider and speeding the pace of service evolution. I conclude that COAST is well-suited for client-driven service extension and customization.

The first security study offers a proof, by case analysis, that COAST is authority-safe and that *Motile* is a capability-safe language, hence authority-safe. These results place COAST and *Motile*/*Island* squarely within the growing body of work on the security and safety of capability-safe languages and confirms that the idioms and patterns of object-capability security are available to COAST. The two proofs rely heavily on the semantic consequences of the COAST style rules, the implementations of *Motile*/*Island*, and their conformance to the COAST style. Capability-safety is sufficient to implement security practices such as revocation and multi-level security, non-delegation, and information flow control.

In the second security study I evaluated COAST with respect to architectural accountability and obtained outcomes that indicate COAST is well-suited for capability accounting, a system accounting practice in which capability is the basic unit of exchange within and among systems. The results include a review of a model financial transaction system implemented in the COAST style under *Motile*/*Island*, a mapping of critical control points to capability actions, and a sample analysis of capability traces for validating system behavior and detecting process and security faults.

Using *Motile*/*Island* we constructed and demonstrated a small video streaming test bed, COASTCAST [113], that orchestrates high-definition video streams, from camera to display, as collaborating video ser-

vices. A single stream may transit many computations scattered across the network: the source (real-time HD video cameras), video encoders, midpoints (relays and publish/subscribe servers), video decoders, and eventual sinks (user interfaces and displays), each potentially managed by a separate sphere of authority. Streaming and manipulating live video is a demanding application domain; in COASTCAST it serves as a substitute for any high-bandwidth, soft real-time, data stream.

All of COASTCAST is fabricated from live computations dispatched to execution sites across the network that: read the raw video frames generated by cameras, distribute raw frames to intermediate consumers such as encoders, downsamplers and special effects transforms, relay compressed video frames (themselves live computations) to video decoders, and finally, display one or more live video streams. Communications among all computations are managed by CURLs.

COASTCAST exhibits interesting emergent properties, courtesy of the architectural style. For example, computation exchange allows any user to share a stream with any other user on demand. Video streams are shared by directing a local decoder d_1 to spawn a copy d_2 of itself to an execution site elsewhere in the network, where the new decoder d_2 automatically resubscribes to the relay and publication service r used by d_1. The CURL necessary to subscribe to r is present in the run-time state of computation d_1 and, when d_2 is spawned, that CURL travels along with the mobile code.

Contributions

To address the intertwined challenges of security, constant and perhaps unpredictable change, the fluidity of service interfaces, and an ever-expanding attack surface I undertook to resolve the long-standing tension between system adaptation and system security. COmputationAl State Transfer (COAST), an architectural style for decentralized systems, is the centerpiece of the work. There are two principal contributions: the COAST architectural style and its embodiment in a reference implementation *Motile*/*Island*.

The COAST Architectural Style

The COAST style rules define the instantiation of services, their execution, inter-service messaging mediated by capability URLs, and the interpretation of messages. From a concise statement of the style I explore its architectural consequences and draw a detailed system architecture comprising intercommunicating address spaces where both intra- and inter-address space interactions are compliant with the

style. In all cases, whether the interaction is intra-authority or inter-authority, the same style rules apply. Consequently, COAST-compliant systems enjoy defense in depth, as each and every interaction, whether local or remote, is mediated by a small set of consistent, but flexible, security mechanisms. Developers can adjust these mechanisms to accommodate a broad variety of security policies and varying degrees of risk.

Four distinct but interrelated COAST mechanisms: *communication by introduction*, *mobile code*, *execution sites*, and *capability URLs* combine to guard vital assets, forestall sabotage, and hinder interference. Each of the four plays a dual role, affording both adaptation and security. Embedding computation exchange in an object-capability infrastructure achieves what neither can do alone — decentralized adaptation and security in equal measure, balanced one against the other.

Motile/Island

A reference implementation for COAST has been developed and tested. Deployed in a number of laboratory-scale experiments it comprises *Motile*, a single-assignment, mobile code language, and *Island*, a peering infrastructure for decentralized computation exchange. While alternate implementations are certainly possible, the style strongly influenced the language and peering infrastructure in numerous ways, large and small. The unit of authority in the peering implementation is an *island*, a single, homogeneous memory address space that is network-addressable.

Motile at its core is an assignment-free, abstract, lambda calculus engine confined to lambda expressions, function calls, conditional expressions, and function definitions. Closures and continuations are the reification of computations exchanged among islands. The lambda calculus engine contains no ambient authority and all functional capability for visiting closures and continuations is defined by the host island. Unlike many other mobile code implementations, *Motile* does not rely on a byte-coded virtual machine; the core of the network exchange representation is a graph-based, high-level, assembly language reflecting the abstract, lambda calculus engine.

Thesis Roadmap

Architecture can induce valuable system attributes such as security, adaptation, scaling, and resilience. These attributes do not arise by accident. An architectural style that induces decoupling, isola-

tion, and stateless interaction can yield systems for which security and adaptation are both natural outcomes. The COAST architectural style is deceptively concise. Situating it in the context of prior work is an important first step to understanding the constraints that the style levies on COAST-compliant implementations. Chapter 1 briefly presents an informal abstraction of the style rules as a preface for a statement of the thesis and its claims. The remainder of Chapter 1 is devoted to the origin and history of *computation exchange* as a modality of interaction for distributed and decentralized systems. Computation exchange underlies the precepts of COAST and was the impetus for the development and investigation of the style.

Any realization of COAST must implement some manner of mobile code. Chapter 2 relates the history of mobile code, which can be traced as far back as the early 1960s. Mobile code displays diverse semantics with respect to the mobility of language constructions, exhibits broad variations in the treatment of variable bindings, and is influenced by the choice of transport representation for its network transfer from one host to another. The same issues play out for *Motile*/*Island*, the reference implementation of COAST as well, and I illustrate how the choices made for *Motile*/island are often dictated by either the style or other considerations, including simplicity, ease of implementation, or the strictures of decentralization.

Chapter 3 presents the complete COAST style and is devoted to a detailed discussion of the consequences of the style rules. The style is deliberately underspecified to allow for a variety of implementations; the reference implementation offered here is just one of many possible implementations. However, the transport model embraced by the reference implementation is critical and some of the benefits of capability URLs can be traced back to the transport model chosen here. Many of the design decisions for *Motile*, the mobile code language of the reference implementation, reflect the interplay of the rules of the style.

Chapter 4 discusses the technical obligations of the COAST style rules for implementations that seek to be COAST-compliant with an emphasis on the role of binding environments and the necessity for late binding under COAST, the rationale for execution sites and expectations of message ordering among COAST services. The COAST style deliberately leaves two concepts undefined: spheres of authority and the grounds for the confinement of computation arguing that they are best left to service providers.

Chapter 5 examines *Motile*/*Island*, the reference COAST implementation, with respect to the obligations enumerated in Chapter 4. The obligations are reflected in the language binding semantics, message passing, and the execution of visiting mobile closures. To illustrate the generality of the message passing

primitives I define *decentralized promises*, a language feature that bridges the gap between functional programming and decentralized computation. The chapter concludes with an extended example that illustrates two forms of decentralized computation: *spawning* and *remote evaluation*.

Chapter 6 details the mobility semantics of *Motile*, emphasizing the security considerations. These considerations are reflected in the two implementations of messaging: one for transmission between computations in separate and distinct address spaces and the other for transmission between computations that share an address space. The binding semantics of *Motile* and messaging interact and the behavior differs between the two implementations. That difference can be used, without harm, to good advantage however, there are cases when messaging within a single shared address space can lead to unwanted transfers of functional capability. Here I show that serialization, required for inter-address space message transfers can be applied within a single address space to contain and prevent leaks of functional capability.

Chapter 7 is a concise reference manual for *Motile*, a single-assignment, Scheme-dialect that is based on R5RS Scheme, with some R7RS extensions. In addition, *Motile* supports persistent, functional hash maps and vectors; their API is detailed here as well.

Chapters 8 –11 describe four studies of the adaptation and security behavior of *Motile*/*Island*: two each directed at adaptation and security. The first pair of studies, Chapters 8 and 9, evaluate the claim of adaptation in two domains: cooperative live update and client-driven web API design.

The second pair of studies, Chapters 10 and 11, evaluate the claim of security from two distinct perspectives. Chapter 10 proves that COAST is authority-safe and that *Motile*/*Island* is capability-safe. Chapter 11 evaluates COAST with respect to architectural accountability, an aspect of system-level security.

Finally, Chapter 12 summarizes the work as a whole while Chapter 13 considers a few unanswered questions and offers suggestions for future work.

Chapter 1: Computation Exchange — From Idiom to Style

COmputAtional State Transfer (COAST) shifts decentralized systems away from the pervasive practice of content and data exchange that dominates client/server interactions to a practice of *computation exchange* where peers routinely exchange live computations rather than static content. My motivation was to transition web-scale systems to a new set of consistent constructive principles that encouraged adaptive emergent behavior [70, 110]. With the goal of computation exchange in mind, the COAST architectural style can be informally summarized as:[1]

- All services are computations whose only interaction is the asynchronous messaging of *mobile code*.
- All computations execute within the confines of an *execution site*, a flexible sandbox that confines functional and resource capability.
- A computation x can transmit a message to a computation y only if x holds a *capability URL* (CURL) for y, *communication by introduction*. CURLs are cryptographic structures; they cannot be forged and are tamper-proof. Only y can create a CURL for y.
- The interpretation of a message delivered to a computation is not only receiver-dependent but also delivery-dependent.

These four distinct but interrelated COAST mechanisms — mobile code, execution sites, communication by introduction, and capability URLs — act in concert to guard assets, forestall sabotage, and hinder interference within decentralized systems. These four mechanisms play dual roles, simultaneously affording both adaptation and security. In other words, adaptation and security need not be mutually exclusive, in fact, identical mechanisms can serve both.

This chapter traces the evolution of computation exchange from an idiom to a well-defined architectural style. I first specify, in Section 1.1, the thesis of this work and the twin attendant claims of security and adaptation as linked emergent behaviors in COAST-based systems, to set the context for the arguments that follow. Sections 1.2 and 1.3 consider how software architecture can induce security and adaptation,

[1] A formal statement of the COAST architectural style is reserved for Chapter 3.

where I argue that mobile code is a powerful and compelling tool for adaptation. In other words, it is plausible that an architectural style can induce security *and* exploit mobile code as an instrument of adaptation.

But mobile code introduces considerable risk and the question, discussed in Section 1.4, becomes how to confine and modulate that risk. A close examination of the literature, detailed in Section 1.5, reveals ample precedent for computation exchange. I close in Section 1.6 with an examination of security for computation exchange based on capability security. Capability security adequately confines mobile code and secures decentralized systems overall. But mobile code is a touchstone for a high degree of adaptation. In searching for an answer to the problem of security for computation exchange I happened upon a set of mechanisms that served two masters simultaneously: security *and* adaptation.

1.1 Thesis and Claims

Like CREST, COAST is an architectural style consistent with the paradigm of computation exchange; COAST, however, treats security as a primary concern. My thesis, *adaptive and secure decentralized services are feasible under COAST*, rests upon two claims:

- COAST is sufficiently *expressive* to implement adaptive decentralized services
- COAST is sufficiently *secure* to ensure that service providers can adequately protect themselves against erroneous or malicious visiting computations

Here I define *adaptive* as "able to alter to conform to new or altered circumstances," *expressive* as "serving to express or represent," and *secure* as "comparatively free from danger; not exposed to serious damage or attack." The evaluation of the claims, Chapters 8–11, rests upon a few critical definitions: *adequate* meaning "meeting an acceptable standard of fitness," and *sufficient* meaning "equal to what is required".[2]

The concerns are twofold: *peer safety and integrity*, thwarting erroneous or malicious actions undertaken via computation exchange against a peer, and *noninterference*, preventing one computation executing on a peer from accidentally or deliberately interfering with the execution of a co-resident computation. The vulnerabilities include:

- *Fungible Resources*. Excessive consumption or exhaustion of fungible resources such as processor cycles, memory, network bandwidth, and permanent or scratch storage.

[2] All definitions taken from [258].

- *Fixed Assets*. Unauthorized access to, and exploitation of, fixed peer assets such as sensors (for instance, a camera), actuators (valves in an industrial process), specialized physical processors (say, a GPU), confidential storage devices, or sensitive information sources (a database or file).[3]
- *Sabotage*. Disruption of critical peer-specific computations that manage the fungible or fixed assets of a peer.
- *Interference*. Unwanted interactions, explicit or otherwise, among computations residing on the same peer, for example, an untrusted computation needlessly communicating with a sensitive, trusted computation.

Many solutions are available to restrict or eliminate the excessive consumption of *fungible resources*; consequently this vulnerability is not addressed here. Processor cycles can be allocated by combinations of thread priority scheduling, explicit thread lifespans, load caps, and load balancing. Accurate memory accounting without partitions is feasible in garbage-collected languages [262] and can prevent memory exhaustion. User-level file systems such as FUSE [48, 100] implement tailored file systems that are access- and resource-controlled as an unprivileged user process/daemon. Network bandwidth throttling and traffic shaping are well-known strategies for protecting network resources. Sandboxing is also a rich, active research topic whose results are applicable [6, 94, 144, 212]. The other three vulnerabilities — *fixed assets*, *sabotage*, and *interference* — are each principal concerns of my thesis.

COAST springs from three insights:

- Architecture can induce security. From an architectural perspective, it is far better that the essential mechanisms of security arise from the constraints of the architecture.
- Architecture can induce adaptation. Requiring service consumers to specify exactly which services they want simplifies many interactions between the client and the service provider.
- Capability confines risk. By defining and confining what mobile code can do (functional capability) and where, when, and with whom mobile code can communicate (communication capability) risk can be modulated and reduced to an acceptable level.

[3] Leaking confidential or sensitive information is not a concern here nor is distributed information flow control. These and related topics are out of scope.

1.2 Architecture Can Induce Security

The *Principle of Least Authority* (POLA) [19, 208] dictates that every computation and service should operate with the least capability necessary to the task where a *capability* is an unforgeable reference whose possession confers both authority and rights to a principal [58]. If security is a direct consequence of the capability afforded to principals, then as the capability available to principals grows so do the risks. In other words, it is at best unwise, and at worst catastrophic, for a service to offer great power to an erroneous, irresponsible, or hostile client. In this work I express POLA in the language of capability security [170].

POLA is but one of several related principles for secure system design:

- *Start from zero.* The starting point for any service is *zero capability*, as omission is always fail-safe. If a necessary capability is missing then an action may fail, but that failure, in all likelihood, will be quickly detected. The opposite case, the exercise of a superfluous capability, may go undetected for long periods of time. Capability should be granted to the service only if strictly necessary.
- *Divide capability.* Dividing capability among several distinct services may significantly improve security and safety by simplifying individual service interfaces but at the expense of increasing the number of interactions among those services. This design precept is closely related to the "fail-fast" philosophy of Erlang [7] and sees application in the design and implementation of "crash-only" software [29]. Further, fine-grain, highly-specific capability leads naturally to fine-grain, highly-specific permissions. The decomposition and selective recombination of capability can be a powerful security tool [170].
- *Exclude ambient capability.* All capability must be explicitly granted. It must be possible to trace a grant of capability back to its source either by analysis or by monitoring. Static analysis of the allocation of capability demonstrates that the software is safe by design while runtime monitoring ensures that it's behavior is safe. Rejecting ambient capability is a corollary of the principle of the explicit allocation of capability.
- *Embrace firewalls.* POLA tends to minimize potential interactions among computations, thereby reducing the odds of unintentional, unwanted, or unwarranted exploitation of capability (the Confused Deputy [122] is a classic example). If an architecture permits firewalls (barriers to the transfer of capability) then POLA provides rationale for their emplacement and design.

- *Defend in depth.* Where possible, deploy multiple, overlapping defensive mechanisms to defend against vulnerabilities. For example, live tracing and auditing of the exercise and transfer of capability may provide early warning of behavior that could lead to a breach in system security. POLA, which requires the explicit identification and characterization of capability, lends itself to a variety of in-band and out-of-band mechanisms that can reduce the severity of breaches or mitigate their consequences.

Security arising from architectural style has ample precedent in the work on capability-based operating systems KeyKOS [24] and EROS [213], modern microkernels such as seL4 [146] and Singularity [130], and capability languages such as E [170] and Caja [171]. At many levels an architecture:

- Enumerates the origins, kinds, and consequences of the available capability.
- Offers a rationale and mechanisms for minimizing capability.
- Dictates significant interactions, including both the exercise of capability and its migration.

In this way an architectural style makes plain the mechanisms of creating and transmitting capability. Even better, if those mechanisms are made integral to the architectural style, and enforced by design and analysis tools [53], developers may be prevented from committing common security errors (just as type systems in programming languages forestall a common class of errors). Lacking such, developers often fall back upon ad hoc, ill-fitting substitutes subject to subtle, but dangerous, security flaws [16]. From an architectural perspective, it is far better that the essential mechanisms of security be built in—impossible for an application developer to circumvent—with the architectural style itself encouraging development of additional, higher-level or domain-specific security measures. In other words, an architectural style for services should, by its construction and constraints, encourage the development and deployment of security services, themselves services.

1.3 Architecture Can Induce Adaptation

Message bus architectures, exemplified by Field [202] and C2 [237], introduced message-passing and message buses as mechanisms for software adaptation. Early work by Gorlick and Razouk [112] demonstrated Weaves, an architectural style grounded in data flow, for dynamic adaptation. Later work by Oreizy and colleagues [182, 183, 239] laid the foundations for architecture as the centerpiece of run-time adaptation. However, the broadly accepted architectures for SOAs, be they based on remote procedure call (RPC),

remote method invocation (RMI) or REpresentational State Transfer (REST), offer scant architectural support for dynamic adaptation when viewed through the lenses of BASE [239] or a connector-centric taxonomy ([238], Chapter 5).

1.3.1 In Search of an Adaptive Architecture

For an answer I turn to a different architectural paradigm where service consumers program service providers to obtain the services they desire: *remote evaluation*, a contribution of Stamos [227, 228], in which clients dispatch mobile code [99] programs to servers for server-side execution. Remote evaluation introduces a division of labor between service providers and service consumers where providers create fine-grained and sufficiently general interfaces for their services and assets, and consumers orchestrate those fine-grained interfaces with mobile code to produce exactly the services they require. Remote evaluation:[4]

- Amortizes the cost of network transmissions by reducing the frequency and volume of network communications in comparison to other service architectures; an important consideration for mobile devices with limited power reserves.[5]
- Encourages a broad spectrum of functional partitions between client and server as the client can vary the operations executed server-side versus the operations executed client-side subject only to the functions exposed by the service. For example, a client may reserve some operations for itself in order to preserve confidentiality. Alternatively, a client executing on a low-power processor may conserve power and improve performance by offloading a larger percentage of the computation service-side.
- Executes many service functions in a single request, allowing clients to distribute and parallelize computations in a client- and application-specific manner, while other techniques execute just one service function per request.
- Generalizes remote procedure call, remote method invocation, and request/response. The remote evaluation of a single function call is semantically equivalent to RPC/RMI and can easily emulate the request/response pattern common to client/server architectures.

[4] Adapted from [228], pages 21–23.

[5] On average I expect the frequency and volume of mobile code transfers to be less than than the comparable frequency and volume of network transfers required for other service architectures.

- Offers a broad degree of service customization as client requests can contain embedded utility functions to be invoked service-side by the client-defined program.
- Admits of higher-order services since remote evaluation permits closures as arguments to service functions, a powerful technique long appreciated by functional programmers for dynamic customization.
- Frees the service provider of the burden of defining a generic "one size fits all" service interface. Instead the provider defines a fine-grained interface whose primitive operations clients can compose to suit their needs.
- Allows the service provider to define a differentiated service — a variety of interfaces to the same service, some more general or capable than others.
- Fits loosely-coupled distributed and decentralized computation, particularly when shared data is immutable.

1.3.2 The Drawbacks of Remote Evaluation

However, despite its appeal, remote evaluation (REV) is not without drawbacks. In his thesis (see [228], Section 7.4, pages 124–125) Stamos points out that REV incurs both compile-time and run-time overhead above and beyond that of RPC, where the total cost of the more general mechanisms of mobile code may be higher than the execution of a comparable sequence of remote procedure calls. REV exacerbates the challenges presented by RPC such as protection, adequate service resources, and load balancing — inter-request isolation is far easier for RPC than it is for REV (a reasonable claim) and, as service routines for RPC calls are implemented by trusted developers, the threats of non-terminating RPC requests, excessive resource consumption, or attack are much diminished (a claim disproved by modern experience). Last, but certainly not least, REV service developers must be wary of each REV request. It may execute for far longer than is customary (or may never terminate at all) and mobile code bugs or malicious clients can interfere with the remote evaluations of other clients. Protection mechanisms common to operating systems such as "*separate address spaces, authorization checks, resource accounting, and preemptive scheduling*" may prove necessary.

1.3.3 Is Remote Evaluation a Viable Alternative?

There is another drawback that Stamos failed to mention. REV pushes service development onto the service consumer and consumers may be unwilling to shoulder or incapable of meeting that obligation. The utility of REV is predicated in part on the assumption of a cooperative relationship between provider and consumer: service providers offer APIs whose individual functions can be combined in useful ways and service consumers invest the time and effort to assemble domain- and consumer-specific mobile code programs targeted to the provider-defined APIs.

The relationship is symbiotic; without useful service APIs consumers will not construct mobile code programs and without mobile code programs REV-based APIs are useless frippery. Is there any evidence that service providers are willing to undertake the effort to construct domain- and service-specific APIs? In other words, can we foresee an *API economy* where enterprises, governments, non-profits, communities of interest, and individuals implement APIs for direct programmable access to their resources, systems, and processes?

The market says *yes* and over the past decade three distinct classes of service APIs, public, partner and private, have emerged. A *public* API exposes the data, information, and functions of a sphere of authority to outside third parties that do not necessarily have a formal relationship (business, economic, legal, or organizational) with the authority. For a provider, the advantages of a public API include delegated research and development, increased traffic, reach beyond customary borders, and additional revenue streams. A *partner* API facilitates communication and integration between a sphere of authority and its partners, can include value-added services, act as a gateway to premium services, and is often regarded as a precondition for partnership. A *private* API is used with the confines of a sphere of authority to rationalize infrastructure, reduce costs, increase flexibility, or improve internal operations.

1.3.4 Public and Partner APIs

Closer inspection of public and partner APIs reveals a thriving ecosystem. As of November 2015 the Programmable Web[6] had cataloged approximately 14,300 distinct public and partner APIs for outward-facing web services.[7] The explosion of public and partner APIs (RESTful and otherwise) over the past

[6] `www.programmableweb.com`

[7] The Programmable Web catalog is no doubt incomplete; the actual number of public and partner APIs is likely far higher.

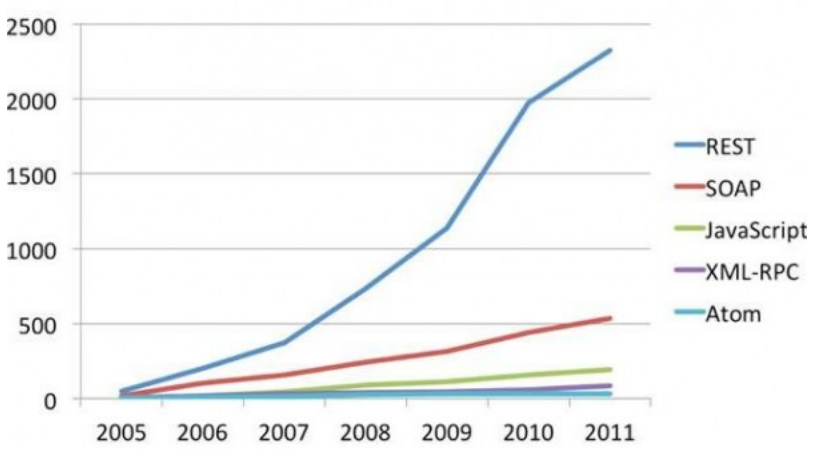

Figure 1.1: Growth in public web APIs by protocol from mid-2005 to mid-2011 (taken from [151]).

decade, illustrated by Figures 1.1 and 1.2, is evidence that service providers are motivated to construct *platforms*: services complemented by a programmable API.

The roles of public and partner APIs, covers a broad spectrum including, but not limited to, mobile backends, customer and partner ecosystems, content and information distribution, transaction distribution, the API as a business, collaboration, and cross-API integration [264]. Examples include:

- *Kinvey*[8] offers APIs for *mobile backend as a service* to ease the development and deployment of applications for mobile devices. and Its service APIs include data storage and access, dentity, engagement, business logic, analytics and compliance reporting, deployment, scaling, and device customization.
- The *Dropbox*[9] Core API supports a *customer ecosystem* that encourages customers to develop tailored storage and synchronization solutions and integrate Dropbox services into customer-specific mobile applications.
- *Amazon Web Services*[10] underpins a broad *partner ecosystem*. A partner ecosystem layers distinct and varied partner-specific server offerings over the core platform that extends the reach and functionality of a core platform.
- The API clients of a content or information distribution platform are partners with the platform for content syndication and delivery. The Guardian Media Group offers *N0tice*[11], a *content distribution*

[8] www.kinvey.com
[9] www.dropbox.com
[10] https://aws.amazon.com/
[11] n0tice.com

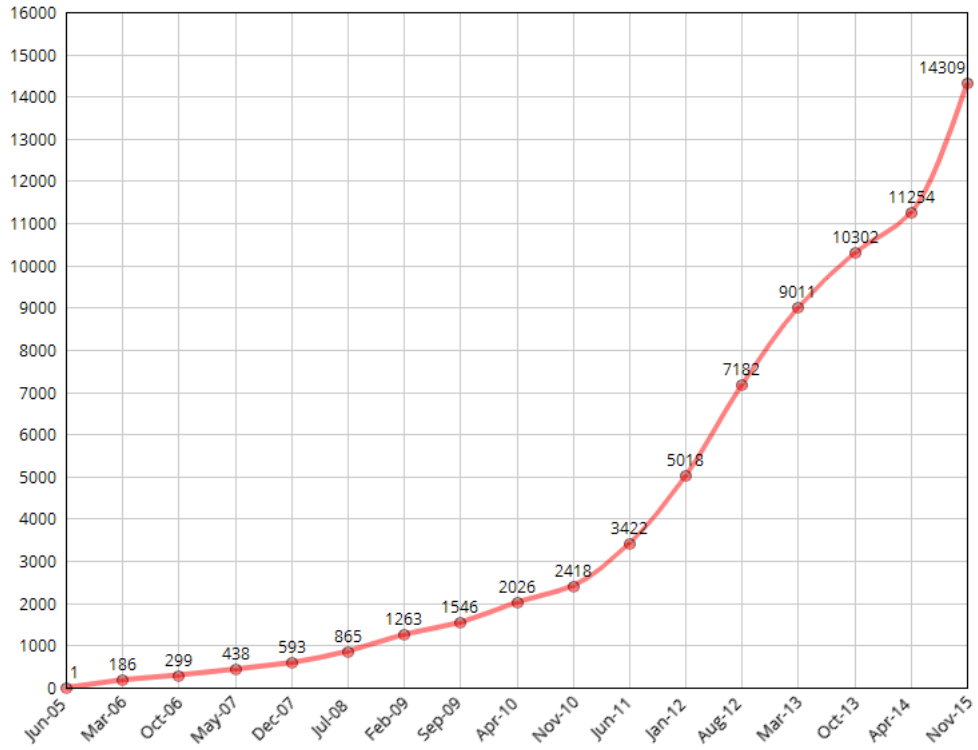

Figure 1.2: Growth in public web APIs from mid-2005 to late 2015. The data for Jun-05 through Oct-13 is taken from [265], slide 2. The data for Apr-14 is taken from [264], slide 9. The data for Nov 15 was taken from the Programmable Web `www.programmableweb.com` in mid-November, 2015. The first period, from Jun-05 to Mar-06, covers 8 months, the penultimate period of Oct-13 to Apr-14 covers 6 months, and the last period, from Apr-14 to Nov-15, covers 19 months (and consequently, the rate of API growth from Apr-14 to Nov-15 is in fact far less steep than it appears here). All other periods, from Mar-6 to Oct-13, are 7 months long.

API for sponsored citizen journalism that includes services for authentication, search, noticeboards, and content management. *Factual*[12] provides a location-centric *information distribution* API and operates as a trading desk partner for location-based mobile advertising. *Xignite*[13], an *information diistribution* platform. supplies cloud-based, financial market data APIs that deliver real-time and reference market data to clients and applications.[14]

- *Transaction distribution* deploys APIs for transactions on the core business model of the provider (think Amazon, eBay, Expedia, or Target). Here the API drives additional transactions and expands the business.
- For some enterprises the API itself *is* the business. *Sendgrid*[15] promotes an API for transactional, marketing, and triggered emails while *Twilio*[16] implements multi-device cloud-mediated communications accessed via a public API for initiating and receiving telephone calls, VOIP exchanges, and text messages. The *Weather Underground*[17] API is devoted to accurate and timely weather and forecast data and offers three tiers of service covering current conditions, hourly and multi-day forecasts, satellite and radar imagery, severe weather alerts, tides and currents, and travel planning.
- *Github*[18], a predominantly a *collaboration* API focused on the needs of software developers, includes: repository-centric actions, metadata, events, issues, statistics, and search. In contrast, the *Slack*[19] API provides team collaboration via messaging and event notification.
- The *Slack* API also encourages *cross-API integration* across services in which Slack users engage other services via text-based robots that respond to commands and issue notifications in a conversational manner [162].

1.3.5 Private APIs

The number of private APIs is unknown but we can safely assume that their number and variety exceeds that of the public and partner APIs by a wide margin. A private API, as an expression of *internal innovation and adaptation*, reflects the efforts of a sphere of authority to improve its agility and internal

[12] `www.factual.com`
[13] `www.xignite.com`
[14] For a perspective on Xignite, its fellow upstarts, and Bloomberg Financial, the 800-pound gorilla in the world of real-time financial data, see [244].
[15] `www.sendgrid.com`
[16] `www.twilio.com`
[17] `www.wunderground.com`
[18] `www.github.com`
[19] `www.slack.com`

efficiency. Best known, thanks to Stephen Yegge [269], is Amazon's campaign to rationalize its internal infrastructure with private APIs. Yegge relates that in the early 2000s Jeff Bezos, the founder, Chairman and CEO of Amazon.com, issued a mandate to his development teams:

> All teams will henceforth expose their data and functionality through service interfaces.
>
> Teams must communicate with each other through these interfaces.
>
> There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
>
> It doesn't matter what technology they use.
>
> All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

The mandate closed with: *Anyone who doesn't do this will be fired.*

Over the span of a few years Amazon transformed its internal operations into a services architecture and along the way learned several important lessons, two of which are relevant to the arguments against REV (again from [269]):

> ... every single one of your peer teams suddenly becomes a potential DOS attacker. Nobody can make any real forward progress until very serious quotas and throttling are put in place in every single service.
>
> ... Organizing into services taught teams not to trust each other in most of the same ways they're not supposed to trust external developers.

I draw several observations from the Amazon experience. First, service APIs are essential at scale irrespective of their position (public, partner, or private) with respect to the boundary of the sphere of authority. Second, every service API no matter the technology or protocol (RPC, RMI, REV, REST, and so on), presents significant security and performance risks to the service provider. In this respect the argument of Stamos that REV is more dangerous than RPC is wrong and while the risks may differ somewhat in degree or kind arguing that REV is inherently more unsafe or open to attack does not comport with modern experience

for large-scale services.[20] Finally, there is a place for remote evaluation across the API spectrum (public, partner, or private) provided that REV is at least as expressive as the other contenders (RPC, RMI, SOAP, and REST) and that the risks REV presents can be beaten down to a level comparable to, or better than, those presented by REST.

1.3.6 Revisiting the Drawbacks of Remote Evaluation

Stamos published his doctoral thesis [228] in January, 1986 more than five years before the first public announcement of Boerner-Lee's "World Wide Web" in August, 1991[21], almost 15 years before Fielding's exposition of REST [83], and 20 years before the rise of the programmable web. We, unlike Stamos, have the benefit of a large body of accumulated experience with distributed request/response protocols (HTTP 1.1 and RPC among others) whose degrees of demand and scale[22] would have been unthinkable in the mid-1980s.

From our vantage point the objections to REV can be laid, with equal vigor, against RESTful interactions [70]. For an HTTP server the fungible resources needed to satisfy HTTP requests can vary markedly from one request to another and the strict ordering of HTTP requests and their responses can introduce unwanted latency when a client issues multiple related requests over a short span of time. The ascent of mashups and its follow-on, the web as service infrastructure [70], brings us web clients (service consumers) that issue many requests in quick sequence against multiple elements of a RESTful API. The experience of Amazon demonstrates that even a private service API based on REST or RPC is subject to resource exhaustion, excessive load, accidental misuse, or erroneous invocations. In this light it is hard to argue that a sequence of RESTful request/response pairs consumes fewer resources, exhibits less latency or jitter, or is less subject to misuse and error than the comparable REV program.

Websockets [81], with its bidirectional asynchronous communications and out-of-order responses, addresses these shortcomings, and in this respect moves web clients and servers away from REST and closer to RPC semantics. Meanwhile, RPC has been repudiated by its most ardent supporters [252–254], SOAP has been largely abandoned by developers, and REST-based APIs now dominate the programmable

[20] Frankly, anyone who thinks that a RESTful API guarantees safety and security is gravely misinformed [230].

[21] See `https://groups.google.com/forum/#!msg/alt.hypertext/eCTkkOoWTAY/urNMgHnS2gYJ` for the message that Tim Boerner-Lee posted to `alt.hypertext`.

[22] Xignite, a provider of real-time, financial market data, served more than 50×10^9 API requests in July 2015. *Xignite Records 50 Billion Financial Data API Calls per Month*, Press Release, November 3, 2015, `http://finovate.com/xignite-surpasses-50-billion-api-calls-in-a-single-month/`

web.[23] However, RESTful APIs are no safer than the alternative and the performance of HTTP 1.1 is an ongoing concern[24]. Despite the shortcomings of REST, SOAP, RPC and RMI service consumers are more than willing to program against a service API; the API economy is thriving, the APIs evolve and adapt in response to market forces, service consumers harness these APIs to suit their own needs and goals, and service providers are learning that all APIs must be hardened irrespective of their location (public, partner, or internal), architectural style, or underlying technology (request/response, RPC, RMI, or others).

At first glance, mobile code as an architectural alternative to RMI, RPC or REST is patently unsafe, exacting a high price for service adaptivity. The dangers include waste of fungible resources (processor cycles, memory, storage, or network bandwidth), denial of service via resource exhaustion, service hijacking for attacks elsewhere, accidental or deliberate misuse of service functions, or direct attacks against the service itself. However, nearly a quarter century after the public debut of the web, we know that these dangers exist no matter what architectural style we adopt and that the risks vary only in degree, but not in kind, from one style to another. Nonetheless, the benefits of shifting service computation from service consumer to service provider are too persuasive to ignore. If a service provider can tamp down the risks to a tolerable level then a generalization of remote evaluation, one for which security and safety are first-order concerns, can be a feasible alternative for the implementation, deployment, and exploitation of decentralized services.

1.4 Capability Confines Risk

If mobile code is the answer to client-driven service adaptation, how can architecture sufficiently confine visiting mobile code to forestall the risks to services? From the perspective of mobile code there are two distinct forms of capability: functional and communication. *Functional capability* defines and confines what mobile code can do; that is, the actions it may take and the outcomes of those actions. For example, assume that visiting mobile code f is executing within the confines of service s. If no function to modify the data base d of s is available to f then it is impossible for f to directly change d. In other words, f lacks the functional capability to change d. *Communication capability* defines and confines where, when, how, and with whom mobile code can communicate. If f cannot communicate with any

[23] As of late November 2015 the Programmable Web catalog (`www.progammable.web`) contained 14,315 APIs of which 65% (8,911) are REST-based and 17% (2,432) are SOAP-based.

[24] QUIC (Quick UDP Internet Connection) (see `https://www.chromium.org/quic`), an experimental protocol for reducing web latency over that of TCP, is but one of many efforts to address that concern.

other computation g that itself may have access to functions to modify data base d (either directly or by way of a proxy), then f can not modify d by proxy. In other words, f lacks the communication capability to affect the contents of d.

Here capability security confines risk and minimizes the destructive consequences of erroneous or malicious mobile code—functional and communication capability are independent variables adjusted as needed to maintain an acceptable level of risk and confine the repercussions of abuse. An architectural style for which functional and communication capability are principal concerns must identify the sources of functional and communication capability and the means by which capability of both forms is propagated while giving services free rein to offer exactly the capability they deem appropriate.[25]

1.5 Computation Exchange: A Historical Perspective

REST [85, 86], the architectural theory underlying the web, explains much of the web's success, including its explosive growth and capacity to scale. The web's adaptivity and plasticity amply demonstrate the efficacy of stateless data and content exchange. Remote evaluation hints at an alternative to REST where the stateless exchange of mobile code augments or supplants data and content exchange. For example, the strict synchronous request/response exchanges of HTTP/1.1 [84] are now augmented by WebSocket [81] for two-way communication with servers that does not rely on opening multiple HTTP connections. And rare is the web page that does not contain server-generated JavaScript (essentially source-level mobile code) for the display and management of dynamic content.[26]

Erenkrantz, Gorlick, Suryanarayana, and Taylor [70] (hereafter Erenkrantz et al.) analyzed the evolution of web protocols from the perspective of *network continuations* in client/server interactions where each continuation encapsulates the "rest of the interaction;" that is, those computations left to perform before the interaction cycle is complete. Three earlier papers were particularly instructive here: Queinnec [197], Matthews et al. [160], and Shieh, Meyers, and Sirer [214].

Queinnec demonstrated that server-side continuations are an elegant mechanism with which to capture and transparently restart the state of ongoing evolving web interactions. He portrayed server/client

[25] There is a third form of capability that we address only tangentially, namely *resource capability*, which defines and confines fungible resources, such as memory, processor cycles, network bandwidth, and file space. While resource confinement is vital when considering mobile code this work is limited to functional and communication capability. There is a substantial body of literature devoted to resource sandboxing. I do not consider it further in this exposition.

[26] Ajax [101] and mashups [164] are excellent examples of these practices.

interactions as a form of "web computation" (represented by a program evaluated by the server) where continuations are employed to suspend and resume stateful page tours or service-oriented sessions that are client-parameterized but generated and executed server-side.

Matthews et al. presented a set of automated transformations based on continuation-passing style, lambda lifting, and defunctionalization that serialize the server-side continuation and embed it in a web page returned to the client. When the client responds to the web page the (serialized) continuation is returned to the server where it is "reconstituted," with the server resuming execution of the continuation. This example of continuation exchange (from server to client and back again) preserves context-free interaction (that is, the stateless interaction of REST [86]) between client and server and allows the server to scale by remaining largely stateless. In addition, service replication and restart are mostly transparent. Server replicas can easily respond to any request from any client at any point in the interaction cycle, while server restart is seamless since clients now supply the server with exactly the computational state it needs to resume the interaction.

Continuations can also be applied to the transport protocols underlying server/client interactions. The TCP protocol requires that connection state be maintained at both endpoints of the connection. Shieh et al. introduced a TCP-like alternative, called *Trickles*, in which all per-connection state is held on a single endpoint (the client), thus allowing the opposite endpoint (the server) to operate without any server-resident per-connection state. The Trickles protocol stack encapsulates and pushes initial server-side connection state to the client. Thereafter the client presents this state to the server in subsequent packets to reconstitute the server-side state and receives server packets and an updated connection state in return. This cycle continues until either the server or the client terminates the connection. With Trickles server reboots, connection redirection and failover execute transparently, easing the implementation and administration of large-scale, reliable services.

In each of these three works evaluation of a continuation "restarts" a prior interaction (computation) exactly at the point at which it left off. While under Queinnec those continuations never left the web server, Matthew et al. took it one step further. Their continuations were transferred across the network to the web client in response to a request and then back again to the server in a subsequent request, an ongoing exchange of continuations as the client/server interaction evolved. Shieh et al. proved that continuations could appear well below the application layer in an underlying transport protocol and, like

a web exchange of continuations, offer comparable benefits at the protocol layer.

To Erenkrantz et al. network continuations were a tantalizing hint of a more general form of interaction. In particular, continuations are a well-known control mechanism in programming language semantics: many languages, including Scheme, Smalltalk, and Standard ML implement continuations.[27] Functional programming is also a source of early implementations of mobile code systems. Halls' *Tubes* [121] explored the role of "higher-order state saving" (that is, continuations) in distributed systems. Using Scheme as the base language for mobile code, Tubes provides a small set of primitive operations for transmitting, receiving, and executing mobile code at Tubes sites. Tubes automatically rewrites Scheme programs in continuation-passing style to produce an implementation-independent representation of continuations acceptable to any Scheme interpreter or compiler. Halls demonstrated the utility of continuations in implementing mobile distributed objects, stateless servers, active web content, event-driven location awareness, and location-aware multimedia.[28]

MAST [255] is also a Scheme variant for distributed programming that introduced first-class distributed binding environments and distributed continuations (in the spirit of Tubes) accompanied by a security model. MAST also provided primitives for code mobility and granted developers fine-grain network control by supplying potent control and execution primitives.

Both Tubes and MAST achieve "computation mobility," the movement of "live code" from one network host to another. Other language bases are also feasible. Tarau and Dahl [234] achieved the same for the logic programming language *BinProlog*, again employing serialized continuations that are transferred from one host to another and then reconstituted.

In each case the mobile code implementation relied on the transfer of serialized continuations from one network host to another. We have multiple examples, at multiple levels (protocol, application and system) in which the network exchange of continuations conferred additional benefits. As Shieh et al. point out in a later paper on Trickles [215]:

> *The flexibility, performance, and security of networked systems depend in large part on the placement and management of system state, including both the kernel-level and application-*

[27] It comes as no surprise that the work of Queinnec [197] and Matthews et al. [160] were widely known throughout the functional programming community.

[28] Matthews et al. [160] were apparently unaware of Halls earlier implementation of stateless web servers based on continuations.

level state used to provide a service. A critical issue in the design of networked systems is where to locate, how to encode, and when to update system state.

REST [86] and its instantiation, the World Wide Web, is a premier example of the critical system-wide consequences of the placement and exchange of state.

With these examples in mind Erenkrantz et al. offered a generalization of network continuations: *computation exchange* (the computational analogue of content exchange) as the bilateral exchange of computations among decentralized peers. In this regime content delivery is a by-product of the evaluation of computations exchanged among peers. Computation exchange exploits existing core organizational functions, processes, and assets to create higher-level customized services, but imposes significant security obligations.

Computation exchange subsumes several well-known styles for distributed computing, including remote procedure call [18, 177], remote evaluation [227, 228], REST [86], and service-oriented architectures [72]. From the perspective of computation exchange remote procedure call is an exchange containing a single function call, remote evaluation is an exchange containing an entire function body, REST is an exchange of a small set (`GET`, `PUT`, `POST`, `DELETE`, and so on) of single function calls accompanied by call-specific arguments and metadata, and service-oriented architectures are higher-order compositions of remote evaluation.

Taking a cue from functional programming, computations are defined as closures, continuations, and binding environments [67]. Peers evaluate (execute) the computations they receive as messages from other peers—data and content are side effects of the evaluation of computations. Computation exchange generalizes remote evaluation; it broadens the domain of exchanges and eliminates the distinctions between clients and servers. It also clarifies the meaning of "service oriented architecture" where service providers offer execution sites for the evaluation of visiting computations and return the outcome of an evaluation as the response of an exchange.

Computation exchange was expressed as an architectural style, Computational REST (CREST) [69–71] and demonstrated in two decentralized applications: *Feed Reader* [69, 110, 236], for collaboratively viewing, filtering and sharing RSS feeds[29] and Firewatch [111], a system for wildfire detection and response.

[29] A video of the Feed Reader in action is available at `http://www.youtube.com/watch?v=0DYU1ffdTcg`.

So why is computation exchange interesting? After all we've had mobile code broadly available in the form of Java applets and JavaScript embedded in server-generated web pages since late 1995, almost two decades ago.[30] But look at the landscape from an architectural perspective. From early 1996 on every web browser was also, courtesy of Java applets and JavaScript, a computing platform that servers (hence services) could exploit on demand. Since its introduction JavaScript has become the fundamental technical enabler for delivering sophisticated services and applications (software as a service) to web browsers. The relationship is asymmetric as clients (browsers in this case) are still unable to transfer mobile code to servers for execution server-side but nonetheless this asymmetry generated a Cambrian-like explosion of system and software innovation. What would emerge if the relationship between server and client were fully symmetric—restated as the exchange of computations among peers? I argue that symmetric mobile code (or more generally mobile *live computations*) executed on both sides of a peering relationship is as radical and significant an advance as the adoption of mobile code executed client-side.

1.6 Security for Computation Exchange

COmputAtional State Transfer (COAST), the topic of this thesis, revisits computation exchange from the perspective of security [113, 114]. As noted in [110], CREST does not address security; while Erenkrantz [69] clearly demonstrated that CREST was possible and allowed for novel system structures and modes of interaction, it was never claimed that CREST-based systems were safe or secure.

Computation exchange between peers presents significant vulnerabilities, including waste of fungible resources (processor cycles, memory, storage, or network bandwidth), resource theft and exhaustion, interference among computations, service hijacking for attacks elsewhere, accidental or deliberate misuse of service functions, destruction of valuable assets (for example, databases, sensors, or actuators), noncompliance with service policy, and violations of peering agreements among multiple peers. A computation accepted from a trusted source may be erroneous or misapply a service function due to honest misunderstanding or ambiguity. Even a correct computation may expose previously unknown bugs in critical functions, leading to an inadvertent loss of service or serving as the starting point for a targeted exploit.

[30] Netscape Communications Corporation, *Netscape and Sun Announce JavaScript, the Open, Cross-platform Object Scripting Language for Enterprise Networks and the Internet*, PR Newswire, December 4, 1995.
`https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html`

A *vulnerability* is far more than a specific threat or attack tactic; it represents an entire class of computational hazards. Further, the complete elimination of a vulnerability is often impossible, as there are many reasonable circumstances in which a peer must balance the risks and rewards of engaging in computation exchange. The safest posture for a peer is to *never* execute a visiting computation and, as we well know by example and practice in the modern web [230], even that posture is no guarantee of safety.

The security goal here is peer integrity, in the senses of survival, internal consistency, lack of corruption, and bounded behaviors. Every visiting computation that the peer executes is a latent crippling or lethal intrusion. A robust peer aims at maintaining essential services at an acceptable level in the presence of hostile system compromise, the hallmark of an *intrusion tolerant system* [93]. Protecting peers against specific narrow threats is a fool's chase; an arms race against a talented, resource-rich, and tireless enemy.

For decentralized systems authentication, secrecy, and integrity are necessary but insufficient for asset protection as there is no common defendable security perimeter when function is integrated across the multiple, separate trust domains [270]. Here an attack on one authority threatens all. At best a breach may lead to failures in other trust domains. At worst a breached authority may undertake an "insider" attack against its confederates. With this in mind decentralization demands that security be everywhere always. Applications that cross authority boundaries inherently bring security risks; adaptations in such contexts only increase the peril, hinting that security should be a core *architectural* element.

Here I take a systems perspective of security. Losses are inevitable; the best that we can do is forestall untoward exploitation and minimize the consequences. Yes, a service may be abused and service providers should do what they can to raise obstacles to abuse, but the price paid for a useful service may include tolerating some level of misuse.[31] Strategic protection against a vulnerability requires that we weigh the benefits and consequences of exposure. In this light, the security goal for COAST is *adequate protection* against the vulnerabilities presented by visiting computations. To this end I focus on general, fundamental mechanisms that can be combined in different ways to protect a peer against broad classes of vulnerabilities.

[31] For example, credit card providers accept some degree of fraud as simply the cost of doing business. Reducing the extent and frequency of that fraud is important but no one, including the credit card providers, imagines that the rate of fraud will ever drop to zero.

1.7 Summary

The transformation of computation exchange from an idiom to a concrete, practical style rests upon the ability of architecture to imbue conforming systems with predictable and desirable characteristics. Mobile code is a *must* for computation exchange; therefore, safely confining visiting mobile code is a first-order concern for systems that embrace computation exchange. Confinement can reduce risk to acceptable levels and in so doing allows that same mobile code to be exploited for both security and adaptation. The evaluation, Chapters 8–11, demonstrates that the four mechanisms of COAST — mobile code, execution sites, communication by introduction, and capability URLs — can be employed by developers in many different ways to serve two distinct goals simultaneously: security *and* adaptation.

Chapter 2, immediately following, traces the long and rich history of mobile code while Chapter 3 provides a formal definition of COAST and articulates the consequences of the style rules. The semantics of *Motile*/*Island*, related in Chapter 4, further illustrate how language and infrastructure are shaped to conform to the style.

Chapter 2: COAST Roots — A Brief History of Mobile Code

Computation exchange insists on the transfer of live computations: *state plus code* — a snapshot of the (partial) execution state of a computation. Mobile code under COAST must comply with this requirement. There are three related, well-known representations of state plus code: objects, closures, and continuations.[1] A snapshot of a live computation will, of necessity, be a directed (possibly cyclic) graph of objects, closures, or continuations. Objects can embed other objects, closures can embed other closures and continuations, and continuations can embed other continuations and closures; there are no other alternatives.

For these reasons, with the exception of the early examples of mobile code, Section 2.1 1960–1985), this discussion is confined to those languages and systems for which live computations are the unit of network transfer. Excluded from consideration are remote procedure call (RPC), remote method invocation (RMI), the network transfer of source code as mobile code, or any mobile code systems whose arguments are limited to primitive values (numbers, strings, and so on).[2]

2.1 Early History of Mobile Code (1960–1985)

Mobile code has appeared in many guises, even predating the appearance of networks. For example, during the 1960s IBM introduced the concept of remote job entry [23] where remote minicomputers dispatched batch programs to mainframes for execution and received files in response for post-processing (typically printing). Here mobile codes (batch job submissions) were transported from less capable, distributed minicomputers to large centralized mainframes. While not mobile code as we think of it today, it is the first example of an executable representation transferred from one machine to another specifically for the purpose of remote execution.

[1] Objects, closures, and continuations differ in their formulation of "state plus code." Objects come in roughly two flavors, those that include state and code in each instance (such as a prototype-based language) or object instances that contain state and a reference to a code-bearing object class. On the other hand, a closure's lexical scope bindings are its mechanism of state transfer. Continuations, unlike a generic object or closure, capture the "rest of the computation," that is, both the state of the computation at the point of capture and the code that will carry the computation forward from that point. Continuation state appears in its stack frames, registers, and in the transitive closure of references from stack frames and registers into the heap.

[2] Consequently, the inclusion of source code JavaScript in dynamic web pages is ignored as is remote batch invocation [45], where the only arguments permitted are primitive values.

Proposals for the inclusion of mobile code were almost simultaneous with the invention of packet-switched networking. Long before a working network was available to the first four test sites of the ARPANet mobile code concepts were discussed during the early meetings (late 1968 through early 1969) of the ARPA Network Working Group (NWG), the progenitor of the Internet Engineering Task Force[3] [1]. One of the first Request For Comments of the NWG, RFC 5 issued in June 1969 [207], defined the *Decode-Encode Language* (DEL), a scripting language for managing input/output devices such as keyboards, mice, joysticks, consoles, and graphic displays. DEL included provisions for a machine-independent representation suitable for network transmission and host-specific interpreters. Shortly thereafter, Michel Elie, inspired by DEL, elaborated the exchange format and its interpretation as the *Network Interchange Language* in RFC 51 [68].

In the late 1970s, NODAL [28, 223] scripts, an interpreted language for real-time data processing, were transmitted among networks of minicomputers at CERN (the European Organization for Nuclear Research, Geneva, Switzerland), for remote real-time data collection and processing during experiments.

Mobile code was employed in Softnet, a packet radio network developed at Linkoping University in Sweden from 1980–1985 [272–274]. Each Softnet link layer packet contained an individual Forth program [271] whose execution at the Softnet nodes defined and managed network services at many levels including packet routing, datagrams, multihop virtual circuits, file transfers, and mailboxes.

In 1982 Wall observed [256]

> *network algorithms . . . can often be stated more clearly from the viewpoint of an* active message*, a process that intentionally moves from node to node . . .*

a harbinger of both *active packets* [218], network packets whose headers contain a program that is executed at each transited node, and *mobile agents*, programs that travel from host to host in pursuit of a computational goal or task.

In the same time period (approximately 1982) source-level mobile code appeared in another domain, laser printing, as the language Interpress, a page description language developed at the Xerox Palo Alto Research Center (PARC) for driving laser printers that took many of its features from Forth and an earlier PARC graphics description langage JAM [201, 224]. PostScript [2], a descendant of Interpress, was intro-

[3] http://www.ietf.org

duced by Adobe Systems in 1984 and adopted by Apple in March 1985 for the Apple LaserWriter, the first commercial printer to ship with an embedded PostScript interpreter. Here the execution of the mobile code (PostScript source) by the printer generates a printed page—the representation of the computation. In other words, from the perspective of a Postscript-capable printer, content is literally a side-effect of computation.

2.2 Remote Function Evaluation (1987)

The first application of bidirectional mobile code, symmetric mobile code exchanges between client and server, appears in the work of Falcone [76] where a LISP-like *Network Command Language* (NCL) is the expressive medium for distributed heterogeneous services. Implemented in 1984, NCL relies upon *remote function evaluation* (RFE). Like *remote procedure call* (RPC) [18, 177], RFE presents a single function call $(f\ a_1\ \ldots\ a_m)$ to a server for evaluation. However, unlike RPC where each argument a_1 is a mere data value such as an integer or a record structure, each argument a_i of a RFE invocation may recursively be a RFE.[4]

Functional nesting permits a single RFE $f(a_1,\ldots,a_m)$ to perform arbitrary computation since pure functional composition (with appropriate primitives) can express sequences, conditionals and loops—the necessary control structures of programming languages.[5] Further, the return value of an RFE is a list $(s\ e_0\ \ldots\ e_{n-1})$, $n \geq 0$ comprising a status s and zero or more RFE expressions e_i, Thus the mobile code exchanges are symmetric since the client must evaluate RFE expressions $e_0,\ldots,e_{n-1}$ to obtain the individual return values.

While RFE allows a client to ship arbitrary code to a server for execution, including definitions of subfunctions, the client program must ensure that any nonprimitive functions not available server-side are included in the shipment. Thus an RFE may fail if it invokes a function g for which there is neither a definition server-side nor an included implementation. Finally, in presenting RFE, Falcone explicitly articulates one of the principal advantages of mobile code, namely the opportunity for the client to combine services (functions) in manners not anticipated by the service providers, noting the concomittant increase in service flexibility and decrease in server complexity, to my knowledge, its first mention in the open literature.

[4] One can reformulate an RFE as a restricted form of closure and so, given its compound construction, it (barely) skates past the restrictions for review.

[5] Sequences, conditionals, and loops are also *sufficient* control structures in the sense of a Turing-complete language.

2.3 Remote Evaluation (1986)

In 1986 Stamos offered an alternative to RPC and RFE, *remote evaluation* (REV), for compiled languages [226–228] dealing with the challenges presented by compilation and seamless integration with the host language CLU [155]. REV, like RFE, executes a single procedure call, supplied by a client, at a specified server, written

$$\text{at } s \text{ eval } p(a_0, \ldots, a_{n-1})$$

where s is an expression denoting a server, p a denotation for a procedure, and each a_i a procedure argument expression. However, unlike RFE, the compiler, relying on service declarations that describe the types and procedures available at a server, ensures that the request it generates for the execution of $p(a_0, \ldots, a_{n-1})$ at server s includes all of the code and types required to execute the call server-side. Hence, unlike RPC, REV permits arbitrary programming language expressions as arguments and, unlike RFE, ships as much code as necessary to execute the call, including user-defined types and user-defined procedures q on which p explicitly or implicitly depends. Finally, unlike RFE, REV is asymmetric as code moves solely from client to server and not back again as it does in RFE.

However, REV implemented, for the first time, nested mobility, wherein a relocated (mobile) procedure may itself contain one or more REV requests. This feature improves modularity since a user may relocate (remotely evaluate) a procedure p without concern for the REV requests embedded within p. Noting that it is trivial to emulate RPC using REV, Stamos [227] offers an eloquent discussion of the advantages of REV over RPC, including reduced communication overhead among hosts, fine grain service interfaces better suited to the evolving (and perhaps unexpected) needs of clients, ease of application-specific decomposition into local and remote execution, and the decoupling of the evolution and refinement of service interfaces from applications.

2.4 Mobile Objects

Object mobility combines code mobility with *state mobility* as an object instance comprises both the instance-specific object state along with the code (methods) for manipulating that state—suggesting that object mobility is more challenging to implement than mobile code alone. Object-specific issues arise immediately, for example, given a class-based object system, transporting the code of the class definition C along with its instance state may be grossly inefficient if multiple distinct instances of class C

are repeatedly exchanged among hosts. The semantics of the object system itself may present significant obstacles, for example, the method definitions of object systems based on generic procedures (such as the Common Lisp Object System [55]) may be scattered throughout an application[6], making it difficult to locate the method definitions of a specific class.[7]

2.4.1 Emerald (1988)

The complex semantics of objects favor language-specific solutions to object mobility. One of the earliest implementations of object mobility, *Emerald*, was reported by Jul et al. [137] in 1988. Unlike other object-oriented languages of the time, such as Smalltalk, Emerald has no class/instance hierarchy (today we would call Emerald a classless or prototype-based langauge). Instead each object instance contains the instance-specific state and a reference to an immutable code blob, hence many instances may safely share the same code blob. Each code blob (itself an object in the Emerald language), like all other Emerald objects is assigned a global unique identifier. When an object *o* is moved from one location to another only the object instance state is moved including the global unique identifier of its code blob. Upon arrival of an instance the Emerald language kernel determines whether or not a local copy of the relevant code blob of *o* is resident. If so, then the kernel transparently relinks the state of *o* with the local copy. If not, the kernel locates and transfers a copy (using an algorithm detailed in [137]). Emerald was designed assuming a comparatively small number (at most hundreds) of homogeneous hosts (Digital Equipment Corporation VAXes in the first implementation) interconnected by a local area network and the techniques Emerald employed neither scaled to larger numbers nor applied to heterogeneous hosts [137]. However, Emerald is a landmark system—the very first to implement unrestricted object and thread mobility—and a breathtaking combination of functionality and performance.

Though the main line of Emerald development at the University of Washington came to a close in the late 1980s [22], significant extensions were demonstrated at the University of Copenhagen in the 1990s: unrestricted object and thread mobility among heterogeneous hosts (including the Sun 3, based on the Motorola 68000 processor, and the Sun SPARC) using a common intermediate bytecode virtual machine as the bridge between disparate processor architectures [229] and extending the reach of Emerald to wide area networks [188].

[6] One of the claimed advantages of generics since methods may be defined incrementally where and as needed.

[7] An object mobility implementation dealing with generics may have to transport method definitions "on demand" whenever remote evaluations invoke methods whose definitions were not included with the original transfer of the instance.

2.4.2 Obliq (1993)

A notable successor to Emerald was *Obliq* [30, 31], a prototype-based (in the style of Self [249]), lexically-scoped, interpreted language modeled after Modula-3 supporting distributed object-oriented computation and the network transfer of computations:

- Objects are local to a site and never transmitted to another site automatically
- Every object is transparently a *network object*, the subject of a network reference or the target of a remote method
- Closures, on the other hand, can be freely transmitted from one site to another but their lexically-scoped free variables retain their bindings to the originating site of the closure

The language primitives allowed programmers to implement full object mobility in less than 10 lines of code and its distributed semantics permits compute servers (remote execution engines), remote agents, agent migration, object migration, safe execution, application partitioning, application servers, and application migration [32]. Developed at the Digital Equipment Corporation Systems Research Center (Palo Alto, California), Obliq was available by summer of 1993.

2.4.3 Class-Based Languages (1987–1994)

Object-oriented mobile code for a class-based language is far more challenging to implement than for a prototype-based language such as Emerald or Obliq.

Smalltalk (1987–1990)

An early attempt to implement object mobility is the work of Bennett [13, 14] in 1987 for Smalltalk. However, Bennett only implements state mobility and assumes that the correct class definition is available in the image of the destination Smalltalk environment. To this end two primitives are provided, *move* and *copy*, where the former transfers the state of an object instance to a remote destination and deletes the instance at the origin and the latter replicates the instance at the destination while leaving the instance at the origin intact. Nonetheless, state mobility requires the marshalling (serialization) and unmarshalling (deserialization) of arbitrarily complex (and perhaps self-referencing) data structures. Here, Bennett relied on the earlier work of Vegdahl [251] whose techniques for transfering data structures among Smalltalk images addressed many, but by no means all, of the the technical obstacles.

COOL (1990)

The Chorus Object-Oriented Layer [120], implemented full object mobility in 1990, but did so at the operating system level with distributed shared memory, a mechanism that is patently infeasible for internet-scale code migration. In 1992 Davies, Blair and Mariani [54] report on object migration in the ANSAware distributed systems toolkit where objects are migrated, code included, at the level of granularity of a *capsule* containing one or more objects. However in a typical implementation, say for Unix, a capsule is equivalent to a process.

Telescript (1994)

The earliest example (circa 1994–1995) of full object mobility for a class-based language is *Telescript* [243], which permitted a Telescript agent to include application-specific classes in its continuation (the unit of Telescript code migration).

2.5 Functional Mobile Code Languages (1995–2007)

While many mobile code and agent languages are object-oriented, there are a number of experimental, functional mobile-code languages. This body of work, whose earliest examples date from the mid-1990s, emphasizes the transfer of closures and continuations among distributed hosts and exploits higher-order functional constructions to implement powerful distributed infrastructure.

2.5.1 Kali Scheme (1995)

Kali Scheme [34, 35] is a distributed Scheme dialect whose efficient network transmission of higher-order functional objects (closures and continuations) made it the first mobile code Scheme dialect (circa 1995). Closures and continuations are integrated into a message-based framework that allows any Scheme object to be sent and received in a message. Threads, the unit of Kali concurrency, are implemented in terms of continuations and thread migration from one address space to another amounts to continuation migration and restart. To reduce the latency of continuation transmission, continuation frames were faulted lazily on demand between address spaces, thereby reducing the bandwidth requirements for migration. Interthread communication within an address space relies on shared memory, locks, and higher-level abstractions constructed atop these primitives; communication between address spaces relies on message-passing. Kali Scheme compiles into a processor-independent bytecode virtual

machine striking a balance between an efficient native code compiler and a pure interpreter. The system also offers a variety of reflective primitives that give developers control over thread scheduling, messaging, access to system-wide global variables for the construction of higher-level concurrency, communication, and state sharing. As the authors themselves note, fault tolerance and security are two important issues left unaddressed by Kali Scheme.

2.5.2 Dreme (1995)

Dreme [98], like Kali Scheme, is a distributed Scheme dialect where all first-class language objects are mobile, including closures and continuations. Among the claimed benefits are: on-demand delivery of new software components, flexible use of network resources, fine-grained load balancing, and significantly *greater security control*, pointing out that while mobility allows better exploitation of resources selective *immobility* can protect resources from access by an untrusted agency and allow protected information to move to a trusted agency without changing the application. The fundamental insight of Dreme, that *servers are closures*—preexisting functions that supply an interface to encapsulated information—resembles in spirit the binding environments of COAST execution sites.[8] Each Dreme object is either mutable/immutable and mobile/immobile but mobile objects may be "pinned" by command to a particular address space, thereby preventing their migration to other address spaces. An immobile Dreme object is akin to an *Island* fixed asset, an immobile island resource that for reasons of security, access, or use restrictions may not be moved off-island.[9] Dreme language objects are compiled to processor-independent, bytecode instructions which are interpreted.

2.5.3 Tube (1997)

David Halls' Tube [121] is another distributed Scheme dialect whose goal is the transparent migration of stateful ongoing computations Capturing computations as closures and continuations, both of which combine snapshots of computational state with code, Halls demonstrates implementations of distributed objects, stateless servers, and session mobility where computations migrate to follow a user as she changes locations among a network of computers. Tube compiles Scheme programs into an intermediate form expressed in continuation passing style (CPS) (see [97], Chapters 5 and 6) for which continuation capture is trivial. For the purposes of network transmission (or checkpointing computations

[8] See Section 3.3.2 for an introduction to binding environments and execution sites.
[9] See Section 3.5 for a brief introduction to "islands."

as files) the CPS intermediate form is serialized into a structure-preserving byte sequence that can be converted back in memory into the original CPS representation. The CPS intermediate form is executed by an interpreter, itself written in Scheme. For those Scheme implementations that compile Scheme programs into native code[10] the additional layer of overhead introduced by the Tube interpreter can be compiled away.

2.5.4 MAST (2002)

MAST [255] is a Scheme dialect targeted toward network, distributed, and peer-to-peer programming.[11] MAST introduces distributed binding environments (that is, binding environments that are remotely accessible) and distributed continuation as first-class language constructions and, unlike many other languages, introduces a security model directed toward mobile code. Binding environments with multiple inheritance are first-class and map symbols (names) to MAST values using depth-first, left-to-right name resolution. Binding environments can be exported to local network access points, becoming execution contexts for visiting mobile code. In addition, function definitions support scoping operators that control where a symbol is evaluated and bound, in local scope or in the dynamic context of a binding environment. Remote evaluation in MAST is blocking, but a closure may be executed asynchronously and its return value bound to a promise on which a thread may wait (with an optional timeout). The MAST security model is designed to protect against three broad threats:

1. Attacks against the language interpreter
2. Attacks that exploit a computational environment (a binding environment exported by a MAST host)
3. Man-in-the-middle attacks against two communicating MAST hosts

Several distinct security mechanisms are defined:

- An *access controller* may be attached to a remote binding environment to filter incoming visiting code based on source address or other criteria
- A *decoder* may be attached to a remote binding environment that can be used to verify the mobile code, authenticate code signatures, and attach code credentials to visiting code

[10] For example, Bigloo (http://www-sop.inria.fr/indes/fp/Bigloo/) or Gambit (http://gambitscheme.org/).
[11] I'm not certain that MAST was ever implemented. To my knowledge no implementation was ever made available.

- An *encoder* may be attached to any remote binding-environment reference by a client to encrypt mobile code payloads, attach credentials, or provide application-specific attestations
- *Isolation* of binding environments and environmental sandboxing

2.5.5 Termite Scheme (2006)

Termite Scheme [103] is a distributed, mobile-code language whose message-based concurrency model is inspired by Erlang [7]. Termite implements both closure and continuation transfer—the base mechanism by which actor-like computations are spawned on remote nodes. Termite actors are extremely lightweight; a single Termite node can easily support tens to hundreds of thousands of individual actors. Actors communicate by exchanging messages and each actor is assigned an actor-specific mailbox in which incoming messages are deposited in arrival order. A message may be any first-class serializable value (number, string, character ...) or any compound structure such as a list, record, closure or continuation containing serlializable values. Messages are immutable and any shared structure within a message is preserved on message delivery. However, shared structure is not preserved across successive messages—in this case the structure is replicated.

