ANDROMEDA—A DISTRIBUTED USERSPACE

Nikos Vasilakis

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

Στους γονείς μου

Acknowledgment

I want to thank my advisor Jonathan M. Smith for his support, advice, and guidance—even on topics far outside the strict realm of a PhD. Jonathan gave me an unparalleled level of freedom, trust, and immunity to research anything I found interesting—no matter how risky or creative it seemed.

If Jonathan was a conscious probe, then Ben Karel and André DeHon were just luck. I have been incredibly lucky to work next to Ben for all these years; he has been the model collaborator for me, influencing my work—and, more generally, thinking—profoundly. Ben is also responsible for coining several terms, including the "distributed userspace". I have been at least as lucky to work with André; the problems I tackled and many of the techniques I employed are direct products of André's vision. Both, I can't thank you enough.

I am grateful for the advice of my Ph.D. committee—Andreas Haeberlen, Joe Devietti, Boon Thau Loo, James Mickens, and Stelios Sidiroglou-Douskos. Their time and effort have improved the quality of my dissertation in *multiple* ways.

Several people have contributed code to ANDROMEDA—you, hackers, I salute: Yash Palkhiwala, Mikael Mantis, Pranjal Goel, John Sonchack, Henry Maxime Demoulin, and Claudia Zhu. I am especially thankful to Yash, who went above and beyond to ensure we meet several conference deadlines.

I am indebted to the rest of my collaborators, from which I have learned a

My research has been supported by DARPA contract no.FA8650-10-C-7090 (SAFE), NSF grant no.CNS-1513687 (HALCYON), the Pompa chair, and gifts from GitHub, VMware, and IntelliJ.

great deal—especially Benjamin C. Pierce, Boon Thau Loo, Cătălin Hriţcu, Isaac Pedisich, Konstantinos Kallas, Linh Thi Xuan Phan, Nathan Dautenhahn, Nick Roessler, Raphael Rubin, Udit Dhawan, and Vincent Liu. I am also grateful to many collaborators outside Penn: Asaf Kariv, Christian-Alexandru Staicu, Dahlia Malkhi, George Chrysanthakopoulos, Jiasi Shen, José Pablo Cambronero, Lena Kanellou, Martin Rinard, Michael Pradel, Pieter Noordhuis, and Thurston Dang. Also—the broader distributed-systems, security, and programming-language groups at Penn.

Advice from and interactions with several other people has been instrumental: Arthur Azevedo De Amorim, Benoît Montagu, Daniel Genkin, Gabrielle De Micheli, John Cotton, Jonida Cali, Justin Hsu, Kostas Gatsis, Leo Lampropoulos, Luke Valenta, Malte Schwarzkopf, Marcella Hastings, Martin Maas, Perry Metzger, Radoslav Ivanov, Robert Broberg, and Yang Wu. Short but pointed advice from other faculty has been extremely helpful: Camillo, George, Kostas, Linh, Matt, Sampath, Stephanie, Steve, Sudipto, Val, Zack. I'm especially grateful to Chris Murphy and Andreas Haeberlen who were always there to listen and consult, and to Mike Felker who always made things easier.

I would not be working in distributed systems if it wasn't for Ioannis Chatzigiannakis; two formative courses and subsequent research quickly convinced me that the accidental complexity of distributed systems is something worth fixing.

I also want to thank the rest of my friends for their support, and apologize for failing to keep in touch as much as I'd wanted: Chris, Dustin, Elpiniki, Giorgo, Irene, Jacques, Josh, Konstantine, Liana, Liz, Katerina, Pavlo, Seylah, Stathi, Venetia, Yanni. Mike, Rado, Ben, and Arthur were always around and ready to help on any problem imaginable—and during very difficult times. The last stretch of the PhD would not have been even remotely as productive if Ania did not bring music into my life—both literally and figuratively.

My deepest gratitude goes to my family. Δύο γραμμές δεν μπορούν καν να εκφράσουν την ευγνωμοσύνη την οποία νιώθω. Σας ευχαριστώ, για όλα.

ABSTRACT

ANDROMEDA—A DISTRIBUTED USERSPACE

Nikos Vasilakis

Jonathan M. Smith

Computing is on a steady trajectory from standalone personal computers to pervasive, heterogeneous, networked computing resources. Distributed software systems are becoming indispensable, but involve a significant (and often unjustifiable) degree of *complexity*. This problem is evident in their setup, development, deployment, and use—all of which are nowhere near their centralized counterparts. Whereas anyone today can write a Bash or Python script to compute on a single computer, scaling out to multiple computers requires expert labor around "point" solutions with expensive setups, restricted programming interfaces, and exorbitant composition costs.

To address this problem, I develop the concept of a *distributed userspace*—a language-based overlay environment that bolts distribution onto a conventional (and light) language runtime as an extensible service module. The environment's programming style melds an unusual combination of elements aimed at lowering the aforementioned complexity without any loss of generality. To tackle a growing space of unavoidable distribution trade-offs, it also provides pervasive support for high-level configurability—*e.g.*, node grouping, context rebinding, and interface transform.

I build a distributed-userspace prototype, Andromeda, targeting the JavaScript ecosystem. While most of its services are "textbook" implementations, three are novel and demonstrate characteristics that are applicable beyond the context of Andromeda: (i) the storage subsystem, which supports efficient queries on dynamic data, (ii) the node group management subsystem, which adds first-class support for network overlays, and (iii) the task execution subsystem, which supports runtime interface rebinding for safety and performance. The key result is that Andromeda lowers the complexity of employing distributed software while addressing setup, transition, and ecosystem challenges. Particularly telling are three frameworks built on top of ANDROMEDA that retrofit desirable properties into legacy applications with minimal developer effort: BREAKAPP spawns components in remote compartments, IRIS protects co-located components from each other, and IGNIS scales out bottlenecked components. These frameworks serve a double purpose: first, they address security and performance issues arising from the complexity-lowering techniques powering ANDROMEDA; and second, they show the benefits of a general-purpose distributed environment outside the strict context of data-intensive computations, highlighting the benefits of ANDROMEDA's design decisions.

Preamble

This section provides a few guidelines on how to read this dissertation.

The PDF version of this document is thoroughly hyperlinked, and reads best in color and on a device supporting linking. However, links are in the same font style and color as the rest of the text, to avoid hindering readability on physical printouts and avoid visual noise on digital devices. To "see" the links, move your cursor over chapter, section, figure, and citation numbers and you will notice it change.

The short git commit identifier for this version of the document is: c9e5c76.

Chapter Interdependencies and Intended Audience Chapter 1 provides a high-level map of the problems and solutions presented in the dissertation; it is geared towards a general audience. Chapter 2 presents a closer overview of overall system; (solutions presented in) later chapters depend somewhat on the design decisions presented here. Chapters 3–6 present four different frameworks built on ANDROMEDA and have no interdependencies—they can be read in any order. Chapter 7 discusses implementation details and low-level challenges—this should be interesting primarily to implementers of similar systems and less so to researchers. Chapter 8 presents evaluation results of workloads executed on the system, including the frameworks presented in Chapters 3–6 (and depends on these). Chapter 9 discusses work related to all of the ideas presented in this dissertation; its sections are clearly marked to correspond to different chapters. Chapter 10 offers concluding remarks, and should be of interest to researchers interested in ANDROMEDA' limitations and ones looking for promising problems to work on; it is geared towards a general audience, too.

Appendix A targets Andromeda' users and everyone else wanting to get an idea of how it feels to program in Andromeda. It is also intended as a short but "hands-on" introduction to JavaScript, the language used throughout the dissertation. Readers lacking that background may want to start here. Appendix B describes the internals of a module system. As most readers are not expected to be familiar modern module systems in dynamic programming languages, this appendix provides background information to the module-level transformations used in Chapters 4–6.

Published Work Andromeda's preliminary design was presented at HotOS [225]. The design and preliminary evaluation of its distributed storage system was presented at APSys [226]. The motivation for BreakApp—and, more generally, today's ubiquitous reliance on third-party packages—was presented at PLOS [223]. The design and implementation of BreakApp was presented at NDSS [224]. The design and implementation of Ignis was presented at PLDI [222]. Andromeda's port for web browsers was presented at EdgeSys [220]. Two Andromeda-related papers have been recently submitted for review: (i) the design and implementation of Iris was submitted to CCS [227]; and (ii) the design and implementation of Andromeda was submitted to SOSP [221].

Correspondence between Papers and Chapters Much of Chapter 1, most of Chapter 2, and the first sections of Chapters 7 and 8 are drawn from the SOSP and HotOS papers [225, 221]. Chapters 3–6 are drawn (mostly) verbatim from the papers they describe [226, 223, 224, 227, 222]. The rest of Chapters 7 and 8 combine implementation and evaluation sections of all the papers. Much of Chapter 10 is drawn from papers, but as open questions in some of the papers were addressed in followup work, this section has been updated and augmented to summarize the most promising open problems and future directions. Appendix A is new, whereas Appendix B is drawn from Iris [227].

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Statement The complexity of building and employing distributed software systems can be significantly improved without sacrificing generality or security.

1.1 Motivation

Computing hardware is becoming a commodity, to the point that devices with non-trivial capabilities such as Raspberry Pis [172] cost a fraction of the price of a graduate textbook. Such an overabundance of cheap hardware resources makes computing increasingly pervasive and networked. Individuals own and operate a growing number of computing devices of various types and forms; companies and organizations employ large fleets of computers to process and store data that exceed any single machine's capacity.

This trend, in turn, shifts distributed software—*i.e.*, one that treats many computers as one—from desirable to indispensable. Distributed software already offers multiple benefits over its centralized counterpart—for example, it can speed up computations, mitigate resource-exhaustion attacks, improve fault-tolerance, and balance load during spikes. But with the one-way transition towards pervasively networked hardware resources, distributed software is becoming the *only* option.

Unfortunately, distributed software involves a significant degree of *complexity*: compared to centralized software, the development and deployment of distributed software remains different and significantly more difficult. Anyone can write a Bash or Python script to compute on a single computer—domain-experts routinely glue scripts together to process and share data, without the help of a computing expert. However, scaling out to multiple computers requires expert labor around "point" solutions with expensive setups, restricted programming interfaces, and exorbitant composition costs [121, 189, 225, 94, 219, 93]. When, contrary to hardware, software is expensive and highly dependent on the inherent cost of human labor, the cost of distribution becomes prohibitive for the average user.

As a result of this complexity, only a minority of computing experts, employed by the select few companies that deal with massive datasets, have the luxury of engineering software systems with distribution baked in from the start. When confronted with a computing ensemble that numbers more than one computer, the remaining majority starts with a laborious process of setting up layers over layers of software—much of which developed to operate in a centralized fashion and utilizing only a fraction of the ensemble's capabilities.

1.2 Case Study: Building & Sharing Results

To understand this complexity, consider the case of converting a few Markdown files to HTML and sharing them over the network. For a user owning *one* computer, this is achievable with a single Unix pipeline:

```
find . '*.md' | xargs mdc | nc -l 80                                  (p1)
```

Program p_1 starts by finding all files with suffix `.md` in the current directory, passes them to `mdc` (a third-party Markdown-to-HTML compiler), and feeds the resulting HTML to `nc`, which makes it available over the network (on port 80).

There are several details about such an integrated environment worth highlighting. UNIX provides a portable system for individual program execution and piecewise synthesis. It abstracts away inessential details such as program naming, loading, and communication. Utilities such as `find` and `nc` are modular building blocks designed for composition under various configurations; defaults are omitted, whereas non-defaults use flags like `-l`. The system is trivially extensible, with add-ons such as `mdc` practically indistinguishable from built-ins such as `find`. Most importantly, *it favors developer convenience over raw performance*, but without loss of generality nor unacceptable performance: p_1 takes a few seconds to compose and execute, when an optimized solution crafted from scratch—*i.e.*, without any UNIX services—would have taken weeks, if not months [83, 18, 19].

Hiding all this complexity is not necessarily a special feature of the UNIX userspace. Other environments, such as Smalltalk or Lisp, would have offered similar, if not better, convenience—all of them, that is, until the user owns more than one computer.

1.3 Prior (Classes of) Approaches

Distribution gives rise to issues that do not exist in centralized systems, such that centralized abstractions cannot be straightforwardly extended to the distributed setting. At a minimum, two computers need to resolve names, communicate results, and direct output back to the user—generally *coordinate* so that they look and feel like a single system. The abstractions available in centralized systems alone are not adequate in the context of distribution as they are far too low-level, impeding the user rather than supporting them. Several classes of approaches have been proposed to aid developers in tackling such issues.

Unfortunately, today's most popular approaches fail to produce a program as simple as p_1. Data-intensive cluster computing frameworks [49, 146, 248, 145, 168] can be only of limited benefit, as they exploit functional purity—applicable only to p_1's `mdc`. Domain-specific languages [9, 22, 128, 57, 140] suffer from a similar lack of generality by making strong assumptions about the nature of distribution: none of p_1's segments feature, say, strong-eventual commutative computations that can proceed in parallel. Finally, paid cloud providers [10, 73, 133] are the poorest match, as they offer coarser building blocks and often on remote, virtualized infrastructure that cannot be employed on the user's two computers. Because of specialization, all three approaches must combine multiple point solutions in a laborious setup process that takes place outside of the distributed environment. Composition is especially tedious and generally dealt *ad hoc*, whereas other issues, such as dependency management and `mdc`-like add-ons, simply remain unaddressed.

Distributed operating systems [169, 236, 159, 117, 25, 143, 166, 178, 56, 14, 189, 182] (DOSes) are better suited to address this problem. In fact, Unix-based ones offer solutions that are identical to p_1. However, their setup and configuration requires a significant investment in terms of time and effort. (Setting up a virtualization layer mostly adds to the base effort, but also affects runtime performance.) Falling out of favor, DOSes end up lacking support and associated ecosystems—*e.g.*, applications, drivers, documentation, *etc.* Worst of all, their one-size-fits-all "distribution transparency" fixes expectations about the nature and scale of distribution: for example, how could `nc` expose results on one, a subset, or all computers?

Distributed programming languages [232, 119, 194, 101, 59, 240] (DPLs) give their users full control over the resulting distribution and lack the ecosystem problems that plague DOSes. However, as in the centralized case, writing software from scratch is difficult, time-consuming, and error-prone. Assumptions about the nature of distribution, such as scale and consistency requirements, are baked into the program at development time. Worst of all, programming languages alone lack any support of a surrounding system: many chores, such as synchronizing configurations, managing tasks, and administering computers, require the user dropping out of the language and back into the underlying centralized system.

Could we lower the complexity of employing distributed systems—say, bring it down to ordinary scripting?

1.4 My Approach: A Distributed Userspace

My approach is a language-based overlay environment built around an existing language runtime and presented to users as an extensible software module. It can be imported into programs written in a conventional language to provide a set of distributed services. These services lift node-local capabilities to their distributed equivalents—*e.g.*, distributed storage, inter-node communication, and task orchestration. I term this approach a *distributed userspace* and prototype it with ANDROMEDA, which renders p_1 as:

```
1 store.get(/.md$/,                                                    (p2)
2   (_, o) => mdc.compile(o,
3     (_, d) => web.expose(d, {port:80}) ) )
```

Program p_2 queries the object-based storage (`store`) across all nodes for objects with names ending in `.md` (1), compiles the results (bound to `o`) to HTML (2), and shares (a copy of) the resulting HTML on every node part of the system (3). Services like `store` and `web` come bundled with ANDROMEDA, whereas `mdc` is a wrapper around a publicly available package.

A couple of tweaks illustrate ANDROMEDA's malleability: (i) have `store` return the most recent update of a strongly-consistent read; and (ii) have `web` expose results on a subset of the user's nodes, possibly different from the one handling files:

```
andromeda.rpi.store.get({k: /.md$/, c: 'strong'},                    (p3)
  (_, d) => andromeda.dsl.web.expose(d, {port:80}) )
```

Three points are worth mentioning.[1] ANDROMEDA's library is namespaced under `andromeda`, a map from groups of nodes to services. Specifying a group has effects on the scale and nature of distribution, by targeting a subset of nodes and a specialized

[1] For brevity, p_3 and p_4 omit Markdown compilation; `key`, `consistency`, and `replication` are shortened to `k`, `c`, and `r`, respectively.

service instance. For example, `store` on `rpi` might default to eventually consistent, $3\times$ replication across six local micro-controllers. Services can be further re-configured at the level of individual calls or messages (*e.g.*, `strong` consistency).

While p_3 addresses several distributed-systems complexities, it could achieve more. Consider tallying `web` requests from all nodes using a global counter on a new node:

```
let count = (opts, cb) => {                                    (p4)
  let f = () => { andromeda.self.mem.total++ }
  andromeda.rpi.store.get(/.md$/, cb)
  andromeda.singleton.tasks.exec(f, ...) }
andromeda.dsl.web.expose(count, {port: 80})
```

Function `count` creates a function `f` that updates `total` on any node it runs. It queries `store` à la p_1, but also sends and (atomically) executes `f` on a single-node group named `singleton`. Methods `get` and `exec` run concurrently, as `exec` is not part of `get`'s continuation.

Program p_4 requires less effort than the equivalent in DPLs[2] and is near-impossible to express in the shell of popular DOSes. It illustrates several aspects related to the programming style: (i) first-class support for value distribution, meaning that functions can be transmitted across nodes, (ii) a cooperative concurrency model, meaning that the executing code decides when to yield, and (iii) dynamic variable name (re-)binding, affecting the incoming code's view over exposed interfaces. Absence of (i) and (ii) would mean that state update would require a transaction to safeguard against race conditions from competing nodes. Absence of (iii) would mean that third-party code would have unrestricted access to Andromeda's capabilities. Andromeda favors developer convenience over raw performance but, like Unix, without loss of generality nor unacceptable performance.

[2] Cloud Haskell [59] presents a simplified version in 48 lines of Haskell.

1.5 Dissertation Outline and Contributions

Employing distributed software today involves a significant degree of complexity. Following Brooks [30], part of this complexity is essential to translating an algorithm to an executable, but part of it is accidental and includes chores such as system setup, program loading, interfacing with the surrounding environment, and task orchestration. As many of these chores grow proportionally to the number of nodes involved, distribution amplifies accidental complexity.

Several of Andromeda's design decisions aim at minimizing accidental complexity (§2). Andromeda's overlay nature and lightweight kernel accelerate system setup and teardown, its pervasive use of configurability permits addressing distribution trade-offs, and its programming style simplifies development. For example, p_3 (§1.4) needs only a simple configuration parameter to achieve strong consistency, and p_4 circumvents the need for transactions by relying on support for function communication and cooperative concurrency.

Lowering essential complexity is only possible by avoiding parts of the algorithm-to-program translation. Andromeda aids this by allowing development and interaction using a conventional language, by providing access to an existing package ecosystem, and by wrapping distribution in an extensible module (§2). For example, p_2 (§1.4) looks identical on a conventional (non-distributed) userspace, and its use of the third-party `mdc` package is indistinguishable from Andromeda built-ins. Andromeda's approach adds the benefit of performance, by leveraging engineering effort already invested in the language runtime and associated libraries.

This dissertation's primary result is that a distributed userspace can significantly lower the complexity of employing distributed software—a key result for anyone interested in composing distributed software.

Secondarily, the dissertation should inform future designers of both distributed operating systems and distributed programming languages. To the former group, it shows that a language-system hybrid presented as a library can be implemented at a

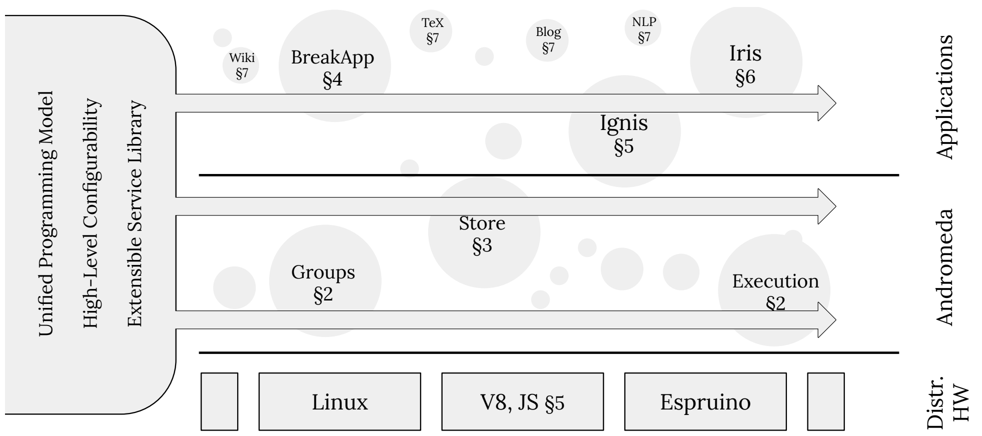


Figure 1.1: **Dissertation Outline**. The core of ANDROMEDA is described in Ch. 2 and 3; applications and frameworks on top of ANDROMEDA are described in Ch. 4–7.

fraction of the development cost required to build a distributed operating system—primarily because it leverages already-invested engineering effort. To the latter group, it shows that not many distribution-specific features need to be baked into the language: a small core with extensible semantics (ANDROMEDA uses a subset of ES5) can be more than enough—most other features can be added as extensions.

Perhaps unsurprisingly, using the complexity-lowering techniques presented in this dissertation may lead to security and performance issues. I address both classes of issues by developing a series of frameworks on top of ANDROMEDA; apart from addressing security and performance issues, these frameworks show the benefits of a general-purpose distributed environment outside the strict context of data-intensive computations and highlight the benefits of several design decisions.

Outline The dissertation describes the design and implementation of ANDROMEDA and its services (Fig. 1.1), starting with an overview of ANDROMEDA (§2): the ideas guiding its design, the distributed service library, and three representative services. Three ideas are worth mentioning now: (i) its approach of bolting distribution onto a conventional (and light) language runtime as an extensible service module, allowing its users to leverage an existing module ecosystem; (ii) its high-level programming style, melding a synergistic combination of elements—a dynamic typing discipline, a

Table 1.1: **Security and performance problems.** While solving complexity, some of ANDROMEDA's design decisions create security (S) and performance (P) challenges that are addressed in Chapters 3–6.

Design Decision	Complexity Benefits	Key Problem	Solution	Ch.
Dynamic Typing	Brevity, runtime rewiring	Indexing (P)	`store` [226]	3
Cooperative Scheduling	No race conditions, *etc.*	DoS *etc.* (S)	BREAKAPP [224]	4
-as-a-Library	Existing language and ecosystem	Interface attacks (S)	IRIS [227]	5
High-level Language	Brevity, safety,	Runtime overheads (P)	IGNIS [222]	6

cooperative concurrency model, and a continuation-passing call style; (iii) its pervasive support for high-level configurability, aimed at tackling a growing space of unavoidable distribution trade-offs. These ideas succeed at lowering complexity without sacrificing generality, but create several issues of their own (Tab.1.1).

A dynamic typing discipline contributes significantly to lowering complexity [158], but creates the problem of query-inefficient storage partitioning: efficient queries require *a priori* indexing, whereas dynamic types are known *a posteriori*—and can change during runtime. I address this challenge addressed by developing a series of generally-applicable techniques presented in Chapter 3's `store`, ANDROMEDA's underlying distributed storage subsystem.

Similarly, cooperative scheduling offers significant complexity benefits but suffers from the danger that third-party code may decide not to yield control. I address this problem in Chapter 4 by developing a general tool atop ANDROMEDA, BREAKAPP [223, 224], that sandboxes selected third-party modules with minimal developer effort. It does so by spawning third-party modules on remote nodes and maintaining their interfaces intact—forwarding calls to remote modules and intervening in cases of problems. My experience developing BREAKAPP was simplified by ANDROMEDA's service library, and its transformation and serialization primitives.

The notion of bolt-on distribution-as-a-library lowers complexity significantly, but due to its simplicity allows third-party modules to re-write library interfaces similarly similar to how ANDROMEDA does. BREAKAPP's sandboxing is of no help here, as it is too coarse-grained for individual service fields and too heavyweight for use as the common case. My solution, presented in Chapter 5, was to build a system atop ANDROMEDA, IRIS [227], that leverages language-based protection to offer finer-grained control and lower performance overheads. Interestingly, IRIS is powerful enough to enable distributed re-direction used in the next framework: code *thinks* it is calling, say, the distributed object storage when in fact it may be accessing a subset of the storage available on a specific node.

Although high-level, dynamic languages offer significant benefits in terms of complexity their performance overheads may be higher than those of carefully handwritten C or assembly code [47]. I tried to minimize this disparity by building on a high-performance runtime with state-of-the-art machinery (*e.g.*, tracing, optimizations, JIT compilation). This performance loss is insignificant compared to the benefits of productivity and safety, and can be partially recouped by re-engineering or scaling out bottlenecked components. To confirm the latter hypothesis, I developed IGNIS [222], presented in Chapter 6. IGNIS leverages ANDROMEDA's transformation infrastructure to detect and scale out bottlenecked modules in legacy applications, as long as their developers have sprinkled selective imports with soundness-related annotations. IGNIS also adds the last missing element in the quest to lower complexity, and allows porting larger (often legacy) applications on ANDROMEDA.[3]

Chapter 7 presents implementation details of a distributed-userspace prototype and associated frameworks targeting the JavaScript ecosystem. The goal of this chapter is to summarize lower-level technical challenges and how I addressed them. Both the challenges and solutions are not worthy of publication, but might still be useful to developers of similar systems.

Chapter 8 presents key results, showing that ANDROMEDA lowers the complexity

[3] The reader may be wondering about the differences between ANDROMEDA's and IGNIS' goals: While ANDROMEDA's users can use an existing language, their programs will not necessarily take advantage of distribution if they do not use ANDROMEDA's APIs. With IGNIS, however, their programs will leverage distribution even if they were developed in a completely distribution-oblivious fashion. At the same time, ANDROMEDA is much more powerful—programs that *do* use ANDROMEDA's APIs can see many more distribution benefits that what is possible with IGNIS.

of developing distributed applications. It offers short set-up, scale-out, and tear-down times, lowering the barriers to entry into distributed systems; it allows for small programs that execute atop a lightweight, high-performance runtime; and it enables program composition through an existing package ecosystem numbering over a million packages. ANDROMEDA is increasingly self-supporting and, as discussed earlier, has itself enabled novel research [224, 222, 227].

Finally, Chapter 9 compares with related work from several different fields, and Chapter 10 discusses experiences, limitations, and avenues for future research (§10).

Two Appendixes complement the main content of the dissertation. Appendix A presents a quick introduction to Andromeda, with two key objectives: (i) provide a quick guide on the JavaScript language—used pervasively in the dissertation—and (ii) demontrate the development of programs in ANDROMEDA. Appendix B outlines how module systems handle modules at runtime—by discussing the implementation of the Node.js module system; the goal here is to ensure that the techniques presented Chapters 4–6 are appreciated by a wider audience.

Chapter 2

An Overview of Andromeda

This chapter presents an overview of Andromeda and its key services. It starts with the principles behind Andromeda's design (§2.1). It then discusses three representative subsystems; while most of Andromeda's subsystems are "textbook" implementations, easily replaceable by more sophisticated versions, these three are novel and illustrate contributions applicable beyond Andromeda (§2.2–2.4):

- the storage subsystem (§2.2), which supports efficient-query partitioning on dynamic data and illustrates a use of high-level, fine-grained configurability;
- the node group management subsystem (§2.3), which enables first-class support for network overlays and exemplifies cross-scale concerns; and
- the task execution subsystem (§2.4), which supports automated runtime transformations and interface rebinding.

2.1 High-level Design Principles

This section overviews the high-level design principles underpinning Andromeda: (i) its distribution-as-a-module approach (§2.1.1) along with the collection of built-in

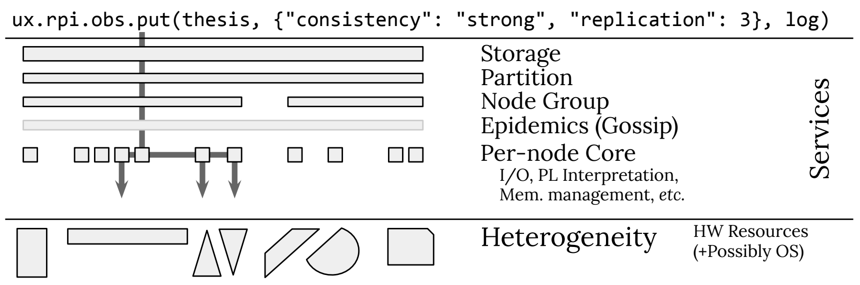


Figure 2.1: **Andromeda overview.** Andromeda bolts distribution on an existing language and runtime by providing a configurable service architecture, an extensible service library, and an associated programming model (*Cf.*§2).

services (§2.1.2), (ii) its pervasive support for high-level configurability (§2.1.3), and (iii) its programming style for service composition (§2.1.4).

2.1.1 Core: Distribution as a Software Module

Andromeda is accessible as a software module, implantable into (often, pre-existing) programs using an import statement similar to `require("andromeda")`. Upon import, Andromeda (i) launches one or more nodes, (ii) loads and binds distributed services, (iii) transforms local interfaces—*e.g.*, the global-variable map *etc.*—to detect changes, and (iv) returns the `andromeda` object (§1.4), which is used to access services running on nodes.

Launching nodes can be significantly slower than loading a module. As such, the aforementioned series of events occurs in an asynchronous, non-blocking fashion; a call to the user code is placed in an event handler that is invoked when the startup process completes. If nodes are already up and running, the `andromeda` module simply attaches onto the local node. If the `andromeda` module is called standalone (top-level), it additionally launches the interactive shell, which supports saving programs and its context to the distributed storage subsystem and loading them at a later time (or remote node).

On each node, Andromeda maintains service bindings, handles communication with other nodes, and executes incoming code. Its core is a routing function: given an incoming message as an argument, it routes that message to the service responsible for handling it. Messages can arrive from services running on the same node, or over the network from different nodes. A message can encode any language value, including primitives, functions, and objects. Andromeda's lower-level internals also recognize types that do not have a direct representation in the source language, such as object streams, raw-data messages, and raw-data streams.

Messages that encode functions are particularly important, because they provide the necessary means for shipping services among nodes. This allows extending Andromeda at runtime, by registering additional entries in the routing core. Built-in services are also loaded and instantiated dynamically, often by pulling their functionality from peer nodes.

Andromeda's core provides only the essential infrastructure for building an extensible service architecture. It is used to create and bundle a small library of built-in services, which in turn supports the critical functions of the system proper and its applications. Services can be configured, augmented, or replaced at runtime from within the system. The result is a lightweight kernel (<1MB) that offloads assumptions about the nature and scale of distribution to individual services.

2.1.2 A Library of Built-in Services

Andromeda's utility comes from services, objects that encapsulate control-level internal state and expose a small set (typically 4–6) of methods. The service abstraction is a flexible one, intended to capture the multifaceted concerns of distribution while facilitating structured, modular program composition.

To simplify composition, routing, and replacement, services conform to a common interface (§2.1.4) and are available within a flat namespace. Both built-in and add-on services depend hierarchically on other services to provide supporting functionality. For example, `store` depends on `partition`, which in turn depends on `nodes`, itself (loosely) depending on a service implementing an gossip protocol (Fig. 2.3).

Table 2.1: **Selected built-in services.** Example calls take a continuation function as additional argument (omitted) that will be called once results become available (*Cf.*§2.1.2).

	Service	Use	Example Call	Representative Options
State	mem	Distributed shared memory	`get({an:"obj"}, {key: "0c3f"}, l)`	key, replication, consistency
	kv	Persistent key-value storage	`put(obj, {replication: 3})`	— ” —, cache
	store	Persistent multi-dimensional storage	`search({dimensions: ["id"], val: "1"})`	— ” —, dimensions, keys
Nodes	nodes	Node information and management	`spawn({number: 3, peer: local.node})`	halt, peer, number, version
	groups	Node group info. and management	`create({name: "primegen", inherit: []})`	name, services, inherit
	service	Service and instantiation infrastructure	`wrap({methods: ["get", "put"]})`	query, constructor, methods
Execution	task	Task management	`exec(pow, {nodes: "first", args: args})`	nodes, replicas, completion
	sandbox	Software-based isolation primitives	`run(f, {id: "s1", ctx: {log: false}})`	globals, context, maxTime
	modules	Module fetch/share	`get("crawler", {source: "local"})`	interface, source, repo
Comm/on	message	Scalable communication primitives	`send({a: "msg"}, {order: false})`	order, consistency, path,
	request	Request-oriented communication	`head("http://up.ar.net", null)`	path, domain, type, stream,
	routes	Bidirectional mapping of names to services	`put(add, {path: "/add"})`	path, access-control
Support	package	Distributed package management	`get("mdc", {caps: {fs: "fs", net: null}})`	selection, top, type
	info	Information on underlying infrastructure	`get({keys: /cpu/i, summary: true})`	summary, keys, values
	events	Interrupt event bus	`on("peer-down", (v) =>{v.f()})`	(no callback)

Tab. 2.1 presents a subset of the services bundled with ANDROMEDA. The first three groups are responsible for storage and query, node and group management, and task execution (§2.2–2.4). The next group of services provides communication abstractions with configurable semantics (*e.g.*, ordered multicast) and allows applications to register custom routing paths for message delivery. Support and utility services complete the standard library—*e.g.*, an event bus provides support for asynchronous publish-subscribe, and a package manager provides support for module fetching and loading.

2.1.3 Pervasive, High-Level Configurability

To address inescapable trade-offs of distribution [71, 3, 120], ANDROMEDA supports (re-)configurability. In most cases, this ability is expressed via high-level declarative properties rather than complex low-level program fragments intertwined with system logic. One example is service specialization, in which control-level configuration affects key service-internal logic—such as `store`' indexing, replication, and consistency guarantees (§2.2). This ability is available at runtime and at very a fine granularity—that of individual calls or messages—as illustrated by program p_3 (§1.4).

Although configuration knobs are declarative and high-level, their sheer number can easily overwhelm ANDROMEDA's users. This problem is alleviated using a series of automation techniques that include node group (overlay) facilities, automated runtime transformations, and startup configurations.

Node groups enable stateful service customization, allowing services to make their default configuration context available for customization. Prior to loading, services start as templates that expose a number of control-plane configuration "holes". When loaded to a node group, their context is instantiated to a default configuration, binding context entries to instance-specific values. For example, `store`' replication may be configured differently between the `rpi` and `dsl` groups. After a service has been specialized, its default configuration can still be overridden at the level of individual calls.

Runtime transformations, employed pervasively by ANDROMEDA, use reflection to traverse and rewrite composite values such as objects or interfaces. For example, they are used for serializing language values into strings, for spawning remote services while maintaining the local interface intact, and for sandboxing third-party modules.

ANDROMEDA's startup configuration (`usconf`, similar to `.shrc`) is by default shared across all nodes. This behavior can be overridden by the `andromeda` import statement, command-line arguments, or a local startup object (saved on disk). Rather than constraining expressions to data-only objects, `usconf` supports function evaluation. As ANDROMEDA emits various events while configuring the kernel and binding services, user code listens for these events and registers handlers that launch associated tasks. One example is the launch of additional nodes when the primary completes its startup sequence, defaulting to as many nodes as processors.

The three techniques outlined are themselves highly configurable. For example, transformations are runtime-configurable to address semantics-related concerns—*e.g.*, permissions in sandboxed modules or soundness in distributed ones. Although more malleable than conventional approaches that bake the configuration into the implementation, such high-level knobs are not expected to cover all possible scenarios. For these cases, ANDROMEDA provides support for programmatically augmenting or replacing entire subsystems (§2.3).

2.1.4 Programming Style

ANDROMEDA is a language-based environment whose programming style melds an unusual combination of aspects: high-level semantics, dynamic typing, cooperative concurrency, the use of continuations, and location independence. While individual features are not novel *per se*, their combination succeeds in lowering the complexity of composing distributed programs without any loss of generality. For example, p_4 (§1.4) illustrates the joint effects of cooperative concurrency with first-class support for function distribution; their absence would require transactions, to safeguard state updates against races from competing nodes.

A high-level language addresses the impedance mismatch between the low-level semantics and high-level challenges of distribution—too high-level for systems concerned with tasks such as memory allocation and device driving. Compared to a UNIX-like system, files are replaced by objects, processes and process images by functions and function application, object linking by variable binding, system calls by library calls, and program installation by package fetching and loading.

Dynamic semantics, an unusual feature, offers the ability to traverse, discover, and transform interfaces at runtime. The fact that type information is associated to values rather than names allows names to be re-bound to values of different types. This aspect enables several of ANDROMEDA's features and is critical to its (re-)configuration abilities (§2.1.3). These techniques are available during execution—when the need to respond to changes is critical—by providing runtime reflection and introspection. Built-in runtime code evaluation is used in interactive scripting, in module loading, and especially in distribution of code such as functions and objects.

Cooperative concurrency was chosen primarily to lower complexity, and only secondarily to improve performance. Complexity is lowered by relieving programmers from race conditions, locking, and state inconsistencies; their code is preempted only when it chooses to do so. As a convenient side-effect, performance is improved by avoiding costs associated with scheduling and context switching. Along with cooperative concurrency, continuations allow developers to be explicit about parallelism. A continuation is guaranteed to be called only after its caller completes, while independent continuations do not impose any ordering constraints—two continuations κ_1 and κ_2 of sequenced calls $f_1(\ldots, \kappa_1)$; $f_2(\ldots, \kappa_2)$ can be interleaved in any order (otherwise, f_2 would have been included in κ_1).

Location and platform independence lower complexity at scale, by exposing a single unified set of abstractions across a (potentially heterogenous) distributed infrastructure. All nodes have access to the same interfaces, leading to code that looks identical irrespective of node structure—except when developers explicitly intend otherwise. Invoking, say, `rpi.store` from *any* node will resolve to the same node/service, except if, say, a third-party module is restricted to local-only access. Moreover, there is no distinction between calling a service and sending a message: argument evaluation strategy is strict (eager) and call-by-value—given an argument, ANDROMEDA will evaluate it, serialize it, and send it to a remote service.[1] Service methods conform to a unified interface type:

```
Op :: Maybe Value -> Maybe Options ->                                  (p5)
      Maybe (Error -> [Value] -> ()) -> ()
```

Arguments express (i) state (any language value), (ii) optional configuration properties (§2.1.3), and (iii) a continuation to be called when the operation completes. To guard against cases where the return value is of type `Error`, the continuation distinguishes the two arguments provided by the service: an `Error` and, if `null` `Error`, a list of results.

2.2 Object Storage and Query

The distributed storage service, `store`, is used extensively to store and query both user and system data. Given its central role in the system, `store` must solve several challenges: (i) be flexible, providing tunable guarantees for replication, consistency, and indexing; (ii) support indexing of dynamic values (§2.1.4), whose structure is not known beforehand; (iii) provide querying efficiency queries, by avoiding naively querying all of ANDROMEDA nodes for a single object.

To solve these, `store` builds on Hyperspace Hashing [62]. This baseline is further augmented with *formulas* to capture user insight, improve performance, and provide tunable replication, consistency, and indexing guarantees. We give a brief summary

[1] Co-located services skip the last two stages, passing pointers instead.

of `store`, focusing on how it interfaces with the rest of ANDROMEDA, and present the technical details in the next chapter (§3).

2.2.1 Internal Structure

ANDROMEDA's scheme permits efficient queries without fixed schemas by hashing object property names and values. We illustrate this with an example:

```
ep = {id: "EAP09", fst: "Eddy", lst: "Poe"}                    (p6)
store.put(ep, λ)
```

By default, `store` treats secondary properties (*e.g.*, `fst` and `lst`) the same as primary ones (*e.g.*, `id`). Storage operations are parameterized by D and r: D dimensions with r regions per dimension. For the example, assume $D = 10$ and $r = 3$, with hash and modulo functions $h()$ and %. Operation `put(ep)` first hashes the object's keys, calculating h(`"id"`)%10, h(`"fst"`)%10, and h(`"lst"`)%10, all integers in 0–9, inclusive. It then calculates h(`"EAP09"`)%3, h(`"Eddy"`)%3, and h(`"Poe"`)%3, all integers in 0–2, inclusive. If the results are $(3, 4, 8)$ and (1, 1, 2), the final coordinate vector is:

```
[0, 0, 0, 1, 1, 0, 0, 0, 2, 0]
```

This vector can be used to greatly narrow the search space. Operation `get({id: *, fst: "Eddy", lst: *})` searches for all objects with an attribute name of `fst` whose value is ”Eddy” and any `id` and `lst` properties. The resulting vector requires looking only into *nine* out of 59,049 regions:

```
[0, 0, 0, {0-2}, 1, 0, 0, 0, {0-2}, 0]
```

Querying for similar objects that either do not have an `id`, or have an `id` of “ep”, the respective operations are `get({fst: "Eddy", lst: *})` and `get({id: "ep", fst: "Eddy", lst: *})`. Assuming h(`"ep"`)%3 is 2, they result in vectors:

```
[0, 0, 0, 2, 1, 0, 0, 0, {0-2}, 0]
[0, 0, 0, 0, 1, 0, 0, 0, {0-2}, 0]
```

Each one will hit only three different regions.

2.2.2 Interface Considerations

Without user input, Andromeda's hashing scheme supports efficient queries and dynamic objects, but has several issues. First, it severely penalizes properties that can *uniquely* identify an object. While a query by primary key like `id` should hit only a *single* node, it instead reduces the search space by a mere order of magnitude. Second, `store` by default devotes resources to support efficient queries on *all* object properties; being more selective would improve performance. Third, by assigning equal priority to all properties, it does not allow favoring some queries over others—for example, looking up people by last name is more common than by eye color.

These challenges require user insight, and are addressed by a second argument for each operation (*e.g.*, `put`, `get`) termed a *formula*—a subtype of `Option` (§2.1.4). Formulas are configuration objects that allow developers to provide guidelines at the level of individual operations.

Formulas provide a mechanism for the user to enforce policies related to classic distributed systems concerns such as consistency (strong vs eventual), replication (majority vs n nodes), and quorum selection. Formulas also allow tuning the storage mechanism itself, for example by devoting more resources to indexing particular object properties.

While formulas give the user a great deal of power, they also introduce significant inconvenience. The reason is that once used to insert an object, the same formula must be used for all subsequent queries, updates, and deletions of that object, even across nodes. To solve these challenges, Andromeda exploits a key insight: formulas are *themselves* dynamic data objects. Thus, the techniques and operations already described for normal objects can be used to store and retrieve formulas efficiently.

To simplify their management, formula objects are augmented with an identifier (ID) property, similar in spirit to a distributed pointer. IDs are unique and either provided by the user or deterministically[2] generated by `store`. Normal operations are overloaded to also accept the ID in place of a formula argument:

```
var cf = {id: "cf1", spaces: {...}}; put(cf);                    (p7)
put(obj, cf); put(obj, "cf1");
```

The two `put` operations are semantically equivalent.

2.3 Node Group Management

Although significantly configurable, a single service instance cannot operate concurrently at multiple scales [61]. This limitation is addressed by the node group abstraction, which provides first-class support for context-aware network overlays and allows overlapping instances to coexist simultaneously.

The key insight behind groups is the addition of a level of indirection between nodes and services running on these nodes. Leveraging this insight, ANDROMEDA can (i) refer to a set of nodes using only a single, memorable name, (ii) capture semantic differences in multiple (potentially overlapping) node groups, (iii) specialize services, by binding service-context information at the group level.

The group abstraction occupies a somewhat central point in ANDROMEDA, as it prefixes all accesses to nodes (and, subsequently, services). Such prefixing makes ANDROMEDA *distributed-first*: handling distributed programming with multiple nodes is the common case, but it is general enough to support centralized programs—by defining singleton groups (p_4, §1.4). By declaring a common node group as the default one, operations on a pre-configured set services can be prefix-free (p_2, §1.4).

2.3.1 Nodes and Node Groups

Nodes are an abstraction of virtual or physical computing devices of varying capabilities (each with its own storage, processing *etc.*)—*e.g.*, a UNIX userspace process or

[2] Formulas semantically equivalent map to the same ID.

a single-board micro-controller. Each node binds on a new $(IP, port)$ pair used as a communication handle, and is addressable by a cryptographic identifier (NID). Users can ask ANDROMEDA to generate a new NID, but cannot explicitly name the node themselves. This is because several system invariants depend on the expectation that NIDs are unique (and, ideally, long-lasting).

A node group, at its core, combines two parts into a single entity addressable by a user-generated group ID (GID): (i) a collection of nodes, and (ii) a collection of services running on these nodes. Three GIDs are built into ANDROMEDA: (i) `global`, addressing all nodes, (ii) `local`, addressing nodes running on the same host, and (iii) `self`, addressing the current ANDROMEDA node. Addressing multiple nodes is expected to be the common pattern, with the exception of `self`.

The collection of nodes is arranged into a set. Node sets are *not* disjoint between node groups, in the sense that a node can exist in and be addressable from multiple groups. Most groups will refer to a fixed set of nodes, but there are exceptions to this rule. For example, the `self` group always depends on the execution context. The mapping of GIDs to nodes is purely a node-local structure, allowing different nodes to see different sets for the same GID, which enables encoding other overlay types beyond cliques.

2.3.2 Service Instantiation and Loading

A group's services are arranged in a map from a service ID (SID) to a service instance, which is the result of an instantiation process. This process specializes services and employs runtime transformations (§2.4.1). Newly instantiated services are provided a context, which maps variable names to values that serve as defaults during the execution of the service. Similar to the collection of nodes, the service map may contain the same service addressable under different SIDs (with potentially different configurations).

Services inherit functionality from an abstract service template, part of the `service` service (Tab. 2.1). Aside from low-level infrastructure for serialization and communication, the template provides a structure for organizing configuration parameters so that they can be discovered at runtime. Service implementers declare a list of configurable variables along with their default values. Service templates are kept around for further instantiation, but can be saved on persistent storage if they are not needed during runtime. Prior to loading, just-in-time static analysis [171] is used to ensure that the service interface and configuration parameters are in the expected form.

2.3.3 Group Management

The `group` service provides several primitives for manipulating groups, such as creation and rebinding.

Group creation is quite flexible. It is achieved by providing either an explicit list of nodes or by mixing one or more existing node groups. In the first case, a system-specified set of services is by default added to S, which users can amend by passing SID-to-service tuples. In the second case, services are inherited as the union of all the services by the groups specified. For usability purposes, the specified order of services has some significance: earlier method names are replaces by later ones.

Replacing a group service can be achieved in two ways:

```
eandromeda.dsl.store = fs                                          (p8)
eandromeda.dsl.service.setup(store, "andromeda/dsl/fs", κ)
```

When updated by property assignment on the group object, replacement defaults to eventual consistency. This choice is related to the fact that service update—like any distributed operation—is non-blocking, but the assignment interposition wrapper is blocking. When updated through the node group API, typical continuation-passing rules apply (§2.1.4): code in the continuation will execute only after the operation succeeds (or throw an error), whereas code right after the call will execute immediately, similar to assignment.

```
txfm (e: Field) : Field := match e with
  | {(s, v) :: vs} → {(s, txfm v) :: txfm vs}
  | [v :: vs] → [(txfm v) :: txfm xs]
  | λ.f → (...args).{v ← f(args) // lng(this, args, v)}
  | __ → interpose(copy(e))
end
```

Figure 2.2: **Example runtime transformation for lineage.** The transformation (simplified) is presented in functional style to ease variable binding; types, whose structure is used for pattern matching, are shown in light *turquoise* The `//` operation implies non-blocking call; internally, `lng` only records the operation$v$ (*i.e.*, *this*) (*Cf.*§2.4.1).

2.4 Task Execution

When a function or service gets shipped to a remote node some of its names have to be bound to values local to the remote node—at the very least, the `ux` object (p_4, §1.4). Even node group features such as the ability to use prefix-free services, discover service configurations, and replace groups require the ability to walk, transform, and rebind interfaces at runtime. These (and other common) challenges are addressed using a combination of runtime transformations (§2.4.1) and context rebinding (§2.4.2).

2.4.1 Runtime Transformations

Transformations are used pervasively throughout Andromeda, and are abstracted by a parametrizable template. The template maps different types of values (*e.g.*, `Numbers`, `Functions`) to a generic handler for each type. Transformations have these handlers parametrized to achieve concrete goals such as generating remote procedure call (RPC) stubs, serialized string values, security interposition wrappers *etc.*

Transformations can be applied to any value in the language. The most general value is an object, a bag of records holding key-value fields; other value types include primitives, functions, and booleans. Transformations walk objects from their root, processing component values based on their types (Fig. 2.2). More specifically, (i)

Table 2.2: **Example automated transformations.** Transformations are used by ANDROMEDA to rewrite (or re-wire) interfaces and other values at run-time (*Cf.*§2.4.1).

Transformation	Explanation	Example Parameters
Serialization	Convert values to strings—*e.g.*, to send to disk / network	boxed values, class hierarchy
Beautification	Pretty-print values; add control characters for coloring	color scheme, control set
Profiling	Wrap functions with ones recording profiling information	call frequency, queue size
Protection	Interpose on context accesses for environment modifications	white-listing, mocking
RPC generation	Generate RPC stubs redirecting accesses to original objects	remote location, types
Scale-out	Generate thin clients scheduling calls across replicas	min-max replicas, order
Sandboxing	Introduce permission contracts on module boundaries	`RWX`-perms, filters
Lineage	Capture data derivation by recording function application	check-pointing, versioning

function values are wrapped by closures specific to the goal of the transformation; (ii) *object* values are recursively transformed, with their getter and setter methods replaced similar to function values; (iii) *primitive* values are either transformed directly or copied unmodified and wrapped with an access interposition mechanism.

For example, consider transforming a `Math` object with the goal of profiling its interfaces or generating a remote interface (Fig. 2.3*a,b*). ANDROMEDA traverses the object returned by `Math` and replaces functions such as `div` with wrappers. The wrappers depend on the intended goal: profiling wrappers would record invocation statistics, and RPC wrappers would forward calls to a remote replica.

For brevity, we omit several technical details that are specific to individual transformations (and are often achieved with multiple additional transformation passes). These details are discussed in Chapters 4–6. Among other primitives, ANDROMEDA provides ones for converting local-memory pointers to meaningful distributed ones, forwarding side-effects such as memory allocation and collection, providing distributed versions of core built-in libraries, and enforcing event ordering (when required). For further processing, ANDROMEDA often maintains a handler to the root of both the unprocessed and the newly processed values. The unprocessed value is used for further transformations, and the new value is often used to partially reverse or alter the result of a transformation (*e.g.*, to revoke access permissions). Tab. 2.2 summarizes more transformations applied in a similar fashion.

```
Math = {
  …
  mul: (a, b) => a * b,
  div: (a, b) => {
    log.info.(b);
    return a / b,
  }
  …
}
```

(a) a `Math` object

```
let _ = Math;
Math = {};
  …
Math.div = (…args) => {
  let p = prologue(args);
  let v = p ? _.div(args) : p;
  return epilogue(p, args, v);
}
…
```

(b) Object transformation

```
var ctx = {
 fs: ux.local.fs,
 log: ux.tf(ux.p3.log,
            myPrlg,
            myEplg),
 …
}
```

(c) Custom context creation

```
function (cxt) {
  var fs = cxt.fs;
  var log = ctx.log;
   …
  div: (a, b) => {
    log.info.(b);
    return a / b,
  }
}
```

(d) Context rebinding

Figure 2.3: **Example transformations and rebinding.** Applying runtime transformations (b) and context rebinding (c, d) on a simple `Math` object (*Cf.*§2.4).

2.4.2 Runtime Context (Re-)Binding

Aside from the correctness of communicated values (§2.4, start), names are often rebound for security or performance reasons. For example, interposing on a third-party logger's `store` access can ensure both that it does not alter sensitive data and that it avoids inefficient API calls. To achieve these goals, ANDROMEDA provides the ability to transform and rebind the context of a value at runtime (Fig. 2.3*c,d*). This responsibility is divided between two phases: context creation and linking.

Context Creation First, ANDROMEDA needs to prepare a new context to bind to the current value. It first creates an auxiliary hash table (Fig. 2.2*c*), mapping names to (new) values: (i) names correspond to user-declared variables or language built-ins—including globals, module-locals *etc.* (ii) values are provided either verbatim (*e.g.*, primitives) or by transforming (§2.4.1) values in the original context. For example, transformed values can point to different nodes (Tab. 2.2:*5*) or control access to the original value (Tab. 2.2:*4*).

User-defined global variables are stored in a well-known location (*i.e.*, a map accessible through a global variable named `global`). However, traversing the global scope for built-in objects is generally not possible. To solve this problem, Andromeda collects such values by resolving well-known names hard-coded in a list. (Different lists will be needed for different environments and versions of the language, but we haven't faced this problem yet.) Using this list, Andromeda creates a list of pointers to unmodified values upon startup.

Care must be taken with module-local variable names of third-party modules. Examples include the module's absolute filename, its `export`ed values, and whether the module is invoked as the application's `main` module; each module refers to its own copy of these variables. Attempting to access them directly from within Andromeda's scope will fail subtly, as they will end up resolving to module-local values of Andromeda *itself*—and specifically, the module within Andromeda applying the transformation. Andromeda solves this issue deferring these transformations for the linking phase (*i.e.*, from within the module).

Context Linking Andromeda needs to link the code whose context is being transformed with the freshly created context.

To achieve this, it leverages lexical scoping to inject a non-bypassable step in the variable name resolution process. The technique works because the code is still at a loading stage prior to interpretation, and still represented as a string—*e.g.*, code that arrived from the network, service templates loaded from disk, modules fetched remotely, or strings typed in the interactive shell. Wrapping and then evaluating such code ensures that their variables will point to the transformed context.

Specifically, Andromeda first applies a form of source-to-source code rewriting. The rewrite wraps the code with a closure that starts by redefining and enclosing variable names corresponding to modified values as local ones (Fig. 2.2*d*). The closure accepts as an argument the customized context and assigns its entries to their respective variable names. This is arranged in a preamble comprised of assignments, which executes before the original code is called. Module-local variables (a challenge outlined earlier) are assigned the transformation call, which will be applied only when code is evaluated. Finally, Andromeda evaluates the resulting closure, invokes it with the custom context as an argument, and possibly applies further transformations to its return value.

2.5 Summary

This chapter presented an overview of Andromeda and its key services. It started with the principles behind Andromeda's design (§2.1). It then discussed three representative subsystems; while most of Andromeda's subsystems are "textbook" implementations, easily replaceable by more sophisticated versions, these three are novel and illustrate contributions applicable beyond Andromeda (§2.2–2.4):

- the storage subsystem (§2.2), which supports efficient-query partitioning on dynamic data and illustrates a use of high-level, fine-grained configurability;
- the node group management subsystem (§2.3), which enables first-class support for network overlays and exemplifies cross-scale concerns; and
- the task execution subsystem (§2.4), which supports automated runtime transformations and interface rebinding.

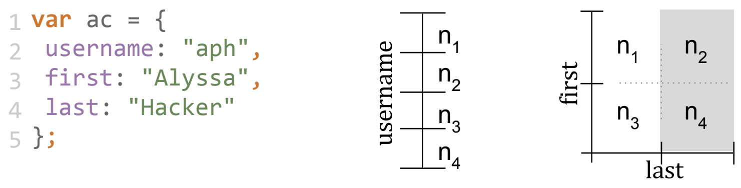
The next chapter 3 dives into the details of the storage subsystem and, specifically, its partitioning scheme.

Chapter 3

Query-efficient Partitioning for Dynamic Data

Given its central role in the system, `store` must address several challenges: (i) be flexible, providing tunable guarantees for replication, consistency, and indexing; (ii) support indexing of dynamic values (§2.1.4), whose structure is not known beforehand; (iii) provide querying efficiency, by avoiding naively querying all of ANDROMEDA nodes for a single object. The combination of (ii) and (iii) is critical, because ANDROMEDA's programming style follows a dynamic typing discipline. As the structure of data is not known beforehand, nor remains static, ANDROMEDA needs to support efficient queries on data stored without static indices. "Efficient" means avoiding the naive solution of flooding all nodes in a group. This efficiency should ideally extend to structural queries returning all objects matching a certain structure; structural queries depend on efficient union, negation, and intersection operations, which in the practice of dynamic languages are often referred to as "duck typing". These needs make `store` the canonical example of a service design that is configurable at both a high level and fine granularity (§2.1.3).

To address these issues, we develop a new set of techniques that together enable query-efficient partitioning of *dynamic* data. First, unispace hashing (UH) extends



```
var ac = {
 username: "aph",
 first: "Alyssa",
 last: "Hacker"
};
```

Figure 3.1: **Partitioning Example.** Left: a dynamic object with three properties. Middle: object placement by `username`. Right: object placement by `first–last`(right) (*Cf.*§3.2).

multidimensional hashing to data of unknown structure with the goal of improving queries on secondary properties. Second, compression formulas leverage user insight to address UH's inefficiencies and further accelerate lookups by certain properties. Third, formula spaces use UH to simplify compression formulas and accelerate queries on the structure of objects. The resulting system supports dynamic data similar to single-dimension NoSQL systems, efficient data queries on secondary properties, and novel intersection, union, and negation queries on the structure of dynamic data.

3.1 Broader Motivation

Scalability requirements during the last decade have led to the development of distributed, non-relational databases (NoSQL). Single-dimension NoSQL [51, 184, 107] divides data into partitions over the dimension of a "key" property whose values are unique for each object (Fig. 3.1 middle). Since the partitioning scheme depends only on a single property, the structure of the rest of the object (*i.e.*, its "secondary" properties) does not need to be known *a priori* nor does it need to remain fixed. Data can be dynamic and have their structure change during the program's runtime. This flexibility worked well with dynamic programming languages (*e.g.*, Ruby, Python, JavaScript, PHP) and interchange formats (*e.g.*, XML, JSON) popular in application development. However, an inability to exploit structure means that queries on properties other than the primary key become inefficient, as all partitions must be searched.

Multidimensional key-value stores, as pioneered by Hyperdex [62], attempt to remedy this problem by partitioning on multiple dimensions (Fig. 3.1 (c)). To create such a hyperspace, however, the system depends heavily on structure: it requires *a priori* knowledge of the structure of objects, it does not support changes to the object's properties, and needs to maintain a mapping from regions of a property's values to underlying nodes on the side.

The goal of our work is to enable efficient partitioning and querying of *dynamic data* using three techniques. *Unispace hashing* is a generalization of hyperspace hashing [62] that uses property names to identify which dimensions an object represents. This enables support for dynamic data and accelerated queries on secondary properties, but does not make ideal use of the available space of dimensions (§3.3). Therefore, *compression formulas* can be used to tune space use by configuring queryable dimensions at the granularity of individual objects. Formulas bring many benefits (§3.4), but their use needs to be consistent between all operations targeting a specific object. To alleviate this potential for inconsistencies, the system employs *formula spaces* (§3.5): it takes advantage of the fact that formulas are themselves dynamic objects to store and query them, adding a layer of indirection between their description and their use. This additional layer can be used to accelerate queries on the structure of dynamic objects (*e.g.*, unions, intersections). The resulting hybrid aims to support dynamic data similar to single-dimension NoSQL, efficient data queries on secondary properties similar to multi-dimension NoSQL, and novel queries on the structure of stored data (§8).

3.2 Background

Consider four nodes with ids n_1 to n_4; a function $H(s)$ that maps strings to nodes n_i; and an object `ac` that we want to store to one of our nodes. For now, we can think of $H(s) = h(s)\%4$, where h is a hash function. Objects are sets of properties:

each property is a pair of a property *name* and a property *value*. In Fig. 3.1, the `ac` object has three properties: `username`, `first`, and `last`.

One of these properties takes values that are – or can be made – unique across all objects (*e.g.*, `username`). This property is often termed "key" in the distributed key-value store literature and is used to partition the data on a single dimension (Fig. 3.1 mid). Assuming the same nodes and "key", operations by "key" require contacting a single node, namely $H($`ac.username`$)$. The result is independent of the node receiving the request, independent of the property names and overall structure of the object, and is achieved without maintaining any indices or side-structures. Unfortunately, however, searching by other properties (*e.g.*, `first`, `last`, or both) requires contacting *every* node.

Hyperspace hashing [62] is a generalization of the previous idea to multiple dimensions. Assuming `first` and `last` are enough to uniquely identify an object, it partitions the two-dimensional plane into the four nodes n_1 to n_4 (Fig. 3.1 right). Insertion and retrieval require contacting the node at coordinates $(H($`ac.first`$), H($`ac.last`$))$. Retrieval by partially-specified queries on secondary attributes is still more efficient than exhaustive search: to return all "Hacker"s, the system needs to contact only half of the nodes (shaded area). The system successfully solves queries on secondary attributes, but requires *a priori* knowledge of object structure, disallows changes to the number, names, and types of its properties, and maintains an explicit, centralized mapping from dimensions to nodes. Moreover, since partitioning is determined statically, changes in the number of available nodes may render the partitioning scheme void.[1]

Based on the previous discussion, our scheme has to solve three main challenges (Table 3.1): (i) handle objects whose structure is not known beforehand, (ii) provide efficient queries on "secondary" properties, and (iii) remove the need of a mapping

[1] This is different from fault tolerance: there might not be enough nodes to even partition the data! In our example, if node n_3 did not exist, the scheme collapses because there are not enough servers to support the required dimensions.

Table 3.1: **Summary of features.** Techniques: (a) single-dimension NoSQL (1D), (b) multi-dimensional (or Hyperspace Hashing) (HH), and (c) Unispace Hashing (UH) (*Cf.*§3.2).

	1D	HH	UH
Dynamic Object Structure	✔	✘	✔
Efficient Search on "Secondary" Properties	✘	✔	✔
No Dimension-to-Node Mapping	✔	✘	✔
No Bounds on Number of Nodes	✔	✘	✔
Queries on Structure (*e.g.*, Union, Intersection)	✘	✘	✔

from dimensions to nodes. A solution should not pose any requirements on the number of nodes (*e.g.*, work on a single node) to ensure use in any environment. Finally, since all objects are dynamic, it should offer efficient queries on their structure (*e.g.*, return all objects with a property name "model").

3.3 Unispace Hashing

The core technique is an extension to hyperspace hashing. To allow querying, each object is represented as a point in a multi-dimensional space. As with hyperspace hashing, the coordinate for each dimension is determined by hashing the object's property values. Unlike hyperspace hashing, dimensions are determined by hashing the object's property *names*.

All operations draw deterministically from a set of dimensions D with size $|D|$. For now, we assume a fixed number r of regions (nodes) per dimension. In single-dimensional systems r can be thought as the number of nodes in the cluster. We will later use r to assign multiple regions per physical server as a way to "even out" differences in the server's relative capabilities. Hashing the name of each property returns an integer from 0 to $|D| - 1$. Using this number to index in D returns a dimension D_i. Hashing the value of the property corresponding to this name returns a value from 0 to $r - 1$. This is the coordinate value for dimension D_i. Coordinate values for dimensions corresponding to property names that are not present get a default value of 0. Coordinate values for dimensions whose property values are unknown get the full range of values in r.

Insertions and updates require fully-specified objects. That is, the value of each property needs to be present in order to determine the location of the object. Queries and deletions fill unknown coordinates with wildcards: they will need to search all regions that fall under the values of a specific dimension.

To illustrate insertion and query, we will be using the `ac` object from Fig. 3.1, an r of three regions per dimension, and a 10-dimensional D. The `put(ac)` operation inserts `ac` into the database. It first calculates h(`"username"`)%10, h(`"first"`)%10, and h(`"last"`)%10, all integers in the range from 0 to 9, inclusive. It then calculates h(`"aph"`)%3, h(`"Alyssa"`)%3, and h(`"Hacker"`)%3, all integers in the range from 0 to 2, inclusive. Suppose the first set of results is 3, 4, and 8 respectively; and the second is 1, 1, 2. The coordinate vector is the following:

```
[0, 0, 0, 1, 1, 0, 0, 0, 2, 0]
```

The `get({username: ANY, first: "Alyssa", last: ANY})` operation looks for all objects with an attribute of name `first` whose value is "Alyssa" and any `username` and `last` property. The resulting coordinate vector requires looking into all regions with coordinates:

```
[0, 0, 0, {0-2}, 1, 0, 0, 0, {0-2}, 0]
```

It will only search within nine out of 59,049 regions. If we want to look for similar objects that either do not have a username or have a username of "aph", the respective operations are `get({first: "Alyssa", last: ANY})` and `get({username: "aph", first: "Alyssa", last: ANY})` resulting in the following coordinate vectors:[2]

```
[0, 0, 0, 2, 1, 0, 0, 0, {0-2}, 0]
[0, 0, 0, 0, 1, 0, 0, 0, {0-2}, 0]
```

[2] Suppose h(`"aph"`)%3 results in 2.

Each one of them will only hit three different regions.

It is important to note that this scheme works identically with any number of physical nodes. That is, it hides the distinction between distributed and non-distributed regions. For example, with a single node, regions can correspond to memory partitions. Tessellation, the process of assigning regions within a dimension to storage buckets (*i.e.*, IDs – they could refer to nodes or memory cells), can be done dynamically during runtime as long as all nodes agree on the same ordering of IDs. This is the only agreement required upon system startup or reconfiguration.

3.4 Compression Formulas

Unispace hashing as presented in §3.3 solves the challenges enumerated in Table 3.1. However, issues remain:

- It severely penalizes properties that can *uniquely* identify an object (*e.g.*, the "key" property). Using the previous examples, a query that only includes `username` should be enough to return a *single* node. Instead, it just reduces the search space by an order of magnitude (in base r). In fact, we need a fully-specified query to fill a single coordinate vector completely and get a single node – but then, we already have the object we are looking for!

- It wastes dimensions for properties not used for queries. Usually, there exist properties that are used only *after* the result is retrieved, but are never as exact search terms. Examples include multi-word text, template metadata, multimedia, lists of property values, and methods (code). Even if we wanted to search within some of these types, they require special pre-processing.

- It assigns equal query priority to all properties. Given a specific number of nodes, users should be able to accelerate selected queries at the expense of others. For example, it is more common to look up people based on their first and last name, and less common to look them up by eye color.

These issues require user insight, which is supplied by augmenting all operations with a second argument specifying a *compression formula*. For instance, users can insert objects using `put(`obj_1`, `ϕ_1`)` and query using `get(`q_1`, `ϕ_1`)`.

Formulas are configuration objects that specify structural preferences at the level of individual objects. They instruct the system on how to (re)construct the coordinate space on each operation. The q_1 argument above does not need to include wildcard properties (*e.g.*, `ANY`) of an object any more. Knowledge about which dimensions contain known values, which contain wildcards, and which are not even indexed can all be expressed using the second argument, formula ϕ_1.

Compression formulas are centered around three configuration parameters: queryable dimensions, weights, and space overlays.

Queryable Dimensions At the very least, users can specify a subset of dimensions that are important for queries. To locate where to place the object, the system will run the scheme described in the previous section *only* on the dimensions specified in this subset. If any of the properties specified does not exist, it will get a value of 0. The following formula is equivalent to setting a `username` as a primary key in a distributed key-value store.

```
{ space: ["username"] };
```

Weights Users can specify the relative ratio of regions per dimension between the dimensions they plan to index. A higher number of regions for a property means that queries with this property will be serviced more efficiently. The example formula below specifies that queries on `first` should be twice as efficient as queries on `last`.

```
{ space: {"first": 4, "last": 2} };
```

Space Overlays Users can create multiple overlays that are optimized for different types of queries. Each overlay can either contain a copy of the object or a pointer

```
var ac_f = [
 "username",
 "ssn",
 { "first": 4,
   "last": 2 }
];
```

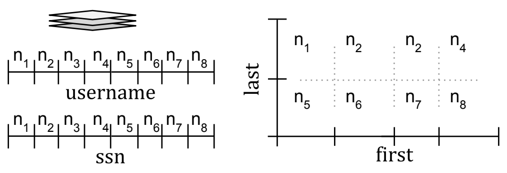


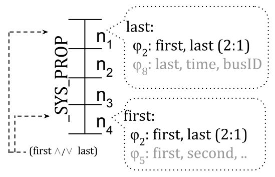
Figure 3.2: **Space Overlays.** Three space overlays resulting from the compression formula on the left (*Cf.*§3.4).

to a single location for this object (specified by, say, hashing all its contents). Since updates to any of its values changes the location of the object for all overlays that include updated values, the former is ideal for read-heavy systems and the latter for write-heavy systems. For queries that touch multiple overlays, the system can process queries with the goal of querying the smallest number of regions. The example below specifies three overlays; the previous two, and a third one for `ssn`:

```
{ spaces: [
  { space: ["username"] }, { space: ["ssn"] }
  { space: {"first": 4, "last": 2} }
]}
```

Fig. 3.2 illustrates the resulting spaces, and a more concise syntax actually used by the system today. If the `ac` object from Fig. 3.1 was updated to include, say, a `notes` property, none of the resulting spaces would use it to index `ac`.

Formulas have several features. They are dynamic: they can be generated during runtime for individual objects. They are also optional; if no formula is provided, the system will still operate as described in the previous section — at a possibly non-ideal configuration. Finally, they maintain the pure, deterministic nature of operations: given (a) a set of nodes (implicitly), (b) a data or query object (as before), and now (c) a compression formula (new), the system will return the same node *independently of the node receiving the request.*



```
function union(propertyList):
 formulaSet = new Set();
 for (p in propertyList):
   F = get({_SYS_PROP: p}, "-1");
   for (f in F):
     if (f.containsAny(propertyList)):
       formulaSet.add(f)
 return formulaSet;
```

```
function intersection(propertyList):
 formulaSet = new Set();
 for (p in propertyList):
   F = get({_SYS_PROP: p}, "-1");
   for (f in F):
     if (f.containsAll(propertyList)):
       formulaSet.add(f)
 return formulaSet;
```

Figure 3.3: **Inverted Formulas.** Use of inverted formulas and structural queries. Union returns ϕ_2, ϕ_8, and ϕ_5; intersection returns ϕ_2 (*Cf.*§3.5).