Unlike other languages, not all data types are serializable (for example I/O ports), hence not mobile, and closures or continuations containing such data types can not be transmitted from one Termite node to another. Like Erlang, Termite exceptions can be propagated from node to node, which simplifies the implementation of "nanny" actors whose sole responsibility is to monitor the health and progress of remote computations. Implemented in Gambit Scheme[12] it relies on the (undocumented) Gambit bytecode virtual machine for mobile code serialization and transfer. The network transfer representation is also undocumented. Termite does not address mobile code security or safety and assumes that all Termite nodes are trustworthy.

2.5.6 Mobit (2007)

Mobit [192] is a portable implementation of Termite. Termite is intimately connected to the implementation of Gambit Scheme as its serialization format (both data and code) depends on the undocumented serialization primitives and bytescode engine of Gambit. Mobit is a portable implementation of Termite Scheme that translates Scheme source code into a specialized, continuation-passing style

[12] `http://gambitscheme.org/wiki/index.php/Main_Page`

Scheme interpreter. The translator, a closure compiler [78], is implemented as a set of nonhygienic macros that are executed by the underlying Scheme host. Each compiled Scheme closure "knows" how to decompile itself into a Scheme-independent list-based directed graph and the Mobit Scheme-independent serializer converts that graph into a flat, Scheme-independent representation. Mobit nodes deserialize incoming messages containing Mobit mobile code, primitive values, and Scheme compound structures, reconstructing the directed graph, and (re)compile the graph into the native closures of the Scheme host of the node. Essentially Mobit is a Scheme-in-Scheme interpreter designed expressly to support mobile code.

2.5.7 Other Functional Languages

Other statically-typed functional languages besides Scheme have also been used as the starting point for mobile code languages, notably Haskell [131] and OCaml [152].

JoCaml (1999)

JoCaml [44] combines primitives of the join calculus [96] with OCaml and extends OCaml with concurrency, message passing, and message-based synchronization. JoCaml is intended for agent programming and relies on the join calculus, a process calculus for distributed programming, as its network abstraction.

Haskell (2005)

mHaskell [65] is a mobile code dialect of Haskell based on the π-calculus, emphasizing closure transfer and higher-order combinators for distributed programming.

Summary

mHaskell and JoCaml are notable for their demonstrations of static typing in mobile code systems (a comparative evaluation of the two languages can be found in [82]) but neither have a security model, access control mechanisms, or address decentralized safety or security for mobile code.

2.6 Mobile Agents

Agents draw from both distributed systems and programming languages. As recalled by Alan Kay [141], agents, as a concept,

... originated with John McCarthy in the mid-1950s and the term was coined by Oliver G. Selfridge a few years later, when they were both at the Massachusetts Institute of Technology.

Mobility as a fundamental defining characteristic of agents was introduced in the early 1990s by Telescript [260, 261], an interpreted object-oriented language designed expressly for *remote programming* (RP), where continuations execute in remote environments and migrate from host to host as goals and immediate needs dictate.[13] White [260] observed that RP offered two distinct advantages, one tactical and the other strategic. As a tactic, RP reduces network demand in comparison to RPC by favoring computation at the server over repeated RPC requests between client and server. As a strategy, RP promoted client-directed customization since clients were no longer beholden to service providers to offer exactly the large-grain service (RPC entry point) a client might require rather, the particular services demanded by a client are dynamically installed by the client in the form of an agent.

2.6.1 Telescript (1994)

Telescript [242, 243] was an early (circa 1994) object-oriented, mobile agent system for which security was a concern [235]. Mobile Telescript agents were executed by a host-independent virtual machine within *places*, virtual locations devoted to a particular service: for example, a ticket purchase or catalog search. Mobile agents and places were tagged with a designation of authority (the originating organization). Agents were granted permits by the managing authority of the place, which confined the capabilities granted to an agent and set resource caps. From our perspective Telescript is an early mobile-code-based, decentralized service architecture.

2.6.2 Agent Tcl (1997)

Another language, Agent Tcl [116, 149] (now D'Agents), had four principal goals: ease of agent migration, transparent communication among agents, support for multiple agent languages, and effective security. Agent Tcl implements "whole" agent mobility, where the unit of code mobility is the entire binary image of the agent. Unlike many other agent systems agent identity is not preserved across jumps. Agent Tcl employs public/private key cryptography for authenticating binary images as they jump from server to server while Safe Tcl [187] confines the executing Tcl agents; a set of trusted scripts provide limited access (based on access control lists) to unsafe functionality.

[13] A draft of the landmark 1994 Telescript white paper was circulating among Silicon Valley startups in 1993. I recall reading it while working as an independent consultant in Silicon Valley during that time.

2.6.3 Other Work (1998–2013)

More recent work includes the Foundation for Intelligent Physical Agents [95], AgentScope [179], AgentOS [37], SeMoA [205], JADE [11, 12], and Ajanta, a Java-based system for mobile agent programming [138, 139, 248]. Like SeMoA, Ajanta addresses protecting hosts from hostile agents and protecting agents from malevolent hosts. Each Ajanta agent performs actions on behalf of some principal, an entity that has a unique identity in the system. Each agent travels with credentials including its owner, home site, code base (for loading agent-specific classes), and restrictions on its access rights. Signed credentials are used for access control by the Ajanta agent servers that provide confined environments for executing visiting agents and allow the host to regulate access to vital host resources. Agents migrate from host to host on demand using an encrypted TCP connection with standard Java object serialization. Agent server resources are never directly revealed to the agents; instead a unique proxy instance is allocated for each resource an agent requests (modulo access restrictions). In effect, the proxy acts as a capability for the resource, an example of the object-capability model of E [170]. Finally, Ajanta agents can protect themselves against some forms of tampering by malicious agent servers.

Additional work focuses on agent-based adaptation [259] and decentralized, agent self-adaptation [148], but neither addresses the relevant security concerns.

2.7 Mobility Semantics (1996)

In 1996 Cugola, Ghezzi, Picco and Vigna [50], recognizing mobility as a powerful enabling mechanism for constructing internet-scale distributed applications, undertook a survey of the concepts of code and state mobility in eight "mobile code languages" (MCLs) of the day: Java, Telescript, Obliq, Agent Tcl, TACOMA, M0, Tycoon, and Facile. Their survey criteria included scoping and name resolution, dynamic linking, state distribution, migration strength, and replication/sharing. I discuss each briefly below.

2.7.1 Scoping and Name Resolution

The (re)interpretation and binding of names appearing in mobile code is critical, as those names may refer to values left behind at the site of origin. Name resolution may be automatic or the environment may supply hooks allowing developers to define application-specific resolution procedures.

2.7.2 Dynamic Linking

When mobile code m arrives at a destination it may require access to code modules or packages that are available at the destination but not yet loaded into the execution image of the "virtual machine" executing m. In this case the requisite modules must be incorporated on-the-fly and their entry points linked to the invocation sites buried within m. In addition, in an effort to conserve network bandwidth, the mobile code m may leave supporting codes behind relying on code-on-demand downloads from other sites (which may or may not include the origin of m) to fetch the code as execution proceeds.

2.7.3 State Distribution

The degree and manner in which mobile codes may also include state and non-code resources in their transport varies among mobile code languages—affecting expressiveness, ease of use, distribution, and security.

2.7.4 Migration Strength

Two distinct forms of mobility are identified, *strong* and *weak*. Strong mobility combines both state and code transfer while weak mobility is restricted to code transfer alone (including free variables captured in lexical scope). From a functional perspective continuation transfer is strong mobility while closure transfer is weak mobility.

2.7.5 Replication, Sharing, and Interdiction

Non-code resources may be replicated, shared, or interdicted.[14] For example, the value v of a binding α/v within a closure λ at an origin site x may be an input stream. When λ transits to a destination y the binding α/v may be:

- Replicated at the destination in which case v may be either retained or deleted at the origin x. In either case α will be rebound to a comparable input stream at the destination; however, the original input stream may or may not be retained at the origin.
- Shared between origin and destination, in which case α remains bound to the input stream of the origin x.

[14] Cugola, Ghezzi, Picco and Vigna [50] omit *interdiction* as a resource management strategy, presumably as it was not employed by any language included in their survey; however, a more recent example, Termite [103], does exactly this for particular data types.

- Interdicted, where α is rebound to an innocuous constant such as `nil` or `void` at the destination y and v is left unchanged at the origin x.

2.8 COAST Dictates Language Semantics and Infrastructure

Almost without exception prior work on mobile code is language-centric and, to the extent that software architecture plays a role, it is at best, implicit. That said, the prior work is innovative and compelling, and the examples presented by these works give credence to the claim that mobile code is a powerful tool for adaptation. Unlike many other mobile code systems, computation exchange, the COAST style, and the security goals dictate many of the choices for *Motile* the language, and *Island* the peering infrastructure.

Computation exchange, from this point forward, is restricted to the exchange and evaluation of *closures* and *continuations*, among peers. The COAST style embeds the exchange of closures and continuations within an object-capability model that strictly confines the functional and communication capability of executing closures and continuations. As I demonstrate the COAST style spans both inter- and intra-peer executions, that is, each peer, in isolation, is itself implemented as a COAST-compliant system in miniature. The interplay among computation exchange, functional capability, and communication capability can be manipulated for the sake of both adaptation and security. The mechanisms that ensure safe interactions among peers are exactly the mechanisms that ensure safe interactions within peers.

The topics of Chapter 3, presents a formal statement of the style (Section 3.3), and treats topics that go unmentioned in many (though certainly not all) language-centric investigations including: a security-centric perspective that includes a threat model (Section 3.1; a communication model for communication by introduction, Section 3.2; capability URLs, Sections 3.2.4 and 3.6; and the requirements for a notional peering infrastructure, Section 3.5.

Chapter 4 carries this one step further, analyzing the degree to which *Motile* is COAST-compliant. COAST the style, impinges on *Motile*, in numerous ways, large and small, and security concerns dictate specific language features. In particular, COAST security dictates the choices for scoping and name resolution, dynamic linking, state distribution, migration strength, replication/sharing, and interdiction — the principal distinguishing language characteristics identified by Cugola, Ghezzi, Picco and Vigna [50]. In these respects my work as a whole reflects a notable architecture- and security-centric perspective.

Chapter 3: Computational State Transfer: The Style

Unlike many architectural styles COAST is deeply security-centric, as befits a style dedicated to the safe exchange and execution of live decentralized computations. Since service adaptability and integrity are principal concerns the style mandates architectural elements that provide functional, and communication security. In exchange for the complexity imposed by these security mechanisms, COAST allows the construction of on-demand tailored services and enables a wide range of dynamic adaptations in decentralized services.

To adequately present and explain the COAST style I must, by necessity, introduce terminology and concepts critical to understanding security policy for computation exchange, and hence COAST. Section 3.1 discusses the threat model of computation exchange identifies four distinct layers of security policy — *oracle*, *feasible*, *configured*, and *actual* — as put forth in Carlson's layered security model, *The Unifying Policy Hierarchical Model* [33].

Section 3.2 describes the communication model underlying the capability URLs of COAST and relates how security policy for communication among computations seeks to prevent, delay, or minimize the damage of insider attacks by modulating the communication behaviors of computations. This section presents an overview of:

- The underlying abstractions of COAST communication: ingress points, transports, and egress points;
- The trajectory of communication, the path that a message follows as it is transmitted from one computation to another;
- Capability URLs (CURLs), the only capablity that grants a computation to transmit a message to another computation; and
- Gates, contracts by predicate to regulate inter-computation communication.

Section 3.3 presents a formal statement of the COAST style and defines the style in terms of services, execution sites, messaging, and message interpretation.

Section 3.4 relates a few informal scenarios to illustrate how COAST puts the four elements discussed in Section 3.3 (services, execution sites, messaging, and message interpretation) to work to provide security and adaptation. These scenarios rely on the communication model of COAST (Section 3.2) and the notional COAST infrastructure presented in Section 3.5.

Sections 3.6 and 3.7 provide additional details on CURLs, gates and computations — subjects that were briefly introduced earlier in Section 3.2. Section 3.8 summarizes the salient points made in this chapter.

3.1 The Threat Model of Computation Exchange

Computation exchange presents a distinctive security challenge; since peers exchange and evaluate computations, nearly every interaction amounts to an *insider threat*. Nor is there any hope of escape. A peer denying execution access to visiting computations constitutes denial of service. A peer that broadly refuses the computations of other peers may find itself shunned when it exports its computations for evaluation elsewhere. Local computations (those that originate on the peer) are operationally indistinguishable from visiting computations. Absent knowledge of a point of origin the only differences among computations lie in the capabilities granted to each by the hosting peer. Consequently, "insiderness" is not a binary measure, local or non-local, but a nuanced assessment of threat and risk [20].

Carlson's layered security model, the *Unifying Policy Hierarchy* [33], identifies four distinct layers of security policy: *oracle*, *feasible*, *configured*, and *actual*. Each layer is a (perhaps partial) decision procedure over the space

$$\langle s, o, a, e \rangle \in \text{S(ubjects)} \times \text{O(bjects)} \times \text{A(ctions)} \times \text{E(xternals)}$$

where (in the world of computation exchange) a *subject* is a computation, an *object* is a resource of interest (sensor, data object, message, or computation, to name but a few of the possibilities), an *action* names the act to be undertaken on the object by the subject, and *external* represents externalities such as intent, law, regulation, or standard practices.

Oracle policy, the topmost layer, is a non-deterministic (and likely non-computable) total oracle over $S \times O \times A \times E$ and for any $\langle s, o, a, e \rangle$ returns either true or false. This layer is the ideal security policy for

Table 3.1: The four levels of Carlson's Unifying Policy Hierarchy (adapted from [21] Table 1).

Level	Domain	Description
Oracle Policy	$S \times O \times A \times E$	Idealized authorizations accounting for externalities.
Feasible Policy	A subset of $S \times O \times A$ restricted to system-*definable* entities	Authorizations in practice taking into account system constraints.
Configured Policy	A subset of $S \times O \times A$ restricted to system-*defined* entities	Authorizations consistent with the system configuration.
Actual Policy	A subset of $S \times O \times A$ restricted to system-*defined* entities	A system implementation including any flaws or vulnerabilities.

the system and is capable of capturing external intangibles such as intent or nuanced legal concepts such as "the preponderance of the evidence." The remaining three lower layers ignore externalities and are restricted to the domain $S \times O \times A$.[1] The next layer is *feasible policy*, a decision procedure over a subset of the domain of the oracle that is feasible to compute with regard to the system of interest. This partial decision procedure represents the outer limit of effective security policy given known characteristics of the system (such as architectural style, algorithms, principal data structures, programming languages, and implementation). In systems of any complexity the domain of the feasible policy is a proper subset of the domain of the ideal policy. There can be many distinct feasible policies consistent with a given ideal policy. The *configured policy* is the instantiation of the feasible policy on a particular system with a specific configuration (for example, by convention all "sensitive" files have a file name whose prefix is `SENSITIVE_`). The configured policy reflects the details of mapping the feasible policy onto a known system. Multiple configured policies, each configuration reflecting the idiosyncrasies of a particular system, can be consistent with a single feasible policy. Finally, the *actual policy* is the working implementation of the configured policy on the live system. Table 3.1 summarizes the layers and their restrictions.

Bishop et al. [20] define an *insider threat* as any gap or inconsistency between any two layers of a system's Unifying Policy Hierarchy. For example, an oracle policy may state that an employee has access to a sensitive database only if the employee is a manager and access is exercised on behalf of company clients. Alice, a manager, can mount an insider attack and exfiltrate valuable data since the lower level policies can not detect a critical externality—the rationale or motivation for Alice's access. In all but the simplest systems the feasible policy captures only a subset of the decisions of the oracle and, for many

[1] Here, equivalently $S \times O \times A \times \perp$

$e \in E$, may grant access

$$\text{feasible}(s, o, a) = \text{true}$$

when

$$\text{oracle}(s, o, a, e) = \text{false}$$

dictates otherwise.

Crampton and Huth [49] observe that identity alone is insufficient for trust decisions and advocate for a dynamic (re)evaluation of trust so that

> *... privileges for which users are authorized can be modified automatically (i.e. without human intervention) when those users are revealed, or suspected, to be untrustworthy.*[2]

As a step towards this goal Crampton and Huth propose strengthening the definitions of feasible, configured, and actual policy shown in Table 3.1. Let an access request be a tuple $\langle s, o, a, c \rangle \in S \times O \times A \times C$ where C represents the set of feasible externalities (context); for example, the time of day, the communication history of a computation, or the the recent behaviors of computations originating from peer p that executed on other peers q. Feasible, configured, and actual policy now compute over subsets of $S \times O \times A \times C$. Given $X \subseteq S \times O \times A \times C$, p_X^+ denotes a policy that authorizes all requests in X, while its dual p_X^- authorizes a request if and only if it is not in X. We define a *request predicate* as any total boolean function whose domain is such an X. A *context predicate* is any total boolean function whose domain is $Y \subseteq C$. Extending the Unified Policy Hierarchy opens the door to:

- Richer forms of policy decisions that go well beyond the classic "grant" or "deny" to identify, articulate, accommodate, and propagate: inconsistencies, gaps between policy levels, error conditions, and assessments of trustworthiness or risk
- Explicit construction of request and context predicates as piece parts for actual policy.
- Policy meta-interpretation to extract and analyze subsets of permissible requests, identify inter-policy conflicts, or gaps within a policy.
- Policy combinators for constructing a policy from generic and application-specific subpolicies.

Computation exchange places insider attacks front and center as a principal risk that is impossible

[2] [49], Section 4, page 186.

to ignore. Moreover, gaps between the levels of the Crampton and Huth version of the Unifying Policy Hierarchy are also inevitable for any realistic system of even modest complexity. For the gaps between the oracle policy and the feasible policy the best that we can do is to expand the domain of the feasible policy ($S \times O \times A \times C$) to include necessary and sufficient context $C \subseteq E$ that approximates (again where feasible) otherwise incomputable externalities E. Many of the gaps between an oracle policy and the designated feasible policy must be addressed by nontechnical means such as laws, regulations, customs, social norms or organizational policy, and societal remedies such as fines, restitution, incarceration, legal restraints, audits, social restrictions, or boycott. However, improving the expressiveness, efficiency, and breadth of the three lower layers of the unified policy (feasible, configured and actual) is a practicable goal.

Here I apply security policy to regulate communications among computations and prevent, delay, or minimize the damage of insider attack by modulating the communication behaviors of computations. At the level of configured and actual policy the domain $S \times O \times A \times C$ comprises:

- S, a set of (decentralized) computations
- O, a set of communication capabilities for sending and receiving messages
- A, the two functions send and receive
- C, implementation- and application-dependent contextual state.

Following the lead of Peyton-Jones and Eber [191] I introduce a stateful, embedded, domain-specific, functional language for expressing security policy predicates and low-level policy monitoring over the communication behavior of decentralized computations. To the extent that C, contextual state, can be introduced into a system[3] that state can be exploited by the domian-specific policy language. While far from a complete solution it points the way toward a declarative, functional policy language that spans all three lower levels of the unified policy called out by Crampton and Huth.

3.2 The Communication Model of COAST

COAST computations, each an individual thread of control, communicate by transmitting and receiving messages. COAST computations:

[3] Obviously questions of performance may dictate the exclusion of otherwise useful or informative contextual state.

- *Partition message transmission, reception, and transport.* The right to transmit a message is distinct and separate from the right to receive a message. In many models, for example the π-calculus [172], possession of a *channel* conveys the rights for bidirectional communication—message reads and writes are permitted on either end of the channel. COAST channels are *unidirectional* with distinguished endpoints, one reserved for transmitting messages and the other reserved for receiving messages (much as a Unix pipe has a read end and a write end), but mere possession of a channel does not grant either the right to transmit or receive. Each of those two rights is conveyed independently by a separate abstraction. The semantics of message transport are independent of transmission or reception and can vary from channel to channel, say a queue in one case and a sliding window in another.
- *Communicate by introduction.* A computation x can transmit a message directly to computation y only if x was *introduced* to y. A computation x acquires an introduction in exactly three ways: it was granted at birth, received in a message from another computation (itself introduced to x at some point in the past), or returned as the result of a function call.
- *Communicate asynchronously.* Message transmission and message reception are asynchronous (non-blocking) operations.
- *Differentiate communication.* Communication by introduction is necessary but insufficient protection. Consequently the introductions acquired by a computation over its lifespan may be differentiated in time, space, and context to restrict when and under what circumstances it can transmit a message to its acquaintances.

3.2.1 The Underlying Abstractions of COAST Communication

A *transport* t is an unidirectional channel and is the only means by which messages pass from one computation to another [112]. A transport t may have zero or more *ingress points* $t\bullet_1, \ldots, t\bullet_m$ and zero or more *egress* points $t\triangleright_1, \ldots, t\triangleright_n$ as shown in Figure 3.1.[4] Possession by a computation of a transport

Figure 3.1: A generic transport with multiple ingress and egress points.

[4] As a matter of notation $t\bullet$ denotes any ingress point $t\bullet_i$ and $t\triangleright$ denotes any egress point $t\triangleright_j$.

ingress point $t\bullet_i$ conveys the capability to transmit a message via transport t. Conversely, possession of a transport egress point $t\triangleright_j$ by a computation conveys the capability to receive a message via transport t.[5] The ingress and egress points of a transport separate the capability to transmit a message from the capability to receive a message. Since ingress and egress points are transport-specific (each ingress or egress point refers to exactly one transport) a pair $< t\bullet, t\triangleright >$ defines a specific *partition of communication*.

Ingress and egress points can be shared among computations; multiple computations can simultaneously transmit and receive messages via the same transport t. The semantics of transmission and reception are transport-, ingress-, and egress-specific; however, a common model is a simple queue in which messages are enqueued and extracted in transmission order. In Figure 3.2 x and y are two distinct computations sharing a transport t for which there exists exactly one ingress point and one egress point. As long as neither x nor y share the their respective ingress and egress point t is a private and secure channel from x to y (an application of the object-capability security of Miller [170]).

Figure 3.2: A dedicated transport between computations x and y.

COAST strives to be practical, in the sense that efficient and effective implementations are feasible. To this end I distinguish between computations that share a memory address space and those that do not. There are a variety of effective shared memory architectures, for example, multiple processes in a shared-memory multiprocessor, preemptively scheduled (at the operating system level) threads within a single operating system process, or green threads animated by one or more process-level threads. Shared memory reduces message passing to reference passing and with sufficient care and attention to blocking operations, such as input/output, green threads can be preemptively scheduled.

3.2.2 Gates

In a single address space computations can share the ingress and egress points of transports; this allows the communication topology within the address space to change over time.[6] Ingress and egress

[5] The orientation of ingress and egress are relative to the transport t. An ingress point $t\bullet$ conveys the capability to insert a message into transport t (transmission) while an egress point $t\triangleright$ conveys the capability to extract a message from transport t (reception). In other words, an ingress point is where messages enter a transport and an egress point is where messages exit a transport.

[6] For security reasons transports are often closely held in trusted computations and are not generally shared or transferred among computations. In contrast, the ingress and egress points of those transports are frequently shared or transferred among

points can carry constraints, called *gates*, that regulate communication. Distinct ingress (egress) points for a given transport t may enforce complex temporal, state, and performance constraints including use counts, rate limits, congestion or load-based throttling, time locks, restrictions in message structure and contents, restrictions based on trust or roles, and expiration dates, to name but a few of the possibilities. Shared ingress (egress) points can implement a common policy across multiple computations in an address space; for example, enforcing workload-dependent access to a transport or restricting access to a select few computations.

In shared memory ingress points implement *communication by introduction*. Let y be a computation holding the egress point $t\triangleright$ of a pair $< t\bullet, t\triangleright >$. If computation x acquires $t\bullet$ (at birth, via a message, or as the return value of a function call) then x has effectively been introduced to y. Absent possession of some $t\bullet$ it is impossible for x to directly communicate with y. If there exist multiple egress points $t\triangleright_1, \ldots, t\triangleright_n$ held by multiple computations $y_1, \ldots, y_s$ then the acquisition of $t\bullet$ by x introduces x simultaneously to each computation y_i. For example, the $y_1, \ldots, y_s$ may be a pool of identical computations (services) sharing a common work queue for which x is but one of many clients. Conversely, given pair $< t\bullet, t\triangleright >$ if a computation y acquires a $t\triangleright$ then any computation x holding a $t\bullet$ is now an acquaintance of y. The ordering of the acquisition of ingress and egress points by computations is immaterial to introduction. A computation may acquire over its lifespan many ingress and egress points for many distinct transports and a computation may simultaneously be an acquaintance of, and acquainted with, many other computations.

The shared ingress points of a transport t can implement a common admission policy across multiple computations, for example granting access to a service only if the workload falls below a set level or coordinating multiple computations; computation y is allowed to transmit a message to computation z only if computation x has not transmitted a message to z within the last minute. Similarly, shared egress points can implement a common extraction policy across multiple computations, for example, regulating the rate, for a group of computations, at which messages may be extracted from a transport or guaranteeing that only the members of a distinguished set of computations can extract messages via a particular egress point.

---

computations.

3.2.3 The Trajectory of Communication

A *transmission trajectory* is the path

$$t\bullet \longrightarrow t \longrightarrow t\triangleright$$

that a message m follows as it transits through an ingress point $t\bullet$, into a transport t and exits via egress point $t\triangleright$. Given a transport t, ingress points $t\bullet_1, \ldots, t\bullet_m$, and of egress points $t\triangleright_1, \ldots, t\triangleright_n$ we have $m \times n$ distinct transmission trajectories

$$t\bullet_i \longrightarrow t \longrightarrow t\triangleright_j \text{ where } 1 \leq i \leq m, 1 \leq j \leq n$$

The delivery of a message m is $t\bullet$, t, and $t\triangleright$ dependent since the constraints of $t\bullet$ and $t\triangleright$ are evaluated before m is injected into (or extracted from) transport t and the semantics of t can vary independently of its ingress and egress points. A CURL u defines two of the three elements of a trajectory, the ingress point $t\bullet$ and indirectly, the transport t. Any computation holding two CURLs u and v can, at a minimum, distinguish whether or not u and v both denote the same ingress point.

The interpretation of a message received by a computation may be egress point- and computation-dependent even if the full trajectory of the message is unknown to the computation. For example, a computation may expect data from sensor a via $t\triangleright_a$ and data from sensor b via $t\triangleright_b$. In summary, there are several avenues by which a computation may discover a portion of the trajectory of a message and that knowledge can be exploited by the computation to refine or differentiate its response.

3.2.4 Capability URLs

At this point we have ingress points $t\bullet$ as the capability for message transmission within a single address space. However, in a decentralized world collaborating computations may be scattered across many distinct and separate address spaces on many hosts. How can two computations x and y, each residing in a different address space, communicate when neither has direct access to the ingress points in the address space of the other? To span the gap between address spaces I employ the *Capability URL* (CURL) and, at the same time, eliminate the distinction between intra- and inter-address space communications.

Each CURL u denotes an address space A and an ingress point $t\bullet$ in that address space. CURLs grant

the capability to transmit messages both within and between address spaces. Computation x in address space A can transmit a message to computation y in address space B (where A might be B) only if x holds a CURL u that denotes address space B and an ingress point $t\bullet$ of a transport t for which y holds an egress point $t\triangleright$. Holding a CURL u is necessary but not sufficient for message delivery to y:

- The policy of the ingress point $t\bullet$ denoted by u may, at the time of transmission, block insertion of the message into transport t
- The policy of the egress point $t\triangleright$ held by y may, at the time of reception, block extraction of the message from transport t
- There may be be multiple egress points $t\triangleright_1, \ldots, t\triangleright_n$ for transport t in address space B and consequently, there is no guarantee that computation y, holding $t\triangleright_i$, will be the computation that receives (extracts from transport t) the message sent by x

As CURLs can be exchanged in messages between computations within a single address space or between remote computations in distinct address spaces the communication topology of both co-resident and remote computations can change over time. CURLs unify communications irrespective of the relative locations of the transmitting computation and the ingress point $t\bullet$ referenced by the CURL. This serves two purposes:

- It pushes transports and their ingress and egress points into a lower layer of the communications stack while preserving their fine-grain division of communication capability. Separating and hiding critical details of communication capability improves system integrity
- In combination with critical features of *Motile*, the mobile code language, it largely erases the distinction between intra- and inter-address space communication. Unifying local and remote communications simplifies the programming model, reduces or eliminates several possible sources of errors, and eases experimentation and development

Since *Motile* is a single-assignment language with persistent functional data structures [180] and all data is read-only a computation may largely ignore the distinction between co-resident computations (those sharing the same address space as itself) and remote computations (those residing in a different address space). Data can be safely shared among computations irrespective of location.[7] This eliminates

[7] Obviously communicating with a co-resident computation is far less expensive than communicating with a remote computation. The subtle semantic differences between co-resident and remote communication are discussed in detail in Chapters 4 and 7.

many forms of data races and simplifies interactions among computations. Finally, developers can begin with an application whose computations are entirely co-resident and gradually, over time, move computations to other address spaces to improve robustness, enhance security, increase isolation, or leverage specialization.[8]

Transports, ingress points, and egress points cleave computation from communication. CURLs reinforce that separation by masking the distinction between co-resident computations and remote computations. Consequently, services can be implemented "under the hood" as a closely knit collection of specialized co-resident and remote computations and present themselves, via CURLs, as a single "virtual" computation, irrespective of the location of the client computation, be it co-resident or remote.[9] The communication model of COAST was inspired in part by Weaves [112], an architectural style that supports run-time software adaptation [183, 184], and whose queues, read ports, and write ports anticipated the capability semantics of transports, ingress points, and egress points. The "blind communication" policy of Weaves, where active computations (tool fragments in Weaves parlance) are ignorant of the sources of their inputs or the destinations of their outputs, is an early example of a strict separation of computation from communication and was a crucial element of the dynamic adaptation of Weaves.

3.3 The COAST Architectural Style

For any mobile code system there are two fundamental issues: communication and confinement; that is, how independent units of mobile code contact one another and exchange information and how their executions are confined to prevent damage to their hosts or other computations. COAST **Services** specifies the form and content of communications: asynchronous messaging of mobile code comprising primitive values, closures, continuations, and binding environments and implicitly the meaning of service: computation-specific interpretation of mobile closures, continuations, and binding environments. **Execution** defines execution sites as a basic mechanism for functional and resource confinement. **Messaging** regulates how communication capability is allocated among computations. In particular, there is no ambient communication capability; without CURLs a computation is mute and without egress points a computation is deaf. Finally, **Interpretation** specifies that message interpretation is not only receiver-dependent but also delivery-dependent; both the CURL denoting the ingress point of the message and

[8] Like advantages accrue to Erlang [7], a single-assignment language for distributed systems.

[9] This mirrors the behavior of web servers whose services, organized as collaborating helper threads, process requests from distant browsers [10].

the consequent transmission trajectory of a message can influence the interpretation.

The COAST style states:

- **Services:** All services are computations whose only interactions are the asynchronous messaging of primitive values, *closures*[10], *continuations*[11], and *binding environments*[12].
- **Execution:** 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.
- **Messaging:** Computation x can transmit a message to a computation y only if there exists $\langle t\bullet, t\triangleright \rangle$ such that x holds a CURL u denoting $t\bullet$ and y holds $t\triangleright$.
- **Interpretation:** The interpretation of a message delivered to computation y via CURL u is y- and u-dependent.

3.3.1 Services

The definition of service varies from one architectural style to another:

- COAST emphasizes the exchange of closures, continuations, and binding environments, each stateful representations of live computations, and the CURL-dependent interpretation of that state by the recipient—a *service* interprets computational state on behalf of another agency [113]. There is no distinction between service provider and service client, as the behavior of both is essentially identical.
- REST emphasizes the transfer of representational state; that is, the state of a resource named by an URL for which a server produces a specific representation when requested by a client [86]. More generally, a RESTful service interprets a small number of methods in a manner that is both URL-dependent and context-free [70].
- CREST, the predecessor of COAST, identifies computations, named by URLs, as the fundamental network resource. Resource representations include primitive values, closures, continuations, and binding environments plus optional metadata. Computations, named by URLS and hosted by service providers, are interpreters for the closures, continuations, and binding environments dispatched by service clients (who themselves may be service providers). Like CREST, a COAST

[10] Functions plus their lexical-scope bindings.
[11] Snapshots of execution state [79].
[12] Maps of name/value pairs [133].

service interprets computational state on behalf of another agency; however, the interpretive mechanisms and naming structures (URLs and CURLs respectively) differ significantly in both form and meaning [69, 71]

- The SOA definition of service is far more muddled [71, 72]. The most charitable concrete interpretation possible is that a SOA exchange amounts to the remote evaluation by the provider of a single function call issued by a client

Unlike other distributed architectural styles such as REST or SOA, higher-order service composition (services composed from services) and service combinators (functions whose arguments may themselves be services) arise in COAST naturally from computation exchange and the symmetry of providers and clients. This flexibility expands the range of behaviors available to both sides of the exchange.

3.3.2 Execution

Over its lifespan each COAST computation is confined to an *execution site* $\langle E, B \rangle$. The execution engine E may vary from one execution site (and computation) to another: for example, a Scheme interpreter or a JavaScript just-in-time compiler. The execution engine defines the execution semantics of the computation and the machine-specific limits (such as resource caps) imposed upon the computation. The binding environment B contains all of the functions and global variables available to the computation at that execution site. Names unresolved within the lexical scope of a closure f (that is, the *free variables* of f) are resolved, at time of reference, within the binding environment B. If B fails to resolve the name the computation is terminated.

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

3.3.3 Messaging

CURLs convey the ability to communicate between computations.[13] A CURL u issued by a computation x is an unguessable, unforgeable, tamper-proof reference to an ingress point $t\bullet$ held by x; it contains cryptographic material identifying $t\bullet$ and is signed by x's execution host. A CURL u referencing $t\bullet$ may be held by one or more other computations y. CURL u is a capability that designates the network address of $t\bullet$, contains arbitrary x-specific metadata (including closures), and grants to any computation y holding u the right to transmit values to $t\bullet$. When y transmits a value v via CURL u the pair $\langle u, v\rangle$ is injected into t via $t\bullet$. The CURL u specifies which execution site $\langle E, B\rangle$ is to be used for the interpretation of value v. Binding environment B strictly confines the functional capability granted to the mobile code contained in the incoming message $\langle u, v\rangle$. Informally we say that u is a *CURL for* computation x_i, $1 \leq i \leq n$ if x_i holds the ingress point $t\bullet$ denoted by u. If multiple computations $x_1, \ldots, x_n$ hold the ingress point $t\bullet$ there is no guarantee that any x_i will receive a value transmitted via u, much less a specific x_i.

For both security and safety a computation x wields the CURLs it issues to constrain its interactions with other computations and to bound the services it offers. A computation y, holding a CURL for x, can send arbitrary closures to x in the expectation that x will evaluate those closures in the context of some x-specific execution site $\langle E, B\rangle$. At the very least x must, if it can, avoid communicating with malicious or untrusted computations and defensively minimize the functional capability that it exposes to visiting closures. A computation can accumulate communication capability in the form of additional CURLs. For any computation, CURLs conveying additional communication capability are contained in the closure defining the computation, returned as values by functions, or embedded in messages.

3.3.4 Interpretation

How must services (computations) interpret the messages that they receive? The answer is simple: anyway they like. The interpretation of a message depends upon the predilections of the computation to which the message is delivered and the specifics of the CURL by which that message is delivered. Anything less liberal runs the risk of needlessly stifling service diversity and innovation. This does not preclude conventions and practices arising with regard to CURLs or the behaviors of computations but those matters are well outside the scope of the style.

[13] Or more specifically, the capability to inject a message into a transport via a specific ingress point.

3.4 COAST Scenarios

The following brief examples suggest the breadth of the COAST style. The examples are given in the body font of the thesis while commentary on these examples is given in an *italic font*.

Alice operates a COAST-based image processing service. Her clients dispatch computations to her service for processing, enhancing, and analyzing commercial, industrial, and scientific imagery. The execution sites in her server farms are managed by her own COAST computations that generate and distribute service-specific CURLs denoting processing varying across a spectrum of performance and functionality.

Execution sites and CURLs are the two principal mechanisms by which COAST defines highly differential services whose exact characteristics may depend upon time, place, context, and prior history. In essence Alice offers platforms as a service and her clients deploy their applications on the platforms (the execution sites) that best suit their needs. CURLs can be client-specific and have a human-readable textual representation (among others) and can be circulated out of band in email, embedded in web pages, or represented as QR codes.

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

The typical binding environment of a Motile execution site contains hundreds of functions for constructing and manipulating primitive values (numbers, bytes, characters, strings, symbols, and booleans) and data structures (lists, vectors, queues, hash maps, and binding environments, among others) as well as domain-specific APIs. [14]

Using CURLs generated by Alice, Bob transmits Motile closures to her services (computations) for remote evaluation. Bob's images can be embedded as lexical scope bindings in his closures; or the closures can reach back, fetching the images from a data store within Bob's sphere of authority. Reach-back can be tuned for Bob's patterns of use; for example, read-ahead to minimize latency or application-specific compression, as

[14] The `BASELINE` *Motile* binding environment contains a substantial fraction of the functionality found in R5RS Scheme [143]. Execution sites frequently use the `BASELINE` environment as a starting point for their own site-specific binding environment.

the temporary "image server" is spun up by closures that Bob's computations have dispatched back to the data store.

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

Application- and domain-specific libraries written in C or Racket can be integrated into Motile binding environments. Higher-level libraries that contain only Motile functions can be transmitted as binding environments from one agency to another to deliver software as a service. [15]

Carol obtains analytical tools for her imagery from Dave, whose biotechnology company deploys computations for narrowly targeted tissue analyses to COAST sites. Carol dispatches service requests (as computations) to Dave's COAST services. Each of her requests prompts Dave's computations to generate a custom analysis (as a closure or continuation) optimized to meet her request-specific needs and constraints. Included in each of Carol's requests is a non-delegable, "single-use" CURL referencing an ingress point that acts as a firewall between Dave and Carol's execution sites that analyze the imagery.

Motile is a functional language for which closures are first-class objects. Closures can be passed as arguments to functions or returned as values. Using closures to generate closures is a common functional form of software as a service. Carol reduces the computing load on her own COAST sites by relying on Dave to generate custom analytics (as closures) on demand.

A non-delegable CURL can be used only by a specific set of clients (just Dave in this case). A single-use CURL can transmit at most one message; thereafter it is useless. By limiting Dave's ability to communicate Carol reduces her exposure to malware that may infest Dave's execution sites.

Dave deploys his customized analysis to Carol's site via Carol's CURL. As Dave's analysis executes on Carol's COAST infrastructure her execution site prevents Dave's computation from accessing any other confidential imagery or communicating with any computations besides her own. Her COAST-based monitoring and auditing infrastructure tracks the execution of Dave's analysis from beginning to end,

[15] Racket (http://www.racket-lang.org) is a popular Scheme dialect.

ensuring that it does not violate patient privacy regulations.

Both the execution engine and binding environment of an execution site can be customized to include site-specific logging, tracing, or software reference monitors [73]. Carol also ensures that the binding environment she uses when executing the software that Dave generates contains the baseline environment (which is guaranteed to be safe) plus only the domain-specific libraries that Dave's software requires and nothing more. She can even inspect Dave's mobile code to determine, prior to execution, exactly which functions Dave expects to be available in the binding environment. If Carol finds anything suspicious or out of the ordinary she can refuse to execute the mobile code and immediately revoke any CURLs of hers that Dave holds. If Dave is required by contract to sign any mobile code that he transmits to Carol then Dave's mobile code is non-repudiable and Carol can demonstrate to Dave that, barring a bug in Dave's application-specific software generators, his COAST infrastructure may have been attacked by malicious mobile code.

The non-delegable, single-use CURL prevents Dave from sharing the CURL with any other COAST site (non-delegation)[16], and, as it can never be used more than once, neither Dave nor any attacker that infiltrates Dave's infrastructure can ever send more than a single computation to Carol's privacy-sensitive infrastructure. In general, Carol can monitor Dave's computations executing on her infrastructure or her own computations executing elsewhere using automated "report-back" CURLs, selectively wrapped closures, monitoring functions provided at the service execution site, publish/subscribe events originated by a service provider, or third-party monitoring.

In Motile it is trivial to implement a CURL that reports on its own use, so Carol can detect if any agency other than Dave used her CURL to transmit a message. In this manner she can discover if a CURL intended for Dave alone has left his control and is now in the hands of a third party.

3.5 A Notional COAST Infrastructure

The scenarios of Section 3.4 rely heavily on the communication model of COAST (Section 3.2) and a notional COAST infrastructure shown in Figure 3.3.

COAST computations are sequestered within one or more network-accessible address spaces, here A_1, A_2, and A_3. There are six computations, $x_1, \ldots, x_6$. Four of those computations $x_1, \ldots, x_4$ are co-resident

[16] More precisely, the CURL can be shared (transmitted by Dave to another authority) but no other authority but Dave can exercise the communication capability granted by the CURL.

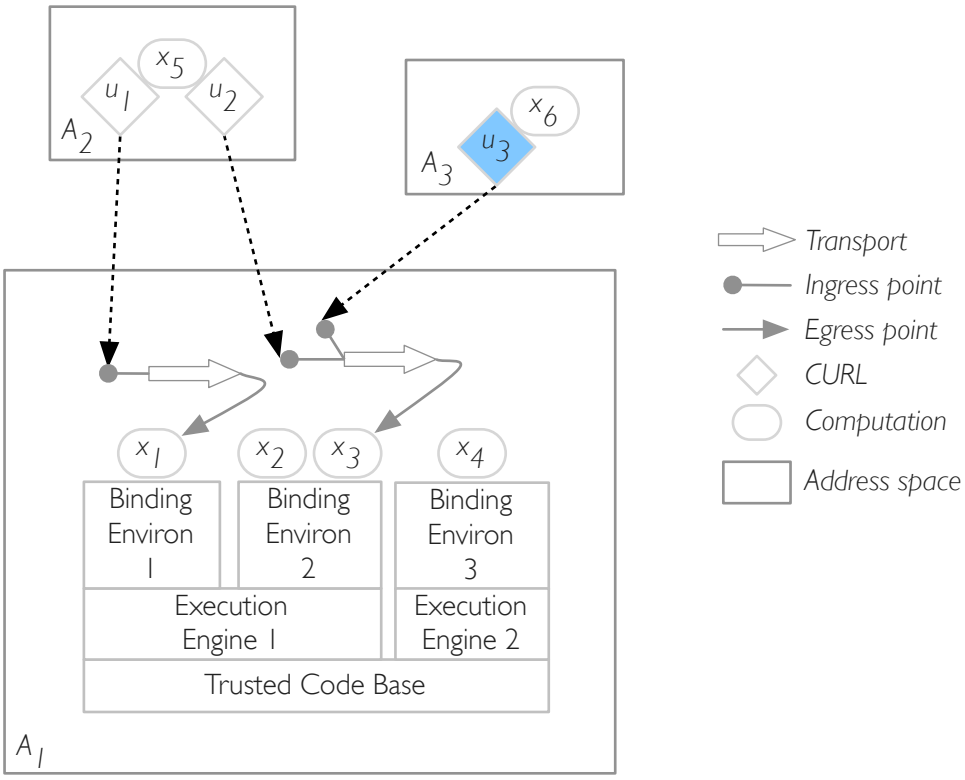

Figure 3.3: Notional sketch of a COAST infrastructure.

in a single address space A_1. The other two computations, x_5 and x_6, reside in address spaces A_2 and A_3 respectively.

The diagram of address space A_1 details its internal structure. A computation x_i within an address space is isolated from its co-resident computations and the only means of interaction, either within an address space or between address spaces, is message passing. Each x_i executes within the confines of an execution site comprising an execution engine and a binding environment. Computations within an address space may have an execution site, execution engine, or binding environment in common. For example, x_1 shares an execution engine with x_2 and x_3, while x_2 and x_3 share an execution site. Sharing reduces overhead but must not compromise isolation. The execution site of computation x_4 contains a different execution engine and binding environment from the execution sites of x_1, x_2, and x_3.

The trusted code base underlying the execution sites of an address space manages the fungible resources of the address space including memory, processor cycles, storage, and network bandwidth. It also enforces the security policy of the address space by regulating the construction and allocation of:

- Encrypted network connections to address spaces elsewhere in the network
- Execution engines and binding environments to execution sites
- Execution sites to visiting computations
- Ingress points to CURLs
- Egress points to computations

and is responsible for routing message traffic to and from the address space.

Computation x_5 in address space A_2 holds two CURLs, u_1 and u_2, both denoting ingress points within address space A_1. Computation x_1 within A_1 holds an egress point whose transport is the transport of the ingress point of CURL u_1. Consequently, computation x_5 in A_2 meets the necessary condition of the COAST style rule **CS3** for transmitting a message to x_1 in A_1: there exists $\langle t\bullet, t\triangleright\rangle$ in A_1 such that x_5 holds a CURL u_1 denoting $t\bullet$ and computation x_1, residing in A_1, holds $t\triangleright$. Holding a CURL u does not guarantee message delivery; it is necessary but not sufficient, as delivery may depend upon complex conditions within the address space in which the ingress point $t\bullet$ appears, or, in the general case, on conditions in multiple address spaces. Likewise, computations x_5 (via CURL u_2) and x_6 (via CURL u_3) both hold the necessary (but not sufficient) capability to message computation x_3.

From here on my technical arguments will rely heavily on the specifics of the reference infrastructure constructed for COAST, *Motile* the mobile code language, and *Island* the peering infrastructure. In the terminology of *Island* an *island* is a network-accessible address space uniquely identified by a public key K, one half of the pair $\langle K, K^{-1}\rangle$. I assume the existence of a *public key infrastructure* (PKI) whereby islands advertise their public identity key K and public signing verification key V.[17] An extremely modest, file-based PKI, suitable for laboratory work, is included in the distribution of *Island*. An authentication infrastructure for COAST is a topic for future investigation; for example, implementing `sayI` [219], a system for internet-scale authentication, as a COAST service is an intriguing possibility.

To simplify deployment and experimentation islands do not rely on fixed IP addresses; an island periodically announces its presence on a network via UDP broadcast and other islands, detecting its

[17] Given $\langle \sigma, V\rangle$ the signature S for plaintext M is given by $S = \mathrm{Sign}_\sigma(M)$. The verification of S is given by $\mathrm{Verify}_V(S) \in \{\mathrm{true}, \mathrm{false}\}$.

beacon, may (or may not) establish an encrypted TCP connection with their network neighbor.[18][19] The *Island* implementation assumes IPv4 but can also be hosted on IPv6 or some mix of the two.

Obviously the COAST communication model and the style are deliberately underspecified; both *Motile* and *Island* are just examples of many possible conforming implementations. That said, the devil is in the details and the specifics of COAST capability semantics, topics addressed in Chapter 4 following. The style impacts the implementation in many ways and at many points — some expected and some surprising. However, these style- and threat-specific considerations are largely implementation-independent. *Any* implementation must address the fine details of functional and communication capability and I speculate that similar mechanisms must appear in every COAST-compliant implementation.

3.6 Capability URLs in Detail

Given the critical role of CURLs in COAST I present a brief tour of their essential semantics and implementation. A CURL u comprises:

- A public key K, the global, unique identifier of an island I
- A unique and persistent 128-bit Globally Unique IDentifier (GUID) [150], the *CURL id*
- An island-unique symbol denoting an I-resident ingress point $t\bullet$, the *ingress id*
- A list of symbols $(s_1\ s_2\ \ldots\ s_n)$, the *path*
- Two binding environments: *policy* and *metadata*. Each can contain closures, continuations, binding environments, and other CURLs. The contents (if any) of *policy* are set by island I. The contents (if any) of *metadata* are set by the computation (residing on I) creating the CURL.

CURLs may be freely transmitted from one island to another. To ensure the integrity of "communication by introduction" it must be effectively impossible to guess, forge, or alter a CURL; consequently, each CURL u contains at least one u-specific cryptographic-grade random number (the CURL id) and is signed by the issuing island's secret signing key σ. CURLs are a first-class, immutable capability in *Motile*; within the confines of a legitimate island, it is impossible for a *Motile* computation to forge a CURL or alter

[18] Obviously this doesn't scale but a single network segment can support several hundred islands simultaneously. In a laboratory setting islands can be reliably deployed, shutdown, and restarted with a minimum of fuss and bother. You should thank me.

[19] The *Island* implementation relies on the ZeroMQ [126] implementation of CurveCP [51] for session encryption and the ZyRE library (https://github.com/zeromq/zyre) for the dynamic discovery of individual islands. A alternate implementation based on MinimaLT [190] could provide public keys as island unique global identifiers, strong encryption, full IP mobility, and seamless integration with DNS while eliminating the scaling constraints.

one surreptitiously. Any island J knowing the signing verification key k_v of I can verify the integrity of any CURL u claiming to be issued by I. In a distributed system cryptography is the only guarantee of capability. Island I must ensure that any message it receives was transmitted via an unmodified I-issued CURL u and any trustworthy island J should insist that any I CURL in its possession is properly verified.

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

3.6.1 Gates

An island's ingress points play a critical defensive role. Attached to each ingress point $t\bullet$ is a $t\bullet$-specific *gate*, a functional contract whose conditions must hold before a message m can transit across the ingress point into the underlying transport t. If, at the time of delivery, m passes the gate then m immediately transits into t; if m fails to pass the gate, it is discarded.

A gate is either a primitive gate or a gate that is the result of a gate combinator. Informally, a gate g comprises two closures, p and a (predicate and action respectively) both closed over the same lexical scope, and whose single argument is the message m transiting the ingress point. The predicate p is always a pure predicate (no imperative side-effects). Actions a come in two flavors: (1) a no-op `(lambda (m) #t)` in which case the gate is *stateless*, and, (2) actions that have imperative side effects, in which case the gate is *stateful*. If the gate predicate `(`p m`)` is true then the gate's action `(`a m`)` is called and the message m transits the ingress point into the underlying transport t. If the gate predicate `(`p m`)` is false then message is discarded and no further action is taken. Primitive gates and gate combinators (in effect a small, embedded domain-specific-language) are defined by trusted *Island* code.[20]

[20] The gate primitives and combinators are also available to *Motile* programs via the binding environments of execution sites.

Primitive Gates

Two simple primitive gates illustrate the distinction between stateless and stateful gates. A *whitelist* gate is a stateless primitive gate that confines transit to those islands for whose public key K, $K \in \{K_1, K_2, \ldots, K_n\}$ holds. The predicate `(p m)` of a whitelist gate extracts the public id K of the transmitting island from message m and tests for its membership in the gate-specific whitelist. The action a of a whitelist gate is always a no-op. A *uses* gate is a stateful primitive gate whose predicate `(p m)` returns true if the current use count is positive and false otherwise. The action `(a m)` of a uses gate decrements the use count by 1. Two other trivial and stateless primitive gates deserve mention: the *always* gate whose predicate always returns true and the *never* gate whose predicate always returns false.

There are three primitive, stateless temporal gates: *before*, *after*, and *during*. Let α denote the current time in epoch milliseconds and t a timestamp for some date and time; then:

- `(gate/before t)` returns a gate where `(p m)` is true if and only if $\alpha < t$
- `(gate/after t)` returns a gate where `(p m)` is true if and only if $\alpha > t$
- `(gate/during t1 t2)` returns a gate where `(p m)` is true if and only if $t_1 \leq \alpha \leq t_2$

For all three gates action a is a no-op since each gate is stateless.[21]

The uses and rate gates are examples of primitive *stateful* gates. A uses gate limits the total number of inbound messages delivered via a particular ingress point. The related *rate* gate limits the rate (messages per second) at which messages may transit a particular ingress point. Unlike stateless gates (which are essentially pure predicates), stateful gates must reside on their island of origin. Obviously a CURL, a read-only structure, cannot reflect changes in imperative state since the same identical CURL u can be used multiple times and and by multiple distinct islands. A stateful gate g is safely shared among multiple ingress points $t\bullet_i$ of a single transport t.[22]

Gates From Pure Predicates

A stateless gate can be constructed from an arbitrary pure predicate p over messages m where, given p, the gate predicate is p and the gate action is a no-op. This reflects Crampton and Huth [49], where

[21] Note also that the argument m for each gate predicate is irrelevant.

[22] Each transport t implements a lock that is shared by all of the ingress and egress points of t. This lock serializes both the transit of messages through the transport and the imperative side-effects of the actions of stateful gates. It is an error to share a stateful gate among ingress points for two or more distinct transports.

predicates over state or context serve as piece-parts for constructing security policy.

Gate Combinators

Gate combinators are higher-order functions `(`$f\ g_1\ \ldots\ g_n$`)` where each argument g_i is either a primitive gate or a gate constructed by a gate combinator. The return value of a gate combinator is itself a gate h comprising a predicate p and an action a. For any message m the call `(`$p\ m$`)` evaluates one or more of the predicates $p_1,\ldots,p_n$ of gates $g_1,\ldots,g_n$ in a combinator-specific manner. If `(`$p\ m$`)` returns true then the call `(`$a\ m$`)` evaluates one or more of the actions $a_1,\ldots,a_n$ in a combinator-specific manner. If `(`$p\ m$`)` returns false then message m is discarded and no further action is taken. The gate h is stateful only if at least one constituent gate g_i is stateful, otherwise it is stateless.

There are four boolean combinators.

`(gate/not` g`)` returns a gate h with predicate `(lambda (m) (not (`p `m)))` where p is the predicate of gate g. If g is stateless then h is stateless; otherwise, h is stateful and the action of h is the action a of g.

`(gate/and` $g_1\ \ldots\ g_n$`)` returns a gate h whose predicate `(`$p\ m$`)` evaluates `(`$p_1\ m$`)`, …, `(`$p_n\ m$`)` in order from left to right. If any `(`$p_i\ m$`)` returns false the remaining `(`$p_{i+1}\ m$`)` … `(`$p_n\ m$`)` are left unevaluated, message m is discarded, and `(`$p\ m$`)` returns false. If all `(`$p_i\ m$`)` return true then `(`$p\ m$`)` returns true and the action `(`$a\ m$`)` of h is evaluated as `(begin (`$a_1\ m$`)` … `(`$a_n\ m$`))`. h is stateful only if some g_i is stateful.

`(gate/or` $g_1\ \ldots\ g_n$`)` returns a gate h whose predicate `(`$p\ m$`)` evaluates `(`$p_1\ m$`)`, …, `(`$p_n\ m$`)` in order from left to right. If any `(`$p_i\ m$`)` returns true the remaining `(`$p_{i+1}\ m$`)` … `(`$p_n\ m$`)` are left unevaluated, `(`$p\ m$`)` returns true, and the action `(`$a_i\ m$`)` is evaluated as the action of h. If all `(`$p_i\ m$`)` return false then `(`$p\ m$`)` returns false and message m is discarded. h is stateful only if some g_i is stateful.

`(gate/xor` $g_1\ \ldots\ g_n$`)` returns a gate h whose predicate `(`$p\ m$`)` evaluates `(`$p_1\ m$`)`, …, `(`$p_n\ m$`)` in order from left to right. There are three cases to consider:

- All `(`$p_i\ m$`)` return false, in which case `(`$p\ m$`)` returns false, no action is taken, and message m is discarded.
- There is exactly one p_i such that `(`$p_i\ m$`)` is true. In this case `(`$p\ m$`)` returns true and `(`$a_i\ m$`)` is evaluated as the action of h

- There exists p_i, p_j $i < j$ such that both $(p_i\ m)$ and $(p_j\ m)$ are true and j is the least index $> i$ such that $(p_j\ m)$ is true. In this case the remaining $(p_{j+1}\ m), \ldots, (p_n\ m)$ are left unevaluated, $(p\ m)$ returns false, no action is taken, and m is discarded.

h is stateful only if some some g_i is stateful.

3.6.2 Implementation of CURLs

Capability URLs, unlike many other access control mechanisms[43], dictate that a service client re-send their access capability on each and every access (in other words, from the perspective of the provider access is stateless), Islands communicate by connections secured with CurveZMQ[23], an implementation of CurveCP [51] [16] over TCP. When a connection is requested by one island to another, they identify themselves one to the other via an exchange of their public elliptic curve keys — these keys serve as long term, globally unique, island identifiers. The two island endpoints create an encrypted session with perfect forward security using short term keys. When the session terminates both endpoints discard their short term keys leaving their encrypted transmissions unreadable, even if the long term, island public keys are compromised.

CurveZMQ, like CurveCP, provides weak non-repudiation. An island I can prove to itself, packet by packet, that island J generated each packet, but it cannot after the fact, prove that assertion to anyone else. Every service request r to island I comprises a CURL u_i and a payload. In this context, island I on receiving a CURL u_I, has at that point in time proved that r originated with island J and that is sufficient to implement non-delegable CURLs (that is, CURLs that may be legitimately used by a finite, enumerated set of islands). In particular, if the use of CURL u_I is restricted to islands $J_1, \ldots, J_m$ then unless $J \in J_1, \ldots, J_m$ then request r will be rejected by I.

Every CURL issued by an island contains an island-unique cryptographic identifier therefore an island can generate multiple distinct CURLs for a single access point and distinguish one from the other for the purposes of service, customization, or security. The embedded cryptographic identifier of a CURL u makes it effectively impossible to guess u and since every request r is fully encrypted no man-in-the-middle attack can expose either the CURL u or the payload of r. Therefore it is impossible to steal a CURL by snooping on the communications between islands.

[23] `http://curvezmq.org`

Each CURL u_I is also signed by I hence any attempt to pass off a forged or tampered CURL u_I in a request as genuine is detected by I and the request is discarded. CURLs u_I issued by an island I meet the fundamental requirements for a capability, namely each u_I is an unforgeable reference. While many different kinds of objects may move between islands including primitive values, closures, binding environments, and persistent data structures, CURLs are the only capability that confers the power to communication. Without a CURL u_I in hand it is impossible for an island J to communicate with island I.

Finally, an island I can protect itself against unwanted communication even with legitimate CURLs u_I by placing time limits, time locks, use limits, or rate limits, or other arbitrary island-specific constraints on the use of all or specific CURLs. A CURL u_I may be limited to a single use and I, at its discretion, grants an additional use by including a fresh CURL u'_I (also single use) in its response to a request r that was issued with u_I. Each and every inter-island transmission is treated afresh and it may well be that an ongoing exchange between service provider and service client requires a fresh CURL for each transmission (in either direction) between provider and client.

3.7 Computations and Identity

Computations are no less critical to COAST than CURLs. I discuss here a vexing problem: the meaning of "identity" for computations; arguing that for security purposes identity is a red herring and that encapsulation and security are better served when individual computations are anonymous.

A *computation* is a single thread of control executing in the context of a an execution site $\langle E, B \rangle$ within the confines of an island (a network-addressable homogeneous memory space). Each island is capable of hosting multiple computations simultaneously. Safe execution of mobile code demands that threads be preemptively scheduled; otherwise an infinite loop in any executing mobile code amounts to a denial-of-service attack. Threads may be implemented natively by the underlying operating system (common in Unix-like systems such as FreeBSD, Linux, and Mac OS X) or as green threads scheduled by a virtual machine, itself driven by a native operating system thread.

Each island, as sketched in Figure 3.3, is identified by a globally unique public key. To what extent then are island-resident computations distinguished one from the other? For guidance we turn to the implementations of modern web servers. When a web client issues an HTTP request to a server the

client has no idea of which specific server-side computations generated the response. Here "ignorance" promotes server-side encapsulation and encourages variation among server implementations. In any case, a client can no more trust the response it receives then it trusts the origin server.

Like arguments can be made for "anonymous" computations on islands where the identity of a computation x residing on island I is unknown both to other islands J and to other computations y co-resident on island I. There are obvious benefits. Anonymity eases evolution, recovery, and adaptation, thus freeing an island to create, reorganize, replace, or destroy computations as it sees fit. The obfuscation of identity may frustrate attacks that target island-specific computations, in particular, computations for critical intra-island services. Security goals may also be served by anonymity; for example, an island may implement a family of computations whose individual instances offer identical service interfaces but where each member emphasizes distinctive security concerns. More generally, an island may implement "aligned" services, each outwardly identical but whose non-functional characteristics vary for the sake of performance, security, testing, monitoring, or analysis.

Identity and security are not synonymous — identity is neither necessary nor sufficient for security. The identity of a mobile code computation is ill-defined and, irrespective of identity, no computation is more trustworthy than the island on which it is executing. At a minimum an executing island I has:

- Deep knowledge of the implementation of f (absent effective cryptographic obfuscation [118]), since I reconstructed f in executable form from f's on-the-wire network representation.
- Complete control of f's execution. I can deploy a f-tailored execution site $\langle E_f, B_f \rangle$ for any closure or continuation f that it receives.

While one can know the proximate point of origin of mobile code, say a closure f transmitted from island Alice to island Bob for execution, tracing the seminal origin of f (using provenance as an intuitive substitute for identity) is another task altogether. For example, Alice may have acquired f from island Carol. Even if Carol signed f, (almost) nothing prevents Alice from reconstructing and (re)signing f with Alice's private signing key before passing f on to others. The transmission of a closure f from one island to another says next to nothing about the identity, point of origin, or provenance of f. All the receiver knows with certainty is the public key (nominal identity) of the island that transmitted f to the receiver.[24]

[24] However, if Bob was expecting to receive f from Carol with Alice acting solely as an intermediary, then Bob can verify (using the public signing key of Carol) that f came from Carol. However, that doesn't prove that Carol was the origin of f as

Identity is a higher-order concept outside the bounds of the COAST style. While identity is important for assigning blame in the event of an error, breach, or disruption, it is not the linchpin of decentralized security. Others might argue that identity is fundamental to trust and to the extent that is true the public key of each individual island is a hook for a rich, nuanced exposition of identity. Nonetheless, there is no single defensible security perimeter in a decentralized system; consequently, in a mobile code regime, trust may be a weak indicator of safety, reliability, integrity, or error-free execution.

Naturally computations are tagged with island-specific unique identifiers necessary to allocate resources, assign execution sites, monitor behavior, log events, and defend island integrity. But this is woefully insufficient for decisions of trust, security, and capability. The criteria for these decisions may be highly island-, domain-, and application-specific and for these reasons *Island* implements a generic mechanism to underpin attribute-based security; specific applications include attribute-based access control (ABAC)[129] and attribute-based encryption (ABE) [36, 193].

The generic mechanism is simple. Affiliated with each island thread is a thread-specific key/value store where the keys are attribute names (given as *Motile* symbols) and the values are arbitrary *Motile* values. This alone is sufficient to implement complex forms of access control. For example, attribute-based gates[25] can be implemented for both ingress and egress points to regulate intra-island communication among co-resident computations (threads). Critical and sensitive functions in binding environments can query the attribute store to determine if the calling thread has sufficient authority for the call — to regulate access to functional capability. More generally, a function may modify its behavior based on the attributes of the calling thread; for example, logging the call as a precaution if the attributes suggest that the caller is noteworthy for any reason, including risk or patterns of behavior.

Attribute-based security generalizes identity, as oftentimes we are not interested in identity per se but attributes or characterizations. Consequently mechanisms that capture "pattern and practice" may be more practical and effective then mechanisms devoted to establishing identity. Furthermore, even "good" islands will occasionally host "bad" computations — an inevitable fact of life in a world of computation exchange. Should higher-order formulations of identity become available, for example attestations of identity backed by a web of trust, then these can be accommodated by additional bespoke attributes in

Carol herself may have acquired f from yet another island, deleted the original signature, and (re)signed f before passing it on to Alice. More generally, the problem of mobile code provenance is a technical challenge that lies well outside the scope of my thesis.

[25] Sections 3.2.2 and 3.6.1.

thread-specific attribute stores. Even so, I conjecture that identity offers limited technical utility and is best reserved for legal, administrative, social, regulatory, and economic structures outside of the technical sphere of COAST.

The threat model of COAST, detailed in Section 3.1, also argues against relying on identity as a definitive indicator of threat or risk. Two recent major security incidents highlight the weaknesses of identity. The data breach at Sony Pictures Entertainment in late November 2014 exfiltrated hundreds of gigabytes of sensitive information.[26] The identity of those responsible for the Sony theft is a subject of considerable speculation and debate[27] but in many respects their identity is irrelevant. Better had the breach been prevented outright, irrespective of the identity of the perpetrator, or at worst, detected far earlier. Prevention, early detection, and risk mitigation rest on behavior — the acquisition, dispersion, and exercise of capability over time — not identity.

Identity is also stolen or forged with astonishing ease. The second incident, the Anthem data breach (with data losses for approximately 80,000,000 Anthem customers) reported in February 2015[28], apparently hinged on the theft of an Anthem system administrator's login identity and password; in other words, it was only one (small) step removed from an insider attack. I speculate that in many attacks knowledge of identity is at best a weak preventative.

As I explore in later chapters, the attributes of computations can, in tandem with the base mechanisms of the COAST style, prevent the acquisition, use, and (re)transmission of unwarranted capability. The anonymity of computations (or at best their weak identity) is a distraction. The threat of insider attack (see section 3.1) demands comprehensive mechanisms, for which capability security is superior to "identity security.'

3.8 Summary

Unlike many architectural styles COAST is deeply security-centric — as befits for a style dedicated to the safe exchange and execution of decentralized live computations. The style is deliberately underspecified, leaving many decisions open to implementations of the style. However, all instantiations of the style must account for computations (the embodiment of services), the separation of computation and

[26] http://deadline.com/2014/12/sony-hack-timeline-any-pascal-the-interview-north-korea-1201325501/
[27] https://www.schneier.com/blog/archives/2014/12/more_data_on_at.html
[28] http://www.latimes.com/business/la-fi-anthem-hack-20150207-story.html

functional capability (in the form of execution sites), the separation of computations and communications in which communication is an explicit capability that may be granted or withdrawn (in the form of CURLs), and the service- and communication-dependent interpretation of the messages exchanged among computations.

The notional infrastructure described in section 3.5 distinguishes between large-grain identity, the public key of an island, and small-grain anonymity, the more-or-less anonymous computations sharing the address space of a single island. Islands are the encapsulation boundary of computations; an island is free to organize its resident computations, both native and visiting, in any manner it sees fit. Any COAST-based collaboration is likely to contain multiple islands operating under two or more distinct spheres of authority, where islands are the smallest fundamental unit of decentralization.

The complexity of the style arises largely from its security demands. The execution of arbitrary mobile code of likely unknown origins is inherently dangerous; adequately confining its execution and interactions is imperative. Although there is ample room for debate about the details of the mechanisms there is little room for discussion of their necessity. The functional capability of computations (executions of visiting mobile code) must be restricted for the sake of island safety. The communication capability of computations (message exchanges among computations) must be restricted for the sake of collaborative integrity. The resource capability of computations, the acquisition and release of fungible resources (such as memory or storage space) or fixed assets (such as sensors, actuators, or databases), must be restricted for the sake of island liveness. Out of these necessities arise four mechanisms of considerable power and generality: communication by introduction, execution engines, global binding environments, and CURLs.

The style succinctly intermixes computation exchange with object-capability security. The **Services** rule defines computation exchange as the sole means by which services are presented and confines interaction among computations to messages that may themselves contain computations (reified as closures and continuations). The **Execution** rule delimits execution sites (execution engines and binding environments) as the sole means by which functional capability is granted to executing computations. The **Messaging** rule confines communication capability to the possession of CURLs (for transmission) and egress points (for reception). Finally, the **Interpretation** rule grants computations the freedom to do "as they please" within the confines of the functional and communication capability at their disposal. In

combination the three rules, **Services**, **Execution** and **Messaging**, delineate the only means by which capability may be transferred from to a computation: messages and the binding environments of execution sites. This guarantees that the embedding of computation exchange in the object-capability model is complete; there are no back channels by which capability can be smuggled in undetected.