3.5 Formula Spaces

So far our scheme solves the problems as posed (Table 3.1); and by taking advantage of user insight, it makes judicious use of available resources. However, the use of compression formulas introduces several inconveniences. These can be grouped into two main categories:

- *Formula Management*: Even though formulas are optional, a use upon insertion requires the exact same formula upon query, update, and deletion of the same object. Moreover, users need to manage formulas explicitly and make sure to save and retrieve them between system interruptions.

- *Overlay Reconstruction*: The introduction of overlays makes property-based searching more complicated as it requires knowledge about (i) which overlays include a specific property and (ii) how to reconstruct them, in order to locate the objects. This requires access to all formulas across the system that include a specific property.

It becomes clear that the system needs to store formulas and make their retrieval on secondary attributes efficient. But formulas are themselves dynamic objects, therefore the system can store and query them using the schemes already described. It can use indexable dimensions to avoid indexing metadata that are stored along with the formulas such as inverted indices. It can also use several overlays to accelerate operations on objects that have the structure of a formula. The next few paragraphs explain the details.

Identifiers First, formulas get a property named `ID`. Its value is unique and is used to distinguish between different formulas. IDs can be thought as distributed pointers for naming formulas: normal operations are overloaded to also accept a string in place of a formula argument, which is used to locate and retrieve the formula.

Identical IDs mean identical sets of properties for the formula object. Users can assign human-meaningful IDs such as "Car", or "specialCarInstance". If not provided by the user, IDs are generated by the system using the formula's property names as input. In both cases, users can query or update them similar to any other object. The system also optimizes ID-based operations by using a dedicated space overlay with a single dimension.

The following example shows the use of formula IDs. Suppose we store the following formula:

```
var cf = {id: "cf_user", spaces: {...}};                    (p9)
put(cf); // insert formula to DB
```

Then the following two statements are semantically equivalent:

```
put(obj, cf); put(obj, "cf_user");                          (p10)
```

The first will run as if the formula was given verbatim. The second will first retrieve the formula and then run the operation.

Inverted Formulas To facilitate quick lookup of formulas by property, the system maintains a distributed map from object-properties to formulas containing these properties. It partitions this map by object-property name on a single dimension (`_SYS_PROP` on Fig. 3.3 left). By retrieving formulas, the system can reconstruct each space overlay with its own dimensions, coordinates, and weights.

Inverted formulas are particularly useful for searching for *all* objects that contain a specific property, *independently* of the formula used to store them. For example, the following operation will return all objects that include a property named `first` regardless of formula used:

```
get({first: ANY});                                          (p11)
```

Structural Queries Since the system is already storing compression formulas, we can use the inverted formula space to efficiently answer union, intersection, and negation queries on dynamic data. These queries now amount to getting all the formulas that include the properties needed and running a union or intersection on *them*. This returns only the spaces that are guaranteed to contain the properties the user cares about. The pseudocode in Fig. 3.3 shows how union and intersection queries are handled at the formula space.

This is a lot more efficient than querying the data objects for several reasons: (i) there is a smaller number of formulas, as they get reused for multiple objects (*e.g.*, all objects that look like "car" share the same formula); (ii) formulas are much smaller than the data objects they describe (*i.e.*, on the order of an object's queryable property *names* only); (iii) the resulting object is guaranteed to have the requested structure.[3]

3.6 Summary

Being central to ANDROMEDA, `store` had to address several challenges: (i) be flexible, providing tunable guarantees for replication, consistency, and indexing; (ii) support indexing of dynamic values (§2.1.4), whose structure is not known beforehand; (iii) provide querying efficiency, by avoiding naively querying all of ANDROMEDA nodes for a single object. The combination of (ii) and (iii) is critical, because ANDROMEDA's programming style follows a dynamic typing discipline: as the structure of data is not known beforehand, nor remains static, ANDROMEDA needs to support efficient queries on data stored without static indices. "Efficient" means avoiding the naive solution of flooding all nodes in a group. This efficiency should ideally extend to structural queries returning all objects matching a certain structure; structural

[3] In general, most dynamically-typed languages behave like structurally-typed languages. Under a structural-subtyping [165] lens then, a more precise statement would be that these queries return all the objects that are structural subtypes of or structurally-equivalent with the query object.

queries depend on efficient union, negation, and intersection operations, which in the practice of dynamic languages are often referred to as “duck typing”. These needs make `store` the canonical example of a service design that is configurable at both a high level and fine granularity (§2.1.3).

To address these issues, this chapter developed a new set of techniques that together enable query-efficient partitioning of *dynamic* data. First, unispace hashing (UH) extended multidimensional hashing to data of unknown structure with the goal of improving queries on secondary properties. Second, compression formulas showed how to leverage user insight to address UH’s inefficiencies and further accelerate lookups by certain properties. Third, formula spaces used UH *itself* to simplify compression formulas and accelerate queries on the structure of objects. The resulting subsystem, described in this chapter, supports dynamic data similar to single-dimension NoSQL systems, efficient data queries on secondary properties, and novel intersection, union, and negation queries on the structure of dynamic data.

The next three chapters describe three frameworks built atop of Andromeda, starting with BreakApp (§4)—a framework for automatically sandboxing selected third-party modules.

Chapter 4

Automated Module Sandboxing

Andromeda's cooperative concurrency model and its decision to leverage an existing ecosystem offer significant usability benefits (§2.1.4). However, when combined, these two features introduce a significant challenge: third-party code may decide not yield control, *ever*. This particular denial-of-service (DoS) risk can be partially addressed in several different ways, including static (non-)termination analysis [80, 231], runtime defibrillation techniques [37], and two-level scheduling [150]) (*i.e.*, cooperative, except when user issue a high-priority signal that prefers termination over state inconsistencies). Our solution was the development of a general tool built atop Andromeda, BreakApp [223, 224], that automatically spawns third-party code on remote nodes. BreakApp maintains module interfaces intact, forwarding calls to remote modules and intervening in cases of problems. Our experience developing BreakApp was simplified by Andromeda's service library, and its transformation and serialization primitives.

4.1 Broader Motivation

Software development is changing in scale, process, and basis for trust. Early open-source software such as the Linux kernel or Apache had many people focused on the quality and security of a single codebase [170]. Yet even such cohesive efforts failed to prevent a slate of vulnerabilities [35, 39].

Current software makes extensive use of third-party modules created by different authors and accessed via language-specific package repositories. For example, JavaScript's Node Package Manager [187] hosts more than half a million packages from over 100K authors and serves hundreds of millions of package downloads *per day*. Such public repositories provide no guarantees on modules beyond availability; anyone can create an account and share packages.

A sample of large-scale applications (§8) shows that foreign code accounts for up to 99.9% of that released to clients, and thus most code is neither written nor reviewed by its nominal developers. In practice, glue code stitches together many specialized modules comprising the application into a system with deep, intricate interdependencies. As we show, several hundred third-party dependencies occur in an average application due to recursive imports. This gives rise to security vulnerabilities, as these modules execute with no isolation or privilege separation beyond what type safety provides.

Further problems increase these risks. With popular modules averaging tens of thousands of lines of code, understanding the internals of a complex package and verifying that it will not behave in unintended ways [66, 132] are both extremely difficult tasks. The popularity of certain packages—depended-upon by thousands of other packages—allows vulnerabilities deep in the dependency graph to cause widespread difficulties [123, 106, 246]. Discovered vulnerabilities are becoming harder to eradicate, since some updates are fetched automatically [181], and module unpublishing is becoming a multi-step process in order to avoid breaking dependency chains [242].

Software supply chain attacks are becoming an important concern. Instead of merely reacting to announced vulnerabilities [118, 36, 200] or avoiding composition altogether due to security concerns, *we propose leveraging the trend towards more and smaller modules to enhance, or retrofit, application security.* The core idea is

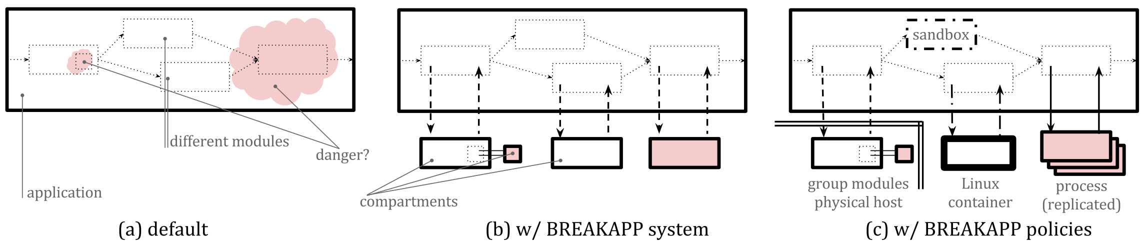


Figure 4.1: **Multi-module server, and possible decompositions.** A simplified server application with multiple third-party modules of varying trust.

to exploit programming language properties (*e.g.*, abstraction, encapsulation, trust boundaries) to automatically transform a program at the module boundaries and offload enforcement to the operating or runtime system (*e.g.*, address space isolation, LXC/namespaces, sandboxing).

BreakApp is a drop-in replacement for a language runtime's module system that pioneers the use of module boundaries as a guide to placing code into protected compartments. BreakApp is centered around a *parametrizable* transformation technique that spawns modules in their own compartments during runtime. Automated transformations (*e.g.*, function calls to remote procedure invocations, garbage collection propagation) hide compartment boundaries, providing the benefits of compartmentalization with low developer effort.

Optional runtime policy expressions fix the aforementioned parameters, effectively decoupling assumptions made during module development from requirements present during module composition. Certain powerful linguistic features, such as introspection and global variables (§4.3.1), pose high risks for inter-module attacks. Allowing developers to disable them when *their* side of the code does not use them eliminates classes of attacks. Since the same module can be used by several different applications, each with its own assumptions and sensitivities, it is important to let the application developer choose which module behaviors to disallow based on *their* side of the code instead of whether vulnerabilities for the modules in use have already been discovered. Moreover, the aforementioned transformations for creating compartments and maintaining the illusion of a single runtime open a rich space of security and performance trade-offs. Thus policies can also improve BreakApp's performance by allowing programmers to customize the provided functionality on a per-import basis.

BreakApp does not require any annotations, does not require any tracing or inference (pre-)runs, and does not require manual rewriting of source code. Policy expressions are backward-compatible with existing codebases and forward-compatible with unmodified module systems. The system lowers potential barriers to widespread adoption and makes incremental security retrofit in existing systems possible.

We demonstrate a prototype of our system targeting JavaScript. We leverage JavaScript's flourishing package ecosystem to show that it mitigates several classes of discovered vulnerabilities, as well as wider classes of hypothetical vulnerabilities. We show that good parameter choices can give acceptable performance results, hitting a sweet spot between security and performance.

Our contributions include:

- identifying an opportunity in today's applications; the use of many third-party modules, although risky, offers clear boundaries of trust (§4.3).
- formulating a parametrizable technique for automatically transforming modules to standalone compartments, allowing users to compartmentalize applications at the boundaries of untrusted modules (§4.4,4.6).
- proposing a concrete set of policies for configuring the aforementioned parameters, effectively allowing users to fine-tune the security, compatibility, and performance trade-offs (§4.5) during runtime.

BreakApp's implementation, its evaluation using a combination of micro-benchmarks and real systems, comparison with prior work, and application of BreakApp's ideas in other environments are presented in the implementation (§7.2), evaluation (§8.3), related-work (§9.3), and discussion (§10.2) chapters, respectively. Appendix B provides technical background on the module system's internals, to ensure that BreakApp's transformations are appreciated by a wider audience.

4.2 Overview

While our concerns with third-party code are language-independent, the problem and proposed solution are developed here for an interpreted language where source code is available.

4.2.1 A Blogging Platform

To highlight typical module usage in modern applications, we consider Ghost, an open source blogging platform. Ghost imports 62 top-level packages and makes use of 981 packages in total. The snippet below presents a simplified version of the core functionality behind such a blogging platform's search capability:

```
var dbc = require("./dbc.json");                                   (p12)
var ejs = require("ejs");
function search(req, res) {
  var f = db.getFiles(dbc);
  var m = require("minimatch");
  var r = m.match(f, req.query);
  res.body = ejs.render(template, r);
}
```

Function `search` takes a request and a response object. It populates results in the response object based on query data from the request and the result-page template from the developer. Each `require` statement imports a module into the current scope. More specifically, it returns the value assigned to the module's `module.exports` variable.

Fig. 4.1-a shows the simplified application running. Boxes correspond to the context of different modules, with the outer box corresponding to the top-level context. All these *logically unrelated* packages execute within the same address space; a problem in any one of the packages exposes other packages, too. But what can go wrong when we are talking about a high-level, memory-safe, managed programming language?

Problem 1 As a simple example, suppose that Ghost generates HTML from templates using `ejs` (line 2). Since this package is susceptible to remote code execution (Table 4.1), a malicious version of this module or a user using the service could try to get access to the database credentials (`dbc`) by a number of different execution paths: (i) attempt to read the global, singleton `dbc` object by taking advantage of JavaScript's default-is-global variable resolution mechanism, the complex `this` semantics, or by reaching to the caller's environment if strict mode is not being used; (ii) access the `dbc` object by dynamically patching the top-level object and interposing on its access; (iii) access the loaded config module by traversing the cache of loaded modules; or (iv) directly import the `dbc.json` config file.

Problem 2 Suppose Ghost provides search functionality using the `minimatch` module, which converts "glob" expressions to regular expressions (line 5). Even if we assume that `minimatch` itself is benign, our application is still vulnerable. Because it is supplied user-generated strings, a malicious user can launch a RegEx DoS attack by providing pathological regular expressions. Since most JavaScript implementations follow an event-driven, cooperative concurrency model, a problematic search query will cause the application to freeze until the pathological request completes.

4.2.2 Strategy: Disabling Features

Language features illustrated in §4.2.1 above and detailed in §4.3.1 below are good to have some of the time, and may be vital in certain cases. For example, users should be able to introspect and rewrite state dynamically, share global variables, compile code, access unsafe code, and traverse the module cache. However, with the prevalence and simplicity of third-party code there are cases when users need to *selectively* disable some of these features. *Within* their dedicated compartments, modules should still have unrestricted access to all these features, but some of them should stop exactly at the compartment boundary. It is the module client who should decide which features are allowed to cross boundaries and which ones cannot. This insight is the driving force behind the design of automated compartmentalization.

The first challenge is giving users the ability to build boundaries within an application as easily as importing third-party packages in their application. This is solved by providing a technique, discussed in §4.4 and illustrated in Fig. 4.1-b, to transparently spawn modules in their own fresh compartments. Isolated modules can only communicate through a tight interface (the module's API) and cannot access state from other modules without using the API. In the earlier example of a simplified blogging platform (§4.2.1), the system can spawn `ejs` in a compartment separate from `dbc.json` and `minimatch` in multiple compartment replicas.

The second challenge is giving users the ability to select the behaviors they need to disable between boundaries and when. This is solved through a policy scheme, discussed in §4.5, that allows users to parametrize several aspects of the compartment types for a particular runtime instance (Fig. 4.1-c). Even if two applications use exactly the same third-party modules, their safety assumptions, use of these modules, the intended deployment environment, and the sensitivity of the non-third-party code justifies disabling very different features — irrespective of whether and what vulnerabilities have been discovered in these modules. For example, a development machine could place all third party code (*i.e.*, *both* `ejs` and `minimatch`) in a single compartment with restricted access to the developer's file system.

The third challenge is having the compartmentalized application maintain the semantics of the original application executing on a single runtime. This is solved using further transformations, discussed in §4.6. Such transformations include converting pointers to distributed references, propagating changes in copied primitive values, and reflecting garbage collection events. For example, when a developer decides to manually unload a module using the interactive interpreter, this should lead to the destruction of the respective compartment and reclamation of its resources.

4.3 Background, Threats, and Opportunities

This section discusses more problems related to third-party modules and outlines which threats fall within our model.

4.3.1 More Problems

The two examples presented in §4.2.1 only scratch the surface of the risks posed by problematic modules. In this section we outline further potential problems. These problems (except (c)) are not specific to JavaScript; substantially similar problems affect languages such as Java, Go, Objective-C, Ruby, and Python. We give each problem a circled letter to allow us to refer back to them when explaining how our system can mitigate each problem (*e.g.*, Table 4.3 in §4.5.2).

Developer Intentions A common source of problems has to do with *development patterns or mistakes.* A common pattern with unintended consequences is global variables ⓐ; for example, having a global, singleton object for an application-wide database or logger configuration [1, 69]. Mistakes include accidentally exposing state at the wrong level ⓑ or making a typo [214] while importing a package (*e.g.*, is it "coffeescript" or "coffee-script"?).

Runtime Capabilities Another set of problems has to do with powerful *reflection and introspection capabilities*, available in many programming languages. These allow any part of the program, including a buggy or malicious module, to examine ⓒ and alter ⓓ the program's own structure and behavior during runtime. Implementations often allow inspecting the call stack for debugging purposes, enabling code to read data from the calling context ⓔ. Runtime code evaluation with `eval` or `exec` in many dynamic languages allows malicious code to hide its intentions ⓕ.

Table 4.1: **Example vulnerable modules.** Eight major vulnerability classes and specific instances of packages available on npm; "++" indicates that many more packages with similar problems exist.

	Problem	Example Package
1	Directory Traversalⓛ	hostr, bitty, restafary, ++
2	Denial of Serviceⓚⓞ	**ejs**, node-uuid, **minimatch**, ++
3	Remote Code Executionⓒⓓ	**ejs**, pouchdb, reduce-calc, ++
4	Timing Attackⓟ	fernet, cookie-signature, ++
5	Uninitialized Mem. Exposureⓔ	mongoose, bl, request, ws, ++
6	Command Injectionⓕ	git-ls-remote, shell-quote, ++
7	Native Code Vulnerabilitiesⓠ	libxmljs, libyaml
8	Sensitive Info. Exposureⓝ	airbrake

Language and Runtime Environment Further problems can arise from *the design of the language.* In the case of JavaScript, for instance, common problems include: default-is-global, where resolution of a name not in scope continues to outer contexts until it reaches the global context (where it might hit something valuable) ⓖ; prototype poisoning, where code at inner contexts can affect objects higher in the prototype chain (effectively mutating all children) ⓗ; a wide range of mutability attacks, where code can edit properties or even rewrite code before calling it ⓘ, meaning there is no guarantee that your code will run as expected; self-reference (*i.e.*, `this`) semantics that might resolve differently depending on how the object is being called ⓙ. Other problems are *implementation-specific*; for example, several high-performance implementations employ cooperative concurrency, choking at seemingly innocuous calls that block the event loop ⓚ [2].

The Module System A largely undiscovered set of problems has to do with *the inherent implementation of the module system.* Module systems do not have the notion of authority: everything is accessible at any point during the execution of the program. A malicious module can use built-in modules to access system resources with the same authority as the rest of the application ⓛ, with code such as `fs = require("fs"); fs.readFile("/etc/passwd")`. Worse, module systems tend to cache loaded modules to avoid overheads from loading and to ensure consistency of any state within the module. As a result, a malicious module can enjoy direct, unrestricted access to the latest instance of any module that is already loaded (*e.g.*, `require.cache`) ⓜ. In many cases, it can even read or write built-in functionality (*e.g.*, overwrite built-in `fs` handlers) ⓜ+ⓘ.

The Broader Environment Some problems are more general and have to do with *the wider system* on which an application is running. For instance, modules have direct access to the calling environment with `process.env` and `process.args` ⓝ. Malicious code within the module or malicious input from users aware of its use can accidentally or intentionally exhaust resources ⓞ. Modules can also leak timing information about their state, computation, or underlying resources from observable changes of how long it takes to execute a request ⓟ.

Native Modules Finally, modules can interface with code written in other languages. Reuse of existing libraries via foreign functions is a compelling proposition, but use of unsafe code nullifies the safety guarantees of a high-level, memory-managed programming language ⓠ.

4.3.2 Threat Model

The general model of threats arising from the use of modules is quite broad. In practice, however, users are expected to shield applications against only a specific subset of these threats.

Source of Attacks In terms of attack origin, we care about three broad types: (i) a malicious module directly attempting an unintended action (*e.g.*, cause a DoS attack by looping infinitely); (ii) a malicious module indirectly coercing a different module in the dependency graph with a known vulnerability into performing an unintended action (*e.g.*, cause DoS by carefully using another module's API); (iii) a user feeding a problematic module input that triggers an unintended action (*e.g.*, cause a DoS by submitting problematic search queries).

Attacks We want users to be able to protect against code that attempts to violate the confidentiality (*e.g.*, read global state, load other modules, exfiltrate data) and integrity (*e.g.*, write global state or tamper with the module cache) of application data and code. Moreover, the code can attempt to read or write the broader environment within which the application is executing, including environment variables, hardware counters, the file system, or network. We want to mitigate these attack vectors by allowing users to disable access to specific variables, specific modules, or system-level capabilities, such as file system or networking primitives.

We additionally seek to weaken attacks on availability. Pathological inputs from attackers can disrupt otherwise benign modules within an application (*e.g.*, RegEx matching [45] or JSON parsing [193]). Potential mitigations range from simple reporting, to back-pressuring malicious input, to decreasing resources of malicious compartments, to shutting down offending compartments.

We also want to make it possible for users to shield time-sensitive modules from timing attacks. In particular, we want to allow users to set specific minimum response times for cross-boundary calls.

Assumptions We assume that the core language runtime and built-in libraries such as `fs` and `net` can be trusted. As we show in Chapter 8, our technique allows spawning built-in modules in their own dedicated compartments. However, the system requires a minimum of trusted functionality from the underlying system, such as the ability to locate and load the right program files required by a module. This can be achieved by including a trusted (say, formally verified) version of a mini standard library (*e.g.*, enough to locate files, load modules *etc.*). We do not explore these options in this work.

BreakApp can be run either as a third-party module loader on a per application basis or as a system-wide module loader replacement. In the former case, we assume that users load BreakApp *before* any other module; otherwise, a malicious module could dynamically rewrite BreakApp's code. Using a defense-in-depth approach, BreakApp checks whether other modules attempt to rewrite any of its core structure using both static checks (the moment BreakApp loads other modules, it parses their source code) and dynamic interposition hooks on its internal objects (§4.6.5). Moreover, most of its core structure is immutable: hidden object properties are set to non-enumerable, non-writable, and non-configurable modes; and policies are by-default frozen after the initial configuration.

Limitations Attacks targeting package managers are related to, but distinct from, those we protect against. Most package managers implement pre-install, post-install, testing, and other scripts that are package-specific. Since these scripts are Turing-complete programs similar to full-fledged modules, they can be used to launch attacks to the system before, during, or after package installation similar to the ones described earlier (*e.g.*, read environment variables, denial-of-service attacks *etc.*). However, these are beyond the scope of this work and are better addressed with other methods [31].

4.4 Transformations: Module Decomposition

This section discusses automation related to spawning modules in their own dedicated compartments.

At its core, BreakApp changes the implementation of all module import statements to (i) spawn a new compartment for each previously unseen module, (ii) modify the return value so that use of module's members (property accesses and function calls) will invoke RPC proxies to the newly-spawned compartment, and (iii) redirect module-specific side-effects, such as console output or exceptions, to the importing compartment. BreakApp also monitors the health and status of all compartments.

```
transform (e : DAG) (toRPC: Fun -> Fun) : DAG :=
match e with
| Obj ((k, v) :: xs) => Obj ((k, transform v) :: mt xs)
| Arr x::xs => Arr ((transform x) :: mt xs)
| Fun f => toRPC(e)
| _ => interpose(copy(e))
end
where mt = map transform
```

Figure 4.2: **Transformation core.** The core transformation; example result in Fig. 4.3

4.4.1 Compartment Setup and DAG Transformations

Whenever the program executes an import statement, control jumps to BreakApp. Consulting the policies (§4.5) associated with this import statement, it chooses whether it should spawn a new compartment. If the policy dictates that it should, it creates a new *child* compartment for the imported module and sets up a new communication channel between the two. It replaces core functionality on the child compartment, such as console printing, in order to propagate certain side-effects to the parent compartment.

Within the child, BreakApp copies and transforms the return value for the raw imported module before sending it to the parent compartment. The general case of such a value is a directed acyclic graph (DAG). The system walks the DAG and transforms its component values so that function and method calls propagate to the compartment.

The exact transformation is parametrizable on several aspects related to policies, but it can be summarized as follows:

- *primitive* values are copied unmodified and wrapped with an interposition mechanism that records changes.
- *function* values are replaced with an RPC stub that, when called, will serialize arguments, send them to the current compartment, and deserialize the return

values.

- *objects* are recursively copied and transformed, with their getter and setter functions replaced with RPC stubs similar to function values.

If the specified module throws an exception while being loaded, the exception is caught by BreakApp running on the child, serialized, and re-thrown by the parent compartment.

If the specified module is already loaded and the policy associated with this import statement allows module reuse, BreakApp simply retrieves the channel pointer and returns the previously-transformed DAG copy. Fig. 4.2 summarizes the transformation algorithm. We discuss an example transformation (Fig. 4.3) at the end of §4.6.

4.4.2 Function Calls as RPCs

BreakApp mediates between the parent and child compartments. Synchronous calls yield to the module scheduler, which serializes arguments, sends them through the channel to the child, and blocks for a response. The child-side wrapper deserializes arguments, calls the required method, and sends results back through the channel. For asynchronous function calls, the parent module wrapper registers an event that invokes the provided continuation (with the available results) when a result is made available on the channel.

In cases when something does not go as expected in the child's execution, its code will throw an exception which BreakApp serializes and returns to the caller compartment. BreakApp on the parent compartment will inspect the exception and, if it is related to any violations (*i.e.*, it is not an exception coming from BreakApp itself), it will re-throw it. BreakApp-specific exceptions are handled specially, depending on the violation.

4.5 Policies: Tuning Trade-offs

This section discusses policies and how they automate control over tradeoffs among security, compatibility, and performance.

4.5.1 Expressing Policies

Policies (Table 4.2) can be expressed both at the level of the whole application and the level of each module. In both cases they are optional. BreakApp's default policy is overriden by application-wide policies, which are in turn overriden by per-module policies.

Application-wide policies generally describe coarse guidelines on how to decompose the application. Typical coarse guidelines include the maximum number of compartments (`LEVEL`), action to take in case of violations (`ON_FAIL`), compartment type (`BOX`), and application-wide global variables (`CONTEXT`). They can be expressed at the point of BreakApp's initialization:

```
require("breakapp")({box: require.boxes.SBX});                    (p13)
```

The line above specifies that all modules should be loaded in their own, fresh, software-isolated sandboxes (`SBX`). It creates a new runtime context with fresh built-ins and top-level objects for each module.

Module-specific policies give developers fine-grained control over decomposition, allowing them to capture intuition about the properties (regarding security or performance) of the modules they use. Per-module policies are expressed at the module's import statement:

```
require("minimatch", {box: require.boxes.PROC});                  (p14)
```

The line above specifies that the `minimatch` module should be loaded in its own, fresh process. It creates a new address space, and lets the operating system provide support for isolation, scheduling, and interprocess communication. If `minimatch` is a module written in C, it cannot even forge a pointer to poke into the main application's address space. However, it may take a bit longer to load and communicate than the rest of the sandbox-based compartments.

The combination of the two policies above should now be clear: load each module in its own software-isolated sandboxes but load `minimatch` in a new process. If `minimatch` is already loaded in its own compartment, BreakApp will spawn a new instance of `minimatch` in its own process.

Notable characteristics of policy expressions include:

Generation: In all cases the policy object can be generated programatically during runtime (*e.g.*, from command line arguments, from the environment, or through a pre-processing stage). This gives programmers considerable flexibility, and allows tools built on top of BreakApp to generate policies dynamically in response to changing load patterns or evolving threat models.

Compatibility: Per module policy expressions are *fully* compatible with existing codebases. Expressing policies is *backward*-compatible with systems that do *not* provide a BreakApp-enabled module system; due to variadic arguments, the policy argument is ignored by the built-in `require` function. Not specifying policies (*i.e.*, all of the code out there today) is *forward*-compatible with systems that *do* provide a BreakApp-enabled module system: as explained earlier, BreakApp will use the application-wide default configuration.

Extensibility: Policies are extensible. BreakApp allows users to override most of the functionality during initialization (*i.e.*, the application-wide policies described earlier) by passing in functions. The sets of policies described here are simply default extensions bundled together with the system. If users need to provide further functionality (*e.g.*, use a different type of compartment or take different actions upon failure), they can hook up their own implementations.

Table 4.2: **Selected policies.** Examples of interesting policies.

Policy	Example Options	Explanation	§
Box	`SBX, PROC, LXC`	Compartment type	4.5.2
IPC	`TCP, UDS, FIFO`	Communication type	4.5.2
Context	`{global: global}`	Share pointers with parent	4.5.2
Level	`0, 1, ..`	Depth at which to decompose	4.5.3
Group	`subtree.json`	Group dependency subtrees	4.5.3
Trust	`["fs", "http"]`	Whitelist trusted modules	4.5.3
Doubt	`["ejs"]`	Blacklist untrusted modules	4.5.3
Instances	`FUSE, PART`	Fresh compartment per import	4.5.4
Replicas	`true, 23`	Multiple replicas (round-robin)	4.5.4
OnFail	`(e) => {..}`	Action upon failure (function)	4.5.5
Compose	`OURS, THEIRS`	Priority in policy conflicts	4.5.6

Table 4.3: **Compartment types.** Different compartment types and problems they mitigate: vanilla module system (NONE), sandboxes (SBX), processes (PROC), and containers (LXC)

	NONE	SBX	PROC	LXC	Notes
ⓐ ⓑ	✘	✔	✔	✔	globals, state
ⓒ ⓓ	✘	✔	✔	✔	introspection
ⓔ	✘	—	✔	✔	stack inspection
ⓕ	✘	—	✔	✔	evaluation
ⓖⓗⓘⓙ	✘	—	✔	✔	context
ⓛ	✘	✘	✘	✔	fs, net (leaks)
ⓜ	✘	✘	✔	✔	module cache
ⓝ	✘	✘	✔	✔	process args
ⓝ	✘	✘	✘	✔	process env
ⓚ ⓞ	✘	✘	✔	✔	denial of service
ⓟ	✘	✘	✘	✔	side-channels
ⓠ	✘	✘	✔	✔	unsafe extensions

4.5.2 Isolation Primitives

Different compartment types provide different guarantees in terms of isolation, but also affect performance directly. Table 4.3 shows how different isolation primitives mitigate different types of problems described in §4.3.1:

- Sandbox Isolation (`SBX`): This creates a new software-isolated context within the same runtime. Built-in utility functions (*e.g.*, `Math.pow`) and top-level objects (*e.g.*, `Function.call`) are fresh, and global variables not explicitly white-listed are not shared. The sandbox shares the same heap and event queue with the rest of the application.

- Address Space Isolation (`PROC`): This creates a new runtime instance as a new process, with its own address space, stream and IPC handles. The system leverages the OS kernel to synchronize communication and set scheduling priorities between compartments.

- Container Isolation (`LXC`): This creates a new runtime instance within a fresh container instance. Containers can restrict process-trees from accessing arbitrary parts of the filesystem. They can also restrict access to the network and set resource restrictions to the use of CPU and memory.

Heavier compartment types and hence more expensive performance costs are positively correlated with better isolation. However, after fixing the compartment type, there is room for further tuning performance and isolation independently of each other. Isolation guarantees can be fine-tuned without affecting performance by declaring which state compartments are allowed to share (`CONTEXT`). For instance, users can share some of the global variables, some of the built-ins, and choose to allow access to the module cache. Performance costs can be fine-tuned without affecting isolation by choosing one of the available communication channel types (`IPC`). For instance, in the case of process-level isolation, TCP streams provide better throughput, but Unix Domain Sockets and Unix FIFO Pipes offer lower latencies (Table 8.6).

4.5.3 Decomposition Granularity

Decomposition granularity affects how many modules to launch into separate compartments and is directly related to the number of compartments created. This, in turn, is positively correlated with finer-grained security, since there are fewer components to which a piece of code has unrestricted access. The increase in security generally assumes that, all other things equal (*e.g.*, programming language, code paths *etc.*), there is a correlation between lines of code and exploitable bugs [127]. However, a larger number of compartments can affect performance negatively by increasing startup times and communication costs.

There are many ways of guiding the BreakApp compartmentalization scope. At a coarse granularity of specification, users have two knobs: vertically, define the level (`LEVEL`) at which to decompose (*e.g.*, only top-level, every level, only last level *etc.*); and horizontally, define the granularity (`GROUP`) of dependency subtrees (*e.g.*, package-level, file-level, *etc.*). At a fine granularity of specification, they can blacklist components that should always launch in a new compartment (`DOUBT`) and whitelist compartments that are trusted and should always stay with the parent compartment (`TRUST`). BreakApp already uses module whitelisting to avoid spawning built-in modules and its own trusted dependencies.

4.5.4 Instantiation and Replication

Identical-looking import statements might get resolved into different absolute filenames depending on where they are called in the dependency chain. By default, BreakApp takes this into account and spawns new compartments only when the vanilla module system would actually import a module. However, users can request BreakApp to further *replicate* a module to address DoS concerns (`REPLICAS`). Replication requires user insight because modules that encapsulate state have the potential to introduce state inconsistencies. When used, the number of replicas can be declared statically upon startup or inferred dynamically in response to changes in the load and module response rate. Users can also select a scheduling policy (`SCHED`) from an existing set (*e.g.*, round-robin) or can define and pass a custom one.

4.5.5 Other Policies

When a violation is detected, BreakApp can select between several actions, depending on the type of the exception (`ON_FAIL`). Among other things, it can log the violation, email an administrator, kill or restart the compartment, or launch a new replica. Other policies include scanning the module's source code to pro-actively spawn compartments in parallel before they are requested (`PRELOAD`), encrypting communication between compartments (`ENCRYPT`), setting minimum response times (`TIMER`), whitelisting environment variables (`ENV`), and soft-reloading modules without restarting the compartments (`RELOAD`).

4.5.6 Conflict Resolution

The introduction of BreakApp to a package ecosystem will inevitably lead to conflicts of policies. First, third-party packages will start importing other packages using what they think are the right policies. Then, applications importing these packages might choose to use different policies. There is no single way for resolving these conflicts: in some cases the library developer knows better, but in others the top-level application developer knows more about the intended audience—until, of course, their application is used as another application's library.

To solve this, BreakApp comes with a number of conflict resolution options (*e.g.*, accept-ours, accept-theirs, accept-most-restrictive, accept-most-permissive). All policies can "lock" at top level, trumping any other policy expression found in third-party modules.

There is no conflict between different versions of BreakApp, since there is only one version running: the one starting with the application at top-level. Even if other modules import BreakApp later, the BreakApp instance that is already loaded will bypass these imports as no-ops (*i.e.*, ignore their application-wide policies) and return its own singleton instance.

4.6 Maintaining a Single Runtime

This section describes several technical details related to transformations intended to maintain the original application behavior.