Are they the only mechanisms appropriate to the task? Certainly not. For example, type systems or proof carrying code, though not explored here, may contribute to the safe and efficient exchange and execution of mobile code. However, it is difficult to imagine any realistic system implementing the idiom of computation exchange that does not account for the restriction and modulation of functional, communication, and resource capability. With this in mind I present, In Chapter 4, the obligations that the style levies on its implementations and then, in Chapter 5 following, explore how the reference implementation, *Motile*/*Island*, reflects those obligations. Finally, it comes as no surprise that code mobility brings with it unique security concerns for *Motile*/*Island*; those issues are addressed in Chapter 6.

Chapter 4: Obligations of the COAST Style

What obligations must an implementation satisfy to conform to the COAST style? What technical alternatives are available to developers who are building COAST-compliant infrastructure? I address these two questions here. As stated in section 3.3 the COAST style contains four rules:

- **Services**: All services are computations whose sole interactions are the asynchronous messaging of *values*, *closures*, *continuations*, and *binding environments*.
- **Execution**: 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.
- **Messaging**: Computation x can transmit a message to a computation y only if there exists $\langle t\bullet, t\triangleright \rangle$ such that x holds a CURL u denoting $t\bullet$ and y holds $t\triangleright$.
- **Interpretation**: The interpretation of a message delivered to a computation y via CURL u is y- and u-dependent.

Sections 4.1–4.4 explore these obligations rule by rule. Section 4.5 briefly considers two important topics on which the COAST style is silent: spheres of authority and grounds for confinement, both are matters of security policy that fall outside the purview of the style. Finally, section 4.6 is a brief summary of the findings of this chapter.

4.1 Obligations of the COAST Services Rule

The primitives of the **Services** style rule include computations, asynchronous messaging, values, closures, continuations, and binding environments. In Sections 4.1.1–4.1.3 following I appraise the technical interpretations for each of these six elements.

4.1.1 Computations

A *computation* is a single, distinct thread of control; however, the granularity of that thread is left unspecified. There are three possibilities. A computation could be a *process* comprising, at a minimum, a thread of control, an address space, and a protection domain (for example, a protection ring — a feature

common to modern CPU architectures). Alternatively, a computation may be one out of many *kernel threads* all sharing the resources of a single process (a system structure known as multithreading); POSIX threads fall into this category.[1] Distinct computations (in the COAST sense) are not necessarily co-resident in the same process. Finally, a computation may be a *green thread* that is scheduled at application-level (and not by the operating system, as is the case for processes and kernel threads).

As a general rule, the state and instruction costs of kernel scheduling for a process greatly exceeds those of a kernel thread as the latter requires little more than a single set of machine registers (including a program counter and top-of-stack pointer). Typically, multiple green threads are executed in a preemptive round-robin by a single kernel thread that is unaware of the application-level scheduling of the green threads. Consequently, green thread scheduling need not cross a protection domain nor require any interaction with an operating system kernel. In all three cases—processes, kernel threads, and green threads—the style implicitly assumes preemptive, rather than cooperative, scheduling; otherwise, a COAST-based system would be prey to denial-of-service attacks by greedy, malicious, or merely erroneous computations.

At a minimum COAST computations are distributed; hence, absent all guarantees of shared memory, they must interact by message passing. The style requires asynchronous messaging, a pattern that is both well-suited to overlapping computation with communication and from which one can easily implement synchronous communication should it be required. Asynchronous network communication is a common feature of modern operating systems and is used by modern web servers to scale to heavy client loads.

4.1.2 Values

Values are the semantic bricks of programming languages, including both primitive data types such as characters, integers, floating point, rationals, booleans and references, as well as composite data types such as strings, lists, vectors, records, associative maps, sets, and bags. The particulars of values are outside the scope of the style; however, for any particular class or type of value to be communicated from one COAST computation to another, it must have a *transmission representation* that is recognized and accepted by both sender and receiver. The details of transmission representations is an obvious concern of implementations but also lie well outside the scope of the style proper.

[1] POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995).

4.1.3 Closures and Continuations

The transfer of live computations is the fundamental interaction among services within computation exchange. The style calls out three abstract values — closures, continuations, and binding environments — that capture the essence of computation exchange. A closure, comprising a procedure plus its lexical scope binding environment, is in practice a low-cost snapshot of the starting point of a computation.[2] For languages (such as Scheme) where closures are first-class values, closure generators (functions that return closures as values) can generate tailored computations whose lexical scope bindings represent the starting state of the computation. In contrast, a continuation is a snapshot of a computation in mid-execution, comprising a stack representing the dynamic call structure of the computation, a program counter, and the set of values, residing in the heap, that can be transitively reached from the contents of the stack.[3]

4.1.4 Binding Environments

Binding environments, maps of name/value pairs, are a generalization of lexical scope. They can be used to represent simple objects, modules, and records as well as the global (ambient) binding environment that appears in an execution site. Research in the semantics and exploitation of binding environments has a long history in functional programming. One of the earliest examples is Pebble (circa 1984) [27], a language based on the typed lambda calculus, intended as a tool for the "precise description of programming language constructs," in particular, parameterized modules and abstract data types. The language T (circa 1984), a pioneering early dialect of Scheme[4], introduced *locales*, first-class binding environments that served as both the global environment and as modules [200].[5] Felleisen and Friedman (circa 1986) [80] described how modules, with varying degrees of delayed bindings, could be implemented in Scheme as purely syntactic extensions (via macros), illustrating the generality and power of `lambda` and `letrec` (itself a syntactic extension).

[2] The programming language PAL (1968) was the first to offer lexically-scoped first-class functions (closures) [75].

[3] Scheme is purportedly the first programming language to support continuations (though not as a first-class value) by reifying the current continuation as a single argument function via `call-with-current-continuation`. One can regard a closure as a "trivial" continuation whose stack contains only one frame, the closure's calling arguments, and whose program counter is set to the entry point of the procedure of the closure.

[4] An informal and entertaining history of T by Olin Shivers can be found at `http://www.paulgraham.com/thist.html`. Jonathan Rees lists errata in Shivers' accounting here: `http://web.archive.org/web/20070120135524/http://mumble.net/~jar/tproject/`.

[5] I wasn't able to locate an online copy of the 1984 Fourth edition of *The T Manual*. However, the 1988 Fifth edition is available at `http://web.archive.org/web/20060925104715/http://mumble.net/~campbell/t/tman.pdf`

Work by Gelernter, Jagannathan, and London (circa 1987) [102] considers first-class binding environments in a dynamically-scoped Lisp dialect that substitutes this single primitive for multiple distinct language constructions including records, structures, closures, modules, classes, abstract data types, partially closed functions, and environments with temporal behavior such as streams or a representation of an evolving file system directory. Miller and Rozas (circa 1991) [165] address replacing the top-level interaction environment in Scheme with first-class binding environments. In those cases where compiled Scheme code does not manipulate the first-class environments their optimizations eliminate performance penalties. Lee and Friedman (circa 1993) [**Lee_Quasi_Static_Scoping-1993**] introduce *quasi-static scoping* into Scheme to allow variables bound in static scope to be shared across independent scopes (such as modules) or among object instances and detail an implementation that extends well-known compiler representations for static scopes. Jagannathan (circa 1994) [134], in the spirit of Miller and Rozas, strives for a balance between first-class binding environments, common program transformations that rely on static scoping, and constraining the effects of reification and reflection while preserving the semantic power of first-class binding environments. Queinnec and De Roure (circa 1996) [198] make a clear distinction between the orthogonal roles of mapping names to locations and then locations to values to meld first-class environments with the quasi-static variables of Lee and Friedman and improve the efficiency of some reflective operations on binding environments. Without exception, all of the work on binding environments cited above, from 1984 through 1996, assumes execution on a single host—the programs were not even distributed, much less mobile.

Fong (circa 1998) [92] was the first to understand the significance of binding environments for code mobility. In his analysis of Queinnec and De Roure [198], Fong observed that a mobile closure undergoes dynamic linking on arrival at the remote execution host wherein the free variables of the closure are linked to the actual resources administered by the computing environment. In other words, free variables name the resource expectations of mobile code and code mobility is the equivalent of remapping closure evaluation into distinct remote resource (binding) environments. Hence remote evaluation can be understood as the evaluation of a form f in a remote environment `(eval` f u`)` where u denotes the network location of the remote environment to which f is dispatched for evaluation. Code-on-demand is a similar construction `(eval` r b`)` where r denotes the network location of the remote program to be imported for execution and b is a local binding environment.[6]

[6] See [92], page 7–8. Interestingly, Fong fails to mention combining the two constructions to obtain remote execution of

Fong's view of mobile code dovetails with the concept of remove evaluation introduced a decade earlier by Stamos (circa 1986) [228] (of which Fong was apparently unaware). His characterization of name spaces (binding environments) as protection domains is explicitly embraced by COAST. Under COAST *environment sculpting* is the procedure by which a binding environment is tailored for a specific service. Binding environments differ along two dimensions: the names α_i available in the environment and the values v_i bound to those names. In particular distinct services may employ binding environments whose name spaces $\{\alpha_1, \ldots, \alpha_n\}$ are identical but whose actual bindings vary with respect to performance, security, logging, and the details of service semantics. For example, a binding environment B may restrict the range of arguments accepted by a critical binding α/f because the execution site $\langle E, B\rangle$ offers a more restricted level of service than an otherwise equivalent execution site whose binding environment contains a like named, but less restricted binding, α/g.

Finally, service is a particularly amorphous and ill-defined notion. From the COAST perspective a service is a computation that responds to the messages sent to it by other computations. Under COAST there is no effective distinction between service providers and service consumers; all are computations interacting under the obligations of the four COAST style rules. *Provider* and *consumer* are just roles adopted by computations over time and a single computation may switch between the two roles as it sees fit (or exercise both roles in tandem when two or more services collaborate). Beyond the bare bones of interaction service is in the eye of the beholder. For example, a computation that accepts and immediately *discards* any message is a legitimate service if your goal is measuring network loading or the overhead of message transmission and reception between computations.

4.2 Obligations of the COAST Execution Rule

All COAST computations, irrespective of origin, are potentially dangerous. Continuous confinement is one of the means by which providers and consumers ensure their own safety and integrity. As I argued in Section 3.7, computations are anonymous — absent any effective methodology or technique for determining provenance or anything other than proximate origin. The confinement imposed by COAST is a computation-specific interpretive model, the *execution site*, $\langle E, B\rangle$, where E is an execution engine and B is a binding environment.

code-on-demand as (`eval` *r u*).

4.2.1 Late Binding is Necessary

The binding environment B of an execution site defines the functional capability that an COAST service provider grants to visiting mobile code executing within the confines of $\langle E, B\rangle$. Late binding of discretionary functional capability is necessary for mobile code. As Fong observed [92], mobile code is mobile for a reason — it is transmitted to a service provider for execution because the provider offers:

- Fungible assets such as processor cycles, memory, network bandwidth, or storage capacity
- Infrastructure as a service, for example, replicated databases, content-delivery, collaboration aids, design tools, continuous integration, and the like
- Functional capability such as libraries for image recognition, data analytics, or circuit analysis
- Fixed assets such as domain-specific databases, peripheral devices (cameras at locations of interest or data sensors and actuators such as the the pressure transducers, pumps, and values of a waste water treatment plant), or live data streams including streams of synthesized or derived data

Late and dynamic (re)binding is a necessity as visiting mobile code executing under the stewardship of an execution site must bind to the implementations of standard functions made available to visitors, the library interfaces offered by a provider, and the site-, location-, and provider-specific configuration parameters detailing resource limits, metadata, defaults, and other provider idiosyncrasies. Moreover, a single closure may visit and execute within several execution sites $\langle E_1, B_1\rangle, \ldots, \langle E_n, B_n\rangle$ in succession on the same island I, hop from island to island, $I_1, \ldots, I_m$ executing in zero, one, or more execution sites along the way, or exhibit some combination of the two behaviors. As a consequence, mobile code may also abandon bindings as it transits from one island to another. There are many scenarios in which leaving one or more bindings behind is a desirable tactic:

- An identical or comparable binding is available at the next destination; there is no point in wasting network bandwidth dragging along the implementation of a function whose exact replica or functional equivalent is available at your destination.
- The binding in hand now may not be appropriate or effective at your destination; for example, it may contain host- or island-specific customizations that are nonportable, it may be implemented in a processor-specific assembly language, or even worse, it is a blob of host binary instructions.
- The binding may be a proprietary implementation that the provider does not want to share.

- The implementation contains trade secrets (for example, the formulation of Pepsi) that may not leave the confines of the island of the provider.
- The service contract of the provider requires that departing mobile code abandon all of the bindings that it acquired on arrival.
- The binding may be of a provider- or island-specific type that cannot be transferred from one island to another (for example, a large database or a subtree of a file system).
- The binding may be a valuable provider- or island-specific asset that must not be transmitted off-island.
- The binding may be so voluminous that transmitting it off-island is unfeasible.
- The binding may contain cryptographic material that must be confined to the island

4.2.2 Binding Variations Across a Single Sphere of Authority

Within the confines of a single sphere of authority many considerations apply to the construction and deployment of binding environments. There is no expectation, much less a guarantee, that the binding environments $B_1, \ldots, B_n$ deployed within a sphere of authority are identical. Aside from a shared subset of basic bindings, they may differ significantly:

- Variations in binding environments can arise for the sake of backwards compatibility where B^i and B^{i+1} are two successive versions (i and $i+1$) of a binding environment containing a library whose API is evolving. Mobile code that relies on the older API can be directed to an execution site $\langle E, B^i \rangle$ while mobile code written to the latest API can be directed to another execution site $\langle E, B^{i+1} \rangle$
- Two or more binding environments may be functionally identical but differ in terms of performance, logging, tracing, debugging aids, or error responses.
- A binding environment may be severely restricted to confine mobile code to a small set of bindings reflecting a domain-specific language or a narrowly defined service.
- Binding environments may be strictly nested $B_1 \subset B_2 \subset \ldots \subset B_k$ reflecting a set of differentiated service offerings with k distinct levels of service.
- Two binding environments B and $\tilde{B}$ may contain functions f and $\tilde{f}$ respectively where, for example f and $\tilde{f}$ are both functions that translate images from one format to another but $\tilde{f}$ supports a larger set of formats than f.

- In response to an emerging security threat or an actual breach a provider may rapidly deploy variants of its binding environments containing instrumentation or run-time checks that can assist in tracking, slowing, or halting the attack

4.2.3 Binding Variations Within an Island

Within the confines of an island different considerations apply. A mobile code closure f, once it arrives on-island, may be shared among multiple on-island computations $x_1, \ldots, x_n$, each x_i executing in the context of an execution site $\langle E_i, B_i \rangle$. The execution semantics of f may vary by caller x_i as, f will be evaluated in a binding environment determined by x_i. In this case, the safety and security of the mechanics of per-computation evaluation come to the fore. The *Motile* compiler defines four special forms for binding environments: query for a binding within an environment, add a binding to an environment, delete a binding from an environment, and evaluate a sequence of expressions within the context of an environment.[7] For the sake of security and safety the first three (query, add, and delete)are restricted to discourage untrusted mobile code from assembling binding environments containing superfluous or unsafe capability.

In particular, absent specific functional capability it is impossible for a closure executing within an execution site $\langle E, B \rangle$. to capture the instantiation of either the execution engine E or the binding environment B. The *Motile* library function `(environ/capture)` returns the binding environment B of the execution site under which it is executed.[8] Without `environ/capture` the only way that a computation can capture a portion of B as a binding environment is

$$\texttt{(let ((}\alpha_1\ \alpha_1\texttt{)} \cdots \texttt{(}\alpha_m\ \alpha_m\texttt{)) (environ/add environ/null }\alpha_1 \cdots \alpha_m\texttt{))}$$

for $\alpha_1, \ldots, \alpha_m \in B$.[9] This is weak protection at best, for though this construction is unwieldly, it is easily mechanically generated and, if all of the names $\alpha_i \in B$ are known, then B can be duplicated in its entirety. But despite their apparent weakness

. . . it is possible to retain the program reasoning capabilities afforded by lexical scoping, while

[7] See Section 7.5.5 for the details of each.

[8] The `BASELINE` binding environment is usually the starting point for constructing specialized binding environments and contains approximately 340 pure (side-effect free) functions. `environ/capture` is deliberately omitted from `BASELINE`.

[9] There are alternative constructions but they all amount to the same thing.

also gaining expressive power through the use of bindings constructed and shared across disjoint lexical contexts. ([133], Section 2, page 75).

Finally, the special form

$$\texttt{(environ/reflect}\ b\ e_1 \cdots e_m\texttt{)}$$

defined by the *Motile* compiler[10] evaluates the expression `(begin` $e_1 \cdots e_m$`)` in the context of execution site $\langle E, b\rangle$ and returns the value of e_m. In this case, the contents of b are the outcome of evaluation at run-time and consequently b may contain arbitrary bindings whose values were delivered via messaging or as the return value of a function contained in a binding environment elsewhere. These confinement targets, messaging and restricting the propagation of specific functions within binding environments, are addressed in Section 6.2. It should be noted that far more powerful and general functions for manipulating binding environments are made available to highly trusted computations, for example, those computations responsible for remote evaluation and spawning. However, because *Motile* requires that all ambient capability be explicit (in the form of execution sites), that capability can also be confined using the same mechanisms that implement *Motile*/*Island* in the first place.

4.2.4 Why Execution Sites?

Execution sites serve several goals. While their principal role is resource and functional confinement they also encourage flexibility and diversity in service offerings. Minor variations in binding environments allow service providers to easily construct and deploy services targeted toward specific categories of service consumers, in other words, the execution sites $\langle E, B_1\rangle, \ldots, \langle E, B_n\rangle$ may each offer an execution site-specific service variant S_i of some general service S. Alternatively, execution sites $\langle E_1, B\rangle, \ldots, \langle E_n, B\rangle$ may offer the same service S to mobile code written in distinct programming languages $L_1, \ldots, L_n$ where each E_i is capable of executing mobile code whose source language is L_i. More generally, perhaps a small number of distinct execution engines $E_1, \ldots, E_m$ may be sufficient for a larger set of mobile code programming languages.[11] Finally, an execution engine may be a meta-interpreter that "executes" mobile code in a distinct semantic domain. For example, a execution engine, acting as a model checker, theorem

[10] See Section 7.5.5 for a complete discussion of the *Motile* primitives for manipulating binding environments.

[11] This highly speculative notion is suggested, in part, by the mere existence of *enscripten* `http://emscripten.org/`, a transpiler that translates LLVM `http://llvm.org/` byte-code to `asm.js` `http://asmjs.org/`, a strict subset of JavaScript that is used as a low-level, efficient target language for compilers. Enscripten allows programs written in C or C++ to be safely executed within a modern web browser.

prover, or transpiler may verify the message passing behavior of a COAST closure, demonstrate liveness and safety guarantees, predict performance or garbage collection behavior, generate proofs that the mobile code satisfies type-safety constraints, or rewrite the mobile code for the purposes of logging, tracing, or optimization. In other words, by expanding our view of execution engines, services may move well beyond the direct execution of mobile code and into other less traditional domains.[12]

An execution engine may take many forms: a native hardware processor in a resource-constrained embedded device, a sandbox[13], a virtual machine such as a byte-code interpreter, or an abstract interpreter (including a model checker or theorem prover). In general, a sandbox is a security mechanism for isolating and confining untrusted software; however, here I use sandbox to denote confinement of machine binaries. One excellent example of a sandbox is the Google Native Client (NaCl) [267, 268], designed to give "web browser-based applications the computational performance of native applications without compromising safety." NaCl requires a specialized tool chain for generating restricted Intel x86 binary modules that are amenable to analysis and strict confinement. The NaCl sandbox itself comprises an inner sandbox that isolates the execution of the binary modules generated by the tool chain and an outer sandbox that provides a restricted set of services for remote procedure calls and safely interacting with a browser. In some respects this structure resembles the abstraction of an execution site $\langle E, B \rangle$ with the inner sandbox playing the role of the execution engine E and the outer sandbox taking the role of the binding environment B.

The Java Virtual Machine (JVM) [153] is a widely used stack-based, byte-code virtual machine. Lua, a popular scripting language, employs a register-based virtual machine [132]. Scheme 48 [142] also relies upon a stack-based, byte-code, virtual machine, and is notable for the clarity of its architecture and the flexibility of its module system. Here the Scheme 48 virtual machine is the equivalent of an execution engine E and its fine-grain, expressive module system can be used to construct customized binding environments B. The latter feature is heavily used in scsh [217], a Unix shell programming language based on Scheme (and written in Scheme 48) that gives the user full access to Unix programs.[14]

Execution engines may be engineered to meet nonfunctional goals such as memory consumption,

[12] This intriguing possibility falls well outside the scope of this thesis. Any takers?

[13] It is not my intention to provide a comprehensive survey of modern sandbox technology. I briefly review selected examples to indicate the range of choices available to the implementer of a COAST-compliant system.

[14] The most recent source code is available at `https://github.com/scheme/scsh`. Older versions including user documentation can be found at `http://scsh.net/`.

debugging, logging, communication patterns, and auditing or the constraints of distinctive processor architectures. For embedded applications minimizing memory consumption and exploiting processor-specific features are two common concerns. For example, BIT [66] is a Scheme dialect whose byte-code virtual machine is optimized for execution by microcontrollers with as little as 3–4 kB of memory and whose runtime implements a soft real-time garbage collector. PICOBIT [5], another Scheme dialect targeted for memory-constrained embedded applications, uses whole-program analysis (including the run-time library) to eliminate superfluous library functions from the binary image generated by the PICOBIT compiler. The PICOBIT virtual machine is also available in two additional feature-constrained versions that lead to significant space savings. Finally, Vandemme [250] ported BIT to the xCore processor [161], making effective use of the hardware's limited memory and multicore, multithread architecture.

The concept of "execution sites" in mobile code was, as far as I can determine, introduced by Agent Tcl (1998) [117] where it functioned as both a security mechanism (with tailored binding environments for confinement) and as a means of adaptation and integration (by supporting execution engines for languages other than Tcl[15]). If one takes "execution engine" to mean native hardware then the the mobile object language Emerald (1995) [229] established an early precedent—mobility among heterogeneous processor architectures — by translating a single common virtual machine code to the native binary of the host processor.

The style is silent on the implementation of binding environments, the only requirement being that binding environments can be included in the messages exchanged among computations.

4.3 Obligations of the COAST Messaging Rule

The COAST **Messaging** rule enforces the separation of communication from computation[16] — a factoring motivated by common constructions in distributed computing. For example, a "reverse proxy" mediates between clients and background servers by delegating a client request to a background server and returning the server response to the client as though the response originated with the reverse proxy itself. The client is ignorant of the details of the computations (including their number, composition, or location) that the proxy server contacted to satisfy the client request. Since clients can never communi-

[15] To my knowledge, only the Tcl execution engine was ever implemented. Nor is it clear to me, absent inspection of the source code, what hooks, if any, were actually available for multiple, distinct execution engines in Agent Tcl. The implementation of *Motile*/*Island* is certainly no better. Support for execution engines other than Scheme is a subject for future work.

[16] See Section 3.2.

cate with any computation other than the reverse proxy the service provider enjoys considerable latitude in its service implementations. Miller [170] constructs an analogous encapsulation in object-capability security where object p, acting as a proxy for one or more background objects $b_1, \ldots, b_m$ prevents any object o, holding only a reference to p, from discovering the existence of objects $b_1, \ldots, b_m$ much less the details of their construction.

Possession of a CURL u does not guarantee successful communication; possession is just that and nothing more. If x is a computation holding a CURL u then the agency A executing x may enforce a policy preventing communication with the agency that cryptographically signed u or A may be unable to verify the signature, consequently rejecting u as a forgery. A may also cap the total number of messages that x may transmit (by agency, in bulk, or some combination of the two), enforce rate limits on the frequency of message transmission, or even limit the periods of time when x can even attempt to transmit a message — mere possession of u does not guarantee efficacy.

4.3.1 Message Ordering

Islands are obligated to preserve message ordering in two senses. First, the inter-island transmission protocol must preserve message ordering, that is, if two messages, m and $\tilde{m}$ are transmitted by island J to island I, m preceding $\tilde{m}$ in the transmission order, then the transmission protocol between I and J must guarantee that the arrival of m precedes the arrival of $\tilde{m}$ at island I. This ordering guarantee simplifies inter-island coordination and is easily implemented by layering the inter-island protocol over a transport protocol, such as TCP, that guarantees byte-ordered or message-ordered delivery. However, message arrival at an island boundary does not mean that the message will be allowed to pass into the island interior. Even if I permits the exercise of CURL u by J there is no guarantee that the communication will be accepted by the island I, the issuer of CURL u. For example, CURL u may be non-delegable, intended for the sole use of computations residing on island H. Island I can detect the misuse and refuse to communicate. In addition, gates embedded in the CURL u itself as metadata or gates in the ingress point denoted by u may also prevent delivery to the underlying transport.

Second, all messages, irrespective of origin — whether off-island or on-island, transiting an ingress point $t\bullet_u$ must be delivered to the underlying transport t in arrival order. COAST messaging must deal with two cases: intra-island and inter-island transmissions. COAST messaging distinguishes between the

receipt of a message m by an island I and the *delivery* of m via an ingress point $t\bullet_u$ to an underlying I-resident transport t. Given the gates affiliated with $t\bullet_u$ m may not necessarily transit through the ingress point to transport t. To wit, a temporal gate acting as a time lock may permit delivery only during a fixed period of the day and only messages arriving within that window will be passed on to transport t. More generally, for a CURL u issued by island I and denoting ingress point $t\bullet_u$ of I COAST messaging guarantees that if island J transmits two messages m and $\tilde{m}$ to I via u then I will receive m and $\tilde{m}$ in transmission order and either:

- Deliver m and $\tilde{m}$ in transmission order to $t\bullet_u$
- Deliver exactly one of m or $\tilde{m}$ to $t\bullet_u$ or
- Deliver neither of m and $\tilde{m}$ to $t\bullet_u$

A temporal gate can give to rise to all three of these behaviors. In the first case messages m and $\tilde{m}$ arrive within the bounds of the window, in the second case only one of m and $\tilde{m}$ arrive within the bounds of the window while the other arrives either early or late, and in the third case both messages arrive outside the bounds of the window, that is, both are either too early, too late, or one is early and the other late.

However, preservation of arrival order may not be guaranteed by the transport t to which an ingress point $t\bullet$ refers. Under *Island* the default transport is a bankers queue, which is arrival order-preserving, but an island is free to substitute other transports, for example a priority queue that always delivers the highest priority message in the queue and thus may deliver messages in an ordering that is not the arrival order. In short, the ingress order must be arrival order but the egress order is the order dictated by the transport.

Finally, a transport may not guarantee message delivery (much less preserve message arrival order). For example, the "null" transport discards every message given to it and each of its egress points returns `#f` when read. A "lossy" transport may randomly drop messages that transit its ingress points, a "fresh" transport may drop older messages in favor of fresher ones, and a "disordered" transport may reshuffle its buffered messages awaiting outbound transit through an egress point.[17]

4.4 Obligations of the COAST Interpretation Rule

The **Interpretation** rule echoes two critical observations of the web [70]:

[17] All of these transports are legitimate. Whether they are useful is another matter.

- A web server s organizes its namespace (reflected in its URLs) independently of the namespaces of other web servers
- The interpretation of an URL u for web server s is s-dependent[18]

Combined these two laissez-faire principles reduced coordination costs to zero (no "web police" per se to license, sanction, or seize your namespace), encouraged semantic diversity (modulo voluntary conformance to common conventions) at the edges of the network, and fueled an explosion in the number and diversity of web servers and web services. The lesson is clear — any architectural style for decentralized services must maximize freedom at the edges of the network and minimize coordination costs.

Interpretation is the COAST-equivalent of laissez-faire web behavior: COAST service providers do as they please whenever they please. Chaotic? Perhaps, but a more restrictive policy stifles growth and innovation — the chaos of the web, though intimidating, did not hinder its utility. Think of each COAST service as a nanoserver. In keeping with the lessons of the web, each nanoserver is free to interpret incoming messages as it chooses. Besides, if the service is a visiting computation x then, as a practical matter, there is little that an island can do to enforce a standardized interpretation of the messages that x receives.[19] That interpretation is also CURL-dependent as the metadata in the CURL may be intended to parameterize the interpretation. Consequently, for CURLs u and v, both issued by computation x, the same message m can be interpreted differently depending on whether it was transmitted to x via u or v.[20]

However, the flip side of freedom of interpretation grants a COAST service provider absolute discretion in confining visiting mobile code. How so? Because the execution of mobile code by a COAST provider is a form of interpretation, as is analyzing the code, or meta-interpretation of the code in an alternate non-standard domain. More mundane interpretations are also valid. Therefore, under the color of freedom of interpretation a COAST provider may limit communications (for example in number, rate, bandwidth, content, or correspondent), confine, in CURL-dependent ways, the execution sites it grants to mobile code, inspect (and confine) both incoming and outgoing message traffic, log message contents for behavioral analysis, conduct traffic analysis of the communications of visiting mobile code, log call traces to

[18] All we ask is that the interpretation conforms to the specifications of the methods (`GET`, `PUT`, `POST`, ...) [84] supported by u; however, even that weak requirement is frequently violated.

[19] It helps to recall a fundamental rule of security: *Do not mandate what you cannot enforce.*

[20] More generally, the CURL may carry, as mobile code, the *implementation* of the response of x to message m.

global bindings in the global binding environments of execution sites, confine in various ways the semantics of global bindings, augment global bindings (for example, to include non-functional semantics such as execution time or sizes of arguments), generate trust measures derived from mobile code behavior, or incorporate trust measures when making communication or capability decisions.

4.5 What the COAST Style Leaves Unsaid

There are two concepts that that COAST leaves undefined: *spheres of authority* and *grounds for confinement*. The limits of a sphere of authority are not necessarily crisp and well-defined; they may be contextual and change over time. Likewise the grounds for confinement, that is, the rationale for confinement decisions, may be highly nuanced and precise, only roughly approximate, or something in-between.

4.5.1 Spheres of Authority

At a minimum a COAST-compliant implementation is distributed and, in its most general form decentralized — operating under multiple spheres of authority. A *sphere of authority* delineates the limits within which any particular power may be exercised. A sphere of authority may be a civil entity, for example a government, court, or legislature; a legal construction such as a limited liability corporation; a commercial enterprise; a not-for-profit entity such as a charity; a social or religious organization such as a soccer club or a church; a social network of any form; an individual; a family unit whether immediate or extended; an interest group; or a temporary organization erected for a single event over a (comparatively) brief span of time, to name but a few examples. An *island*, the COAST analogue of a server, is a network-addressable computation administered by a sphere of authority. A single sphere of authority may control more than one island.

Given an island I, its identity $k_{p,I}$ is a cryptographic token (a public key) that can be verified by any other island J interacting with I (as can $k_{p,J}$, the identity of J, to I). I implicitly assume the existence of one or more mappings $f : k_p \mapsto a$ where k is an island identity and a a sphere of authority. However, these mappings may be incomplete or erroneous in some respect and two distinct mappings f and g may disagree on the sphere of authority for island k_I (that is, $f(k_I) \neq g(k_I)$, reflecting a fundamental uncertainty — an island identity k_I may not be sufficient to determine with confidence its controlling sphere of authority. Even if two mappings f and g disagree one is not necessarily right and the other wrong as they may each map to different levels or aspects of the same sphere of authority, since spheres

of authority are often hierarchical and even distinct spheres may overlap in complex ways.[21] The disagreements and lacuna of these maps reflect the complexity of identity, the subtle distinctions among the multiple roles that a single identity may play, and our deep contextual understanding of identity and role in our personal, social, public, legal, economic, religious, and professional affairs.

To the extent that computation is an extension or expression of identity, computation exchange may lead to an ecology of service where the exchange of computations is a mirror of the exchanges among individuals, organizations, and enterprises. This view originated with mobile agents [25]. Many use cases portray agents as active representatives of individuals; for example, an agent might book a flight and a hotel room for a tourist or purchase theater tickets and reserve a dinner table at a restaurant. However, agents preserve their identity as they migrate from one locale of execution to another while COAST-based computations are anonymous apart from their proximate island of origin. Identity and the meaning and significance of agency are troublesome issues well outside the scope of this thesis. Even for a well-known and easily identified service such as the local dry cleaners on the corner, it can be challenging to trace that identity back to a sphere of authority, for example the absentee owners of the establishment. Offshore corporations, shell companies, holding companies, interlocking directorates, and limited liability corporations are well-known legal constructions for obscuring a chain of corporate authority. Yet many of us interact daily with multiple organizations whose sphere of authority is obscure or even unknown to us.

From the complexities of agency and the contextual meaning of identity spring difficult questions of trust and capability. These questions are neither addressed nor resolved here. My goal is far simpler; namely,to explore mechanisms that when taken in isolation are comparatively simple and inexpensive to implement but, when combined in various ways, offer significant and constructive protection against erroneous or malicious mobile code. Reliable and accurate estimations of trust will figure in the safe exchange and execution of mobile code but the technology of accumulating, expending, and accounting for trust will come from other work.[22]

[21] Think of questions of jurisdiction or precedence that frequently arise in courts.

[22] Distributed consensus algorithms for cryptocurrencies (such as Bitcoin) may play an important role in trust management.

4.5.2 Grounds for Confinement

On what grounds can a COAST provider make confinement decisions? Watson, in *Network Protocol Design with Machiavellian Robustness* [257], considers robust protocol design in the face of overtly malicious actors, a common problem for public-facing, network-based services. There he argues that the model of *expectations* in which identity plays a role is a far more effective defensive posture than sole reliance on identity which presumes "known parties and their permissions," an assumption that is even less appropriate for COAST than it is for modern web services. Identity is a slippery notion in the modern internet; it is ambiguous (it may be only loosely associated with one or more spheres of authority) and agile (able to present itself anew at will).

Consequently *specific* identity may be far less important than the approximate sphere of authority. Access control, in the sense of a yes or no grant of privilege, becomes far more nuanced — *approximate* decisions replace good or bad decisions. Like a credit card company, a COAST service provider focuses on reducing abuse by visiting mobile code to a tolerable level and minimizing its consequences.[23] Given this far more lax, but far more realistic goal, a COAST service provider is open to alternate sources of information including, but not limited to: historical behavior, current network conditions, indications of abuse reported by other COAST services, time of day, execution patterns over multiple spans of time, patterns of abuse observed elsewhere, and anomalous behavior within the other islands operating under the same sphere of authority.

Systematic exploration of grounds for confinement is also out of scope for my purposes here. However, exploiting alternate sources of information, particularly events elsewhere within the network at multiple scales (local, cloud, across multiple spheres of authority, and so on) is likely a fruitful area of research. COAST is amenable to instrumentation at many levels and the event streams of COAST service providers or post-analysis of island logs may supply valuable data for statistical measures of both safe and threatening behavior.

4.6 Summary

The **Services** rule requires the exchange of closures, continuations, and binding environments for the simple reason that these three well-known objects of functional programming reify the exchange of

[23] Credit card providers tolerate a modest level of fraud and abuse as a cost of doing business, recognizing that it is impossible to eliminate these losses altogether.

live computations: state plus code. Alternatives are possible, for example, objects as implemented in the style of Emerald or Obliq (see section 2.4), but I leave that avenue to others.

There is a long history of functional programming research in binding environments and I draw heavily on that work in formulating the binding semantics of *Motile*. The **Execution** rule demands late binding, otherwise the full value of execution sites is lost and with it half of the adaptabilty of COAST-based systems (the other half comes from the mobile code itself). In addition, as shown in Chapter 10, late binding is instrumental in eliminating *ambient authority*, a crucial step in establishing that *Motile*/*Island* is *capability-safe*.

The **Messaging** rule establishes the role of capability URLs in segregating communicaton from computation and introduces communication by introduction; this too is necessary, but not sufficient, for establishing capability-safety (see Chapter 10). The **Messaging** rule does not levy any requirements for message ordering however, the FIFO order implemented by *Motile*/*Island* is both efficient and sufficient for many purposes. Stronger ordering guarantees can be implemented by application-level protocols if needed.

The **Interpretation** rule reflects a basic web sentiment; provided that a web server observes the rules of HTTP 1.1 it can interpret an URL any way it damn well pleases. COAST adopts a like philosophy.

Finally, the rules interact with one another. For example, COAST offers two distinct views of service: *service as interpretation* where a service provider interprets each service request in a provider-specific manner to generate a response and *service as execution site* $\langle E, B \rangle$ where clients construct client-specific services by spawning a computation executing in the context of $\langle E, B \rangle$. The first category is a consequence of the COAST **Services** and **Interpretation** rules; RESTful web services are a proper subset of this category. The second view is a COAST contribution and arises from the confluence of the COAST **Execution**, **Services**, and **Interpretation** rules. A single computation may offer both forms of service simultaneously where the target CURL and/or message content dictates one rather than the other.

Both computation exchange and object-capability security appear in the COAST style rules. The **Services** rule is a transparent recasting of computation exchange. The model of object-capability is reified in COAST by three constructions: execution sites (the **Execution** rule), and messaging via capability URLs (the **Messaging** rule). The former regulates functional capability and the latter two regulate communica-

tion capability. As the four evaluation studies illustrate (Chapters 8, 9, 10, and 11) manipulating functional and communication capability independently can, in a regime of computation exchange, lead to a variety of useful design patterns for both security and adaptation.

In the two chapters that follow, Chapters 5 and 6, I examine the COAST style in light of the details of *Motile*, the mobile code language. Chapter 5 examines how a programming language can reflect an architectural style, while Chapter 6 explains why *weak mobility* is a sensible choice for a language in which decentralized security is an ongoing concern.

Chapter 5: Motile: Reflecting the Style

With an understanding of the obligations enjoined by the COAST style (Chapter 4) we come to where the rubber meets the road: how are those obligations reflected in *Motile*/*Island*? There are numerous examples of programming languages whose semantics and features are influenced by one or more architectural styles. I begin with a review of frameworks, domain-specific languages, and style-specific languages (sections 5.1–5.3); all three offer insight into the intersection of architectural style and language design.

In particular, those languages intended for mobile-code distributed systems each display a close affinity between their communication model and the mobility of their objects. This holds true for *Motile* as well, whose binding semantics for closures (section 5.5) is driven by the obligations of the COAST **Execution** rule (section 4.2). Further, the *Motile* semantics for message transmission (section 5.6) directly reflect the communication model of COAST (section 3.2). However, the style affords considerable generality, exemplified by the *Motile* implementation of *decentralized promises* (section 5.7).

COAST is silent on the details of how closures become computations. *Motile* follows models established by earlier languages. *Spawning*, employed by Erlang[1], is detailed in section 5.8 and an example, demonstrating the interplay of spawning, messaging and promises, is given in section 5.9. Spawning is a generalization of an earlier model, *remote evaluation*, that was invented by Stamos [226]. Remote evaluation, which has security guarantees not shared by spawning, is also supported by *Motile*; it is detailed in section 5.10.

5.1 Frameworks

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 [240]; Rails,

[1] Termite [104] and Mobit [192], two Scheme-based mobile code languages, also use Erlang as the model for spawning computations.

a RESTful web application framework written in Ruby [206]; and GStreamer, a dataflow style framework written in C for multimedia applications [119].

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 cannot be safely or conveniently 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.

5.2 Domain-Specific Languages (DSLs)

Domain-specific languages (DSLs) are a common alternative to frameworks. While many DSLs are clearly not style-specific—such as SQL for relational database queries, `flex` for lexers[2], `bison` for parsers[3], 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 [145] 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-

[2] `http://flex.sourceforge.net`
[3] `https://www.gnu.org/software/bison`

order functions. Orc illustrates that a rich and robust concurrency semantics requires only a few simple but general primitives combined with higher-order functions.

galsC is a C dialect targeted to event-driven, resource-constrained embedded systems such as networks of sensors and actuators [38, 39]. 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 and galsC are domain-specific (orchestration and embedded systems respectively) but each is also strongly influenced by an architectural style: the former REST, and the latter, event-driven.

5.3 Architectural Style-Specific Languages

The REST architectural style is fertile ground for style-specific languages. Links is a programming language for web applications that generates code for all three tiers of a web application from a single source [46]. 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 [164].

The dataflow architectural style has influenced many domain-specific programming languages, chief among them LabVIEW, a visual dataflow language for laboratory instrumentation, data acquisition, and industrial automation [246]. Distributed dataflow offers several examples: Swift is a scripting language for executing domain-specific applications repeatedly on large collections of file-based data [263]; 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 [175, 176].

Distributed systems, though a broad 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 [22, 137]. Erlang, first developed for telephone switches, is a functional language

modeled after Prolog for highly reliable, soft real-time, distributed systems [7]. 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 [91].

The object-capability language E [167], intended for secure distributed systems is, in spirit, a near neighbor of *Motile*. Deeply influenced by security concerns [168], E is domain-specific in the sense that it is an embodiment of object-capability security but, as Miller, Tulloh and Shapiro observed, it was not the first language of its kind:

> *Our object-capability model is essentially the untyped lambda calculus with applicative-order local side effects and a restricted form of* `eval` *— the model Actors and Scheme are based on. This correspondence of objects, lambda calculus, and capabilities was noticed several times by 1973 [109, 124, 174], and investigated explicitly in [199, 247].*

The communication semantics of E are deeply intertwined with a model of distributed remote method invocation based on *promise pipelining* [166], an optimization that reduces the latency of multiple, chained, remote method invocations.

5.4 Lessons Learned

In each case, be it framework, DSL or architectural style-specific, the details of the style exert an influence in many ways: naming conventions, flow of control, level of abstraction, specialized data objects and communications, and generally relieve the developer of dealing with tedious, error-prone details. Of note though is that frameworks, domain-specific languages, and style-specific languages are not mutually exclusive. One example is the `gen_server` framework of the Erlang Open Telecom Platform [158] which automates many of the painful details of deploying highly reliable and robust servers for enterprise-critical systems. Another is galsC, which reflects both an architectural style, event-driven dataflow, and a domain, resource-constrained embedded systems.

The object-capability security model for E lies at the core of the language however, one can argue that Emerald, Erlang, and Clojure [90] are all object-capability languages, or at worst, very near misses.[4] However, for each the communication model lies at the core of the language and the semantics of remote

[4] Hinging on the degree to which each language restricts ambient authority.

communication often dominate the semantics of the objects of the language. This lesson holds true in *Motile* where mobile code transmission dominates the binding semantics of the language.

5.5 Motile Binding Semantics

When a *Motile* closure is transmitted from one islet to another (both intra- and inter-island) 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? The *Motile* semantics:

- Afford a plain and obvious reading when inspecting *Motile* source code. Other mobile code languages such as MAST [255], allow programmers to pick and choose which free variable bindings to drag along, as do some formal models [17]. These alternative formulations are are more difficult to explain or 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 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 avenue for evolution and innovation.

5.5.1 Alternative Binding Semantics

The binding semantics for *Motile* are but one of many possible alternatives. To better understand the foundational issues of dynamic rebinding for mobile code Bierman et al. [17] present core dynamic rebinding mechanisms as extensions to a typed, call-by-value lambda calculus and define λ_{marsh}, a statically-typed lambda calculus that supports the dynamic rebinding of marshaled values. They also offer a three-axis taxonomy of dynamic binding in programming languages:

1. Resolve variables with respect to a dynamic environment versus static scoping with explicit rebinding.
2. Unify all variables into a single class or cleave them in two, one class static, and the other dynamic.
3. Rebind explicitly per name or alternatively rebind all variables bound within a certain context (for example, a labeled static scope or the contents of a module).

λ_{marsh} employs static scoping with explicit rebinding, has only a single class of variables, and implements rebinding with respect to named contexts. In contrast, *Motile* resolves free variables with respect to a dynamic environment (the binding environment B of an execution site), has two classes of variables: those that are lexically scoped and free variables (those variables not bound in lexical scope), and rebinds all free variables implicitly.

To ease understanding of the semantics the dynamic rebinding of λ_{marsh} occurs only when unmarshaling values and not during normal computation. *Motile*, on the other hand, delays until the last possible instant; all free variables are *resolved at execution time at point of reference* relative to the binding environment B of the execution site of the mobile code.

Bierman et al. point out that the receiver of λ_{marsh} mobile code can securely encapsulate untrusted code[5] but the combination of *Motile*, CURLs, and binding environments can achieve identical effects and with a higher degree of security and confidence (noting that λ_{marsh} is a theoretical model intended for distributed computation and not the security demands of decentralized computation).

In contrast to *Motile*, Obliq, a mobile, object-based language for distributed systems [30] that also supported closure transfer, did not transfer the lexical scope bindings or the free variables of a closure, rather all such references were remote. This preserved *network transparency* — the property that the

[5] See [17] Section 3.6 and Figure 5.

behavior of a closure is independent of its network location. This reduces the burden on a developer as there is no distinction (expect perhaps with respect to performance) between the behavior of a closure λ that executes locally and the remote execution of a migrated copy of λ.

However, COAST, as a style, explicitly rejects network transparency as inappropriate, unrealistic, and unenforceable in a loosely coupled, decentralized system. Even within the confines of a single island a closure λ may have different behaviors when executed in the context of two distinct on-island execution sites $\langle E, B\rangle$ and $\langle E, B'\rangle$. In particular, spheres of authority may, for many reasons, legitimately distinguish themselves, one from the other, by hosting execution sites that offer authority-specific variations in binding environments. The rationales include: competitive and market pressures, variations in customer demand, historical evolution of service requirements, experimental and trial offerings, alternative forms of regulatory compliance, and backwards compatibility. In this light the mix of computation exchange and object-capability security found in the COAST style aims for service variation rather than service uniformity.

5.5.2 Motile is `eval`-Free

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 an arbitrary Motile expression from source strings and then execute it. This restriction guarantees that *Motile*/*Island* can authoritatively determine the complete set of global functions that a visiting closure *may* reference.[6] 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 the authority may moniter the visitor's execution more closely than it might otherwise. In particular, even if a *Motile* closure f contains a reference to a suspicious function α it is always safe to execute the closure if $\alpha/v \notin B$, the binding environment of its execution site, or if v is a harmless stub. It may also be the case that the reference α appears in a closure g, contained within the body of f, where g will be transmitted for execution elsewhere and is never dereferenced by f. For example, f may be a helper or "carrier" closure that distributes helper and service computations across a set of islands.

[6] For similar reasons, no execution site for visiting live computations has access to the *Motile* compiler.

5.5.3 Deconstruction of a Closure

To illustrate *Motile* binding semantics consider the function `palindrome?` shown in Figure 5.1:

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, that is, *not* bound within lexical scope.

The free variables (`sub1`, `string-length`, `>=`, `char=?`, `string-ref`, and `add1`) 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 (that is, those not in lexical scope) are left behind and rebound (redefined) under 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 5.1: A *Motile* predicate to test for palindromes. It returns `#t` (true) if its string argument s is a palindrome and `#f` (false) otherwise.

5.6 Islets and Message Passing

Under *Motile*/*Island* an *island*, a locus of computations executing under the supervision of a sphere of authority, is a single, network-accessible, uniform address space. Each island is identified by a globally unique, island-specific public key [15] that allows any two communicating islands to verify the identity of the other.[7] Each *Motile* islet resides, over its entire lifespan, on the island on which it was created.

By default, an islet is incapable of transmitting or receiving messages. Communication capability must be explicitly granted to an islet by bound variables within lexical scope, by bindings within the bind-

[7] The current implementation uses a 32-byte public key and a 64-byte secret key.

ing environment of its execution site, or both. There are four distinct combinations of communication capability for an islet. It may be: unable to transmit or receive, able to transmit only, able to receive only, or able to both transmit and receive. An islet that does not hold any egress points $t\triangleright$ cannot receive. An islet that does not hold any CURLs cannot transmit. However, possession of an egress point or a CURL alone is insufficient. The binding environment of the islet's execution site must also contain the functional capability to communicate: a function to transmit a message via a CURL, a function to extract a message from a transport t via an egress point $t\triangleright$, or both.[8],[9]

5.6.1 Motile: `curl/send`

The primitive function for message transmission is `(curl/send` u v`)` where u is a CURL and v a *Motile* value. Recall (section 3.6) that a CURL u comprises:

- The public key identifier, k_p, of an island J.
- A denotation of an ingress point, $t\bullet$ residing on island J[10]
- Arbitrary metadata (included by J) to assist J in the routing and interpretation of the messages transmitted via u

Each *Island* message transmission is a *murmur*, a *Motile* structure $\langle k_p, u, v\rangle$ containing:

- k_p, a public key, the identifier of the transmitting island
- u, a CURL, the target of the transmission
- v, a *Motile* value, the payload of the murmur

This message structure follows from the COAST **Interpretation** rule (section 3.3) that grants each islet the right to interpret a message μ in the context of the CURL u employed by the sender. Implementations may provide additional context to aid, expand, or constrict the interpretation.[11] The identity of the transmitter, captured in a murmur, may prove useful in formulating island safety and security policy. On the other hand, it does not present a significant barrier to anonymity as islands can be created and discarded on a

[8] The remainder of this section relies on the notation established in section 3.2.1.

[9] Communication under *Motile*/*Island* is an example of rights amplification, where the possession of at least two distinct capabilities, $c_1, c_2, \ldots, c_n$, confers an additional capability that cannot be obtained with any proper subset of $c_1, c_2, \ldots, c_n$.

[10] The denotation is a island-specific *Motile* symbol that uniquely identifies a J-resident ingress point.

[11] In other words, **Interpretation** sets a floor for the information that is available to a computation for the interpretation of a message.

whim and nothing prevents an island X from acting as a anonymizing proxy on behalf of an island Y that wants to communicate with, but shield its identity from, island I.

Weak identity is a feature of COAST and implicitly acknowledges that nontechnical mechanisms may be required to regulate behavior among multiple spheres of authority; for example, X may have a contractual agreement with I that restricts X's proxying to a limited number of well-known, third-party spheres of authority or that requires X to assume liability for the behavior of any island to which X grants proxy service.

If the transmission is successful then the murmur $\langle k_p, u, v \rangle$ will be delivered to the J-resident ingress point $t\bullet$ denoted by CURL u. However, while the function `curl/send` guarantees reliable delivery of $\langle k_p, u, v \rangle$ to J (in the sense of TCP byte stream delivery) there is no commitment that J will accept the mumur much less inject it into transport t. There are many possible reasons for rejection:

- J, in response to a security alert, rejects all murmurs originating from the sending island I.
- J interdicts the payload of the murmur. For example, it may contain CURLs for islands that are known sources of malware or mobile code in the murmur payload may contain calls to global functions[12] that are forbidden to visiting mobile code.
- J has revoked the ingress point $t\bullet$. In this case any murmur transmitted to J via any CURL u denoting $t\bullet$ will be discarded.
- The circumstances of the delivery of the murmur (time or rate of arrival at J, processor load, available memory, or J-specific domain state, to name just a few) contravenes the contract enforced by CURL u.
- The murmur violates the semantics of the underlying J-resident transport t. For example, t may be a bounded queue that discards overflows or, in the event of high processing loads, t accepts high priority murmurs but refuses low priority murmurs.
- The murmur is accepted by transport t but on extraction from t, via an egress point $t\triangleright$, fails to pass the gates of $t\triangleright$ when extracted from t and is consequently discarded.

Message transmission is always nonblocking and `curl/send` returns immediately.[13] Nonblocking transmission addresses several characteristics fundamental to COAST messaging. Let J be the transmitting

[12] That is, those functions that are resolved by lookup in the binding environment of an execution site.

[13] Each islet is animated by its own green thread. Message transmission is delegated to a dedicated and trusted island green thread that is separate and distinct from the green thread animating any islet calling `curl/send`.

island, I the target island, u the CURL used by J to contact I, and μ the murmur transmitted from J to I. First, message delivery is subject to arbitrary delay. The transit of a murmur μ to I may be delayed, for any one of several reasons: backlog in the transmission queue of J, congestion in the network connection between I and J, or delay in the transit of the murmur through an ingress point of J onto a J-resident transport. I may be offline, halted or crashed, or in transit from one network access point to another (I is executing on a mobile device that is between Wi-Fi hot spots). Second, delivery alone is insufficient. Successful delivery of murmur μ to the J-resident transport t does not guarantee that that μ will pass the gates of an egress point $t\triangleright$. Even if murmur μ passes the I-resident gauntlet of ingress $t\bullet$, transport t, and egress $t\triangleright$ there is no guarantee that the I-resident receiving computation will honor the request encoded in the payload of μ.

There is a fundamental constraint that precludes an error response at this level — communication by introduction forbids target island I from responding to J absent a CURL for J. In general, an island I is under no obligation to report any information of any sort to J much less provide a useful error response. In particular, security policy may prevent I from reporting success or failure to its correspondents or give any indication of the disposition of the transmission from J, including a rationale for rejecting the transmission. Silence can thwart or frustrate attackers who maliciously probe the computational resources, functional capability, and fixed assets of I [257]. Furthermore, island J may not want target island I to reach back. Why? To preserve anonymity, husband limited resources such as network bandwidth or power[14], minimize traffic to fly under the radar of pervasive state-sponsored or corporate network snooping, or defend against probing[15] or attacks by I. Finally, communication responses may be highly domain- and application-specific. In a world dominated by computation exchange effective responses are likely higher-order constructions erected atop simpler and narrower primitives.

5.6.2 Motile: `receive`

To receive a message an islet must hold an egress point $t\triangleright$; in other words, an egress point conveys the capability to receive. Three primitive functions are available for message reception: (`receive` e), (`receive/try` e), and (`receive/wait` e δ) where e is an egress point $t\triangleright$ from which a message is expected:

[14] For example, J may be a battery-powered, embedded sensor that periodically "phones home."
[15] The COAST equivalent of UDP or TCP port knocking.

- `(receive` e`)` implements blocking receive and the calling islet blocks until a murmur arrives. The return value is a murmur μ.
- `(receive/try` e`)` is nonblocking and returns immediately with either a mumur μ or the value `#f`. The latter indicates that no murmur is available from transport t.
- `(receive/wait` e δ`)` blocks at most δ fractional seconds (given as a real number) for a mumur μ to arrive. It either returns μ prior to the expiration of the time period or `#f` if no murmur arrives within the time period. `(receive/wait` e 0.0`)` is equivalent to `(receive/try` e`)`.

The *Island* communication primitives guarantee that for any receiving island I all transmissions from any other island J are delivered to I in transmission order and the bankers queue implementing the default transport preserves that order. However, in general a transport is not required to preserve delivery order; for example, a transport t that implements a priority queue may release a high priority murmur μ_j through an egress point $t\triangleright$ before releasing a lower priority murmur μ_i even though μ_i arrived prior to μ_j. Finally, a single egress point $t\triangleright$ may be shared among multiple co-resident islets, for example, to implement an n-way fanout.

5.7 Decentralized Promises

To bridge the gap between functional programming and asynchronous messaging *Motile* implements *promises*, a proxy object p for a murmur that, at the outset, is undefined because the computation of its value is incomplete. Unlike other languages, *Motile* promises are fully decentralized, that is, a promise held by one authority may be resolved by another authority elsewhere in the network. In addition, the mechanisms: transports, ingress points, egress points, and CURLs, are identical for both the intra- and inter-island resolution of promises.

A computation waiting on a promise p anticipates that eventually a mumur μ (from some computation somewhere) will appear as the resolution of the promise. A promise p comprises two distinct and separable capabilities: r_p, a *resolver*, and s_p, a *settlement*. The resolver r_p grants a computation the power to *resolve* p, that is, to supply a murmur μ as the resolution for the promise. The settlement s_p grants a computation the power to read the *settlement* (resolution) of the promise. These two capabilities, r_p and s_p, are separable, that is, each can be given to separate computations and, in the most general case, each can be given to multiple computations.

A simple promise p can be composed from the communication elements presented in Section 3.3.3. Let $\langle t\bullet, t, t\triangleright\rangle$ be given where transport t is an instance of a a first-in, first-out queue, $t\bullet$ the sole ingress point for t, $t\triangleright$ the sole egress point for t, and

$$t\bullet \longrightarrow t \longrightarrow t\triangleright$$

is the only possible transmission trajectory.[16] Any CURL u referencing ingress point $t\bullet$ of transport t, denoted $\mathscr{C}(t\bullet)$, is a resolver r_p for p, while egress point $t\triangleright$ is the settlement s_p of p. Computation x grants CURL u to any computation y that it selects to resolve the promise and then waits for the the resolution of the promise — the first message transmitted by y via u, to be emitted by settlement $t\triangleright$.

The separability of r_p and s_p is critical. The computation capable of resolving p will likely not be the computation x that created the promise (otherwise x could have computed the resolution itself and not bothered with the promise in the first place). A computation y capable of resolving the promise may or may not be co-resident with the computation x that created the promise and, in either case, object-capability security dictates how y can acquire r_p:

- The closure with which y is spawned contains r_p among its lexical scope bindings
- Functions in the binding environment of the execution site of y may return CURL r_p as a value when called or the value of a binding may contain r_p
- r_p may be embedded in the payload of a murmur that y receives

It must be possible to transmit r_p to y via messaging as this is the only means by which two COAST computations on distinct islands communicate. The settlement s_p may be closely held by x and in the case of a one out of n settlement, such as blind competitive bidding or first to complete, it may be unnecessary, inappropriate, or illegal for a resolving computation to learn the resolutions proffered by competing computations for promise p. Finally, if s_p is closely held by x then x guarantees that it is the only computation capable of providing the resolution of p to others — an essential security property that is impossible to enforce unless r_p and s_p are separable.

These arguments are analogues of those given for separating read/write/update capabilities in other domains and appear at many levels in modern systems. Likewise, the communications model for COAST

[16] See Section 3.2.3.

$\langle t\bullet, t, t\triangleright\rangle$, eschews bidirectional communication where read and write privileges are inseparably bound to a single structure. Communication by introduction is impossible without separating reception (read) from transmission (write); otherwise an introduction would always grant bidirectional communication — violating POLA in many circumstances. With separation computation y can transmit to computation x only because x (or a proxy for x) granted y that capability while, using ingress points, gates and CURLs, x can unilaterally enforce communication contracts on all parties to which x extends this grant. The separation of powers is emblematic of the "self-determination" of computations in a decentralized world. As a matter of security, safety, and resilience every computation must be capable of dictating its terms of interaction. By accepting a grant of write (transmission) capability from x computation y does not immediately place itself at risk since accepting the grant never dictates that y must reciprocate.[17] Finally, x can preemptively revoke a transmission capability (a CURL u) it granted earlier by revoking the ingress point that u references—allowing x to isolate itself from computations that are abusing its grant.

5.7.1 Promises: The Details

The actual implementation of promises differs in detail from the introductory account above. Those details follow.

In the *Motile*/*Island* implementation of promises t_p is a specialized "promise" transport that accepts at most one message m (a murmur) and allows that message m to be read repeatedly, $t_p\bullet$ is an ingress point with a single-use gate for t_p, and $t_p\triangleright$ is an egress point with a whitelist gate for t_p. The triple $\langle t_p\bullet, t_p, t_p\triangleright\rangle$ presents the transmission trajectory

$$t_p\bullet \longrightarrow t_p \longrightarrow t_p\triangleright$$

for any message m that resolves the promise. Any CURL $u_p = \mathscr{C}(t_p\bullet)$ referencing ingress point $t_p\bullet$ of transport t_p is a resolver r_p for promise p and egress point $t_p\triangleright$ is the settlement s_p of p. By default, the whitelist gate of $t_p\triangleright$ is restricted to the computation x that created p.[18] CURL u_p is given by x to those computations y_i capable of computing a resolution to the promise. Since u_p is a single-use CURL (a consequence of the single-use gate attached to $t_p\bullet$) at most one of the computations y_i will resolve p.

[17] The higher-level protocol in which x and y are participating may demand reciprocity but that is a matter of application-specific protocol (hence policy) and is not required by the underlying mechanism.

[18] A whitelist (or blacklist) attached to an ingress or egress point is an *access control policy* that restricts which islets may exercise the capability granted by the ingress (egress) point.

The resolution of p is either a murmur if some computation y_i resolves the promise and `#f` otherwise. Since x is the *only* computation that can read (receive) from $t_p\triangleright$ it alone can proxy the resolution of the promise to any other computation z.[19]

An ingress point $t\bullet$ may be *pinned* (restricted) to accept: only intra-island messages, only inter-island messages, or both intra- and inter-island messages (the default). Any CURL u referencing an ingress point $t\bullet_{\text{intra}}$ pinned for intra-island messaging cannot be used off-island; only islets residing on the same island I as the ingress point may transmit messages via u to $t\bullet_{\text{intra}}$. Similarly, a CURL u referencing an ingress point $t\bullet_{\text{inter}}$ pinned for inter-island messaging cannot be used on-island; only islets residing on a remote island J may transmit messages via u to $t\bullet_{\text{inter}}$.

By default, the settlement of a promise is open to both intra- and inter-island messaging and the resolver is a single-use CURL r. A single-use CURL may be used as the target for a message transmission only once; thereafter any mumur transmitted via r will be immediately discarded by the receiving host.[20] CURL r may be given (via messaging) to multiple islets (both on- and off-island) capable of resolving the promise but only one message transmission via u_p will resolve the promise; any others will be silently rejected. By default, the sole egress point $t_p\triangleright$ is gated with a whitelist whose only member is the islet that created the promise. The whitelist of the egress point $t_p\triangleright$ guarantees that only the islet that created the promise can read the murmur (if any) representing the resolution of the promise. Normally, reading a murmur from a transport via an egress point removes the murmur from the transport; however, for convenience, reading a "promise" transport leaves the murmur in place to be reread if need be.[21]

5.8 Spawning

Spawning is the creation of a fresh islet executing a closure on an island. A minimal set of trusted islets are *spawned* ab initio at island creation. The primitive function `(spawn` B f`)` takes two arguments, a binding environment B and a thunk (a zero-argument closure) f, and returns a fresh islet executing *Motile* function f in the context of execution site $\langle \mathbf{E}, B\rangle$ where $\mathbf{E}$ is the default execution engine of the

[19] Once x passes the resolution μ of the promise to an untrusted computation z, either on- or off-island, it loses control as nothing prevents z from sharing μ (or its payload) with whomever it pleases.

[20] Only the first response (out of $n \geq 0$) will be accepted; all others will be rejected by the ingress point $t_p\bullet$.

[21] In the more general case, where the promise settlement is whitelisted to multiple $n > 1$ islets this feature allows all n islets to `receive/wait` on the settlement simultaneously and to read the settlement without any inter-islet coordination. It is an elegant example of the flexibility and power of the COAST communication model.

birth island.[22]

Remote (inter-island) spawning is implemented via spawn and inter-island message-passing. Spawning is a potent capability; safety and security dictate strict regulation by any island offering the service. *Multiple ingress points referenced by multiple CURLs* is a powerful and versatile design pattern for service flexibility, safety, and security. Let t be a given transport with ingress points $t\bullet_1, \ldots, t\bullet_m$. Each ingress point can support a $t\bullet_i$-specific collection of gates that reflect a distinct access and security policy. For example, one ingress point $t\bullet_i$ has a whitelist gate that restricts its use to a small set of trusted islands and another ingress point $t\bullet_j$ is accessible to co-resident islets only and has a rate gate that severely restricts how frequently it will accept requests. Each ingress point $t\bullet_i$ can be referenced by zero or more CURLs $u_{i,1}, \ldots, u_{i,n}$ where the metadata of any CURL $u_{i,j}$ reflects additional constraints or service-specific parameters. Multiple ingress points $t\bullet_i$, each enforcing its own gate contract, allow an island to offer *highly differentiated services* where each service variation is specific to a CURL/ingress pair $u_{i,j}/t\bullet_i$.

Islands can create ingress points (and distribute CURLS that reference them) whenever needed to address fluctuations in demand or usage or shifts in policy. Islands may define domain- and island-specific gates for ingress points that reflect their relevant points of service inflection. Further, as an ingress point $t\bullet$ can be revoked at any time, any CURLs referencing $t\bullet$ are also subject to immediate revocation, whereby an island can quickly defend against abuse, erroneous use, or accidental misuse of its CURLs. In other words, highly differentiated services not only increase service variety and specificity but also contribute to service resilience since an island can modify or terminate access to a single member of a service family while leaving its other members undisturbed.

Figure 5.2 is a conceptual sketch of an island's implementation of spawning using the design pattern of CURL/ingress pairs. Under *Island* a single trusted islet, the *hatchery* (the only islet that holds the sole egress point for the transport t that enqueues spawn requests), responds to spawn requests that either arise on-island or that arrive from remote islands elsewhere in the network. The hatchery awaits incoming murmurs of the form

$$\langle k_p, s_i, (\texttt{"SPAWN"}\ f)\rangle$$

where k_p is the public key of the transmitting island (that is, the island requesting a spawn), s_i is one of

[22] The general form of spawn is (spawn E Bf a_1 $\cdots$ a_n) where E is an execution engine, B a binding environment, f an n-argument closure, and $a_1, \ldots, a_n$ the calling arguments of f. As of this writing the only execution engine implemented is a threaded Scheme interpreter.

the four CURLs $s_1, \ldots, s_4$, and the payload, (`"SPAWN"` f), is a request to spawn an islet executing closure f. Four islands, U, V, W and X, each hold a CURL that conveys the capability to spawn computations on island J. Islands U and V both hold a copy of CURL s_1 which references ingress point $t\bullet_2$. Island W holds CURL s_2 which references ingress point $t\bullet_3$ while island V holds CURL s_3 which references ingress point $t\bullet_4$. Finally, island J ingress point $t\bullet_1$ is for on-island use only and consequently, CURL s_4 referencing $t\bullet_1$ is restricted to islets residing on J.[23]

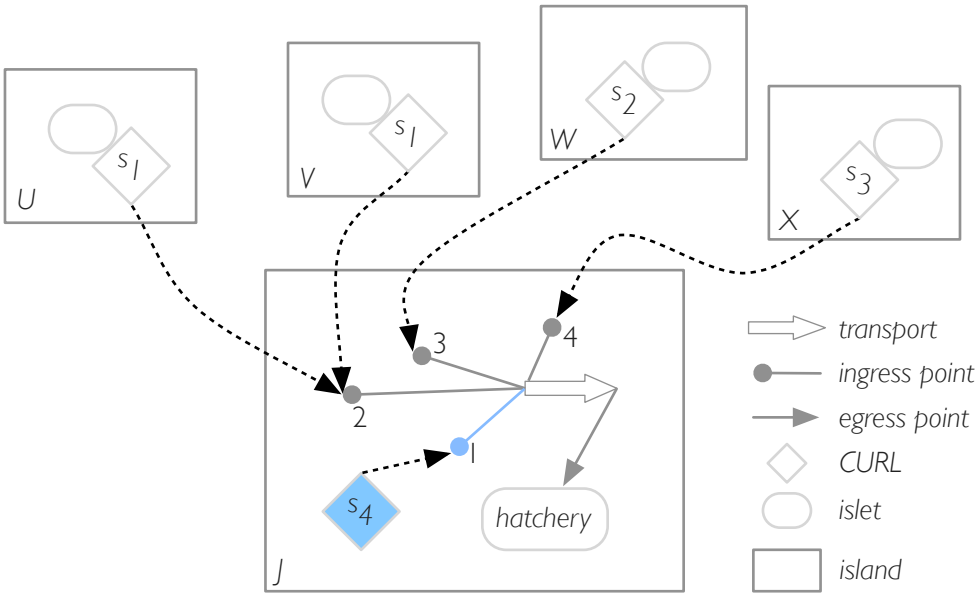

Figure 5.2: The design pattern of multiple CURL/ingress pairs implements variants of island J's spawning service.