4.6.1 Maintaining Pointers

Generally, since BreakApp starts its transformation from the object returned from a module (*e.g.*, `module.exports`), values *are* associated with a name: the name of the attribute associated with that value. However, not all values in messages include a meaningful name. For instance, a function can be anonymous and an object can just be a bytebuffer. To facilitate cross-compartment addressing, the child compartment maintains a hash table mapping object and function IDs (*e.g.*, SHA256 checksums) to their in-compartment pointers. These IDs can be thought as distributed, shared-memory pointers which RPCs include in their messages. Whenever it receives an RPC message, BreakApp on the child compartment looks into the table and routes freshly-deserialized arguments to the right function or object method.

4.6.2 DAG Structure and Reference Equality

The creation of object copies during transformation and serialization breaks reference equality. BreakApp takes care to preserve it. When an RPC leads to a new memory alias in the remote compartment, the return message from the remote compartment will include an `alias` entry containing the remote object ID. BreakApp on the child compartment then creates and returns a reference to an existing object. The same consideration must be extended to preserving reference equality for the root of the DAG between RPCs. A common pattern in many languages is to have methods that return `self`; such code would break if the return value of the RPC was a fresh copy of the method receiver.

```
var Point = (x, y) => {
  this.x = x; this.y = y;
};
Point.prototype.toStr = () => {
  `(${this.x}, ${this.y})`
};
module.exports = {
  create: (x, y) => {
    new Point(x, y)
  }
};
```

```
var _create = (..args) => {
  var o = create.apply(args);
  var id = generateId(o);
  return _BA.Proxify({
    x: o.x, //copy
    y: o.y, //copy
    toString: (..args) => {
      return _BA.RPC({
        "mod": "point.js",
        "obj": id, //07c2b7..
        "fun": "toStr",
        "arg": _BA.from(args)
        "fun": "toStr",
      });
    };
  });
};

module.exports = {
  create: (..args) => {
    return _BA.RPC({
      "mod": "point.js",
      "obj": "exports",
      "fun": "_create",
      "arg": _BA.from(args)
    });
  };
};
```

Figure 4.3: **Example application of transformation.** A simplified example of BreakApp-related transformations. `_BA` corresponds to the BreakApp library.

4.6.3 Ordering

Messages get assigned a sequence number. Although communication primitives are reliable, messages should be received at the correct call order. For example, an asynchronous call to the printing function will be shown before the next call to the same function.

4.6.4 Calls to Constructors

Constructors, usually prefixed by the `new` keyword, slightly change the semantics of a function call: at the very least, new memory may need to be allocated. The RPC stub uses additional logic to detect this case.[1] If the function is indeed called with as a constructor, the RPC message has a special type signifying that the target function should also be called with `new`. The return value from a constructor is itself an object whose methods are RPC stubs as described earlier: the true object lies within its compartment.

[1]There are many possible ways of doing this; in JavaScript, the simplest one is to check the value of `new.target` within the wrapped function's scope.

4.6.5 Move vs. Copy Semantics

It is worth clarifying the distinction between values that are remotely referenced and ones that are copied to the parent compartment. When all nodes in the returned DAG are methods, they are transformed to RPC stubs referencing values that live within the remote compartment. State updates targeting such well-encapsulated modules or objects call directly into the remote object. When some nodes in the DAG are primitive values however, they result in deep copies of values. Writes to such values or the RPC stubs themselves[2] need to be detected and propagated to the original object.

To achieve this, we wrap the transformed output DAG with an interposition mechanism that provides reflection capabilities and gets invoked upon attribute accesses. A special BreakApp `Proxy` wrapper[3] detects and records changes to any of the object's properties. Property values that are themselves objects require nested proxies (Fig. 4.2). These state updates are compressed into changesets, and propagated lazily by piggybacking on future RPC calls.

4.6.6 The Class Hierarchy

In object-oriented programming languages, an object might invoke methods inherited from an object higher in the class hierarchy. These superclasses (or prototypes, for prototype-based languages such as Lua and JavaScript) might have been imported from a different module. A naive implementation of transformations to RPC stubs can then lead to a series of nested boundary-crossings until the outer RPC reaches its final destination. BreakApp detects class (prototype) hierarchy levels while traversing the DAG and crafts RPC stubs so that they immediately redirect to their final destination.

[2]Whether this is allowed or not is a policy-specific question, discussed in §4.5; here, we merely show how our mechanism *has* the ability to detect it.

[3] Metatables in Lua or `reflect` in Java provide similar capabilities.

4.6.7 Native Functionality

Objects high in the prototype chain are supported natively. Functionality is either implemented internally in the runtime (*e.g.*, serialization and cryptography modules) or wraps OS-level subsystems (*e.g.*, networking and filesystem modules). In most cases, a copy of these objects can be found in the trusted copy of the runtime (see §4.3.2) which BreakApp includes in the new compartment. Examples include modules such as `crypto`, `http-parse`, and `fs`, and globals such as timer functions and top-level objects.

There are cases when this is not possible, however. Specific global or pseudo-global[4] objects in the child compartment require redirection to the top-level compartment. Examples of such objects include `console` and `process` to refer to terminal output and process-level data, respectively.

If compartments live in different address spaces, writes to the child compartment's out and error streams must be transmitted to the top-level process. Upon first import, the system shadows `log`, `warn`, and `error` with such redirecting proxies. Similarly, it shadows stream input functions with functions that request this functionality from the top-level compartment, which sends the results back to the child.[5]

4.6.8 Garbage Collection

The standard runtime garbage collector (GC) cannot "see through" compartment boundaries to collect objects within translation tables. So, in addition to reflecting method calls between compartments using RPCs, BreakApp also propagates garbage collection events by adding a GC hook to every object that is the result of a transformation. When such an object is about to be collected, BreakApp sends

[4]Server-side JavaScript implementations make several objects that are not part of the EcmaScript specification available in the global scope, such as `process` and `console`.

[5]In practice, modules asking for top-level user input are extremely rare.

a message to the child compartment to remove any references to this object.

Whole modules are more difficult to go out of scope for the GC to kick in and reclaim their memory. This is because there are multiple references to a module in the cache of the loaded modules. However, modules are often unloaded or reloaded manually, which should destroy or restart the child compartment. To maintain this behavior, BreakApp wraps the module cache structure, detects invalidations, and forces the child compartment to exit. Malicious modules cannot cause other modules to exit, because child compartments do not have access to other cache entries.

4.6.9 Monitoring

BreakApp interposes on inter-compartment communication, tracking the load placements and frequency of calls on each channel. It monitors the health (*i.e.*, crashed, not responding) of child compartments periodically and upon remote invocations. It takes curative actions based on the compartment's status (*e.g.*, restart, kill, or spawn more compartments). This is helpful in cases where the module within the compartment is launching a DoS attach or where asynchronous execution has lead to exceptions. Child compartments use OS primitives (*e.g.*, `SIGHUP` on Linux) to be notified upon parent exit.

4.6.10 Wrapping Up

Fig. 4.3 shows the result of a simple module after two stages of transformations. The first transformed the return value (`create`) of the module, and the second transformed the return value of a call into the module (a `Point` object). These transformations are done during runtime and captured only for illustrative purposes.

The left-most column contains most of the original module and its export statement. The right-most column exports a wrapper for `create` that serializes arguments and calls back to the original function. The middle two columns show the result of transforming a newly `_created` `Point` instance: `generateId` will store the object to a translation table, and return a remote reference. The transformed `toStr` will always call back into the original object, whereas access to its `x` and `y` fields is `Proxy`ied through the interposition object.

4.7 Summary

Andromeda's cooperative concurrency model and its decision to leverage an existing ecosystem offer significant usability benefits (§2.1.4). However, when combined, these two features introduce a significant challenge:

This chapter described a general tool built atop Andromeda, BreakApp [223, 224], that addresses the problem arising from the combination of a third-party ecosystem and Andromeda's cooperative concurrency model—namely, that malicious third-party code may decide not yield control, *ever*. BreakApp automatically spawns third-party code on remote nodes. It maintains module interfaces intact, forwarding calls to remote modules and intervening in cases of problems. Our experience developing BreakApp was simplified by Andromeda's service library, and its transformation and serialization primitives.

The next chapter (§5) expands on BreakApp's techniques to offer lower overheads.

Chapter 5

Language-based Module-Level Compartmentalization

Compartmentalization á la BreakApp is hindered by several practical limitations: the introduction of concurrently executing compartments may break soundness; synchronous, blocking module interfaces at the compartment boundary may need to be manually rewritten to non-blocking ones; the granularity of privilege control is often coarse (*i.e.*, at the level of entire modules or system calls); and the use of heavyweight isolation mechanisms hinders the runtime performance of the resulting compartmentalized system. In a sense, the price paid for protecting a module is way too high—when applications today use thousands of modules.

5.1 Broader Motivation

Today's ubiquitous reliance on third-party modules[1] has led to an explosion of supply-chain attacks [123, 110, 118, 36, 223, 200]. Subvertible bugs and actively malicious code in imported modules provide attack vectors that are exploitable long after modules reach their end-users. Standard module systems provide no meaningful

[1] The terms package, module, and library are used interchangeably.

isolation or privilege separation between untrusted modules and the trusted portions of an application, as module boundaries are a purely development-time construct. With popular modules averaging tens of thousands of lines of code, understanding their internals and verifying their behavior are both extremely difficult tasks [66, 132]. The popularity of certain libraries—depended upon by tens of thousands of other libraries or applications—allows vulnerabilities deep in the dependency graph to affect a great number of applications [106, 246, 252]. Discovered vulnerabilities are becoming harder to eradicate, as updates are fetched automatically [181] and module unpublishing is becoming a multi-step process to avoid breaking applications [242]. Worst of all, leaked publishing tokens allow anyone to update packages with code that will eventually reach end-users via package updates [152, 125].

Recent work [144, 108, 213, 224] has shown that *module-level compartmentalization* (MLC) can serve as an attractive mitigation by reducing the privilege of third-party libraries. The key insight behind MLC is that module boundaries already specify *trust* boundaries, which can guide security-oriented application compartmentalization. MLC works by transforming legacy applications into ones where each module (or group of modules) runs in its own dedicated compartment, with enough privilege only to perform its own task. Isolation is typically offloaded to the operating system (OS), which offers tangible protection guarantees with powerful mechanisms such as processes and containers. MLC shows great promise, as it can (i) retrofit security into *legacy* systems not designed with security as their primary concern, (ii) protect against a plethora of *real* attacks stemming from defective, subverted, and malicious elements, and (iii) shield against attacker-controlled components with *unknown* vulnerabilities and powerful *runtime* code evaluation.

Despite its promise, MLC is hindered by several practical limitations (§5.2.2): the introduction of concurrently executing compartments may break soundness; synchronous, blocking module interfaces at the compartment boundary may need to be manually rewritten to non-blocking ones; the granularity of privilege control is

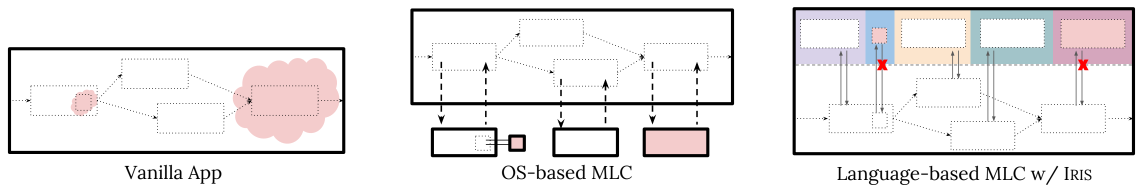


Figure 5.1: **Overview of Iris** Left: application with many third-party modules, some of which may be malicious (*Cf.*§5.2.1). Middle: conventional, OS-based module-level compartmentalization (*Cf.*§5.2.2). Right: language-based module-level compartmentalization, such as the one enabled by Iris (*Cf.*§5.2.3).

often coarse (*i.e.*, at the level of entire modules or system calls); and the use of heavyweight isolation mechanisms hinders the runtime performance of the resulting compartmentalized system.

The key insight behind our work, Iris, is that all these limitations can be lifted by replacing operating-system protection mechanisms with language-based ones. This is enabled by observing that (i) the vast majority of applications that use third-party modules today are written in memory-safe languages [50, 249], and (ii) assuming memory safety, *most* of MLC's goals (§5.3) can be achieved using a combination of context customization and language-level interpositioning. Any memory-unsafe modules can be *individually* contained using prior compartmentalization systems [167, 24, 78, 213], at a fraction of the original cost.

At its core, Iris extends a programming language's module system with the notion of privilege. Specifically, it allows developers to refine the privilege available to a module, and such privilege refinement is expressible and enforceable at a very fine granularity.

To provide this ability, Iris introduces *privilege-interface contracts* (PICs). PICs specify the functionality accessible within a module—at the level of individual fields of each module's return values. Such functionality includes functionality *explicitly* imported by modules (*e.g.*, a `log` module's `info` function) as well as that *implicitly* available to modules (*e.g.*, top-level objects, global variables, import capabilities). Examples include "can only access the `PWD` environment variable", "can only write once the first element of an array", and "can only import `fs`".

To enforce PICs, Iris provides a set of *automated runtime transformations* that introduce security monitors at module boundaries. During execution, monitors interpose on all of each module's accesses that cross its boundary, ensuring they conform to the policies specified by its PIC. This is achieved by shadowing each module's return value and its surrounding environment. Such shadowing forces name resolution to go through Iris-wrapped values, ensuring the module can only access names that resolve to Iris's interposition wrappers. These wrappers, in turn, contain logic that enforces the PICs as specified by the developer.

Key evaluation results demonstrate that language-based module-level compartmentalization à la Iris improves performance over OS-based MLC by several orders of magnitude, reduces developer effort in identifying and rewriting module interfaces, enables fine-grained privilege control, and avoids the introduction of unsoundness—to the point of scaling up to applications with thousands of modules and hundreds of thousands lines of code.

We begin with an example of the problems caused by third party modules, the challenges faced by prior MLC systems, and how they are addressed by Iris (§5.2). We then discuss the threat model and assumptions behind Iris (§5.3). §5.4–5.5 present our key contributions:

- a fine-grained compartment specification primitive, the PIC, that gives developers control over module-external functionality individual modules can access (§5.4),

- a set of techniques for interjecting, transforming, and rebinding the module's context, introducing security monitors that enforce PICs at compartment boundaries during program execution (§5.5),

While third-party module problems and language-based MLC are pertinent to any memory-safe programming language, we expound on the JavaScript ecosystem—

primarily because (i) it boasts the largest collection of modules (and number of problems) [50], and (ii) it simplifies comparison with prior MLC systems [224]. As such, our implementation (§7.2) of the augmented module system as an easily pluggable package, along with its performance and security evaluation (§8.4) focus on JavaScript. Discussion of prior work and application of Iris's ideas in other environments are presented in the related-work (§9) and discussion (§10) chapters, respectively.

Appendix B provides technical background on the module system's internals, to ensure that Iris's transformations are appreciated by a wider audience.

5.2 Background and Overview

We illustrate the problems of third-party modules (§5.2.1) by revisiting the recent `event-stream` incident [201, 153]. We then introduce conventional MLC and its limitations (§5.2.2), and close with an overview of Iris (§5.2.3). The discussion is accompanied by Fig. 5.1.

5.2.1 Running Example: A Bitcoin Wallet

To understand the problems with third-party libraries, consider the `event-stream` incident [201, 153]. Meant to simplify working with data-streams, at the time of the incident `event-stream` was used by hundreds of applications and averaged about two million downloads per week. At one point, its author handed off maintenance to a volunteer; this is common practice when an open source developer reaches a saturation point. The new maintainer added an obfuscated, malicious package as a dependency to `event-stream`, called `flatmap-stream`, with code that was designed to harvest account details from selected Bitcoin wallets.

Fig. 5.2 zooms into (simplified) fragments of the attack. If run as part of a specific Bitcoin application (Copay), it starts by loading the `account.js` module *event-stream*

```
 1 let es = exports;
 2 es.map = require("flatmap-stream");
 3 es.pause = …
```

flatmap-stream

```
 1 let s = require("stream");
 2 exports = (m, o) => {…}
 3 …
 4 let c = require("btc-wallet/account.js");
 5 let gk_old = c.getKeys;
 6 c.getKeys = (...args) => {
 7   let k = gk_old(args);
 8   require("http").request(RMT_SRV).end(k);
 9   return k;
10 }
```

Figure 5.2: **Use of third-party modules.** Malicious code, highlighted in red, patches the `getKeys` method, spoofing credentials and sending them over the network to a remote node.

related to the credentials of the user's Bitcoin wallet (line 4). Since the module is already loaded by the runtime, the language's module system returns a cached copy of the module's return value. This gives the attacker the ability to overwrite (*i.e.*, monkey-patch) its `getKeys` method at runtime (l.5,6). The new method accesses the sensitive account credentials by invoking the original method (l.7). It then loads the `http` module, and transmits the credentials to a remote, third-party server (l.8). Finally, it returns the expected results to the caller method (l.9).

While this is a simplified version of the attack, used to illustrate MLC and Iris, *the real attack is not detectable via static or dynamic analysis*. Static analysis would not have helped because the attacker employed a series of encryption passes over the malicious code [191]. Dynamic analysis would not have helped either, because the malicious code activated very selectively: only when `event-stream` was part of Copay's dependency tree, only when the application run on the "live" bitcoin network, and only on users that had a balance of ₿100 or more.

This case is not alone: malicious modules have exfiltrated sensitive environment variables [26], public SSH keys [154], and credit card numbers [155]. While the JavaScript package ecosystem dominates headlines—with >1M packages, >150K authors, >1Bn downloads per day—the trends are widespread across languages and worsening: a record-setting 16,000 new vulnerabilities due to third-party packages were disclosed in 2018 [208].

5.2.2 Conventional, OS-enforced MLC

The key issue underlying the `event-stream` attack is that any third-party fragment of an application has unrestricted access to the functionality available to the rest of application. Some of this functionality is *explicitly* provided by other libraries such as `fs` and `http`. Other functionality is *implicitly* provided by the programming language; examples include the ability to use global variables, import modules, and access the cache of the loaded modules. Both explicitly and implicitly provided functionality is exploitable by third-party code. While it may be needed for the application to function as a whole, it is not necessarily needed by `event-stream`.

OS-enforced MLC (OSMLC) [144, 108, 224] leverages this insight to restrict functionality at the boundaries of third-party modules. For example, OSMLC would protect against `event-stream` and its malicious dependency by spawning them as separate processes. Address space isolation makes top-level and global objects of the application inaccessible; process containment restricts the module's access to `fs` and the `http`; and local copies of interfaces localize the effects of overwriting module APIs. OSMLC is a powerful tool for security; however, it creates a few challenges of its own.

Soundness Challenges OSMLC can introduce *unsoundness* by creating concurrently executing compartments [78, 32, 224]. Ensuring that the compartmentalized program is sound with respect to the original program is challenging; after all, introducing concurrency is prone to introducing additional behaviors that are not possible in the original program. Examples include new sources of non-determinism, multiple copies of code and data, and conversion of cooperative scheduling to preemptive. Access to state shared among several modules can lead to inconsistencies, race conditions, deadlocks, causal ordering violations, and other classic distributed-systems problems [78]. Execution must also deal with the potential for compartment failures, which in turn can also compromise assumptions about execution behavior and global invariants.

Manual-Effort Challenges In many cases, OSMLC implicitly alters the programming model at the module boundary, which itself requires significant *manual effort*: as I/O is introduced, module interfaces that are blocking often need to be converted to ones that are non-blocking, asynchronous, and concurrent. When a function evolves from being compute-only to potentially yielding, all functions along the path from the function whose call semantics have changed up to the root of the call graph may potentially have to be split in two—a side-effect known as “function ripping” [5].

Granularity Challenges OSMLC is often restricted to *coarse-grained* monitoring and enforcement, because OS mechanisms are applied over coarse boundaries—such as processes *etc.*—rather than fine ones (*e.g.*, individual object fields or methods). It can easily allow or deny access to an entire module, and often monitor OS-level interfaces—with mechanisms such as system-call interpositioning and container namespacing; however, semantic monitoring at the granularity of module fields requires support from inside the compartment, and outside the strict visibility of OS mechanisms.

Performance Challenges Most importantly, OSMLC incurs significant *performance overheads* due to heavyweight OS mechanisms enforcing boundaries across the entire dependency graph. Invocations at the module boundary become excessively costly due to (de-)serialization, interprocess communication, and context switching costs. Boundary crossing is especially problematic in cyclic dependencies where a boundary has to be crossed multiple times for a single top-level call. Spawning copies of the runtime system, built-in libraries, and the compartmentalization framework for every new compartment requires vastly more resources (*e.g.*, memory, file descriptors, buffering) than shared-address-space modules. When deployed at scale, all these overheads translate to prohibitive financial costs.

5.2.3 Language-based MLC with Iris

Iris solves these challenges by taking a different approach: restricting the ability to *name* disallowed functionality. Rather than relying on the operating system to protect memory accesses, it "shadows" variable names from within the programming language. Names are assigned new, customized values that are augmented with interposition mechanisms. A module *thinks* it is accessing original values, when in fact it is accessing wrappers that verify accesses before propagating them to the original values. For example, the `event-stream` module cannot bypass Iris to name the application's top-level and global objects, cannot access the `fs` and the `http` modules which are namespaced from within the variable context, and cannot overwrite the `getKeys` field of a read-only `wallet`.

To meaningfully restrict the privilege of third-party modules, Iris allows developers to explicitly specify privilege when importing code—potentially, right at the `require` statement. Privilege specification takes the form of function predicates, PICs, that can guard any module imports—be it explicit like `fs` or implicit like `globals`. PICs operate at a very fine granularity—that of individual fields of individual objects that form the context of individual modules—and can be Turing-complete functions. For example, a developer can express that the `event-stream` module can use the built-in `require` function as long as it only imports `console` (for writing to the standard output stream).

Allowing PIC specification at such a fine granularity is necessary, as neither static nor dynamic analysis are powerful enough to deal with the most general case of malicious modules (see §5.2.1, ¶3). Iris improves MLC by not breaking semantics at the boundary, by not altering module interfaces, by enabling fine-grained control over module privilege, and by achieving order-of-magnitude performance improvements for single-module containment.

5.3 Threat Model

The general model of threats arising from the pervasive use of third-party modules is quite broad (§5.3.1). To be able to protect against these threats, Iris needs to place trust in a few underlying components (§5.3.2).

5.3.1 Threats and Goals

Iris protects against (i) attackers sending vulnerable modules input that triggers unintended actions (*e.g.*, call a method to access a secret key); (ii) malicious modules indirectly coercing other modules into performing unintended actions (*e.g.*, use introspection to get a secret key); and (iii) malicious modules directly attempting unintended actions (*e.g.*, exfiltrate a secret key directly).

Threat Classes Broadly, Iris protects the confidentiality (*e.g.*, read global state, load other modules, exfiltrate data) and integrity (*e.g.*, write global state or tamper with the module cache) of data and code. These extend to the broader environment within which the application is executing, including environment variables, hardware counters, the file system, or the network. Broadly, "ambient" overprivilege enabling such attacks comes from: (i) common features in programming languages (*e.g.*, call stack inspection, reflection capabilities, monkey patching); (ii) unusual language features or deficiencies (*e.g.*, in JavaScript: default-is-global, prototype poisoning, mutability attacks); (iii) implementation-specific concerns (*e.g.*, module cache, import capabilities); (iv) authority confusions, where parts of the application have equal access rights (*e.g.*, read `process.env` or `process.args`, write to the filesystem or network).

Security Goal Iris aims to mitigate the aforementioned attack vectors that enable "ambient" overprivilege, by allowing users to create compartments, controlling module-to-module access at a *very fine* granularity—that of individual names, functions, and fields of individual module return values.

5.3.2 Assumptions and Limitations

Iris places trust in the language runtime and built-in modules such as `fs`. Although it can confine built-in modules, a minimum of trusted functionality is needed from the module system to locate and load contracts. Moreover, when Iris is introduced as a module, it is assumed to be loaded *before* any other module—otherwise, a malicious module could dynamically rewrite Iris's code.

Limitations Iris does not handle native modules or denial-of-service (DoS) attacks. Native modules are written in lower-level languages such as C, C++, or assembly, and can bypass any security guarantees provided by the higher-level programming language and enforced by the runtime; any operation that violates memory safety (*e.g.*, pointer creation) would allow arbitrary modules to bypass Iris's language-based protection techniques. DoS attacks attempt to prevent legitimate use of a module by overloading it (*e.g.*, sending many requests, or few carefully crafted ones; infinite loops); weakening of DoS attacks by automatically scaling out would not work in Iris's single-process environment. For these two classes, developers can use (i) OS-enforced MLC systems [32, 224] and pay the aforementioned costs for a small fraction of the codebase, or (ii) a system targeting that particular class (*e.g.*, NaCl [245] for native code; DeDos [52] for DoS attacks).

5.4 Interface Privilege Contracts

Iris's goal is to reduce the privilege individual third-party modules possess. As a first step, it arms developers with the ability to explicitly specify a subset of the rights granted to modules by default upon import. This ability is available on a per-import basis, through each module's so-called *interface privilege contract* (PIC).

A key observation behind Iris is that even functionality that is accessible by default can be modeled as being provided by implicitly imported modules (§5.4.1). PICs are expressible at a very high granularity (§5.4.2), using a subset of the source

Table 5.1: **Implicit imports.** Examples names resolving to a scope outside that of a module: (i) core built-in objects, (ii) the standard library, (iii) implementation-specific objsects, (iv) module-locals, and (v) global variables.

Source	Example Variables
Built-in Objects	`Object`, `Function`, `Array`, `eval`, `parseInt`, ...
Standard Library	`Math`, `Number`, `String`, `JSON`, `Reflect`, ...
Node globals	`Buffer`, `Process`, `console`, `setTimeout`, ...
Module locals	`require`, `_dirname`, `exports`, `_filename`, ...
Globals	`GLOBAL`, `global`, `Window`, ...

language (§5.4.3–5.4.5).

5.4.1 Explicit *vs.* Implicit Imports

Iris unifies two different (and broad) classes of mechanisms that modules can use to import functionality. Functionality available to modules is either (i) *explicitly* imported from other modules, or (ii) *implicitly* available through their context.

Explicit Imports Functionality can be imported with the use of an explicit `import` statement. The `import` statement returns a value—the general case of which is an object—that becomes available to the module's scope. For example, the `log` module returns an object with method fields `info`, `warn`, and `err`, of which only `info` might be needed.

Implicit Imports Functionality can also be available by default through (and shared with) the module's "outer" *context*. Examples of such implicit functionality (Tab. 5.1) include top-level objects and functions (*e.g.*, `process.args`, `eval`), functions to output messages (*e.g.*, `console.log`), and capabilities related to module-importing *itself* (*e.g.*, `require`, `exports`). Naming this functionality resolves to a scope outside that of a module, and is pervasively accessible from any point in the code. What ends up hitting such an implicit context depends on the language's variable name resolution[2] and represents ambient authority [137]: a module need

[2] For example, in some cases variables need to be prefixed with `global` (*e.g.*, Python) while

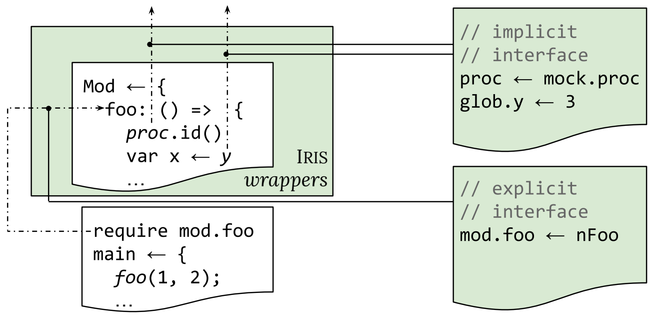


Figure 5.3: **Contract segments.** Cross-module variable name resolution (left) augmented with Iris (green boxes), which interjects non-bypassable steps resolving to Iris-augmented values (right-top: implicit module imports; right-bottom: explicit import).

only specify the names of the objects and methods associated with an operation in order to invoke it. Similar to explicit imports, implicit ones are accessible through the module's return value.

5.4.2 Specification Granularity

A PIC restricts the privilege available to a module m (or set of modules) by refining privilege across all the fields across all import values accessible by m. Privilege refinement is achieved through predicates that need to hold for an access to take place; semantically, an access is allowed only if a predicate evaluates to `true` when the access is attempted at runtime.

To make it easier to describe PICs and associated transformations (§5.5), PICs follow the distinction between explicit and implicit imports (Fig. 5.3). The part of a PIC targeting explicit imports is termed the *explicit segment*, and defines how a module's interface is used by its consumers. The part of a PIC targeting implicit imports is termed the *implicit segment*, and defines what implicit functionality it imports. A module's explicit segment may be different every time the module is imported, whereas its implicit segment does not generally change. Examples of PIC

in others resolution is attempted on increasingly outer scopes until it hits the global scope (*e.g.*, JavaScript).

segments include “can only write the first element of an imported array” (explicit) and “can only read the `PWD` environment variable” (implicit).

The combination of modules and PICs results in a new mental model, in which developers think of `import` as an augmented operation that supports privilege specification. Upon import, they can attenuate the privilege of individual modules by making privilege explicit at a very fine granularity. Under this extended model, Iris can be thought as an “object-path” protection service: access rights are expressed as privilege associated with a path from the program’s module roots to the field currently being accessed. While implicit imports are part of the model, module-local and function-local values are *not*: Iris does not allow specifying (or enforcing) access restrictions on, say, arbitrary objects or function return values. Whenever such restrictions are needed, values have to be wrapped in and imported from standalone modules.

5.4.3 PIC Expressions

PICs can be expressed using a subset of the source language, enough to map aforementioned object-paths of a module’s return value to access predicates. They can be specified directly at the point where a module is imported (*e.g.*, `require("ev-stream", `σ`))` or as part of an auxiliary PIC specification file (*e.g.*, `pic.json`). The former offers convenient control over individual imports, allowing developers to refine privilege on individual modules they import—possibly while developing the application. The latter is intended to simplify interaction with automated tools built atop Iris.

Privilege refinement is expressed via function predicates over the type of the attempted access. They take as input the values associated with the access and return a boolean, `true` for allowing access to go through and `false` for denying access. Input parameters to predicate functions for values that are read, written, and deleted include (i) a pointer to the target object, (ii) the name of the field to be accessed, and (iii) for writes, the new value. For executable values and constructors,

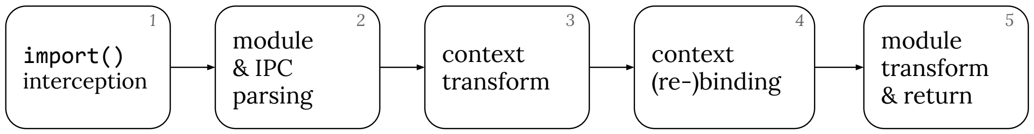


Figure 5.4: **Transformation pipeline.** Module loading is augmented with several stages that transform the module and its context before returning its value to the importing code. Stages (1), (3), and (5) manipulate in-memory objects; stages (2) and (4) operate on source code (*Cf.*§5.5).

input parameters include (i) a pointer to the target object (ii) the implicit "self" argument for method calls, and, most importantly, (iii) the list of arguments provided for the call. Predicate functions are Turing-complete, in that they can run programs of arbitrary complexity. In practice, however, they tend to be purely functional computations over the aforementioned inputs.

PICs work in conjunction with the existing language restrictions and protection mechanisms. The privilege associated with a value depends on the value's type. Specifically, it is meaningful only in the context of accesses related to that type, and can be thought as further constraining over the language's base types. Conversely, to retain the full capabilities of the language, a right to perform an access does not place any constraints on the type or form of the new value.

5.4.4 Example PICs

To illustrate the use of PICs in practice, consider the Bitcoin application presented in §5.2.1 (Fig. 5.2).

Example 1 The first privilege we might want to constrain is the use of `require`: simply granting a module all-or-nothing privileges over `require` unleashes too much power, as it then allows the module to import *any* module into the current scope. A PIC can constrain the import to *only*, say, `log`. This is accomplished with the following PIC for `event-stream`:

```
event-stream:                                                    (p15)
  globals:
```

```
3     require: (m) => m === "log"
4 [...]
```

During execution, the anonymous function on line 3 will be provided the arguments to `require`, of which it binds only the first—corresponding to the module's name. It returns true if it matches "log", allowing execution to proceed uninterrupted.

Example 2 For a more advanced use, consider a predicate that only allows reads and writes to files that belong in a certain directory. Module `event-stream` is provided a privilege-refined version of `fs`'s `readFile` method that can only access the "/var/www" directory subtree. This is accomplished with the following PIC:

```
event-stream:                                              ($p_{16}$)
  fs.readFile: (f) =>
    path.resolve(f).startsWith("/var/www")
[...]
```

The `resolve` method of the built-in `path` module returns an absolute pathname; subsequent prefix-matching returns a boolean value. This predicate simulates heavy-weight process jails [95] and container [129] namespacing, but goes beyond all-or-nothing configurations by allowing different modules to access different directories. This works because PICs specify privilege for *individual* modules: a different module importing `fs`, such as `log`, could restrict `fs` access to the local directory.

5.4.5 Subtleties

The use of predicate functions could open several subversion vectors. To avoid this problem, Iris restricts the language of PIC expressions, leveraging the techniques described in this paper. Casual readers may want to skip this subsection on a first read, as full understanding of the techniques presented here requires reviewing the enforcement section (§5.5).

The first problem is that importing third-party modules in predicates could open

```
wrap (e: Value) : Value := match e with
  | {(s, v) :: vs}  →  {(s, wrap v) :: wrap vs}
  | [v :: vs]       →  [(wrap v) :: wrap vs]
  | λ(...args).f    →  λ(...args).{ σ(e, args)? f : () }
  | _               →  interpose(σ, e)
end
```

Figure 5.5: **Base transformation.** The algorithm (simplified) is presented in functional style to simplify variable binding; types, whose structure is used for pattern matching, are shown in light *turquoise* (*Cf.*§5.5.2).

the doors to malicious code within the PICs, foregoing any of Iris's benefits. To solve this, Iris disallows the use of `require` within PICs. Instead, by using context transformation and rebinding (§5.5.4), it only permits access to built-in modules directly by *name*. For example, instead of `require("path")`, path's functionality is available directly through the `path` variable.

Even while restricting to built-in modules only, issues remain: asynchronous, non-blocking functions return before the results of the checks are made available. As a result, a sensitive resource can be accessed before the corresponding PIC has any time to be enforced. To solve this, Iris allows calling only synchronous, blocking interfaces of the built-in modules. This is achieved by guarding the privilege to read non-blocking interface with a PIC that disallows access.

A last problem is related to PIC side-effects. PICs may need to mutate state outside the PIC; this is useful, for example, to enforce access to an interface only once or within a certain timeframe. Moreover, state must be shareable between PICs to allow coordinated decisions across PICs. However, state shared between PIC- and application-resident code could lead to problems because malicious application code could attempt to violate confidentiality and integrity of state on which PICs depend (*i.e.*, bypassing the two prior techniques). To solve these problems, Iris introduces a custom global context that is only available to PICs and isolated from the `global` context of the application. The map can be shared across PICs, and stores data that persist across executions of a single PIC.

5.5 Monitoring Transformations

To reduce the privilege modules possess, Iris needs to enforce the privilege granted to modules upon import by developers during the execution of the program. This is achieved by a series of automated runtime transformations that wrap modules with security monitors. Each monitor is responsible for overseeing accesses that cross a single module-to-module boundary and ensuring that they conform to the privilege specified by the accompanying PIC. If a violation is detected, Iris throws a special exception that includes contextual information for diagnosing root cause (§5.5.1).

To transform a module, Iris augments several steps of the module loading process (Fig. 5.4) using a base transformation that wraps an object with a security monitor (§5.5.2). It starts by intercepting calls to the `require` function (§5.5.3), which locates the module's source code and PIC. It then constructs a custom context based on the PIC's implicit segment (§5.5.4). It then encloses the module in a closure to leverage local variable resolution in order to link the custom context with the module (§5.5.5). After source interpretation, Iris transforms the resulting value by consulting the PIC's explicit segment (§5.5.6), updates the Iris-augmented module cache (§5.5.7), and returns the value to the consumer. The following subsections detail these steps.

5.5.1 Privilege Exceptions

The semantics of the augmented model guarantee that a module field will be accessed only if the accessing module is allowed to do so and if the access conforms to the field's predicate; if not, the Iris wrappers throw an access violation. To expose an access violation to the user, Iris introduces a new exception, `PrivilegeException`, that bypasses the program's control flow to notify the user and provide contextual information. This information is intended to simplify diagnosis of the violation and its cause, a challenge compounded by Iris's chosen domain—large dependency graphs with many third-party libraries of which the developer has little understanding. Specifically, the exception includes the type of violation, names of the modules involved, names of accessed functions and objects, and a stack trace.

While Iris guarantees that unauthorized access will not be allowed, a malicious mediating module can still catch and silence an `PrivilegeException`. This does not violate the semantics of the privilege refinement model, but (at worst) may complicate debugging. To simplify this, Iris can be configured to execute in a log-only mode, in which it logs exceptions (high I/O overhead), or a fail-stop mode, in which it prints contextual information to the error stream and immediately stops execution.

5.5.2 Base Transformation

Iris's transformations boil down to a base form `BT` that augments *objects* with runtime security monitors. At a high level, `BT` takes an object O and a PIC σ and returns a new object O'. Every field f of O is wrapped with a method f' defined to enclose f. At runtime, f' checks σ: if the access is allowed, it forwards the call to f; otherwise, it raises an exception.

To achieve this transformation, Iris walks the object graph from the root and processes component values based on their types (Fig. 5.5). Rather than mutating original values, it copies them, applies transformations on the copies, and returns copies to the caller:

- *function* values are wrapped by a closure that forwards arguments to the original function if allowed to do so.

- *object* values are recursively transformed, with their getter and setter methods replaced similar to function values.

- *primitive* values are copied unmodified and wrapped with an access interposition mechanism.

```
let Math = {
 add: (a, b) => {
  return a + b;
 },
 sub: (a, b) => {…},
 …
 …
}
```

```
let _M = Math, Math = {};
Math.add = (…args) => {
if (σ(_M.add, args)) {
 return _M.add(args);
else
 throw Iris.PrivException();
};
Math.sub = (…args) => {…};
…
```

Figure 5.6: **Example base transformation.** Applying the base transformation (*Cf.*§5.5.2) to a `Math` object that contains an `add` method yields a new `Math` object. The new `add` is a closure over the old `add`. When called, it passes arguments to and invokes the old `add` if the original add should be called; otherwise, it throws an `ACException` (*Cf.*§5.5.1).

Direct field accesses, such as assignments, require custom detection upon dereference. To achieve this, Iris wraps fields with an interposition mechanism, essentially treating direct field accesses as function calls. Examples of such mechanisms include `Proxy` objects for JavaScript, metatables for Lua, metaclasses for Python, and direct-accessor metaprogramming in Ruby. Iris's wrappers detect and record changes to any of the object's fields; nested wrappers monitor nested objects.

For further processing, Iris maintains a handle to the root of both the unprocessed and the newly processed values. The unprocessed value is used to create objects that check for different PICs. The new value is used to revoke or alter privilege at runtime, a capability not explored further in this paper.

Fig. 5.6 shows the result of applying the base transform `BT` on simple `Math` object.

5.5.3 Import Interception

Iris's runtime enforcement component is introduced as a backward-compatible, drop-in replacement of the language runtime environment's *module system*—either as an application-specific module (*e.g.*, `iris` package) or bundled with a custom runtime replacing the language's module system for all applications (*e.g.*, Iris-powered Node.js). It always starts by dynamically replacing `require`: instead of simply locating and loading a module, the function yields to Iris.

```
let CONTEXT = {
require: null,       //module-local; null for now
process: Iris.BT(process, σ₁),
setTimeout: Iris.BT(setTimeout, true),
Number: Iris.BT(Number),
Array: Iris.BT(Array),
// […another 150 entries…]
}
```

Figure 5.7: **Creating a custom context.** A custom context mapping variable names to values is created by applying the base transformation (Fig. 5.5) on individual values of the default context (*Cf.*§5.5.4).

Iris checks (i) if the module has already been loaded, and, if it has, (ii) whether it has ever been loaded with the same PIC. If both statements are true, it recovers and returns the transformed module value. If the module was loaded with a different PIC, Iris loads the appropriate PIC and applies a transformation pass on a cached copy of the module (§5.5.7). Otherwise, Iris first invokes the built-in module loader to locate the module.

The process of loading new modules includes a phase of reading the necessary files as *source* and a phase of interpreting them, interspersed by applications of transformations. Reading the source files returns a string representation of the source code. Interpretation uses the language's runtime evaluation primitives to convert the code to an in-memory object. A series of transformations is applied before and after interpretation.

For PICs, this distinction is not as important, other than to note that PIC specification files are loaded as strings. The PIC specification language is embedded in the source language, thus Iris makes use of the language's built-in evaluation primitives to interpret the PIC specification file. The result is identical to PICs specified as arguments to the `require` method.

5.5.4 Context Transformation

Iris needs to prepare a transformed copy of the module's outer context, a map from variable names to their values. To achieve this, it creates an auxiliary hash table mapping names to transformed values. Names correspond to implicit modules (§5.4) such as globals, built-ins, module-locals, *etc.* (Tab. 5.1). Transformed values are created by applying the base transformation `BT` over values in the context, as specified by the PIC's implicit segment (Fig. 5.7).

While user-defined global variables are stored in well-known locations,[3] traversing the global scope for built-in values is generally not possible. To solve this problem, Iris collects such values by resolving well-known names hard-coded in a list; different lists exist for different environments and versions of the language. Using this list, Iris creates a list of pointers to unmodified values.

Care must be taken with module-local variables, which refer to information associated with individual modules. These are accessible from anywhere within the scope of a module (similar to global variables), but each module refers to its own copy of these variables. Examples include the module's absolute filename, its `export`ed values, and whether the module is invoked as the application's `main` module. Attempting to access them directly from within Iris's scope will fail subtly, as they will end up resolving to module-local values of Iris *itself*—and specifically, the module within Iris that is applying the transformation. Iris solves this problem by deferring binding for later, and specifically from within the scope of the module. This process is detailed in the next section (§5.5.5); for now Iris leaves the entries empty.

Fig. 5.7 illustrates the context transformation for the `flatmap-stream` module (Fig. 5.2–bottom), based on the PICs defined in its implicit segment (§5.4.4).

5.5.5 Module Enclosure

Iris needs to link the module with the newly transformed version of its context. To achieve this, it wraps the module with a closure that starts by redefining and enclosing global variable names as module-local ones (Fig. 5.8). The closure accepts

[3] For example, `globals` in JavaScript and `_G` in Lua.

```
 1(cxt) => {
 2 let require = Iris.BT(require, σ0);
 3 let process = ctx.process;
 4 let setTimeout = ctx.setTimeout;
 5 let Number = ctx.Number;
 6 let Array = ctx.Array;
 7 // […another 150 entries…]
 8   let s = require("stream");
 9   exports = (m, o) => {…}
10   (flatmap-stream code)
11}
```

Figure 5.8: **Module enclosure.** The original module (shaded lines 10–19) is wrapped with a function (lines 1–8 and 20) that (i) takes the custom context (Fig. 5.7) as a parameter, and (ii) shadows globals (2), language built-ins (3), and module-locals (4) by re-defining them as function-locals (*Cf.*§5.5.5).

as argument the customized context and assigns its entries to their respective variable names. This is arranged in a preamble comprising of assignments, which executes before everything else in the module. When the closure is applied to the customized context, the module's return value is recovered in a side-effectful manner (as in the unmodified module system) by reading the module's `exports` variable—that is, the closure does not end with an explicit `return`.

This technique leverages lexical scoping to inject a non-bypassable step in the variable name resolution process. Instead of resolving to variables in the context, resolution will first "hit" module-local values augmented with security monitors. As a result, even if new code is runtime-evaluated at a later point, it will still be constrained to modified values by the language's resolution algorithm. The technique works because the module is still at a loading stage prior to interpretation, represented as a string. It is a limited form of source code rewriting, special in that it conveniently occurs at runtime.

Late-bound, module-local variables (§5.5.2) are the result of applying `BT` over variable names in the current scope, which is now bound to the correct module-local value.

```
1 let compiled = [native].eval(enclosed);
2 let rewired = compiled(CONTEXT);
3 let bucket = require.cache["flatmap-stream"];
4 bucket[σ.implicit] = rewired;
5 let instance = Iris.BT(rewired, σ.explicit);
6 bucket[σ.implicit][σ.explicit] = rewired;
7 return instance;
```

Figure 5.9: **Module linking and export.** After interpretation, the wrapped module (Fig. 5.8 is linked to the customized context (Fig. 5.7) by function application (*Cf.*§5.5.6); the return value is cached before (4) and after (6) the final transformation responsible for attenuating the return value (*Cf.*§5.5.7).

5.5.6 Module Transformation and Return

Returning the module's value to its consumer comprises interpreting the module, linking it with the custom context, and applying transformations to its return value (Fig. 5.9).

When interpreted, the source code returns (the in-memory representation of) a function. Passing the custom context to the function evaluates the module. During this phase the module may attempt to access global objects and cause side-effects, but all these are subject to the privilege granted to the module. the module can "see" only a limited subset of interfaces throughout its lifetime. Evaluation returns the module's exported values, similar to the vanilla module system.

Right before returning the exports to the consumer, IRIS runs the return value through a final application of `BT`, consulting the privilege specified by the PIC's explicit segment (Fig. 5.9, lines 1, 7). At the compartment boundary between `event-stream` and the imported `flatmap-stream`, only the `map` field should be accessible; the base transformation makes all other fields of the object inaccessible.

5.5.7 Augmenting The Module Cache

Applications may import the same module at different points of the dependency graph. For consistency and performance purposes, module systems maintain a cache of loaded modules. When an already-loaded module is imported again, vanilla module systems return a cached reference to the return value of the original module. The new privilege-augmented model, in which the same module can be loaded multiple times with *different* privilege, requires augmenting the module cache and associated operations (§5.5.3).

Iris extends the cache with two additional levels. The first level is indexed by module identifiers as before, the second level is indexed by implicit segments, and the third level is indexed by explicit segments. For each module, the second level contains a collection of entries corresponding to mostly-transformed modules, and the third level contains fully transformed modules. Mostly-transformed modules have gone through the entire transformation pipeline except for the last stage: they have been interpreted and have had their context transformed and linked, but their return value has not been processed to enforce the explicit segment (§5.5.6). The reason for splitting the two levels is that modules are usually governed by a single implicit segment but multiple explicit segments, one for each of its consumers. A context transformation is applied only a few times, whereas a return-value transformation is applied on every `require`. The second level also contains a special entry (indexed by `"_"`) for the non-transformed return value, so that subsequent transformations can skip loading the module from disk.

When a module is already loaded, Iris indexes by implicit segment to retrieves the right module instance. It then applies the transformation required by the explicit segment. New modules are inserted into the second-level cache right after evaluation of the interpreted function (line 3 of Fig. 5.9).

5.6 Summary

This chapter presented Iris, a system for solving key challenges in module-level compartmentalization: the introduction of unsoundness, manual synchronous-to-asynchronous conversions, coarse-grained privilege control, and significant performance overheads. Iris addresses these challenges by replacing OS protection mechanisms with ones provided by the programming language. Rather than relying on the OS to restrict syscall interface accesses at the compartment boundary, it restricts (naming) functionality from within a compartment. This ability is available as fine-grained executable access predicates termed *privilege-interface contracts* (PICs). A series of techniques for interjecting, transforming, and rebinding the module's context introduces non-bypassable security monitors that intercede at the compartment boundaries and enforce PICs during program execution.

The nest chapter (§6) expands on Iris' techniques to automatically scale out bottlenecked modules.

Chapter 6

Light-Touch Scaling of Distribution-Oblivious Code

One of ANDROMEDA's key arguments is that a high-level, dynamic programming style (and associated runtime) should lower However, a challenge that arises is that performance of high-level languages may be lower than carefully handwritten C or assembly code [47]. There are mainly two approaches for minimizing such a performance disparity—and ANDROMEDA uses both of them. The first approach is to leverage a high-performance runtime, with state-of-the-art machinery (*e.g.*, optimization, JIT compilation) that has been the focal point of non-trivial industrial engineering investment (§7). The remaining loss is insignificant compared to the benefits of productivity and safety, and can be mostly recouped by re-engineering or scaling out bottlenecked components—the second approach. We tested the latter hypothesis of directly exploiting distribution by developing IGNIS [222], a framework that leverages ANDROMEDA's transformation infrastructure (§2.4.1) to detect and scale out bottlenecked modules in legacy applications, as long as their developers have sprinkled selective imports with soundness-related annotations.

Light-touch distribution is a new approach that converts a legacy system into a distributed one using automated transformations. Transformations operate at the

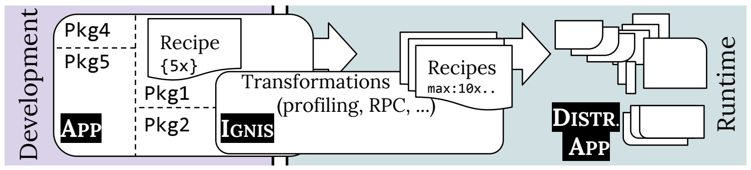


Figure 6.1: **Schematic of light-touch distribution.** Module structure is used at runtime to automatically scale systems out, guided by recipes (*Cf.*§6.1).

boundaries of bottlenecked modules and are parametrizable by light distribution recipes that guide the intended semantics of the resulting distribution. Transformations and recipes operate at runtime, adapting to load by scaling out only saturated components. Our Ignis prototype shows substantial speedups, attractive elasticity characteristics, and memory gains over full replication, achieved by small and backward-compatible code changes.

6.1 Broader Motivation

Distributed systems offer notable benefits over their centralized counterparts. Reaping these benefits, however, requires burdensome developer effort to identify and rewrite bottlenecked components. This is why the majority of developers starts by developing and deploying software in a centralized manner—that is, *until* there is a significant change of requirements, such as a load increase.

When this happens, developers try to identify affected parts of the system and manually re-write them to exploit distribution. The scope of such rewrites, and therefore the cost of manual effort, can vary considerably. Often, they only focus on a few parts of the system—for example, upgrading to a distributed storage layer. More rarely, companies rewrite entire systems (*e.g.*, Twitter's Ruby-to-Scala rewrite [126]), a process that is notoriously difficult under schedule constraints and competitive pressures [202, 250]. The manual effort is expensive, and can introduce new bugs, cascading changes, or regressions of previously fixed performance issues, especially since software today makes extensive use of third-party modules [223]. Could the

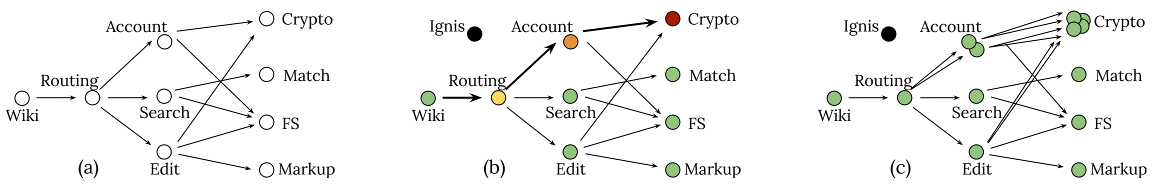


Figure 6.2: **Case study.** Wiki module relationships (a) monitored continuously at runtime to identify bottlenecks (b) and automatically scale them out (c) (*Cf.*§6.3).

process of identifying bottlenecks, generating a distributed version of the system, and scaling it out at runtime be *significantly* automated?

The core insight behind this work is that, instead of manually building scalability into the system, valuable human effort should only be spent on instructing the system *how* to scale. As long as developers have sprinkled the program with hints, it should *automatically detect* and *dynamically adapt* to load. We term this automation and associated control-plane hints *light-touch distribution* (Fig. 6.1).

Light-touch distribution involves two components: (i) automated programmatic *transformations* that operate at module boundaries for detecting and adapting to load; and (ii) distribution *recipes*, lightweight annotations that guide the semantics of the resulting distributed application. Transformations automate most of the process, but depend on recipes for key semantic decisions that affect soundness. Both transformations and recipes operate at runtime, which offers significant benefits: applications can respond dynamically to increased load by scaling out, can selectively replicate saturated components instead of whole applications, and can avoid over-provisioning by scaling back when load subsides.

Light-touch distribution occupies a known middle-ground [27, 122, 224] between flexibility and automation (§9.5), and is enabled today by a confluence of trends in software development—namely, the increasingly pervasive use of (i) dynamic, interpreted languages, (ii) fine-grained modules with clear boundaries, and (iii) cooperatively concurrent, continuation-passing programming styles (CPS). Examples of such environments include JavaScript, Julia, and Lua; our IGNIS prototype targets server-side JavaScript (§7.2).

We begin with an example of applying light-touch distribution (§6.2), and continue with an overview of Ignis (§6.3). §6.4–6.6 highlight our key contributions:

- §6.4 introduces *load-detection transformations* that collect windowed statistics about load at each module boundary. Control-plane coordination with a global view of load and available resources helps decide when to initiate scale-out of a bottlenecked module.

- §6.5 presents a set of parametrizable *distribution transformations* that transparently scale modules out. These transformations can create module replicas, hook communication channels among them, schedule requests, and forward side-effects such as mutation and collection of memory.

- §6.6 outlines *distribution recipes*, lightweight annotations that guide the semantics of the resulting distribution. They offer significant flexibility by parametrizing transformations, including tuning state management, replication consistency, event propagation, and colocation preferences.

Ignis' implementation, its evaluation using a combination of micro-benchmarks and real systems, comparison with prior work, and application of Ignis's ideas in other environments are presented in the implementation (§7.2), evaluation (§8.5), related-work (§9.5), and discussion (§10.4) chapters, respectively. Appendix B provides technical background on the module system's internals, to ensure that Ignis's transformations are appreciated by a wider audience.

6.2 Background and Motivation

We use a wiki engine to illustrate difficulties in scaling out applications (§6.2.1), outline light-touch distribution (§6.2.2), apply Ignis to alleviate the aforementioned difficulties (§6.2.3), and outline the trends that enable this approach today (§6.2.4).

6.2.1 Case Study: A Wiki Engine

Fig. 6.2*a* shows the (simplified) module structure of a wiki engine [70]. Modules—development-time constructs usually glued together without a full understanding of their internals—are represented as vertices. The resulting dependencies, which in modern applications can be thousands [223], are depicted as edges connecting importing *parent* modules with imported *child* modules.

A sharp increase in sign-in attempts can saturate the `account` module. Logically unrelated parts of the system competing for the same resource (*e.g.*, CPU), such as document `edit`ing and `search`ing, will also be affected.

Developers use various techniques to understand such problems. For example, collected traces can be replayed against off-line versions of the system and statistical profiling can identify hot code-paths. These techniques, however, require some degree of *manual* effort: capturing traces, setting up testbeds, replaying traces, analyzing statistics, and debugging performance are all tedious and time-consuming tasks. Pervasiveness of third-party modules and heavy code reuse in modern applications compound the challenge, as the causes may lie deep in the dependency chain.

Detecting bottlenecks is not easy, but its effort is dwarfed by that of rewriting parts of an application to exploit distribution. Extensive code changes, orchestration of multiple jobs, service discovery, and scheduling over multiple replicas are all difficult and error-prone tasks, and must be repeated for every new bottleneck.

Light-touch distribution attempts to automate as much of this process as possible without requiring development in a new programming language or model.

6.2.2 Light-touch Distribution with Ignis

Ignis detects and scales out bottlenecked components by interposing on the application's module boundaries. It is introduced as a backward-compatible, drop-in replacement of the language's module system. It can be imported as an application-specific module (*e.g.*, `ignis` package) or can be bundled with a custom language run-

```
1 let pbkdf2 = require("crypto").pbkdf2; //module
2 let slt = users[usr].salt; // usr, pswd via form
3 let h = users[usr].hash.toString();
4 pbkdf2(pswd, slt, 10000, 512, (e, d) => {
5  (h == d)? resp.send(200) : resp.send(401);
6 });
```

Figure 6.3: **Example bottleneck.** `pbkdf2` at the `account`–`crypto` boundary makes the `crypto` module a good candidate for scale-out (*Cf.*§6.2.3).

time (*e.g.*, Ignis-powered Lua) replacing its system-wide module system. It starts by dynamically replacing the import function: instead of simply locating and loading a module, the function yields to Ignis, which applies a series of transformations to modules with the goal of interposing on their boundaries.

Transformations depend on several configuration details related to recipes, but can be coarsely grouped into three broad classes: (i) profiling and decision-making, (ii) spawning and distribution, and (iii) single-system retrofitting. Profiling transformations build a statistical model of module pressure (Fig. 6.2*b*) and rank candidate modules. Distribution transformations replicate bottlenecked modules, create communication channels among them, and balance load across all replicas (Fig. 6.2*c*). Single-system transformations selectively back-port the semantics of a single runtime. To guide the intended semantics (and associated trade-offs), developers annotate transformations with optional distribution recipes.

6.2.3 Ignis-powered Wiki Engine

To show how to apply Ignis on the performance problem outlined earlier (§6.2.1), Fig. 6.3 zooms into the authentication section of the `account` module: it imports the built-in `crypto` module (line 1) and invokes `pbkdf2` (4) which, upon completion, calls a provided continuation function (5). Ignis augments `require` (1) to return a wrapper of `pbkdf2`. The wrapper monitors `pbkdf2`'s calls at the `account`–`crypto` boundary. Upon load increase, it identifies `pbkdf2` as a bottleneck and marks `crypto` as a candidate for scale-out.

To scale out, IGNIS launches a few fresh replicas of the `crypto` module and starts spreading remote procedure calls (RPCs) among them. RPCs require serializing arguments, sending them to one of the remote replicas, and calling `pbkdf2` there. Results are sent back to the `account` module, which passes them to the provided continuation.

IGNIS also augments `require` to take a recipe as an additional, optional argument. A recipe σ at `require("crypto", `σ`)` (1) would constrain `pbkdf2`'s scale-out. For example, a σ equal to {`order:true`} would have forced ordering semantics on calls and their results across all `crypto` replicas. Luckily, `pbkdf2` is a pure function, which can be determined even in the complete absence of annotations. This exemplifies a case where light-touch distribution can obtain benefits even without any developer effort.

6.2.4 Simplifying Trends

Light-touch distribution is significantly simplified by the increasingly pervasive use of certain features today.

Packages and Modules Modules provide an implicit and fine-grained component architecture [60, 67, 147, 85] that applications can be partitioned across [89, 247, 185, 46, 148]. They encapsulate state behind small and tight interfaces, simplifying and minimizing transformations. Their boundaries clearly mark self-contained components that can be configured to execute remotely—including built-ins, such as `crypto`. Multi-thousand-module dependency graphs enable profiling and decomposition at a very high resolution, aiding bottleneck detection and memory consumption at scale.

Programming Styles Today's popular programming styles blur the line between local and remote execution. Cooperative concurrency grants scheduling control to the executing code, event-driven programming is naturally message-passing, and continuations enable (hidden) parallelism: independent continuations do not impose ordering constraints—two continuations λ_1 and λ_2 of sequenced calls $f_1(\ldots, \lambda_1)$; $f_2(\ldots, \lambda_2)$ can be interleaved in any order (otherwise, f_2 would have been included in λ_1). As such, components can be distributed across multiple nodes, even if there is no underlying single system image [111, 116, 25, 38, 17, 12, 251].

Dynamic Language Interpretation Dynamic languages have features—*e.g.*, name (re-)binding, value introspection, dynamic code evaluation, and access interposition—that enable runtime transformations [99, 100]. They conveniently unify module identification with interposition: a single function or function-like operator locates a module, interprets it, and applies transformations before exposing its interface in the caller context. As a result, monitoring and distribution can be performed at runtime and without forcing users into specific programming models [49, 146, 248, 145, 168].

6.3 System Overview

This section presents an overview of Ignis (§6.3.1) and outlines the structure of transformations (§6.3.2).

6.3.1 Transformations vs. Recipes

Ignis' responsibilities are divided between transformations and recipes, similar to the separation of mechanism and policy in the operating systems literature [112]. Transformations provide the mere mechanism for automating profiling and distribution, including creating remote references, copying structures, propagating events, *etc.* Semantics-related concerns are offloaded to recipes—*i.e.*, policies that encode developer knowledge about the behavior of modules.

Distribution Recipes Developers start by annotating selected imports with distribution recipes. Recipes are declarative runtime configuration objects, expressed using a domain-specific language embedded in the source language. They declare the intended semantics of the resulting distribution, tuning trade-offs that are fun-

```
p  ::=  s ∈ String  |  n ∈ Number  |  b ∈ Bool  |  ∅
v  ::=  p  |  (x,...)  =>  {e}  |  {s:v,...}  |  [v,...]
e  ::=  x  |  v  |  (x  =  e)  e  |  e(e)  |  e[e]  =  e
```

Listing 6.1: **Module interface language, used in transformations.** Modules return non-primitive values v, manipulatable via expressions e.

damental in distributed systems [71, 3, 120]. For example, calls to a module may need to maintain ordering and changes to a module's state may need to be reflected across its replicas.

Profiling Once provided with a few recipes, IGNIS starts monitoring the performance of the corresponding modules in order to detect opportunities for distribution. A key observation is the semantic isomorphism between calling a function and passing a message [205, 109]. This allows viewing a series of calls as a stream of messages. Module boundaries can be viewed as (virtual) queues of messages that await processing. Overwhelming a module causes its ingress queue to grow. At some point, the waiting time of newly-arrived messages becomes longer than the time to send the messages to a remote copy of the module, run the call there, and return the results back to their intended recipient.

Scaling Out Once this point is reached, IGNIS attempts to scale out a module while selectively maintaining single-runtime semantics. Scaling out is achieved by spawning a module replica and replacing its local use with a thin client that disperses calls across all replicas. On each call, arguments are sent to a remote replica and results from the replica are returned to the thin client. The selection of which single-runtime semantics to maintain is tunable by distribution recipes, and implemented via additional transformations: converting local-memory pointers to meaningful distributed ones, forwarding side-effects such as memory allocation and collection, providing distributed versions of core built-in libraries, and enforcing ordering (when required).

6.3.2 Structure of Transformations

Transformations are used pervasively throughout Ignis, and are abstracted via a few parametrizable templates. Templates map different types of values (List. 6.1) to a generic handler for each type. Transformations have these handlers parametrized to achieve concrete goals such as monitoring, serialization, and scale-out. Simplified instances are described in the following two sections (§6.4–6.5).

Transformations can be applied to any value in the language, such as an object returned from a module or an exception about to be sent across the network. The general case of such a value is a directed acyclic graph (DAG). The types of its vertices can be coarsely grouped into primitives, functions, and objects. Objects map strings to other values, pointing to other vertices in the DAG. Transformations start by walking the DAG from the root vertex and processing component values based on their types. They do not mutate original values, but first copy them, apply transformations to the copies, and return copies to the caller. They only partially explore the object graph, as they do not peak through function closure environments. Fortunately, this aligns well with our goal of monitoring activity at module boundaries and ignoring module-internal activity.

As an example, consider transforming the `crypto` module (§6.2.3). Ignis traverses the object returned by `crypto` and replaces functions such as `pbkdf2` with wrappers whose specifics depend on the intended goal: profiling wrappers call the original function in between statistics collection, and RPC wrappers forward the call to a remote replica.

6.4 Decision-Making

The task of monitoring performance and detecting opportunities for scale-out is logically split into (i) a decentralized set of profiling agents that operate at module boundaries (§6.4.1), and (ii) a centralized coordinator that builds a holistic understanding of what—and when—to scale out (§6.4.2).

Profiling is accomplished by wrapping module interfaces with logic that generates a model of the current workload. Each module boundary collects its own statistics based on a combination of recipes and instructions from the coordinator. Profile generation can operate at a high resolution in time and space: (i) at every function call entering a module, and (ii) on thousands of modules across an application.

The coordinator oversees all profile-generation agents, collects periodic summaries from them, and ranks their needs. It is also responsible for creating a map of available resources and checking their status and health. While coordination operates at a lower resolution than profile generation, it allows monitoring some boundaries more closely than others.

The division of labor in deciding when to initiate scale-out is somewhat delicate. Agents running at the boundaries should not depend on the coordinator for online decision-making, as after a first scale-out they may be executing on different nodes. As such, they should have enough logic to make an online decision. To solve this, the coordinator pushes periodic guideline updates to the agents, which merge them with their local configurations (*e.g.*, module-specific recipes that override default/global recipes).

6.4.1 Profile Generation

Profiling transformations insert agents modeling queues at the module boundaries. Observing queue metrics such as arrival rate and wait time, agents build an understanding of the pressure applied at their boundary.

Transformations Profiling transformations focus on function (and function-like) values (Fig. 6.4), for which they generate and attach code that monitors calls and returns. Specifically, each vertex in the DAG returned from a module is recursively replaced with a wrapper:

```
ptf (e: DAG) : DAG := match e with
| Obj ((s, v) :: xs) -> Obj ((s, ptf v) :: ptf xs)
| Arr (v :: vs)      -> Arr ((ptf v) :: ptf xs)
| Fun f -> Fun (args) => {nq(); f(args); dq();}
| __                 -> toStats(e)
end
```

Figure 6.4: **Profiling transformation**. Functions are wrapped with prologue (`nq`) and epilogue (`dq`) operations that record statistics (*Cf.*§6.4.1).

- function values are wrapped in functions that add a prologue/epilogue pair recording profiling data.
- mutable values have their getter and setter methods similarly wrapped with prologue/epilogue wrappers.
- values of all other types are left unmodified.

IGNIS now mediates between parent and child modules. Wrappers record statistics about the their encapsulated functions as well as queue characteristics of outstanding calls.

In the cases of non-blocking (*i.e.*, asynchronous) interfaces, the prologue includes code for wrapping the continuation argument (*i.e.*, callback function) before passing it to the encapsulated callee; the continuation wrapper records metadata similar to the case of returns for blocking (*i.e.*, synchronous) interfaces. As function invocations may support re-entrant concurrency (*e.g.*, `fs` module), marks are added (§6.5.2) to match prologues with their respective epilogues.

Statistics The wrapper epilogue has access to several raw metrics related to profiling (Tab. 6.1). These metrics need to be composed into a model that allows IGNIS to decide whether to scale out a module.

A simple idea would be to detect when the number of concurrent requests ϱ exceed the number of replicas R. When this happens, assuming R does not exceed the number of CPUs P, IGNIS could start a fresh new replica:

Table 6.1: **Example metrics**. Module boundary agents see only (high-frequency) local metrics; coordinators receive weighted summaries but have end-to-end visibility across the system (*Cf.*§6.4.1).

Metric	Module Agent	Controller
Arrival Freq.	Call Freq.	Arrival Th/put
Queue Size	Queued Items	Avg. size
Processing Time	Downstream Latency	Lifetime
Call Type	Func/Method/Prop. Access	Call Ratios
Failure Ratio	Exceptions	Summary
CPU Number/Type	Locally available	Total
etc.	*etc.*	*etc.*

$$R_{new} = \begin{cases} R+1, & \varrho > R \wedge R < P \\ R-1, & \varrho < R \end{cases} \tag{6.1}$$

This approach omits a few important issues. First, we would like to model and account for the overheads of scaling out. These overheads involve context switching, round-trip times, and other systemic overheads δ, as well as one-off startup costs δ_0 when spawning a replica (§8.5). Second, as some operations (*e.g.*, slow I/O) are intrinsically concurrent, we would want to allow for at least some concurrency before paying the cost of scale-out. The available room for concurrency, however, is not visible at the level of individual boundaries; thus, agents can model a virtual queue by taking a windowed, weighted average of wait-times l_i:

$$R_{new} = \begin{cases} R+1, & \varrho \times \sum_{i=1}^{t} w_i l_i > \delta + \delta_0 \\ R-1, & \varrho \times \sum_{i=1}^{t} w_i l_i < \delta + a * \delta_0 \end{cases} \tag{6.2}$$

Scale-in (*i.e.*, -1) might not seem as important, but constrained environments benefit from quick re-allocation. However, after scale-in, the system often ends up just scaling out the same module. To avoid such oscillation, a small reclamation delay a increases the system's confidence that the workload has moved away from a specific pattern.

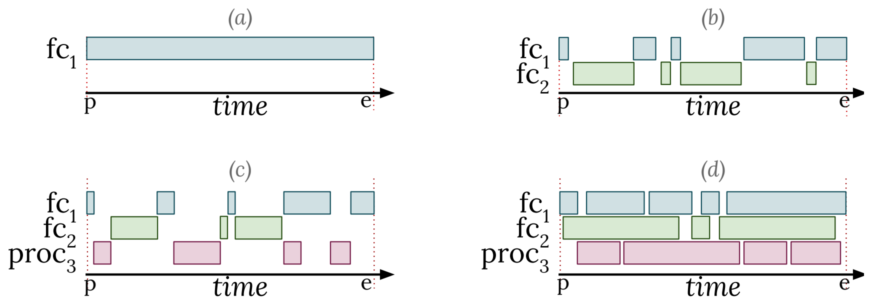


Figure 6.5: **Accounting accuracy**. Only observing the prologue-to-epilogue timings $e - p$ for a function fc_1 does not allow distinguishing among (a) uninterrupted function call, (b) concurrent interleaving with another call, (c) descheduled in favor of a different process, (d) parallel execution (*Cf.*§6.4.1).

Our prototype (§7.2) uses a combination of eq. (6.1) and eq. (6.2). Eq. (6.1) is used for short-running processes where there are not enough samples to feed the weighted average.

Challenges A few details on call styles, exception handling, and the module cache are worth noting.