In fact, the *Island* peering infrastructure is itself structured as COAST-based system, that is, the internals of an island are a COAST system in miniature comprising a set of collaborative "nanoservers" that interact solely by exchanging messages containing values, closures, continuations, and binding environments (**Services**), and where each nanoserver executes within a confined binding environment (**Execution**). The nanoservers intercommunicate via CURLs (**Messaging**) and for each nanoserver message interpretation is both CURL- and nanoserver-dependent (**Interpretation**). In this manner, *Island* is a rare example of a "recursive" style in which the style itself (COAST) is used to implement an instantiation of the style.

Figure 5.3 is the implementation `curl/spawn`, the client-side of islet spawning. Argument $u@$ is a

[23] CURL s_4 can never be shipped off-island. Any attempt will be detected as the murmur payload is prepared for transit and aborted prior to transmission.

```
(define (curl/spawn u@ f)                                    1
  (curl/send u@ (list "SPAWN" f)))                           2
```

Figure 5.3: The client implementation of spawning.

CURL denoting denoting a transmission trajectory (either on- or off-island) for a spawning service and f is the *Motile* closure to be spawned (line 1). The body of `curl/spawn` (line 2) is trivial — the request `("SPAWN" f)` is delivered to the target service as murmur $\langle k_p, u@, \texttt{("SPAWN" } f\texttt{)}\rangle$ where k_p is the public key of the transmitting island.

Each CURL/ingress pair $s_i/t\bullet_j$ describes and designates, by way of ingress gates and CURL-specific metadata, a distinct variant of the generic spawning service. Here each ingress point enforces access policy for the spawning service while the CURL metadata details resource allocation bounds (such as memory, lifespan, and file storage) for an islet, the characteristics of the execution site $\langle E, B\rangle$ of an islet including the contents of binding environment B, and any other island-specific parameters relevant to islet construction and deployment. The pair $s_1/t\bullet_2$ is the capability for a restricted form of remote spawning with sharp limits on memory, processor cycles, frequency of use, and functional capability[24] and is intended for use by islands that have no prior history with J. Islands W and X are paying customers of J but have purchased different levels of service. The pair $s_2/t\bullet_3$ ensures that the islets spawned by W are allocated fewer processor cycles and less memory than the islets spawned by X under the capability granted by pair $s_3/t\bullet_4$. In addition, only island W can exercise the capability of $s_2/t\bullet_3$ as $t\bullet_3$ contains a whitelist gate that restricts access to W alone. On the other hand, island X is one of a small set of islands that banded together to negotiate a bulk discount on spawning services hosted by J.[25] Consequently, the ingress point of pair $s_3/t\bullet_4$ contains, in addition to other gates, a whitelist enumerating the member islands eligible for a discount.

Service variation within the confines of a single service follows from the COAST style rules. The COAST **Messaging** rule (section 3.3) separates computation from communication. Islands encapsulate computations and an island has no obligation (within the style) to reveal the topology of its internal communications, the constitution of its islets within its borders, or their numbers. To the extent that communications are differentiated by CURLs and ingress points the services they denote can be charac-

[24] The latter by sculpting the contents of the binding environment B of the execution site $\langle E, B\rangle$ in which the spawned islets will execute.

[25] Or more precisely, the spheres of authority governing the members of that set of islands.

terized, one from the other, along a broad spectrum of policy, functionality, and performance. The COAST **Interpretation** rule (section 3.3) allows the evaluation of each murmur μ to vary with both the CURL u that effected the delivery of μ and the islet to which μ was routed. The construction of murmurs

$$\langle k_p, u, v \rangle$$

and the trajectory of communication ensures that the receiving islet can base service decisions on point of origin (k_p), the ingress point of the trajectory (denoted within CURL u), the egress point of the trajectory (known to the receiving islet), the metadata of CURL u, and the content v of the payload.

Access and security policy for spawning arises from the CURL/ingress pairs, $u/t\bullet$ (that grant the capability to communicate with the *hatchery*) and the *hatchery*'s interpretation of those pairs $u/t\bullet$. While an island can have a custom *hatchery* implementation, one can write a generic *hatchery* that knows nothing of the details of the pairs $u/t\bullet$ or the policy which each pair represents. A generic hatchery is parameterized by two binding environments, $\mathscr{E}$ and $\mathscr{B}$: the first a binding environment of execution engines $\mathscr{E} = \alpha_1 : E_1, \ldots, \alpha_m : E_m$ and the second a binding environment of binding environments $\mathscr{B} = \beta_1 : B_1, \ldots, \beta_n : B_n$. The engines $E_1, \ldots, E_m$ and binding environments $B_1, \ldots, B_n$ constitute the engines and binding environments that may appear in the execution sites of spawned islets. Each α_i (β_j) is an island-specific name for the respective E_i (B_j).[26]

Each CURL u of a pair $u/t\bullet$ can contain arbitrary metadata (included in u by the island J that created u) and in particular, a u-specific *Motile* closure $\texttt{spawn}_u$ with five arguments:

- k_p, an island public key
- u, a CURL
- f, a thunk
- $\mathscr{E}$, a binding environment of execution engines
- $\mathscr{B}$, a binding environment of binding environments

In essence $\texttt{spawn}_u$ is a u-specific wrapper around the primitive function `(spawn E B f)`. The call

$$(\texttt{spawn}_u\ k_p\ u\ f\ \mathscr{E}\ \mathscr{B})$$

[26] For example, in *Motile* `BASELINE` is the binding environment that comprises a large number of well-known, side-effect free, R7RS Scheme functions.

is itself executed within the confines of a restricted execution site $\langle E_s, B_s \rangle$ tailored by J to meet the functional and resource demands of the family of $\texttt{spawn}_u$ functions generated by J.[27] Argument k_p identifies the client island, u is the CURL by which the client island transmitted the spawn request, and f is the thunk to be executed by the spawned islet. Each variant, $\texttt{spawn}_u$, validates the spawn request, selects (and, if need be, customizes with respect to a u-specific security and safety policy) an execution engine $\alpha_i : E_i$ from $\mathscr{E}$, a binding environment $\beta_j : B_j$ from $\mathscr{B}$, and, if appropriate, finally calls $(\texttt{spawn}\ E_i\ B_j\ f)$ to create a new islet executing closure f in the context of execution site $\langle E_i, B_j \rangle$. By construction $\texttt{spawn}_u$ has all the information it needs to validate, parameterize, and execute a spawn request.

The generic implementation of spawning is an example of *fire and forget*, a model of stateless service in which the functions implementing the service are delegated to CURLs as mobile code. In other words, an island J constructs a CURL u whose metadata implements a service variant and then "forgets" the implementation. This is another form of highly differentiated services in which an island can craft on-demand a service variant deliver the implementation of that variant as mobile code embedded in a CURL u and, when presented with a service request transmitted via u, resurrect the custom service in its entirety for the sake of its client.

Here a trade is being made: larger, more complex CURLs that require more bandwidth to transmit and processor cycles to digest in favor of a higher degree of service variance and customization, less server-side state, and perhaps, less server-side complexity. Finally, fire and forget is safe; since all CURLs u are cryotographically signed by the issuing island it is impossible for any other island to forge or modify u and its metadata without the issuing island detecting that u is illegitimate. One possible security risk is that the CURL-embedded service functions may expose sensitive details of the service implementation. If this is a concern then the relevant *Motile* mobile code, service-related metadata, and parameters can be encrypted using a CURL- or service family-specific secret key. While this burdens the serving island with additional state the memory requirements for storing even a large number of secret keys may be less than retaining the implementations of an equivalent number of service variants.

[27] For example, binding environment B_s must contain a binding (name/value pair) $\texttt{spawn}/\lambda$ where λ is the closure that is the implementation of spawn.

```
(let*                                                                          1
    ((reply (promise/new)) ; Create a promise.                                 2
     (reply/resolver (promise/resolver reply))) ; The CURL of the promise.     3
                                                                               4
  ;;  Return #t if string s is a palindrome and #f otherwise.                 5
  (define (palindrome? s)                                                      6
    (let loop                                                                  7
      ((left 0)                                                                8
       (right (sub1 (string-length s))))                                       9
      (or                                                                      10
       (>= left right)                                                         11
       (and                                                                    12
        (char=? (string-ref s left)                                            13
                (string-ref s right))                                          14
        (loop (add1 left)                                                      15
              (sub1 right))))))                                                16
                                                                               17
  ;; Transmits a list of palindromes, computed at the                         18
  ;; dictionary service, back to the client via the resolver                  19
  ;; of the promise.                                                          20
  (define (palindromes)                                                        21
    (curl/send reply/resolver (words/filter palindrome?)))                     22
                                                                               23
  ;; Send the palindromes thunk to the dictionary service for execution.      24
  (curl/spawn J@ palindromes)                                                  25
  ;; Block until the promise is resolved, returning the list of palindromes. 26
  (promise/block reply))                                                       27
```

Figure 5.4: Spawning an islet to remotely compute palindromes at a dictionary service. The definition of `palindrome?`, lines 6–16, is taken from Figure 5.1.

5.9 Example: A Client Computation for Palindromes

With messaging, promises, and spawns in hand we can construct a client-defined service. Figure 5.4 illustrates how a COAST client constructs a client-defined service from the primitive functions `curl/send`, `promise/new`, and `curl/spawn`, and the functions in the binding environment of a provider-defined execution site.

The service provider, island J, implements a dictionary service (execution site) and the binding environment of that service provides a higher-order function (`words/filter` p) that applies predicate p to each word in J's dictionary and returns a list of all words α such that (p α) is `#t`. Regrettably, service provider J neglected to include a palindrome filter in its collection of dictionary functions and client island I has constructed a *Motile* closure `palindromes`[28] for execution by J that computes a set of palin-

[28] Specifically, a *thunk*—a zero-argument closure.

dromes and transmits them back to I to remedy the oversight.

First, `palindromes` (lines 17–18) must contain a CURL u_I, the target for the transmission of the result back to I. For this I allocates a promise `reply` (line 2) and extracts the resolver of the promise as binding `reply/resolver` (line 3). Next I defines a predicate, `(palindrome? s)` (lines 6–16), that returns `#t` if string s is a palindrome and `#f` otherwise. It steps two indexes, `left` and `right` (lines 8–9), starting from the first and last characters of the string respectively, to the opposite end of string; comparing the characters at the two positions at each step. If the indexes ever meet or cross then s is a palindrome and `#t` is returned. If the characters fail to match at any step then s is not a palindrome and `#f` is returned.[29] Note that the implementation of `palindrome?` (lines 6–16) depends entirely on well-known primitive functions of R7RS Scheme. Those primitives are defined in the `BASELINE` binding environment of *Motile*, a starting point for the binding environments of many execution sites.

The last constructive step, the thunk itself, is defined in lines 21–22. Variables `reply/resolver` and `palindrome?` (line 22) are bound within the lexical scope of thunk `palindromes` and when transmitted from island I to island J for execution it will carry those bindings along with it to J. The thunk itself is trivial. It executes a higher-order, domain-specific filter (`words/filter`) implemented by J and transmits the result (`curl/send`) back to I via the promise resolver (`reply/resolver`) generated by I (line 22).

Finally, island I spawns (`curl/spawn`) an execution of thunk `palindromes` on island J (line 25). Variable `J@` (line 25) is bound to a CURL generated by J for J's dictionary service (execution site).[30] The client islet on island I executing lines 1–27 blocks (`promise/block` at line 27) waiting for the promise (`reply` at line 2) to be resolved by the spawn dispatched to island J.

5.10 Remote Evaluation

Spawning (section 5.8) generalizes, but does not replace, remote evaluation. Remote evaluation is a useful restriction and a peer may be inclined to accept a remote evaluation over a spawn — a remote evaluation is a one-shot, finite computation while a spawn may (legitimately) execute indefinitely. I briefly describe the *Motile*-level primitives for remote evaluation and recast the example of Figure 5.4 using these primitives. These primitives automate, to various degrees, work done by the client in Figure 5.4

[29] The definition of the predicate function `palindrome?` is taken from Figure 5.1, in section 5.5.3 where it is used to illustrate the difference between variables bound in lexical scope versus free variables.

[30] For the sake of brevity and ease of presentation I omit the details of how island I came to possess that CURL. For now, assume that `J@` is bound in some outer lexical scope enclosing lines 1–27.

```
(let*                                                                    1
    ((reply (promise/new))                                               2
     (reply/resolver (promise/resolver reply)))                          3
                                                                         4
  (curl/remote*                                                          5
   J@                                      ; u@                        6
   (lambda () (words/filter palindrome?)) ; f                          7
   reply/resolver)                         ; r@                        8
                                                                         9
  (promise/block reply))                                                 10
```

Figure 5.5: Using `curl/remote*` to compute palindromes at a dictionary service. Assume that `palindrome?` is defined in an outer lexical scope (see Figure 5.1, section 5.5.3).

where the client: constructs a promise (lines 2–3), transmits the resolver for that promise to the service provider (lines 17–18 and 20), and, at some point waits for the resolution of the promise or alternatively, repeatedly polls until the promise is resolved (line 21).

There are three related *Motile* primitives for remote evaluation: `curl/remote*`, `curl/remote`, and `curl/remote/block`.

5.10.1 `curl/remote*`

When called by island I (`curl/remote*` $u@$ f $r@$) transmits thunk f and CURL $r@$ to island J via CURL $u@$ for evaluation. Island J will execute f and transmit its return value via CURL $r@$. There is no requirement that $r@$ be a CURL for I nor that $r@$ be the resolver (CURL) of a promise.[31] Island J can provide any CURL $r@$ it finds appropriate, including a CURL for an island $K \neq J$ or one that is not single-use (for example, J may dispatch an identical remote evaluation to multiple islands — all with CURL $r@$, accept an odd number $n > 1$ of return values, and use majority voting to select a result. `curl/remote*` always returns immediately. Figure 5.5 illustrates how a client implements the palindrome service of Figure 5.4 using `curl/remote*`.

5.10.2 `curl/remote`

(`curl/remote` $u@$ f) relieves the caller of the burden of generating a promise and returns a fresh promise p as its value. Again, as before, $@u$ is the CURL for the service provider, the island J that will

[31] Though J can easily determine if $r@$ is (or is not) a CURL for I it cannot determine if $r@$ is a resolver. As a matter of security policy island J may refuse remote evaluations for which CURL $r@$ is not a CURL for the island I that transmitted the remote evaluation to J.

```
;; Block on the promise returned by curl/remote.                          1
;; The resolution of that promise will be a list of palindromes.          2
(promise/block                                                            3
 (curl/remote                                                             4
  J@                                        ; u@                          5
  (lambda () (words/filter palindrome?))))  ; f                           6
```

Figure 5.6: Using `curl/remote` to compute palindromes at a dictionary service. Assume that `palindrome?` is defined in an outer lexical scope (see Figure 5.1, section 5.5.3).

```
;; curl/remote/block generates a promise and blocks until it is resolved.  1
;; Here the resolution will be a list of palindromes.                      2
(curl/remote/block                                                         3
 J@                                        ; u@                          4
 (lambda () (words/filter palindrome?))))  ; f                           5
```

Figure 5.7: Using `curl/remote/block` to compute palindromes at a dictionary service. Assume that `palindrome?` is defined in an outer lexical scope (see Figure 5.1, section 5.5.3).

execute thunk f on behalf of island I. The reimplementation of Figure 5.4 using `curl/remote` is given in Figure 5.6.

5.10.3 `curl/remote/block`

Finally (`curl/remote/block` u@ f) blocks until the return value x of f is received. It returns x as the value of the call. The reimplementation of Figure 5.4 using `curl/remote` is given in Figure 5.7.

5.10.4 Implementation

Remote evaluation is implemented by *Island* as a spawn where the islet executes with tight resource limits[32] and in an execution site $\langle E, B \rangle$ where all capability for intra- and inter-island communication has been stripped from the binding environment B.[33] Even if the closure for the remote evaluation carries one or more CURLs among its lexical scope bindings it cannot exploit those CURLs in any effective way.

Transmitting the return value of the evaluation via the CURL accompanying the thunk of the remote evaluation is the responsibility of the island consequently, it is impossible for a remote evaluation to abuse communication capability and, barring an "evil island," there is no question of the integrity of the return value of the remote evaluation. As remote evaluations can neither send nor receive messages they

[32] Those resource limits, such as memory, processor cycles, total lifespan and file space, are dictated by island policy and vary from island to island and request to request.

[33] In almost all cases the functions for generating CURLs are also expunged.

are more likely to be narrowly defined with a single purpose, hence may be easier to further confine, track, and audit.

A remote evaluation, with its smaller attack surface, presents less risk to both provider and consumer than a spawn. Since a remote evaluation ρ arriving from another island neither holds egress points nor receives them from other co-resident computations it is also impossible to generate an effective CURL that allows any computation, either on- or off-island, to communicate with ρ.

Alternatively, let λ be a thunk on island J that contains ingress points, egress points, or CURLs in its lexical scope bindings. For *all* remote evaluations whether they originate on- or off-island the binding environment of the remote evaluation is *stripped* of any function that can: create a CURL given an access point, read a murmur from an egress point, or transmit a message via a CURL.

If λ is the thunk for a "remote" evaluation ρ on J that is initiated by a computation x *residing* on J then, even if ρ could somehow generate a CURL $u_{t\bullet}$ for an ingress point $t\bullet$ bound in the lexical scope of λ, then $u_{t\bullet}$ might appear as the return value of the remote evaluation ρ but ρ itself is incapable of exploiting $u_{t\bullet}$ in any way.

In particular, if λ contains both $t\bullet$ and $t\triangleright$ for a transport t it cannot smuggle out a CURL $u_{t\bullet}$ that would allow any computation anywhere to transmit a message to ρ. Without access to the primitive `receive` ρ cannot read a murmur from any egress point $t\triangleright$ that may be present in the lexical bindings of λ; even if a computation could somehow transmit a murmur via transport t, ρ has no means by which to read it. In other words, ρ is held incommunicado, and the transmission of its return value is at the discretion of the hosting island J, not ρ.

Finally, both `curl/remote` and `curl/remote/block` are easily implemented with `curl/remote*`. Figure 5.8 illustrates their implementations.

5.11 Summary

Like other languages intended for mobile-code distributed systems, communication and mobility are intertwined. However, under COAST, the influence extends further— communication, mobility, *and* security are braided together and each of the four COAST rules has an important role.

The influence of COAST is played out in the palindrome examples of Figures 5.4–5.7 which illustrate

```
(define (curl/remote u@ f)                                          1
  (let* ((p  (promise/new))                                         2
         (r@ (promise/resolver p)))                                 3
    (curl/remote* u@ f r@)                                          4
    p))                                                             5

(define (curl/remote/block u@ f)                                    1
  (let* ((p  (promise/new))                                         2
         (r@ (promise/resolver p)))                                 3
    (curl/remote* u@ f r@)                                          4
    (promise/block p)                                               5
```

Figure 5.8: Implementations of `curl/remote` and `curl/remote/block` using `curl/remote*`.

the roles of each of the four COAST rules (section 3.3): **Services**, **Execution**, **Messaging**, and **Interpretation** and their realization in *Motile*/*Island*. Under *Island*, *spawning*, a basic interaction for systems founded on computation exchange, is itself a COAST service. The *hatchery* (Figure 5.2) is a trusted computation that is a recipient of values and closures (hence a COAST **Service**). It executes within the confines of a tailored execution site (COAST **Execution**), whose communication privileges are constrained (conforming to COAST **Messaging**). The hatchery's interpretation of the messages it receives are both CURL- and *hatchery*-dependent (COAST **Interpretation**).

The style is silent on the execution semantics of computations, leaving that to the specifics of the computation's execution site $\langle E, B \rangle$. *Motile*/*Island* defines two distinct "flavors" of computations: remote evaluations and spawns. The distinctions between the two have significant security consequences. Live computations under COAST span the spectrum from one-shot computations to long-lived client-defined services and the *Motile* abstractions match that span.

The `palindromes` thunk (see Figure 5.4, lines 16–22) that island I dispatches to the dictionary service of island J is a *one-shot* computation; it executes once and transmits a single result via a CURL bound in lexical scope (see Figure 5.4, lines 3 and 22); an application of *Motile* binding semantics. Decentralized promises (section 5.7) are a convenient abstraction for the result of a remote evaluation (by definition a one-shot computation). Importantly, promises are constructed entirely from COAST, using nothing but a CURL and the piece parts of the underlying communication model: an ingress point, transport, and egress point. Promises also play an important role in orchestration and coordination, as shown in the live update protocols of Chapter 8. Again, the influence of COAST reaches deep into infrastructure of *Motile*/*Island*.

Finally, the `palindromes` example illustrates the critical role that binding environments (COAST Rule **Execution**) play in COAST-based services. However, while the COAST rules call for closure transfer as a form of computation exchange (rules **Services** and **Execution**), the style is silent on the specific binding semantics for the variables of transported closures. *Motile* closure transfer always rebinds free variables in the context of the binding environment of the destination execution site while retaining all lexical scope bindings. In other words, the bindings for free variables are always left behind.

There are many alternatives for binding semantics, both languages (such as MAST [255] or Obliq [30]) and formal models [17]. The choice for *Motile*, leaving all free variable bindings behind and transporting all lexical scope bindings is unusual but, 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 (**Execution**) deeply influences the semantics, adaptivity, and security of a style-specific language.[34]

The implementation details of *Motile*/*Island* directly reflect the embedding of computation exchange within the object-capability model of security. *Motile* binding semantics leverage object-capability to guarantee that a hosting island controls all of the capability that a visiting live computation may exercise. Message transmission among computations is explicit: it requires both `curl/send` and possession of one or more CURLs, Likewise message reception demands that a computation holds two distinct capabilities, an egress point *e* and, via the binding environment of an execution site, access to a form of `receive`. Decentralized promises are implemented using only the base mechanisms for COAST **Messaging**; consequently promises are likewise constrained by object-capability. Spawning, implemented as a COAST service, evinces an internal island micro-architecture that is itself COAST-compliant. In turn, a remote evaluation is a spawn absent any communication capability and is an excellent example of the intersection of computation exchange and object-capability security. Finally, the integration of remote evaluation with decentralized promises (`curl/remote*`, `curl/remote`, and `curl/remote/block`) illustrates the complementary roles of computation exchange and object-capability.

[34] Alternatively, an implementation may, for certain assets, support remote references back to the origin site. One obvious example is a remote reference to a Scheme input or output port. However, I regard that behavior as unsafe and in those cases where it is required it should be supported by a proxy computation (service) executing on the island of origin that responds to CURL-mediated requests directed to the fixed asset in question (here reads or writes on a Scheme input or output port).

There is one remaining area, not yet discussed, where the style and the reference implementation also meet: the mobility semantics. Just what is mobile and why has profound security and safety consequences and those issues are the topic of Chapter 6 following.

Chapter 6: How Mobile is Motile and Why?

The COAST style sets a floor for mobility that includes well-known immutable values (strings, numbers and such), data structures (lists, vectors, hash maps and the like), and finally, the triumvirate of closures, continuations, and binding environments — these last three are the reification of live computations, that is, state plus code. However, a language could choose to expand mobility to include for example, input/output streams, green threads, or database cursors.

In this chapter I argue that restrictions on mobility above and beyond what is mandated by COAST are motivated by concerns for security, safety, confidentiality, competitive advantage, and trade secrets among others. In section 6.1 I present the general case for *weak mobility*, wherein mobility is restricted to a subset of language types.

However on-island, where everything is passed by reference, mobility has no meaning and absent any additional safeguards arbitrary assets can be transmitted from one co-resident computation to another (modulo possession of an appropriate CURL). Section 6.2 describes how *fixed assets*, elements that must be confined to their island of origin, arise naturally in *Motile* (and in any implementation of COAST that supports the transmission of binding environments). Serialization, which translates language objects into a flat representation suitable for network transmission, is a critical tool for the interdiction of fixed assets both inter- and intra-island (sections 6.2 and 6.3 respectively). Given that intra-island messaging is by reference, where fixed assets can pass among co-resident computations, one might ask if it is COAST-compliant? In section 6.4 I answer that question in the affirmative; a necessary step in proving that *Motile*/*Island* is capability-safe (Chapter 10).

6.1 Weakness is a Greater Good

There are two forms of code mobility: strong and weak [99]. Under strong mobility all of a language's values are mobile, while weak mobility limits the values that may move from host to host (or in the case of *Motile*/*Island* from one address space to another). The COAST style is silent on the question of strong versus weak mobility and leaves that decision to the implementation. *Motile*/*Island* implements weak mobility because strong mobility:

- Presents a significant risk to system security and integrity in a decentralized environment
- Demands a more complex and less secure implementation
- Threatens the encapsulation and isolation of computations

Individual islands are structures for encapsulation and isolation. As COAST separates computation from communication an island is free to arrange its on-island computations as it pleases — their role, number, and communication topology is hidden from the view of other islands in the network.[1] To guarantee the integrity of inter-island communications the base components of island messaging — transports, ingress points, and egress points — are *fixed assets*; objects confined to their island of origin. A fixed asset is not serializable, that is, there is no serial, on-the-wire representation available that allows a fixed asset to be transmitted to a remote island and reconstructed. All access to a fixed asset of an island must be mediated by a co-resident computation. Active mediation helps to ensure security and integrity by enforcing island-specific preconditions and constraints for each and every access. Fixed assets discourage (but do not necessarily eliminate) capability amplification.[2] Active mediation can consult arbitrary on-island state to thwart unwanted amplification, using island state, for example, to implement and enforce fine-grained, non-delegable capabilities.

Weak mobility also draws a bright line between code mobility versus asset mobility. Object-based distributed systems, for example the language E [170], blur the line between mobility and assets by generating a proxy redirect for every exported object reference.[3] Others, such as Emerald, transport the object wholesale including, if need be, the code of its class [137]. However, neither E nor Emerald deal with the challenges of decentralized systems. While code mobility necessarily includes mobile values and mobile data structures, the weak mobility of *Motile*/*Island* strives for a balance between convenience on one hand and safety and security on the other. A decentralized service may have legitimate and compelling reasons for restricting or forbidding outright forms of asset transfer, including safety, legal, contractual, proprietary, or regulatory considerations. A mobile code infrastructure for a decentralized milieu must not overreach and should allow developers the *freedom to restrict or confine* as circumstances demand.

[1] CURLs testify to the services that an island offers but give no hints to their implementation or organization.
[2] Where two or more capabilities $\kappa_1, \ldots, \kappa_m$ are combined to obtain an additional capability κ' not included among the κ_i.
[3] In Miller's terminology [170] this would be an *inter-vat* reference.

Treating transports, ingress points, and egress points as fixed assets illustrates many of the factors that must be considered. A transport is an on-island container for messages that arrive from elsewhere in the network or are transmitted from one co-resident islet to another. In general, the *Island* implementation hides transports from untrusted code just as programming languages (such as Scheme) leave the evaluation order of the arguments of procedures undefined to prevent programmers from making unwarranted assumptions about run-time behavior. The separation of computation from communication strongly suggests that computations should be ignorant of the details of message transport to permit variation where demanded by domain or circumstance [112]. To this end transports are isolated from computation by the mechanisms of ingress and egress points (an analogue of the input and output ports of Weaves [112]). Including transports among the primitive structures (in contrast to lists or vectors for example) for which serialization is automatic threatens the separation of communication and computation.

Accommodating the transmission of an I-resident transport t to a remote island J by introducing a default proxy object on I subverts the intent and safeguards of ingress points and CURLs. Consider instead two COAST-compliant alternatives. First, island I can implement and deploy a co-resident islet x_t whose sole function is to proxy remote access to t for reading, writing, or both. That access, itself mediated by one or more CURLs, ingress points, and egress points, enjoys the same guarantees of safety, integrity, and control as any other inter-island message exchange — no additional mechanisms or special case circumventions are required; the security analyses and guarantees are simpler, easier to understand and explain, and less likely to fall victim to subtle faults. Alternatively, one can deconstruct (with trusted primitives) a transport t and convert t and its contents into a data structure (for example a vector, list, or record) that is serializable for transmission elsewhere. This provides a "copy" of transport t that cannot affect in any way the operation of t. As in the first alternative, this task must be undertaken on the island of origin; a restriction that is consistent with the philosophy and intent of COAST. In both alternatives the transport is strongly encapsulated and in the second no transport can ever leave its island of origin without the island's full knowledge and cooperation.

For ingress points a mechanism already exists for off-island transport: CURLs. Ingress (and egress) points are a locus for stateful gates whose evaluation depends on island-resident state. Again the best that one might do for the transmission of an ingress point off-island is to transmit a reference to a proxy object (co-resident with the ingress point) but that is roughly how CURL-based messaging is implemented

in *Island* in the first place. The gates of an ingress (egress) point may be proprietary, confidential, or concealed for the sake of security. Egress points, unlike ingress points, have no off-island representation of any kind and are never serialized. This posture guarantees that only only islets co-resident with the egress point and that hold the egress point can ever read messages from the transport to which the egress point refers. In other words, no message that is received by an island I is ever (re)transmitted off I unless its transmission is mediated by an islet x of I; in which case islet x must hold a CURL u_J to deliver any message to island J. Any other policy violates the COAST **Messaging** rule and is an unwarranted exception to communication by introduction.

Scheme ports[4], objects which represent input and output devices, are another domain-independent example of fixed assets. Ports cannot be denoted by CURLs or serialized for inter-island transmission. An islet may be granted ports for its use via the binding environment B of its execution site or it may receive a port in an intra-island message. However, in most cases those ports will be embedded in the closures of domain-specific primitives appearing in B to better regulate, monitor, and confine an islet's exercise of those ports. The more interesting fixed assets of a COAST system are likely domain-dependent: databases, digital sensors, actuators, legacy services including RESTful web services, cyber-mechanical devices, network devices, log files of all descriptions, cyber-monitoring, authentication and defensive services, special-purpose computing devices, as well as critical computational and storage resources. In these cases there will likely be a constellation of domain- and asset-dependent islets executing under customized execution sites that actively mediate access to, and exercise of, these valuable resources. As islands are inexpensive to create and deploy one can imagine individual islands each of whose sole purpose is to encapsulate, protect, regulate, command, and monitor a particular mission-critical resource.

Fixed assets are commonplace in the bricks and mortar world. Coca-Cola has its secret formula, manufacturers restrict access to portions of factory floors where closely-held tooling or production processes are employed, and product design labs are the Fort Knoxes of consumer electronics. In cyberspace software-based services such Amazon, Google, and Uber rely on proprietary algorithms and software for core business operations while, in the twilight zone of the intersection of the physical world and cyberspace, domain-dependent design and analytical software tools are critical, valuable assets. Given the ubiquity of fixed-assets strong mobility in a decentralized ecology appears ill-considered and reckless.

[4] [216], Section 6.13

Weak mobility defaults to effective protection and where stronger mobility is required *Motile*/*Island* developers can construct domain-specific protocols for those cases. This stance aligns with POLA where the default is strong and adaptive protection but with sufficient expressiveness and flexibility to oblige service- and domain-specific needs.

Strong mobility is more difficult to implement than weak and in many cases the semantics of strong mobility are unclear or of questionable value.[5] Complexity often increases the attack surface and while strong mobility is both seductive and convenient its allure is outweighed by the risks of malicious abuse. Even an outwardly innocent extension may give attackers valuable clues, expose vital information, or threaten the integrity of an island. As should be clear the security considerations for mobile code systems are nontrivial and even a detailed, conscientious security review of a mobility extension may overlook a subtle, but dangerous flaw. Absent compelling use cases conservative security argues against strong mobility furthermore, strong mobility may violate POLA in many specific instances.

Finally, strong mobility threatens island encapsulation and isolation. *Motile* mobility has been designed and implemented to ensure effective closure transfer without threatening island integrity. Where the mobility primitives are inadequate islands can implement higher-level protocols to simulate strong mobility while minimizing risk. Note that if island I can deploy closures to execution sites (with sufficient functional capability) on remote island J then I can deploy any protocol it likes, implemented as closures, to J. Since mobile code itself can be used to overcome many of the limitations of weak mobility without introducing additional security risks (beyond those already at play) it makes little sense to undermine that security for the sake of questionable benefits. It may be more productive and far safer to create additional *Motile* libraries (themselves written in *Motile*) that implement common use cases for strong mobility as higher-level protocols. This reduces the complexity of the serializer, gives greater freedom to developers without risking fundamental security safeguards, does not require the introduction of new (and likely complex) language semantics, minimizes the size of the *Motile*/*Island* base infrastructure, and allows for considerable island diversity and adaptation.

[5] Serialized representations for common data values (numbers, strings, booleans, characters, strings and byte vectors) are straightforward, as are common data structures (lists, vectors, and hash maps). The serialization of *Motile* closures and CURLs is non-trivial and goes directly to questions of effective and safe serial representations. Stateful objects such as ports or islets are far more complex and present technical challenges. Serialization for more exotic domain-dependent objects may be infeasible, impractical, or used so infrequently that developers are better off with an object-specific protocol that does not require extending the serializer. However, nothing prevents collaborating service providers from augmenting the *Motile* (de)serializer to selectively strengthen mobility for a valued use case (see Section ??? for a brief discussion of amending or extending the *Motile* (de)serializer).

Weak mobility confines sensitive (fixed) assets to their island of origin. However, it has no effect within the confines of an island (that is, intra-island). If computation x holds a CURL u_y for co-resident computation y then nothing prevents x from transmitting to y a reference to any fixed asset that x may hold. Since many hundreds of untrusted co-resident computations may reside "alongside" x at any one time it behooves a prudent and suspicious authority to confine x to prevent the unwanted transfer of capability into untrusted hands. This security issue and its solution are the topics of sections 6.2 and 6.3 below.

6.2 Intra- versus Inter-island Messaging

While fixed assets cannot be transmitted inter-island they can be transmitted intra-island from one co-resident islet to another. Islets residing on the same island share a single, homogeneous address space (by definition of an island). *Motile*/*Island*, for reasons of efficiency, implements intra-island message passing by reference. To partially 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 [180], that is, immutable and version-preserving. All objects (including murmurs) shared among an island's islets are immutable; this improves islet isolation and eliminates the classic data races seen in threaded imperative languages.[6]

However, there are subtle distinctions between intra- and inter-island message passing that are intimately intertwined with fixed assets and the implementation of serialization, translating *Motile* values (including closures) into flat representations suitable for network transmission. Any *Motile* value, fixed asset or otherwise, may be included in a intra-island message. However, if islet y receives a reference to a fixed asset ι in a message from islet x we cannot perforce conclude that islet y has the necessary functional capability (in the binding environment of its execution site) to exercise or probe ι. For y, in that case, fixed asset ι is just an opaque reference that islet y may be able to transmit to other co-resident islets or embed in a data structure, but nothing more. There is good reason to allow fixed assets to be

[6] There are two caveats. First, single assignment does not eliminate races in message ordering in the application-specific messaging protocols implemented by *Motile*/*Island* computations (see [41] for a discussion of this problem in Erlang and tools that detect message races). Second, the binding environments of execution sites may contain functions that rely upon library functions or system primitives by which multiple islets can share mutable data, test their values, and update them destructively (see [40] for a discussion of this problem in Erlang and the use of *dialyzer*, a popular static analyzer for Erlang, that detects the race conditions these functions can introduce). Such functions are inherently dangerous and must be used with caution. They are best avoided altogether if possible.

transmitted from one islet for another. For example, an islet x may be a factory for a fixed asset ι that x wraps in closures $f_{\iota,1}, ..., f_{\iota,m}$ for transmission to other co-resident islets.[7] Alternatively, islet x can dispense adaptations for fixed assets to its fellow islets in response to changing conditions such as asset load variations, asset failure and recovery, threats of attack, or software updates. In a reversal of roles x could be a trusted service that manipulates a set of fixed assets on behalf of co-resident computations holding members of that set, but whose own execution sites lack the requisite functions.

The calling convention for *Motile* closures differs from that of the *native* closures of the underlying Scheme implementation.[8] The vast majority of the closures appearing in execution-site binding environments are native Scheme closures wrapped with *Motile* calling conventions. But these are rightfully fixed assets:

- Many, though by no means all, of the native Racket Scheme closures are written in C. Their binary representation is inaccessible, unsuited for serialization, and deeply processor-dependent; inter-island transmission is infeasible and unwanted.
- Other Racket Scheme functions are written entirely in Racket Scheme and are compiled into Racket virtual machine bytecodes. That bytecode representation is poorly documented and frequently changes from one version of Racket to another and, in any case, there is no guarantee that a remote recipient is even running atop Racket much less a compatible version.
- A binding environment may often contain domain-specific library functions written entirely in C/C++ by developers who never imagined that their library would be rehosted within Scheme wrappers. Here the Scheme interface is supplied by the Racket FFI (Foreign Function Interface) and then wrapped, like any other native Scheme closure, in the *Motile* calling conventions. In appearance and behavior these functions are identical to any other native Scheme function.
- Binding environments may contain proprietary functions, developed by the island authority, or offered under license from a third party. In any case the island can be obligated by policy or contract to take reasonable technical measures to protect the details of their implementation.

These considerations: machine-language binary blobs unsuited for (de)serialization and exchange, library functions written in languages for which mobile code is unavailable or infeasible, and proprietary

[7] The poor man's version of a singleton object that is shared among multiple computations.

[8] The *Motile* compiler is a transpiler, compiling *Motile* closures into the native closures of Racket Scheme (`www.racket-lang.org`)).

restrictions on function implementations, will hold, to one degree or another, for any *Motile*/*Island* implementation. Inevitably there are closures for which inter-island transport is impossible, impractical, or unwanted.

To prevent native, foreign, or proprietary closures migrating from one island to another *Motile*/*Island* relies on two interlocking mechanisms:

- Every closure that respects the *Motile* calling conventions can "decompile" itself into a graph structure (possibly cyclic) called a Motile Assembly Graph (MAG). The graph nodes are abstract machine instructions for a high-level lambda calculus engine and every appearance of a free variable α in the *Motile* source is represented by the instruction

  `#(global/get` α`)`

  where `global/get` names the abstract machine instruction and α is a symbol denoting the binding name of the function. The *Motile* recompiler, invoked during deserialization, recompiles this instruction into a native closure g that fetches the binding of symbol α from a binding environment B (given as an argument of g).
- A set of higher-order functions for the construction of binding environments accept a native function α as an argument and return an equivalent function $\tilde{\alpha}$ that (1) obeys the *Motile* calling conventions (thereby bridging the gap between *Motile* and the underlying host Scheme) and (2) always decompiles into a single node MAG, `#(global/get` α`)`. Every native closure α included in the binding environment of an execution site appears as $\tilde{\alpha}$.

In the case of inter-island closure transmission, these two mechanisms guarantee that free variable bindings are always left behind and resolved on-demand in the context of an execution site at the destination island.[9] The same closure λ can be transmitted to multiple islands and the semantics of its execution can vary from island to island as the free variables of λ are rebound in the context of distinct, island-specific execution sites.

To illustrate these mechanics consider the code snippet[10] of Figure 6.1 as it executes in the context of an execution site $\langle E, B\rangle$ on island I:

[9] A free variable is resolved at time of reference in the course of the execution of the *Motile* expression in which it appears.
[10] *Motile*, like many other programming languages, is case sensitive. Variables `v` and `V` are distinct.

```
(let ((LIST list) ; Capture native function list in local binding.          1
      (MAP  map)) ; Capture native function map in local binding.           2
  (curl/send                          ; Native funcion.                     3
    @u                                ; Target CURL.                        4
    (lambda ()                        ; Anonymous thunk.                    5
      (MAP (LIST 1 2 3 4) even?))))   ; even? is an unbound free variable.  6
```

Figure 6.1: Bring native bindings (from the binding environment of the execution site) into lexical scope and transmit those bindings in a closure.

1. There are three variables in lexical scope: `LIST` (line 1), `MAP` (line 2), and `@u` (line 4).[11]
2. There are four free variables: `list` (line 1), `map` (line 2), `curl/send` (line 3), and `even?` (line 6).
3. At execution time, the local scope variables, `LIST` (line 1) and `MAP` (line 2), are bound to the values of `list` and `map` respectively. Those two values are *native closures*, obtained by resolving (at run-time) the symbols `list` and `map` with respect to binding environment B.

Assume here that CURL `@u` (line 4) is an inter-island CURL for island J. The anonymous closure (lines 5–6), passed as the message payload to `curl/send` (line 3), contains lexical scope references to two native closures, the values of `LIST` and `MAP` respectively (line 6), and a single free variable, `even?` (line 6).

Inter-island transmission is, of necessity, somewhat elaborate. Under the covers, out of sight of the *Motile* programmer, the message payload is serialized prior to transmission and, in doing so, the transitive closure of every closure in the payload is decompiled to its MAG representation. The combined MAG representations, data structures, and data values of the payload are then converted into a flat, on-the-wire representation suitable for network transmission. On receipt by the destination island the on-the-wire representation is silently reconstituted as a *Motile* value and any MAGs therein are recompiled into *Motile* closures.

In this regime both lexical scope references, `MAP` and `LIST`, each bound to a native closure, decompile to a single MAG instruction, `#(global/get map)` and `#(global/get list)` respectively.[12] The free variable `even?` decompiles likewise to `#(global/get even?)`. Thus, all native closures are abandoned and left behind when the representation of the closure (lines 5–6) is transmitted to J. At island J the closure is reconstituted as

```
(lambda () (map (list 1 2 3 4) even?))
```

[11] For the sake of brevity and simplicity I omit the lexical scope binding of `@u`.

[12] Decompilation preserves the original name assigned to a native closure in the resolving binding environment.

where `map`, `list`, and `even?` are free variables. When island J executes the reconstituted closure in a J-specific execution site $\langle E, B\rangle$ those three free variables will be resolved relative to binding environment B on J. In this way the semantics of execution of the received closure are J-dependent and may or may not match the execution semantics of the original closure at island I.

6.3 Intra-Island Messaging

The code snippet of Figure 6.1 also illustrates the distinction between intra- and inter-island transmission. In this case assume that target CURL `@u` is a CURL for island I, the same island on which this code snippet is executing. The anonymous closure (lines 5–6) is transmitted intra-island via CURL `@u` (line 4) by `curl/send` (line 3). Variables `LIST` and `MAP` are within lexical scope and their bindings (native closure implementations of well-known Scheme functions `list` and `map`) are carried along in the closure. However, when the anonymous closure is executed elsewhere on island I (in the context of some other execution site $\langle E', B'\rangle$) then `even?` (line 3), the only free variable, will be resolved against binding environment B'.

What is happening here? The anonymous closure is a vehicle for transferring native functional capability among co-resident islets. In other words, intra-island transmission can "leak" functional capability.[13] This is both an opportunity and a threat. An islet x can generate functional capability for other co-resident islets by confining native closures (available in x's binding environment) within specialized *Motile* closures. For example, the closures that x exports may limit the arguments of the "inline" native closures, supply context-specific error checking, apply the native closures in a manner known to be safe, in short, the confinements sacrifice power and generality for safety and specificity. Islet x can limit these closures in many other ways, for example, rate of use, time of use, number of uses, or non-delegation and use can also depend upon time-varying state of I. It is also possible to implement revocation so that a closure λ, distributed earlier, can never be used again. Alternatively, the purpose of the closure may just be encapsulation. It is impossible for an untrusted islet to inspect the interior of a *Motile* closure — any native closures bound within are inaccessible and cannot be extracted, called directly, or tampered with. Consequently, by mixing inline native closures and free variables a closure generated by one islet and transmitted to another co-resident islet can selectively amplify the functional capability of a receiving islet.

[13] We can be far less subtle about it, for example: `(curl/send @u map)`.

Native functions are but one example of fixed assets and intra-island transmission can "leak" any fixed asset from one on-island islet to another. Egress points, a fixed asset common to all islands, can be transmitted from islet to islet within the confines of a single island just as easily as native functions. Sharing fixed assets portends a blend of opportunity and threat comparable (in both mechanism and technical measures) to the exchange of native closures. In all cases, fixed assets can be encapsulated in closures and the mechanisms outlined above for confining native closures apply in equal measure to the confinement of non-procedural fixed assets. The same benefit holds — an islet can generate asset-specific capability for other co-resident islets by confining a fixed asset within one or more asset-specific closures.

Sharing or distributing fixed assets is critical as it allows for "factory" islets whose role is to create tailored or encapsulated (via closures) fixed assets for use elsewhere on an island. Nonetheless, sharing a fixed asset is not required and is asset- and application-specific. For example, an islet may regulate access to, and exercise of, a fixed asset and never share it with any other islet. However, the islet holding the fixed asset may be a service provider to other islets — it implements a service protocol via messaging that allows other islets (on-island, off-island, or perhaps both) to issue service requests and receive service responses.

On the other hand, an untrusted or malicious islet may use intra-island transmission to distribute native capability that is best left confined. There are several mechanisms available to restrict or prevent the unwanted distribution of functional capability among untrusted islets on an island:

1. Never allocate an execution site containing unsafe functional capability to an untrusted islet. POLA applies equally to functional and communication capability. Where feasible, confine functional capability to the bare minimum required to satisfy the service goals of the execution site.
2. Poison the unsafe functions $\alpha_1, \ldots, \alpha_m$ in the binding environment B of an execution site by pinning their execution to a given islet. In other words, if α_j is pinned to islet x and ever executed by another islet y then α_j fails softly (for example by immediately returning a constant safe value). Even if islet x confines α_j by embedding it in a custom closure λ_{α_j} that closure, absent effective execution of α_j, is likely useless or harmless.
3. Mark each CURL for off-island use only or on-island use only.[14] Consistently marking CURLs as

[14] Each CURL can be marked for use off-island, on-island, or both.

either "off-island only" or "on-island only" will prevent a CURL u_x, intended for off-island communication with untrusted islet x from ever being appropriated and used by any co-resident islet y. This will frustrate, though not necessarily prevent, efforts to leak functional capability from y to x by exploiting a CURL that granted excessive communication capability.

4. Confine `curl/send` in the binding environment of untrusted islet x so that it rejects any on-island CURL. If x cannot transmit to co-resident islets then it simply cannot disseminate functional capability.[15]
5. Expunge `curl/send` from the binding environments of those execution sites where intra- and inter-island transmission is not required. For example, in remote execution (as opposed to spawning) the result of the evaluation of the received thunk λ will be transmitted back by the executing island and not by λ itself. In this case, an island I as a matter of policy and practice can omit `curl/send` from the binding environment B of the execution site $\langle E, B\rangle$ that I deploys for remote execution. Where this is not feasible `curl/send` can be customized to restrict intra-island communication to a small number of trusted ingress points. Likewise, the same instance of `curl/send` can be restricted to a small whitelist of islands to which which inter-island transmission is acceptable.
6. Expunge `receive` from the binding environments of those execution sites where intra- and inter-island message reception is not required. Any islet x executing in the context of such an execution site lacks the functional capability to receive messages and is incapable of receiving additional functional capability from any source.
7. Poison each egress point $t\triangleright$ with a whitelist or blacklist that restricts the set of islets that may receive messages. If possible, ensure that an islet x holds no egress points at all as then it cannot receive messages from any source whatsoever.
8. Augment `curl/send` in the binding environment of untrusted islet x so that whenever the target CURL is an on-island CURL it substitutes the payload `(deserialize (serialize` p`))` for the given payload p. This has the effect of expunging all native functional capability from the payload (just as would occur in an inter-island transmission); therefore no co-resident islet y can ever acquire additional functional capability from x. While this has the disadvantage of significantly increasing the overhead of intra-island transmissions from x the additional security assurances may be worth

[15] However, x might obtain an unlimited `curl/send` from another co-resident islet and circumvent the ban.

the cost.

9. Inspect the payloads of inbound messages sent via intra-island CURLs for unwanted or suspicious objects such as other intra-island CURLs or native closures. Using an inspector (derived from the *Motile*/*Island* serializer) it is possible to extract every native closure and intra-island CURL embedded in the payload and embargo any message whose contents is a potential threat.[16]
10. Define specialized transports that include deep payload inspection and capability interdiction in their delivery. For intra-island transmissions this also ensures that the transmitting islet bears the cost.[17]

All of the techniques outlined above apply to all fixed assets and are not limited to confining the transfer of native closures. The rules of COAST detail where and how capability is exercised and thereby precisely identify the choke points for the transfer of capability of all forms.

6.4 Is Intra-Island Messaging COAST-Compliant?

In a word, yes. The outward mechanisms of intra-island message passing: CURLs, ingress points, transports, and egress points are identical to those inter-island messaging and conform with the COAST **Messaging** rule. The only difference between the two is *serialization*, a necessity for inter-island messaging but overkill for the common use cases of intra-island messaging.[18] There is a delicate balance at work here. An island is an encapsulation and isolation boundary established by a single authority. Nothing prevents an authority from having many islands within its sphere and island multiplicity and specialization can play important roles in system reliability, recovery, security, and adaptivity. Highly efficient, low overhead intra-island communication is a practical necessity and reflects the expected costs of commu-

[16] The inspector would be embedded within a customized version of `receive` placed in the binding environments of the execution sites of untrusted islets. Each such untrusted islet then pays the execution cost (accounted in memory consumption and processor cycles) of deep inspection. By limiting the total execution cost and lifespan of an islet to prevent resource exhaustion we ensure that untrusted islets "pay" for the burdens that they impose upon the island on which they reside.

[17] The *Island* implementation guarantees that on-island transmission is always executed by the same thread of control that is delegated to the sending islet.

[18] Serialization is a necessity irrespective of the detailed architecture chosen for the infrastructure of a COAST-conformant system. First, it is unlikely that distinct spheres of authority will share address spaces and even a single sphere of authority will turn to multiple, distinct address spaces as a well-understood, defensive measure against attack — distinct address spaces will be commonplace in any COAST-based infrastructure. At a minimum, serialization (of some form) is demanded by the computation and value exchange lying at the heart of COAST. Even if exchange is based on a bytecoded virtual machine or processor-dependent binary blobs, it still must be possible to transmit and reconstruct live data structures and for this some form of serialization is required. Finally, CURLs, besides having an in-memory representation must also, for practical reasons, have a human-readable representation that is both roughly consistent with the CURL on-the-wire format for network transmission and that allows them to be freely exchanged out-of-band; scribbled on the back of a cocktail napkin, tucked into the body of an email, embedded in a web page, or placed as a comment in source code.

nication within the confines of a single authority. As a rule, intra-organization communication tends to be more efficient and less burdensome than inter-organization communication. The reduced costs reflect the physics of locality: we expect higher communication rates intra-island versus inter-island and intra-island latency and overhead magnify the total cost of intra-island messaging disproportionately.

However, with ease of communication comes complacency. The threat model for COAST categorizes all threats, irrespective of source, as variants of insider attacks. Hardening the interior of an island is a necessity and the mechanisms for confining intra-island communication capability are identical to those for confining inter-island communication. As Miller points out ([170], Chapter IV, *Emergent Robustness*, pp. 145–161) object-capability security is fractal and can be reapplied in ever-finer granularity on ever-smaller scales of interaction and capability. Confining communication and functional capability *within* an island is as necessary as confinement in inter-island exchanges.

However, the capability relationships in a large system (not to mention a system-of-systems) are invariably rich and complex. And while a single island is a small, encapsulated domain it may be the residence of several thousand islets of which only a small fraction are trusted. While tools for capability analysis [136, 220, 221] are outside the scope of this thesis it is a rich area for future work.

6.5 Summary

From a security and safety perspective I hold a skeptical view of languages that support strong mobility or that, by default, grant mobility to sensitive local assets such as input/output streams, file systems, or green threads. As an alternative to mobility for these sensitive assets I argue for the use of asset-specific proxies, themselves COAST services, that function as protective intermediaries between off-island computations and the asset itself. This approach has several advantages:

- It affords island- and asset-specific protection that can vary from one asset instance to another, from one asset class to another, and from one island to another.
- It does not require any new mechanism and builds upon the mechanisms at hand: communication by introduction, mobile code, execution sites, and CURLs.
- It is flexible and easily encapsulated in asset-specific frameworks.
- Since asset proxies are just another COAST service other adaptation mechanisms apply as well, for example, live update (see Chapter 8).

In general, the weak mobility of *Motile* seems more than adequate to meet the needs of many systems and in our experimental work to date we have not encountered a use case that presented a critical and compelling need for strong mobility. For many common fixed assets it may be sufficient to implement a proxy and, if the frequency of the use case warrants, construct a *Motile* framework or library for the common cases, such as input/output ports.

To the extent that fixed assets must be interdicted *intra-island* proxies can be used as well however, there is an alternative mechanism, *serialization*, that can thwart any effort to surreptitiously pass a sensitive asset to another co-resident computation. To my knowledge, with two notable exceptions, there is little discussion in the literature of mobile-code systems on the role of serialization as a tool for security and safety. The first is Kornstaedt [147], the only doctoral thesis that I know of devoted to the topic of serialization, and the second is Rossberg, Tack and Kornstaedt [204], who describe a general framework for serialization in the context of Alice ML, a mobile-code variant of ML.

Since an island must deserialize every single message transmission that crosses its border and reconstruct, element by element, the contents of a message, an island can conduct deep inspection of the objects that are entering its sphere of authority. Using (de)serialization as a starting point for logging, debugging, event tracing, and instrumentation is an interesting avenue for future work.

Nothing in computation exchange *demands* strong mobility and in a decentralized world much argues against it. However, computation exchange can simplify the deployment of the protocols that an island may require for access (via proxy) to a fixed asset. In this case the protocol implementation is simply bundled as an additional closure λ that either travels with, or is transmitted to, a live computation executing elsewhere. The owner of the fixed asset can impose whatever restrictions it likes on asset access that those restrictions can be implemented, in whole or in part, by λ. Since closures are a fundamental form of encapsulation the live computation may be ignorant of the restrictions being imposed on its access to the remote asset. Once again we see a productive interplay between computation exchange and object-capability.

The exploration of *Motile* concludes with a description of the language from the perspective of two well-known Scheme standards, R5RS [143] and the more recent R7RS [216], in Chapter 7.

Chapter 7: Motile: The Details

Motile strongly resembles Scheme R5RS and shares its syntax [143] though it differs in substantial details such as its lack of imperative assignment (for example, no `set!` for lexical variables or `set-car!` and `set-cdr!` for cons cells). *Motile* also borrows a few features from Scheme R7RS [216]; however, *Motile* is missing features now commonplace in a mature Scheme implementation such as hygienic macros and exceptions. I assume that the reader is familiar with R5RS and R7RS and, for the sake of brevity, devote effort to describing the differences between *Motile* and R5RS rather than rehashing the obvious congruences.

7.1 Expressions

7.1.1 Variable References

An ⟨identifier⟩ α that appears in an ⟨expression⟩ is a *variable reference*. Each variable reference is either *closed*, there exists a ⟨local binding⟩ (*α* ⟨expression⟩) in lexical scope, or *free*, no ⟨local binding⟩ for α appears in lexical scope. In the later case the free variable reference is resolved at execution time by consulting the global binding environment B for a binding α/u. It is a runtime error to evaluate a free variable reference that is not defined in the given global binding environment B.

7.1.2 Procedures

The *Motile* syntax and semantics for procedures, that is the evaluation of a ⟨lambda expression⟩ (lambda ⟨formals⟩ ⟨body⟩), is that of R5RS.

7.1.3 Procedure Calls

The syntax and semantics of a *Motile* ⟨procedure call⟩ (⟨operator⟩ ⟨operand⟩*) is identical to that of R5RS with the exception that the evaluation of variable references is always relative to an explicit global binding environment B.

7.1.4 Conditionals

Like R5RS `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)` is the fundamental conditional form. However, the form `(if` ⟨test⟩ ⟨consequent⟩`)` is not available in *Motile*. Instead the alternatives, `(when` ⟨test⟩ ⟨sequence⟩`)` and `(unless` ⟨test⟩ ⟨sequence⟩`)`, are provided (with the obvious semantics).

7.1.5 Assignment

Motile is a single-assignment language, that is, every variable is initialized at binding time and is always read-only thereafter. Consequently, the special form `(set!` ⟨identifier⟩ ⟨expression⟩`)` is unavailable in *Motile*.

7.2 Derived Expressions

7.2.1 Conditionals

The derived construct `cond` is identical to that of R5RS. The well-known and much loved `case` construct is not defined in *Motile*.[1]

7.2.2 Binding Constructs

The binding constructs `let`, `let*`, and `letrec` are identical to those of R5RS. The binding construct `letrec*`, taken from R7RS, is also available in *Motile*.

7.2.3 Sequencing

Motile provides two of the three variants of `begin` defined in R7RS[2]:

- `(begin` ⟨definition⟩$^+$ ⟨expression⟩$^+$`)` is equivalent to a `letrec*` where each ⟨definition⟩ appears, in definition order, as a ⟨local binding⟩ and each ⟨expression⟩ appears, in textual order, as an ⟨expression⟩ in the ⟨sequence⟩ comprising the ⟨body⟩ of the `letrec*`.
- `(begin` ⟨expression⟩$^+$`)` is equivalent to `(let ()` ⟨expression⟩$^+$`)`.

7.2.4 Iteration

The iteration construct `(do (`⟨iterator⟩*`) (`⟨test⟩ ⟨do result⟩`)` ⟨command⟩*`)` is identical to that of R5RS.

[1] I know, I know . . . chalk it up to laziness. I kept on meaning to throw it in and never got around to it.
[2] The third variant, used as a library declaration in R7RS, is not included in *Motile*.

7.2.5 Quasiquotation

Quasiquotation in *Motile* is identical to that of R5RS.

7.3 Macros

The form `(define-macro (⟨name⟩ ⟨define formals⟩) ⟨body⟩)` defines a nonhygienic macro with the given ⟨name⟩ and whose extent is the lexical scope in which the macro definition appears.[3] Each macro definition generates a function `(lambda (⟨define formals⟩) ⟨body⟩)`. Given a macro with ⟨name⟩ α then, at compile time, any macro form $(\alpha\ s_1\ \ldots\ s_n)$ appearing in the lexical scope of macro α is evaluated with compile-time expressions $s_1, \ldots, s_n$ as arguments and the return value v of that evaluation is substituted for the form $(\alpha\ s_1\ \ldots\ s_n)$. The value v may, in turn, contain forms that require macro expansion. Macro expansion repeats until the expansion is free of macro forms.

7.4 Standard Procedures

All *Motile* closures are evaluated in the context of a given global binding environment, for which the `BASELINE` environment is the default. The `BASELINE` environment is severely constrained and includes only pure functions free of imperative side effects. For example, input/output functions such as `read` or `write` are absent as are all of the *Island* messaging primitives. `BASELINE` is often the starting point for the binding environments of richer, domain-specific *Island* execution sites.

7.4.1 Type Predicates

Procedure	Standard	Procedure	Standard	Procedure	Standard
`boolean?`	R5RS	`bytes?`	Racket	`char?`	R5RS
`number?`	R5RS	`null?`	R5RS	`pair?`	R5RS
`procedure?`	R5RS	`string?`	R5RS	`symbol?`	R5RS

The predicate `bytes?` is equivalent to the R7RS predicate `bytevector?`.

7.4.2 Equivalence Predicates

Procedure	Standard	Procedure	Standard	Procedure	Standard
`eq?`	R5RS	`eqv?`	R5RS	`equal?`	R5RS

[3] The *Motile* compiler maintains a distinct, scoped, compile-time namespace for macros.

7.4.3 Numbers

Numerical Types

Procedure	Standard	Procedure	Standard	Procedure	Standard
number?	R5RS	complex?	R5RS	real?	R5RS
rational?	R5RS	integer?	R5RS	exact?	R5RS
inexact?	R5RS				

Numerical Predicates

Procedure	Standard	Procedure	Standard	Procedure	Standard
=	R5RS	>	R5RS	<	R5RS
>	R5RS	<=	R5RS	>=	R5RS
zero?	R5RS	positive?	R5RS	negative?	R5RS
odd?	R5RS	even?	R5RS		

Numerical Functions

Procedure	Standard	Procedure	Standard	Procedure	Standard
max	R5RS	min	R5RS	+	R5RS
*	R5RS	-	R5RS	/	R5RS
abs	R5RS	quotient	R5RS	remainder	R5RS
modulo	R5RS	gcd	R5RS	lcm	R5RS
numerator	R5RS	denominator	R5RS	floor	R5RS
ceiling	R5RS	truncate	R5RS	round	R5RS
rationalize	R5RS	add1	Racket	sub1	Racket

Motile also includes 1+ and 1- as common shorthand for add1 and sub1 respectively.

Transcendental Functions

Procedure	Standard	Procedure	Standard	Procedure	Standard
exp	R5RS	log	R5RS	sin	R5RS
cos	R5RS	tan	R5RS	asin	R5RS
acos	R5RS	atan	R5RS		

Exponents

Procedure	Standard	Procedure	Standard	Procedure	Standard
`sqrt`	R5RS	`expt`	R5RS		

Representations

Procedure	Standard	Procedure	Standard	Procedure	Standard
`exact->inexact`	R5RS	`inexact->exact`	R5RS	`number->string`	R5RS
`string->number`	R5RS				

7.4.4 Pairs and Lists

Motile lists are immutable hence the imperative procedures `set-car!` and `set-cdr!`, found in R5RS and later Schemes, are unavailable.

Whenever a function returns a pair or a list that pair or list is immutable.

Procedure	Standard	Procedure	Standard	Procedure	Standard
`append`	R5RS	`assoc`	R5RS	`assq`	R5RS
`assv`	R5RS	`car`	R5RS	`cdr`	R5RS
`caar` ... `cddr`	R5RS	`caaar` ... `cdddr`	R5RS	`caaaar` ... `cddddr`	R5RS
`cons`	R5RS	`list`	R5RS	`list-ref`	R5RS
`list*`	R7RS	`member`	R5RS	`memq`	R5RS
`memv`	R5RS	`null?`	R5RS	`reverse`	R5RS

7.4.5 Symbols

Procedure	Standard	Procedure	Standard	Procedure	Standard
`symbol?`	R5RS	`string->symbol`	R5RS	`symbol->string`	R5RS

7.4.6 Characters

Procedure	Standard	Procedure	Standard	Procedure	Standard
`char->integer`	R5RS	`char-alphabetic?`	R5RS	`char-ci<=?`	R5RS
`char-ci<?`	R5RS	`char-ci>=?`	R5RS	`char-ci>?`	R5RS
`char-downcase`	R5RS	`char-lower-case?`	R5RS	`xchar-numeric?`	R5RS
`char-upcase`	R5RS	`char-upper-case?`	R5RS	`char-whitespace?`	R5RS
`char<=?`	R5RS	`char<?`	R5RS	`char=?`	R5RS
`char>=?`	R5RS	`char>?`	R5RS	`char?`	R5RS
`integer->char`	R5RS				

7.4.7 Strings

Motile strings are immutable; hence the imperative procedure `string-set!`, found in R5RS and later Schemes, is unavailable. The function `string-copy` is omitted as well since in *Motile* it amounts to the identity function.

Whenever a function returns a string as its value that string is immutable.

Procedure	Standard	Procedure	Standard	Procedure	Standard
`format`	Racket	`list->string`	R5RS	`make-string`	R5RS
`string`	R5RS	`string->list`	R5RS	`string->number`	R5RS
`string->symbol`	R5RS	`string-append`	R5RS	`string-ci<=?`	R5RS
`string-ci<?`	R5RS	`string-ci=?`	R5RS	`string-length`	R5RS
`string-ref`	R5RS	`string<=?`	R5RS	`string<?`	R5RS
`string=?`	R5RS	`string>=?`	R5RS	`string>?`	R5RS
`string?`	R5RS	`substring`	R5RS	`string-append`	R5RS

Motile implements the two argument form of (`make-string` k c) given in R5RS and returns an immutable k-character string $c \ldots c$. The single argument form (`make-string` k) is not provided as it must return a mutable k-character string.

7.4.8 Bytes

The procedures for *Motile* byte strings are modeled after those of Racket Scheme with the difference that all byte strings are immutable.[4] A *byte* is an exact, non-negative integer in the range 0…255.

(bytes? x) returns #t if x is a byte string and #f otherwise.

(bytes $i_0 \ldots i_{n-1}$) returns a byte string whose individual elements are i_0, … i_{n-1}, in index order, where each i_j is an exact, non-negative integer in the range 0…255. The length of the byte string is n.

(bytes/length x) returns the length of byte string x.

(bytes/ref x i) returns byte x_i of byte string x, $0 \leq i < n$ where n is the length of x.

(subbytes) x i) returns an immutable byte string comprising bytes x_i, … $xn-1$ in index order. If $i = 0$ then byte string x is returned.

(subbytes x i j) returns an immutable byte string comprising bytes x_i, … x_{j-1} in index order. If $i = 0$ and $j = n$ (the length of x) then byte string x is returned.

(bytes/append $x_0 \ldots x_m$) returns an immutable byte string that is the concatenation of byte strings x_0, … x_m.

(bytes/append* $x_0 \ldots x_m$ y) returns an immutable byte string that is the concatenation of byte strings x_0, … x_m and the byte strings s_0, … s_j of list $y = (s_0 \ldots s_j)$. In other words, bytes/append* is the byte string analogue of list*.

(bytes-to-list x) returns an immutable list comprising the individual bytes x_0, … x_{n-1}, in index order, of immutable byte string x.

(list-to-bytes y) returns an immutable byte string constructed from the contents of list $y = (y_0 \ldots y_m)$ and is equivalent to (bytes $y_0 \ldots y_m$).

(bytes=? $x_1 \ldots x_m$) returns #t if all arguments x_i are eqv?. Otherwise #f.

(bytes<? $x_1 \ldots x_m$) returns #t if the arguments are sorted in increasing lexigraphic order (the comparison of individual bytes are ordered by <). Otherwise #f.

[4] *Motile* was defined and implemented well before the endorsement of R7RS by the Scheme community. Future versions of *Motile* will likely conform to the R7RS conventions for bytevectors rather than the nomenclature of Racket Scheme.

(bytes>? $x_1 \ldots x_m$) returns #t if the arguments are sorted in decreasing lexigraphic order (the comparison of individual bytes are ordered by >). Otherwise #f.

7.4.9 Control Features

Procedure	Standard	Procedure	Standard	Procedure	Standard
procedure?	R5RS	apply	R5RS	map	R5RS
for-each	R5RS	call/cc	R5RS		

Motile deliberately omits eval as it is significant threat to island integrity and security.

7.5 Persistent Functional Data Structures

Persistent functional data structures erase the semantic differences between sharing and copying. A *persistent* data structure offers nondestructive update while a *functional* data structure omits any imperative side effects. Modern implementations of these data structures balance efficiency with structure sharing. Persistent functional structures are thread-safe, reduce memory consumption, reduce locking, and minimize the distinctions between intra- and inter-island messaging.

There are five principal persistent functional data structures: vectors, tuples, hash maps, unordered sets, and binding environments. Vectors and hash maps are modeled after the implementations of the comparable structures in Clojure (hash array mapped trie—HAMT) implementations of the structures invented by Bagwell [8]). Unordered set are also HAMTs and binding environments are implemented as a specialization of hash maps.

7.5.1 Persistent Vectors

A *Motile* persistent vector is a functional structure whose constituent values are indexed by a finite contiguous sequence of non-negative integers $0, \ldots, n-1$ for $n \geq 0$ (n is the *length* of the vector). Their implementation is modeled on the persistent vectors of Clojure[5]. Just as one can prepend a new pair p to the head of a list x without modifying x one can append a new element e to the end of a vector v without modifying v. In both cases the original structure (here list or vector) remains unchanged and a successor structure is returned as the result of the operation. To the extent possible the original vector and its successor share substructure to minimize needless copying.

[5] http://clojure.org/

`vector/null` is the persistent vector equivalent of the empty list `nil`. The length of the null (empty) vector is 0.

(`vector/persist?` x) returns `#t` if x is a persistent vector and `#f` otherwise.

(`vector/null?` x) returns `#t` is x is the null persistent vector and `#f` otherwise.

(`list-to-vector` l) where l is a list of length n returns a persistent vector x such that the length of x is n and each element x_i is `eq?` to e_i where $l = (e_0 \ldots e_{n-1})$. If list l is `nil` then `vector/null` is returned.

(`vector-to-list` x) returns a list $(x_0 \ldots x_{n-1})$ of the n elements of x.

(`vector/build` n f) returns a persistent vector x of length n where $x_i = (f\ i)$ for $0 \leq i < n$.

(`vector/cons` x v) appends value v to persistent vector x returning a persistent vector y such that:

- (`and` (`eq?` x_0 y_0) ... (`eq?` x_{n-1} y_{n-1})) is `#t` where n is the length of x
- $n+1$ is the length of y and
- (`eq?` y_n v) is `#t`

The contents of x are left unchanged.

(`vector/update` x i v) returns a persistent vector y identical to x with the exception that $y_i = v$. x is left unchanged and where feasible y shares common structure with x.

(`vector/length` x) returns the number of elements in x as an exact, non-negative integer.

(`vector/ref` x i) returns element x_i, $i = 0, \ldots, n-1$ where n is the length of x.

(`vector/cdr` x) returns a persistent vector y that omits the last element x_{n-1} of x for any non-empty persistent vector x of length $n > 0$.

(`vector/subvector` x i) returns a persistent vector y formed from the elements of x beginning with x_i and ending with x_{n-1} where $n > 0$ is the length of persistent vector x.

(`vector/subvector` x i j) returns a persistent vector y formed from elements of x beginning with x_i and ending with x_{j-1} for $i < j \leq n$ where n is the length of x.

(`vector/map` x f) returns a persistent vector y whose elements are (f x_0), ..., (f x_{n-1}) where n is the length of persistent vector x.

`(vector/filter` x p`)` returns a persistent vector y whose elements are those elements x_i (in index order) such that `(`p x_i`)` is not `#f`.

`(vector/fold/left` x f s`)` returns the value of folding persistent vector x from left to right over the function `(`f x_i v_i`)` where v_i is the value of the prior fold $i-1$. The seed value for fold 0 is s.

7.5.2 Tuples

A *Motile* tuple is an immutable, fixed-length, heterogeneous vector whose length and contents are fixed at creation time. Its constituent values are indexed by a finite set of contiguous non-negative integers $0,\ldots,n-1$ for $n \geq 0$ (n is the *length* of the tuple).

`(tuple` v_0 ... v_{n-1}`)` returns a tuple of length n whose elements, in index order, are the objects $v_0,\ldots,v_{n-1}$. It is the tuple analogue of `list`.

`(tuple?` x`)` returns `#t` if x is a tuple and `#f` otherwise.

`(tuple/build` f n`)`
Given a single argument function f and a non-negative integer n returns a tuple x of length n where $x_i = (f\ i)$, $i = 0,\ldots,n-1$.

`(tuple/length` x`)` returns the length $n \geq 0$ of tuple x.

`(tuple/ref` x i`)` returns value x_i of tuple x.

`(tuple-to-list` x`)` returns a list of the elements of tuple x in tuple order.

`(tuple-to-list` x i`)` returns a list `(`x_i ... x_{n-1}`)` of the elements of tuple x where n is the length of x.

`(tuple-to-list` x i j`)` returns a list `(`x_i ... x_{j-1}`)` of the elements of tuple x, $0 \leq i < j \leq n$.

`(list-to-tuple` x`)` returns a tuple x_0 ... x_{n-1} of length n where x is the list `(`x_0 ... x_{n-1}`)`.

`(tuple/copy` x i`)` returns a tuple whose elements are elements $x_i,\ldots,x_{n-1}$ of tuple x where n is the length of x. If $i = 0$ then `tuple/copy` returns x.

`(tuple/copy` x i j`)` returns a tuple whose elements are elements $x_i,\ldots,x_{j-1}$ of tuple x. If $i = 0$ and $j = n$, the length of x, then `tuple/copy` returns x.

`(tuple/append` x $y_1 \ldots y_n$`)` returns a tuple whose elements are the concatenation of tuples $x, y_1,\ldots,y_n$.

`(tuple/map` x f`)` returns a tuple whose elements are `(`f x_0`)`, ..., `(`f x_{n-1}`)` where n is the length of tuple x.

`(tuple/filter` x p`)` returns a tuple whose elements x_{i_0}, ..., x_{i_m}, where $i_0 < \ldots < i_m$, are exactly the elements of tuple x for which `(`p x_i`)` is not `#f`.

`(tuple/partition` x p`)` returns a pair `(`u `.` v`)` where u is a tuple containing, in index order, all elements x_i such that `(`p x_i`)` is not `#f` and v is a tuple containing, in index order, all elements x_j such that `(`p x_j`)` is `#f`. In other words it partitions x into two disjoint tuples, the first containing all of the elements of x that satisfy predicate p and the second containing all of the elements of x that do not satisfy p.

7.5.3 Persistent Hash Maps

A persistent hash map is an immutable collection of key/value pairs for which the keys are unique. Hash maps are distinguished by the predicate the map uses to test key equality: `eq?`, `eqv?`, or `equal?`.[6] After adding a key/value pair k/v to a persistent hash map h the result is always a successor hash map $h \setminus \{k/x\} \cup \{k/v\}$ where k/x is any key/value pair in h with key k (there is at most one). The predecessor hash map h is left unchanged.

`(hash/persist?` x`)` returns `#t` if x is a persistent hash map. Otherwise `#f`.

Every hash map has the same starting point, an immutable and *empty* hash map. There are three different empty hash maps, `hash/eq/null`, `hash/eqv/null` and `hash/equal/null`, whose names reflect the the map's predicate for key equality.

`(hash/null?` x`)` returns `#t` if x is one of `hash/eq/null`, `hash/eqv/null`, or `hash/equal/null`. Otherwise `#f`.

`(hash/eq?` x`)` returns `#t` if the equality predicate of hash map x is `eq?`. Otherwise `#f`.

`(hash/eqv?` x`)` returns `#t` if the equality predicate of hash map x is `eqv?`. Otherwise `#f`.

`(hash/equal?` x`)` returns `#t` if the equality predicate of hash map x is `equal?`. Otherwise `#f`.

`(hash/length` h`)` returns the total number of key/value pairs in hash map h.

`(hash/empty?` h`)` returns `#t` if hash map h is empty. Otherwise `#f`.

[6] Since two Scheme values can have the same print representation but fail to be eq? or eqv? two objects can "look" the same but be distinct (non-identical) from the perspective of the hash map predicate for object equality.

(hash/add h k_1 v_1 ... k_n v_n), given a hash map h, returns the successor hash map

$$h \cup \{k_1/v_1, \ldots, k_n/v_n\}$$

adding the pairs k_i/v_i in sequence order. If the same key k appears in the list of key/value pairs more than once then the last pair k/v_i in the sequence appears in the new hash map. hash/new can be used to create new hash tables (when h is one of hash/eq/null, hash/eqv/null, or hash/equal/null) or to derive an update of a non-empty hash table h.

(list-to-hash h (k_1 v_1 ... k_n v_n)) is equivalent to (hash/new h k_1 v_1 ... k_n v_n).

(pairs-to-hash h ((k_1 . v_1) ... (k_n . v_n))) is equivalent to (hash/new h k_1 v_1 ... k_n v_n).

(hash/cons h k v) returns the successor hash map $h \cup \{k/v\}$ where h is a hash map and k/v a key/value pair.

(hash/remove h k) returns the successor hash map $h \setminus \{k/v\}$ when $k/v \in h$ for some value v. Otherwise returns h.

(hash/merge h g_1 ... g_n) returns the successor hash map that is the merge of hash maps $g_1, \ldots, g_n$ *into* hash map h. Any key/value pair k/v in g_i whose key k appears in h is replaced with k/v in the successor hash map. Any key/value pair k/v in g whose key k does not appear in h is added to the successor hash map. The merge proceeds in sequence order and for any key k appearing in multiple binding environments g_i the key/value pair k/v in the rightmost g_i will supplant all prior pairs with key k in the successor hash map.

(hash-to-list h) returns a flat list (k_1 v_1 ... k_n v_n) of all key/value pairs k_i/v_i in hash map h. The key/value pairs are not in any particular order.

(hash-to-pairs h) returns an association list ((k_1 . v_1) ... (k_n . v_n)) of all key/value pairs k_i/v_i in hash map h. The key/value pairs are not in any particular order.

(hash/keys h) returns a list (k_1 ... k_n) of all keys in hash map h. The keys are not in any particular order.

(hash/values h) returns a list (v_1 ... v_n) of all values in hash map h. The values are not in any particular order and the same value may appear more than once.

(hash/ref h k f) returns the value indexed by key k in hash map h. If key k does not appear in h then the failure value f is returned instead.

(hash/car h) returns the "first" key/value pair as (k . v) in non-empty hash map h. "First" here has no particular meaning; it is merely the first k/v pair appearing in an implementation-dependent walk of the hash map. If hash map h is empty then an error is raised.

(hash/cdr h) returns the hash map $h \setminus \{k/v\}$ where k/v is the first key/value pair appearing in an implementation-dependent walk of the hash map. In other words, hash/cdr returns the "rest" of a hash map in the sense that cdr returns the "rest" of a list. If h is empty then an error is raised. hash/cdr is equivalent to (hash/remove h (car (hash/car h))) but far more efficient.

(hash/contains? h k) returns #t if hash map h contains key k. Otherwise #f.

(hash/fold h f s) folds function f over the key/value pairs of hash map h and returns the value of the final call of f. f is a function of three arguments: a key, its affiliated value, and the current seed. In the very first call to f the seed is s. In all later calls the seed is the value of the immediate prior call to f. f is called exactly once for every key/value pair in h and the presentation order of the key/value pairs to f is implementation-dependent.

(hash/map h f) applies function f to every key/value pair in hash map h and returns a hash map $g = \{k'/v' \mid (k' \text{ . } v') = (f\ k\ v), k/v \in h\}$. The equality predicate of g is that of h and the presentation order of key/value pairs to f is implementation-dependent. hash/map is the hash equivalent of map for lists.

(hash/for-each h f) applies function f, as (f k v) to every key/value pair $k/v \in h$ for the sake of the side-effects of f. hash/for-each is the hash equivalent of for-each for lists.

(hash/filter h p) applies predicate p, as (p k v) to every key/value pair $k/v \in h$ and returns a hash map $g = \{k/v \mid (p\ k\ v) \neq \#\mathtt{f}, k/v \in h\}$. The equality predicate of g is the equality predicate of h and the presentation order of key/value pairs to p is implementation-dependent.

(hash/partition h p) returns a pair (g_t . g_f) of hash tables where

$$g_t = \{k/v \mid (p\ k\ v) \neq \#\mathtt{f}, k/v \in h\} \text{ and } g_f = \{k/v \mid (p\ k\ v) = \#\mathtt{f}, k/v \in h\}$$

The equality predicates of g_t and g_f are the equality predicate of h. The presentation order of key/value pairs to p is implementation-dependent.

(hash/and h p) returns #t if (p k v) is not #f for all $k/v \in h$. Otherwise #f is returned. This combinator performs short-circuit evaluation and returns #f immediately if (p k v) is #f for any $k/v \in h$. The presentation order of key/value pairs to p is implementation-dependent.

(hash/or h p) returns #t if (p k v) is not #f for any $k/v \in h$. Otherwise #f is returned. This combinator performs short-circuit evaluation and returns #t immediately if (p k v) is not #f for any $k/v \in h$. The presentation order of key/value pairs to p is implementation-dependent.

7.5.4 Persistent Unordered Sets

A persistent unordered set is an immutable collection of distinct objects. Unordered sets are distinguished by the predicate the set uses to test object equality: eq?, eqv?, or equal?.[7] Since sets are persistent all updates are nondestructive and leave the original set unchanged.

(set/persist? x) returns #t if x is a persistent set. Otherwise #f.

Every unordered set has the same starting point, an immutable and *empty* unordered set. There are three different empty sets, set/eq/null, set/eqv/null and set/equal/null, whose names reflect the the set's predicate for object equality.

(set/null? x) returns #t if x is one of set/eq/null, set/eqv/null, or set/equal/null. Otherwise #f.

(set/eq? s) returns #t if the equality predicate of set s is eq?. Otherwise #f.

(set/eqv? x) returns #t if the equality predicate of set s is eqv?. Otherwise #f.

(set/equal? s) returns #t if the equality predicate of set s is equal?. Otherwise #f.

(set/length h) returns the total number of objects in set s.

(set/empty? s) returns #t if set set s is empty. Otherwise #f.

(set/contains? s o) returns #t if $o \in s$. Otherwise #f.

[7] Since two Scheme values can have the same print representation but fail to be eq? or eqv? two objects can "look" the same (have the same print representation) but be distinct (non-identical) from the perspective of the setpredicate for object equality.

(set/add s o_1 ... o_n), given a set s, returns the successor set $s \cup \{o_1, \ldots o_n\}$. set/add can be used to create nonempty sets (when s is one of set/eq/null, set/eqv/null, or set/equal/null) or to add objects to a non-empty set s. The equality predicate of the successor set is the equality predicate of s.

(set/remove s o_1 ... o_n) returns the successor set $s \setminus \{o_1, \ldots, o_n\}$.

(set/union s_1 s_2 ... s_n) returns $s_1 \cup s_2 \ldots \cup s_n$. The equality predicate of the union is the equality predicate of s_1.

(set/intersection s_1 s_2 ... s_n) returns $s_1 \cap s_2 \ldots \cap s_n$. The equality predicate of the intersection is the equality predicate of s_1.

(set/difference s_1 s_2) returns $s_1 \setminus s_2$. The equality predicate of the difference is the equality predicate of s_1.

(set/difference s_1 s_2 ... s_n) returns $s_1 \setminus (s_2 \cup \ldots \cup s_n)$. The equality predicate of the difference is the equality predicate of s_1.

(set/subset? s_1 s_2) returns #t if $s_1 \subseteq s_2$. Otherwise #f.

(set/superset? s_1 s_2) returns #t if $s_1 \supseteq s_2$. Otherwise #f.

(set-to-list s) returns a list of the objects contained in set s. The ordering of the objects in the list is implementation-dependent.

(set/fold s f i) folds function f over the objects in set s and returns the value of the final call of f. f is a function of two arguments: an object and the current seed —in the first call to f the seed is i. In all later calls the seed is the value of the immediate prior call to f. f is called exactly once for every object in s and the presentation order of the objects to f is implementation-dependent.

(set/for-each s f) applies function f, as (f o), to every object $o \in s$ for the sake of the side-effects of f. set/for-each is the set equivalent of for-each for lists.

(set/filter s p) applies predicate p, as (p o) to every object $o \in s$ and returns a set $t = \{o \mid (p\ o) \neq \#\mathtt{f}, o \in s\}$. The equality predicate of t is the equality predicate of s and the presentation order of objects to p is implementation-dependent.

`(set/partition` s p`)` returns a pair `(`r_t `.` r_f`)` of sets where

$$r_t = \{o \mid (p\ o) \neq \texttt{\#f}, o \in s\} \text{ and } r_f = \{o \mid (p\ o) = \texttt{\#f}, o \in s\}$$

The equality predicates of r_t and r_f are the equality predicate of s The presentation order of objects to p is implementation-dependent.

`(set/and` s p`)` returns `#t` if `(`p o`)` is not `#f` for all $o \in s$. Otherwise `#f` is returned. This combinator performs short-circuit evaluation and returns `#f` immediately if `(`p o`)` is `#f` for any $o \in s$. The presentation order of objects to p is implementation-dependent.

`(set/or` h p`)` returns `#t` if `(`p o`)` is not `#f` for any $o \in s$. Otherwise `#f` is returned. This combinator performs short-circuit evaluation and returns `#t` immediately if `(`p o`)` is not `#f` for any $o \in s$. The presentation order of objects to p is implementation-dependent.

7.5.5 Binding Environments

Motile is one of the few languages for which *binding environments* are first class objects.[8] A binding environment is a map whose keys are identifiers (Scheme symbols) and whose values are arbitrary *Motile* values. A single identifier/value pair α/v is a *binding* in *Motile* parlance. The notation $\alpha/v \in b$ means that b contains the binding α/v; $\alpha \in b$ denotes that for some v, $\alpha/v \in b$; and $\alpha \notin b$ means that α does not appear among the keys of b.

A binding environment is a functional persistent structure; each is immutable and every update is nondestructive. Augmenting, updating, or restricting a binding environment b always produces a successor binding environment b' reflecting the change while leaving b unmodified. If $\alpha/u \in b$ and binding α/v is added to b then $\alpha/v \in b'$ and b is left unchanged with $\alpha/u \in b$. If $\alpha \notin b$ and binding α/v is added to b then $\alpha/v \in b'$ and $\alpha \notin b$.

There is a distinguished empty binding environment `environ/null` that is used as the starting point for the construction of all binding environments.

Motile draws a sharp distinction between lexical scope and global scope. Global scope contains the ambient procedures, functions, and constants on which programs rely, for example: `cons` for list

[8] Other examples include Symmetric Lisp [102], Bla [181], MAST [255], and MIT Scheme [173].

construction, `car` and `cdr` for list deconstruction, or + and − for arithmetic to name but a few. Every *Motile* expression is evaluated in the context of some explicit *binding environment*, B. Over the course of the execution of a *Motile* closure its individual constituent expressions $\epsilon_1, \ldots, \epsilon_n$ may be evaluated in several distinct binding environments $B_1, \ldots, B_m$. Unlike many other Scheme implementations, the *Motile* global binding environment is explicit (rather than implicit), a consequence of the COAST **Execution** rule — closures are evaluated in the context of an execution site $\langle E, B \rangle$ where E is an execution engine and B a global binding environment.

There are four *Motile* special forms[9] devoted to binding environments:

- `environ/get` obtains the value affiliated with an identifier in a given binding environment.
- `environ/add` augments a binding environment with one or more bindings to obtain a modified or enlarged binding environment.
- `environ/reflect` evaluates expressions in the context of a given binding environment.
- `environ/delete` eliminates bindings from a binding environment to obtain a more restricted binding environment.

These special forms accept literal symbols that denote the names of bindings (denoted as α) or literal expressions (denoted as ϵ) as might appear in the body of a `(lambda () ...)`.

`(environ/get` b α u`)` returns value v if $\alpha/v \in b$ otherwise value u is returned. α is a literal (unquoted) symbol.[10]

`(environ/get` b_0 `(`α_1 $\ldots$ α_n`)` u`)` interprets a list of literal symbols `(`α_1 $\ldots$ α_n`)` as a "path" name for a binding as:

$$b_0 \xrightarrow{\alpha_1} b_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} b_{n-1} \xrightarrow{\alpha_n} v$$

where:

- $b_1, \ldots, b_{n-1}$ are binding environments
- $b_i =$ `(environ/get` b_{i-1} α_i `#f)`, and
- $v =$ `(environ/get` b_{n-1} α_n u`)`

[9] Special forms are implemented directly by the compiler.

[10] Consequently, `environ/get` cannot be used, for example, in a loop to explore the contents of a binding environment. This is a security feature to frustrate efforts by visiting mobile code to discover the contents of any binding environments that they encounter.

If any α_i of the path fails to resolve then u is returned instead. Note that `(environ/get` b `(`α`)` u`)` is equivalent to (though less efficient than) `(environ/get` b α u`)`.

`(environ/add` b α_1 ... α_n`)` returns successor environment b' that contains the bindings in b plus the bindings of identifiers $\alpha_1,\ldots,\alpha_n$ appearing in the lexical scope of the call to `environ/add`.[11] The bindings $\{\alpha_1/v_1,\ldots,\alpha_n/v_n\}$ are added in sequence order.[12] If $b =$ `environ/null` then the successor environment b' contains just the bindings $\alpha_1/v_1,\ldots,\alpha_n/v_n$ and nothing more.

`(environ/reflect` b ϵ_1 ... ϵ_n`)` evaluates expression `(begin` ϵ_1 ... ϵ_n`)` in the context of the binding environ b and returns the value. In other words, b is the global binding environment for `(begin` ϵ_1 ... ϵ_n`)`.[13]

`(environ/delete` b α_1 ... α_n`)` returns a successor binding environment that contains all of the bindings of b with the exception of those with names $\alpha_1,\ldots,\alpha_n$.

The remaining functions for binding environments are available as procedures in island-supplied binding environments. The *baseline* binding environment provides just two bindings: `environ/null` and `environ/merge`.

`environ/null` is the canonical empty binding environment.

`(environ/merge` b b_1 ... b_n`)` returns the successor environ b' that is the merge of environs $b_1,\ldots,b_n$ into environ b. If $\alpha_1/v_1,\ldots\alpha_m/v_m$ is the contents of binding environment b_1 then

`(environ/merge` b b_1`)`

is equivalent to

`(let ((`α_1 v_1`) (`α_2 v_2`)`$\cdots$`(`α_m v_m`)) (environ/add` b α_1 α_2 $\cdots$ α_m`))`

[11] The *Motile* compiler will raise a compile-time error if an α_i does not appear within the lexical scope of the call to `environ/add`.

[12] Because the α_i are literal symbols the *Motile* compiler knows (at compile-time) exactly which bindings are being added to b. This prevents *Motile* code from performing automated sweeps of lexical scope bindings and eases tracking the propagation of functional capability.

[13] Any expression ϵ_i in a call to `environ/reflect` must contain variables that either lie in lexical scope or are available as bindings in b. This makes it more difficult (though not impossible) for visiting mobile code executed in the body of an `environ/reflect` to systematically explore the bindings of environ b.

and

$$\texttt{(environ/merge } b\ b_1 \cdots b_n\texttt{)}$$

is equivalent to

$$\texttt{(environ/merge (environ/merge } b\ b_1 \cdots b_{n-1}\texttt{) } b_n\texttt{)}$$

In other words, the merge proceeds in sequence order and for any name α appearing in multiple binding environments b_i the binding pair α/v in the rightmost b_i will supplant all prior binding pairs with name α in the successor environ.

Other functions that may be available in some binding environments are: `environ/length`, `environ/capture`, `environ/map`, `environ/and`, and `environ/or`.

`(environ/length` b`)` returns the total number of bindings in binding environment b.

`(environ/capture)` returns the current global binding environment in effect at the point of call.[14]

`(environ/map` $b\ f$`)` applies f to each binding $\alpha/v \in b$, as `(`$f\ \alpha\ u$`)`, in an unspecified order and accumulates the results in a list.

`(environ/and` $b\ p$`)` applies predicate p to each binding $\alpha/v \in b$,as `(`$p\ \alpha\ u$`)`, in an unspecified order. It returns `#t` if `(`$p\ \alpha\ u$`)` is `#t` in each case and `#f` otherwise. `environ/and` performs short-circuit evaluation and will return `#f`as soon as any `(`$p\ \alpha\ u$`)` evaluates to `#f`.

`(environ/or` $b\ p$`)` applies predicate p to each binding $\alpha/v \in b$, as `(`$p\ \alpha\ u$`)`, in an unspecified order. It returns `#t` if `(`$p\ \alpha\ u$`)` is `#t` in at least one case and `#f` otherwise. `environ/and` performs short-circuit evaluation and will return `#t` as soon as any `(`$p\ \alpha\ u$`)` evaluates to `#t`.

7.6 Syntax

Here I sketch the syntax of *Motile*—a subset of R5RS [143] plus a few constructions borrowed from R7RS [216]. Nonterminals (anything of the form ⟨xyz⟩) not defined here borrow their definitions from R7RS.

[14] There is a strong security argument for omitting `environ/capture` from the baseline binding environment as it allows visiting mobile code to discover, with the assistance of a malevolent island, the contents (in the form of the binding names) of the global binding environment assigned to its execution site. While no global binding environment available to visiting mobile code should contain `environ/map` (which allows one to trivially construct a list of the binding names) visiting mobile code can easily transmit the binding environment returned by `(environ/capture)` to an island service specifically constructed for the logging and/or analysis of the names of the bindings.

7.6.1 External Representations

A ⟨datum⟩ is anything that the `read` procedure can parse. Note that any string that parses as an ⟨expression⟩ will also parse as a ⟨datum⟩.

⟨datum⟩	⟶	⟨simple datum⟩ | ⟨compound datum⟩
⟨simple datum⟩	⟶	⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨character string⟩ | ⟨byte string⟩ | ⟨symbol⟩
⟨symbol⟩	⟶	⟨identifier⟩
⟨compound datum⟩	⟶	⟨list⟩ | ⟨vector⟩
⟨list⟩	⟶	(⟨datum⟩*) | (⟨datum⟩$^+$. ⟨datum⟩) | ⟨abbreviation⟩
⟨abbreviation⟩	⟶	⟨abbreviation prefix⟩ ⟨datum⟩
⟨abbreviation prefix⟩	⟶	'|`| ,| ,@
⟨vector⟩	⟶	#(⟨datum⟩*)

Motile is implemented as a transpiler, consequently it relies on the underlying Scheme host for datums. However, *Motile* the language is single-assignment, meaning that the simple datums, such as ⟨character string⟩ and ⟨byte string⟩, as well as the compound datums ⟨list⟩ and ⟨vector⟩, are immutable in *Motile* programs.

The notations '⟨datum⟩,`⟨datum⟩, ,⟨datum⟩, and ,@⟨datum⟩ denote two-element lists whose first elements are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing` respectively.

7.6.2 Expressions

The expression syntax is essentially R5RS[15] with a few elements of R7RS, specifically `when`, `unless`, and `letrec*`. Unless explicitly noted otherwise in the grammar below all elements reflect R5RS.[16]

As a matter of convention a ⟨command⟩ is a procedure that does not return a useful value to its continuation.

[15] Although `case` is not implemented.

[16] The grammar itself is liberally cribbed from R7RS [216].

⟨expression⟩ ⟶ ⟨identifier⟩

| ⟨literal⟩

| ⟨procedure call⟩

| ⟨lambda expression⟩

| ⟨conditional⟩

| ⟨environ expression⟩

| ⟨derived expression⟩

⟨literal⟩ ⟶ ⟨quotation⟩ | ⟨self-evaluating⟩

⟨self-evaluating⟩ ⟶ ⟨boolean⟩ | ⟨number⟩ | ⟨character⟩ | ⟨string⟩

⟨quotation⟩ ⟶ '⟨datum⟩ | `(quote` ⟨datum⟩`)`

⟨procedure call⟩ ⟶ `(`⟨operator⟩ ⟨operand⟩*`)`

⟨operator⟩ ⟶ ⟨expression⟩

⟨operand⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ (lambda ⟨formals⟩ ⟨body⟩)

⟨formals⟩ ⟶ `(`⟨identifier⟩*`)` | ⟨identifier⟩ | `(`⟨identifier⟩$^{+}$. ⟨identifier⟩`)`

⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩

⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩

⟨command⟩ ⟶ ⟨expression⟩

⟨identifier⟩ ⟶ See Section 7.1.1, page 62 of [216]

⟨conditional⟩	⟶	⟨if⟩ | ⟨when⟩ | ⟨unless⟩	
⟨if⟩	⟶	`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`	
⟨when⟩	⟶	`(when` ⟨test⟩ ⟨sequence⟩`)`	*R7RS*
⟨unless⟩	⟶	`(unless` ⟨test⟩ ⟨sequence⟩`)`	*R7RS*
⟨test⟩	⟶	⟨expression⟩	
⟨consequent⟩	⟶	⟨expression⟩	
⟨alternate⟩	⟶	⟨expression⟩	
⟨environ expression⟩	⟶	`(environ/cons` ⟨environ⟩ identifier$^+$`)`	*Motile*
		| `(environ/ref` ⟨environ⟩ ⟨identifier⟩ ⟨substitute⟩`)`	*Motile*
		| `(environ/reflect` ⟨environ⟩ ⟨sequence⟩`)`	*Motile*
		| `(environ/remove` ⟨environ⟩ identifier$^+$`)`	*Motile*
⟨environ⟩	⟶	⟨expression⟩	
⟨substitute⟩	⟶	⟨expression⟩	

⟨derived expression⟩ ⟶ `(cond` ⟨cond clause⟩$^+$`)`

| `(cond` ⟨cond clause⟩* `(else` ⟨sequence⟩`))`

| `(and` ⟨test⟩*`)`

| `(or` ⟨test⟩*`)`

| `(let (`⟨local binding⟩*`)` ⟨body⟩`)`

| `(let` ⟨identifier⟩ `(`⟨local binding⟩*`)` ⟨body⟩`)`

| `(let*` `(`⟨local binding⟩*`)` ⟨body⟩`)`

| `(letrec` `(`⟨local binding⟩*`)` ⟨body⟩`)`

| `(letrec*` `(`⟨local binding⟩*`)` ⟨body⟩`)` *R7RS*

| `(begin` ⟨body⟩`)`

| `(do (`⟨iterator⟩*`)` `(`⟨test⟩ ⟨do result⟩`)` ⟨command⟩*`)`

| ⟨quasiquotation⟩

⟨cond clause⟩ ⟶ `(`⟨test⟩ ⟨sequence⟩`)`

| `(`⟨test⟩`)`

| `(`⟨test⟩ `=>` ⟨recipient⟩`)`

⟨recipient⟩ ⟶ ⟨expression⟩

⟨local binding⟩ ⟶ `(`⟨identifier⟩ ⟨expression⟩`)`

⟨iterator⟩ ⟶ `(`⟨identifier⟩ ⟨start⟩ ⟨step⟩`)`

| `(`⟨identifer⟩ ⟨start⟩`)`

⟨start⟩ ⟶ ⟨expression⟩

⟨step⟩ ⟶ ⟨expression⟩

7.6.3 Definitions

⟨definition⟩ ⟶ `(define` ⟨identifier⟩ ⟨expression⟩`)`

| `(define (`⟨identifier⟩ ⟨define formals⟩`)` ⟨body⟩`)`

| `(define-macro (`⟨identifier⟩ ⟨define formals⟩`)` ⟨body⟩`)`

⟨define formals⟩ ⟶ ⟨identifier⟩* | ⟨identifier⟩$^+$. ⟨identifier⟩

Chapter 8: Evaluation: Live Update

To demonstrate that COAST is *sufficiently expressive to implement adaptive decentralized services* I consider two challenge problems. The first, *live update*, the practice of modifying a service in place without interrupting the service[1], is presented here. The second, *client-driven service APIs*, is presented in Chapter 9.

Even under a single sphere of authority the *simultaneous* update of a set of related services is often impractical if not infeasible. In a decentralized system, simultaneous update may be impossible, as the loosely aligned needs, goals, and timelines of multiple, distinct organizations may leave incremental updates as the only path forward. Further, an organization may be forced to update services in response to immediate legal, financial, security, regulatory, or consumer pressures. Here, incremental and dynamic service update may be the wisest course; repairing and improving while preserving, without change, the bulk of the existing service structure.

Live update[2] modifies the code, structures, and data values of a running system in place without halting the system or interfering with it [125]. Repairing bugs, upgrading features, inserting monitors and probes for engineering analysis, patching security exploits, tuning performance, logging, auditing, and early warning are facts of life for vital services. Live update strives to minimize service interruptions even as the software is modified. Architectural styles that ease the complexity and risk of live update for large-scale systems can be beneficial.

In section 8.1 I present the evaluation criteria for the two live update problems posed here. Section 8.2 illustrates and evaluates a live update for the faulty binding environment of an island execution site. Section 8.3 revisits the problem of the previous section using "fire and forget" CURLs, a form of service invocation that is unique to COAST. Section 8.4 briefly turns to more general forms of live update where islands are provisioned with live update in mind, suggesting how the COAST rules can secure the broad deployment of island services specific to live update.

[1] Interruption is in the eye of the client. There will a brief delay, from the perspective of the client, while the service is undergoing update but no outstanding service requests will be lost and clients can continue to issue service requests over the course of the update.

[2] The technical phrase *dynamic software update* (DSU) also appears frequently in the literature.

Section 8.5 tackles four distinct forms of service live update: in place (without hot backup), intra-island with hot backup, in place with hot backup, and inter-island with hot backup. The latter three, which implement transparent service recovery in the event that the update fails, are a gold standard for live update. Section 8.6 compares the COAST solutions to other work in system-level live update. Finally, section 8.7 summarizes the contributions of the COAST architectural style to live update for decentralized services.

8.1 Evaluation Criteria

We know that architectural structure can aid dynamic updates. Early examples of dynamic software adaptation drawn from architecture include the software buses Polylith [195, 196] and FIELD [202], the dynamic data flow environment Weaves [112], and C2, an architecture for event-based components and connectors [241].

How can we be assured that COAST-compliant systems will be adaptive? The evaluation criteria are rooted in the observations of Oreizy and Taylor [185]:

- Architectural style can have a pervasive influence on the ease and range of dynamic adaptation for systems constructed in the style; and
- Specific elements of a style contribute to the ease (or difficulty) of dynamic adaptation.

From this perspective a style is *sufficiently expressive for adaptive decentralized services* to the extent that the style rules contribute to (or hinder) dynamic adaptations that are likely to arise in practice.

Taylor, Medvidovic, and Oreizy [239] examine the roles of architectural style in implementations of live update and present a unified model, BASE (*Behavior*, *Asynchrony* of change, *State*, and *Execution* context), for evaluating the innate capacity of an architectural style for dynamic adaptation. They observe that styles amenable to live update provide the means to:

- Identify the system elements subject to change;
- Manipulate those elements;
- Restrain or redirect system interactions with those elements; and
- Extract, transform, and inject system state that pertains to those elements.

They also isolate two strategies that appear repeatedly in such styles: (1) malleable bindings and (2) explicit messages/events in communication (both of which are embraced in COAST).

The BASE model has four aspects:

Behavior: What aspects of system behavior can be changed? Can new behaviors be added and existing behaviors replaced or deleted? How are behavioral changes instigated and deployed? To what extent are behavioral changes confined in extent, place, or duration?

Asynchrony of change: Is system execution suspended during change or does it continue to run in some capacity? If the latter, how are partial changes dealt with? Can multiple changes occur simultaneously?

State: How is state captured and transformed for update? At what point are state updates applied? Are state updates synchronized across the system? Is state updated lazily as it is accessed?

Execution context: System execution may proceed by virtual machine, interpreter, physical processor, just-in-time compiler, or a combination of such. Updates must be mindful of the engine of execution and the execution representation to ensure that that the updates do not introduce errors or unwanted effects.

COAST has much in common with its predecessor Computational REST (CREST) [69, 70] and whose BASE factoring appears in [239]. Both COAST and CREST embrace computation exchange as a fundamental interaction among peers. Both permit peers to behave as service providers and service consumers simultaneously. Both employ execution sites as flexible sandboxes for confining the resource, functional, and communication capability of visiting computations. However, they differ in two critical respects: CURLs and communication by introduction.[3]

While CREST had a generalization of URLs for naming computations those URLs were not cryptographic structures, as are CURLs. Further, under CREST there was no hint of communication by introduction and CREST could not dictate patterns of communication. Under COAST explicit control of communication is woven into the fabric of the style. A CREST URL names either a computation or an

[3] Overall, the security posture of COAST is far more aggressive and comprehensive than that of CREST. *Motile* treats binding environments as a first line defense to confine functional capability and, in general, the binding environments of its execution sites are far more confined than were those of CREST. In part this is attributable to additional experience in confining visiting computations and to the *Motile* implementation of binding environments, where multiple binding environments efficiently share common subsets of bindings. Sharing encourages greater variety among a family of binding environments while minimizing memory consumption.

execution site. A COAST CURL names an ingress point while the communication model underlying COAST (section 3.2) severs any connection between computation and communication. Hence computations can remain anonymous and service providers are free to rearrange the linkage between communication and computation to suit provider needs (say for live update).[4]

That same communication model allows COAST service providers and consumers to enforce arbitrary contracts on communications including stateful constraints (section 3.2). CURLs are tamper-proof and cannot be forged or guessed. While both styles permit arbitrary metadata in their communication names (URLs for CREST and CURLs for COAST), CREST could not guarantee the integrity of that metadata, leaving open a door to a broad class of attacks. COAST, on the other hand, guarantees the integrity of a CURL and its contents. In consequence, CURL metadata can safely carry island-generated mobile code for the purposes of contract enforcement, service definition, traffic tracing, service accounting, and other service- and client-specific functions.

Taylor and company acknowledge that CREST was an elastic architectural style for dynamic adaptation. We expect no less from COAST, a successor to CREST, which incorporates the elements (mobile code, messaging, execution sites, and peers) that account for the plasticity of CREST. With respect to the BASE framework:

Behavior: COAST peers (islands) may join and depart arbitrarily. Computations (state + code) are exchanged for execution among services (themselves computations) allowing for arbitrary run-time behavior (in principle) among islands. The dispatch of closures and continuations is the mechanism by which behavior is deployed and changed. The lifespan of a computation may vary from milliseconds to months and computations can be added, replaced, or killed. Computations may be executed in any execution site for which a client holds an appropriate CURL. A single computation may dispatch multiple computations to multiple execution sites on multiple islands. A single island may host multiple computations originating from the same island. In general there are many-to-many relationships among islands and the computations residing on those islands.

Asynchrony of change: By definition of the COAST style the only interaction among computations is asynchronous messaging. There is *no other* means by which computations can interact. Update

[4] A CURL names the ingress point $t\bullet$ of a transport t. A client, holding a CURL u, has no idea of which computation (and there may be more than one) holds an egress point $t\triangleright$ of t and is therefore capable of receiving a message sent via $t\bullet$.

is also asynchronous to the extent that the inter-computation relationships permit asynchronous update. Inter-dependence among computations does not necessarily preclude asynchronous update. Though the details are system-specific there is nothing in the style that prevents multiple simultaneous updates.

State: All state transfers among COAST computations are explicit in the messages exchanged among computations: the state is contained in ordinary values, the lexical scope bindings of closures, the bindings of binding environments, or the stack frames and heap references of continuations. All state transfers are immutable, which simplifies reliable state capture. The style does not make explicit provision for state capture; however, service design that is sensitive to the requirements of live update can reduce state capture and state transform to a simple deterministic procedure. State updates can be applied at points within a service where state quiescence is guaranteed. State updates can be synchronized across a system though this is both system- and update-dependent.[5] Lazy state update is not addressed in the style (though nothing in the style prevents or forbids it).

Execution context: Execution context is always explicit, namely the execution site $\langle E, B\rangle$ of a computation. The smallest unit of update is a single closure and all visiting mobile code is "recompiled" for its target execution site. It may be the case that a free variable α in an update closure has no binding $\alpha/v \in B$, in which case the update will fail post-restart if α is ever referenced.[6]

The BASE evaluation above indicates that COAST is hospitable to live update. Several aspects of COAST and *Motile*/*Island* contribute to this conclusion:

- *Execution sites are a natural locus of live updates.* Computation exchange suggests that execution sites are the locus of service provisioning and we see that reflected in the implementation of remote evaluation and spawning (section 5.8). The binding environments B of execution sites $\langle E, B\rangle$ are natural targets for live update when the scale of the update is at most a small number of bindings $\alpha_i/g_i \in B$.
- *A nanoservice is a natural target for live update.*[7] COAST supports an architecture of cooperating,

[5] In the most general case simultaneous live update across multiple spheres of authority may require decentralized barrier synchronization. Decentralized barrier synchronization can be implemented using a dedicated (but ephemeral) service b as the barrier. Each service s_i undergoing update transmits the resolver r_{s_i} of an s_i-specific promise p_{s_i} to b (via a CURL u_b for b) and blocks, waiting on the resolution of the promise. When all s_i have delivered a resolver to the barrier b resolves all of the promises p_{s_i} to a value indicating success. Timeout for failure is easily added, the requisite primitives are available in *Motile*.

[6] The unresolved free variable could be detected by the *Island* deserializer in cooperation with the *Motile* recompiler. This has not been implemented although the hooks are available in both the deserializer and the recompiler.

[7] I use the term "nanoservice" to denote a *single Motile* islet that provides a service or side effect on behalf of a client. A

isolated, and independent nanoservices that are orchestrated by clients (who themselves may be collectives of nanoservices). This view of service design, construction, and deployment favors live update on the scale of an individual nanoservice for both clients and providers.

- *Message flow is a natural target for achieving state quiescence.* Manipulating message flow is a powerful mechanism to ensure state quiescence, a necessary, but not sufficient condition, for state capture, state transformation, and safe restart — each essential for reliable and scalable live update.[8]
- *The Motile/Island reference implementation of COAST is "turtles all the way down."* The internal architecture of *Island* adheres to the COAST style and the communication mechanisms that govern inter-island communication apply in equal measure to the intra-island communications of co-resident islets (that is, nanoservices, implemented as green threads, residing on the same island). In fact, all of the internal island services (such as resource allocation, message routing and delivery, remote evaluation, spawning, and reaping dead or expired services) are themselves implemented as nanoservices bound together by patterns of immutable-message exchange. In short, the reference implementation of *Motile*/*Island* is "turtles all the way down" and, consequently itself a promising target for live update. To my knowledge COAST is one of the few architectural styles for which an implementation of the style conforms to the style.
- *COAST frameworks may reduce the difficulty of constructing and deploying systems that are live update ready.* Higher-order COAST service constructions, that is, services comprising a collective of cooperating services, can safely accommodate live update provided that the live update respects the underlying communication topology and semantics of the service. Live update can be woven, by design, into the service frameworks for common patterns of organization and communication among nanoservices; for example, the pattern of a reverse proxy that distributes incoming requests to a collection of identical, but independent, services $s_1, \ldots, s_m$.[9]

nanoservice may be one-shot (for example, a remote evaluation), issued solely for: (1) its side effect, for example, a remote evaluation whose thunk (`lambda ()` ($f \cdots$)) is effectively an RPC; (2) the return value of the thunk; or (3) both. Alternatively, a nanoservice may be long-lived (a remote spawn) and may respond to requests issued elsewhere. Finally, a service provider may perform a local spawn to create a nanoservice for internal use; local use by islets created on-island by remote evaluation or remote spawn; or remote use by clients elsewhere in the network.

[8] The server framework of the Erlang OTP library [158, 186] is an example of exploiting message flow for the sake of live update. *Motile*, like Erlang, is a single assignment, multi-threaded language with immutable messages and many of the design elements in the OTP frameworks have direct analogues in *Motile*/*Island*. Hicks ([125], section B.3.7, page 205–206) calls out Erlang as a hospitable target for dynamic update and attributes this to two language features: all data is immutable (write-once) and all thread communication occurs via message passing. Both features are requirements of the COAST style.

[9] Again, the Erlang OTP server framework is an industrial-strength implementation of framework-based live update.

- *COAST provides essential security for live update.* As attractive as live update might be for the maintenance and upgrade of nonstop systems it is also a potent tool for attack by which malicious agents may introduce subtle flaws or security back doors into critical software infrastructure. COAST, driven by the exigencies of decentralized systems, offers general, powerful, and adaptive security mechanisms to reduce the risk of damage due to innocent error, oversight, or deliberate malicious attack. These same mechanisms can obstruct, thwart, or prevent malicious attempts at live update. To my knowledge this contribution to the security of live update is novel.

8.2 Exploiting Binding Environments for Live Update

The binding environment B of an execution site $\langle E, B\rangle$ can play an important role in service-side updates where the update is limited to a few bindings within B. For example, many minor faults can be remedied by wrapping a binding $\alpha/g \in B$ (where g is the faulty implementation) with a layer of adaptive code that repairs the fault.

```
(curl/remote                                                              1
 Delta@ ; CURL for live update at island I.                               2
 (lambda ()                                                               3
   (let* ((B (island/environ/lookup ...)) ; Obtain defective environ B.   4
          (g (environ/get B alpha #f)) ; Fetch defective closure g from B. 5
          (repair                                                         6
           (lambda (x) ; Transforms vector argument to list.              7
             (g (if (vector? x) (vector-to-list x) x))))                  8
          (S (environ/add B alpha repair))) ; Successor environ.          9
     ...))) ; Stage successor environ S for deployment within island.     10
```

Figure 8.1: Repair a defective implementation of function α in binding environment B and construct a successor binding environ S that will be used in an execution site $\langle E, S\rangle$ as a replacement for $\langle E, B\rangle$. The function `curl/remote` (line 1) delivers a thunk (lines3–10) for remote evaluation and returns as its value a promise whose resolution is the value of the remote execution of the thunk.

Figure 8.1 sketches the dynamic repair of binding $\alpha/g \in B$ on an island I. Assume that function α was specified to accept either a list or a vector as its argument but its implementation g accepts only a list. The repair is constructed as a remote evaluation (lines 1–2) delivered to I for evaluation within an execution site $\langle E, \Delta\rangle$ whose binding environ Δ contains service functions tailored for live update. CURL u_Δ (line 2) is closely held and non-delegable; it is not known to any outside authority, and even if it were, it can be used only by a tiny handful of islands $J_1, \ldots, J_m$ (and perhaps only one island J) within a trusted sphere of authority.[10] The anonymous thunk (lines 3–10) implements the live update:

[10] The public keys of those islands $J_1, \ldots, J_m$ are encoded within CURL u_Δ which cannot be altered without detection by I.

- The binding environment B containing the defective binding α/g is fetched from an island-level registry of binding environments maintained by island I (line 4)
- The erroneous implementation g is fetched from B (line 5) and wrapped in an anonymous closure to correct the flaw (lines 6– 8).
- A successor environ S is derived from B (line 9) where the repair is substituted for the binding of α in B; and S is subsequently substituted for B in the execution sites constructed by I for service requests from other islands.

The repair is delivered to I as mobile code using a closely-held CURL u_Δ specifically constructed for island live update.

This live update, though trivial, illustrates several key advantages of COAST:

- An island can offer a customized execution site $\langle E_\Delta, B_\Delta \rangle$ whose execution site E_Δ and binding environment B_Δ have been customized to support live update. For example, E_Δ may implement detailed logging that assists system administrators and developers in verifying an update and B_Δ may contain unsafe, low-level functions for live update that are not available in any of the island's other execution sites.
- CURLs can be used to enforce access and security to prevent unauthorized agents from surreptitiously modifying portions of a system. Critical elements of system infrastructure can be protected by multiple, distinct CURLs to help ensure that no one single authority can act alone to conduct live update. In short, CURLs can act as layers of protection to minimize damage where the attack vector is the live update of the system itself.
- Live updates are mobile code in their own right executing within the confines of an execution site devoted to live update. Consequently, developers and system administrators are not restricted to a single update strategy and individual updates can be tailored as update-specific mobile code. In addition frameworks for the deployment and monitoring of live update can be constructed entirely as mobile code for delivery and execution at islands (in the same spirit as JavaScript applications that are delivered browser-side for execution).
- Live update can be delivered as a nanoservice. The example of Figure 8.1 presented its live up-

CURL u_Δ may narrowed in other ways, for example, it may have a limited lifespan (on the order of minutes), and may contain restrictions limiting exactly which bindings of environ Δ the update can invoke.

date as a remote evaluation (itself a form of nanoservice) however, it could also be issued as a remote spawn (by using `curl/spawn` in place of `curl/remote`), hence a long-lived COAST-based nanoservice. The return value of a remote evaluation can contain information on the successive stages of an update and a remote spawn can, in real-time, transmit on the incremental progress of an update. A more elaborate nanoservice-based update may wait for confirmation at critical points in the update before proceeding further or may notify administrators and developers of a critical failure and then rollback its portion of the update to a prior version or a known safe state. Live update nanoservicescan be deployed for many purposes: debugging, regression or A/B testing, performance tuning, penetration analysis, adaptation to security threats, or as transitionary infrastructure in preparation for large-scale pervasive updates.

- A live update is a closure, by definition, an amalgam of code and state. Consequently, it is trivial to include needed additional state in a live update that may: parameterize the update, act as a transition state enroute to a complete update, or to be incorporated into the end state of the update. Further, since rich, live *Motile* data structures may be captured in lexical scope of the closure that state may be incorporated into the target of the update without translation or restructuring or, when interpreted, drive the orchestration of the update. Closures may be a important vehicle for the delivery and installation of live updates.

However, the technique illustrated in Figure 8.1 has several limitations:

- Here the unit of service update is a single closure, however the technique can be trivially generalized to accommodate the atomic update of multiple closures in a single binding environment.
- Service requests already in progress will continue to execute unchanged, using the deficient binding environment B. However all future service requests will be executed in the remediated execution site $\langle E, S \rangle$.
- An island must be designed and implemented with live update in mind however, that burden can be eased by frameworks for island implementation.
- It does not make provision for updates to internal island data structures and is limited to the targeted modification of the binding environments of execution sites.[11]

[11] The design of libraries for COAST-based live update (that is, collections of functions to be included in the binding environment of a specialized island execution site) is an open research topic and the example of Figure 8.1 is merely suggestive and not definitive.

The technique has the advantage of atomicity and graceful transition from old to new but an islet that is already executing in the context of execution site $\langle E, B \rangle$ continues unchanged. If g migrates, via intra-island messaging, then other on-island islets can hold g (the deficient implementation) and edge cases can arise in which in which visiting computations, post-update, can conspire to circumvent the patch.[12]

8.3 CURL-based Live Update

A variant technique takes advantage of a CURL's ability to carry arbitrary metadata, including closures. In this case the remedy is code similar to that of Figure 8.1 but with the difference that the corrective code is delivered to island I by way of a *remedial* CURL u_r that service clients J would use to deliver a client closure to I. Recall that every CURL u for island I is either generated by I over its lifespan or offline by the sphere of authority responsible for I and that all of the metadata in any such CURL, including any mobile code, originates with either I or its authority.[13] Why bother to include repair code in CURLs when live update is available? For several reasons:

- Client-dependent or client-specific repair.
- Support for legacy repair or backwards compatibility that is rarely required or strongly deprecated.
- Downgrading a service to limit or eliminate access to particular service features for the sake of security, preparation for essential but disruptive upgrades, excessive workloads, or mitigation following an attack, theft of service, or abuse of fixed assets.
- Time-, load-, or state-dependent degradation of the precision or accuracy of service to reduce resource demands during periods of high load or when reducing latency takes priority over other considerations.
- Experimental deployment of a repair to selected clients.
- Early deployment of a repair to favored clients.
- Differential repair (as dictated by distinct repair-carrying CURLs $u_{r_1}, \ldots, u_{r_m}$) permits service clients to continue using legacy mobile code as they migrate to reformulated interfaces.

The mechanism works as follows. By convention, per-CURL metadata is represented by a binding environment (essentially a wrapper around a functional, persistent hash table) whose keys are symbols

[12] The security precautions detailed in Section 6.2 can be applied as a preventative.

[13] Every CURL u for island I must be signed by I's secret signing key. Only two entities hold that key: island I itself and the authority for I.

and whose values may include closures, data structures, continuations, binding environments, or other CURLs. The symbol `environ/repair` can be bound to a closure `(lambda (B) ...)` that accepts a binding environment as its single argument, performs the necessary repairs, and returns the successor binding environment S as its value. A "repair" CURL is yet another example of a highly differentiated service and can enforce complex preconditions to limit its scope or application.

8.4 More General Forms of Live Update

More general forms of repair and live update are possible if an island implementation offers a degree of self-reflection and manipulation. The general concept is simple, a closely guarded execution site $\langle E_\rho, B_\rho \rangle$, an execution engine and binding environment tailored for reflection and repair, to which the service provider deploys targeted mobile code for live update. As an example, an island itself, as well as the service domain-dependent bindings of its execution-site binding environments, will often depend on critical shared libraries written in other languages. To illustrate, *Island* itself depends upon `libsodium`, an open-source cryptographic library written in C.[14] and an experimental island binding environment supporting video encoding and streaming relied on `libvpx`, an open-source video codec (implemented in C) for VP8 video streams [113]. Libraries such as these, and countless others large and small, are repeatedly updated to correct bugs or omissions, improve performance and stability, extend functionality, refactor interfaces, eliminate security flaws, or improve integration with other libraries.

Here COAST mobile code can deliver a shared library to a host, load that library into an island, and reconfigure the island appropriately. Dynamic software update of this form can be used to: debug island services, modify or refine logging, generate measurements, compare A/B configurations, patch library-induced security flaws, update or refine services, introduce additional services, or repair flaws in existing services.

For the sake of security and safety an island may configure multiple, distinct execution sites $S_i = \langle E_{\rho_i}, B_{\rho_i} \rangle$, where each S_i is tailored for a specific live update task (such as debugging, logging, patching, starting, halting or restarting services, downloading a software library, or linking island services to a library) and access to each S_i is guarded by one or more S_i-specific live update CURLs u_{S_i} To help ensure that the island administrative services S_i are not abused islands can employ:

[14] See `https://www.gitbook.com/book/jedisct1/libsodium/details` for additional information.

- Use count and rate limit gates in the ingress points named by u_{S_i}. A use count of 1 enforces single-use making it impossible for an attacker to reuse u_{S_i}.
- Temporal gates within u_{S_i} to enforce "time locks" or constrain the lifespan of the CURL.
- Non-delegation within u_{S_i} to limit which islands may direct live update requests to a live update target island.
- A strong secret password (an n-bit cryptographic-grade random number x) to validate a live update request r. Here the CURL u_{S_i} accompanying r would include a cryptographic hash of $h(x)$[15] Requst r itself contains x and the live update target island validates x against $h(x)$ before executing r; if the validation fails then r is discarded and the request ignored.[16]

8.5 Live Update for COAST

Under COAST the distinctions between service provider and service client may be blurred and a single peer, acting as a client, may deploy remote spawns to peers elsewhere (both within and outside the sphere of authority of the client) to establish long-lived nanoservices that take advantage of remote resources: processor cycles, network bandwidth, databases, domain-specific binding environments, sensors, actuators, and the like. In other words, an island can serve as a host for nanoservices that are deployed by co-resident computations or remote computations on other islands without necessarily distinguishing between the two. A computation x on island I that has established nanoservices on other islands elsewhere in the network may also employ live update to patch bugs or deploy service upgrades. I briefly explore variations on the update of a single islet: in-place, intra-island, and inter-island.

8.5.1 Live Update of a Remote Evaluation

COAST service clients may often act as a service provider, either on their own behalf or for the sake of other service clients operating in the same sphere of authority. Every dispatch of a thunk f in a request for remote evaluation is the enactment of a single-shot service request (the evaluation of f and the transmission of its return value back to the client). Here live update is trivial; the client issues a successor closure f^+ when the service is next enacted.

[15] Using a modern, well-tested algorithm such as SHA256, SHA512, or SHA-3 for example) possibly augmented by a key-stretching algorithm (such as PBKDF2, bcrypt, or scrypt).

[16] All inter-island communications are encrypted. The current implementation uses a minor variant of `CurveCP` [16] incorporated into ZeroMQ [52]. See `http://curvecp.org/` and `https://github.com/zeromq/libcurve` for additional details.

```
(let loop ; Service loop.                                                    1
    ((s_1 ...) (s_2 ...) ... (s_m ...)) ; Initial state s_1, ..., s_m.       2
  (let ((request (receive))) ; Block waiting for an incoming service request.3
    (case (car request) ; Switch on request name.                            4
      ((r_1)                                                                 5
       ... ; Body for response to request r_1.                               6
       (loop s_1/r_1 ... s_m/r_1)) ; Fresh state after request r_1.          7
      ...                                                                    8
      ((r_n)                                                                 9
       ... ; Body for response to request r_n.                               10
       (loop s_1/r_n ... s_m/r_n)) ; Fresh state after request r_n.          11
      (else                                                                  12
       ... ; Body for response to unknown request.                           13
       (loop s_1/e ... s_m/e))))) ; Fresh state after unknown request.       14
```

Figure 8.2: A notional nanoservice loop whose state variables are $s_1, s_2, \ldots, s_m$. Each case r_i is a request of the form $(r_i\ a_1 \cdots a_{r_i})$ where r_i is a symbol, the name of the request, and the a_j are request arguments. For each request some actions are taken (suggested by the freestanding ellipsis of lines 6, 10, and 13) and the loop is recursively called with updated state variables $s_1/r_i, \ldots, s_m/r_i$.

8.5.2 Intra-Island Live Update: Background

In many cases a nanoservice is structured as an infinite loop that accepts a service request, acts on that request, generates the successor internal state of the service, and then loops again awaiting the next request; that request loop is an ideal transition point for a safe, live update [107]. Figure 8.2 is a notional example of that structure and with that structure in mind I sketch a framework for three distinct forms of live update under *Motile*/*Island*:

update/here The update occurs in-place, that is, a nanoservice f executing in the context of an execution site $\langle E, B\rangle$ "overwrites" itself — invoking a successor nanoservice f^+ atop the prior nanoservice f. In other words, the islet that once executed f now executes f^+.

update/near The successor f^+ executes on the same island I as the prior nanoservice f but as a different islet; the successor may (or may not) execute under the same execution site as the prior nanoservice. In this variant f is a hot backup for the successor f^+ and remains in place to resume service should the live update fail. Once the update concludes and the successor f^+ is in place the prior nanoservice f commits suicide.[17]

update/far The successor nanoservice f^+ executes on an island J different from the island I on which the prior nanoservice f resides. This variant also preserves f as a hot backup should the live update

[17] The thunk comprising f returns and the islet that animated f terminates.

fail. Once the update concludes and successor nanoservice f^+ is in place f updates itself in place as a forwarding proxy f^p for f^+. Service clients holding CURLs u_f for f can continue to use those CURLs; each service request routed via a CURL u_f will be received by f^p and forwarded to f^+.

In all three forms the live update is transparent and maintains continuity of service, though there may be a delay as service transitions from the prior nanoservice to its successor.

In general live update comprises five distinct stages:

Capture the internal execution state S_f of nanoservice f for restart of a successor nanoservice f^+ elsewhere. This includes domain-specific state, state specific to the service instantiation (for example CURLs, ingress points, and egress points), and any other service-specific state that must, for the sake of continuity, survive the transition from prior to successor.

Translate the output of state capture, S_f, to a form compatible with the expectations of the successor f^+. Abstractly, the state translation function $\tau : S_f \mapsto S_{f^+}$, is a bridge between the internal state of the prior service f and the initial state of the successor f^+. By definition τ is specific to both f and f^+.

Transition from f to the successor nanoservice f^+.

Rollback to the prior nanoservice f in the event state translation fails.

Retire the prior nanoservice f once the successor f^+ begins executing.

In essence, state capture delivers sufficient state to restart a service however, the meaning of "restart" is, by necessity, imprecise and elastic.[18] In many cases live update improves a service by repairing a defect, improving performance, increasing transparency of service (for example, additional logging) eliminating security flaws or adding additional security measures, or increasing the variety of requests to which the service responds. However, live update is not necessarily additive or monotonic. An update can just as easily: remove a service request that is deprecated or no longer used, eliminate a service request that has been assumed, or supplanted, by another service, restrict a service under attack, or impose client-specific restrictions in response to systemic abuse. Finally, updates may be nonfunctional including: moving a service to an execution site tailored for debugging, monitoring for the sake of performance, usage or detecting attacks, A/B testing prior to the introduction of an upgrade, load balancing, fleeing compromised computing resources (for example, a cloud experiencing a DDOS attack), transfer of service from one cloud to another or large-scale, live testing of disaster recovery procedures [26].

[18] I use *restart* in the sense of crash-only software [29] where start and restart are synonymous.

There are three distinct manifestations of state capture that reflect the three different forms of live update:

capture/here The captured state S will remain on-island and evaluated within the context of the execution site $\langle E, B\rangle$ in which it was captured. At a minimum it includes all of the fixed assets that the service requires. By definition, fixed assets are *Motile* or *Island* objects that cannot be transferred inter-island, for example, an input/output port, the ingress and egress points associated with islet communications, the bindings of an execution-site binding environment, or a physical asset such as a camera or pump. Since an island is a single address space shared among the islets executing on the island and intra-island messages among islets are exchanged as shared references then references to fixed assets can be transmitted from islet to islet on-island. However, given variations in the execution engines and binding environments across execution sites there is no guarantee that any execution site, other than the execution site of origin for S provides the capability to manipulate the fixed assets a (if any) within S.

capture/near The captured state S will remain on-island but may be evaluated in the context of an execution other than the site in which it was captured. S includes all of the fixed assets that the service requires but some assets may be represented in a manner that is execution-site independent. The execution site in which S will be evaluated may contain capability that either permits an islet to reconstitute a fixed asset from its execution-side independent representation or, directly manipulate the fixed assets a whose references appear in S.

capture/far The captured state S will be transmitted off-island. Consequently, no fixed asset may appear in S however S may contain descriptions of fixed assets expressed solely as *Motile* or *Island* values that are suitable for inter-island transmission. Those descriptions may be interpreted by a recipient of S to "reconstitute" a fixed asset (or its equivalent).

Because *Motile* is a single-assignment functional language the lexical scope bindings of a closure λ are fixed and immutable at the point at which the definition of λ is evaluated. Consequently, a nanoservice preserves static state in the lexical scope bindings of the thunk f dispatched to an islet for execution and maintains dynamic state in the bindings of its service loop (see Figure 8.2, lines 2, 7, 11, and 14). Implementing live update as a service request guarantees the integrity of state capture; the static state, comprising the closure lexical scope bindings is immutable and, by construction, the dynamic state,

comprising the bindings $s_1, \ldots, s_m$ of the service loop, is complete and self-consistent.[19]

With this as background I sketch an implementation of the two forms of intra-island live update for nanoservices: *update/here* and *update/near* in terms of the five stages of live update: state capture, state translation, transition, rollback, and retirement. Each nanoservice f recognizes three service requests:

$$\texttt{(UPDATE/HERE } \tau_{f^+} \ f^+\texttt{)}$$

$$\texttt{(UPDATE/NEAR } r_{f^+}\texttt{)}$$

$$\texttt{(UPDATE/FAR } r_{f^+}\texttt{)}$$

where $\tau_{f^+} : S_f \mapsto S_{f^+}$ and r_{f^+} is the resolver for a promise p_{f^+} held by f^+. I discuss UPDATE/HERE in Section 8.5.3, UPDATE/NEAR in Section 8.5.4 and the third, UPDATE/FAR, in Section 8.5.5.

8.5.3 Intra-Island Live Update: In Place

We assume there exists a specific CURL for f, $u_{f\Delta}$, by which f receives the request

$$\texttt{(UPDATE/HERE } \tau \ f^+\texttt{)}$$

That CURL is closely held; it may be time-locked, use-count or rate-limited, non-delegable or delegable to only a small set of islands, or contains pure predicates (the equivalent of stateless gates) that rely on go/no-go indicators residing on other islands. Unless a request UPDATE/HERE is delivered via $u_{f\Delta}$ it is ignored. The workflow for UPDATE/HERE is:

capture f collects its state inline as a *Motile* record S_f. The field names and their values are f-specific.

translate f invokes $(\tau_{f^+} \ S_f)$ to generate a record S_{f^+}, a state record for f^+. The field names and their values are f^+-specific.

rollback If state translation fails then f restarts its service loop using the state arguments $s_1, s_2, \ldots, s_m$ (see Figure 8.2, lines 2, 7, 11 and 14).

transition f halts with a return value of $\texttt{(RESTART } f^+ \ S_{f^+}\texttt{)}$. An island wrapper (itself structured as a nanoservice) around nanoservice f recognizes the RESTART command and calls $(f^+ \ S_{f^+})$ — starting the service loop of successor f^+.[20]

rollback There is *no* rollback to f if the transition to f^+ fails.

retirement f self-terminates as it transitions to successor f^+.

[19] In other words, state capture is performed inline.

[20] The presence of the wrapper guarantees that the closure f and its stack frames are garbage collected. One way to think of the wrapper is that it executes in stack frames below that of the prior service f.

In other words, a live *update/here* is performed *in place* where the successor nanoservice f^+ inherits the state of the prior version and occupies the same address space and execution site. In particular, since all of the CURLs issued by f are still effective under f^+ other computations that held CURLS u_f for f can continue to use those same CURLs to transmit service requests to f^+. To the extent that f^+ behaves as did f the live update will be transparent to any former client of f.

8.5.4 Intra-Island Live Update: Co-resident

A more elaborate workflow is required for *update/near* where the live update f^+ executes on the same island I as the prior version of the nanoservice but requires a different execution site, $\langle E^+, B^+ \rangle$. This can arise if update f^+ must have access to island resources different from those available in the execution site $\langle E, B \rangle$ of f, such as more (or less) liberal allocations of processor cycles, memory or network bandwidth, updated libraries, and the like. There are also security considerations that arise: nanoservice f may be barred from creating spawns on island I (this would be enforced by ensuring that the primitive `curl/spawn` does not appear in the binding environment B of f). Consequently, while f must cooperate in the update we can't rely on it to spawn f^+ and instead must turn to another computation x elsewhere that has the capability to spawn computations on I in $\langle E^+, B^+ \rangle$ — understanding that x may reside elsewhere on an island other than I.

```
(begin                                                                  1
  (define (f+/skeleton) ; Skeleton thunk for spawning f+.               2
    (let* ((promise/f+  (promise/new)) ; Promise to be fufilled by f.   3
           (resolver/f+ (promise/resolver promise/f+)))                 4
      ; Ask f for state capture S_f.                                     5
      (curl/send f_Delta@ (list (quote UPDATE/NEAR) resolver/f+))       6
      ; Block waiting for pair (S_f . r_f) from f.                      7
      (let* ((response (promise/block promise/f+))                      8
             (S_f (car response))  ; State S_f from f.                  9
             (r_f (cdr response))) ; Resolver r_f for promise p_f of f. 10
        (cond                                                           11
         ((tau_f+ S_f) => ; Translate S_f to S_f+ for f+.               12
          (lambda (S_f+)  ; State translation OK.                       13
            (promise/fulfill r_f (quote HALT)) ; HALT f.                14
            (f+ S_f+))) ; Start f+.                                     15
         (else ; tau_f+ failed.                                         16
          (promise/fulfill r_f (quote RESTART)))))))) ; RESTART f.       17
                                                                        18
(curl/spawn I@ f+/skeleton)) ; Spawn skeleton on I in site <E+, B+>.    19
```

Figure 8.3: The closures that x, executing on island J, spawns on island I to create nanoservice f^+, the successor to nanoservice f, itself executing on island I.

There are three additional complications:

- The state S_f that f assembles for the sake of the update cannot be transmitted off-island I as it may contain fixed assets intended for the successor nanoservice, also executing on I (after all, this is what update/near means). But absent a guarantee that computation x resides on I, the captured state S_f cannot be handed off to x as an intermediary between f and f^+ (the fixed assets will not survive transmission off-island). f must therefore transmit its captured state S_f directly to its successor f^+.
- State translation τ_{f^+} must be executed in the execution site, $\langle E^+, B^+ \rangle$, of the successor nanoservice as it may rely upon bindings found only in B^+.
- For the sake of safety and security we must ensure that f^+ interacts with f only when and as prescribed.