In the case of blocking interfaces, no items will be arriving at the boundary before previous items finish executing. As such, there is no meaningful notion of queues (*i.e.*, queues will always have zero items). Instead, IGNIS wrappers calculate windowed averages over *individual* past calls, without modeling concurrently pending ones. If a blocking interface is marked as a potential bottleneck, IGNIS will suggest conversion to a concurrent, non-blocking version as an intermediate step before distribution; runtime transformations (§6.5) generate and link the new interface automatically.

In the case of non-blocking interfaces, functions will be called as soon as items arrive. Gathering accurate statistics for individual calls can be challenging, because OS-internal queuing and reordering is not visible to the boundary wrapper. Specifically, a prologue–epilogue time interval can mean any one of several scenarios (Fig. 6.5). Fortunately, as services are deployed for some time prior to saturation, IGNIS has the benefit of collecting accurate long-term runtime statistics. The problem is further alleviated by IGNIS's ability to distinguish the concurrent from the non-concurrent case, by checking the number of concurrently pending calls in the function wrappers.

In cases of runtime exceptions, the control flow bypasses the epilogue, skipping statistics collection. For these cases, Ignis wraps the encapsulated function call with an exception handler that calls the epilogue and rethrows the exception.

For consistency and performance purposes, module systems maintain a cache of loaded modules. When an existing module is imported again in a different part of the codebase, they return a cached reference to the original module. To precisely attribute load to the right bridge between modules, Ignis' profiling transformations return a *fresh* wrapper function upon each load. For example, if modules `A` and `B` import `C`, `B` may be placing 10× more load on `C` than `A` does. Such a 1:M function mapping does not affect module consistency and has negligible effects on performance (except in cases of meta-programming (§6.5.2)) but offers noticeable improvements to load attribution.

6.4.2 Application-wide Coordination

Ignis starts by setting up an application-wide coordinator—a logically centralized control hub that builds a registry of the available resources, interfaces with the agents at module boundaries, analyzes distribution recipes, and pushes guidelines to the boundary agents. Ignis daemons, cut-down versions of the coordinator, execute on other hosts that be used for scale-out, reporting on local resources, listening for replication requests, and transforming replica interfaces (§6.5).

Resource Registry On every node, coordinators poll the underlying environment for software and hardware information. Newly configured daemons report this information up towards the parent coordinator.

Information on the software environment focuses on the operating system, language runtime, and various built-in libraries. Among other reasons, this is important for modules compiled to execute natively as well as module dependencies that need to be installed globally. For example, imagine a module that needs root permissions but happens to not be available on a specific node. As IGNIS cannot set up this module during runtime, it will not be able to replicate and schedule calls to that module on this particular node.

Hardware information includes memory configuration, CPU speed, and bus speed. Information about the characteristics of the network (*e.g.*, latency) is continuous, and extracted periodically from the performance of call traffic.

Boundary Registration After import (§6.2.2) and transformation (§6.4), the newly transformed boundary registers with the coordinator. The rewired `require` function notifies the coordinator with a message that includes identifiers of the parent and child modules, pointers to the original and transformed DAG handles, and a pointer to the local recipe.

The new boundary is added to a list of boundaries monitored by the coordinator. The list is ranked by need-to-replicate, recalculated with every new update the coordinator receives from a boundary. The calculation focuses on the fraction of the execution time taken by each module, and is updated at a frequency set by the coordinator.

If the update leads to changes in the list, boundary agents whose ranking changed are notified. This notification contains guidelines that allow modules to make online decisions, including the minimum and maximum number of replicas and the threshold values of the formulas (6.1) and (6.2).

Load Attribution Agents running at each boundary cannot "see through" modules in order to attribute load among a module and its dependencies correctly. For example, an agent at the outermost boundary `A` of a dependency tree `A`$\rightarrow$`B`$\rightarrow$`C` does not know how much of the latency comes from `B`.

To solve this problem, IGNIS relies on the coordinator, which understands the structure of dependencies and can correctly calculate how much of the latency comes from each module. Such a calculation is more complicated than a simple subtraction, as a module may invoke interfaces from multiple modules at the same time. The technique of creating 1:M wrappers (end of §6.4.1) alleviates much of the problem.

6.5 Distributing Modules

Transforming a system into a distributed version amounts to scaling out individual bottlenecked modules (§6.5.1) while selectively maintaining the illusion of a single runtime (§6.5.2).

6.5.1 Scaling Out

To scale a module out, Ignis (i) creates a new process importing the module and Ignis-specific libraries, (ii) sets up a communication channel between the old and new process, and (iii) schedules calls across all the replicas.

Setup Ignis spawns a new operating system process on a node that fits certain criteria, such as light load, acceptable latency, and compatible versions of packages.

The new process binds to a fresh ($IP, port$) pair, used both as a communication handle and as a unique node identifier. Nodes communicate over TCP even for processes colocated on the same machine, as TCP is system-agnostic and hides the distinction between local and remote communication. The newly spawned node first loads a copy of Ignis to (i) set up the channels and, when needed, extract characteristics of the hardware, (ii) interface with the coordinator on the parent process (or higher in the module/process hierarchy), and (iii) further respond to increased load *within* that module.

Transformations Scale-out transformations focus on replacing a local module with a thin client that forwards calls to a set of remote modules (Fig. 6.6). To create a thin client of the same type as the original module, Ignis inspects the DAG returned by the import call in the new process. It recursively replaces each node in the DAG with a wrapper:

```
dtf (e: DAG) : DAG := match e with
| Obj ((s, v) :: vs) -> Obj ((s, dtf v) :: dtf vs)
| Arr (v :: vs)      -> Arr ((dtf v) :: dtf xs)
| Fun f              -> toRPC(e)
| __                 -> toInterposed(e)
end
```

Figure 6.6: **Distribution transformation.** IGNIS' `toRPC` function takes a function f_1 and returns a function f_2 that, when called called, sends the arguments to f_1 and calls it. Primitives are wrapped with interposition (*Cf.*§6.5.1).

- primitive values are wrapped with an interposition mechanism that records and propagates changes.
- function values become RPC stubs that serialize arguments, send them via the channel, and collect results.
- mutable values have their getter and setter functions replaced with RPC stubs similar to functions.
- exceptions are re-thrown in the parent context after inspection from IGNIS running on the parent module.

IGNIS maintains a distributed map from module identifiers to (a set of) channel pointers. If a module is already loaded, IGNIS retrieves the channel pointer and returns the previously-wrapped DAG; this is useful, for example, in cases where dependencies have a diamond shape (*i.e.*, two different modules import the same module).

Scheduling Unless instructed otherwise (§6.6), calls to replicas are scheduled in a round-robin fashion.

Blocking calls yield to the IGNIS scheduler, which picks a replica, serializes given arguments, sends them through the channel to the chosen replica, and waits for a response. The child-side wrapper de-serializes arguments, calls the required method, and sends results back through the channel. For non-blocking calls, the parent module wrapper registers an event listener that invokes the provided continuation when results become available on the channel.

Challenges A few technicalities on asynchronous replica spawn and module resolution are worth noting.

Spawning a new process takes several tens to hundreds of milliseconds (§8), which should be off the critical path—especially at the point in the execution of the program when IGNIS is in utmost need for performance. Thus, IGNIS spawns each module replica in asynchronous, non-blocking mode while calls are served by the original module. When the replica completes initialization and is ready to handle calls, the IGNIS coordinator on the child sends a message with all the information described in §6.4.2 to the parent process.

To be able to locate modules in the new environment, IGNIS needs to patch the module resolution algorithm at the replica. For replicas running on the same physical host, locating a module is relatively easy: absolute modules are stored in well-known locations in the environment (*e.g.*, Python's `sys.path`), modules local to the project are stored within the application (*e.g.*, JavaScript's `node_modules`), and modules relative to the current location are prefix-resolved at runtime, by having IGNIS prefix the module path appropriately.

Replicas running on different hosts require a more complex runtime resolution, which is avoided by prefetching modules. To prefetch modules, the dependency chain is analyzed upon startup by the coordinator (§6.4.2). Project-local modules are extracted at startup and pre-fetched upon daemon setup. While some disk space is wasted as the majority of these modules will not be used, latency on the critical path is avoided. Absolute modules are resolved similarly and re-introduced as project-local modules on the new host. Modules relative to the current location either work unmodified (first replica on a host) or are prefix-resolved by IGNIS (subsequent replicas).

6.5.2 Maintaining (the Illusion of) a Single Runtime

This section describes several techniques related to transformations intended to maintain the original application behavior. Whether each one of these techniques is required or not is a recipe-specific question, discussed in (§6.6); here, we merely show how Ignis implements each technique.

The techniques below require additional metadata to be attached on the serialized value. In Ignis, this is achieved by adding a new "hidden" `__ignis` property instead of embedding the entire value in another message.

Distributed References To facilitate cross-replica addressing, transformations at the replica boundary assign identifiers to non-primitive values. These IDs can be viewed as distributed, shared-memory pointers which RPCs include in their messages. Replicas then maintain a "decoding" hash table, mapping IDs to their in-replica pointers: whenever they receive a message, replicas use the table to route freshly deserialized values to the right function (or method).

Generating a fresh ID requires that the new pointer is different from all other pointers; otherwise, two pointers refer to the same location and should be assigned the same ID. This is achieved by maintaining a second "encoding" hash table from non-primitive values to IDs. In constant time, Ignis checks if an entry already exists (extracting the associated ID) or not (inserting a new (val, ID) entry).

The creation of copies during transformation and serialization breaks reference equality. To solve this, when an RPC leads to a new memory alias in a replica, Ignis attaches an `alias` entry to the serialized value. When receiving such a value, Ignis on the child will create and return a reference to an existing object. The sender side uses the "encoding" map to assign a distributed pointer to an object.

The same consideration applies to preserving reference equality for the root of the DAG between RPCs. A common pattern in many languages is to have methods that return `self`; such code would break if the return value of the RPC was a fresh copy of the method receiver.

Call Types Constructors, prefixed by `new`, are different from typical functions (*i.e.*, memory allocation outlives the function call). Ignis inserts additional logic into the RPC stubs to detect this case,[1] and augments RPC messages to signal that the target functions should also be called as constructors. The return value from a constructor is in turn transformed into an object whose methods are RPC stubs (§6.5 and Fig. 6.6): the true object lives within its own replica.

Similarly, standard garbage collection (GC) cannot "see through" boundaries. To solve this, Ignis also propagates garbage collection events by adding GC hooks to every object that is the result of a transformation. These "weak finalizers" fire when the object is about to be collected, causing Ignis to broadcast a message for removal of that object from any auxiliary tables. With no inbound pointers, objects in replicas will be collected in the next cycle.

Although state updates via method calls are redirected to remote objects, direct updates via property values require custom detection and propagation. Ignis wraps the transformed output DAG with an interposition mechanism that provides reflection capabilities and gets invoked upon property accesses.[2] This wrapper detects and records changes to any of the object's properties. Further nested wrappers monitor nested objects (Fig. 6.6). A similar mechanism is used to detect program- or user-initiated invalidation in the module cache (*e.g.*, to reload a module): a cache wrapper detects and broadcasts entry invalidation.

Although unusual, modules other than Ignis may dynamically rewrite module interfaces. If Ignis is loaded earlier, these modules will attempt to overwrite the RPC stubs instead of the encapsulated methods. The DAG interposition wrapper described above detects accesses and applies rewrites internally. Such rewrites are simplified by not having to cross channels, as they occur long before scale-out.

Environment Binding Generally, names defined by the programming language

[1] For example, `__call` metamethod in Lua and `new.target` in JavaScript.

[2] For example, metatables in Lua and `Proxy` objects in JavaScript.

or standard libraries are valid on all replicas: language constructs such as the top-level `Object` are implemented in the runtime; stateless libraries such as `crypto` and location-agnostic OS-wrappers such as `net` are made available unmodified within a replica.

Certain global or pseudo-global[3] constructs may require redirection to the top-level process. For example, a replica's `out` and `error` streams must appear in the respective streams of the top-level process. Ignis on each replica transforms and shadows these methods with RPCs that redirect their arguments to the top-level process.

Other built-in libraries are replaced by scalable, distributed versions. This replacement is achieved by transforming the DAG of the specified built-in module on every replica with functions that call into the distributed version. A notable example is the `fs` module, with a large and well-explored space of available trade-offs. At one end of the spectrum lies a partitioned `fs` with strong consistency guarantees that redirects accesses to a single authoritative node. At the other end of the spectrum lies a replicated `fs` that distributes accesses in an eventually consistent fashion across a subset of nodes, using a consistent-hashing scheme [96, 211].

Finally, objects may invoke methods inherited from classes higher in the hierarchy. These superclasses—or prototypes, for prototype-based languages such as Lua and JavaScript—may have been imported from a different module. A naive implementation of transformations to RPC stubs can thus lead to a series of nested round-trips until a call reaches the correct destination. Ignis detects class (prototype) hierarchy levels while traversing the DAG and creates a dispatch table with RPC stubs that route calls to the final destination.

Maintaining Ordering Although communication primitives across a single edge are reliable and in-order, messages that cross multiple edges may arrive out of order.

[3] JavaScript implementations introduce objects that are not part of the EcmaScript specification into the global scope, such as `process` and `console`. Similarly, Lua's Luvit introduces its own globals, such as `p()` and `exports`.

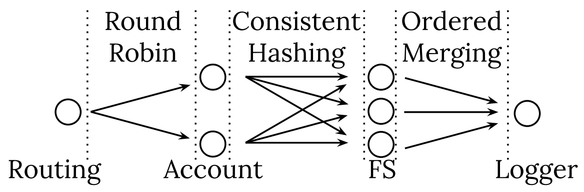


Figure 6.7: **Effects of recipes**. Call distribution across replicas of the wiki's authentication submodules, as a result of recipes (*Cf.*§6.6).

To maintain ordering, serialized values are assigned an internal sequence number. Sequence numbers are generated at the call site and follow the value as it travels through the system. In cases where the value is changed or replaced by a new one, the sequence number is extracted and transferred along.

6.6 Distribution Recipes

Recipes are runtime configuration objects that give developers the ability to tune several trade-offs related to the resulting distribution [71, 3, 120] without requiring manual development. They are responsible for generating transformation parameters and configuring deployment details. The latter is important in order to identify which nodes can be used for scaling out a module. The large number of modules and their different requirements make this challenging (but we do not discuss it in more detail here).

Fig. 6.7 shows the (semantic) result of annotating the wiki's authentication subsystem (§6.2.3) with recipes. Module `routing` spreads calls homogeneously over the two `account` replicas. Module `account` spreads consistently over `fs`—that is, identical arguments at any of the `account` call sites will hit the same `fs` node. Module `logger` orders calls to avoid mixing logs from different requests.

Ordering highlights some interesting features. First, as it degrades performance, it can be enabled only for a limited amount of time—benefiting from recipes being *runtime* constructs. Second, order is not enforced for direct accesses, such as assignments that change logging levels. While stronger consistency can be guaranteed with different recipes, the developer here expects logger levels being set only once and probably at the start. This is a case where causal consistency between assignments and calls (but not between calls) can be exchanged for strong eventual consistency.

Writing recipes is equivalent to specifying a system's distribution properties. Reasoning about them, rather than the mechanisms by which these properties are implemented, reduces the chance of errors while scaling out a system.

Recipe Expressions Recipes can be expressed at the level of a program, propagating down to the rest of the dependency chain, or at that of individual modules. Ignis' built-in recipes are overridden by program-level recipes, which are in turn overridden by recipes accompanying individual modules.

System-wide recipes describe how to configure the distributed system, and include: profiling details such as queue depths and saturation levels, decomposition limits such as replica counts and module groups, expected semantics of augmented built-in libraries (*e.g.*, `fs`), and module priorities, such regex patterns for modules that should or should never scale out. Users also need to provide the details of the daemons running on remote nodes; Ignis can then configure the details by querying the daemons. Here is a typical program-wide recipe, extracted from an experiment:

(p_{18a})

```
1  nodes:[{ip: "128.30.2.133",  port: 8013}],
2  fs: ignis.fs.EVENTUAL,
3  cold: [/process/],
4  hot: [/ejs/, /.*dash/]
```

It configures Ignis so that an additional node can be used to launch replicas (1), the standard `fs` module is replaced by an Ignis-provided, eventually consistent one (2), the `process` module should not be replicated (3), and `ejs` and `lodash` should be distributed by default (4).

Module-specific recipes give developers fine-grained control over distribution, allowing them to express intuition about individual modules they import:

Table 6.2: **Ignis recipes.** Selected, example recipes and their default parameters (*Cf.*§6.6).

Recipe	Options	Default	Explanation
Nodes	`[{ip:...}]`	`localhost`	Nodes running Ignis
Level	`0, 1, ..`	`1`	Decomposition depth
Group	`subtree.json`	—	Group module subtrees
Hot	`["util"]`	`parse, crypto`	Always distribute module
Cold	`["fs", "os"]`	`process, fs`	Never distribute module
Copies	`true, [2, 8]`	`[0, CPU]`	Multiple replicas
Sched	`RR, MLF, Weigh`	`RR`	RPC scheduling policies
Q_Depth	`10--100K`	`Inf`	Inter-module queue depth
On_Fail	`(e) => {..}`	`throw`	Module failure hook
Saturation	`(lvl) => {..}`	—	Saturation level hook
Comm	`TCP, UDP`	`TCP`	Communication type
Preload	`true, false`	`false`	Load proactively, not lazily

```
1  copies: [2, 10],
2  fs: ignis.fs.LOCAL,
3  order: false,
```
(p_{18b})

This module should inherit the global recipes specified earlier. Moreover, it should have a minimum of two and a maximum of 10 replicas running (1), use the local `fs` on each node (2), and not need any ordering (3).

Discussion Tab. 6.2 summarizes more recipes, a few non-obvious characteristics of which are worth clarifying.

Being dynamic objects, recipes are flexible. They can be re-generated at runtime and change during the lifetime of the program—even between different imports of the same module. For example, different branches of the control flow can load the same module with different recipes.

Currently, Ignis defaults to recipes that are conservative, in the sense that they will never attempt distribution that breaks the semantics of the program. For example, purely functional built-in libraries such as `parse` and `crypto` default to permitting distribution, but `process` and `fs` do not.

Since recipes affect the semantics of the resulting program, an obvious question is whether developers can get them wrong. The answer is yes, but this is no worse than other approaches that aid distributed programming: for example, developers are free to introduce side-effectful computations in MapReduce's purely functional primitives [49]. Ignis makes reasoning about semantics easier, as it concentrates the decisions that could break semantics into the recipes, rather than forcing them to be interleaved with application logic. Our position is that there is *much* more room for error by avoiding aid from tools like Ignis and MapReduce altogether and building distributed systems from scratch.

6.7 Summary

This chapter introduced *light-touch distribution*, a first step towards automating the generation of distributed systems from distribution-oblivious programs. This is achieved using a set of programmatic *transformations* parametrizable by optional *distribution recipes.* These operate at module boundaries during runtime to collect profiling information, detect bottlenecked components, and dynamically separate and coalesce parts of the application.

The next chapter (§7) covers implementation details of Andromeda and the surrounding frameworks.

Chapter 7

Implementation

Decisions related to backwards compatibility (§2.1.1) and programming style (§2.1.4) constrained our implementation candidates to Lua and JavaScript. Both feature heavily engineered run-time environments that perform comparably (for our purposes). As the benefits of portability outweigh the problems of working of a quirkier language, we decided to go with JavaScript.

The current version of Andromeda is about 11K lines of code, the majority of which is written in ES5.1.[1] About 70% of code implements built-in services (about half of which supports the `local` group). Internal utility libraries (§2.1.2) are about 2KLoC and code related to interactive use (*e.g.*, shell, value beautification) is about 1.1KLoC. Andromeda can be executed atop any JavaScript runtime, but on Unix it defaults to Node.js [48], which bundles (i) Google's V8, a fast JIT compiler, (ii) libUV, asynchronous cross-platform OS wrappers, and (iii) a small set of standard libraries (*e.g.*, `crypto`).

Andromeda treats a multiprocessor as a distributed system [33, 16], lunching a number of nodes equal to processors. For each node colocated on the same host, it spawns a userspace process. Remote communication occurs over TCP, with experi-

[1] We refrained from using ES6 features beyond arrow functions to simplify serialization. Fully-supporting ES6 is a matter of engineering effort.

mental support for reliable UDP [162].

7.1 Challenges

Andromeda critically depends on the ability to serialize and communicate (or store) arbitrary objects. However, JavaScript's default data interchange format (JSON) does not support several key features—most notably: function literals, cyclic references, and property descriptors (*i.e.*, if a value is writable, enumerable, configurable, and prototype-owned). To support these features, Andromeda implements its own serialization library.

The mismatch between the synchronous nature of shell output and the asynchronous nature of system calls poses a different challenge. Asynchronous calls that are provided a `log` continuation for printing results on the shell interfere with the state of the shell (*e.g.*, overwrite the prompt, input line, *etc.*). To solve this, `log` detects whether it is provided as a continuation on a node running the shell. After beautifying and printing input, it flushes control sequences on the output stream that restore the shell's state. Due to cooperative scheduling (§2.1.4), there is no concern of garbled output.

Another challenge was due to Andromeda node multiplexing atop a single Unix host. Early versions of Andromeda used the node SHA256 identifier as the node-local Unix directory name (within a parent `.andromeda` directory). IDs were generated by nodes (or loaded from disk) rather than provided as arguments by the node spawning them. As a result, (i) knowing whether a node ID on disk is already executing was challenging, as it required binding to the node's state (to query them) before deciding whether this is the right node one to bind to; (ii) running `local` calls such as `local.obs` was at times inconsistent after a Andromeda reboot because the shell was bound to a different node (due to `SHA256` not maintaining order). These problems were solved by using an integer identifier: new nodes are provided an ID, incremented by the node spawning them. Node-local state is organized on disk as directories named by consecutive integers such as `/0`, `/1`, and `/2`. Rebooting the system (or starting a single-node system) starts from `/0`.

7.2 Frameworks: BreakApp, Iris, and Ignis

This section discusses technical details related to BreakApp, Iris, and Ignis.

BreakApp Excluding all Andromeda code, the BreakApp prototype is approximately 2K lines of JavaScript. One third of them is for handling policies and other configuration parameters, and the rest supports transformations, serialization and low-level handling of different isolation primitives. For encryption, we use Dan Bernstein's NaCL library [21], compiled to JavaScript using Emscripten (adding another 2K LoC). Our implementation does not make use of any non-JavaScript features beyond the GC hooks mentioned in §4.6.8. For these, we use V8's weak finalizers with callbacks that fire when an object is about to be garbage collected.

There are several challenges related to policies that depend on the structure of the dependency tree. First, there is a one-to-one correspondence between source files and modules. There is little to no information during runtime about which packages file-level modules belong to. Second, deduplication causes the Node Package Manager to not install dependencies in a deterministic way. As a result, high-level granularity policies (*e.g.*, `LEVEL`, `GROUP`) can lead to different compartmentalization results depending on the package installation order. To mitigate these problems, BreakApp statically analyzes information upon startup in order to make policy expressions meaningful. Starting from the leaves of the dependency tree, it creates a map from files to their source package. It also uses information from the various `package.json` files and the directory structure to infer a canonical dependency structure. This functionality is accessible through the tool's command line interface via `breakapp --create-map` (or `-c`), which makes it easy to use as a post-install hook for any package manager (*e.g.*, Yarn).

A key challenge in the implementation of process and container isolation had to do with a conflict between Node.js's (i) non-blocking I/O and (ii) blocking `require` statement. To ensure that the `require` call returns *only after* the compartment has been created, BreakApp in the parent compartment polls the filesystem constantly for a file that confirms that the child compartment has created the channel. Polling allows the system to actively check a number of different sources during each iteration (and timeout after a while). Since the channel is created *before* launching the new compartment, an alternative solution was to block on the channel for an ACK message. However, compartment creation might fail; thus, there is no guarantee that the parent compartment will not block indefinitely. Environments that expose any kind of preemptive multi-threading should not face similar issues.

Iris PIC specification (§5.4) uses JavaScript's object notation, along with syntactic sugar for accessing built-in modules (§5.4.5). Such sugaring is achieved by wrapping PICs upon import with a function that resolves the names of built-in libraries (*e.g.*, `path`, `fs`) to objects modified to allow only synchronous methods.

Andromeda's runtime enforcement component (§5.5) is implemented using ES6 `Proxy` objects [216] and the `Reflect` API. Module loading necessitated augmenting several functions in the built-in `Module` module. The `Module` module is responsible for locating a module, wrapping it and exposing module-local variables, interpreting in the current V8 isolate, and updating the module cache. Andromeda adds several steps: (i) the `require` module-local method is augmented to capture a PIC as a second parameter, (ii) the internal `compile` method is augmented to apply transformations to the context and interface (§5.5), (iii) the cache is augmented to support storage and recovery of the unmodified, context-customized, and interface-attenuated versions of each module. The internals of Node.js' built-in module system as well as Iris' changes are outlined in Appendix §B.

Ignis Ignis builds on the primitives provided by Andromeda to implement monitoring, adaptive scaling, and wrappers for built-in libraries such as `fs`.

Applications interact with IGNIS via JavaScript's built-in `require` function. Recipes are encoded as JavaScript objects and passed as the second parameter. Due to variadic arguments, recipe-infused codebases remain backward-compatible with IGNIS-less runtimes.

Chapter 8

Application Programs

ANDROMEDA's primary goal is to significantly lower the complexity of developing distributed systems. To evaluate complexity, we use a combination of micro-benchmarks (accidental) and larger distributed applications (essential). As performance is a secondary concern, we want to (i) confirm it is within reasonable limits, and (ii) understand where to aim future engineering effort. For these aspects, we use targeted performance microbenchmarks to stress ANDROMEDA's core infrastructure. Our goal is *not* to evaluate individual, trivially replaceable subsystems, but to evaluate ANDROMEDA's end-to-end benefits.

Several takeaways are worth highlighting. Setting up ANDROMEDA to the point where commands can be run interactively takes about 1.3s—*i.e.*, 2–3 orders of magnitude lower than low-effort specialized systems. Even for trivial tasks, ANDROMEDA's combination of language and library lowers development effort than general-purpose distributed environments. For more complex tasks chosen by the designers of these platforms, ANDROMEDA lowers development complexity by 7.3–78.5×. Core overheads, such as interposition and transformations, are negligible—but value-serialization can be a significant source of overhead.

Our hardware setup uses a network of five workstations connected by 1Gbps links: one large machine ($\mathtt{a_1}$) with 251GB of memory and 64 2.1GHz Intel Xeon E5-2683 cores, and four smaller machines ($\mathtt{q}_{\mathtt{1-4}}$), each with 4GB of memory and two 3.33GHz Intel Core Duo E8600 processors. Most experiments were run on $\mathtt{a}_{\mathtt{1}}$ to avoid orthogonal network overheads, but include full costs of serialization and communication (§7). No special configuration was made beyond disabling hyper-threading—specifically, the network protocol stack was left unoptimized.[1] Our software setup uses Node.js 6.14.04, bundled with V8 v.5.1.281.111, libUV v.1.16.1, and npm v.3.10, executing on top of Linux kernel v4.4.0-134. BreakApp's `LXC` option uses Docker 17.06.0-ce for its containment infrastructure.

8.1 Complexity Micro-benchmarks

To understand the effects of the essential and accidental complexities developing and deploying distributed systems, we present a few "complexity microbenchmarks", showing that Andromeda is significantly easier to pick up and operate.

Setup What is the complexity associated with non-essential tasks, such as environment setup and program loading? To understand this, we compare Andromeda with more established systems on typical setup and tear down tasks. These tasks are scripted to the extent possible. All experiments are run on a minimal Ubuntu image from docker hub (*i.e.*, all setup times include the time taken to install dependencies).

Setting up Andromeda can be achieved via `curl -s up.ndr.md | bash` or, if the JavaScript package manager `npm` is installed, `npm -ig @andromeda/andromeda`. The former takes 0.56s and 1.01s to download and extract the bundle, respectively; the latter takes 8.22s, primarily spent resolving and downloading further dependencies. It then takes 142.63ms to launch the first node—which includes preparing the directory structure to support Andromeda. Additional nodes complete startup on an average of 92.81ms. Individual built-in services take an average of 3.1ms per service. The combined total setup time is about two seconds.

[1] Features such as kernel bypass [175] should yield significant improvements.

We compare with representative other systems that we used in other parts evaluation section. Fully-scripted setups of Hadoop [11] (v.2.7.3) and Spark [248] (v.2.0.1) take 1166s and 184s, respectively. Setting up via Docker [129] (v.18.09.2), a system created partly to simplify the process bumps these numbers by 40.751s. These are representative of "product" systems that are easy to deploy. Newer systems do not provide docker images (like ANDROMEDA) and required significantly more effort: ScalaLoci [240] (c.e1958be) took about an hour of mostly manual effort (partly due to its dependencies), we gave up on Husky [244] (c. 9e66349) after failing to resolve dependency issues. The timings for these systems do not include fault tolerance of master nodes (*e.g.*, Zookeeper [90]) nor any distributed resource allocation capabilities (*e.g.*, Yarn [230]), although preliminary versions of these come with ANDROMEDA.

One-liners How does ANDROMEDA programming fare in comparison with other environments? For a first answer, we compare trivial tasks that in ANDROMEDA can be written in a single line (Tab. 8.1). We compare the effort with Erlang/OTP [232] (v21.3) and Plan 9 [166] (v.4); we believe that these are the most successful candidates for low-complexity general-purpose distribution. These examples are not expressible in more specialized systems such as Hadoop [241] or Bloom [9] (§1.3). Lines-of-code comparisons do not necessarily translate to complexity, but differences in orders of magnitude could be interpreted as qualitative rather than quantitative.

While the essential complexity of these tasks is trivial, the results highlight a trade-off: Erlang's abstractions excel at tasks that require nodes to perform computations, but fail at more administrative tasks such as setting up a package; conversely, Plan 9's abstractions excel at treating the entire deployment as a single system, but fail at tasks that require node-specific programmatic manipulation. For example, Erlang's default logggers cannot be configured dynamically, thus requiring non-trivial implementation of custom log handlers (about 50 LoC). In contrast, Plan 9's support for file namespacing allows trivial forwarding with a single line. None of the two

Table 8.1: **Andromeda one-liners.** These programs were collected from ANDROMEDA's source code and documentation, and slightly modified to alleviate dependencies and idioms. They omit `ux` prefix and the default node group (§2.4). Columns on the right present LoC for other environments—Erlang and Plan 9.

	Distributed Task	ANDROMEDA Program	Erlang	Plan 9
1	Have nodes send their IDs	`task.exec(() => self.config.getNid(), log)`	1	3
2	Get sum of processors	`hw.get("summary", (e, d) => log(d.cpu))`	1	1
3	Mixed criticality writes	`store.put(act, () => {store.put(likes)})`	> 100	>100
4	Set up package on all nodes	`package.fetch("pow", log)`	X	1
5	Parallel proof of work	`i = 0; task.exec((s, s) => self.pow(s), {args: utils.toArgs(i++)})`	1	20
6	Gen/te local API from remote	`task.exec(() => txfm.rpc(printer), useApi);`	< 10	X
7	Point n_1's `loggers` to n_2	`group.new([n2], "n2", () => self.log = n2.log)`	< 100	1
8	Point g_1 node's `fs` to self	`g1.group.new(self, "me", () => {g1.task.exec(() => {self.fs = me.fs})})`	< 100	1
9	Notification on node failure	`self.events.on("peer-down", log)`	1	5
10	Shut a set of nodes down	`dsl.nodes.delete(halt: true)`	1	1

Table 8.2: **Rendering prior work on Andromeda** These distributed tasks are inspired from ones in other distributed environments.

	Distributed Task	Environment	LoC	Andromeda LoC
1	Remote counter [59]	Cloud Haskell	48+	2
2	Max Temperature [241]	Hadoop + Shell	50 + 16	9
3	TF-IDF [244]	Husky	157	3
4	Counting [56]	Inferno	26	5
5	Chat Service [240]	ScalaLoci	42 + 12	5

supports programmatic runtime transformations or tunable distribution guarantees, posing difficulty in expressing some tasks in a simple fashion. Andromeda provides a convenient middle ground between the two, additionally solving transformation and configurability challenges.

Examples from literature How does Andromeda fare when compared with programs written primarily for other environments? To correct any possible bias by starting from Andromeda, we turn our attention to small examples drawn from other frameworks. Our assumption is that, for systems demonstrating low complexity, their designers chose examples that elegantly and succinctly demonstrate this point.

Table 8.2 includes five examples from several platforms. The most interesting case is Max Temperature (2), in which an end-to-end Hadoop example takes several tens of lines of code [241]. A preprocessing step is added to fetch the dataset and bring it to a form that is better for Hadoop; this task is performed on a centralized computer, without leveraging any parallelism or distribution. This is unfortunate, because it could have been trivially distributed: there are no dependencies between nodes, and the size of data downloaded and processed are the same going into Hadoop. While it is near-impossible to express this pre-processing step from within Hadoop, it take 9 lines in Andromeda.

Another interesting case is the TF-IDF pipeline (3). While the Husky prototype is all written from scratch, Andromeda uses an existing natural-language processing package. This illustrates an example where ANDROMEDA lowers essential complexity of the task at hand by leveraging a rich package ecosystem.

8.2 Performance Micro-benchmarks

We now turn our attention to overheads of different underlying primitives used pervasively across ANDROMEDA: interpositioning, serialization, and transformations. Results are averaged over 1K repetitions, with .1K warmup.

Interpositioning To understand the costs of boundary interposition, we measure the time to access deeply-nested properties of two versions of an object: unmodified and proxy-wrapped. Paths to the properties (*e.g.*, `a.b.c....`) are random but generated prior to running the experiment. We construct 500MB-sized objects, each with a fanout of 8 fields nested for 12 levels. The proxy-wrapped version introduces interposition at every level. Traversing one million 12-edge paths (*i.e.*, root to leaves) averages 167.2ms and 595.7ms for the unmodified and proxy-wrapped versions, respectively.

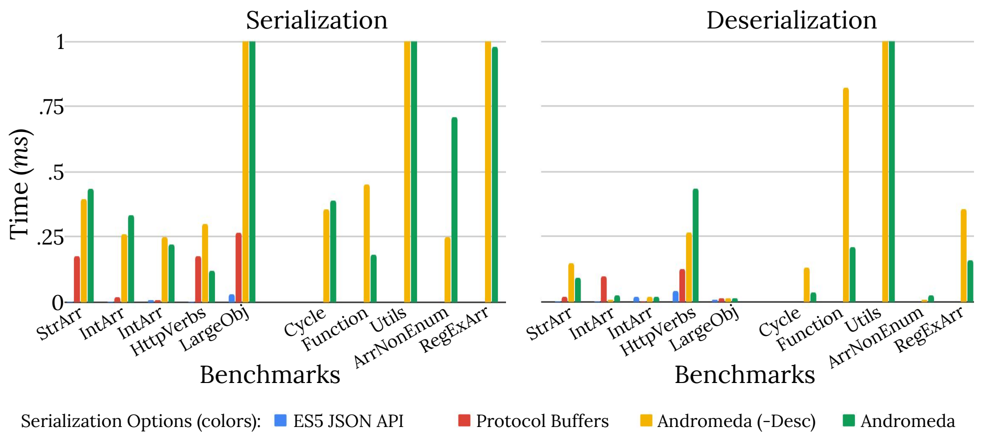


Figure 8.1: **Serialization overheads** Each plot (serialization and deserialization) is broken into two parts: simple values (left) and complex values (right). JSON and Protocol Buffers support only simple values (*Cf.*§8.2).