The scenario also presents a "chicken and egg" problem that is a consequence of communication by introduction: we cannot start f^+ without S_f but f must transmit S_f directly to f^+, for which f requires a CURL u_{f^+} for f^+ that itself must be generated by f^+. By spawning a skeleton on I for f^+ that executes a closure generated for it by x we resolve all four problems:

- f communicates its captured state S_f directly to co-resident computation f^+ (via a promise generated by the skeleton).
- τ_{f^+}, the state translation from S_f to S_{f^+}, executes in $\langle E^+, B^+ \rangle$.
- x dictates all interactions with f and f^+ does not execute until those interactions are complete. Thereafter, it is impossible for f^+ to communicate with f in any manner: either f^+ starts and f halts or f^+ fails to start and f resumes service exactly at the point prior to the live update.

Given these assumptions x holds:

- CURL $u_{f\Delta}$ for f.
- CURL u_I granting x the capability to spawn a computation on I under the desired execution site $\langle E^+, B^+ \rangle$.
- Closure f^+, the successor nanoservice.
- Closure $\tau_{f^+} : S_f \mapsto S_{f^+}$, a state transform function for the live update of f to f^+.

Figure 8.3 contains the *Motile* code that x will use to spawn f^+ on island I. For the sake of simplifying

the presentation, assume that the bindings for:

Variable	**Notation**	**Line** (Figure 8.3)
`f_Delta@`	$u_{f\Delta}$	6
`I@`	u_I	19
`f+`	f^+	15
`tau_f+`	τ_{f^+}	12

are in an outer lexical scope.

All of the five stages of live update: capture, translation, transition, rollback, and retirement, are embodied in the thunk, `f+/skeleton` (lines 2–17 of Figure 8.3) that x spawns on island I in the execution site $\langle E^+, B^+\rangle$ (line 19). The back and forth between the skeleton, executing on I, and f follows this course:

capture The skeleton creates a promise/resolver pair p_{f^+}/r_{f^+} (lines 2–3) and dispatches to f the service request (`UPDATE/NEAR` r_{f^+}) via the CURL $u_{f\Delta}$ (line 6). It then blocks awaiting the fulfillment of the promise by f (line 8).

capture In response to the request (`UPDATE/NEAR` r_{f^+}) f creates a promise/resolver pair p_f/r_f and transmits its state, S_f, and resolver r_f as pair (S_f . r_f) via CURL r_{f^+} to f^+ (thereby fulfilling the promise p_{f^+} held by the skeleton). S_f will include any CURLs, ingress points, and egress points that f has accumulated. f will block, waiting on the fulfillment of promise p_f.

translate The skeleton, on receiving (S_f . r_f) (lines 8–10), computes (τ_{f^+} S_f) to translate state S_f to a state S_{f^+} for the start of f^+ (line 12).

retire If the translation is successful then a `HALT` request is sent to f by the skeleton via resolver r_f (line 14). f, blocked on p_f, will receive the `HALT` and cleanly terminate.

transition The skeleton, once it has commanded f to halt, transitions into f^+ passing S_{f^+} as the initial state of f^+ (line 15). Control will enter the nanoservice loop of f^+ and, barring a fault or deliberate halt, will not return.

rollback If the translation fails (the return value of (τ_{f^+} S_f) was `#f`) then a `RESTART` request is sent to f via resolver r_f (line 17). The skeleton terminates without starting f^+ and f transparently resumes service with the first pending service request.[21]

[21] No service requests are ever lost in the transition; neither f nor f^+ read a service request until either f^+ has assumed service or f restarts. This same protocol could also be used where both f and f^+ execute under the same execution site $\langle E, B\rangle$ — to ensure that f remains behind as a hot backup for f^+ should the transition fail. The only difference is that CURL u_I must target an execution site $\langle E, B\rangle$ on I rather than $\langle E^+, B^+\rangle$.

8.5.5 Inter-Island Live Update

It may be advantageous to create a successor nanoservice f^+ on an island J other than the island I inhabited by the current version f. However, each CURL u_f is tied to (and signed by) its island of origin, I in the case of nanoservice f. If we instantiate f^+ on another island J then none of the CURLs u_f issued by f can be used to deliver service requests directly to f^+. How then to preserve the CURLs of f for the sake of transparent transition from f to f^+? Update f as a proxy that forwards service requests directed to f onward to the successor nanoservice f^+. In other words, the price of updating a service to execute elsewhere on another island is that the prior service must itself be updated and left behind as a proxy to guarantee service transparency and continuity.

Assume that nanoservice f resides on island I and that the successor nanoservice f^+ will reside on island $J \neq I$. A computation x, distinct from f, directs the live update by generating a custom closure and spawning it on island J. Computation x has two intertwined goals: first, start the successor service f^+ on island J, and second, repurpose the prior service f as a forwarding proxy p^+ for f^+. The rules of communication by introduction require that p^+ be introduced to f^+ otherwise it will be impossible for p^+ to forward service requests to f^+ that are issued via older, outstanding CURLs u_f. However, CURLs u_{f^+} can only be issued by island J on behalf of f^+ and, absent access to the secret signing key and internal run-time state of J, x must spawn a computation on J to generate such CURLs.[22]

To simplify the mechanics x relies on two higher-order, single-argument generators, F^+ and P^+, for the successor closures f^+ and p^+ respectively. The first generator, $(F^+\ d)$, given a duplet d[23], returns a closure f^+ where duplet d is bound within the lexical scope of f^+. This technique, illustrated in Figure 8.4, is a trope of introductory functional programming.

The second generator, $(P^+\ u_d)$, given the CURL u_d of duplet d, returns a closure p^+ where CURL u_d is bound within the lexical scope of p^+. Proxy p^+ will use CURL u_d to forward service requests from

[22] The sphere of authority responsible for J will closely hold the secret keying material of J. Any agency in possession of those keys can generate CURLs that will (given assumptions about J's conventions for naming ingress points) be accepted by J as valid, genuine CURLs for J. The distribution of *Motile/Island* contains examples of this and it is the mechanism by which "bootstrap" CURLs u_β are generated offline and distributed out-of-band (for example, via email or embedded in web pages) to island developers and users. However, as a matter of practical security, such CURLs u_β count little for live update and any industrial-strength, COAST-based framework for live update will, by necessity, rely on dynamic communication-by-introduction.

[23] A *duplet* is a *Motile* object comprising a transport t, an ingress point $t\bullet$, an egress point $t\triangleright$, and an unguessable CURL u_d for the ingress point $t\bullet$. It establishes a private, communication-by-introduction path from a source computation x to a sink computation y.

```
(define (+n n)                                                    1
  (lambda (x) (+ x n)))                                           2
                                                                  3
(let ((+17 (+n 17)))                                              4
  (+17 33))                                                       5
```

Figure 8.4: The higher-order generator +n (lines 1–2) returns a single argument closure (line 2) that adds n to its argument x. Such a closure is generated and bound to the variable +17 (line 4) that, when called with an argument of 33 (line 5), returns the value 50.

island I, the former residence of nanoservice f, to its successor nanoservice f^+ residing on island J. Since f^+, on island J, contains duplet d within its lexical scope it can receive (read) the service requests forwarded to it by p^+ on island I.

Given these assumptions x holds:

- CURL $u_{f\Delta}$ for f.
- CURL u_J granting x the capability to spawn a computation on J under the desired execution site $\langle E^+, B^+\rangle$.
- F^+, a higher-order generator for f^+, the successor nanoservice.
- Closure $\tau_{f^+} : S_f \mapsto S_{f^+}$, a state transform function for the live update of f to f^+.
- P^+, a higher-order generator for p^+, a forwarding proxy for f^+.
- A closure $\tau_{p^+} : S_f \mapsto S_{p^+}$, a state transform function for the live update of f to p^+.

As illustrated in Figure 8.5 thunk `skeleton/f+` (lines 2–21) is spawned on island J (line 23) under execution site $\langle E^+, B^+\rangle$. That computation orchestrates the live update: from nanoservice f on island I to nanoservice f^+ on J. The skeleton, executing on island J, must first capture the state of f, S_f.

capture The skeleton, on island J:

- Constructs a promise p_{f^+}/r_{f^+} (lines 3 and 4).
- Transmits the service request (`UPDATE/FAR` r_{f^+}) to island I via CURL $u_{f\Delta}$ (lines 4 and 6).
- Blocks waiting for the promise to be fulfilled (line 8) as pair (S_f . r_f) (lines 9–10).

The service f, on island I:

- Receives the service request (`UPDATE/FAR` r_{f^+})
- Assembles state S_f. Since state S_f will be transmitted off-island to J it may not contain any fixed assets of I. It may however contain CURLS or island-independent descriptions of fixed

```
(begin                                                                    1
  (define (skeleton/f+) ; Skeleton thunk for spawning f+.                 2
    (let* ((p/f+ (promise/new)) ; Promise to be fufilled by f.            3
           (r/f+ (promise/resolver p/f+)))                                4
      ; Ask f for state capture S_f.                                       5
      (curl/send f_Delta@ (list (quote UPDATE/FAR) r/f+))                 6
      ; Block waiting for pair (S_f . r_f) from f.                        7
      (let* ((response (promise/block p/f+))                              8
             (S_f (car response))  ; State S_f from f.                    9
             (r_f (cdr response))) ; Resolver r_f for promise p_f of f.   10
        (cond                                                             11
         ((tau_f+ S_f) => ; Translate S_f to S_f+ for f+.                 12
          (lambda (S_f+)  ; State translation OK.                         13
            (let* ((forward (islet/curl/new ...)) ; Duplet includes CURL. 14
                   (f+@ (duplet/resolver forward)); CURL of duplet.       15
                   (p+ (P+ f+@)) ; Closure p+ has CURL f+@ of duplet.     16
                   (f+ (F+ forward))) ; f+ has duplet.                    17
              (promise/fulfill r_f (list (quote UPDATE/HERE tau_p+ p+)))  18
              (f+ S_f+))) ; Start f+.                                     19
         (else ; tau_f+ failed.                                           20
          (promise/fulfill r_f (quote RESTART)))))))) ; RESTART f.        21
                                                                          22
  (curl/spawn J@ skeleton/f+)) ; Spawn skeleton on J in site <E+, B+>.    23
```

Figure 8.5: Thunk `skeleton/f+` (lines 2–21) is spawned on island J (line 23) to undertake the live update of nanoservice f on island I to nanoservice f^+ on J.

assets expressed as *Motile* values that can be serialized.

- Creates a promise/resolver pair p_f/r_f.
- Transmits, via resolver r_f^+, the pair $(S_f \,.\, r_f)$ to island J in reply to the service request.
- Blocks waiting for promise p_f to be fulfilled.

translate The skeleton, on island I, on receiving response $(S_f \,.\, r_f)$ (line 8):

- Maps state S_f into a state S_{f^+} for f^+ by executing $(\tau_{f^+}\ S_f)$ (line 12).
- Fails to map state S_f (τ_{f^+} returns `#f`). In this case we enter the rollback phase.

rollback The skeleton has failed to map state S_f to the state S_{f^+} that the skeleton expects:

- Promise p_f of f is fulfilled via resolver r_f as the symbol `RESTART` by the skeleton and the skeleton terminates.
- f, waiting on the resolution of promise p_f, transparently restarts its service loop exactly where it left off (all of its state is intact and unchanged). No service requests are lost and service clients experience, at worst, a modest delay.

transition If state mapping (line 12) succeeds then the anonymous, single-argument closure of lines

13–19 is called with S_{f^+}, the return value of τ_{f^+}; starting the transition phase:

- A single J-specific duplet is created (line 14) that will allow the proxy $p+$ on I to forward service requests to f^+ on J.
- CURL u_d is extracted from the duplet and bound to `f+@` (line 15).
- A closure for forwarding proxy $p+$ is generated and bound to `p+` (line 16). That closure contains the CURL u_d that will allow the proxy to transmit service requests to f^+.
- A closure for f^+ is generated and bound to `f+` (line 17). That closure contains the duplet (line 14) that will allow nanoservice f^+ to receive the service requests forwarded to it by proxy p^+.
- Promise p_f of f is fulfilled via resolver r_f as the list `(UPDATE/HERE` τ_{p^+} p^+`)`.
- f, waiting on the fulfillment of promise p_f, receives `(UPDATE/HERE` τ_{p^+} p^+`)`. In response f transitions in place (*update/here*) to nanoservice p^+ (see Section 8.5.3).
- f^+ is started as `(`f^+ S_{f^+}`)` (line 19) and enters its service loop awaiting the arrival of service requests from proxy p^+.[24]

retire nanoservice f is "overwritten" by proxy p^+.

8.6 Related Work

A substantial body of work is devoted to the live update of legacy software[25] while striving to maintain adequate levels of safety, flexibility, and reliability. However, that work is largely untethered to either an architectural style or even a specific architecture. Giuffrida and Tanenbaum [108] argue that progress in live update is hobbled by an emphasis on legacy software and propose, as an alternative, live update for systems that actively cooperate with updates to ensure a safe transition from old system to new. Given the BASE model and prior work on the role of architecture in dynamic software update [182], I regard the *cooperative live update* of Giuffrida and Tanenbaum as a call to ground live update in architectural styles deliberately crafted to simplify large-scale system updates.

COAST-based live update bears comparison to the mechanisms developed for PROTEOS [107], an operating system expressly designed for automated live update. While the techniques employed in PRO-

[24] Careful readers may suspect that lines 18-19 (Figure 8.5) hide a race condition in which the proxy p^+ can forward service requests to f^+ before f^+ is ready to receive them and consequently, one or more service requests may be lost in transition. Not so. The transport underlying the duplet constructed for forwarding (line 14) guarantees that all service requests arriving on island J will be enqueued until received (read) by f^+.

[25] Hayden et al. [123] contains an excellent review of recent work in dynamic software update (see Section 7, *Related Work*, pages 13:32–36).

TEOS are different from those detailed here, the PROTEOS concept of *state quiescence* is equally useful in COAST and the event loops of PROTEOS subsystems are conceptually identical to the service loops of COAST-based nanoservices.

The update-specific *state filters* of PROTEOS could be implemented per-island by update-aware execution sites. The *interface filters* of PROTEOS that are used to improve convergence guarantees could be implemented by specialized transports and a few modest island-level reflective mechanisms. It could be presented to clients as a privileged island nanoservice for which access is limited to closely-held CURLs similar in spirit to the CURL $u_{f\Delta}$ for nanoservice update.

PROTEOS employs an *update manager* that orchestrates updating multiple processes simultaneously in a single update transaction. The COAST analog would be a specific per-island nanoservice for coordinating and sequencing a set of nanoservice updates for nanoservices $f_1, \ldots, f_m$ simultaneously. Under PROTEOS the update manager is responsible for *hot rollback* of all services if an update transaction fails. Their update manager can detect and automatically recover from a variety of errors including timeouts in state quiescence that arise from erroneous orchestration conditions, deadlocks, or other synchronizations failures, timeouts for state transfer and translation (due to crashes, errors in state translation, or service loops that fail to converge), and finally crashes or errors in the updates themselves. These issues, vital for reliable and safe industrial-strength update, are ignored in the protocols presented in section. Nonetheless, there are no apparent technical obstacles to the inclusion (as a nanoservice) of a an update manager on each COAST island with comparable safeguards for multiple nanoservice upgrades in a single transaction.

8.7 Summary

COAST provides fundamental mechanisms that reduce the complexity of live update. Mobile closures encapsulate both state and code; a critical prerequisite for simplifying live update. As *Motile* is based on Scheme, closures are first-class values that can be generated on-the-fly to satisfy the demands of each specific update strategy. COAST is message-centric, reducing closure transmission and update coordination to basic primitive operations. The serialization of structured data (lists, vectors, hash maps, sets, records, and CURLs) encourages constructing state expressed as rich compositions of these primitive structures, thereby simplifying state capture and state interpretation. Finally, the *Island* infrastructure,

which lies outside of the strict boundary of the COAST architectural style, is a hospitable environment for live update. Spawning, where closures are dispatched to an island for execution, is the COAST-equivalent of live code migration and is a necessary element of live update.

Services expressed as endless loops illustrate how a well-formed architectural construct can be exploited for sound state quiescence and capture. The mechanisms described for intra- and inter-island live update (Sections 8.5.3–8.5.5) have their roots in COASTCAST, a video streaming service in which live streams can be moved in near-realtime from source to source (camera or forwarder) and from sink to sink (forwarder or display service) [113]. There video encoding services (implemented as closures executing in binding environments specialized for video capture and compression) were migrated from island to island (and host to host) within a local area network. Video display services, implemented as closures executing in binding environments specialized for video decompression and display, could simultaneously be migrated or duplicated live within the network. The use of CURLs to regulate access to video sources or video sinks allowed COASTCAST users to securely share video feeds and selectively restrict their distribution.

More generally, these service structures hint at large scale, decentralized, simultaneous, live update across multiple islands and multiple spheres of authority where a designated authority orchestrates a hierarchy of decentralized update managers to execute a cross-authority update in a single transaction. Update services of this breadth and depth are an interesting topic for future research.

Computation migration under COAST is a special case of intra- and inter-island live update where $f^+ = f$ and τ_{f^+} is either the identify function or a minor remapping of state S_f. Migration is itself a useful capability with many applications, for example:

- Services fleeing an island, host, or cloud that is under attack.
- Transferring a nanoservice f to a tailored execution site $\langle E^+, B^+ \rangle$ for debugging, logging, performance analysis, or security monitoring.
- Reallocating service resources to compensate for hardware or network failures.

Notably, migration, like live update, is transparent in the sense that service clients may experience a minor delay in service but not a significant interruption. In particular, no service requests would be lost as a consequence of migration. The techniques of state capture, state transfer, and state translation

can be used to generate nanoservice checkpoints as safe points for emergency restarts or to collect live nanoservice state for fault predication, post-analysis, or soft real-time analytics. Lastly, replicating multiple, identical nanoservice instances, for example, hot backups or members of a worker clan living behind a reverse proxy, can be implemented using techniques similar to those of computation migration. Solutions to managing large-scale, COAST-based infrastructure may provide fresh insights into the problems of operating other large-scale services and infrastructure.

While many architectural styles offer dynamic adaptation, it is the rare style for which the adaptations are expressed in the style itself. Every single step in the live update protocols is a consequence of one of the COAST style rules. All of the mechanics of live update are either primitive mechanisms in *Motile*/*Island* drawn directly from the style rules or constructed from those mechanisms (decentralized promises are an example of the latter). The ease with which live update with hot backup can be implemented wholly within the style by the style speaks to the adaptive span of COAST.

COAST offers unique security and safety benefits for live update. All access to the update machinery of each individual service f is mediated by an unguessable f-specific CURL $u_{f\Delta}$. Update-specific CURLs protect services from malicious updates; granting the capability to update based on predicates including time of day, rate of update, origin of update, number of updates, the state of f, and arbitrary island or system conditions. Without access to CURL $u_{f\Delta}$ it is impossible for any authority, either by accident or design, to update f. Safe hot rollback is transparent (without loss of service) thanks to persistent, functional data structures, immutable messages and the underlying communication model for COAST. Communication by introduction and the systematic use of promises minimizes contact between successive versions of a service, shrinks the number of synchronization points, and reduces the risk of livelock or deadlock during update.

Both computation exchange and object-capability contribute to the ease with which COAST-based systems support safe and secure live update with hot backup. Computation exchange reduces the transfer of state and code, a necessary precondition for live update, to a near triviality. Spawning is exactly the mechanism one requires for both live update and migration. Object-capability plays multiple roles: the destination execution site for the successor service, closely held CURLs devoted solely to the operations required for live update, and decentralized promises for securely communicating the intermediate results of the update process between the prior service and the successor service.

Chapter 9: Evaluation: Web API Design

In this chapter I explore the claim *sufficienlty expressive for adaptive services* from the perspective of service APIs for web services.[1] Service APIs are not born perfect or immutable—they change over time reflecting: customer demands for additional features, repairs to remedy semantic or security flaws, accumulated experience with the convenience, function, or efficiency of the service, business pressure to differentiate the service offerings from those of competitors, changes in regulatory requirements, or contractual obligations, to name but a few of the "new or altered circumstances" influencing service evolution. In this context I evaluate the ability of COAST to accommodate the ongoing, natural evolution of service APIs from the perspective of service consumers, that is: can service consumers eliminate or repair inconsistencies and omissions in the service APIs of providers?

Section 9.1 briefly reviews web APIs from a COAST perspective, focusing on the *Delicious* API, a well-known service interface for web bookmark services, Section 9.2 contains a review the HTTP version of the *Delicious* API and characterizes its flaws. Section 9.3 restates the *Delicious* API, reducing it to a mere three functions for inclusion in the binding environment of an execution site of a service provider for web bookmarks. Section 9.5 turns to the core problem of (re)constructing a service-rich API using nothing but consumer-generated mobile code and presents a rich set of examples that demonstrate the ease with which service consumers can remediate the deficiencies or omissions of provider-side APIs. Section 9.6 briefly touches on the performance of consumer-defined APIs. Finally, section 9.7 summarizes the contributions of COAST-based live adaptation for API repair and extension.

9.1 Web Service Design from a COAST Perspective

From a service perspective COAST is as expressive as REST or WS-* since *Motile*/*Island* can emulate HTTP/1.1 or SOAP [156, 157]. Further, a *Motile*/*Island* peer can interact seamlessly with HTTP/1.1-compliant web sites and browsers; acting as a web client or web server respectively. One can craft COAST execution sites that emulate the essential semantics of either REST or WS-* since an execution site can

[1] Public facing web services are commonly decentralized as the the web browser (the client) often operates under a sphere of authority that is distinct from the sphere of authority of the web server. Web services internal to an enterprise, for example, time card reporting or purchase requests, are distributed but likely not decentralized.

be populated with the functional equivalent of the basic REST methods `GET`, `PUT`, `POST`, and `DELETE` or, since RPC is a proper subset of the COAST semantics, the functional equivalent of WSDL service methods. However, both REST and WS-* service APIs fall prey to the many failings of software libraries including awkward or incomplete primitives, irregular coverage of the domain, inappropriate granularity, complex arguments, specification errors, and inconsistencies within and among like services [9].

Despite the historical failings of the APIs of software libraries or frameworks, an integrated collection of functions dedicated to a specific domain or task can offer many advantages including ease of use, precision of interface, and a high degree of utility. With respect to service COAST offers a "third way" in which fine-grain service primitives are made available to service clients as individual functions within the binding environment of an execution site and, unlike REST or WS-*, this primitives are combined as the client, not the provider, dictates. Even if the COAST service primitives are the direct analogues of RESTful requests, COAST mobile code allows multiple requests to be batched together and refined server-side to deliver just the results the client requires and nothing more. In addition, as we argue below, deficiencies, errors, or omissions in these APIs can be remedied by the service provider, the service client, or both, thereby accelerating the evolution and refinement of decentralized services.

Web service APIs and the details of their HTTP requests and responses are often hidden behind a language-specific facade (C, Go, Python, or Ruby for example). Denaro, Pezzè, and Tosi [56] examine integration issues in a case study of Web 2.0 social applications that integrated the published HTTP-based APIs of *Delicious* or *OpenSocial*.[2] I focus here on the *Delicious* API.

The Delicious web site, `https://delicious.com/`, is a social tool for saving and sharing bookmarks to web sites of interest and for also browsing popular web content. Its API—Plain Old XML (POX) over HTTP—and far from RESTful has been adopted by other social bookmark web sites. In addition, on the client side, multiple libraries are available for the Delicious API in Java, PHP, and Python. Given m distinct Delicious-compatible web sites and n available Delicious API client libraries (hence $m \times n$ combinations of sites and client libraries) Denaro et al. explore the integration problems this diversity presents and I use their findings to guide an evaluation of COAST-based adaptation and remediation for service providers and service clients alike. I begin with a synopsis of the Delicious API and from there use the findings of Denaro et al. as a springboard for presenting COAST-based provider- and client-side remediations.

[2] `https://delicious.com/developers` and `http://docs.opensocial.org/display/OSD/Specs` respectively.

Table 9.1: The Delicious HTTP interface for searching and manipulating bookmarks.

Method	Path	Description
GET	/posts/update	Check to see when a user last posted an item
POST	/posts/add	Add a new bookmark
GET	/posts/delete	Delete an existing bookmark
GET	/posts/get	Get bookmark(s) for a single date, or fetch specific bookmarks
GET	/posts/recent	Fetch recent bookmarks
GET	/posts/dates	List dates on which bookmarks were posted
GET	/posts/all	Fetch all bookmarks by date or index range
GET	/posts/all/hashes	Fetch a change detection manifest of all bookmarks
GET	/posts/suggest	Fetch popular, recommended, and network tags for a given URL

9.2 The Delicious API

The Delicious API accommodates three distinct classes of objects: bookmarks, tags, and bundles.[3] I confine my discussion to the two fundamental classes, bookmarks and tags. Each bookmark contains a web URL and metadata including tags, an indexing mechanism for bookmarks. Each bookmark is assigned one or more tags $t_1, \ldots, t_m$ and each tag t may be affiliated with multiple bookmarks $b_1, \ldots, b_n$. The API is far more RPC-like than it is RESTful [135]:

- The first class objects of the Delicious data store, bookmarks and tags, are not exposed as resources.
- The HTTP methods are used incorrectly and the HTTP method GET is repeatedly used to implement non-idempotent operations.
- The resource operations are not interconnected. It is impossible to traverse from a set of bookmarks to a single bookmark or from a tag to the set of bookmarks indexed by that tag.

Bookmarks are the building blocks of Delicious and each contains a web link (URL) and several elements of metadata. There are nine API requests for bookmarks, enumerated in Table 9.1. *Tags* are brief, descriptive text strings that users attach to an URL to describe and categorize a bookmark. There are three API requests for manipulating tags, shown in Table 9.2.

Denaro, Pezzè, and Tosi [57] uncovered four distinct classes of *integration faults*:[4]

- *Inconsistent interpretation of parameters or values*. Here the API interpretations are reasonable but

[3] To simplify the discussion I ignore the *Delicious* authentication protocol https://github.com/avos/delicious-api/blob/master/api/oauth.md. The specification of the API, https://github.com/SciDevs/delicious-api, is thin, incomplete, and omits many critical details. I also ignore the specifics of the XML representation of the *Delicious* responses as XML (or JSON for that matter) as the format for responses is a matter for the client mobile code and not the COAST service provider.

[4] Taken from page 256, Table 3 of [56]. Detailed experimental data available in [245].

Table 9.2: The Delicious HTTP interface for searching and manipulating tags.

Method	Path	Description
GET	/tags/get	Fetch all tags from all of a user's posts
GET	/tags/delete	Delete a tag from all of a user's posts
GET	/tags/rename	Replace all occurrences of a tag with a new tag within a user's posts

incompatible: old parameters or their values are now interpreted differently, parameter orders may have changed, or new positional parameters have been inserted between older parameters.

- *Violations of value domains or capacity/size limits.* The older API relies on implicit assumptions on ranges of values or sizes that no longer hold for the newer API. For example, a newer API my arise from improvements in scaling or performance that allow services to cope with larger workloads.
- *Side-effects on parameters or resources.* The implicit effects of an API on resources (whether sever-side or elsewhere) are not explicitly documented in its interface.
- *Missing or misunderstood functionality.* Incorrect assumptions about results due to incomplete documentation.

In addition, many REST and WS-* APIs rely heavily on strings for parameter passing. For example, the *Delicious* API uses strings with embedded separators to represent list parameters and results but fails to define the set of legal separators [245]. This is but one example of the dangers of string parameters: they may require parsing and can be exploited in scripting attacks [230]. In *Motile* this flaw is easily remedied as immutable lists are a basic data type, easing the representation of common structures such as sequences, trees, and acyclic directed graphs. In many cases the specific flaws and omissions in the *del.icio.us* API can be eliminated outright by using well-defined *Motile* data types, reducing the API to simpler primitives, and exploiting closures and binding environments as functional supplements.

9.3 Delicious as a COAST-Based Service

I restate the Delicious API as a small set of functions for inclusion in a binding environment B of an execution site $\langle E, B\rangle$. Clients deploy remote computations to "Delicious" islands for evaluation to add bookmarks, search sets of bookmarks or tags, and add or tags. Unlike Delicious I assume that the bookmark database maintains a complete temporal history of a user's collection of bookmarks, that is, each bookmark is maintained, in temporal order, from most recent to oldest. In other words, the database of bookmarks is persistent and any user u can effectively snapshot her bookmarks by noting the time of

the most recent bookmark of interest. Further for any other user $v, v \neq u$ user u can snapshot $v's$ bookmarks in the same way. Unlike Delicious tags are treated as first-class objects and can be manipulated independently of the bookmarks with which they are affiliated.

To simplify the presentation I ignore the details of user authentication and assume that for each user u there exists a secret (known only to u and the bookmark service) cryptographic token s_u that identifies u to the bookmark service provider. Each user u also has a public token, known to other users v, that uniquely names u's database of bookmarks and tags. Again for the sake of simplicity, I assume that the bookmarks and tags of each user are public and, given the public token for a user u any user v may read the bookmarks and tags of u.

For convenience I adopt the following nomenclature: an *url* is an URL given as a string, an *id* is a cryptographic-grade random number (given as a byte string), a *datestamp* is a POSIX-like timestamp (also known as Unix epoch time), the number of elapsed milliseconds since 1970-01-01T00:00:00Z. and a *date* is a datestamp or a cons cell `(`s `.` e`)` where s and e are each datestamps, $s \leq e$. The cons cell defines the half-open range $t \in [s, e)$, $s \leq t < e$. Affiliated with each bookmark is metadata comprising a list of *tags*. Each *tag* is a three element list `(`n α v`)` where n (a string) denotes a namespace, α (a string) is an attribute within that namespace and v is a *Motile* value associated with p in n. For the sake of simplicity we restrict a value to be a primitive value (boolean, number, symbol, string, byte string), a list, vector, or a recursive construction of same. A tag namespace defines (perhaps informally) a set of recognized identifier names and their types. The interpretation of a tag value is namespace- and attribute-dependent.

Figure 9.1 contains example tags for the namespaces `location`, `rollercoaster`, `restaurant`, and `siunit`. The `location` namespace contains attributes such as `street`, `city`, `state`, `country`, `latitude` and `longitude` where the values for `longitude` and `latitude` are in degrees. The `photo` namespace with attributes: `format` (with values, `"JPG"`, `"PNG"`, or `"GIF"`), `height` and `width` (both in pixels) may be used to describe photographs. Tag namespaces, such as `rollercoaster` or `restaurant`, may be ad-hoc or left open to experimental expansion and variation. Here we see that the namespace `rollercoaster` has a `height` identifier whose value is given in meters and that the `restaurant` namespace has a `cuisine` attribute that specifies the style or method of cooking offered by a restaurant.

In the examples to follow I assume that clients define a simple set of access and query functions, enumerated in Figure 9.2, to inspect individual tags. The three functions, `tag/namespace`, `tag/attribute`

```
("location" "country" "France")
("roller coaster" "height" 99.06)
("location" "latitude" +33.6454)
("location" "longitude" -117.8426)
("restaurant" "cuisine" "Tex-Mex")
("photo" "format" "JPG")
("photo" "height" 1200)
("photo" "width" 1600)
```

Figure 9.1: Sample bookmark tags.

```
(define (tag/namespace t) (first t))                                    1
(define (tag/attribute t) (second t))                                   2
(define (tag/value t) (third t))                                        3
(define (tag/namespace=? t s) (string=? (tag/namespace t) s))           4
(define (tag/attribute=? t s) (string=? (tag/attribute t) s))           5
```

Figure 9.2: A few utility functions for inspecting and querying tags.

and `tag/value` (lines 1–3), are accessors for the elements of tags. The two functions, `tag/namespace=?` and `tag/attribute=?` (lines 4–5), test for a given tag namespace or attribute.

Finally, a *bookmark* is a *Motile* record containing the fields shown in Table 9.3.

Table 9.3: The fields of a bookmark.

Name	Value	Description
`id`	*id*	Unique record identifier
`created`	*datestamp*	Creation date and time of the bookmark
`user`	*string*	Public token of the user that created the bookmark
`url`	*string*	URL of the bookmark
`tags`	*list*	$(t_1 \cdots t_m)$ where each t_i is a tag

9.4 A Motile API for Delicious

A *Motile*/*Island* service provider can gracefully extend and adapt a service API in many ways. One strategy is to delegate the evolution and adaptation of the service API to the service clients; observe how clients overcome the perceived shortcomings or inconsistencies of the API and allow their innovations to guide alterations and improvements. For example:

- Add or extend service primitives to include use cases that are not accommodated by the current service API

- Add service primitives that address commonly observed patterns of extension to improve performance, reduce resource consumption, or ease the burdens of client adaptation
- Eliminate, generalize, or refactor service primitives that are infrequently used
- Refine existing service primitives to eliminate unanticipated edge cases

In other words, a service provider can crowdsource the evolution of its API and offer distinct versions of the API a_i (using CURLs that reference distinct execution sites S_{a_i} for the sake of testing, experimentation, and evaluation.

Open-source software developers implement libraries for almost every imaginable domain and the culture of open-source encourages those who incorporate these libraries into their own work to repay the gifting with bug reports, constructive criticism, patches, and suggestions for improvements. These artifacts, captured in issue trackers, mailing lists, test reports, blogs and the commits of revision control systems, are invaluable for experimental and data-driven software engineering. However, mobile code systems such as *Motile*/*Island* offer yet another perspective on the evolution of software — a service provider can observe, in detail, how a service API is employed by its service clients.

There are two opportunities for observation, one static and the other dynamic. Static observation occurs when mobile code arrives at an island for execution and is recompiled from its on-the-wire network representation into closures. At this point, an island can detect closures that reference island-specific APIs of interest and these closures can be captured and archived for later analysis by program comprehension tools [4].[5]

Dynamic observation of visiting closures that invoke elements of a provider API may give service providers additional insight into client-generated extensions. Service providers may want to enforce *computational contracts*, that go beyond verifying a function's input and output conditions to include events that occur within the function such as *prohibiting or verifying that certain functions are called, checking access permissions, time or memory constraints, or interaction protocols* [211].

The static analyses (generated by program comprehension tools) of visiting closures can be combined with the dynamic observations of those same closures to create detailed, trace-like views of the behavior

[5] There is a single instruction in the *Motile* assembly graph (MAG) that references a global binding by name. It would be a straightforward extension to include to those references, say as a tuple of symbols, in the closure representation of visiting mobile code.

of client closures that detail when, how, and why domain-specific API functions are invoked [74]. In this sense, service clients inform service providers of their intentions and interests. The combination of static analysis and dynamic measurement of visiting mobile code suggests an alternative model of software development in which the alteration, growth, and extension of an API is informed (and indirectly crowdsourced) by the mobile code that clients submit to providers.[6]

What should be the starting point for a client-driven API? I suggest that a minimal API can maximize client innovation as it imposes fewer constraints, offers greater scope for interesting forms of specialization, and can suggest designs, use cases, or optimizations that might not arise otherwise. A more expansive API imposes the preconceptions of the service developers on its clients at the risk of closing off avenues of diversity. With this in mind, I adopted three constraints for a *Motile*-based alternative to the Delicious API :

- Reduce the service-provider API to a bare minimum: one function each for search, read, and create
- Restrict the binding environment of the service execution site to a pure functional subset[7] and
- Assume that only remote evaluation is available to service clients

These constraints are the backdrop for a rich, purely functional API for a Delicious-like bookmark service — constructed by *clients* from a microscopic collection of domain-specific, service-provider primitives (three in all) and the basic functions found in any modern Scheme. Since the purpose of *Motile* remote evaluation is to sharply confine the communication capability of the visiting mobile code the primitives for both inter- and intra-island communication are absent from the provider's execution site, as is `curl/spawn`, the primitive for spawning islets (either on- or off-island) and its lesser cousins, the variants of `curl/remote` (see Section 5.10). Consequently, no visiting client closures have the luxury of constructing concurrent data extraction pipelines or exploiting transports (see Section 3.2.1) as buffers for intermediate data products.

The fundamental organizing principle of the database is temporal: bookmarks are ordered by the time of their creation, from most recent backwards to the oldest. Users may add new bookmarks but no bookmark is ever deleted (hence the database is persistent and a user/timestamp pair $(u\ .\ t)$ amounts to a checkpoint of the database for user u at time t. More recent bookmarks take precedence over older

[6] See [47] for an excellent survey of recent work in program comprehension via dynamic analysis.

[7] In line with the BASELINE binding environment defined by *Motile/Island*as the starting point for constructing binding environments.

bookmarks but all bookmarks, irrespective of age, are available via temporal search. The bookmarks of a user u may be inspected (but not modified) by any user $v \neq u$ and only u may add to the bookmarks of u. With these constraints in mind I model a *Motile* API roughly comparable to a RESTful API proposed by James [135] as I shift the burden of formulating service extensions to the service consumer.

Only three API functions are available for searching, creating, and manipulating bookmarks and tags:

- `bookmarks/iterate` to read portions of the bookmarks database
- `bookmark/get` to fetch a specific bookmark
- `bookmark/post` to create a bookmark.

Each is detailed below.[8]

The bookmark database is append-only and persistent. Bookmarks may be added but not deleted or updated and tags are immutable. To update the tags of a bookmark b with `url` x and `tags` $(t_1 \cdots t_m)$ a user creates a new bookmark b' with `url` x but with different tags. For example to delete tag t_k of bookmark b posts a new bookmark b' with `url` x and `tags` $(t_1 \cdots t_{k-1}\ t_{k+1} \cdots t_m)$. Consequently, the addition and deletion of tags with respect to a bookmark is atomic.

The client-side extensions rely on iterators defined by the API of the execution sites of the bookmarks service. An *iterator* g is a specialized continuation (in these examples implemented as a thunk) that implements a traversal of a collection of values $x_1, x_2, \ldots$.[9] On success (g) returns a pair $(x\ .\ h)$ where x is a value from the traversal and thunk h is the successor of iterator h, itself an iterator that returns the successor *next* value/iterator pair (if any) in the traversal. If no more values x remain in the traversal then (g) returns `#f`. Iterator g is idempotent and each invocation (g) of g returns the same pair $(x\ .\ h)$. In this sense g is a both a continuation of the iteration and a snapshot of the iteration; g can be called at any point in the future to "restart" the iteration at exactly the same point in the traversal.

`(bookmarks/iterate` $u\ d$`)` is a *generator* (a function that returns a function) that returns an iterator g over the set of bookmarks for the user whose public token is u over *date* d. The iterator traverses the bookmarks of user u in temporal order from most recent (first) to oldest (last). If d is a single datestamp then the set of elements is restricted to the day given by the datestamp. If $d = (s\,.\,e)$ then the set of

[8] As a matter of convenience I assume that wherever a datestamp is called for it may also be given as a string in ISO 8601 format, for example, `"2015-11-28T10:16:42Z"`.

[9] The order of the traversal depends upon the structure of the collection and the semantics of the traversal, for example, the iterator of a tree may perform a depth-first or breadth-first walk, thus the traversal is some sequence $x_{k_1}, x_{k_2}, \ldots$ of values x_i.

elements is restricted to the time period $[s, e)$. If d is `#f` then the traversal begins with the most recent bookmark. The iterator (g) returns `#f` repeatedly if the set of bookmarks is exhausted otherwise it returns $(x\ .\ h)$ where x is the next bookmark in the temporal order and h is the successor iterator.

`(bookmark/get` r`)` returns the bookmark whose `id` is r. If no such bookmark exists then `#f` is returned.

`(bookmark/post` $s\ u\ t$`)` creates a bookmark b for the user whose secret token is s. The `url` of b is u and the tags of b are the list $t = (t_1 \cdots t_m)$, each t_i a tag. On success the return value is the `id` of b; otherwise `#f`.

9.5 Client-defined Extensions of the Motile API for Delicious

I examine a selection of client-constructed functions based on the minimalist API sketched above.[10]

```
(begin                                                                  1
  (define (bookmark/latest user)                                        2
    (let* ((g (bookmarks/iterate user #f)) ; Most recent bookmark first.  3
           (pair (g))) ; May return #f if no bookmarks.                 4
      (and pair (car pair)))) ; Return bookmark if any, otherwise #f.   5
                                                                        6
  (curl/block/remote ; Blocking remote evaluation.                      7
   Bookmarks@ ; CURL for bookmark service.                              8
   (lambda () ; Thunk for remote evaluation.                            9
     (let ((b (bookmark/latest "alice")))                               10
       (and b (time-to-ISO8601 (:: b created)))))))                     11
```

Figure 9.3: A client-defined routine for discovering the latest bookmark (lines 2–7). Lines 10–16 illustrate a remote evaluation to obtain the datestamp of the latest update for user `alice`.

Discovering a user's latest update is a common requirement in social service APIs and is easily constructed from the available primitives as illustrated in Figure 9.3. The function `bookmark/latest` (lines 2–5) returns the most recent bookmark posted by a given *user* or, if that user has never posted a bookmark then `#f`. The thunk (lines 9–11) of the remote evaluation (lines 7–11) obtains the latest bookmark for user Alice and returns its creation date in ISO 8601 format (a string, for example `"2015-02-22"`) or `#f` if Alice has never created a bookmark. The mobile code transmitted to the bookmark service relies on two service-side functions `bookmarks/iterate` and `time-to-ISO8601`, a function that converts Unix epoch time to an ISO 8601 representation.

[10] For the sake of brevity and ease of presentation I assume that the CURL of the bookmark service is defined elsewhere and bound (somewhere within lexical scope) to the *Motile* name `Bookmarks@`. User public tokens for the API are given as simple strings such as `"alice"` or `"bob"`.

Of more interest are client-defined combinators for client-driven extensions to the service-side, iteration primitive `bookmarks/iterate`. I discuss four combinators below: fold, filter, map, and pipe.

9.5.1 Fold

```
(define (iterate/fold g f s)                                                    1
  (let ((pair (g))) ; pair is (x . h), x an object and h a successor to g.      2
    (if pair                                                                    3
        (iterate/fold (cdr pair) f (f (car pair) s)) ; Fold again.              4
        seed))) ; Iterator is exhausted.                                        5
```

Figure 9.4: `iterate/fold`, a client-defined combinator.

`(iterate/fold` g f s`)` takes three arguments: an iterator g, a folding function f, and s, an initial seed value for f. For each object x generated by g the folding function is applied to x and the current seed as `(`f x s`)`. The return value of `(`f x s`)` is the successive seed for the next iteration of the fold, in effect an accumulator of the successive return values of f.

9.5.2 Filter

```
(define (iterate/filter g f)                                                    1
  (lambda () (subiterate/filter g f))) ; Return iterator that filters.          2
                                                                                3
(define (subiterate/filter g f) ; Traverse until an object passes filter f.     4
  (let ((pair (g))) ; (x . h) or #f. h successor iterator to g.                 5
    (if pair                                                                    6
        (if (f (car pair)) ; Test object x against filter f.                    7
            (cons                                                               8
             (car pair) ; Object x.                                             9
             (lambda () (subiterate/filter (cdr pair) f))) ; Successor to g.    10
            (subiterate/filter (cdr pair) f)) ; x failed filter.                11
        #f))) ; Iterator is exhausted.                                          12
```

Figure 9.5: `iterate/filter`, a client-defined combinator.

`(iterate/filter` g f`)` is a generator which takes two arguments, an iterator g and a predicate f, returning an iterator g_f whose return value is the first pair `(`x `.` h_f`)` in the traversal order of g for which `(`f x`)` is `#t` and h_f is the successor iterator. If the traversal is exhausted then `#f` is returned. In other words if the sequence $x_1, x_2, \ldots$ is the traversal order of g then the traversal order of g_f is $x_{j_1}, x_{j_2}, \ldots$ where $j_1 < j_2 < \cdots$, x_{j_i} precedes $x_j i+1$ in the traversal order of g, $f(x_{j_i})$ is `#t`, and if any x_i satisfies f then x_i appears in the sequence $x_{j_1}, x_{j_2}, \ldots$.

9.5.3 Map

```
;; Generate a derived iterator that maps f over iterator g.
(define (iterate/map g f)
  (lambda () (subiterate/map g f)))

;; Map every x in the traverse of g to (f x).
(define (subiterate/map g f)
  (let ((pair (g)))
    (if pair
        ; pair is (x . h), x an object and h a successor to g.
        (cons
         (f (car pair)) ; (f x)
         (lambda () (subiterate/map (cdr pair) f)))
        #f))) ; Iterator is exhausted.
```

Figure 9.6: `iterate/map`, a client-defined combinator.

`(iterate/map` g f`)` is a generator which takes two arguments: an iterator g and a transform f returning `((`f x`) .` h`)` for each `(`x `.` h`)` in the traversal order of g. If the traversal is exhausted then `#f` is returned.

9.5.4 Pipe

```
;; Construct a "pipeline" of generators = (g0 g1 ... gN)
;; where g0 is a zero-argument lambda and each gi thereafter is a
;; single argument lambda.
;; For example (iterate/pipe g0 g1 g2 g3) becomes (g3 (g2 (g1 (g0)))).
(define (iterate/pipe generator . generators)
    (fold
     (lambda (x g) (x g)) ; Fold function.
     (generator)          ; Seed
     generators           ; List of generators to fold over.
```

Figure 9.7: `iterate/pipe`, a client-defined combinator.

`(iterate/pipe` g_0 g_1 $\cdots$ g_m`)` accepts $m+1$ arguments for $m > 0$ where:

- g_0 is a zero-argument function for which `(`g_0`)` returns an iterator
- Each g_i, $1 \leq i < m$ is a single-argument function for which `(`g_i `(`g_{i-1}`))` returns an iterator, and
- g_m is any single-argument function that accepts an iterator as a parameter

The effect is to compose functions $g_0, g_1, \ldots, g_m$ as a single nested invocation where

$$\texttt{(iterate/pipe } g_0\ g_1 \cdots g_m\texttt{)} \equiv \texttt{(}g_m\ \texttt{(}\cdots\ \texttt{(}g_2\ \texttt{(}g_1\ \texttt{(}g_0\texttt{)))))}$$

in other words, a functional pipeline that starts with g_0 and ends with g_m, where the return value of `iterate/pipe` is the return value of $(g_m\ (\cdots\ (g_2\ (g_1\ (g_0)))))$.

9.5.5 Count

The "classic" Delicious web API allows a user to count the number of bookmarks posted per day with optional filtering by tag. I duplicate and generalize that functionality here. Counting bookmarks-per-day captures variations in a user's engagement with a bookmark service and is illustrated (without filtering by tag) in Figure 9.8. This client-defined query computes the number of bookmarks constructed by user Bob per day over the month of December, 2014. The value of that query is a *Motile* persistent hash table whose keys are days (strings `"2014-12-19"` for example) and whose values are integers $n > 0$ (days on which Bob did not construct bookmarks are ignored).[11]

```
(begin                                                                    1
  (define (bookmark-to-day b)                                             2
    (let ((s (time-to-ISO8601 (:: b created))))                           3
      (substring s 0 9))) ; "2015-11-28T10:16:52Z" => "2015-11-28".       4
                                                                          5
  (define (total-per-day day h)                                           6
    (let ((count (hash/get h day 0))) ; Return 0 if day not in hash.      7
      (hash/cons h day (+ count 1)))) ; Add day/count pair to hash.       8
                                                                          9
  (curl/remote/block ; Dispatch remote evaluation, block for return value. 10
   Bookmarks@ ; CURL for bookmark service.                                11
   (lambda () ; Client thunk for evaluation.                              12
     (iterate/pipe                                                        13
      (bookmarks/iterate                                                  14
       "bob" ; Public user token for Bob.                                 15
       (cons ; Time period is December, 2014.                             16
        "2014-12-01T00:00:00Z" "2015-01-01T00:00:00Z"))                   17
                                                                          18
      (lambda (g) (iterate/map g bookmark-to-day))                        19
                                                                          20
      (lambda (g) (iterate/fold g total-per-day hash/equal/null))))))     21
```

Figure 9.8: Generate a persistent hash map of d/n pairs where d is a day (a string such as "2015-09-29") and $n > 0$ is the total number of bookmarks introduced on day d. We assume that the client-defined combinators `iterate/pipe`, `iterate/map`, and `iterate/fold` of Figures 9.7, 9.6, and 9.4 respectively are available in an enclosing lexical scope.

The *Motile* code of Figure 9.8 demonstrates a three-stage pipeline — *generate* bookmarks, *map* the bookmarks into the desired representation, and then *fold* that representation — within the confines of

[11] If Bob did not contribute any bookmarks over the month of December, 2014 then an empty persistent hash table is returned.

the client combinator `iterate/pipe` (lines 13–21). The *generate* stage (lines 14–17) is implemented by the service-side primitive `bookmarks/iterate` , the *map* stage (line 19) is implemented by the client combinator `iterate/map` and the client function `bookmark-to-day` (lines 2–4), and the *fold* stage (line 21) is implemented by the client combinator `iterate/fold` and the client folding function *total-per-day* (lines 6–8). Note that since *Motile* hash tables are persistent and functional the hash table h is left unchanged and the value returned by `hash/cons` (line 8) is a fresh and distinct hash table h' that shares the structure of h to the fullest extent possible modulo the update.

Since the client combinators and the two client functions, `bookmark-to-day` and `total-per-day`, are defined within the lexical scope of the client thunk they are carried along when the client thunk (lines 12–21) is transmitted to the bookmarks service for remote evaluation. The only service-side primitives that the client query requires are `time-to-ISO8601` (line 3), a service-side function that converts a Unix epoch time to its ISO 8601 representation, and `bookmarks/iterate` (line 14–17).

The value returned to the client is a persistent hash table of key/value pairs d/n where d is a day (say string `"2014-12-09"`) and $n > 0$ is the total number of bookmarks with that creation date.

9.5.6 Index by Date

The same pattern, — *generate*, *map* and *fold* — again within the context of combinator `iterate/pipe`, can be used to by clients to create indexes of their own bookmarks or the bookmarks of others. Figure 9.9 illustrates a client-generated index whose keys are dates (such as "2014-10-30") and whose values are a list of the ids of the bookmarks created on that day.[12]

As in the previous example (Section 9.5.5) bookmarks are generated (lines 116–19) by the service-side primitive `bookmarks/iterate`.

An anonymous function (line 24)

```
(lambda (b) (cons (bookmark-to-day b) (:: b id)))
```

maps (lines 21-24) each bookmark b to the pair `(`d `.` x`)` where d is the day on which b was created (as a string, say "2014-10-29") and x is the *id* of b.

Folding (line 26) is performed by `iterate/fold` (line26) and `collect-per-day` (lines 7–13). The latter accepts two arguments: a pair `(`d `.` x`)` where d is the creation day of a bookmark b and x is the *id* of b

[12] Recall that the service-side primitive `(bookmark/get` x`)` returns the bookmark whose id is x.

```
(begin                                                                          1
  (define (bookmark-to-day b)                                                   2
    (let ((s (time-to-ISO8601 (:: b created))))                                 3
      (substring s 0 9))) ; "2015-11-28T10:16:52Z" => "2015-11-28".             4
                                                                                5
  (define (collect-per-day pair h) ; Seed h is a persistent hash table.         6
    (let* ((day (car pair))  ; pair is (day . id)                               7
           (x   (cdr pair))                                                     8
           (values (hash/get h day null))) ; Returns null if day not in h.      9
      (hash/cons h day (cons x values))))  ; Accumulate id for day.             10
                                                                                11
  (curl/remote/block ; Dispatch remote evaluation, block for return value.      12
   Bookmarks@ ; CURL for bookmark service.                                      13
   (lambda () ; Client thunk for evaluation.                                    14
     (iterate/pipe                                                              15
      (bookmarks/iterate                                                        16
       "alice" ; Public user token for Alice.                                   17
       (cons ; Time period is September-December, 2014.                         18
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z")))                        19
                                                                                20
     (lambda (g)                                                                21
       (iterate/map                                                             22
        g                                                                       23
        (lambda (b) (cons (bookmark-to-day b) (:: b id)))))                     24
                                                                                25
     (lambda (g) (iterate/fold g collect-per-day hash/equal/null)))))           26
```

Figure 9.9: Generate a persistent hash map of d/X pairs where d is a day (such as "2015-09-29") and $X = (x_m\ x_{m-1}\ \ldots\ x_1)$ is a list of bookmark ids. If bookmarks $b_1,\ldots,b_m$ were created on day d then x_i is the id of b_i. We assume that the client-defined combinators `iterate/pipe`, `iterate/map`, and `iterate/fold` of of Figures 9.7, 9.6, and 9.4 respectively are available in an enclosing lexical scope.

and a seed h, a persistent hash table. In each invocation the fold function returns a hash table h' identical to h except that h' contains the key/value pair $d/(x\ .\ X)$ when $d/X \in h$; otherwise, if the key d did not appear in h then $d/(x) \in h'$.

The client receives a hash table with key/value pairs $d/(x_m\ \cdots\ x_1)$ where x_i is the id of a bookmark b_{x_i} created on day d. The list is in reverse temporal order from oldest, x_m, to most recent, x_1, on day d. The customary temporal order, from most recent to oldest, is easily restored in one of two ways. It can be reversed service-side if the client wraps the call `(iterate/pipe ⋯)` (lines 15–26) as:

```
(hash/map (iterate/pipe ⋯) (lambda (day ids) (cons day (reverse ids)))))
```

or alternatively, amends the client side by wrapping lines 12–26

```
(curl/remote/block Bookmarks@ (lambda () ⋯))
```

with a call to `hash/map` as

```
(hash/map (curl/remote/block ···) (lambda (day ids) (cons day (reverse ids))))
```

In other words, the client can shift portions of the work from client to provider or back again as circumstances dictate.[13]

9.5.7 Reverse Index by Tag

```
(begin                                                                          1
  (define (tag/namespace t) (first t))                                          2
  (define (tag/attribute t) (second t)) ; Unused.                               3
  (define (tag/value t)     (third t))  ; Unused.                               4
                                                                                5
  (define (collect-tags pair h)  ; Seed h is persistent hash table.             6
    (let loop ((id (car pair)) (tags (cdr pair)) (h h))                         7
      (if (null? tags)                                                          8
          h ; No more tags for reverse index.                                   9
          (let* ((alpha (tag/namespace (car tags)))                             10
                 (ids   (hash/get h alpha null)))                               11
            (loop                                                               12
             id (cdr tags)                                                      13
             (if (member id ids) h (hash/cons h alpha (cons id ids))))))))      14
                                                                                15
  (curl/remote/block ; Dispatch remote evaluation, block for return value.      16
   Bookmarks@ ; CURL for bookmark service.                                      17
   (lambda () ; Client thunk for evaluation.                                    18
     (iterate/pipe                                                              19
      (bookmarks/iterate                                                        20
       "alice" ; Public user token for Alice.                                   21
       (cons ; Time period is September-December, 2014.                         22
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z"))                         23
                                                                                24
      (lambda (g) (iterate/map g (lambda (b) (cons (:: b id) (:: b tags)))))    25
                                                                                26
      (lambda (g) (iterate/fold g collect-tags hash/equal/null))))))            27
```

Figure 9.10: Generate a persistent hash map of α/X pairs where α is a tag namespace and $X = (x_m \cdots x_1)$ is a list of bookmark ids. Each bookmark b_{x_i} contains at least one tag whose namespace is α. The list X is ordered from oldest (x_m) to most recent (x_1). We assume that the client-defined combinators `iterate/pipe`, `iterate/map`, and `iterate/fold` of of Figures 9.7, 9.6, and 9.4 respectively are available in an enclosing lexical scope.

Using the pattern of generate, map, and fold clients can construct a variety of specialized indexes. Figure 9.10 demonstrates a reverse index whose keys α are the namespaces of bookmark tags and whose values are lists of bookmark ids $(x_m \cdots x_1)$ where each bookmark b_{x_i} contains at least one tag whose namespace is α. The ids are in reverse temporal order, that is, bookmark b_{x_m} is the oldest bookmark with

[13] For example, if the service is heavily loaded the client may reduce latency by shifting to client-side that portion of the computation that does not require the service-specific API of the provider.

a tag in namespace α and b_{x_1} is the most recent bookmark with a such a tag.

In this form the index is ideal for locating the oldest bookmark containing a tag with namespace α.[14] A varient of this index, whose construction is shown in Figure 9.11, can efficiently resolve compound queries over tags, for example, *the set of all bookmarks that refer to photographs of roller coasters*. Figure 9.10 differs from Figure 9.11 in one critical detail: the folding function of the latter (lines 4–10) uses a persistent unordered set to capture bookmark *ids* (lines 9–10) rather then the simple lists of the former example. In other words, both examples return hash tables whose keys are tag namespaces but in Figure 9.10 the hash table values are ordered lists (from oldest to most recent) of bookmark ids while in Figure 9.11 the values are unordered sets of bookmark ids.

```
(begin                                                                          1
  (define (tag/namespace t) (first t))                                          2
                                                                                3
  (define (collect-tags pair h)  ; Seed h is persistent hash table.             4
    (let loop ((id (car pair)) (tags (cdr pair)) (h h))                         5
      (if (null? tags)                                                          6
          h ; No more tags in this bookmark for reverse index.                  7
          (let* ((alpha (tag/namespace (car tags))) ; Hash key.                 8
                 (ids   (hash/get h alpha set/equal/null))) ; Hash value.       9
            (loop                                                               10
             id (cdr tags)                                                      11
             (if (set/contains? ids id))                                        12
                 h ; Set of ids already contains id.                            13
                 (hash/cons h alpha (set/cons ids id)))))))                     14
                                                                                15
  (curl/remote/block ; Dispatch remote evaluation, block for return value.      16
   Bookmarks@ ; CURL for bookmark service.                                      17
   (lambda () ; Client thunk for remote evaluation.                             18
     (iterate/pipe                                                              19
      (bookmarks/iterate                                                        20
       "alice" ; Public user token for Alice.                                   21
       (cons ; Time period is September-December, 2014.                         22
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z"))                         23
                                                                                24
      (lambda (g) (iterate/map g (lambda (b) (cons (:: b id) (:: b tags)))))    25
                                                                                26
      (lambda (g) (iterate/fold g collect-tags hash/equal/null))))))            27
```

Figure 9.11: Generate a persistent hash map of α/X pairs where α is a tag namespace and X is a persistent unordered set of bookmark ids $x_1,\ldots,x_m$ for bookmarks b_{x_i} that contain at least one tag whose namespace is α. We assume that the client-defined combinators `iterate/pipe`, `iterate/map`, and `iterate/fold` of of Figures 9.7, 9.6, and 9.4 respectively are available in an enclosing lexical scope.

[14] As sketched in Section9.5.6 above it is trivial to convert this index to the opposite temporal order in which the most recent bookmark id appears first.

Let h be a reverse index constructed in the manner of Figure 9.11. A client in possession of h can, for example, fetch *the set of all bookmarks that refer to photographs of roller coasters* as Figure 9.12 illustrates. By combining pure predicates, the standard logical connectives `and`, `or` and `not`, and the basic set operations of `set/contains?`, `set/union`, `set/intersection` and `set/difference` a client can pose complex selections against a set-based reverse index. More generally, client-generated indexes can take many forms and, as the examples of Sections 9.5.6 and 9.5.7 suggest, are easily constructed. Here we can see the beginnings of a service ecology in which independent indexing services arise around a bookmark service. The simplicity and generality of the bookmark service API sketched here (comprising three functions: `bookmark/iterate`, `bookmark/get` and `bookmark/post`) grants COAST-based clients the freedom and independence to implement and deploy sophisticated indexing services that serve special needs or offer domain-specific access to bookmarks that the bookmark service does not provide.[15] These simple examples lie at the heart of COAST; every service can be a programmable, extensible platform — the underlying model elevates innovation to a network effect where development and evolution can be client- rather than provider-driven.

```
(begin                                                                    1
  (let* ((roller-coaster (hash/get h "roller-coaster" set/equal/null))    2
         (photo (hash/get h "photo" set/equal/null))                      3
         (ids (set/intersection roller-coaster photo)) ; Wanted ids.      4
         (list-of-ids (set-to-list ids))) ; Convert set of ids to list.   5
    (if (null? list-of-ids)                                               6
        null ; List is empty.                                             7
                                                                          8
        (curl/return/block ; Dispatch thunk for remote evaluation.        9
         Bookmarks@ ; CURL of bookmark service.                           10
         (lambda () (map bookmark/get list-of-ids))))))) ; Thunk.         11
```

Figure 9.12: Let h (lines 2 and 3) be a reverse index whose keys are tag namespaces α and whose values are unordered sets of bookmark ids x such that each bookmark b_x contains a tag whose namespace is α. A client, holding a copy of h, determines the ids of all bookmarks that refer to photographs of roller coasters (lines 2–4) and converts that set of ids to a list of ids $(x_1 \cdots x_m)$, $m \geq 0$ (line 5). If the list of ids is empty then the empty list (line 7) is the value of `(begin` $\cdots$`)`; otherwise the client dispatches a remote evaluation to a bookmark service to fetch those bookmarks and return them as a list $(b_{x_1} \cdots b_{x_m})$ (lines 9–11).

[15] This does not necessarily reflect ill on the bookmark service as their focus on core data storage for bookmarks and allowing others to supply and maintain indexes may be a strategic choice.

9.5.8 Filtering

The RESTful query primitive defined by James [135] allows a user to simultaneously query on both a temporal period in which the bookmark was created and a single bookmark tag. However, far more flexible and general filtering is easily incorporated in a pipeline by way of the client-defined combinator `iterate/filter` that passes only those bookmarks b for which a given predicate $(p\ b)$ is true. To simplify the construction of filters against bookmark tags our client also defines an additional combinator `list/find`, shown in Figure 9.13, that returns the first item x in a list of *items* for which predicate $(p\ x)$ is `#t`. If no item satisfies p then `#f` is returned.[16] `list/find` is used to search through the list of tags (possibly empty) that appears in each bookmark.

```
(define (list/find p items)                                    1
  (cond                                                        2
   ((null? items) #f)                                          3
   ((p (car items)) (car items)))                              4
   (else (list/find p (cdr items)))))                          5
```

Figure 9.13: Function `list/find` returns the first item x in a list `items` that satisfies predicate $(p\ x)$. If no such item is present in the list then it returns `#f`.

```
(define  (has-namespace b n))                                  1
  (list/find (lambda (t) (tag/namespace=? t n)) (:: b tags)))  2
                                                               3
(define (has-namespace-attribute b n a)                        4
  (list/find                                                   5
   (lambda (t) (and (tag/namespace=? t n) (tag/attribute=? t a)))  6
   (:: b tags)))                                               7
                                                               8
(define (bookmark-is? b n) (and (has-namespace b n) #t))       9
                                                               10
(define (country-of-bookmark? b . places)                      11
  (let ((t (has-namespace-attribute? b "location" "country")))  12
     (and (member (tag/value t) places string-ci=?) #t)))      13
```

Figure 9.14: Client-defined helper predicates for searching and querying bookmark tags.

Figure 9.14 contains four client-defined predicates for searching and querying bookmark tags:

- `(has-namespace` b n`)` (lines 1–2) returns the first tag t of bookmark b such that $t = (n\ \alpha\ v)$ for some attribute α and value v. If no such tag exists in b then `#f` is returned.

[16] `list/find` is ambiguous if $(p$ `#f`$)$ returns `#t` however, the examples here guarantee that `#f` never appears in the list of *items*.

- `(has-namespace-attribute` b n α`)` (lines 4–7) returns the first tag t of bookmark b such that $t = (n\ \alpha\ v)$ for some value v. If no such tag exists in b then `#f` is returned.
- `(bookmark-is?` b n`)` (line 9) is a predicate that returns `#t` if bookmark b contains a tag $t = (n\ \alpha\ v)$ for some attribute α and value v; otherwise, if no such tag exists in b then `#f` is returned.
- `(country-of-bookmark?` b $c_1 \cdots c_m$`)` (lines 11-13) returns `#t`if bookmark b contains a tag $t =$ `("location" "country"` v`)` where $v = c_i$ for some i, $1 \leq i \leq m$, each c_i a country name given as a string. If no such tag exists in b then `#f` is returned.

The first example, illustrated in Figure 9.15, is a remote evaluation that returns a list of Alice's bookmark ids x_i for which bookmark b_{x_i} was issued over the period September–December, 2014 and contains a `location` tag — a restriction enforced by the `iterate/filter` (lines 11–12) within the `iterate/pipe` (lines 5–15). This is the equivalent of the Delicious search primitive that combines temporal confinement with a match on a single, specific tag.

```
(begin                                                                          1
  (curl/remote/block ; Dispatch remote evaluation, block for return value.      2
   Bookmarks@ ; CURL for bookmark service.                                      3
   (lambda () ; Client thunk for remote evaluation.                             4
     (iterate/pipe                                                              5
      (bookmarks/iterate                                                        6
       "alice" ; Public user token for Alice.                                   7
       (cons ; Time period is September-December, 2014.                         8
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z"))                         9
                                                                                10
      (lambda (g)                                                               11
        (iterate/filter g (lambda (b) (bookmark-is? b "location"))))            12
                                                                                13
      (lambda (g)                                                               14
        (iterate/fold g (lambda (b seed) (cons (:: b id) seed)) null))))))      15
```

Figure 9.15: Search Alice's bookmarks over the period September-December, 2014 for all bookmarks b_{x_i} that refer to a location return a list of their *ids* $(x_m \cdots x_1)$. If no such bookmarks exist then the empty list `()` is returned to the client. We assume that the client helper functions of Figures 9.13 and 9.14 are contained in an outer lexical scope.

The second example, illustrated in Figure 9.16, is a remote evaluation that returns a list of Allice's bookmark ids x_i for which bookmark b_{x_i} was issued over the period September–December, 2014 and contains a `location` tag for either `"France"` or `"Germany"` — a restriction enforced by the `iterate/filter` (lines 11–13) within the `iterate/pipe` (lines 5–16). This is a generalization of of the Delicious search primitive that combines temporal confinement with a match on a single, specific tag, made possible in

part by the more elaborate tag structure adopted here. This search cannot be expressed in the classic Delicious API or a proposed RESTful equivalent [135].

```
(begin                                                                    1
  (curl/remote/block ; Dispatch remote evaluation, block for return value.  2
   Bookmarks@ ; CURL for bookmark service.                                  3
   (lambda () ; Client thunk for remote evaluation.                         4
     (iterate/pipe                                                          5
      (bookmarks/iterate                                                    6
       "alice" ; Public user token for Alice.                               7
       (cons ; Time period is September-December, 2014.                     8
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z"))                     9
                                                                           10
      (lambda (g)                                                          11
        (iterate/filter                                                    12
         g (lambda (b) (country-of-bookmark? b "France" "Germany"))))      13
                                                                           14
      (lambda (g)                                                          15
        (iterate/fold g (lambda (b seed) (cons (:: b id) seed)) null))))))  16
```

Figure 9.16: Search Alice's bookmarks over the period September-December, 2014 for all bookmarks b_{x_i} that refer to a location in France or Germany and return a list of their *ids* $(x_m \cdots x_1)$. If no such bookmarks exist then the empty list `()` is returned to the client. We assume that the client helper functions of Figures 9.13 and 9.14 are contained in an outer lexical scope.