Serialization To understand the overheads of ANDROMEDA's serialization library, we compare a diverse set of real workloads over four configurations: (i) ES6's built-in JSON module, (ii) Protocol Buffers, (iii) ANDROMEDA without expanding descriptors for properties (§7), and (iv) ANDROMEDA with descriptors. Fig. 8.1 shows the average (de-)serialization times for 10 workloads: the five workloads on the left of each plot are base objects with structure that is supported by all libraries; the rest stress functionality that is only available on ANDROMEDA—cycles, enumerable properties, functions *etc.*ANDROMEDA's overhead is significant, mostly spent on cycle detection.

To explore ANDROMEDA's overheads as a function of object size, we synthesize custom objects with increasing numbers of internal nodes (fields). Fig. 8.2 shows both serialization time and (resulting) string size (affecting communication time).

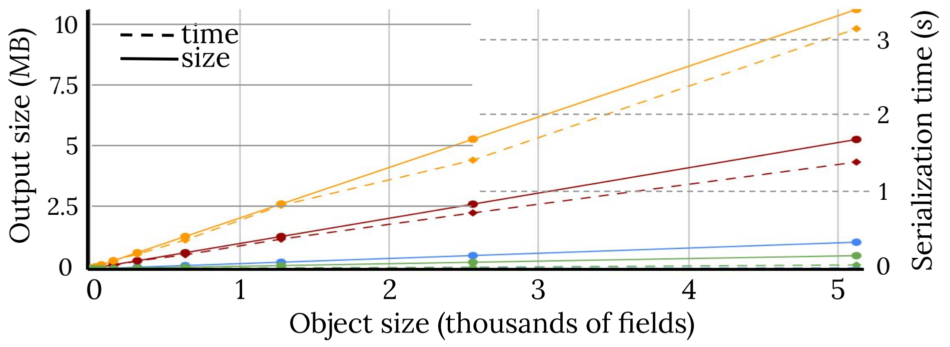


Figure 8.2: **Serialization output size (MB) and processing time (s).** From the bottom: ES6 JSON built-in (blue), Protobufs (green), ANDROMEDA no descriptors (red), ANDROMEDA with descriptors (yellow) (*Cf.*§8.2).

Startup To understand how these overheads affect code startup on realistic code, we use ANDROMEDA's transformation library to load and transform a set of realistic, third-party modules. The transformation is performed dynamically, by ANDROMEDA discovering and loading each module on the remote node, transforming them, and sending the transformed interface to a peer. Modules are tested under three configurations: (i) vanilla, (ii) co-located distribution (on the same host), (iii) networked distribution (over the network).

Fig. 8.3 shows module startup costs. Distribution requires transforming modules

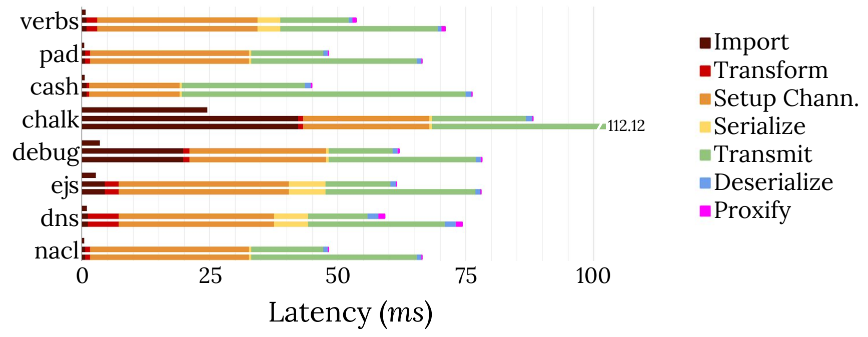


Figure 8.3: **Startup latency breakdown**. Each module is measured under three configurations: unmodified (short bar, only vanilla `import`), co-located on the same physical host (medium bar), and scaled out across the network (*Cf.*§8.2).

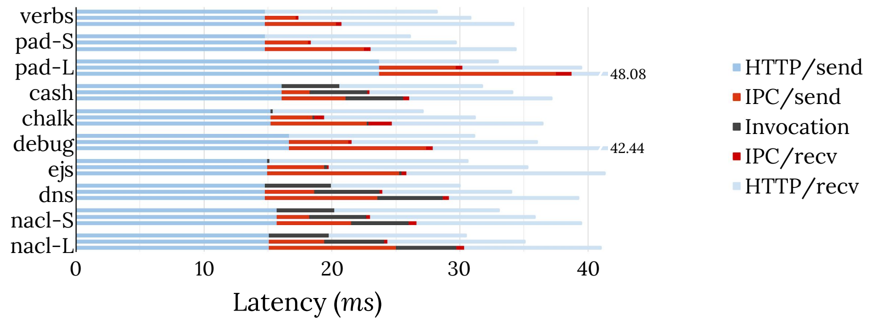


Figure 8.4: **Call latency breakdown**. The configuration is identical to the one of Fig. 8.3 (*Cf.*§8.2).

(RPC stubs), communicating them to the remote node, and launching replicas there. The startup time of larger modules is dominated by file-system accesses (*e.g.*, `cash` takes 798.2–1049.1ms to load all of its files). The startup time of smaller modules, such as `verbs` and `pad`, is dominated by constant factors (*e.g.*, 138.5ms for process spawn and 17.6–35ms for TCP setup).

RPC To understand the serialization and transformation costs on realistic code, we run a series of remote-communication (RPC-invocation) tests. The configurations (modules and setup) are the same as before.

Fig. 8.4 shows inter-module communication costs over the same configurations. To bring these costs into perspective, we embed modules in "no-op" HTTP applications. By placing these modules behind an HTTP server, we can study ANDROMEDA' impact on the end-to-end latency an HTTP client would experience. Processing-heavy modules such as `pad` and `nacl` are tested under input of size 5B (`S`) and 5MB (`L`).

Conclusion The micro-benchmarks presented in this section highlight Andromeda-inherent overheads by using carefully-constructed, worst-case workloads that are nowhere near the ones seen in practice: deeply-nested 0.5GB-sized module interfaces, large and complex serialization workloads, and near-zero module-internal latencies. However, they do show that Andromeda could benefit from a high-performance serialization and communication library, and that the overheads of transformations and interposition are negligible.

8.3 Sandboxing and Compartmentalization

BreakApp's techniques described are predicated on the hypothesis that applications today make extensive use of third-party packages. What are the modularity characteristics of JavaScript applications out in the wild? Table 8.3 outlines the dependency characteristics of popular JavaScript programs drawn from five different classes.[2] The table shows: (i) direct dependencies, referring top-level packages the application imports, (ii) total dependencies, including all packages in the dependency graph, (iii) total of file-level modules, (iv) the depth of the dependency tree, (v) non-third-party lines of code, *i.e.*, lines the author wrote, (vi) total lines in third-party, imported code, and (vii) the average lines of code per third-party file-level module.

Third party code is a non-trivial portion of today's applications. In our sample set, imported code is on average 4 times larger than homegrown; but the ratio is much worse for large applications (1:120 for hackernews vs. 2:1 chalk). Different applications spread third-party code differently. For example, in mobile applica-

[2] We do not discuss client-side web apps, since the emphasis of our work is language-agnostic, system-level decomposition (it just happens to use JavaScript, historically created for client-side web development). To address the reader's curiosity, however, here are some numbers: 1060 modules for apple.com, 1050 modules for the mobile version of reddit.com, and 365 modules for keybase.io.

Table 8.3: Five classes of JavaScript programs along with three widely used instances and their dependency characteristics.

	Application	Direct	Total	Files	Depth	ALoC	TLOC	TLoC/File
	cash	15	84	3554	5	1486	48540	13.84
commands	eslint	34	135	4689	6	187801	74893	39.97
	yo	30	301	5829	6	107713	106393	18.45
	popcorn	46	765	34322	10	14304	411706	12.34
desktop	twitter	10	120	4051	8	2514	165066	41.29
	atom	57	358	5252	9	15939	548642	107.1
	hackernews	5	871	49406	10	309	317144	6.42
mobile	mattermost	17	521	13672	8	6296	286388	21.37
	stockmarket	14	44	1985	5	2440	199119	101.48
	express	26	42	217	3	10159	2261	54.93
server	ghost	62	981	22029	9	42467	386676	19.35
	strider	64	659	10357	8	21090	303527	30.41
	chalk	3	4	9	2	217	10	18.44
utils	natural	3	3	193	1	12483	4116	81.51
	winston	6	6	83	1	4274	2326	79.52
average		26.13	326.27	10376.53	6.07	28632	190453.8	43.09

tions, more than 99% of their third-party code comes from a single package—the mobile framework in use (*e.g.*, Ionic, ReactNative). Server-side applications feature the largest amounts of total third-party counts, followed immediately by desktop applications.

Direct module counts, the boundaries of trust between the code that a developer writes and its third-party dependencies, are somewhere between 2 and 65. These numbers highlight the minimum number of compartments (average: 26.1). More fine-grained compartmentalization at the level of individual packages requires an order of magnitude more compartments (average: 326.2). Since there is a one-to-one correspondence between files and modules, file-level compartmentalization *is* possible but would require 1-2 orders of magnitude more compartments (*e.g.*, popcorn has more than 10K JavaScript files). Interestingly, analyzing more than 1K imports (translating to more than 100K file-level modules) reveals an average ratio of 43 lines of code per file, exceeding our expectations for least-privilege decomposition.[3]

8.3.1 Security

Does the system mitigate vulnerabilities (both discovered and hypothetical) similar to the ones outlined in §4.3? Table 8.4 presents twelve vulnerable modules, along with the compartment types used to mitigate their effects. The first six are public packages, and their vulnerable version is shown in the second column. For these packages, we use the exploit attached to the original vulnerability report. The last six are hypothetical vulnerabilities; although we were not able to find any packages with these specific vulnerability types in any vulnerability databases, we know they are possible to construct. All of them can be found in the online appendix.

Most of the first six packages can be used for a number of attacks. For example, `serialize` makes use of the Turing-complete `eval` function; user code can access

[3] As a point of comparison, Minix 3 [86], a modern microkernel that championed least-privilege separation, comes with userspace servers on the order of thousands of lines of code.

Table 8.4: **Vulnerabilities mitigated by BreakApp.** Various real (top) and hypothetical (bottom) vulnerabilities, and the policies used to mitigate them.

Package	Ver.	Type	R/H	Mitigation
qs	6.0.0	introspection, poisoning ⓒⓓⓖ	R	sandbox
serialize-to-js	0.4.8	eval ⓕ	R	sandbox
fernet	0.0.9	timing attack ⓟ	R	sandbox
uri-js	2.1.1	denial of service ⓚⓞ	R	process
libxml	0.16	unsafe extension ⓠ	R	process
hostr	2.3.2	read file-system ⓛ	R	container
glob.js	—	global variables ⓐⓑ	H	sandbox
this.js	—	context ⓒ	H	sandbox
mod.js	—	module cache ⓜ	H	sandbox
argv.js	—	process args ⓝ	H	process
env.js	—	user environment variables ⓝ	H	process
stack.js	—	inspect call stack ⓔ	H	process

global variables, patch system APIs, and inspect loaded modules. Launching it in a fresh V8 sandbox (`SBX`) defends against inspection and patching of the main program's data and structures. Since it is not written in C, it cannot forge pointers to bypass the language's safety features; therefore, launching a process would only add protection against denial-of-service attacks (*e.g.*, using `eval` to start an infinite loop), process arguments, and the shared environment. This highlights the core benefit of policies: they let developers specify what they care about based on *their* application structure and needs.

Over half of the problems can be mitigating simply by using sandbox (`SBX`). Defending against `glob.js` and `this.js` required explicit whitelisting of references (`CONTEXT`). Shielding against snooping the environment was possible via process-level isolation (`PROC`) and selective shadowing of environment variables and process arguments. Such shadowing (*e.g.*, `ENV`) creates an artificial copy of selected variables right after launching the compartment but before loading the module. Since the module system is grafted atop V8, separate V8 sandboxes do not have access to already loaded modules if not explicitly shared during sandbox construction. `mod.js`

Table 8.5: **Compartmentalization costs: compartment startup times.** Startup times for four basic compartment types, as a function of the number of compartments.

Compartments	Standard	Sandbox	Process	Container
5	4.3ms	12.9ms	342.5ms	5.9s
50	30.2ms	76.6ms	3.2s	32.8s
500	136.4ms	524.7ms	35.2s	332.2s
5K	1.7s	7.8s	362.4s	3330.5s
abs. / +1 cmpt	0.3ms	1.5ms	72ms	666.1ms
rel. / +1 cmpt	(baseline)	5×	240×	2220×

Table 8.6: **Compartmentalization costs: throughput and latency.** Throughput and latency of boundary crossing for four different compartment communication mechanisms, as a function of the number of compartments

Comp/nts	Function	Pipe	UDS	TCP
5	192.3GB/s	18.3GB/s	149.5MB/s	158.1MB/s
	6.5ns	1.3–1.4ms	17.8–73.8ms	17.7–36.6ms
50	157.1GB/s	17.5GB/s	127.0MB/s	134MB/s
	90.18ns	11.6–13.2ms	244.5–536.6ms	210.3–566.8ms
500	46.5GB/s	3.6GB/s	16.4MB/s	20.9MB/s
	294.3ns	154.3–160.3ms	3.71–11.95**s**	6.5–15.6**s**

required whiltelisting the `module` module with a *fresh* module cache. The call stack and event queue are shared between different V8 sandboxes; disabling access to the call stack for `stack.js` requires a separate process. Mitigating timing channels for `fernet` required only a sandbox (`SBX`) and a constant minimum response time (`TIMER`). Mitigating `url-js`'s DoS problems required processes (`PROC`), replication (`REPLICAS`), and a special scheduling policy (`SCHED`). We will see the details in §8.3.3.

8.3.2 Performance

What are the overheads of different isolation mechanisms related to policies? Tables 8.5 and 8.6 highlight the costs of starting up new compartments and crossing compartment boundaries under various configurations.

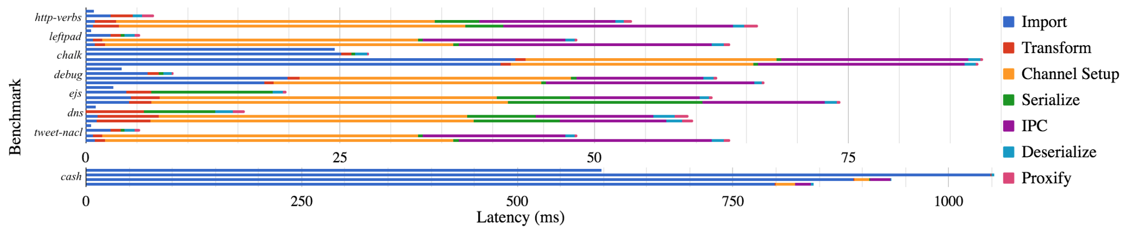


Figure 8.5: Startup breakdown of eight packages under various configurations: vanilla, sandbox, process, and container.

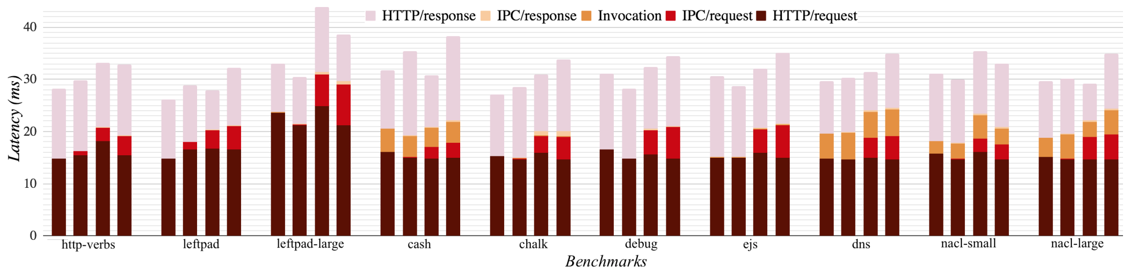


Figure 8.6: Latency breakdown of 10 different workloads with four compartment types for each: vanilla, sandbox, process, container.

For the first experiment (Table 8.5), we minimize the effects of module sizes by making modules return a single integer and launch compartments sequentially. *Standard* is how the vanilla module system works: it looks up a module on the filesystem using a resolution algorithm, wraps it so that its global variables do not leak to the outer context (and to provide some global-looking variables, *e.g.*, filename), and evaluates the code in the current context. *Sandbox* creates a new V8 context for each module and selectively whitelists shared variables from the parent context. *Process* and *Container* use OS processes and Docker containers to isolate compartments between each other, resulting in higher startup costs.

For the second experiment (Table 8.6), we process an in-memory (`/dev/shm/`) stream of 1GB using a linear pipeline of mostly-empty stages. Pipeline stages only flush some timing metadata when they detect the end of a stream. Streaming starts only after all connections have been established (*i.e.*, no TCP handshake costs included).

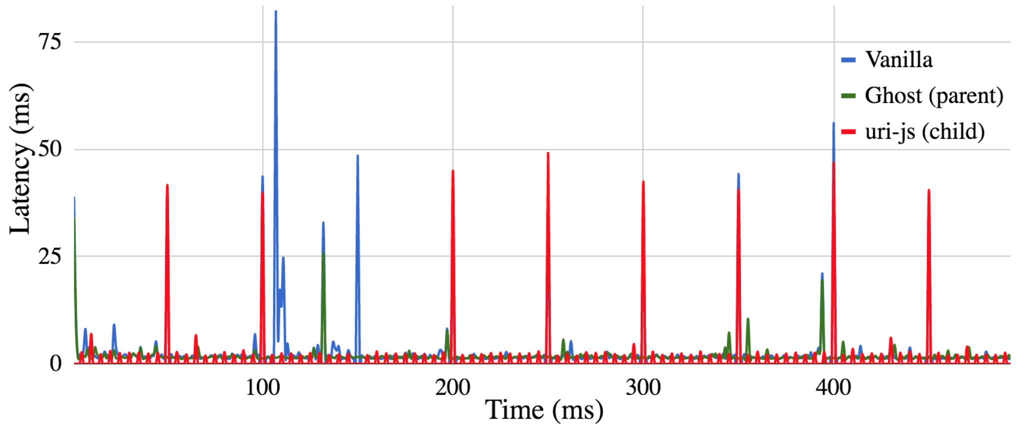


Figure 8.7: Denial-of-service attack against a blogging platorm: (i) vanilla setup (blue); (ii) two process-level compartments (green + red).

8.3.3 Denial of Service Mitigation

Can BreakApp be used to mitigate DoS attacks? Fig. 8.7 shows latencies of issuing 500 HTTP requests to a slightly modified Ghost server where `url-js` is susceptible to DoS attack. The workload consists of 90% benign read requests of various posts; 8% benign search requests; and 2% malicious search requests. Malicious search requests block the event loop for 40–60ms.

In the first configuration (blue line), all modules run in a single process. Read requests reaching the server right after a malicious search query get delayed significantly or, at higher rates (not shown here), timeout. In the second configuration (green line: 90% reads; red line 10% queries), the `url-js` module runs in a separate compartment with a policy of `PROC` and `PIPE`. Although it takes slightly longer to process search requests (+1ms on average), malicious search requests do not block benign read requests: it is only the subset that goes through the search functionality that remains paralyzed by the DoS attack.

In a different experiment, we increased both the number of malicious requests as well as their latency. Due to monitoring, BreakApp is aware that some requests are taking significantly more time than expected. By examining the input to the most recent RPC, it can distinguish between problematic inputs and non-problematic

Table 8.7: **Characteristics of the benchmarked modules.** Module shorthand, size in terms of lines of code (LoC), number of files (Files), number of nodes in the import return object (DAG), depth of dependency graph (DD), average fan-out of the dependency graph (F-O), number of function values (fs), overview of its internals (Notes) (*Cf.*§8.3.2).

Module	LoC	Files	DAG	DD	F-O	fs	Notes
verbs	29	1	28	2	27.0	0	constant string-to-string map
pad	52	1	1	1	1.0	1	small, pure function
cash	451725	10839	75	7	314.0	49	large library with system calls
chalk	145706	9630	5	3	5.3	2	builder objects/cascading calls
debug	554746	8657	34	4	51.3	14	varargs; output to parent stream
ejs	59396	4950	25	4	12.0	11	extensive, pure, testing fixtures
dns	4826	1	60	3	34.0	16	built-in module, async calls
nacl	94686	5387	54	5	40.8	42	CPU crypto processing

ones. We experimented with four `ON_FAIL` policies: (i) shut the child compartment down and report; (ii) restart the compartment; (iii) spawn a new replica and use a scheduling policy (*e.g.*, round robin) to schedule RPC calls to these replicas; or (iv) pushback based on recently-seen inputs. We crafted careful "asymmetric" attacks where a small number of malicious requests blocks the event loop for extended periods of time.

The combination of multiple `utl-js` replicas and caching of results from requests that take longer than 0.5s with a 30s age timeout was successful at mitigating them. This was a serious improvement over the previous setup: we were not able to saturate the system without generating *additional* malicious strings.

In our final experiment, we overrode the round-robin scheduling policy during runtime by passing a function that implements priorities. BreakApp split requests into 100 different queues based on the length of the input string. Benign, small input strings (*i.e.*, nine out of ten requests) always had available resources.

Further mitigation is possible. Parallel spawn of (i) 500 compartments took 2.82 seconds, and of (ii) 5K compartments took 24.3 seconds, indicating good elasticity characteristics until system administrators act upon notifications.

8.3.4 Performance

What is the performance overhead of spawning modules in their dedicated compartments using BreakApp? We opt for single-module, single-compartment setups over multi-level compartmentalization to zoom into the exact sources of overhead under various configurations. We use a diverse set of eight modules to isolate and account for various different behaviors (*e.g.*, interposition, RPCs, *etc.*). These modules are running under the Node.js framework, serving HTTP requests. Table 8.7 summarizes the source-level aspects of the modules used. Generally, lines of code and files correlate with import times; number of nodes, depth, average fanout correlate with interposition transformation costs; and the number of functions correlates with the function-to-RPC transformation costs — a much heavier transformation compared to interposition proxies.

Fig. 8.5 breaks down startup latencies into various sources (*e.g.*, transformations, interposition *etc.*) between the main four different compartment types. `IPC` was set to `TCP` to account for its heavy setup period (yellow segment; 17.6–35ms); this choice affected communication latencies too (purple segment; 11.6–24.8ms). To ease comparisons, we did not include system-level costs of spawning each compartment; these are presented in Table 8.5. *Startup costs are dominated by the number of modules (*i.e.*, files) read from the file system.* A good example is `cash` where importing all the sources takes 798.2–1049.1ms compared to 138.5ms of launching a new process and 847.9ms of launching a new container. For smaller modules such as `http-verbs` and `left-pad` the overhead of launching the compartment is more pronounced, but these modules tend to be used for long-running processes (*e.g.*, web servers); in these cases, a startup time of few hundred milliseconds gets amortized over a period of days. Transformation overheads were generally on the order of 0.2–1.3ms for the addition of interposition proxies and 0.5–6.1ms for all the rest.

Fig. 8.6 shows their execution latencies. `IPC` is set to `PIPE`; each IPC segment on the figure includes the overhead of a serialization and deserialization pair. To account for a more realistic setup, modules are loaded as part of a larger "no-op" HTTP application which does not do any other processing beyond calling the dedicated module. Latencies are averages over 1K requests following 100 warmup requests. Some of the more processing-heavy modules (*e.g.*, `left-pad`, `tweet-nacl`) were tested under different types of workloads: a small 5B workload and a larger 5MB one.

Since HTTP request and response handling in Node.js dominate latency, compartmentalization overheads (IPC request and response) account for a small part of the overall latency. *Even in the case of the heavier compartment types, they are responsible for 2 to 15% of the overall latency.* The vast majority of this overhead is concentrated on the calling side. Returning results is much cheaper because the return values in our experiments were typically smaller.

The overhead of proxy (interposition) objects is barely visible in these plots, and generally much smaller than initially expected. To understand the costs of object proxying better, we created objects with a fanout of 12 for 8 levels (*i.e.*, with 12^8 internal nodes — roughly .5GB memory footprint). Traversing one million 12-hop random paths to access properties of the object took 167.2ms on the original object and 595.7ms on the proxy-augmented object (*i.e.*, all calls went through the proxy object). To put these numbers into perspective, the object allocation took nearly 16 seconds, meaning that *applications will likely hit other bottlenecks before the overhead of interposition becomes noticeable.*

8.4 Language-based Module-level Compartmentalization

Iris's primary hypothesis is that language-based protection mechanisms can address the challenges of conventional compartmentalization systems. In this section, we confirm this hypothesis by discussing (i) the performance overheads associated with runtime enforcement and how they compare with a state-of-the-art OSMLC system (§8.4.1); (ii) the security protection benefits obtained by the use of Andromeda (§8.4.2); (iii) our experiences from running Andromeda on large-scale, real-world applications (§8.4.3). Among other key points, our evaluation shows that:

- For single libraries, Andromeda's overheads are 1–3 orders of magnitude lower than a prior MLC system (§8.4.1).
- Andromeda's enforcement component protects against known attacks from buggy, subvertible, and actively malicious modules (§8.4.2).
- Andromeda makes it possible to run large, compartmentalized applications, without any soundness issues nor any manual effort spent rewriting interfaces at the boundary (§8.4.3).

Experiments were run on a server with 512GB of memory and 80 Intel Xeon E7-8860 cores clocked at 2.27GHz. No special configuration was made beyond disabling hyper-threading. Our software setup uses Node.js version 6.14.4, bundled with V8 v.5.1.281.111, LibUV v.1.16.1, and npm version v.3.10.10, atop a Linux kernel version 4.4.0-134. For memory experiments, Node.js is launched with `--expose-gc` and `--log-gc` to allow triggering garbage collection and verifying that it ran.

8.4.1 Performance Evaluation

How does Andromeda*'s performance compare with conventional MLC systems?* To answer this question, we perform a series of experiments comparing Andromeda to BreakApp [224]—a state-of-the-art compartmentalization framework that can place *selected* JavaScript libraries in process-isolated compartments. The micro-benchmarks chosen to highlight compartmentalization overheads involve six modules of varying size and complexity (Tab. 8.7). These modules were chosen based on two factors: workload diversity and compatibility with BreakApp. Two of these benchmarks, `lpad` and `nacl`, are tested with inputs of different sizes to better

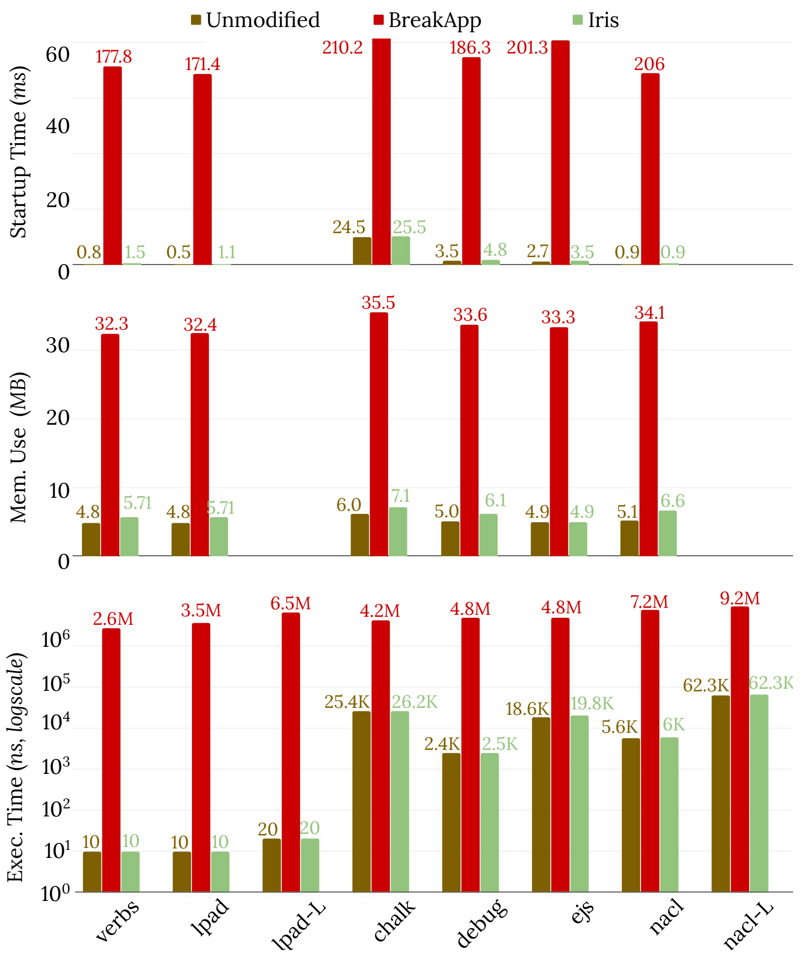


Figure 8.8: **Compartment overheads.** Top: startup times. Med: memory consumption. Bottom: boundary-crossing; as these overheads depend somewhat on the sizes of call arguments, `lpad` and `nacl` include runs with large values (-L) (*Cf.*§8.4.1).

stress the overheads of serialization: a small input of 1B and larger one of 2KB (shown as -L). Results are averages over 1M repetitions, with a warm-up phase of 1K repetitions. We are interested in the following overheads: compartment startup time, compartment memory consumption, and the overhead of crossing boundaries between compartments.

Startup Time The top plot of Fig. 8.8 compares the compartment startup time between Andromeda and BreakApp. Each bar measures the time to complete a `require` call, which includes locating and transforming the module. Compared to BreakApp, Andromeda improves compartment startup times by a factor of 12–168×: compartment creation is accelerated by two orders of magnitude for the heaviest modules such as `chalk`, and three orders of magnitude for lightweight modules such as `verbs` and `lpad`. BreakApp's additional overheads (beyond transformations) stem from setting up a TCP channel (10–20ms), serializing and shipping the return API to the consumer module (5–10ms), and launching a new process executing a copy of the Node.js runtime (80–120ms).

Memory Consumption The middle plot of Fig. 8.8 compares the base memory consumption of individual compartments between Andromeda and BreakApp. Each bar shows the amount of heap memory used by a process, as reported by V8. The values for baseline and Andromeda include only the heap use for the individual module imported. BreakApp's values account for the memory of the new compartment: a fresh copy of the Node.js runtime takes about 20MB; BreakApp adds another few MB, mostly coming from its dependencies to deal with serialization and multiple channel types. Andromeda improves memory consumption overheads by a factor of 19–31×. Notably, in no cases did Andromeda force V8 to request additional memory from the OS—even when a few large modules, such as `nacl` and `chalk`, end up doing this for the baseline. Put differently, while loading large modules may cause performance side-effects that are observable from outside the language runtime (*i.e.*, the OS), Andromeda's overhead remains minimal on top of that.

Boundary-Crossing The bottom plot of Fig. 8.8 shows the cost of crossing a compartment boundary by calling into a module (includes function-invocation time). Generally, Andromeda introduces between two and three orders-of-magnitude lower overheads than BreakApp. The reason is that BreakApp adds serialization, interprocess communication, context switching *etc.*, whereas Andromeda relies only on language-level enforcement mechanisms.

Notably, Iris's performance remains mostly unaffected by the input size (normal and `-L` columns). This is due to Iris passing pointers rather than values between compartments: (i) copying a payload is significantly more expensive, even without considering BreakApp's user-kernel boundary crossing; (ii) serialization and deserialization transforamtions from in-memory values to strings and back. Further experiments (not shown) indicate that BreakApp's serialization overheads are dominated by value expansions due to cycle detection and property ownership. Thus, BreakApp's boundary-crossing cost is a function of the call-argument size.[4]

Take-Away Andromeda *imposes minimal performance overhead, outperforming a state-of-the-art MLC system by several orders of magnitude.*

8.4.2 Security Evaluation

Does Andromeda *succeed in mitigating vulnerabilities that fall under its threat model (§5.3)?* To answer this question, we perform a series of experiments using real and synthetic vulnerabilities (Table 8.8). Real vulnerabilities are drawn from widely-used modules in the Snyk database [200] and include the malicious versions and MITRE's common enumeration identifiers [138, 139]. Synthetic vulnerabilities are constructed artificially to highlight unusual attacks and compartmentalization features: we know these attacks are possible, but were unable to find them *in vivo*. The table's last column shows the PIC size, in terms of number of lines.

Real-World Vulnerabilities For the real-world vulnerabilities, we use the proof-of-concept exploit (PoCE) attached to the original vulnerability report. The `morgan` module is a request logger, which the PoCE exploits to call `console.log`; Iris permits only importing and calling `morgan`. The `merge` module combines multiple objects into one, exploited by the PoCE to alter the object hierarchy with the injection of a `__proto__` property pointing to a new object parent; Iris does not

[4] In theory, Andromeda's overhead is a function of PIC size. In practice, however, PICs are only a few instructions long; being mostly constant values, they are optimized away by V8's tracing compiler.

Table 8.8: **Vulnerable Modules.** A set of real (top) and synthesized (bottom) vulnerabilities mitigated by ANDROMEDA.

	Module	Version	Attack Type	CVE	PIC
1	`morgan`	1.9.0	remote code exec.	2018-3784	9
2	`merge`	1.1	prototype pollution	2018-16469	15
3	`mathjs`	3.16.5	command injection	2017-1001002	6
4	`ns`	all	unsafe serialization	2017-5941	6
5	`st`	0.2.1	directory traversal	2014-3744	13
6	`glob.js`	—	access globals	—	4
7	`mod.js`	—	read module cache	—	6
8	`arg.js`	—	expose process args	—	4
9	`env.js`	—	environment variables	—	4
10	`arr.js`	—	inspect array values	—	5

permit overwriting hidden properties, such as `__proto__`. The popular `mathjs` module includes a math evaluator, which the PoCE exploits to `console.log` a message; ANDROMEDA blocks it similar to `morgan`, not allowing use of `console`. The `ns` serialization module is exploited by the PoCE to import `child_process` and call `exec`; ANDROMEDA blocks this, by only allowing the `ns` import itself. The `st` module offers HTTP routing and caching that is used in conjunction with an HTTP server. It is susceptible to directory traversal: while dots (*e.g.*, `../`) are filtered out, url-encoded ones are not (*e.g.*, `%2e%2e/`). The PoCE attempts to access the filesystem root, but IRIS allows access only in the current directory (§5.4.4).