An `iterate/pipe` may contain multiple filters $f_1, \ldots, f_m$ with each filter f_i acting as the element of a conjunct $f_1 \wedge \cdots \wedge f_m$. Any boolean expression containing boolean variables and pure functional predicates over a bookmark b can be normalized into conjunctive normal form $e_1 \wedge \cdots \wedge e_m$ and each e_i can be rewritten as a filter f_i

$$\texttt{(lambda (}g\texttt{) (iterate/filter }g\texttt{ (lambda (}b\texttt{) }e_i\texttt{)))}$$

To illustrate I amend the example of Figure 9.16 to further restrict its range to bookmarks that refer to a photo of a roller coaster in either France or Germany, where the search criteria are:

Alice's bookmarks (line 7) $\wedge$

created over September–December, 2014 (lines 8–9) $\wedge$

tagged with either France or Germany as the location (lines 11–13) $\wedge$

tagged as a photo (lines 15–16) $\wedge$

tagged with the topic "roller coaster" (lines 18–19)

as shown in Figure 9.17.

The ability to deploy highly-customized and specific filters embedded within an `iterate/pipe` far

```
(begin                                                                          1
  (curl/remote/block ; Dispatch remote evaluation, block for return value.      2
   Bookmarks@ ; CURL for bookmark service.                                      3
   (lambda () ; Client thunk for remote evaluation.                             4
     (iterate/pipe                                                              5
      (bookmarks/iterate                                                        6
       "alice" ; Public user token for Alice.                                   7
       (cons ; Time period is September-December, 2014.                         8
        "2014-09-01T00:00:00Z" "2015-01-01T00:00:00Z"))                         9
                                                                                10
      (lambda (g)                                                               11
        (iterate/filter                                                         12
         g (lambda (b) (country-of-bookmark? b "France" "Germany"))))           13
                                                                                14
      (lambda (g)                                                               15
        (iterate/filter g (lambda (b) (bookmark-is? b "photo"))))               16
                                                                                17
      (lambda (g)                                                               18
        (iterate/filter g (lambda (b) (bookmark-is? b "roller-coaster"))))      19
                                                                                20
      (lambda (g)                                                               21
        (iterate/fold g (lambda (b seed) (cons (:: b id) seed)) null))))))      22
```

Figure 9.17: Search Alice's bookmarks over the period September-December, 2014 for all bookmarks that refer to a photo of a roller coaster in either France or Germany. The search relies upon a small set of query functions for tags, shown in Figure 9.2 and an additional simple client-defined combinator, `list/find`, whose definition is given in Figure 9.13.

exceeds the expressive power of any bookmark service considered here, RESTful or otherwise. While it falls short of a relational algebra (omitting joins and products) the primitives illustrated here can perform selection (gathering bookmarks that satisfy one or more criteria), projection (restricting bookmarks to a subset of fields), and the renaming of attributes. With client-generated indexes one can produce products and joins. However, an efficient and general relational calculus is far more complex to implement. The solutions presented here have the virtue of simplicity, ease of implementation, and modest efficiency.

9.5.9 Append, Interleave, Merge, and Limit

All of the examples so far have dealt with a single stream of bookmarks, that is, the bookmarks of a single user over a single time span. However, clients can define combinators that fuse two or more generators into one, thereby constructing generators that may vary over users (the authors of bookmarks), time spans (over which the bookmarks were created), the order of in which bookmarks are presented, or arbitrary predicates (over the contents of bookmarks). I present three of many possible such combinators that:

- *Append* multiple streams $s_1, \ldots s_m$ into a single stream $s_1 + \cdots + s_m$.
- *Interleave* the bookmarks of multiple streams in a round-robin fashion, repeatedly issuing one bookmark from each stream s_i, $i = 1, \ldots, m$ until all streams are exhausted.
- *Merge* the bookmarks of multiple streams, preserving temporal order, until all streams are exhausted.

Finally, when combining streams, terminating a search early after the first $n > 0$ successful hits can reduce response latency, and I present such a combinator here.

Append

```
(define (iterate/append g1 g2)                                          1
  (lambda () (subappend g1 g2)))                                        2
                                                                        3
(define (subappend g1 g2)                                               4
  (let ((pair1 (g1)))                                                   5
    (if (pair? pair1)                                                   6
        ; g1 is not exhausted.                                          7
        (cons (car pair1) (lambda () (subappend (cdr pair1) g2)))       8
         ; g1 is exhausted.                                             9
        (g2))))                                                         10
```

Figure 9.18: The combinator `iterate/append` appends the stream of generator g_2 to the stream of generator g_1 (but without regard to temporal order) and is intended as the first argument of an `iterate/pipe`.

The simplest of these, `iterate/append`, is shown in Figure 9.18. Recall that the provider-side primitive `bookmarks/iterate` returns a generator, a zero-argument function, which in prior examples, appeared as the first argument f_1 to `(iterate/pipe` $f_1 \cdots f_m$`)`. Argument f_1 of `iterate/pipe` is distinguished from the others $f_2 \cdots f_m$ each of which accepts a single argument, namely the generator g_{i-1} that was the return value of f_{i-1}. The client-side combinator, `iterate/append`, is an alternative to `bookmarks/iterate` and its definition reflects that role; line 10 defines the thunk expected by `iterate/pipe`. Lines 1–7 define the generator proper where the the subgenerator g_2 is not invoked (line 7) until generator g_1 is first exhausted (lines 2–4).

Its use is demonstrated in Figure 9.19 where two distinct sources of bookmarks, those of Sam (bookmarks b_{s_i}) and Rachel (bookmarks b_{r_j}), are combined into a single stream, $b_{s_1}, \ldots, b_{s_m}, b_{r_1}, \ldots, b_{r_n}$. In this particular case temporal order is preserved as both bookmarks b_{s_i} and bookmarks b_{r_j} are in temporal order (from most recent to oldest) and Sam's last bookmark b_{s_m} is more recent than Rachel's first bookmark

```
(curl/remote/block                                                              1
 Bookmarks@                                                                     2
 (lambda ()                                                                     3
   (iterate/pipe                                                                4
    (iterate/append                                                             5
     ; Sam s bookmarks from May 15, 2014 thru the morning of June 1, 2014.     6
     (bookmarks/iterate ; g1                                                    7
      "sam"                                                                     8
      (cons "2014-05-15T00:00:00Z" "2014-06-01T12:00:00Z"))                     9
                                                                                10
     ; Rachel s bookmarks over July, 2014.                                      11
     (bookmarks/iterate ; g2                                                    12
      "rachel"                                                                  13
      (cons "2014-07-01T00:00:00Z" "2014-08-01T00:00:00Z")))                    14
                                                                                15
    (lambda (g)                                                                 16
      (iterate/filter g (lambda (b) (country-of-bookmark? "Scotland"))))        17
                                                                                18
    (lambda (g)                                                                 19
      (iterate/fold g (lambda (b seed) (cons b seed)) null)))))                 20
```

Figure 9.19: The combinator `iterate/append` (lines 5–14) is used as the source of bookmarks for `iterate/pipe` (lines 4–20). Two streams are appended. The first stream (lines 7–9) comprises Sam's bookmarks over the period of May 15, 2014 through the morning of June 1, 2014. The second stream (lines 12–14) comprises Rachel's bookmarks over the month of July, 2014. Lines 16–17 passes only those bookmarks (from either stream) tagged with the country `"Scotland"` and lines 19–20 accumulate those bookmarks b into a list (in reverse temporal order) that is returned to the client as the value of `curl/remote/block` (lines 1–20).

b_{r_1}.

A series of generators $g_1, g_2, \ldots, g_m$, $m > 2$ can be appended by nesting calls to `iterate/append`

$$\texttt{(iterate/append } g_1 \texttt{ (iterate/append } g_2 \cdots \texttt{ (iterate/append } g_{m-1}\ g_m\texttt{)} \cdots \texttt{))}$$

as a right-associative composition. This extension, from two generators g_1 and g_2 to $m > 2$ generators can itself be captured as a general purpose combinator, as illustrated in Figure 9.20. For example, a combinator to append $m > 2$ generators can be written as

$$\texttt{(iterate/compose/right iterate/append (}g_1\ g_2 \cdots g_m\texttt{))}$$

A convenient, equivalent shorthand, `iterate/append*`, defined in Figure 9.21, can be called as

$$\texttt{(iterate/append*}\ g_1\ g_2 \cdots g_m\texttt{)}$$

Preserving temporal order when using `iterate/append` is the responsibility of the client and there are many useful cases for which temporal order is irrelevant. Further, `iterate/append` is indifferent to the detailed composition of the individual generators g_i and each one may itself be a complex construction based, for example, on `iterate/pipe`. Figure 9.22 illustrates appending two streams distinguished both by user, Sam and Rachel, and by country tag, Spain and France. Here generators g_1 (lines 6–10) and g_2 (lines 12–15) produce any bookmark by Sam with a country tag of Spain and any bookmark by Rachel with a country tag of France, respectively. These two substreams, S_1 and S_2, are both in temporal order (from most recent to oldest) but the complete stream $S_1 + S_2$ is not necessarily in temporal order as there is no guarantee that the oldest bookmark of Sam with a country tag of Scotland is more recent than the first bookmark (the most recent) of Rachel with a country tag of France.

Interleave

Each of the three combinators proposed here, *append*, *interleave* and *merge*, are distinguished, one from the other, by their visit pattern — given a sequence of generators $g_1, \ldots, g_m$ in what order and and pattern will the combinator visit the individual generators g_i? The visit pattern of *append* preserves the sequence order $g_1, \ldots, g_m$ and exhausts each generator g_i before moving on to its successor g_{i+1}. The round-robin pattern of *interleave* visits each generator g_i in sequence order in each round but obtains at most one bookmark from each g_i over the course of a round. Exhausted generators g_j are discarded and the rounds continue until every generator g_i in the sequence is exhausted. In other words, every round of the interleave pattern generates at most m bookmarks in the order $b_{g_1}, \ldots, b_{g_m}$ for every round.

A call

`(iterate/interleave` $g_1 \cdots g_m$`)`

is a form of "fair access" for generators since each generator g_i has the opportunity to contribute a bookmark in each round (modulo the exhaustion of g_i). To illustrate, imagine that we have three generators g_1, g_2, g_3 for the bookmarks of Rachel, Sandra, and Ursula respectively. Generator g_1 can produce four bookmarks r_1, r_2, r_3, r_4, generator g_2 can produce three bookmarks s_1, s_2, s_3 and generator g_3 can produce five bookmarks u_1, u_2, u_3, u_4, u_5. The call

`(iterate/interleave` g_1 g_2 g_3`)`

will return a generator that produces the bookmark sequence $r_1, s_1, u_1, r_2, s_2, u_2, r_3, s_3, u_3, r_4, u_4, u_5$ in a

total of five rounds

(1) r_1 s_1 u_1

(2) r_2 s_2 u_2

(3) r_3 s_3 u_3

(4) r_4 u_4

(5) u_5

Figure 9.23 illustrates its implementation. The return value (line 2) is a generator (implemented as a thunk); therefore suitable as the first argument of an `iterate/pipe`. The helper function `subinterleave` (lines 4–22) implements the traversal and maintains (as arguments) two lists for each round: the *generators* yet to be visited and the live generators *visited* so far (line 4); the state of the interleave comprises these two arguments. *Motile* (like Scheme) performs tail-call optimization (see [67] p. 68), consequently the recursive calls of `subinterleave` are optimized as loops-in-place. The anonymous λ of lines 18–21 shields the call of `subinterleave` and is the functional equivalent of a coroutine. The arguments of `iterate/interleave` may themselves be any other combinators whose value is a generator, including `iterate/pipe`, and `iterate/interleave` itself may be used wherever a generator is expected.

Merge

Given two generators g_1 and g_2 whose outputs conform to a total order p (for example the predicate $\leq$ for natural numbers) nerge combines two generators g_1 and g_2 to produce a single generator g whose output contains all of the outputs of g_1 and g_2 and also conforms to the total order p. The general form of `iterate/merge` , illustrated in Figure 9.25, takes three arguments: a two-argument predicate p and two generators g_1 and g_2. The predicate defines the same total order p on the items generated if the outputs of both generators g_1 and g_2, say $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$ respectively, conform to $(p\ x_i\ x_{i+1})$ and $(p\ y_i\ y_{i+1})$ for all $i = 1, 2, \ldots$. The value returned by `iterate/merge` is a thunk, a generator g, suitable for use in any combinator where a generator is expected. Finally, `iterate/merge` can be composed with itself and other combinators to merge the streams of $m > 2$ generators $g_1, \ldots, g_m$.

Figure 9.26 defines a specific form of merge, `iterate/temporal`, that merges the output of two bookmark generators and preserves the customary temporal order of bookmarks (from most recent to oldest) provided the generators g_1 and g_2 produce bookmarks in temporal order from most recent to

oldest. Line 3 defines an anonymous function that serves as the total order p for `iterate/merge`.[17] Left or right associative compositions of `iterate/temporal` can merge $m > 2$ generators, for example, (`iterate/temporal*` $g_1 \cdots g_m$) the temporal equivalent of `iterate/append*`, is shown in Figure 9.24.

Limit

A client may wish to short-circuit a search, that is, terminate a search mid-way once some condition has been met. To meet this need I present a combinator that terminates a stream once a limit $n > 0$ has been reached, for example, halting a search after the first 50 bookmarks with given characteristics have been collected. A definition of `iterate/limit` is given in Figure 9.27 and Figure 9.28 illustrates a use case, selecting a small set of bookmarks from multiple, distinct sources that denote photographs. Here `iterate/interleave` (lines 5–8) generates candidate bookmarks from the bookmark collections of Sam, Rita, and Uri, an `iterate/filter` (line 10) passes on only those bookmarks that refer to photos, and `iterate/limit` (line12) limits the collection of bookmarks to a maximum of 50. An `iterate/fold` (lines 14–15) gathers those bookmarks into an unordered set which is returned to the client as the value of the remote evaluation (lines 1–2) of the thunk (lines 3–15) transmitted by the client to the bookmark server.

9.6 Performance

The primitives defined for the bookmark service may be sufficient but are not always efficient. Two obvious examples come to mind. First, it is often useful to know the total number of bookmarks or tags for an individual user u but, with the API as defined, the only way to obtain that total is to iterate through all of u's bookmarks (or tags) from most recent to oldest. Having the back-end database maintain those totals for each user is far more efficient. A related query is the oldest bookmark or tag for a user u. This can be implemented within the API by conducting a temporal binary search (using either `bookmark/iterate` or `tag/iterate`) with a temporal range of $[s, E)$ where E is a fixed *datestamp* that precedes the inaugural date of the bookmark service. The complexity of that search is roughly $O(\log_2)$ in the number of calendar days in the period between the date u joined the bookmark service and the service's inaugural date. In contrast, for the back-end database, memorializing the *datestamp* of the first bookmark or tag is an insignificant one-time constant cost. Functions for both these examples would be a useful addition to the API proposed here.

[17] Recall that bookmark timestamps are given in epoch time (for example, elapsed seconds since the start of the epoch). If the timestamp of bookmark b is $\geq$ the timestamp of of bookmark b' then b is at least as recent as b'.

9.7 Summary

Not only do the examples demonstrate that COAST clients can compensate for the shortcomings of provider APIs they can also guide the evolution and extension of provider APIs. In some respects the burden on providers may be reduced, as it is sufficient to offer an API that is merely "general enough" rather than an API that is complete or comprehensive. The advantages to service providers includes: reduced costs and resources to develop and field APIs, greater opportunities for API experimentation and variation, continual incremental improvement that reduces time to market and minimizes sunk costs, crowd-sourced API engineering, constant feedback on the efficacy of the API, more effective integration with other provider APIs, ongoing improvements in the vitality of the service ecosystem, greater synergy with other service providers, richer and more varied service infrastructure, more rapid API repair to eliminate bugs or shortcomings, and broad agreement within domains on common API cores while, at the same time, fostering specialized API extensions.

The COAST style encourages a peering infrastructure where the roles of provider and client are largely interchangeable. This "equitable trade" of roles may stimuate variation and specialization in services. I speculate that reducing the friction of service API design, deployment, and improvement may prompt a commensurate increase in the pace of innovation in service offerings. The examples here suggest, that at a minimum, COAST-based APIs can match the richness of REST-based APIs. Future work on replicating and extending well-known RESTful APIs may offer additional insights.

```
(define (iterate/compose/right f generators)
  (let ((m (length generators)))
    (cond
     ((= m 0) (lambda () #f))   ; Return the empty generator.
     ((= m 1) (car generators)) ; Return the only generator as is.
     ((= m 2)
      (f (car generators) (cadr generators))) ; (f g_1 g_2)
     (else
      (f
       (car generators) ; g_1
       (subcompose/right f (cdr generators) (- m 1)))))))) ; (f g_2 ... g_m)

(define (subcompose/right f generators n)
  (if (= n 2)
      (f (car generators) (cadr generators)) ; (f g_m-1 g_m)
      (f ; (f g_i (f g_i+1 ... g_m))
       (car generators) ; g_i
       (subcompose/right f (cdr generators) (- n 1))))) ; (f g_i+1 ... g_m)
```

Figure 9.20: Let f be any combinator f that accepts two generators g_1 and g_2 and combines those two generators into a single stream. The combinator (`iterate/compose/right` $f\ g_1 \cdots g_m$), given f and generators $g_1, g_2, \ldots, g_m$, $m > 2$, returns a generator g that is a right-associative composition of combinator f for generators $g_1, g_2, \ldots, g_m$, in other words, $(f\ g_1\ (f\ g_2 \cdots (f\ g_{m-1}\ g_m) \cdots))$.

```
(define (iterate/append* . generators)
  (iterate/compose/right iterate/append generators))
```

Figure 9.21: A companion combinator to `iterate/append`, (`iterate/append*` $g_1 \cdots g_m$) appends the streams of $m > 2$ generators by composing a right-associative chain of calls to `iterate/append`.

```
(curl/remote/block
 Bookmarks@
 (lambda ()
   (iterate/pipe

    (iterate/append
     (iterate/pipe ; g1 - all of Sam s bookmarks for Spain.
      (bookmarks/iterate "sam" #f) ; #f means all bookmarks.
      (lambda (g)
        (iterate/filter g (lambda (b) (country-of-bookmark? b "Spain")))))

     (iterate/pipe ; g2 - all of Rachel s bookmarks for France.
      (bookmarks/iterate "rachel" #f) ; #f means all bookmarks.
      (lambda (g)
        (iterate/filter g (lambda (b) (country-of-bookmark? b "France"))))))

    ; All of Sam s bookmarks for Spain and Rachel s bookmarks for Framce.
    (lambda (g)
      (iterate/fold g (lambda (b seed) (cons b seed)) null)))))
```

Figure 9.22: Using `iterate/pipe` to form generators for `iterate/append`.

```
(define (iterate/interleave . generators)                                  1
  (lambda () (subinterleave generators null)))                             2
                                                                           3
(define (subinterleave generators visited)                                 4
  (if (null? generators)                                                   5
      ; No more generators remaining in this round.                        6
      (if (null? visited)                                                  7
          #f ; All generators are exhausted.                               8
          (subinterleave  ; Otherwise round-robin again.                   9
           (reverse visited) null))                                        10
                                                                           11
      ; One or more generators remaining in this round.                    12
      (let* ((g (car generators))                                          13
             (pair (g))) ; (item . h) or #f. h is successor of g.          14
        (if (pair? pair)                                                   15
            (cons                                                          16
             (car pair) ; item.                                            17
             (lambda () ; Successor generator for interleave.              18
               (subinterleave                                              19
                (cdr generators)                                           20
                (cons (cdr pair) visited)))) ; (cdr pair) = h.             21
                                                                           22
            (subinterleave (cdr generators) visited))))) ; g is exhausted. 23
```

Figure 9.23: The outputs of multiple generators can be interleaved in useful ways. In each round $j = 1,2,\ldots$ of the interleave the combinator (`iterate/interleave` $g_1 \cdots g_m$) returns, in succession, bookmarks $b_{j,1},\ldots,b_{j,m}$ where bookmark $b_{j,i}$ is produced by generator g_i.

```
(define (iterate/temporal* . generators)                                   1
   (iterate/compose/right iterate/temporal generators))                    2
```

Figure 9.24: The combinator `iterate/temporal*`.

```
(define (iterate/merge p g1 g2)                                                  1
  (lambda ()                                                                     2
    (let ((pair1 (g1))                                                           3
          (pair2 (g2)))                                                          4
      (cond                                                                      5
       ((and (pair? pair1) (pair? pair2))                                        6
        (let ((x (car pair1))                                                    7
              (y (car pair2))                                                    8
              (h1 (cdr pair1))                                                   9
              (h2 (cdr pair2)))                                                  10
          (if (p x y)                                                            11
              ; x precedes y in iteration order.                                 12
              (cons x (lambda () (iterate/merge p h1 g2)))                       13
              ; y precedes x in iteration order.                                 14
              (cons y (lambda () (iterate/merge p g1 h2))))))                    15
       ((pair? pair1) (cons x h1)) ; g2 is exhausted.                            16
       ((pair? pair2) (cons y h2)) ; g1 is exhausted.                            17
       (else #f))))) ; Both g1 and g2 are exhausted.                             18
```

Figure 9.25: A combinator that interleaves two generators g_1 and g_2 in the order defined by the two-argument predicate p where the the outputs of g_1 and g_2 both conform to the order defined by p.

```
(define (iterate/temporal g1 g2)                                                 1
  (iterate/merge                                                                 2
   (lambda (b1 b2) (>= (:: b1 timestamp) (:: b2 timestamp)))                     3
   g1 g2))                                                                       4
```

Figure 9.26: A specialization of `iterate/merge` (Figure 9.25) that merges two streams of bookmarks while preserving their customary temporal ordering (from most recent to oldest).

```
(define (iterate/limit g n)                                                      1
  (lambda () (sublimit g n)))                                                    2
                                                                                 3
(define (sublimit g n)                                                           4
  (cond                                                                          5
   ((<= n 0) #f)                                                                 6
   ((g) =>                                                                       7
    (lambda (pair) ; pair = (x . h) where h is the successor to g.               8
      (cons                                                                      9
       (car pair) ; x                                                            10
       (lambda () (subiterate (cdr pair) (- n 1))))))                            11
   (else #f))) ; g is exhausted.                                                 12
```

Figure 9.27: `(iterate/limit` g n`)` is a limit-stop for generator g where $n \geq 0$ is the maximum number of items that g may produce.

```
(curl/remote/block                                                              1
 Bookmarks@                                                                     2
 (lambda ()                                                                     3
   (iterate/pipe                                                                4
    (iterate/interleave ; Interleave the bookmarks of Sam, Rita, and Uri.       5
     (bookmarks/iterate "sam" #f)                                               6
     (bookmarks/iterate "rita" #f)                                              7
     (bookmarks/iterate "uri" #f))                                              8
                                                                                9
    (lambda (g) (lambda (b) (bookmark-is? b "photo"))) ; Pass photos only.      10
                                                                                11
    (lambda (g) (iterate/limit g 50)) ; No more than 50 bookmarks total.        12
                                                                                13
    (lambda (g)                                                                 14
      (iterate/fold g (lambda (b seed) (set/cons seed b)) set/eq/null)))))      15
```

Figure 9.28: A client searches though the interleaved bookmarks of Sam, Rita and Uri to find the most recent 50 bookmarks referring to photos.

Chapter 10: Evaluation: Security

Here I evaluate the claim that COAST is *sufficiently secure* to ensure that service providers can adequately protect themselves against erroneous or malicious visiting computations. There are two distinct classes of threats: resource exhaustion and abuse of authority. Resources can be *fungible*[1] or *specific*[2] In these discussions I ignore fungible resources whose protection falls well within the purview of classical resource allocation and accounting.[3]

On the other hand, access to, and exploitation of, specific resources is a running theme of this thesis and the principal driver for the formulation of COAST, the design of the *Motile* language, and the details of the reference implementation *Motile/Island*. Given a subject x the authority of x is the upper bound of all of the actions that x may take with respect to the objects for which x holds access. Under COAST every erroneous or malicious visiting computation amounts to an insider attack (see section 3.1 for a discussion of the threat model). Maximizing isolation, minimizing capability, and restricting access are the cornerstones of reducing the risks and consequences of insider attacks.

Further, mutual distrust (or equivalently, limited trust) among principals is a defining characteristic of decentralized systems. That mistrust is magnified when once trusted partners, themselves under attack or unknowingly subverted, become agents of misfeasance. Though absolute protection against insider attack is impossible (as sufficient predicates can lie well outside the boundary of computability) limits on access and resources are essential even when all principals enjoy a high level of trust.

Capability security emphasizes risk management over guarantees as breaches are inevitable in complex systems and the best that one can hope for is to minimize or contain the damage [189, 209]. Two related concepts speak to minimizing and containing system risk: connectivity and authority. Maffeis,

[1] Memory, processor cycles, network bandwidth, mass storage, and the like.

[2] Databases, domain-centric functions such as an image processing library, sensors, actuators, or specialized computing hardware such as a graphical processing units (GPU) or a quantum computer.

[3] There is one caveat: accurate memory accounting in garbage-collected functional languages, a critical tool for bounding the memory consumption of mobile code computations. Two solutions are noted here. Wick and Flatt [262] offers per task (equivalently *thread*) memory accounting without requiring per-task object partitions by leveraging the language garbage collector while Yang and Mazières [266] exploits the features of the Safe Haskell memory allocator, garbage collector, and threads subsystem to account for per-thread references to shared structures. Racket (`www.racket-lang.org`) incorporates [262] in its formulation of *custodians* [89] but *Island* (circa 2015), which is built atop Racket, does not take advantage of it. The omission was deliberate. Life is short and I had more pressing matters to deal with.

Mitchell and Taly [159] define a system *authority-safe* if it satisfies two principles captured in the object-capability literature [170, 220]:

- *Only connectivity begets connectivity*: demands that all access to resources of any description derive from prior access. In other words, a system element x cannot gain access α out of thin air, rather access α derives from the transitive closure of the prior access of x. Communication by introduction is a particular example of this principle.
- *No authority amplification*: restricts the transfer of authority to initial (ab initio) authority, authority acquired through interaction, and fresh authority over newly created resources. There are no other sources of authority.

A system that is authority-safe preserves isolation among isolated system elements and bounds the authority that any subject x can acquire or transfer to others. The total authority held by x is the set of all actions available to x.[4] To ensure that COAST meets the minimal requirements for cooperation among mutually distrustful parties (adequate security) I prove that:

- The COAST architectural style is authority-safe, meaning that it preserves isolation and eschews the implicit propagation of authority; and
- *Motile*/*Island* is capability-safe, hence authority-safe, demonstrating that the *Motile* language and the *Island* peering infrastructure preserve the security guarantees inherent in the style.

As I show in Chapter 11, pervasive authority-safety and capability-safety are crucial for higher-order security.

Section 10.1 proves that the COAST architectural style is authority-safe. Section 10.2 proves by case-analysis and induction that *Motile*/*Island* is capability-safe, and hence, courtesy of Maffeis, Mitchell and Taly [159], authority-safe. Section 10.3 reviews the impact of the work of Maffeis, Mitchell and Taly on web security while Section 10.4 reflects on the significance of our result for *Motile*/*Island* and offers a few suggestions for future work.

[4] In the object-capability model the the conservative upper bound on the total authority of x is the transitive closure of all object references held by x.

10.1 COAST is Authority-Safe

I claim that services in the COAST architectural style are authority-safe. The COAST style states (section 3.3):

- **Services:** All services are computations whose only interactions are the asynchronous messaging of values, closures, continuations, and binding environments.
- **Execution:** 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.
- **Messaging:** Computation x can transmit a message to a computation y only if there exists $\langle t\bullet, t\triangleright\rangle$ such that x holds a CURL u denoting $t\bullet$ and y holds $t\triangleright$.
- **Interpretation:** The interpretation of a message delivered to computation y via CURL u is y- and u-dependent.

Communication by introduction among COAST services is codified in the **Services** and **Messaging** rules. The first guarantees that messaging is the only interaction between two services x and y. The second ensures that x can transmit a message to y only if x holds a CURL u denoting a trajectory of communication $t\bullet \longrightarrow t \longrightarrow t\triangleright$ (section 3.2.3) that terminates in y.[5] How then did x acquire CURL u? It was:

- Obtained ab initio (the closure comprising service x bound u in its lexical scope bindings);
- Returned to x by a function in the binding environment B of the execution site $\langle E, B\rangle$ of x (the **Execution** rule), or;
- Received by x in a message (the **Messaging** rule).

In other words, communication by introduction is an instantiation of *only connectivity begets connectivity*.

A reprise of the argument above demonstrates the absence of authority amplification, that is, the creation of authority out of thin air (of which ambient authority is one example). CURL u in the argument above is an exemplar of authority (here the capability to communicate). How can that authority be acquired? It is:

- Obtained ab initio in a lexical scope binding of a closure;
- Exercised implicitly via the execution of a function f in the binding environment B of an execution site or returned as a value of a call to f (the **Execution** rule);

[5] More formally, y holds an egress point $t\triangleright$ of that trajectory.

- Contained in a message transmitted from one computation to another (the **Services** and **Messaging** rules).

Under COAST the acquisition of CURL u is no different from the acquisition of any other distinct form of authority. In combination the three rules: **Services**, **Execution** and **Messaging**, satisfy the conditions for an authority-safe system.

10.2 Motile/Island is Capability-Safe Hence Authority-Safe

Authority safety, while a crucial underpinning for *sufficiently secure*, is just half the story. It only defines safety conditions for ensuring isolation; not a programming model or methods of enforcement. The object-capability model championed by Miller [170] defines a structure for exercising authority that can guide programming language design (the language **E** defined and developed by Miller is one example). Maffeis, Mitchell and Taly [159] formalize the object-capability model by reducing the model to a set of reachability properties among abstract objects conforming to the object-capability model and then prove that a programming language is *capability-safe* if the upper bound on the authority of a programming language object

> ... is given by the union of the set of resources contained in all objects that are reachable from a given object in the reference graph. This notion is also known as the *topology-only bound on authority*.

A language $\mathscr{L}$ is *capability-safe* if it meets four connectivity conditions[6] that are stated in terms of object references and the methods of those objects. A reference to an object is an unforgeable capability and methods are the actions that a reference holder (another object) may perform. In accordance with Miller connectivity is limited to:

Initial conditions: Object x is granted a reference ab initio as the initial conditions of a computational state.

Introduction: If object x holds references to objects y and z, x can pass to z a reference to y and z can retain that reference to y for subsequent use.

Parenthood: If object x creates object y then x acquires the only reference to the freshly created object y.

[6] Proposition 1 of [159]. These conditions, *initial*, *introduction*, *parenthood* and *endowment*, are the definitions established by Miller [170] for an object-capability language.

Endowment: Let x hold references to objects $z_1, \ldots, z_m$. If object x creates object y then y holds references only to those z_i endowed to y by x. In other words, the object references of y at the time of creation are a subset (possibly empty) of the references held by x at that point in time. In other words, x cannot endow to y references that x does not hold.

These four conditions are often also stated in terms of message passing among objects. The two perspectives, trading references via method invocation or message passing, are equivalent.

Maffeis and his colleagues [159] prove that *if a language $\mathscr{L}$ is capability-safe then it is authority-safe*. In other words, the object-capability model provides sufficient guidance for programming language design to ensure that a system $\mathscr{S}$ composed in $\mathscr{L}$ is authority-safe — within $\mathscr{S}$ *only connectivity begets connectivity* and there is *no authority amplification*.[7]

With this in mind I argue that *Motile*, a functional, single-assignment language with persistent, functional data structures, alongside *Island*, the COAST peering infrastructure, is a capability-safe language and hence authority-safe.

- The *Motile* transpiler translates *Motile* programs into closures, each of which is implemented in a continuation-passing, lexical-scope-passing, binding-environment-passing style. *Motile* is single-assignment, there is no imperative assignment at any level of the language, and all *Motile* objects, including closures, continuations, binding environments, tuples, lists, CURLs, and persistent vectors, hash maps, unordered sets and records, are immutable. All objects are initialized at creation time. There is no means by which any *Motile* program (an executing closure) can alter the contents of any *Motile* object. Consequently, it is impossible for a *Motile* program to use global assignment to lexical scope or a binding environment as a back channel for the communication of capability.
- No *Motile* closure has access to the lexical scope environment of its caller and no caller can access a callee's lexical scope environment of origin.[8]
- Every *Motile* closure executes within the context of some binding environment B. Any function not within lexical scope must be resolved with respect to the binding environment B of the caller; failure to resolve is a fatal error. The binding environments of *Motile* are themselves functional,

[7] One must not read too much into this result, for example: there is no guarantee whatsoever that every system implementation in language $\mathscr{L}$ is secure, safe, free of risk, or absent dangerous distributions of authority. Even a disciplined implementation of a system $\mathscr{S}$ in $\mathscr{L}$ may contain an inadvertent connection that leaks a dangerous authority.

[8] The latter is not strictly true if caller and callee share the same lexical scope environment however, absent imperative assignment neither caller nor callee can affect any portion of that shared lexical scope.

persistent hash maps; therefore immutable and version-preserving. A binding environment B may be amended: bindings (name/value pairs) may be added, deleted, or the value of a binding changed without changing B.[9]

- The *baseline* binding environment of *Motile* contains only pure functions and no effectual capability. The binding environment B of an island execution site oftentimes is a superset of the baseline, containing one or more *procedures* — functions capable of side effects.[10]

Motile, in both spirit and function, bears considerable resemblance to W7, a capability-safe, Scheme dialect developed by Rees for the construction of a capability-based operating system kernel [199]. Two features in particular ensured that W7 met the conditions for capability-safe: the absence of imperative assignment and stringent control of the contents of the global binding environment containing the procedures and functions of the language. *Motile*, like W7, comprises a small, pure functional core for which lambda definition, lambda combination, conditionals, and tail recursion optimization are the principal elements. Consequently, the functional core of *Motile* (like W7) contains *no ambient authority* that exists in an implicit environment where any subject can request it by name [169].

W7 eliminated ambient authority by insisting that all functional and procedural capability be contained in modules explicitly imported at compile-time. *Motile* achieves the same goal at run-time by treating binding environments as first-class objects and evaluating closures in known binding environments that exclude ambient authority. The binding environments of *Motile* computations executing on an island are island-specific and minimize, to the extent possible, the capability granted to closures.

10.2.1 Intra-Island Motile Without Communication is Capability-Safe

Intra-island *Motile* is confined to a single island I supporting multiple computations (islets) $x_1, \ldots, x_m$, each x_i executing a *Motile* thunk f_i within an execution site $\langle E_i, B_i \rangle$. Assume for the moment that no B_i

[9] To the extent possible functional, persistent structures share common substructure. A binding environment B' derived from B by a series of additions, deletions, or value changes will likely share substantial, unchanged substructure with B (roughly proportional to the difference between B and B'). On a single island binding environments are safely shared by reference and the memory costs of derived binding environments B' are modest.

[10] For example, one binding environment used for debugging and testing contains the R7RS procedure `display` that produces human-readable output on a display device of some description. Binding environments intended as targets for spawns contain `curl/send`, a procedure that transmits a message from one computation (islet) to another be it intra- or inter-island.

- *Motile* does not implement `eval` for dynamically evaluating expressions in the context of one or more predefined binding environments. Consequently *Motile* closures cannot amplify their authority by any means other than the acquisition of object references by introduction, parenthood, or endowment.

contains communication capability directly or indirectly[11], so that it is impossible for any computation f_i to message another computation f_j on I.[12]

Even though two closures f_i and f_j may share memory via lexical scope (certainly when $f_i = f_j$), and the same closure g may appear in the bindings of two or more binding environments B_i and B_j, those memory cells are immutable. *Motile* is single-assignment and the binding environments B_i lack any closure capable of imperative assignment to the heap, the lexical scope of any f_i, any binding of B_i, or the lexical scope of any closure in B_i. As a consequence each computation (islet) executing f_i is isolated from the other; there may be shared state but that state is immutable.

How then do the computations x_i acquire connectivity (object references) given these assumptions? There are only four avenues:

Initial Conditions: Since each f_i is a *Motile* closure its initial capability derives from only two sources: its lexical scope bindings or capability acquired from the binding environment B_i of its execution site. As each islet x_i is isolated one from the other and they cannot communicate, no x_i can acquire, by initial conditions, any object reference that has effectual influence on its co-resident computations x_j.

Introduction: It is impossible for any computation x_i to introduce an object reference to any other x_j as there is no communication of any form between x_i and x_j. It is also impossible to introduce a reference of any kind, from anywhere, to any immutable data structure that x_i may hold. Therefore, in this scenario connectivity by introduction is impossible.

Parenthood: x_i can create a new object, for example a list:

```
(let ((a (list 7 11 17)))                                    1
  (f (car a)))                                               2
```

and bind the reference to that object, the list (7 11 17), within its lexical scope (here the variable a).[13] Calling a constructor bound within B_i or the lexical scope of x_i is the only way that x_i can create an object.

Endowment: x_i can endow objects with:

[11] For example, `curl/send` or the ability to create or read promises.

[12] Recall that an island is a single, homogeneous address space where all islets residing on the island share the same heap. For efficiency intra-island communication among islets is by reference via shared memory structures.

[13] The primitive `list` is a pure function bound with the *Motile* baseline environment.

1. References from its lexical scope (or more formally from the lexical scope of f_i),
2. References *to* functions within its binding environment B_i, or
3. References created by calls to functions within its binding environment B_i.

For example given,

```
(tuple a cons (cons (* 3 3) 27))
```

where `tuple` is the tuple constructor (a fixed-length immutable vector): a is a binding within the lexical scope of f_i and `cons` is the constructor for pairs (cons cells), with `tuple` and `cons`. Both `tuple` and `cons` are bound within the baseline binding environment of *Motile*, hence B_i as well.

Clearly, intra-island *Motile*, absent any communication capability, is capability-safe and hence authority-safe (not to mention dull and limited).

10.2.2 Intra-Island Motile with Only Intra-Island Communication is Capability-Safe

Assume that the binding environments B_i of islets x_i contain procedural communication capability — `curl/send`, `promise/new`, and `promise/block` among others — but that procedural capability is restricted to intra-island communication (a restriction that any island can enforce).

There are two cases to consider. First, If no x_i holds a capability for communication with any x_j (a CURL u_{x_j}) then the procedural capability for communication bound within B_i is useless. No x_i has been introduced to any x_j and without an introduction, specifically a CURL u_{x_j} forx_j, no x_i can use any communication procedure to greet x_j and initiate an exchange. Therefore this case is equivalent to that of Section 10.2.1 above and in this instance the construction is capability-safe.[14]

For the second case assume that x_i holds a CURL u_{x_j} for x_j and that x_i is the only computation to hold u_{x_j}. How did x_i acquire u_{x_j}? Absent any communication acts then u_{x_j} was endowed to x_i by a chain of endowment that can be traced back to the parent of x_j. There is simply no other mechanism available in a capability-safe language and we know that, in the absence of any communication, intra-island *Motile* is capability-safe (per Section 10.2.1). From this point, if x_i holds u_{x_j} it can establish bidirectional communication with x_j by creating a CURL u_{x_i} and transmitting it via u_{x_j} to x_j. In other words, x_i introduced itself to x_j. If an x_k introduces itself to x_j then x_j can both reciprocate *and* introduce x_k to x_i

[14] An identical argument applies to promises as well since the resolver of any promise is a CURL.

since x_j holds u_{x_i}. Thus endowment plus successive introductions allows computations x_i to establish cliques $x_{i_1}, \ldots, x_{i_s}$ of inter-communicating computations in which each member x_{i_k} has bi-directional communication with all of the other members of the clique.[15]

I proceed by induction on the number n of pairs of computations $x_i \rightarrow x_j$ for which x_i holds a CURL u_{x_j} for x_j.[16] Assume $n = 1$, that is $x_i \rightarrow x_j$ is the only communication connectivity among the computations $x_1, \ldots, x_n$ residing on an island I. The core of the argument rests on the observation of Section 10.2.1 above: absent connectivity by introduction *Motile*/*Island* on a single island is capability-safe. However, all that $x_i \rightarrow x_j$ provides is one-way intra-island *connectivity by introduction* between a single pair of computations. Since all messages are immutable no message can introduce a shared memory back channel for communication and therefore the other three forms of connectivity — initial conditions, parenthood and endowment — and their restrictions remain unchanged. The only change is that introduction opens a path by which additional references to immutable objects can be propagated from one computation to another. In other words, *an island with a single communicating pair of computations is capability-safe.*

Assume that an island with $n-1$ co-resident pairs of communicating computations is capability-safe. The introduction of an additional pair of communicating computations does nothing to expand the means by which connectivity is extended, that is to say, connectivity begets connectivity only through initial conditions, introduction, parenthood, and endowment. Ergo, intra-island *Motile* with only intra-island communication is capability-safe.

10.2.3 Inter-Island Communication is Capability-Safe

Let I and J be two distinct islands executing *Motile* computations $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ respectively. Assume that:

- x_i holds a CURL u_{y_j} for y_j on island J and that x_i is the only computation residing on I holding a CURL referring to an ingress point on J.
- The binding environment B of the execution site of x_i contains `curl/send`.
- x_i may transmit messages inter-island.

[15] It is trivial to create and endow a promise and bootstrap that, using a trivial exchange protocol, into a mutual exchange of CURLS u_{x_i} and u_{x_j} between computations x_i and x_j.

[16] For bi-directional communication we have $x_i \rightarrow x_j$ and $x_j \rightarrow x_i$.

x_i can transmit any serializable *Motile* object from island I to y_j on island J including CURLs, closures, binding environments, tuples, lists, and all persistent data structures. However, many of the closures found in island binding environments are *Motile*-compliant wrappers around primitives written in languages other than *Motile*.[17] Those closures are *fixed assets*, cannot be transmitted off-island, and are safely ignored.[18]

CURLs may appear embedded within any other *Motile* object including closures, binding environments, and other CURLs. CURLs are the only object that can move between islands and confer additional authority to its recipient, in the sense that possession of a CURL can grant the holder a communication authority that it did not possess before (for example, communication with a previously unknown island or access to service that was previously unavailable but on a known island). On the other hand, It is impossible for any *Motile* closure transmitted inter-island that does not contain CURLs in its lexical-scope bindings to confer additional authority because the capability granted to a *Motile* closure depends entirely on the bindings in the binding environment of its execution site.[19]

How then did x_i come to possess CURL u_{y_j} in the first place? CURLs have two distinct representations: on-island as a Racket record (with a *Motile* API and recognized as a *Motile* object type) and a serialized representation as a human-readable set of attribute/value pairs, the latter is also suitable for out-of-band transmission via web page, email, or pastebin.[20] We can, without loss of generality, presume that x_i obtained the CURL u_{y_j} from initial conditions: the human-readable representation appeared as a string literal in a body of *Motile* source code or was read from a repository of CURLs accessible via other means, such as HTTP. The first two techniques are commonly used to bootstrap inter-island communications. The reference implementation of *Motile*/island (circa 2015) uses UDP broadcast to beacon island presence and simultaneously distribute an island-specific CURL for initial contact; another form of connectivity by initial conditions. More generally, CURLs arrive in messages from other islands, connectivity by introduction and, in the case of x_i, transferred via a chain of endowment until u_{y_j} appears in a lexical

[17] In the reference implementation of *Motile*/*Island* those languages are C and Racket.

[18] Section 6.2 discusses the subtleties of the inter-island transmission of closures. I confine the discussion here to binding environments whose closures are wholly defined in *Motile*, hence not a fixed asset.

[19] Recall that a *Motile* closure λ leaves the bindings of its free variables behind when transmitted off-island. Those free variables are rebound (on each variable reference) by lookup in the binding environment B of of an island-specific execution site $\langle E, B\rangle$ of the destination island. On a single island the same closure λ can safely execute in multiple, distinct, execution sites $\langle E_i, B_i\rangle$ simultaneously, even though the same free variable $\alpha \in \lambda$ may become bound to a different closure $\lambda_{\alpha,i}$ in each binding environment B_i.

[20] That serialized representation is translated back into its on-island record representation by the deserializer. That translation is implemented by an *Island* library devoted to the two representations.

scope binding of a closure f_i executed by some computation x_i. Computations that are born on island almost always acquire a "bootstrap" CURL via endowment and binding in the lexical scope of the thunk comprising the computation.

Irrespective of how CURLs appear on-island in the first place[21] the only form of connectivity that CURLs confer is connectivity by introduction in which objects are serialized on the transmitting island and deserialized on the receiving island. Communication by introduction is always capability-safe hence inter-island communication, which is *always* CURL-mediated, is capability-safe.

This argument is completely independent of the number of computations communicating inter-island between two islands. Induction on the total number of inter-communicating islands shows that if $n-1$ islands are capability-safe then n inter-communicating islands are capability-safe since a pairwise analysis for any $(I_i, I_j), i \neq j$ from among all islands $I_1, \ldots, I_n$ proves that all communication between I_i and I_j is just connectivity by introduction, hence capability-safe.

10.3 Related Work

We are already awash in decentralized systems: the world of web servers and browsers is the largest complex of decentralized principals in the world and the dynamic web pages generated by modern servers for consumption by browsers are stuffed to the gullet with JavaScript (not mobile code by my definition but an executable representation nonetheless).[22] Practical systems for constraining JavaScript execution in the browser include AdSafe[23] [194] and Caja[24] [171]. Maffeis, Mitchell and Taly formalize the key mechanisms underlying these sandboxes and prove that they can be used to create *secure* sandboxes. However, their security rests critically on the underlying capability-safe language, exemplified by [233] which relied upon the semantics of a restricted version of JavaScript expressly devised to isolate browser-side JavaScript, when conducting a static security and safety analysis of server-included JavaScript libraries. JSand [3] is a server-driven, but client-side executed, JavaScript sandbox that is, the executing JavaScript is confined client-side (but with no changes to either the browser or remote JavaScript libraries). Taken as

[21] Bootstrapping via initial conditions is our answer to the "chicken and egg" problem of how islands acquire CURLs to launch communication with other islands. After all, if a CURL is required to communicate how do you communicate to acquire a CURL?

[22] Circa 2011, of the top 100,000 internet web sites a total of 3,300,000 web pages contained 8,439,799 remote JavaScript inclusions [178].

[23] `www.adsafe.org`

[24] `https://developers.google.com/caja/`

a whole, this body of work suggests that capability-security and the twin models of authority-safe systems and capability-safe languages can play a role in decentralized systems that are not browser-centric.

10.4 Summary

Adequately secure may be the strongest *realistic* claim that one can make about decentralized systems. The capability-object model, though far from complete or settled [59, 61–64], offers considerable advantages, not the least of which being a concrete programming model for capability-safe and hence, authority-safe systems. It all boils down to an acceptable level of risk for the stakeholders of a decentralized system and, as my analysis suggests, COAST and its embodiment, *Motile*/*Island*, provide mechanisms that allow a security-conscious developer to modulate access and capability with a high degree of flexibility and precision. I don't claim perfection — adequacy is the best that I can hope to achieve.

The proofs also indicate that computation exchange and object-capability security are not incompatible — the style is authority-safe and the reference implementation is capability-safe hence authority-safe. While computation exchange introduces additional risk above and beyond those ordinarily found in distributed and decentralized systems it does not a priori render systems unsafe or insecure. At the same time, as demonstrated previously in Chapters 8 and 9, computation exchange offers powerful primitives for live system update and enlarges the scope and vigor of service API design, deployment, and exploitation. As shown in Chapter 11 following it also offers the prospect of innovative forms of system inspection and monitoring. I argue that embedding computation exchange in the object-capability model of security strikes a useful and practical balance between the power of computation exchange and the demands of safety and security.

In addition, the arguments for the authority-safety of COAST, the capability-safety of *Motile*/*Island*, and the congruent authority-safety of the two combined are indicative of the degree to which COAST, the style, influenced the semantics of *Motile*, the language [115]. Nonetheless *Motile*/*Island* is not the only formulation of language and infrastructure capable of satisfying the demands of COAST. In fact, Maffeis, Mitchell and Taly prove by example that while capability-safety is a sufficient condition for authority-safety it is *not* a necessary condition; opening the door to languages for COAST that are authority-safe but not capability-safe. In other words, capability-safety may be just one of many language models suitable for COAST-based systems. There are also interesting variations in capability-safe languages, for example:

Oz-E [222], which draws upon experiences with the capability-secure languages E [170] and the W7-kernel for Scheme48 [199], and Pony[25], a statically typed, strongly typed, actor-based language with deny capability [42].

Tools and logics for capability analysis may provide insight into both language design and system architecture. Spiessens [220] contributes both a logic, SCOLL, and a model checker/solver SCOLLAR, based on constraint logic programming, that solves safety problems in capability systems. The capability-safety of *Motile*/*Island* and the "actor-like" islet-based services described in Chapter 8 may sufficiently constrain the semantics of service execution and capability transfer as to permit formal verification of the isolation, transfer, and confinement of capability [231], an appealing avenue for verifying the security behavior of decentralized systems.

[25] `http://ponylang.org`

Chapter 11: Evaluation: Architectural Accountability

The proofs of Chapter 10: that COAST is an authority-safe style, *Motile* a capability-safe language, and *Motile*/*Island* a capability-safe infrastructure, hence authority-safe, offer a fresh perspective for designing and developing secure decentralized systems. This perspective emphasizes capability accounting — for which the fundamental unit of accounting is the creation of a capability, a transfer of capability from one computation to another, or the exploitation of a capability. To use the terminology of the proofs, this means tracking the parenthood, endowment, introduction, and authority of capability. In this context *capability accounting* is a trace of the life history and authority of capability. Capability accounting embodies *architectural accountability* [270], the degree to which an architectural style supports security accounting.

The target systems for capability accounting are typified by decentralized services such as those found in ecommerce, where there are multiple interacting principals, mutual distrust among those principals, and no single overarching authority. The individual principals are autonomous, yet cooperate with other principals to achieve shared goals (such as obtaining goods and shipping them to a consumer in return for payment).

My perspective, however, differs from classic request/response services in several respects, ranging from the granularity of services, to the extensive use of mobile code and capability security, to the very nature of what a service provider is and how it operates. *The core of COAST's architectural accountability is capability accounting* — the practice of producing and maintaining a record or statement of the generation, transfer, or use of capability relating to a particular period or purpose.

Even in a world of perfect security, accounting and auditing are necessary to verify the correct operation and integrity of critical systems. I address the problems of accounting from the perspective of system architecture and architectural styles [238]. Architecture, in addition to its other virtues, also has the potential to induce forms of accountability for security, hence the theme of this final portion of the evaluation: accountability through architecture. COAST is an architectural style where the explicit transfer and exercise of capability is a fundamental structural element. To what degree can an architectural style

induce architectural accountability?

Section 11.1 analyzes the COAST style and its reference implementation, *Motile*/*Island*, for probe points at which one can detect capability events. Section 11.2 presents a synopsis of an early experimental study of capability accounting under COAST. Related work is reviewed in section 11.3 and section 11.4 offers suggestions for future work in this area.

11.1 Approach: Capability is in Plain Sight

Under capability architectures security arises from the deliberate and systematic allocation of minimal capability (the Principle of Least Authority) across system components. In effect, capability security systems erect a "capability workflow" for which the allocation, transfer, and exercise of capability resembles the flow of goods through a logistics system or the steps in a manufacturing workflow.

The proofs of Chapter 10 pivoted on the the propagation of authority and capability through the style, the communication of capability within *Motile* programs, and the exchange of capability among islands. At every turn in the arguments, the exact points at which capability and authority appeared or were transferred were explicitly identified. In other words, flows of capability and authority, like flows of money or goods, appear and move through well-defined points in both the COAST style and its embodiment, *Motile*/*Island*. In short, not only do the proofs tell us *where* to look, they are also *complete* — there is simply nowhere else to look.

The COAST style therefore induces *architectural accountability*. Within the style there are only *three points of inspection*: (1) the actions of mobile code (islets) at island execution sites, (2) intra-island messaging, and (3) inter-island messaging. The explicit construction of an execution site as an $\langle E, B \rangle$ pair encourages independent probes and accounting for both the execution engine E and the binding environment B of each islet computation. At a minimum, both inter- and intra-island message traffic accounting is straightforward. Thus the architecture style dictates where fundamental, effective accounting must occur.

Since *Motile* is a capability-safe language, capability arises solely from initial conditions, introduction, parenthood, and endowment. With all *Motile* types and structures immutable and the absence of imperative assignment there are no back channels for the transmission of capability among co-resident

islets on a single island or between pairs of islets, each on a different island. From the perspective of *Motile* there are only three forms of capability:

- Functional capability embodied in the contents of the binding environment B of an execution site or acquired from co-resident computations via intra-island messaging;
- Fixed assets acquired from lexical scope bindings, the return values of functions bound to the free variables of closures via the binding semantics of *Motile* mobile code, and the contents of intra-island messages;
- Communication capability acquired from the accumulation of CURLs, themselves values appearing in lexical scope bindings, the return values of functions bound to the free variables of closures via the binding semantics of *Motile* mobile code, and the contents of messages, both intra- and inter-island.

For any computation (islet) any exercise of capability can be detected at time and point of use. In the case of inter-island exchanges all of the capability transferred in a message (and in particular CURLs) can be detected by both the transmitting and receiving islands at no additional cost as a side effect of (de)serialization. For intra-island exchanges an island may, at additional cost, detect all transfers of capability at time of transmission or, for trusted islets, detect all incoming capability at time of reception.[1]

The proofs of Maffeis, Mitchell, and Taly [159], showing the link between authority-safe systems and capability-safe languages, also enumerate the touchstones of capability accounting. In other words, the twin conditions of authority-safe and capability-safe are sufficient to induce the points of capability creation, use and migration. These proofs also explain why complementary mechanisms, all serving the same end — capability accounting — appear in multiple levels throughout the architecture. While the details may certainly vary from system to system and language to language, the close integration between COAST the architectural style and *Motile*/*Island* the implementation, reveals that a small set of fundamental mechanisms: communication by introduction, mobile code, execution sites and CURLs, can serve multiple purposes — adaptability, security, and accountability simultaneously — at many levels throughout a service system.

Outside of messaging in *Island* all *Motile* capability creation, endowment, and introduction takes place within an execution site $\langle E, B\rangle$; specifically invoking functions f in the binding environment B.

[1] See Section 6.2 for a discussion of the mechanisms and the security role of (de)serialization in this regard.

How then to monitor, identify, and collect incisive traces of function applications for the purposes of capability accounting? The baseline binding environment of *Motile* is populated almost exclusively with pure functions drawn from standard bindings of R7RS [216]. Almost all binding environments B are elaborations on the baseline environment and whose additional functions $g_1, \ldots, g_m$ are oftentimes domain- and service-specific. I conjecture that in most cases it is sufficient, for the purposes of capability accounting to monitor a select critical subset, $\mathscr{C} \subset \{g_1, \ldots, g_m\}$, and thereby reduce trace and analytic overhead.

11.2 Early Experimental Evidence

A preliminary and exploratory investigation of capability accounting is reported in the work of Giorgio [106] and Giorgio, Gorlick, Nies and Taylor [105] in which a model electronic trading system, constructed as a set of decentralized COAST services under *Motile*/*Island*, was hand instrumented to generate a trace of critical capability events. A notional sketch of the model trading system is given in Figure 11.1. The flow and exercise of communication capability is illustrated in Figure 11.2.

A variety of trading computations were executed; some seeded with faults. Traces of these trading computations were subsequently analyzed with a homebrew, linear temporal logic, model checker. In each case the trace analysis either correctly identified the trace as fault free or was able to pinpoint where the expected capability event failed to occur or where an erroneous capability event did in fact take place. The proofs of Chapter 10 guarantee that the probes captured *all* of the capability events. To the extent that the test cases covered the space of service interactions, we have complete knowledge of the transfer and exercise of communication capability; in other words, completeness guarantees that no capability event will ever go undetected.[2]

Readers interested in the details of this work are invited to consult [106].

11.3 Related Work

There is a significant body of work in computational contracts for functional languages that includes generic mechanisms for generating and placing probes and automatically producing trace streams for analysis. Disney, Flanagan, and McCarthy [60] develop a formal model of module interactions for an

[2] Whether a system can afford to collect and analyze every capability event is another question altogether. If nothing else the completeness property guarantees that for instrumented critical services no capability event will slip by undetected.

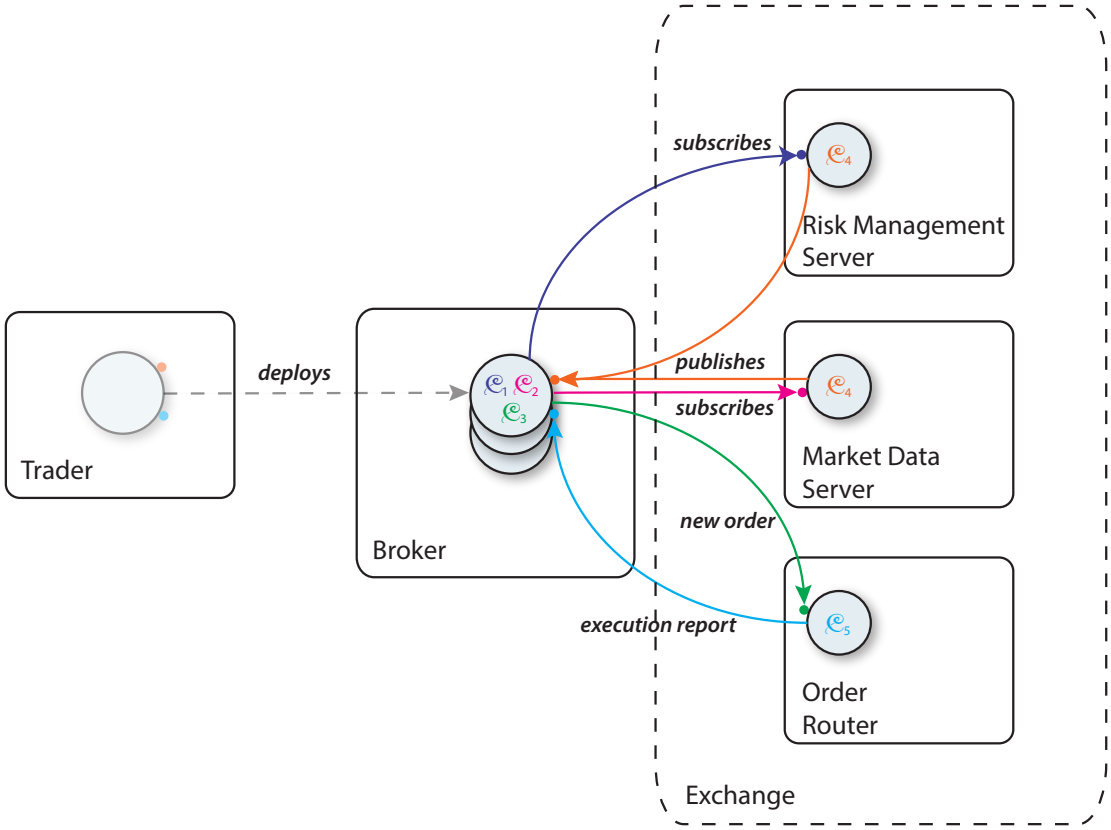

Figure 11.1: A model trading system comprising a trader, broker, and a financial exchange (adapted from [106], Figure 7.1).

imperative untyped lambda calculus with constants and mutable reference cells.[3] An operational abstract machine, the *Code* × *Store* × *Interface* machine, formalizes the semantics of modules. This machine model, in turn, generates traces of the higher-order interactions among functions defined in multiple modules where functions in one module invoke (perhaps recursively) functions in another. These traces capture the temporal behavior (in the sense of the ordering of calls and returns) of higher-order functions and allow the authors to formulate trace predicates as behavioral contracts — exactly what security-centric function-level monitoring requires. To demonstrate the work the authors define and implement a declarative language for HOT (Higher-Order Temporal) behavioral contracts in Racket[4], a well-known Scheme dialect (and incidentally the implementation language of *Motile*/*Island*).

The *computational contracts* of Scholliers, Tanter and De Meuter [211] generalize HOT contracts,

[3] This language, despite its simplicity, is general enough to model many modern languages including encapsulated behavior (via lambda expressions and lexical scope bindings) and shared imperative state. In particular, it is more than adequate for many Scheme dialects including *Motile*.

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

which treat functions as black boxes, to contracts for grey-box verification that include requirements such as mandatory function calls or protocol contracts. Applications include *prohibit* contracts (f may never call g), *promise contracts*, the dual of prohibition (f must call g), *usage* protocols (such as every call of f_0 must later be matched by a call to f_1), and *stateful* protocols whose evaluation guarantees that time-varying state constraints are satisfied, even midway in the execution of a function (for example, bounding memory consumption over the span from the call of f to its return).

11.4 Summary

The concept of capability accounting is novel and the results obtained here suggest that it can be a useful abstraction of the behavior of decentralized services. Capability accounting arises naturally in the COAST architectural style and its reference implementation, *Motile*/*Island*. Preliminary experiments suggest that capability accounting may be useful for debugging, behavioral analysis, and the detection of security faults or breaches. Automated tools for analyzing *Motile* live computations and inserting capability probes is a promising line of investigation.

Contracts for tracing and analyzing the parenthood, endowment, and introduction of capability is an open question, but combining a hazard analysis with contract-decorated functions in binding environments is a suggestive avenue of investigation. Factoring services $s_1, \ldots, s_m$ into multiple distinct binding environments $\langle E_i, B_i \rangle$ simplifies the hazard analysis for each $s_i/\langle E_i, B_i \rangle$ pair by reducing the number of critical control points that must be considered. This in turn increases confidence in the integrity of the analysis and the adequacy of the critical limits, monitoring, and corrective actions at each critical control point. In other words, the incisiveness of the hazard analysis may lead to significant security benefits.

One of the fundamental contributions of contracts to higher-order languages (such as Scheme, and hence *Motile*) is the assignment of blame; that is, identifying exactly which module is at fault when a contract is violated [87, 88]. Two other works cited earlier, Disney et al [60] and Scholliers et al [211], also go to great lengths to ensure that blame is properly apportioned. One wonders if capability accounting can be combined with higher-order computational contracts to at least narrow the scope of blame for the misuse of capability. In a similar vein, the flow of capability through a program in a capability-safe language is both well-defined and confined. Tracking and calculating approximate capability flow may prove an important analytic technique in anticipating or identifying a security breach.

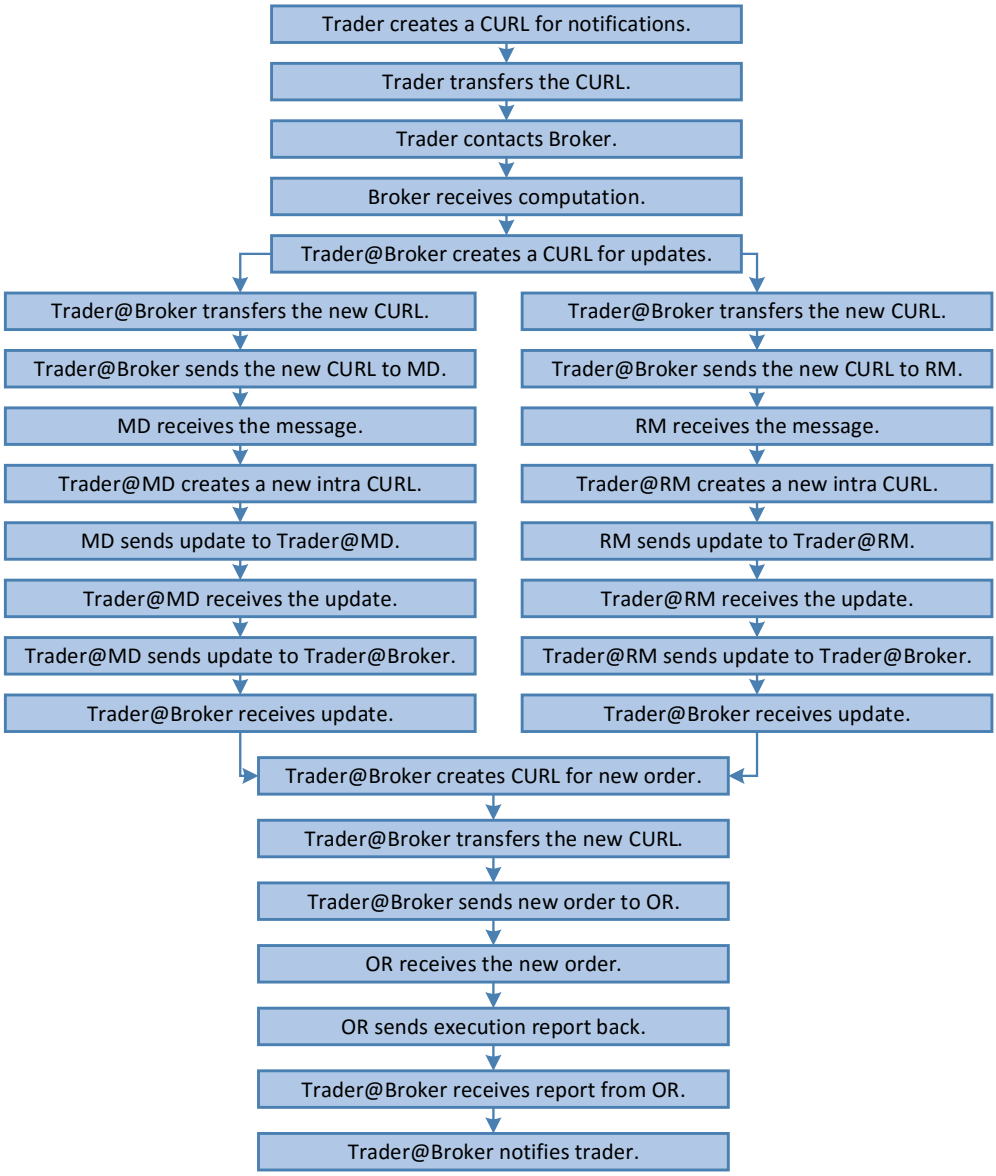

Figure 11.2: A verification model for financial trading that includes each transfer and exercise of communication capability (taken from [106], Figure 7.2). The abbreviations MD (Market Data server), OR (Order Router), and RM (Risk Management server) refer to the services shown in Figure 11.1.

Chapter 12: Summary

A decentralized system is a distributed system that operates under multiple, distinct spheres of authority in which collaboration among the principals is characterized by mutual distrust. Now commonplace, decentralized systems appear in a number of disparate domains: commerce, logistics, medicine, software development, manufacturing, and financial trading, to name but a few. These systems of systems face two overlapping demands: (1) security and safety to protect against errors, omissions and threats; and (2) ease of adaptation in response to attack, faults, regulatory requirements, or market demands. The tension between security and adaptability appears irremediable given that traditional mechanisms for adaptation, such as late bindings and malleable connectors, conflict with defenses that assume static structures and strive to limit variation.

To resolve the conflict between adaptability and security I turned to the idiom of *computation exchange*, a model of services in which peers interact by exchanging live computations for evaluation. A live computation is *state plus code*, a network continuation of a computation in progress. Live computations allow a service consumer to construct exactly the service that it wants from the primitives offered by a service provider. Computation exchange pushes innovation out to the edges of the network, simplifies the service APIs of service providers, accelerates adaptation, and lowers the barrier to service deployment — any computation exchange peer can be both a service provider and a service consumer simultaneously. However, while computation exchange might earn high marks for adaptability it is, at first glance, a security nightmare. How then to balance adaptation against security?

I consider decentralized systems and their security in the context of architectural style, a domain-specific set of rules or constraints that confer benefits to the system in question. COmputAtional State Transfer (COAST) is an architectural style for decentralized systems where the exchange of live computations is the principal interaction among peers.

To ensure a level of security consistent with the risk posture of a service provider COAST embeds computation exchange in the object-capability model of security. COAST exchanges rely on communication by introduction, meaning that a peer x can communicate with a peer y only if peer x holds a Capability

URL (CURL) for *y*; the only means of communication among computations are asynchronous immutable messages.

CURLs are cryptographic structures; they are tamper-proof and cannot be guessed or forged. Live computations received by peers via CURLs are evaluated in the context of execution sites, flexible sandboxes that confine the functional and communication capability of visiting computations. The object-capability model guarantees the integrity of communication by introduction and execution site confinement. There is simply no back channel by which a live computation can obtain a communication or functional capability surreptitiously.

These four fundamentals — communication by introduction, live computations, execution sites and CURLs — are sufficient to protect against many common security threats, including unwanted intrusion, resource theft, or gross abuse of capability. Surprisingly, these same four concepts also account for a considerable degree of adaptation and flexibility. In fact, COAST-based systems are self-similar and fractal. Services, initiated as live computations executing within the confines of an execution site, can themselves initiate services whose rules of engagement, courtesy of the COAST architectural style, are as equally confined by the style as any other service. Self-similarity encourages security at *all* system levels irrespective of level or origin of service. Likewise, it preserves adaptation, ensuring that adaptability and security are both available at all points, irrespective of the location of a service within a constellation of collaborating, but mutually mistrustful, services.

To validate the twin claims of security and adaptation for COAST I constructed a COAST-compliant reference implementation comprising *Motile*, a language for mobile code exchange and execution, and *Island*, a peering infrastructure for decentralized systems. *Motile* is a single-assignment dialect of Scheme in which all values are immutable and all data structures (such as hash maps, vectors, sets, records, and the like) are functional and persistent. Live computations are reified as *Motile* closures, continuations, and binding environments. I performed four studies with the *Motile*/*Island* platform: two each directed at adaptation and security.

The first adaptation study analyzed the problem of live update of a simple service, a gold standard for adaptive systems. Modeling an individual service as an endless service loop reading and responding to service requests I constructed *Motile*/*Island* protocols for three forms of server update: live update in place, live update with hot backup in a single address space, and live remote update with hot backup in

which the update is instantiated in a remote address space.

COAST is one of the rare architectural styles in which live update for the style conforms to the style rules itself; that is, live update is a COAST service that obeys the same rules for security and adaptation as does any other COAST-based service. Further, the mechanisms of live update are either themselves the mechanisms of *Motile*/*Island* or constructions derived from those mechanisms. Consequently, live update is no less secure and no less adaptive than any other COAST service and can be implemented entirely in *Motile*/*Island* — a notable example of fractal COAST computations. This evaluation demonstrates that:

- COAST is capable of fail-safe live update for individual services in a decentralized system;
- Live update does not demand additional mechanisms that lie outside of the COAST style; and
- Live update is safe and secure to the extent that COAST is safe and secure.

The second adaptation study examined the problem of evolving web service APIs, in particular client-driven API evolution. Client-driven extension is a direct consequence of computation exchange. Using a web bookmark service as a test case, I define a minimalist service API that is extended per-client by client-developed live computations deployed provider-side. The examples rely heavily on higher-order combinators that allow clients to concisely express complex searches and data manipulations.

The extensive client code examples (provided in Chapter 9) illustrate that service API extension and adaptation can be shifted from a service provider to the service clients, thereby easing the burdens on a service provider and speeding the pace of service evolution. In addition, the examples illustrate how clients themselves can form a service ecology around one or more core services that offers functionality well above and beyond that of the core. Again, that ecology is self-similiar and, given the COAST style rules and the properties of object-capability security, the ecology —] or any services derived from it in turn — cannot threaten the safety and security of the core services. I therefore conclude that COAST is well-suited for client-driven service extension and customization while maintaining a high degree of safety and security for providers.

The first security study offers a proof, by case analysis and induction, that COAST is authority-safe, *Motile* is a capability-safe language, and that *Motile*/*Island* is capability-safe, and hence authority-safe. The proof, by construction, identifies all of the points within a COAST-compliant system where capability is created, transferred, or exercised. The proof demonstrates that computation exchange and object-

capability security are compatible. Computation exchange confers a significant degree of adaptation while object-capability confines and modulates computation exchange in manners consistent with safety and security. Moreover, the proof is also complete, in that it identifies *all* such points within a system. The object-capability model allows COAST systems to inspect, regulate, and protect those points to the extent that the security policy of a sphere of authority is computable (and the performance overhead is tolerable). Since security and safety is fractal, the threat posed by a weak or even malicious island may be adequately contained in many cases, and where not, it may be possible to minimize or mitigate damage. Unlike many other architectural styles, security is baked in and identical mechanisms are available at all system levels. This result suggests that mistrustful collaboration is a viable strategy for decentralized systems and that the risks of colllaboration and resource sharing can be adequately contained. Finally, these results place COAST and *Motile*/*Island* squarely within the growing body of work on the security and safety of capability-safe languages and confirms that the idioms and patterns of object-capability security are available to COAST.

In the second security study I evaluated COAST with respect to architectural accountability and obtained outcomes that indicate COAST is well-suited for capability accounting, a system accounting practice in which capability is the basic unit of exchange within and among systems. Using a model financial exchange implemented in *Motile*/*Island* and a family of live computations seeded with faults, Matias and Taylor [106] demonstrated that probes placed at points where capability is created, exchanged, or exercised can generate traces that allow a linear temporal logic model checker to validate system behavior and accurately detect process and security faults.

Capability accounting is a novel abstraction of system behavior. The self-similar structure of COAST-based systems ensures that it can be applied at all system levels. It may prove useful for debugging, characterizing normative behavior, early detection of anomalous behavior including trip wires for incipient attacks, performance analysis, and establishing system-wide trust levels among multiple spheres of authority of a COAST system.

Finally, for those that wish to experiment with COAST a web site[1] contains documentation, examples, prepackaged downloads of the reference implementations of *Motile* and *Island*, installation instructions, and a link to the GitHub repository for same.[2]

[1] `http://isr.uci.edu/projects/coast/`

[2] The implementations are open-source under the Apache 2.0 License.

Chapter 13: Future Work

COAST opens many interesting avenues of investigation including:

- Types for live computations;
- Extending execution engines;
- Service discovery as a COAST service;
- Pervasive live update;
- Fractalware;
- Integration with legacy systems; and last, but not least,
- Simplifying *Motile*/*Island*

I touch on each these topics briefly here.

13.1 Types for Live Computations

Motile, like Scheme, is strongly typed but dynamically typed. This improves flexibility and ease of experimentation but may complicate debugging, reduce confidence in the correctness of live computations, or needlessly jeopardize security. Several languages offer type systems of considerable generality and expressive power. Two examples stand out. The first is Alice ML, a statically typed mobile code language base on ML [203] that also offers an interesting generalization of serialization [204]. The second is Shem, an S-expression based dialect of Haskell [225], that is closely integrated with a browser-based IDE.

Gradual typing is a compromise between complete static typing and dynamic typing. However, a recent paper by Asumu Takikawa et al [232] demonstrates that gradual typing can inflict a severe, if not crippling, performance penalty. In any case, typing live computations is a challenging problem worthy of further investigation.

There is also an extensive body of work on session types [127, 128], type systems for the protocol exchanges of concurrent and distributed systems. A recent unpublished paper by Lindley and Morris [154] charts a path forward for integrating expressive session types into a pure functional language like

Motile. Session types might simplify the design and development of the exchange protocols that appear repeatedly in COAST-based systems.

13.2 Extending Execution Engines

Extending the execution engines of execution sites for the sake of debugging, analysis, security guarantees, or multi-lingual integration offers considerable promise. For example, engines that implement trace generation for capability accounting (as a non-functional side effect of execution) may bring capability accounting to the fore as a common and useful technique. Alternatively, an execution engine could enforce temporal higher-order contracts [60] for ensuring the safe execution of the functions of execution-site binding environments.

A path to multi-lingual integration is given by Feeley [77] where he describes a Scheme compiler that targets JavaScript, Python, and Java among other languages. The *Motile* Assembly Graph, an intermediate language used in *Motile*/*Island* as the network representation for live computations, was intended for recompilation into target languages besides Scheme, but no other recompiler besides the one for Scheme has been implemented to date.

13.3 Service Discovery as a COAST Service

Service discovery, here the automated discovery of an island's services and the characteristics of its outward facing execution sites, is a topic that the thesis ignores. One can leverage remote evaluation — a client can explore an island's offerings by dispatching live computations to an island-specific execution site tailored for service discovery. However, the appropriate primitives for service discovery (that is, the bindings available in a discovery-centric execution site) and the representations for service descriptions and specifications are an open question.

More generally, islands could offer execution sites suitable for reflection on island behavior, health and status, performance loads, and traffic statistics. Higher-level services, themselves COAST computatons, that use such sites as data sources may assist in monitoring system performance, measuring progress, or detecting abnormal behavior indicative of a system fault, security breach, or attack.

13.4 Pervasive Live Update

Live update, as discussed in Chapter 8, is a powerful tool for system adaptation. Much could be done by the *Motile* compiler and at the island level to ease state capture, promote continuation transfer, and improve the reliability and performance of live update. Some of the burden might be shouldered by *Motile* frameworks or libraries for implementing live update-compliant services. As a matter of technical interest, it should be possible to design and deploy a "minimal" island that relies exclusively on live computations dispatched to it from other islands and live update for the instantiation of all core island services. This would be a powerful demonstration of the reach and power of COAST-based infrastructure.

13.5 Fractalware

Fractalware, the emergent, self-similar services that arise at many levels of COAST infrastructure and systems, are a notable system structure and hint that the design patterns for COAST compositions of COAST services may differ in interesting ways from the design patterns seen in other distributed systems. I also suspect that fractalware presents beneficial security characteristics, above and beyond the recursive extension of COAST under the flag of COAST. Fractalware extends upward as well as down. The are many higher level services, service discovery being but one example, that themselves can be constructed as COAST services. One can imagine deployment, monitoring, continuous testing, update, and migration all available as COAST-compliant services implemented in COAST. Fractalware may present new forms of fine-grain, multiparty collaboration based upon highly dynamic service sharing, service duplication, service modification, and service migration as the basic elements of collaborative constructions.

13.6 Integration with Existing Legacy Systems

Over the course of the research questions have been raised regarding a path forward to integrate COAST-based infrastructure with existing services. One attractive option is the implementation and deployment of specialized islands bridge between HTTPS on one side and inter-island protocols on the other. There is no significant technical impediment to this approach; however, it is unimplemented and awaits a security analysis of the risks of integrating an outward-facing, island-based HTTPS server with internal island services. On the other hand, its attraction is undeniable and would allow non-COAST systems and applications to both dispatch computations (perhaps in a restricted form) for execution in

a COAST infrastructure *and* receive the results using a well-known and widely implemented protocol — HTTP 1.1.

13.7 Simplifying Motile/Island

Finally, those who have developed applications and test systems in *Motile*/*Island* refer to it as "an assembly language for decentralized systems" thereby simultaneously praising its scope and damning its complexity and detail. This is an important area for further work. Here again, libraries and frameworks can go a long way to hiding tedious, error-prone detail: particularly important where security is at stake.

There is also considerable room for improvement in *Motile*. Language features for deploying, coordinating, and orchestrating decentralized live computations are ongoing topics of investigation and design patterns for web microservices are being accumulated by practitioners. No doubt the lessons learned there can be applied to COAST. It is certainly the case that the *Island* primitives for CURL generation and reflection can be improved. Rarely is the first generation of an API the right API and there is still much to learn about island- and service-level interfaces.

References

[1] *30 Years of RFCs* RFC 2555 The Internet Society, Apr. 1999 (see p. 36)

[2] Adobe Systems Incorporated. *PostScript Language Reference Manual*. (First edition for PostScript Level I) Boston, Massachusetts, USA: Addison-Wesley, 1985 (see p. 36)

[3] Pieter Agten, Steven Van Acker, Yoran Brondsema, et al. "JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications" in: *Proceedings of the 28th Annual Computer Security Applications Conference* ACSAC 2012 New York City, New York, USA: ACM, 2012, pp. 1–10 (see p. 233)

[4] Nouh Alhindawi, Jamal Alsakran, Ali Rodan, and Hossam Faris. "A Survey of Concepts Location Enhancement for Program Comprehension and Maintenance" in: *Journal of Software Engineering and Applications*, **7**:5 (2014), pp. 413–421 (see p. 197)

[5] Vincent St-Amour and Marc Feeley. "PICOBIT: A Compact Scheme System for Microcontrollers" in: *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages* Berlin/Heidelberg, Germany: Springer-Verlag, 2010, pp. 1–17 (see p. 90)

[6] Jason Ansel, Petr Marchenko, et al. "Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code" in: *Proceedings of the 2011 Conference on Programming Language Design and Implementation* PLDI'11 ACM, Jun. 2011, pp. 355–366 (see p. 16)

[7] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Jul. 2007 (see pp. 17, 44, 60, 102)

[8] Phil Bagwell. *Ideal Hash Trees*. PhD thesis Lausanne, Switzerland: Ecole Polytechnique Federale de Lausanne, 2001 (see p. 149)

[9] Jack Beaton, Sae Young Jeong, Yingyu Xie, Jeffrey Stylos, and Brad A. Myers. "Usability Challenges for Enterprise Service-Oriented Architecture APIs" in: *Proceedings of the Symposium on Visual Languages and Human-Centric Computing* VLHCC'08 IEEE Computer Society, 2008, pp. 193–196 (see p. 192)

[10] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. "Capriccio: Scalable Threads for Internet Services" in: *Proceedings 19th ACM Symposium on Operating Systems Principles* SOSP'03 New York City, New York, USA: ACM, Oct. 2003, pp. 268–281 (see p. 60)

[11] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "Developing Multi-Agent Systems with a FIPA-Compliant Agent Framework" in: *Software - Practice and Experience*, **31**:2 (Feb. 2001), pp. 103–128 (see p. 47)

[12] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "JADE: A FIPA-Compliant Agent Framework" in: *Proceedings of the Fourth Conference on the Practical Application of Intelligent Agents and Multiagent Technology* Apr. 1999, pp. 97–108 (see p. 47)

[13] John K. Bennett. "Experience with Distributed Smalltalk" in: *Software - Practice and Experience*, **20**:2 (1990), pp. 157–180 (see p. 40)

[14] John K. Bennett. *The Design and Implementation of Distributed Smalltalk* Technical Report 87-04-02 Department of Computer Science, University of Washington, Seattle, Washington, USA, Apr. 1987 (see p. 40)

[15] Daniel J. Bernstein et al. “High-Speed High-Security Signatures” in: *Journal of Cryptographic Engineering*, **2**:2 (Sep. 2012), pp. 77–89 (see p. 106)

[16] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *The Security Impact of a New Cryptographic Library* `http://cr.yp.to/highspeed/coolnacl-20111201.pdf` Dec. 2011 (see pp. 18, 73, 176)

[17] Gavin Bierman et al. “Dynamic Rebinding for Marshalling and Update, with Destruct-time λ” in: *International Conference on Functional Programming* ICFP’03 Aug. 2003, pp. 99–110 (see pp. 103, 104, 125)

[18] Andrew D. Birrell and Bruce Jay Nelson. “Implementing Remote Procedure Calls” in: *ACM Transactions on Computer Systems*, **2**:1 (Feb. 1984), pp. 39–59 (see pp. 31, 37)

[19] Matt Bishop. *Computer Security: Art and Science*. First Addison-Wesley Professional, Dec. 2002 (see p. 17)

[20] Matt Bishop, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. “We Have Meet the Enemy and He Is Us” in: *Proceedings of the 2008 Workshop on New Security Paradigms* NSPW’08 New York City, New York, USA: ACM, Sep. 2008, pp. 1–12 (see pp. 51, 52)

[21] Matt Bishop, Sophie Engle, Deborah A. Frincke, et al. “A Risk Management Approach to the Insider Threat’” in: *Insider Threats in Cyber Security* ed. by Christian W. Probst, Jeffrey Hunker, Dieter Gollmann, and Matt Bishop *Advances in Information Security*, Springer USA, 2010, pp. 115–137 (see p. 52)

[22] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. “The Development of the Emerald Programming Language” in: *History of Programming Languages* HOPL III ACM SIGPLAN, 2007, **11**:1–**11**:51 (see pp. 39, 101)

[23] J. K. Boggs. *IBM Remote Job Entry Facility: Generalized Subsystem Remote Job Entry Facility* Technical Disclosure Bulletin 752 Armonk, New York, USA: International Business Machines, Aug. 1973 (see p. 35)

[24] Allen C. Bomberger et al. “The KeyKOS Nanokernel Architecture” in: *Workshop on Micro-kernels and Other Kernel Architectures* USENIX Association Apr. 1992, pp. 95–112 (see p. 18)

[25] Peter Braun and Wilhelm R. Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann, Jan. 2005 (see p. 95)

[26] Jon Brodin. *Netflix finishes its massive migration to the Amazon cloud* `http://arstechnica.com/information-technology/2016/02/netflix-finishes-its-massive-migration-to-the-amazon-cloud/` Feb. 2016 (see p. 178)

[27] Rod Burstall and Butler Lampson. *A Kernel Language for Modules and Abstract Data Types* Research Report Palo Alto, California, USA: Digital Equipment Corporation, Systems Research Center, Sep. 1984 (see p. 82)

[28] R. Cailliau. *The writing of NODAL programs* CERN-SPS-Tech-Note 77-011-ACC Geneva, Switzerland: CERN, 1977 (see p. 36)

[29] George Candea and Armando Fox. “Crash-Only Software” in: *Proceedings of the 9th Workshop on Hot Topics in Operating Systems* HotOS’03 May 2003 (see pp. 17, 178)

[30] Luca Cardelli. “A Language with Distributed Scope” in: *Computing Systems*, **8**: (1995), pp. 286–297 (see pp. 40, 104, 125)

[31] Luca Cardelli. *Obliq: A Language with Distributed Scope* DEC/SRC-RR 122 Palo Alto, California: Digital Equipment Corporation, Systems Research Center, Jun. 1994, p. 64 (see p. 40)

[32] Luca Cardelli. *Obliq: A Language with Distributed Scope* http://lucacardelli.name/Talks/1995-05-31 Obliq A Language with Distributed Scope.pdf May 1995 (see p. 40)

[33] Adam Carlson. *The Unifying Policy Hierarchical Model.* MA thesis University of California at Davis: Department of Computer Science, Jun. 2006 (see pp. 50, 51)

[34] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. "Higher-Order Distributed Objects" in: *ACM Transactions on Programming Languages and Systems*, **17**:5 (1995), pp. 704–739 (see p. 41)

[35] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. *Transmission of higher-order objects across a network of heterogeneous machines* United States Patent 5745703 Apr. 1998 (see p. 41)

[36] Melissa Chase and Sherman S. M. Chow. "Improving Privacy and Security in Multi-authority Attribute-based Encryption" in: *Proceedings of the 16th ACM Conference on Computer and Communications Security* CCS'09 ACM, Nov. 2009, pp. 121–130 (see p. 76)

[37] Larry Chen. "AgentOS: The Agent-based Distributed Operating System for Mobile Networks" in: *ACM Crossroads*, **5**:2 (Dec. 1998), pp. 12–14 (see p. 47)

[38] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. "TinyGALS: A Programming Model for Event-Driven Embedded Systems" in: *Symposium on Applied Computing* SAC'03 ACM, 2003, pp. 698–704 (see p. 101)

[39] Elaine Cheong and Jie Liu. "galsC: A Language for Event-Driven Embedded Systems" in: *Proceedings of Design, Automation and Test in Europe* DATE'05 IEEE Computer Society, Mar. 2005, pp. 1050–1055 (see p. 101)

[40] Maria Christakis and Konstantinos Sagonas. "Static Detection of Race Conditions in Erlang" in: *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages* PADL'10 Berlin/Heidelberg, Germany: Springer-Verlag, 2010, pp. 119–133 (see p. 132)

[41] Koen Claessen, Michal Palka, Nicholas Smallbone, et al. "Finding Race Conditions in Erlang with QuickCheck and PULSE" in: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* ICFP '09 New York City, New York, USA: ACM, 2009, pp. 149–160 (see p. 132)

[42] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. "Deny Capabilities for Safe, Fast Actors" in: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* AGERE! '15 New York City, New York, USA: ACM, 2015, pp. 1–12 (see p. 235)

[43] Tyler Close. *ACL's don't* Technical Report HPL-2009-20 HP Laboratories, Feb. 2009 (see p. 73)

[44] Sylvain Conchon and Fabrice Le Fessant. "Jocaml: Mobile Agents for Objective-Caml" in: *Proceedings Third International Symposium on Mobile Agents* Oct. 1999, pp. 22–29 (see p. 45)

[45] William R. Cook, Eli Tilevich, Ali Ibrahim, and Ben Wiedermann. "Language Design for Distributed Objects" in: *Proceedings of the 1st International Workshop on Distributed Objects for the 21st Century* DO21'09 Jul. 2009 (see p. 35)

[46] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. "Links: Web Programming Without Tiers" in: *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* FMCO'06 Springer-Verlag, 2007, pp. 266–296 (see p. 101)

[47] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. "A Systematic Survey of Program Comprehension through Dynamic Analysis" in: *IEEE Transactions on Software Engineering*, **35**:5 (Apr. 2009), pp. 684–702 (see p. 198)

[48] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. "Wayback: A User-level Versioning File System for Linux" in: *Proceedings of the USENIX Annual Technical Conference* Berkeley, California, USA: USENIX Association, 2004 (see p. 16)

[49] Jason Crampton and Michael Huth. "Towards an Access-Control Framework for Countering Insider Threats" in: *Insider Threats in Cyber Security* ed. by Christian W. Probst, Jeffrey Hunker, Dieter Gollmann, and Matt Bishop *Advances in Information Security*, Springer USA, 2010, pp. 173–195 (see pp. 53, 71)

[50] G. Cugola, C. Ghezzi, G.P. 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 Jul. 1996, pp. 309–318 (see pp. 47–49)

[51] CurveCP. *CurveCP: Usable security for the Internet* `http://curvecp.org` Jun. 2011 (see pp. 69, 73)

[52] *CurveZMQ Authentication and Encryption Protocol* `http://rfc.zeromq.org/spec:26` 2013 (see p. 176)

[53] Eric M. Dashofy. *Supporting Stakeholder-driven, Multi-view Software Architecture Modeling*. PhD thesis Irvine, California, USA: University of California, Irvine, 2007 (see p. 18)

[54] N. Davies, G. S. Blair, and J. A. Mariani. *Supporting persistent relocatable objects in the ANSA architecture* tech. rep. MPG-92-04 Bailrigg, Lancaster, UK: Computing Department, Lancaster University, Jan. 1992 (see p. 41)

[55] Linda G. DeMichiel and Richard P. Gabriel. "The Common Lisp Object System: An Overview" in: *European conference on object-oriented programming on ECOOP '87* London, UK: Springer-Verlag, 1987, pp. 151–170 (see p. 39)

[56] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. "Ensuring Interoperable Service-Oriented Systems through Engineered Self-Healing" in: *Proceedings 7th Joint Meeting of the European Software Engineering Conference/ACM SIGSOFT Symposium on Foundations of Software Engineering* ESEC/FSE'09 2009, pp. 253–262 (see pp. 192, 193)

[57] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. "Test-and-Adapt: An Approach for Improving Service Interchangeability" in: *ACM Transactions on Software Engineering Methodology*, **22**:4 (Oct. 2013), 28:1–28:43 (see p. 193)

[58] Jack B. Dennis and Earl C. van Horn. "Programming Semantics for Multiprogrammed Computations" in: *Communications of the ACM*, **9**:3 (Mar. 1966), pp. 143–155 (see p. 17)

[59] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. "Declarative Policies for Capability Control" in: *Proceedings of the 27th IEEE Computer Security Foundations Symposium* CSF'14 Piscataway, New Jersey, USA: IEEE Press, Jun. 2014 (see p. 234)

[60] Tim Disney, Cormac Flanagan, and Jay McCarthy. "Temporal Higher-Order Contracts" in: *Proceedings of the 16th International Conference on Functional Programming* 2011, pp. 176–188 (see pp. 239, 241, 248)

[61] Sophia Drossopoulou and James Noble. "How to Break the Bank: Semantics of Capability Policies" in: *Proceedings 11th Conference on Integrated Formal Methods* vol. 8739 Lecture Notes in Computer Science Sep. 2014, pp. 18–35 (see p. 234)

[62] Sophia Drossopoulou and James Noble. "The Need for Capability Policies" in: *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs* FTJP 2013 New York City, New York, USA: ACM, 2013, 6:1–6:7 (see p. 234)

[63] Sophia Drossopoulou and James Noble. “Towards Capability Policy Specification and Verification” http://www.doc.ic.ac.uk/~scd/CapabilityPolicies_V2.pdf Apr. 2014 (see p. 234)

[64] Sophia Drossopoulou, James Noble, and Mark S. Miller. “Swapsies on the Internet: First Steps towards Reasoning about Risk and Trust in an Open World” http://ecs.victoria.ac.nz/foswiki/pub/Groups/Elvis/ProgrammingLanguages/RiskAndTrust.pdf 2015 (see p. 234)

[65] André Du Bois, Phil Trindler, and Hans-Wolfgang Loidl. “mHaskell: Mobile Computation in a Purely Functional Language” in: *Journal of Universal Computer Science*, **11**:7 (2005), pp. 1234–1254 (see p. 45)

[66] Danny Dubé and Marc Feeley. “BIT: A Very Compact Scheme System for Microcontrollers” in: *Higher-Order and Symbolic Computation*, **18**: (2005), pp. 271–298 (see p. 90)

[67] R. Kent Dybvig. *The Scheme Programming Language*. 4th ed. MIT Press, 2009 (see pp. 31, 216)

[68] Michel Elie. *Network Interchange Language* RFC 51 The Internet Society, May 1970 (see p. 36)

[69] Justin R. Erenkrantz. *Computational REST: A New Model for Decentralized, Internet-Scale Applications*. PhD thesis Irvine, California, USA: University of California, Irvine, Sep. 2009 (see pp. 31, 32, 62, 167)

[70] Justin R. Erenkrantz, Michael M. Gorlick, Girish Suryanarayana, and Richard N. Taylor. “From Representations to Computations: The Evolution of Web Architectures” in: *ACM SIGSOFT Symposium on The Foundations of Software Engineering* FSE’07 Sep. 2007, pp. 255–264 (see pp. 14, 26, 28, 31, 61, 92, 167)

[71] Justin R. Erenkrantz, Michael M. Gorlick, and Richard 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, USA, May 2007 (see pp. 31, 62)

[72] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice-Hall, 2005 (see pp. 31, 62)

[73] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis Ithaca, New York, USA: Cornell University, Jan. 2004 (see p. 66)

[74] Michael D. Ernst. “Static and Dynamic Analysis: Synergy and Duality” in: *WODA 2003: ICSE Workshop on Dynamic Analysis* WODA’03 May 2003, pp. 24–27 (see p. 198)

[75] Arthur Evans Jr. *PAL (Pedagogic Algorithmic Language) A Reference Manual and A Primer* Unpublished Technical Report (see http://www.softwarepreservation.org/projects/lang/PAL/Pal-refman.pdf) Cambridge, Massachusetts, USA: MIT Department of Electrical Engineering, Feb. 1968 (see p. 82)

[76] Joseph R. Falcone. “A Programmable Interface Language for Heterogeneous Distributed Systems” in: *ACM Transactions on Computer Systems*, **5**:4 (1987), pp. 330–351 (see p. 37)

[77] Marc Feeley. “Compiling for Multi-language Task Migration” in: *Proceedings of the 11th Symposium on Dynamic Languages* DLS’15 New York City, New York, USA: ACM, Oct. 2015, pp. 63–77 (see p. 248)

[78] Marc Feeley and Guy Lapalme. “Using Closures for Code Generation” in: *Computer Languages*, **12**:1 (1987), pp. 47–66 (see p. 45)

[79] Matthias Felleisen. “The Theory and Practice of First-Class Prompts” in: *Proceedings of the Symposium on Principles of Programming Languages* ACM, Jan. 1988, pp. 180–190 (see p. 61)

[80] Matthias Felleisen and Daniel P Friedman. “A Closer Look at Export and Import Statements” in: *Computer Languages*, **11**:11 (Jan. 1986), pp. 29–37 (see p. 82)

[81] I. Fette and A. Melnikov. *The WebSocket Protocol* RFC 6455 Dec. 2011 (see pp. 26, 28)

[82] Zara Field, P. W. Trinder, and André Rauber Du Bois. "A Comparative Evaluation of Three Mobile Languages" in: *Proceedings of the 3rd International Conference on Mobile Technology, Applications & Systems* ACM, Oct. 2006 (see p. 45)

[83] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis California, USA: University of California, Irvine, 2000 (see p. 26)

[84] Roy T. Fielding et al. *RFC 2616 Hypertext Transfer Protocol HTTP/1.1* IETF Jun. 1999 (see pp. 28, 93)

[85] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture" in: *Proceedings of the 22nd International Conference on Software Engineering* IEEE Limerick, Ireland, May 2000, pp. 407–416 (see p. 28)

[86] Roy T. Fielding and Richard N. Taylor. "Principled Design of the Modern Web Architecture" in: *ACM Transactions on Internet Technology*, **2**:2 (May 2002), pp. 115–150 (see pp. 28, 29, 31, 61)

[87] Robert Bruce Findler and Matthias Blume. "Contracts As Pairs of Projections" in: *Proceedings of the 8th International Conference on Functional and Logic Programming* FLOPS 2006 Berlin/Heidelberg, Germany: Springer-Verlag, 2006, pp. 226–241 (see p. 241)

[88] Robert Bruce Findler and Matthias Blume. *Contracts as Pairs of Projections* Technical Report TR-2006-01 Chicago, Illinois, USA: University of Chicago, 2006 (see p. 241)

[89] Matthew Flatt and Robert Bruce Findler. "Kill-Safe Synchronization Abstractions" in: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* PLDI 2004 New York, New York, USA: ACM, Jun. 2004, pp. 47–58 (see p. 223)

[90] Michael Fogus and Chris Houser. *The Joy of Clojure*. 2nd ed. Cherry Hill, New Jersey, USA: Manning Publications, 2014 (see p. 102)

[91] Michael Fogus and Chris Houser. *The Joy of Clojure: Thinking the Clojure Way*. Manning Publications, Apr. 2011 (see p. 102)

[92] Philip W. L. Fong. *Viewer's Discretion: Host Security in Mobile Code Systems* Technical Report 1998-19 Burnaby, British Columbia, Canada: School of Computing Science, Simon Fraser University, Nov. 1998 (see pp. 83, 85)

[93] Anders Fongen. *Intrusion Tolerant Systems* FFI-Rapport 2007/02611 Norwegian Defence Research Establishment, Dec. 2007 (see p. 33)

[94] Bryan Ford and Russ Cox. "Vx32: Lightweight User-level Sandboxing on the x86" in: *USENIX 2008 Annual Technical Conference* USENIX ACT'08 Berkeley, California, USA: USENIX Association, 2008, pp. 293–306 (see p. 16)

[95] *Foundation for Intelligent Physical Agents (FIPA)* `http://www.fipa.org` Jan. 2013 (see p. 47)

[96] Cédric Fournet and Georges Gonthier. "The Join Calculus: A Language for Distributed Mobile Programming" in: *Proceedings of the Applied Semantics Summer School* APPSEM'00 Springer-Verlag, 2000, pp. 268–332 (see p. 45)

[97] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. 3rd ed. Cambridge, Massachusetts, USA: MIT Press, 2008 (see p. 42)

[98] Matthew Fuchs. *Dreme: for Life in the Net*. PhD thesis New York City, New York, USA: New York University, Sep. 1995 (see p. 42)

[99] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. "Understanding Code Mobility" in: *IEEE Transactions on Software Engineering*, **24**:5 (1998), pp. 342–361 (see pp. 19, 127)

[100] *FUSE: Filesystem in Userspace* http://fuse.sourceforge.net/ Oct. 2004 (see p. 16)

[101] Jesse James Garrett. *Ajax: A New Approach to Web Applications* http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/ Feb. 2005 (see p. 28)

[102] David Gelernter, Suresh Jagannathan, and Thomas London. "Environments as First Class Objects" in: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages* New York City, New York, USA: ACM, 1987, pp. 98–110 (see pp. 83, 157)

[103] Guillaume Germain, Marc Feeley, and Stefan Monnier. "Concurrency Oriented Programming in Termite Scheme" in: *Scheme and Functional Programming Workshop* 2006, pp. 125–136 (see pp. 44, 48)

[104] Guillaume Germain, Marc Feeley, and Stefan Monnier. "Concurrency Oriented Programming in Termite Scheme" in: *Proceedings of Scheme and Functional Programming Workshop* Sep. 2006, pp. 125–136 (see p. 99)

[105] Matias Giorgio, Michael M. Gorlick, Kari Nies, and Richard N. Taylor. "Capability Accounting in Decentralized Systems as a Means to Achieve Architectural Accountability" unpublished manuscript Mar. 2015 (see p. 239)

[106] Matias Giorgio and Richard N. Taylor. *Accountability Through Architecture for Decentralized Systems: A Preliminary Assessment* Technical Report UCI-ISR-15-2 http://isr.uci.edu/sites/isr.uci.edu/files/techreports/UCI-ISR-15-2.pdf University of California, Irvine: Institute for Software Research, Oct. 2015 (see pp. 239, 240, 242, 246)

[107] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Safe and Automatic Live Update for Operating Systems" in: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* ASPLOS'13 New York City, New York, USA: ACM, 2013, pp. 279–292 (see pp. 177, 187)

[108] Cristiano Giuffrida and Andrew S. Tanenbaum. "Cooperative update: a new model for dependable live update" in: *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades* HotSWUp'09 ACM, 2009 (see p. 187)

[109] Adele Goldberg and Alan Kay. *Smalltalk-72 Instruction Manual* Technical Report SSL 76-6 Available as http://bitsavers.trailing-edge.com/pdf/xerox/parc/techReports/Smalltalk-72_Instruction_Manual_Mar76.pdf Palo Alto, California, USA: Xerox Palo Alto Research Center, Mar. 1976 (see p. 102)

[110] Michael M. Gorlick, Justin R. Erenkrantz, and Richard N. Taylor. *The Infrastructure of a Computational Web* Technical Report UCI-ISR-10-3 University of California, Irvine: Institute for Software Research, May 2010 (see pp. 5, 14, 31, 32)

[111] Michael M. Gorlick, Samuel D. Gasster, Grace S. Peng, and Michael McAtee. "Flow Webs: Architecture and Mechanism for Sensor Webs" in: *Proceedings of the Ground Systems Architecture Workshop* Manhattan Beach, California, USA, Mar. 2007 (see p. 31)

[112] Michael M. Gorlick and Rami R. Razouk. "Using Weaves for Software Construction and Analysis" in: *Proceedings of the 13th International Conference on Software Engineering* ICSE'91 Austin, Texas USA, 1991, pp. 23–34 (see pp. 18, 55, 60, 129, 166)

[113] Michael M. Gorlick, Kyle Strasser, and Richard N. Taylor. "COAST: An Architectural Style for Decentralized On-Demand Tailored Services" in: *WICSA/ECSA'12* Aug. 2012, pp. 71–80 (see pp. 9, 32, 61, 175, 189)

[114] Michael M. Gorlick and Richard N. Taylor. "Communication and Capability URLs in COAST-based Decentralized Services" in: *REST: Advanced Research Topics and Practical Applications* ed. by Cesare Pautasso, Erik Wilde, and Rosa Alarcon New York City, New York, USA: Springer, Jan. 2014, pp. 9–25 (see p. 32)

[115] Michael M Gorlick and Richard N. Taylor. *Motile: Reflecting an Architectural Style in a Mobile Code Language* Technical Report UCI-ISR-13-1 Institute for Software Research, University of California, Irvine, Jun. 2013 (see p. 234)

[116] Robert S. Gray. "Agent Tcl: A Flexible and Secure Mobile-Agent System" in: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop* TCLTK'96 Berkeley, California, USA: USENIX Association, 1996 (see p. 46)

[117] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. "D'Agents: Security in a Multiple-Language, Mobile-Agent System" in: *Mobile Agents and Security* ed. by Giovanni Vigna Springer-Verlag, 1998, pp. 154–187 (see p. 90)

[118] Matthew Green. *Cryptographic obfuscation and "unhackable" software* `http://blog.cryptographyengineering.com/2014/02/cryptographic-obfuscation-and.html` Feb. 2014 (see p. 75)

[119] *GStreamer: open source multimedia framework* `http://gstreamer.freedesktop.org` Feb. 2012 (see p. 100)

[120] Sabine Habert and Laurence Mosseri. "COOL: Kernel Support for Object-Oriented Environments" in: *SIGPLAN Notices*, **25**:10 (1990), pp. 269–275 (see p. 41)

[121] David Alan Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis Cambridge, UK: University of Cambridge, Jun. 1997 (see pp. 30, 42)

[122] Norm Hardy. "The Confused Deputy: (or why capabilities might have been invented)" in: *SIGOPS Operating Systems Review*, **22**:4 (1988), pp. 36–38 (see p. 17)

[123] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. "Efficient, General-purpose Dynamic Software Updating for C" in: *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **36**:4 (Oct. 2014), 13:1–13:38 (see p. 187)

[124] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence" in: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* IJCAI 1973 San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245 (see p. 102)

[125] Michael Hicks. *Dynamic Software Updating*. PhD thesis Philadelphia, Pennsylvania, USA: Computer and Information Science, University of Pennsylvania, 2001 (see pp. 165, 170)

[126] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Mar. 2013 (see p. 69)

[127] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types" in: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* POPL 2008 New York City, New York, USA: ACM, 2008, pp. 273–284 (see p. 247)

[128] Kohei Honda, Raymond Hu, Rumyana Neykova, et al. "Structuring Communication with Session Types" in: *Concurrent Objects and Beyond* ed. by Gul Agha et al. vol. 8665 Lecture Notes in Computer Science Berlin/Heidelberg, Germany: Springer, 2014, pp. 105–127 (see p. 247)

[129] Vincent C. Hu et al. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations* Special Publication 800-162 U.S Department of Commerce: National Institute of Standards and Technology (NIST), Jan. 2014 (see p. 76)

[130] Galen C. Hunt and James R. Larus. “Singularity: Rethinking the Software Stack” in: *ACM SIGOPS Operating Systems Review*, **41**:2 (Apr. 2007), pp. 37–49 (see p. 18)

[131] Graham Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007 (see p. 45)

[132] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. “The implementation of Lua 5.0” in: *Journal of Universal Computer Science*, **11**:7 (Jul. 2005), pp. 1159–1176 (see p. 89)

[133] Suresh Jagannathan. “Dynamic Modules in Higher-Order Languages” in: *Proceedings of the IEEE International Conference on Computer Languages* May 1994, pp. 74–87 (see pp. 61, 88)

[134] Suresh Jagannathan. “Metalevel Building Blocks for Modular Systems” in: *ACM Transactions on Programming Languages and Systems*, **16**:3 (May 1994), pp. 456–492 (see p. 83)

[135] Paul James. *A RESTful Web service, an example* `http://www.peej.co.uk/articles/restfully-delicious.html` Oct. 2005 (see pp. 193, 199, 209, 211)

[136] Yves Jaradin, Fred Spiessens, and Peter Van Roy. *SCOLL: A Language for Safe Capability Based Collaboration* Technical Report Louvain-la-Neuve, Belgium: Université catholique de Louvain, 2005 (see p. 140)

[137] Erik Jul, Henry Levy, Norm Hutchinson, and Andrew Black. “Fine-Grained Mobility in the Emerald System” in: *ACM Transactions on Computer Systems*, **6**:1 (Feb. 1988), pp. 109–133 (see pp. 39, 101, 128)

[138] Neeran M. Karnik and Anand R. Tripathi. “Security in the Ajanta Mobile Agent System” in: *Software - Practice and Experience*, **31**:4 (Apr. 2001), pp. 301–329 (see p. 47)

[139] Neeran Karnik and Anand R. Tripathi. “Agent Server Architecture for the Ajanta Mobile-Agent System” in: *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications* PDPTA’98 1998, pp. 66–73 (see p. 47)

[140] Daniel J. Kaufman. “An Analytical Framework for Cyber Security” in: *Colloquium on Future Directions in Cyber Security* Defense Advanced Research Projects Agency, Nov. 2011 (see p. 2)

[141] Alan Kay. “Computer Software” in: *Scientific American*, **251**:3 (1984), pp. 53–59 (see p. 45)

[142] Richard A. Kelsey and Jonathan A. Rees. “A Tractable Scheme Implementation” in: *LISP and Symbolic Computation*, **7**:4 (Dec. 1994), pp. 315–335 (see p. 89)

[143] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). *Revised^5 Report on the Algorithmic Language Scheme* Feb. 1998 (see pp. 64, 141, 142, 160)

[144] Taesoo Kim and Nickolai Zeldovich. “Practical and Effective Sandboxing for Non-root Users” in: *Proceedings of the 2013 USENIX Annual Technical Conference* USENIX ATC’13 Berkeley, California, USA: USENIX Association, 2013, pp. 139–144 (see p. 16)

[145] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. “The Orc Programming Language” in: *Proceedings of FMOODS/FORTE 2009* Springer, Jun. 2009, pp. 1–25 (see p. 100)

[146] Gerwin Klein, June Andronick, Kevin Elphinstone, et al. “seL4: Formal Verification of an Operating-System Kernel” in: *Communications of the ACM*, **53**:6 (Jun. 2010), pp. 107–115 (see p. 18)

[147] Leif Tobias Kornstäedt. *Design and Implementation of a Programmable Middleware*. PhD thesis Saarbrüken, Germany: Universität des Saarlandes, Apr. 2006 (see p. 141)

[148] Ramachandra Kota, Nicholas Gibbins, and Nicholas R. Jennings. “Decentralized Approaches for Self-Adaptation in Agent Organizations” in: *ACM Transactions on Autonomous and Adaptive Systems*, **7**:1 (May 2012), 1:1–1:28 (see p. 47)

[149] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. "Agent Tcl: Targeting the Needs of Mobile Computers" in: *IEEE Internet Computing*, **1**:4 (Jul. 1997), pp. 58–67 (see p. 46)

[150] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace* RFC 4122 IETF, Jul. 2005 (see p. 69)

[151] Ole Lensmar. *Is REST losing its flair – REST API Alternatives* `http://www.programmableweb.com/news/rest-losing-its-flair-rest-api-alternatives/analysis/2013/12/19` Dec. 2013 (see p. 22)

[152] Xavier Leroy et al. *The OCaml system: Documentation and User's Manual* 4.0 `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/index.html` Institut National de Recherche en Informatique et en Automatique (INRIA) Lille, France, Jul. 2012 (see p. 45)

[153] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. 2nd ed. Prentice-Hall, Apr. 1999 (see p. 89)

[154] Sam Lindley and J. Garrett Morris. "Lightweight Functional Session Types" Available as `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.713.9156&rep=rep1&type=pdf` Feb. 2015 (see p. 247)

[155] B. Liskov, R. Atkinson, T. Bloom, E. Moss, Schaffert J. C., R. Scheifler, and A. Snyder. *CLU Reference Manual*. vol. 114 Lecture Notes in Computer Science New York City, New York, USA: Springer-Verlag, 1981 (see p. 38)

[156] Pu Liu. *Mobile Code Enabled Web and Grid Services*. PhD thesis Binghamton, New York, USA: Computer Science Department, Binghamton University, Aug. 2006 (see p. 191)

[157] Pu Liu and Michael J. Lewis. "Mobile Code Enabled Web Services" in: *Proceedings of IEEE/ACM International Conference on Web Services* ICWS'05 Jul. 2005, pp. 161–174 (see p. 191)

[158] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning Publications, Nov. 2010 (see pp. 102, 170)

[159] Sergio Maffeis, John C. Mitchell, and Ankur Taly. "Object Capabilities and Isolation of Untrusted Web Applications" in: *Proceedings of the IEEE Symposium on Security and Privacy* Washington, D.C., USA: IEEE Computer Society, May 2010, pp. 125–140 (see pp. 224, 226, 227, 238)

[160] Jacob Matthews, Robert Bruce Findler, Paul Graunke, Shriram Krishnamurthi, and Matthias Felleisen. "Automatically Restructuring Programs for the Web" in: *Automated Software Engineering*, **11**:4 (Oct. 2004), pp. 337–364 (see pp. 28, 30)

[161] David May. *The XMOS XS1 Architecture* XMOS Limited, Bristol, UK, 2009 (see p. 90)

[162] Molly McHugh. *Slack Is Overrun With Bots. Friendly, Wonderful Bots* `http://www.wired.com/2015/08/slack-overrun-bots-friendly-wonderful-bots/` Aug. 2015 (see p. 24)

[163] Nenad Medvidovic and Richard N. Taylor. "Software Architecture: Foundations, Theory, and Practice" in: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* vol. 2 ICSE'10 May 2010, pp. 471–472 (see p. 3)

[164] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, et al. "Flapjax: A Programming Language for Ajax Applications" in: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* OOPSLA'09 Oct. 2009, pp. 1–20 (see pp. 28, 101)

[165] James S. Miller and Guillermo Juan Rozas. "Free Variables and First-Class Environments" in: *Lisp and Symbolic Computation*, **4**:2 (Apr. 1991), pp. 107–141 (see p. 83)

[166] Mark S. Miller. *Promise Pipelining: Distributed Programming Made Practical* http://zesty.ca/promises.pdf Nov. 2004 (see p. 102)

[167] Mark S. Miller and Jonathan S. Shapiro. "Paradigm Regained: Abstraction Mechanisms for Access Control" in: *Eighth Asian Computing Science Conference* ASIAN'03 Available as http://www.hpl.hp.com/techreports/2003/HPL-2003-222.pdf Springer-Verlag, Dec. 2003, pp. 224–242 (see p. 102)

[168] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. "The Structure of Authority: Why Security Is not a Separable Concern" in: *Multiparadigm Programming in Mozart/Oz* ed. by Peter Van Roy vol. Lecture Notes in Computer Science 3389 Springer, 2005, pp. 2–20 (see p. 102)

[169] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. *Capability Myths Demolished* Technical Report SRL2003-02 Systems Research Laboratory, Johns Hopkins University, 2003 (see p. 228)

[170] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis Baltimore, Maryland, USA: Johns Hopkins University, May 2006 (see pp. 3, 17, 18, 47, 56, 91, 128, 140, 224, 226, 235)

[171] Mark S. Miller, Mike Samuel, et al. *Safe active content in sanitized JavaScript* http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf Jun. 2008 (see pp. 18, 233)

[172] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge, UK: Cambridge University Press, 1999 (see p. 55)

[173] *MIT/GNU Scheme 9.0.1* http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/ Massachusetts Institute of Technology 2010 (see p. 157)

[174] James H. Morris Jr. "Protection in Programming Languages" in: *Communications of the ACM*, **16**:1 (Jan. 1973), pp. 15–21 (see p. 102)

[175] Derek 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 (see p. 101)

[176] Derek G. Murray and Steven Hand. "Scripting the Cloud with Skywriting" in: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* HotCloud'10 USENIX Association, 2010 (see p. 101)

[177] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 1981 (see pp. 31, 37)

[178] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, et al. "You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions" in: *Proceedings of the 2012 ACM Conference on Computer and Communications Security* CCS 2012 New York City, New York, USA: ACM, Oct. 2012, pp. 736–747 (see p. 233)

[179] Elth Ogston and Frances Brazier. "AgentScope: Multi-Agent Systems Development in Focus" in: *The 10th International Conference on Autonomous Agents and Multiagent Systems* AAMAS'11 International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 389–396 (see p. 47)

[180] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998 (see pp. 59, 132)

[181] Wouter van Oortmerssen. *The Bla Language: Extending Functional Programming with First-Class Environments*. Available at http://strlen.com/bla-language. MA thesis University of Amsterdam, Netherlands, May 1996 (see p. 157)

[182] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, and David Rosenblum. "An Architecture-based Approach to Self-Adaptive Software" in: *IEEE Intelligent Systems*, **14**:3 (May 1999), pp. 54–62 (see pp. 18, 187)

[183] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. "Architecture-Based Runtime Software Evolution" in: *Proceedings of the 20th International Conference on Software Engineering* ICSE'98 Kyoto, Japan, Apr. 1998, pp. 177–186 (see pp. 18, 60)

[184] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. "Runtime Software Adaptation: Framework, Approaches, and Styles" in: *Companion of 30th International Conference on Software Engineering* ICSE Companion 2008 ACM, May 2008, pp. 899–910 (see p. 60)

[185] Peyman Oreizy and Richard N. Taylor. "On the Role of Software Architectures in Runtime System Reconfiguration" in: *Proceedings of the International Conference on Configurable Distributed Systems* CDS'98 Washington, D.C., USA: IEEE Computer Society, 1998 (see p. 166)

[186] *OTP Design Principles User's Guide, Version 5.9.2* Ericsson AB Sep. 2012 (see p. 170)

[187] John K. Ousterhout, Jacob Y. Levy, and Brent B. Welch. "The Safe-Tcl Security Model" in: *Mobile Agents and Security* Springer-Verlag, 1998, pp. 217–234 (see p. 46)

[188] Dimokritos Michael Papadopoulos and Kenneth Folmer-Petersen. *Porting the LAN-based distributed system Emerald to a WAN*. MA thesis Copenhagen, Denmark: DIKU, Department of Computer Science, University of Copenhagen, 1993 (see p. 39)

[189] Sean Peisert, Ed Talbot, and Matt Bishop. "Turtles All The Way Down: A Clean-Slate, Ground-Up, First-Principles Approach to Secure Systems" in: *New Security Paradigms Workshop* NFS'12 Sep. 2012, pp. 15–25 (see p. 223)

[190] W. Michael Petullo, Xu Zhang, Jon A. Solworth, Daniel J. Bernstein, and Tanja Lange. "MinimaLT: Minimal-latency Networking Through Better Security" in: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* CCS'13 New York City, New York, USA: ACM, 2013, pp. 425–438 (see p. 69)

[191] Simon Peyton-Jones and Jean-Marc Eber. "How to Write a Financial Contract" in: *The Fun of Programming* ed. by Jeremy Gibbons and Oege de Moor Cornerstones of Computing Palgrave Macmillan, Jun. 2005 (see p. 54)

[192] Adrien Piérard and Marc Feeley. "Towards a Portable and Mobile Scheme Interpreter" in: *Proceedings of the Scheme and Functional Programming Workshop* Sep. 2007, pp. 59–68 (see pp. 44, 99)

[193] Matthew Pirretti, Patrick Traynor, Patrick McDaniel, and Brent Waters. "Secure Attribute-based Systems" in: *Proceedings of the 13th ACM Conference on Computer and Communications Security* CCS'06 ACM, Oct. 2006, pp. 99–112 (see p. 76)

[194] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. "Type-Based Verification of Web Sandboxes" in: *Journal of Computer Security*, **22**:4 (Jul. 2014), pp. 511–565 (see p. 233)

[195] James Purtilo. *A Software Interconnection Technology to Support Specification of Computational Environments*. PhD thesis Urbana, Illinois, USA: University of Illinois, Urbana-Champaign, 1986 (see p. 166)

[196] James Purtilo. "The Polylith Software Bus" in: *ACM Transactions on Programming Languages and Systems*, **16**: (1991), pp. 151–174 (see p. 166)

[197] Christian Queinnec. "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers" in: *Proceedings of the Fifth ACM International Conference on Functional Programming* ICFP'00 ACM New York City, New York, USA: ACM, Sep. 2000, pp. 23–33 (see pp. 28, 30)

[198] Christian Queinnec and David de Roure. “Sharing Code through First-Class Environments” in: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming* New York City, New York, USA: ACM, Jun. 1996, pp. 251–261 (see p. 83)

[199] Jonathan A. Rees. *A Security Kernel Based on the Lambda Calculus*. PhD thesis Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, Feb. 1995 (see pp. 102, 228, 235)

[200] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual* 4th ed. Computer Science Department, Yale University New Haven, Connecticut, 1984 (see p. 82)

[201] Brian Reid. *PostScript and Interpress: a Comparison* `https://groups.google.com/forum/?fromgroups=#!msg/fa.laser-lovers/H3us4h8S3Kk/-vGRDirzDV0J` Mar. 1985 (see p. 36)

[202] Steven P. Reiss. “Connecting Tools Using Message Passing in the FIELD Environment” in: *IEEE Software*, **7**:4 (Jul. 1990), pp. 57–67 (see pp. 18, 166)

[203] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. “Alice Through the Looking Glass” in: *Trends in Functional Programming* vol. 5 Bristol, UK: Intellect Books, Feb. 2006 (see p. 247)

[204] Andreas Rossberg, Guido Tack, and Leif Kornstaedt. “Status Report: HOT Pickles, and How to Serve Them” in: *Proceedings of the 2007 Workshop on Workshop on ML* ML’07 New York City, New York, USA: ACM, Oct. 2007, pp. 25–36 (see pp. 141, 247)

[205] Volker Roth and Mehrdad Jalali-Sohi. “Concepts and Architecture of a Security-Centric Mobile Agent Server” in: *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems* ISADS’01 IEEE Computer Society, Mar. 2001, pp. 435–441 (see p. 47)

[206] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails (Fourth Edition)*. Pragmatic Bookshelf, Mar. 2011 (see p. 100)

[207] Jeff Rulifson. *Decode-Encode Language (DEL)* RFC 5 The Internet Society, Jun. 1969 (see p. 36)

[208] Jerome H. Saltzer. “Protection and the Control of Information Sharing in Multics” in: *Communications of the ACM*, **17**:7 (Jul. 1974), pp. 388–402 (see p. 17)

[209] Martin Scheffler. *E Capabilities* `http://wiki.erights.org/wiki/Walnut/Secure_Distributed_Computing/E_Capabilities` Jul. 2009 (see p. 223)

[210] Bruce Schneier. *Liars and Outliers: Enabling the Trust that Society Needs to Thrive*. John Wiley & Sons, Feb. 2012 (see p. 2)

[211] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. “Computational Contracts” in: *Science of Computer Programming*, (2013) (see pp. 197, 240, 241)

[212] Z. Cliffe Schreuders, Tanya McGill, and Christian Payne. “The State of the Art of Application Restrictions and Sandboxes: A Survey of Application-Oriented Access Controls and their Shortfalls” in: *Computers & Security*, **32**: (Feb. 2013), pp. 219–241 (see p. 16)

[213] Jonathan S. Shapiro and Norman Hardy. “EROS: A Principle-Driven Operating System from the Ground Up” in: *IEEE Software*, **19**:1 (Jan. 2002), pp. 26–33 (see p. 18)

[214] Alan Shieh, Andrew C. Myers, and Emin G. Sirer. “Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility” in: *Proceedings of Symposium on Networked Systems Design and Implementation* vol. 2 NSDI’05 USENIX Association, May 2005, pp. 175–188 (see p. 28)

[215] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. “A Stateless Approach to Connection-Oriented Protocols” in: *ACM Transactions on Computer Systems*, **26**:3 (Sep. 2008), 8:1–8:50 (see p. 30)

[216] Alex Shinn, John Cowan, and Arthur R. Gleckler (Editors). *Revised[7] Report on the Algorithmic Language Scheme* Jul. 2013 (see pp. 130, 141, 142, 160–162, 239)

[217] Olin Shivers. *A Scheme Shell* Technical Report MIT-LCS-TR-635 Cambridge, Massachusetts, USA: MIT Laboratory for Computer Science, Apr. 1994 (see p. 89)

[218] Jonathon M. Smith. *Reflections on Active Networking* Technical Report MS-CIS-03-05 University of Pennsylvania, Department of Computer and Information Science, Jan. 2003 (see p. 36)

[219] Jon A. Solworth and Wenyuan Fei. "sayI: Trusted User Authentication at Internet Scale" `https://www.ethos-os.org/~solworth/sayIgroups-20130614.pdf` Jun. 2013 (see p. 68)

[220] Alfred Spiessens. *Patterns of Safe Collaboration.* PhD thesis Louvain-la-Neuve, Belgium: Universite Catholique de Louvain, Feb. 2007 (see pp. 140, 224, 235)

[221] Fred Spiessens, Yves Jaradin, and Peter Van Roy. *Using Constraints To Analyze And Generate Safe Capability Patterns* `https://www.info.ucl.ac.be/~pvr/rr2005-11.pdf` 2005 (see p. 140)

[222] Fred Spiessens and Peter Van Roy. "The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language" in: *Proceedings of the Second International Conference on Multiparadigm Programming in Mozart/Oz* MOZ 2004 Berlin/Heidelberg, Germany: Springer-Verlag, 2005, pp. 21–40 (see p. 235)

[223] A. Spinks. *The NODAL system (software) training lecture given by G. Shering* CERN-SPS-Tech-Note 77-015-AOP Geneva, Switzerland: CERN, 1977 (see p. 36)

[224] Michael B. Spring. *Electronic Printing and Publishing.* in: CRC Press, Apr. 1991, pp. 174–178 (see p. 36)

[225] Michal Srb. *Haskell-Like S-Expression-Based Language Designed for an IDE* Individual Project Implementation available at `https://github.com/xixixao/Shem` London, UK: Department of Computing, Imperial College, Jun. 2015 (see p. 247)

[226] James W. Stamos and David K. Gifford. "Implementing Remote Evaluation" in: *IEEE Transactions on Software Engineering*, **16**:7 (Jul. 1990), pp. 710–722 (see pp. 38, 99)

[227] James W. Stamos and David K. Gifford. "Remote Evaluation" in: *ACM Transactions on Programming Languages and Systems*, **12**:4 (Oct. 1990), pp. 537–564 (see pp. 19, 31, 38)

[228] James William Stamos. *Remote Evaluation.* (MIT Technical Report MIT/LCS/TR-354) PhD thesis Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, Jan. 1986 (see pp. 19, 20, 26, 31, 38, 84)

[229] Bjarne Steensgaard and Eric Jul. "Object and Native Code Thread Mobility Among Heterogeneous Computers" in: *In Proceedings of the 15th ACM Symposium on Operating Systems Principles* 1995, pp. 68–78 (see pp. 39, 90)

[230] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.* 2nd edition John Wiley & Sons, Sep. 2011 (see pp. 26, 33, 194)

[231] Alexander J. Summers and Peter Müller. "Actor Services: Modular Verification of Message Passing Programs" in: *Programming Languages and Systems: 25th European Symposium on Programming* ESOP 2016 Berlin/Heidelberg, Germany: Springer, Apr. 2016, pp. 699–726 (see p. 235)

[232] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. "Is Sound Gradual Typing Dead?" in: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* POPL'16 New York City, New York, USA: ACM, 2016, pp. 456–468 (see p. 247)

[233] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. "Automated Analysis of Security-Critical JavaScript APIs" in: *Proceedings of the 2011 IEEE Symposium on Security and Privacy* IEEE, May 2011, pp. 363–378 (see p. 233)

[234] Paul Tarau and Veronica Dahl. "High-Level Networking with Mobile Code and First-Order AND-Continuations" in: *Theory and Practice of Logic Programming*, **1**:3 (May 2001), pp. 359–380 (see p. 30)

[235] Joseph Tardo and Luis Valente. "Mobile Agent Security and Telescript" in: *Proceedings of the 41st IEEE International Computer Conference* COMPCON'96 Washington, D.C., USA: IEEE Computer Society, 1996, pp. 58–63 (see p. 46)

[236] Richard N. Taylor. "Software Architecture: Many Faces, Many Places, Yet a Central Discipline" in: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* ESEC/FSE'09 Amsterdam, The Netherlands, Aug. 2009, pp. 303–304 (see p. 31)

[237] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. "A component- and message-based architectural style for GUI software" in: *Proceedings of the 17th International Conference on Software Engineering* ICSE'95 1995, pp. 295–304 (see p. 18)

[238] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Jan. 2010 (see pp. 3, 19, 236)

[239] Richard N. Taylor, Nenad Medvidovic, and Peyman Oreizy. "Architectural Styles for Runtime Software Adaptation" in: *Proceedings of the Eighth Joint Working IEEE/IFIP Conference on Software Architecture and Third European Conference on Software Architecture* IEEE Computer Society, 2009, pp. 171–180 (see pp. 18, 19, 166, 167)

[240] Richard N. Taylor, Nenad Medvidovic, et al. "A Component- and Message-Based Architectural Style for GUI Software" in: *Transactions on Software Engineering*, (Jun. 1996), pp. 390–406 (see p. 99)

[241] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, et al. "A Component- and Message-based Architectural Style for GUI Software" in: *Proceedings of the 17th International Conference on Software Engineering* ICSE'95 New York City, New York, USA: ACM, 1995, pp. 295–304 (see p. 166)

[242] *Telescript Language Reference* `http://bitsavers.informatik.uni-stuttgart.de/pdf/generalMagic/Telescript_Language_Reference_Oct95.pdf` Sunnyvale, California, USA: General Magic Inc., Oct. 1995 (see p. 46)

[243] *Telescript Programming Guide, Version 0.5 (ALPHA)* `https://www.cis.upenn.edu/~bcpierce/courses/629/papers/Telescript-ProgGuide.ps.gz` Sunnyvale, California, USA: General Magic, Inc., Oct. 1995 (see pp. 41, 46)

[244] Aaron Timms. *The Race to Topple Bloomberg* `http://www.institutionalinvestor.com/Article/3303623/The-Race-to-Topple-Bloomberg.html` Jan. 2014 (see p. 24)

[245] Davide Tosi, Giovanni Denaro, and Mauro Pezzé. *Experimental data on service interchangeability* Technical Report LTA:2008:01 University of Milano-Bicocca, Sep. 2008 (see pp. 193, 194)

[246] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition)*. Prentice Hall, Aug. 2006 (see p. 101)

[247] E. Dean Tribble, Mark S. Miller, Norm Hardy, and David Krieger. *Joule: Distributed Application Foundations* Agorics Technical Report ADd003.4P `http://www.erights.org/history/joule/index.html` Los Altos, California, USA: Agorics, Inc., Dec. 1995 (see p. 102)

[248] Anand R. Tripathi, Neeran M. Karnik, Manish K. Vora, Tanvir Ahmed, and Ram D. Singh. "Mobile Agent Programming in Ajanta" in: *Proceedings of the 19th International Conference on Distributed Computing Systems* ICDCS'99 1999, pp. 190–197 (see p. 47)

[249] David Ungar and Randall B. Smith. "Self: The Power of Simplicity" in: *SIGPLAN Notices*, **22**:12 (1987), pp. 227–242 (see p. 40)

[250] Ruben Vandamme. *Implementing Concurrency Abstractions for Programming Multi-Core Embedded Systems in Scheme*. MA thesis Vrije Universiteit Brussel, 2010 (see p. 90)

[251] Steven R. Vegdahl. "Moving Structures Between Smalltalk Images" in: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* OOPLSA'86 New York City, New York, USA: ACM, 1986, pp. 466–471 (see p. 40)

[252] Steve Vinoski. "Convenience Over Correctness" in: *IEEE Internet Computing*, **12**:4 (Jul. 2008), pp. 89–92 (see p. 26)

[253] Steve Vinoski. *RPC and its Offspring: Convenient, Yet Fundamentally Flawed* `http://qconlondon.com/london-2009/` Mar. 2009 (see p. 26)

[254] Steve Vinoski. "RPC Under Fire" in: *IEEE Internet Computing*, **9**:5 (Sep. 2005), pp. 93–95 (see p. 26)

[255] Dimitris Vyzovitis and A. Lippman. *MAST: A Dynamic Language for Programmable Networks* Technical Report MIT Media Laboratory, May 2002 (see pp. 30, 43, 103, 125, 157)

[256] David W. Wall. "Messages as Active Agents" in: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* POPL 1982 New York City, New York, USA: ACM, 1982, pp. 34–39 (see p. 36)

[257] Brett Keith Watson. *Network Protocol Design with Machiavellian Robustness*. PhD thesis Sydney, Australia: Macquarie University, Nov. 2010 (see pp. 96, 109)

[258] *Webster's New World Dictionary of the American Language, College Edition*. The World Publishing Company, 1962 (see p. 15)

[259] Danny Weyns and Tom Holvoet. "An Architectural Strategy for Self-Adapting Systems" in: *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems* SEAMS'07 IEEE Computer Society, 2007 (see p. 47)

[260] James E. White. *Mobile Agents* White Paper `http://www.cis.upenn.edu/~bcpierce/courses/629/papers/White-Telescript.ps.gz` Sunnyvale, California, USA: General Magic Inc., 1996 (see p. 46)

[261] James E. White. "Mobile Agents" in: *Software Agents*, (Apr. 1997) ed. by Jeffrey M. Bradshaw, pp. 437–472 (see p. 46)

[262] Adam Wick and Matthew Flatt. "Memory Accounting Without Partitions" in: *Proceedings of the 4th International Symposium on Memory Management* ACM, Oct. 2004, pp. 120–130 (see pp. 16, 223)

[263] Michael Wilde et al. "Swift: A Language for Distributed Parallel Scripting" in: *Parallel Computing*, **37**:9 (2011), pp. 633–652 (see p. 101)

[264] Steven Willmot. *Progress in the API Economy* `http://www.3scale.net/2014/04/reflecting-progress-api-economy/` Apr. 2014 (see pp. 22, 23)

[265] `www.programmableweb.com`. *Growth in Web APIs Since 2005* `http://www.slideshare.net/programmableweb/web-api-growthsince2005` Apr. 2014 (see p. 23)

[266] Edward Z. Yang and David Mazières. "Dynamic Space Limits for Haskell" in: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* PLDI'14 New York City, New York, USA: ACM, 2014, pp. 588–598 (see p. 223)

[267] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code" in: *Communications of the ACM*, **53**:1 (Jan. 2010), pp. 91–99 (see p. 89)

[268] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code" in: *Proceedings of the 30th IEEE Symposium on Security and Privacy* IEEE, May 2009, pp. 79–83 (see p. 89)

[269] Steve Yegge. *Stevey's Google Platforms Rant* `https://plus.google.com/112678702228711889851/posts/eVeouesvaVX` Oct. 2011 (see p. 25)

[270] Aydan R. Yumerefendi and Jeffrey 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: USENIX Association, Jun. 2005 (see pp. 33, 236)

[271] Jens Zander. *Multitasking FORTH Implementation for the 6809, Users Manual* Technical Report LiTH-ISY-I-0577 Stockholm, Sweden: Linkoping University, Dec. 1982 (see p. 36)

[272] Jens Zander and Robert Forchheimer. *Preliminary Specifications for a Distributed Radio Packet Network for Computer and Radio Amateurs* Technical Report LiTH-ISY-0424 Stockholm, Sweden: Linkoping University, Jan. 1980 (see p. 36)

[273] Jens Zander and Robert Forchheimer. "SOFTNET—an approach to high-level packet communication" in: *2nd ARRL Amateur Radio Computer Networking Conference* San Francisco, California: www.tapr.org, 1983 (see p. 36)

[274] Jens Zander and Robert Forchheimer. "The SOFTNET project: a retrospect" in: *Eurocon: Area Communication—Proceedings of the 8th European Conference on Electrotechnics* Stockholm, Sweden, Jun. 1988, pp. 343–345 (see p. 36)