Synthetic Vulnerabilities For the synthetic vulnerabilities, we prepare a malicious module `mm` that attempts unauthorized access of global variables, the module cache, the process arguments, environment variables, and values of an importing array `am`. Apart from `am`'s first element, all other accesses are disallowed. IRIS launches `mm` in its own compartment, shielding it from `ma` and its surrounding environment with several PICs. All of `mm`'s accesses are blocked by ANDROMEDA, with the granularity of `am` being particularly telling: IRIS can trivially distinguish across array elements, as they are object fields that just happen to have integer identifiers as

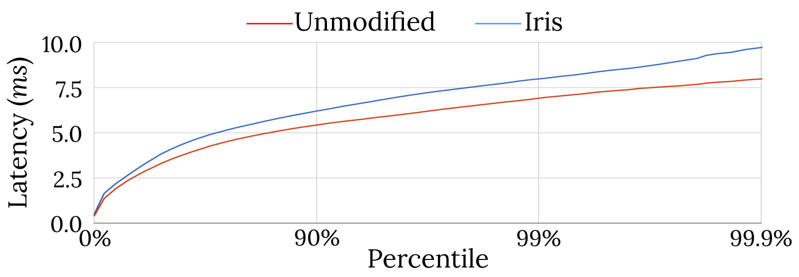


Figure 8.9: **Performance of Andromeda-augmented wiki.** High-resolution cumulative latency distribution, highlighting tail latencies at various percentages.

names.

Take-Away ANDROMEDA *can successfully protect against both known and new attacks, at a very fine granularity.*

8.4.3 Multi-module Evaluation

How does IRIS *perform in larger applications? In particular, does it solve the soundness and rewriting concerns of conventional MLC?* To answer this question, we run two experiments on multi-module applications. Our goal is to focus on potential soundness and interface rewriting issues that challenge conventional MLC, even in the full absence of disallowed behaviors. To avoid disallowing any behaviors, we generate PICs that permit all calls. PICs update invocation metadata to their shared global state (§5.4.5), to avoid V8 optimizing them away.

Synthetic Application In the first experiment, we run IRIS on a small synthetic application comprised of two modules, each one importing two additional modules (total: six imports). Modules hold a consecutive integer as their private state. When called, each module increments its integer by six and appends to a global array. We run the benchmark for several thousand repetitions of 1M calls, and confirm that the global array is always sorted—that is, PIC execution does not introduce any unsoundness due to interleaving. We were not able to detect performance differences between the vanilla and IRIS-augmented application (beyond statistical noise).

Real Application In the second experiment, we run ANDROMEDA on an open-source wiki application comprised of 130 top-level modules, 1640 total modules, and about 597K lines of code. We use `wrk2` [209] from the same physical host to send HTTP "No-Op" requests. The `wrk2` tool produces a load of constant throughput and calculates latencies at various tail percentiles (*e.g.*, 90%, 99%, 99.9% *etc.*). We configure `wrk2` with 2 threads and 100 connections, at about 1.07k requests per second.

The primary result is that IRIS's compartmentalization enforcement does not require manual effort, demonstrating that decomposition remains feasible without any module interface rewriting. In terms of performance, the unmodified wiki responds with an average latency of 4.2ms; introducing IRIS bumps latency to 4.8ms. Fig. 8.9 shows that for the majority of the requests (90%), the overhead introduced by ANDROMEDA is under 1ms. Changes in memory consumption, which averages around 100MB, remain below 0.01%.

Take-Away ANDROMEDA *avoids introducing unsoundness and interface rewriting, making large-scale module-level compartmentalization feasible.*

8.5 Light-Touch Distribution

Several takeaways are worth highlighting. Scaling out with IGNIS can require minimal code changes, less than 0.001% of a complex codebase (§8.5.5). Even for simple applications and scaling goals, this represents 10-20× less effort than manual approaches and avoids the introduction of bugs (§8.5.3). In a case study where we scale a web crawler to better utilize a 60-core host, IGNIS leads to speed-ups of 27× (§8.5.4). Aside from speed-ups, IGNIS reduces memory requirements by up to 11× compared against typical whole-application replication, by only replicating bottlenecked components (§8.5.5).

Table 8.9: **Replication overheads.** Replica (i) startup costs (rows 2, 3) and (ii) communication costs (rows 4–8), as a function of the number of replicas (*Cf.*§8.5.1).

Module Replicas (Single Host)	5	50	500
Startup Overheads:			
Latency (Unmodified)	12.9ms	106.6ms	824.7ms
Latency (IGNIS)	342.5ms	1.4s	6.2s
Communication Overheads:			
Latency (Unmodified)	6.5ns	90.18ns	294.3ns
Latency (IGNIS)	27.15ms	388.55ms	11.05s
Throughput (Unmodified)	192.3GB/s	157.1GB/s	46.5GB/s
Throughput (IGNIS)	158.1MB/s	134MB/s	20.9MB/s

8.5.1 Micro-benchmarks

No-op Modules To understand IGNIS' inherent startup and communication costs, we run a few microbenchmarks comparing IGNIS with the unmodified module system as a function of the number of replicated modules. We remove orthogonal network concerns by running experiments locally on $\texttt{a}_1$, and minimize the effects of module sizes and transformations by creating "no-op" modules.

Tab. 8.9 rows 2 and 3 show the base startup time of modules that return a single integer. IGNIS modules incur significantly higher startup times, but these overheads are amortized as the number of modules increases.

Vanilla JavaScript loads modules sequentially. However, IGNIS-replicated modules can amortize their (much more expensive) startup costs by leveraging parallelism—an idea that (at least in part) motivated asynchronous spawning (§6.5.1). Asynchronous spawning raises concerns regarding interference with the main process on a single host. To investigate this, we start 5K modules in parallel (total time: 15.4s, avg: 3ms/module) while a main module "mines" SHA512 hashes. Hash rate, about 5.4 MH/s, essentially remains unaffected.

Tab. 8.9 rows 5–8 show the cost of communication (remote invocation), by processing an in-memory stream of 1GB across a series of modules. The stream starts only after all connections have been established (*i.e.*, no TCP handshake costs included). As expected, copying the payload (rather than passing pointers) is significantly more expensive.

Real Modules To understand startup and communication costs on real modules, we run single-replica experiments on a diverse set of popular modules under three configurations: (i) vanilla, (ii) co-located distribution (as before), (iii) networked distribution. Their source-level aspects (Tab. 8.7) affect startup times, whereas call argument sizes affect invocation times.

Fig. 8.10 shows module startup costs. Distribution requires transforming modules (RPC stubs), communicating them to the remote node, and launching replicas there. The startup time of larger modules is dominated by file-system accesses (*e.g.*, `cash` takes 798.2–1049.1ms to load all of its files). The startup time of smaller modules, such as `verbs` and `pad`, is dominated by constant factors (*e.g.*, 138.5ms for process spawn and 17.6–35ms for TCP setup).

Fig. 8.11 shows inter-module communication costs over the same configurations. To bring these costs into perspective, we embed modules in "no-op" HTTP applications. By placing these modules behind an HTTP server, we can study IGNIS' impact on the end-to-end latency an HTTP client would experience. Processing-heavy modules such as `pad` and `nacl` are tested under input of size 5B (`S`) and 5MB (`L`).

To understand the costs of boundary interposition, we measure the time to access deeply-nested properties of two versions of an object: unmodified and proxy-wrapped. Paths to the properties (*e.g.*, `a.b.c....`) are random but generated prior to running the experiment. We construct 500MB-sized objects, each with a fanout of 8 fields (DAG child nodes) nested for 12 levels. The proxy-wrapped version introduces interposition at every level. Traversing one million 12-edge paths (*i.e.*, root to leaves) averages 167.2ms and 595.7ms for the unmodified and proxy-wrapped versions, respectively.

Towards Practical Overheads The micro-benchmarks presented in this section

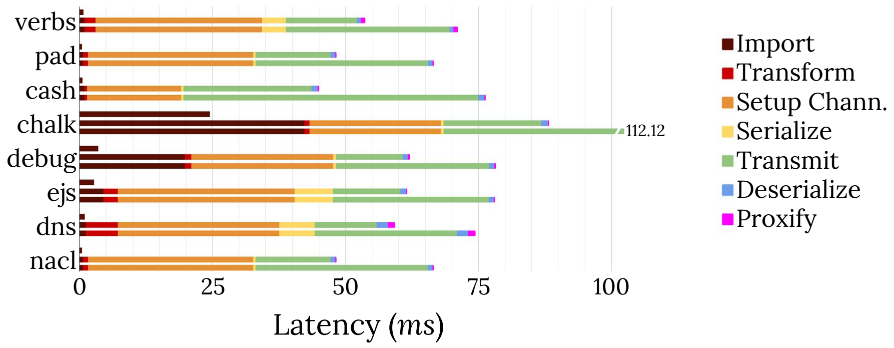


Figure 8.10: **Startup latency breakdown**. Each module is measured under three configurations: unmodified (short bar, only `import`), co-located on the same physical host (medium bar), and scaled out across the network (*Cf.*§8.5.1).

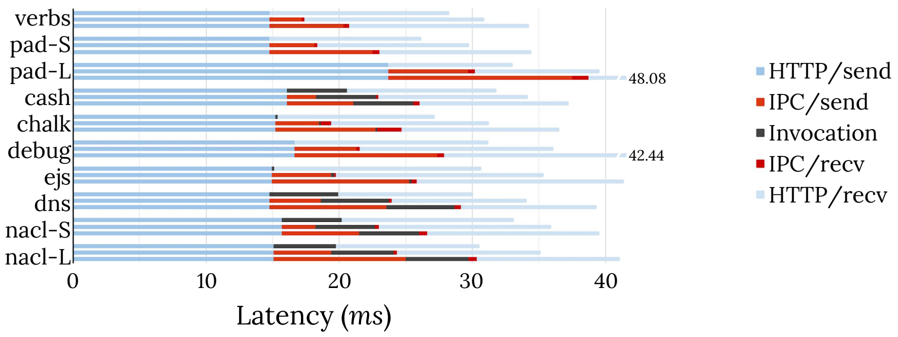


Figure 8.11: **Call latency breakdown**. Configuration as in Fig. 8.10 (*Cf.*§8.5.1).

highlight Ignis-inherent overheads by using carefully-constructed, worst-case workloads that are nowhere near the ones seen in practice (§8.5.3–8.5.5): 1GB-sized function-call arguments, deeply-nested 0.5GB-sized module interfaces, and near-zero module-internal latencies (that in practice would not lead to scale out).

8.5.2 Synthetic Applications

To better understand Ignis' profiling and distribution in a controlled environment, we craft a four-module application with pre-defined bottlenecks. The application can perform one of three operations, depending on its current state: (i) invoke a call to a module it imports, (ii) busy-wait, or (iii) accept calls. State transitions are controlled by a probability distribution; and modules can introduce additional

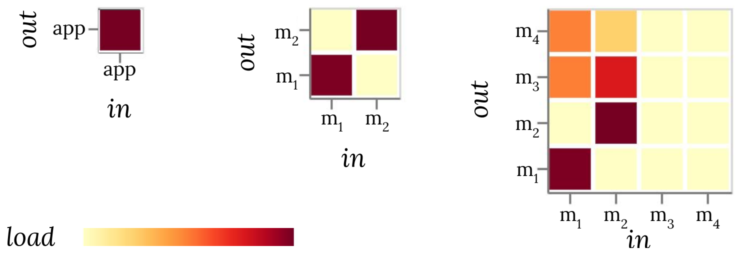


Figure 8.12: **Boundary pressure.** The dependency graph of an application with four modules is presented as an adjacency matrix at different granularities. Instead of a single bit, cells of adjacent modules are labeled with the heat (load) of the boundary they represent, as recorded by the IGNIS coordinator. The diagonal represents pressure internal to a module (*Cf.*§8.5.2).

delays. The application accesses modules `m1` and `m2` with equal probability; `m1` accesses `m3` and `m4` similarly, but `m2` accesses `m3` three times more often than it does `m4`. The four modules add a delay of 10ms $\times$ their depth in the dependency graph. The application is part of IGNIS' testing infrastructure, encoding well-understood scenarios with the goal of verifying that scale-out follows the developer's intuition.

Fig. 8.12 depicts the importance of fidelity in load attribution. Load is visible at the application level (left), but the lack of detail does not help determine which modules must be scaled out. Module-level data, collected by IGNIS' coordinator (right), reveal that `m3` receives the majority of the load.

Manually deciding which module to scale out would require installing `linux-tools`, setting up `perf` [77], resolving JavaScript symbols with V8's `--perf-basic-prof`, recoding event samples, mapping samples to modules (using stack information), and visualizing results. Many of these steps would need to be repeated for every new bottleneck and, after scale-out, combine results from multiple replicas.

IGNIS reaches the decision to scale out within a few hundred milliseconds, primarily because of lack of confidence due to cold-start effects. By observing long inter-module queues and a sufficient number of idle processors, IGNIS launches three replicas of `m3` at once.

8.5.3 Macrobenchmark: A (very) Simple Weblog

Scaling out a system is often as easy as upgrading to a readily-available distributed storage system. This is the simplest case of bolt-on distribution, because it does not require thinking deeply about the structure of the computation. To compare with such minimum-effort rewrites, we plug a distributed key-value store into a simple blog application.

`KoaBlog` [88] (commit 1fd5316) is a small application for learning the `Koajs` web framework. The blog manipulation code totals 50 lines and imports six direct dependencies, for a total of 160 packages and 96 KLoC. Entries are indexed by an integer ID. They are stored on disk using methods from the built-in `fs` module (*e.g.*, `write`, `readdir`).

Even though `KoaBlog` is a trivial application, the manual effort required to use a NoSQL system such MongoDB [41] (v.3.4.10) is considerable. We first use `npm` to download and import the `monk` module for interfacing with MongoDB, similar in effort to importing Ignis. We then create a database schema, and configure the connection to the master node—including details such as binding address, port, and username. We also remove the import of the `fs` module, and rewrite all file-system operations such as `read` and `write` to make use of `monk`'s `find` and `insert`. All the above is expressed in JavaScript and within `KoaBlog`, in a diff that totals 11 lines (22% of end-developer code). However, it omits lines typed in the MongoDB and Unix shells (outside `npm` and requiring `sudo`): fetching and installing MongoDB, configuring administrative users, setting up one master and two slave nodes, and connecting MongoDB's startup with `KoaBlog`.

Adapting `KoaBlog` also required fixing two bugs that we unintentionally introduced. First, by replacing *all* `fs.write`s with `insert`s, we broke updates; `fs.write`s corresponding to updates should have been replaced by `updateById`, not `insert`. Second, we misconfigured the binding address and port of one node. These bugs were not difficult to fix, but illustrate the inherent dangers of manually scaling out an application, even when the modifications *seem* straightforward.

In contrast, with IGNIS this scale-out requires a short recipe at the `fs` import: `require("fs", {copies: 3})` (aside from downloading and importing IGNIS, à la `monk`). The recipe specifies a minimum and a maximum number of three replicas, similar to Mongo's setup. IGNIS launches three replicas of the `fs` module (see "Environment Binding" in §6.5.2 for built-ins), traverses the return DAG of the original `fs`, and rewires methods such as `readFile` and `writeFile` to call Andromeda's distributed storage equivalents. Node.js's `fs` methods take optional arguments such as encoding (*e.g.*, UTF-8) and access mode (*e.g.*, `RW`). As Andromeda stores objects (rather than files), Unix flags such as `RW` were initially a concern; however, existing support for UTF-8 was enough to avoid breakage.

To evaluate performance, we pre-populate `KoaBlog` with 10K posts of 1.1MB each, and issue a 2-minute HTTP `GET` workload of 5K req/s. To saturate disk bandwidth and "force" IGNIS to scale out, we initiate three local, parallel, long-running `cp` commands in the background. Distribution improves request latencies significantly (table below) as well as request throughput and transfer rates: 24 requests

Percentile	50%	90%	99%	99.9%
Baseline	154ms	1028ms	4731ms	8905ms
IGNIS	65ms	231ms	318ms	1187ms

per second (28.2MB/s) become 88.22 (103.5MB/s). No significant difference between IGNIS and MongoDB was noticed; this is expected because no advanced indexing, replication, and consistency features were used, where the two diverge.

These performance improvements are on top of the base benefits of storage distribution—increased capacity (combined, $\texttt{q}_{\texttt{1-4}}$'s disk capacity is over 1TB) and availability (3× replication). (This availability is different from the fault-tolerance of IGNIS *itself*, which is left for future work (§10.4).) The introduction of IGNIS does not impact the memory consumption of the main node. This is because the `koa-*` modules dominate, whereas the newly-introduced `ignis` module is comparable in size to `fs`.

8.5.4 Macrobenchmark: Document Ranking

For a more complex application, we consider a custom natural-language processing (NLP) pipeline that is used as part of a larger web crawler application. As documents arrive, the NLP pipeline extracts word stems, removes stop-words, normalizes terms, creates n-grams of sizes 2–5, and runs frequency analyses. The pipeline is built around v0.6.3 of `natural`, a third-party package for NLP, and applied on 200 books, each averaging 1MB, from the Project Gutenberg corpus [84].

Scale-out depends on the relative overheads of different NLP stages. Skipping the manual effort of profiling (described at the end of §8.5.2), manual scale-out would need to extract the interfaces of used modules, generate RPC stubs (*e.g.*, gRPC [212]) and load them remotely, start communication servers, and balance load at runtime. We did not attempt this; instead, we added `require("ignis", {hot: ["natural"]})`.

The centralized version processes at a rate of 22.2 documents per minute (2.7s per document). Ignis improves throughput by a factor of 27× to 612.8 documents per minute (97.9ms per document). While Ignis sees gains in launching more processes on the local host, it does not see any gains in further distributing across physical hosts. The reason is that the crawler does not feed the NLP pipeline with documents at a rate that is high enough to benefit from networked distribution (in our setup).

8.5.5 Macrobenchmark: Wiki Engine

For a complex application, we turn to `wiki.js` (2.0.0-dev), a popular wiki engine that imports 130 top-level modules; counting recursive imports, the total jumps to 1640 modules and about 597K lines of code. We augment `wiki.js` with `uri-js` v2.1.1, an extensible URI parsing and validation library that is fast in the average case but can sometimes spend upwards of 400ms per URI in pathological edge cases. This leaves `wiki.js` susceptible to denial-of-service attacks [45] (ReDoS) and slowdown in certain non-adversarial workloads. We use IGNIS to mitigate this with a simple two-line recipe (*i.e.*, less than 0.0005% of the codebase) that scales out `uri-js`.

Under normal operation, when no URIs in the workload invoke the edge cases in `uri-js`, IGNIS introduces little runtime overhead. Issuing an HTTP load of 5Kreqs/s on `wiki.js`'s sample dataset, the unmodified wiki responds with an average latency of 34.1ms (σ: 2.1ms). Introducing IGNIS bumps latency to 34.3ms (σ: 2.8ms). Changes in memory consumption, which averages around 100MB, remain below 0.01%.

With a workload that contains even a small fraction of pathological URIs, the benefits of IGNIS become significant. Servicing a workload with 99% benign URIs and 1% pathological URIs, the throughput of the unmodified wiki drops to 197req/s (15.2s per request, σ: 11.04s). IGNIS, on the other hand, achieves a throughput of 208req/s (8.1s per request, σ: 4.9s) when distributing to $\mathtt{q}_{\mathtt{1-4}}$'s four network replicas. When distributing to 60 replicas on $\mathtt{a}_{\mathtt{1}}$, it achieves a throughput of 1,880req/s (42.66ms per request, σ: 1.35ms). IGNIS detects the load pressure applied on `uri-js` within the first few pathological requests, scaling out in under half a second.

Memory consumption of each replica is about 17.3MB, the vast majority of which comes from the Node.js runtime and libraries. For comparison, naive application replication leads to a total memory footprint of 1.1GB (on top of the "master" `wiki.js` consumption).

Chapter 9

Related Work

The techniques employed in this dissertation are related to a large body of previous work in several distinct domains. While many individual features have close analogues in the literature, ANDROMEDA provides a novel synthesis to furnish a distributed userspace. To complement the high-level discussion presented in the introduction (§1.3), this section focuses on more technical discussion between of these features.

9.1 Distributed Environments

ANDROMEDA draws heavily from prior work in the broad field distributed environments.

Language-based Systems ANDROMEDA shares many similarities with language-based DOSes [111, 116, 25, 38, 136, 56, 243], which used programming language abstractions to lower the complexities of distribution; ANDROMEDA, however, does not require the use of a new programming language. The asynchronous programming model unifying local and remote access is similar to Argus [116]. Value mobility, object prototypes, and the per-node performance focus are close to Emerald's, but we address its "locatics" [25] concerns with node groups and configurability.

Andromeda's focus on extreme portability, userspace hosting, and high-level interpretation combined with JITing is similar to Inferno [56, 243], but its (asynchronous) semantics is closer to Actors [6] than Limbo's CSP [87] style.

Several approaches attempt to graft a single-system image (SSI) atop an existing language [12, 251, 27]. However, they usually do not provide transformation primitives, impose a cluster-aware version of the runtime, and do not include a library of services. Andromeda shares (distributed) Smalltalk's [17] extensibility and introspection and its ancestry on interface unification (and minification) from Dataless Programming [13].

Distributed Operating Systems Andromeda shares many similarities with DOSes [169, 236, 40, 159, 143, 166, 178, 56, 14]. Andromeda's group services are related to V [40] and Chorus [178]. Its architecture—*i.e.*, key services outside the core, the naming service *etc.*—is especially close to Amoeba's [143]. Andromeda's local execution binding techniques are similar to Plan 9's [166] namespacing, and its configurability is similar to Nemesis' [176] QoS guarantees (although both at a different level).

Recent proposals to revisit DOSes [189, 182, 121, 93] share Andromeda's intention to revisit older ideas in distributed systems; however, they do not aim at leveraging high-level programming-language abstractions nor at lowering employment complexity—but primarily improving raw performance. Their focus is current workloads on mostly-homogeneous datacenters whereas Andromeda enables programming "in the small" [53] across very diverse owned resources. Among these systems, DiOS [189, 188] and Taurus [121, 122] stand out as more relevant to Andromeda.

DiOS is a distributed operating system for datacenters implemented as a Linux kernel extension module. Although it eschews the POSIX API, its abstractions remain close to those of Unix. Contrary to Andromeda: (i) it is a language-agnostic system that can run ELF executables, (ii) it provides Unix-like primitives such as

FIFOs, processes, namespacess, and (iii) it is written in C and does not assume any (specific) language-level integration of its object notion. Moreover, services from the kernel are not distributed: the file system is flat, but not distributed; there is no node and group management; and there are no primitives for distributed execution. Contrary to ANDROMEDA, it supports legacy applications by having them not use the DiOS syscall API. That is, legacy applications do not see any distribution benefits from running on a DiOS system (except possibly Firmament scheduling by linking with TaskLib). This is different from ANDROMEDA where legacy applications still benefit from distributed system services. Developers in DiOS need to rewrite applications to use the new API, which DiOS aids by exposing a "Limbo" ELF type for binaries that require access to both APIs.

Taurus is a holistic runtime system, in which language runtime environments running on individual nodes coordinate with one another for tasks such as garbage collection, just-in-time compilation, and safety/type checking. While Taurus is similar to ANDROMEDA in that it focuses on high-level runtime environments, it solves a different (and complimentary) problem: performance pathologies arising from lack of runtime coordination. It does not provide abstractions for composing or employing distributed applications. ANDROMEDA's performance would definitely benefit from inter-node coordination, which could be provided as a pluggable service; the garbage-collection hooks used in BREAKAPP are a limited version of this functionality. Conversely, while Taurus is today implemented as a JVM co-process, its mechanisms and policy domain-specific language (DSL) could benefit from the approach of a distributed userspace. Taurus' separation between mechanism and policy, and specifically policy expressions via through a high-level DSL, served as the inspiration for ANDROMEDA's configurability as a remedy to distributed-systems trade-offs.

Unconventional Operating Systems ANDROMEDA shares the treatment of multiprocessors as distributed systems with Disco [33], Cellular Disco [75] and Barrelfish [16]. Similar to unikernels [124, 206], ANDROMEDA runs in a single address space, does not support multiple users, and can be used as a libOS, but it is not intended for executing a single application and does not depend on hypervisors for key functionality. Like Mirage, Andromeda is written in a high-level programming language, uses language-based guarantees for correctness, and depends on memory safety for protection; but does not provide statically checkable guarantees (§10). These systems do not provide a unified distributed userspace.

Component Architectures Andromeda shares philosophy of services as toolkit-libraries. Similar to Andromeda, Horus [218] identifies multifaceted trade-offs of distribution and attempts to provide flexible communication abstractions in a group; however, its specification is lower-level than Andromeda's semantic configurations. ISIS [23] and Circus [44] exploit development-time module structure to replicate modules and provide fault-tolerance. In contrast to Andromeda, these systems operate statically at compile time, without runtime introspection.

Transformations Andromeda's transformations can be seen as aspect-oriented programming (AOP), a programming model that maps program join-points to actions to be taken at these points [99]. In fact, they are enabled by the language's support for metaobjects [100], objects that manipulate object structure, enabling a program to access its own internal structure, including rewriting itself as it executes. Metaobjects are examples of runtime reflection, of which Andromeda makes extensive use to traverse, understand, and rewrite interfaces. Andromeda tries to minimize AOP and metaobject complexity with high-level semantic configurations, rather than requiring developers to alter or introduce code. This is related to recent transformation work that attempt to alter the semantics of the original program in principled ways [173, 163, 197, 198].

9.2 Distributed Storage System

Data Structures High-dimensional database techniques have been of interest to the data-base and data-structure communities for decades (*e.g.*, k-d trees [20], R-trees [81], the grid file [149], z-order [157], R+-trees [192]). They either require a static object (*i.e.*, table) structure [149, 157] or, more commonly, adapt on the data they see [20, 81], which makes straightforward distribution very difficult.

Dimensionality Reduction Other lines of work [141, 91, 72] focus on reducing dimensionality down to a single dimension to then enable single-dimension partitioning schemes. Space-filling curves [141] do this, for example, by tracking a single curve through all regions of a multi-dimensional space. This is a promising direction for dealing with very high dimensionality, and is somewhat similar in spirit to mapping a multidimensional matrix into a single region of physical memory, as done in our system. However, scalability may be hindered by (i) multi-dimensional queries partitioned into single-dimensional ranges of different sizes, and (ii) space regions falling under multiple nodes.

Locality-preserving hashing [91] and locality-sensitive hashing [72] offer multi-dimensional range and nearest-neighbor (*i.e.*, fuzzy, wildcard) queries. However, since they are sensitive to data for preserving locality, they do not offer good load-balancing features. In some cases, space partition is based on previously-seen data – so different nodes might have inconsistent views of the space – and offer only approximate guarantees [72].

Distributed Hashing On the other hand, distributed hashing techniques provide uniform load balancing without depending on previously-seen data. Somewhat more complex hashing schemes (*e.g.*, rendez-vous hashing [211], consistent hashing [96]) can minimize data shuffling in dynamically-changing systems. These ideas have been used in various settings (*e.g.*, P2P [204, 177]) and led to the revolution of distributed, single-dimensional (*i.e.*, key-value) NoSQL storage systems (*e.g.*, Dynamo [51], Redis [184], Cassandra [107]). Hyperdex [62] improved on the idea in many ways (§3.2). Our work extends Hyperdex's partitioning technique, namely hyperspace hashing, for data whose structure is not known beforehand and can change during runtime. Replication and consistency are orthogonal issues and can be served by schemes complementary to ours (*e.g.*, Hyperdex [62], Replex [207]).

9.3 Automated Module Sandboxing

Supply Chain Attacks Our concerns about large-scale reliance on loose supply chains are echoed by both academia [123, 110] and industry [118, 36, 200]. Academic studies have shown the increasing risks of the reliance on third-party code (although they generally do not consider the scope of problems we do in §4.3). Several recent companies [190, 156, 200] provide third-party module assurances by having more people audit and recommend packages in the wild or crawl public repositories for open vulnerabilities. In practice, they do not offer any guarantees similar to compartmentalization, but can be used complementary to our work: users (or libraries that are built on top of BreakApp) can use these recommendations to choose which modules to quarantine. Package managers have added support for locking dependencies between deployments [151]. However, this does not necessarily rule out extant problems; on the contrary, users forego valuable bug and vulnerability fixes, while experiencing a more convoluted dependency management.

JavaScript Isolation In the case of JavaScript specifically, much effort has gone into client-side compartmentalization (*e.g.*, execution isolation [132], object capabilities [135], sandboxing [210, 7] information flow control [203]). These works focused primarily on client-side, web-based setups which are different from our focus in many ways: isolation primitives (*i.e.*, iframes), origin (*i.e.*, explicit sources), threat model (*i.e.*, no C/C++ modules; no valid access to "/etc/passwd"), compartment numbers (*i.e.*, few), and developer effort (*i.e.*, manual annotations or rewrite).

Microservices Microservice architectures, a style for building server applications as sets of loosely-coupled components [67, 147], are often touted as enabling fine-grained, Least-Privilege decomposition inspired by the Unix philosophy. Even more so, lambda architectures [60, 85] are emerging as a lighter-weight, evolutionary step beyond microservices that use runtime contexts to offer improved elasticity. In practice, however, both are vastly more coarse-grained than the applications shown here, with each microservice usually built on top of hundreds of packages similar to the server-side applications outlined in Table 8.3. Moreover, (i) communication between services is request-response style and usually explicitly exposed to the application,[1] (ii) decomposition is a manual process that requires a careful design process (including agreeing on the interfaces) prior to development. These are antithetical to our technique that hides the underlying compartment boundaries.

Containerization Many other system-level sandboxing primitives can be used (*e.g.*, SELinux [164], AppArmor [15]). We experimented with Docker [129] primarily because it is becoming an industry standard: most users are expected to use (and seek numbers about) Docker more than any other container infrastructure. Coarse-grained compartmentalization, however, such as by wrapping a language runtime with a layer of virtualization, is ill-equipped to address risks described in § 4.2.1 and 4.3: a malicious module is still able to access trusted state, exfiltrate data, or launch a DoS attack.

System Decomposition There is a long history of alternative system structures with a focus on least privilege decomposition [183] and, more generally, separation of concerns [55] (*e.g.*, microkernels [4, 114, 86], capability systems [113, 195], and separation kernels [179]). Increasing security requirements brought decomposition to the foreground [167], culminating with systems such as Crowbar [24] and SOAAP [78] that assist programmers into decomposing applications into multiple compartments with reduced privileges. Other systems have focused on abstractions and mechanisms

[1]It usually relies on HTTP — much more heavyweight than the channels described in this work.

that allow efficient separation [54, 238, 237]. Despite their advances, systematic adoption has been impeded by the lack of automation [123], the primary focus of BreakApp.

Unsound Program Transformations There is a recent emergence of unsound program transformations [173, 163, 197, 198] to mitigate failures and attacks via self-healing. Such transformations change the semantics of the original program in principled ways. Our work can be seen as a set of potentially unsound transformations at the module boundaries: users can decide how to break the semantics of the program by choosing which behaviors to disable. However, our transformations are proactive rather than reactive. They are also based on the assumption that there are cases in which a legacy program runs the risk of breaking *either way*: developers can choose whether it will be due to third-party modules (unprincipled way) or due to altered semantics (principled way).

9.4 Language-based Protection

Andromeda's goal is to protect applications against untrusted code—a concern echoed by both academia [123, 110] and industry [118, 36, 200]–and thus can be compared to alternative mitigations.

Ecosystem-focused Non-academic response [190, 156, 200] has focused on (i) having more humans vet and recommend (or recommend against) certain packages, (ii) aggregating such data and combine them with data from source-code repositories, and (iii) using the combined data in tools that check dependency chains for packages that are vulnerable. Unfortunately, this process works only with *known* vulnerabilities and does not offer any guarantees close to compartmentalization. However, it can be used complementary to our work: users and libraries can use these recommendations to choose where to focus Andromeda's PICs.

More recently, package managers have added support for locking dependencies between deployments [151]. However, this does not necessarily rule out extant problems; on the contrary, users forego valuable bug and vulnerability fixes, while experiencing a more convoluted dependency management.

The following few paragraphs outline more principled approaches and how they relate to ANDROMEDA. While industrial response placed utmost importance on maintaining backward-compatibility, this goal is not shared with the vast majority of academic approaches—a major departure from ANDROMEDA.

Decomposition Separation kernels [179] and μ-kernels [4] pioneered manual decomposition in an attempt to lessen the privilege of buggy or malicious components. Privilege-separated system-level applications, such as OpenSSH [167], demonstrated tangible security benefits over monolithic approaches. Manual decomposition enables full control over compartment structure and boundary placement, but comes at a significant productivity cost.

Followup approaches proposed reusable frameworks that can automatically spawn compartments and connect communication channels, at the cost of *lightweight annotations* on program objects—*e.g.*, configurations in Privman [102], `priv` directives in Privtrans [32], tags in Wedge [24], hypotheses in SOAAP [78], and policies in ACES [43]. Annotations typically express a small number of sensitivity levels, translating to an equally small number of compartments.

To ameliorate manual annotations on individual objects, recent attempts exploit runtime information about module boundaries. Unfortunately, module-level compartmentalization [144, 224, 108, 213] does not scale beyond a few modules, as it suffers from several practical challenges—potential introduction of unsoundness, coarse granularity, and significant performance overheads (§5.2.2).

Software Isolation Replacing OS protection with language-based one is not a new idea. Software fault isolation [235] rewrites object code of modules written in *unsafe* languages to prevent them from writing or jumping to addresses outside their domains. Singularity's software-isolated processes [8] ensure isolation through software verification. Leveraging memory safety, Andromeda can be applied in environments with *runtime code evaluation*, for which binary rewriting and software verification might not be an option.

Language-based information-flow control [180, 229] (IFC) emphasizes data flow over code structure and, like Andromeda, depends on memory safety. IFC is more powerful than code-oriented compartmentalization but with performance and usability implications. Power comes from the fact that it provides semantic assurances over entire data flows for multi-user systems that combine many sensitivity levels—often allowing untrusted code to see sensitive data by blocking its subsequent communication. Unfortunately, fine-grained tracking and checking usually translates to higher performance costs. Andromeda decomposes at intuitive trust boundaries and granularity (modules), has negligible performance costs, and remains backward-compatible with existing codebases.

While our techniques are not JavaScript-specific, prior work in JavaScript has largely focused on the client: execution isolation [132], object capabilities [135], sandboxing [210, 7], and information flow control [203]. Although isolation primitives (*i.e.*, iframes), script origin (*i.e.*, explicit sources), threat model (*i.e.*, no system access), compartment numbers (*i.e.*, few), and developer effort (*i.e.*, manual rewrite) are all different, it would be interesting and worthwhile to see adoption of Andromeda's techniques on the web.

Capabilities Capability systems [112, 113, 195] place resource restrictions by distinguishing data values from pointers. *Object*-capability systems, such as Andromeda, achieve this conveniently by restricting the ability to *name* a resource, essentially treating the object reference graph as an access graph.

To make capabilities more widespread, efforts such as Joe-E for Java [130] and Caja for JavaScript [135] restrict popular languages to object-capability-safe subsets. Iris exploits the structure of the dependency graph to introduce *backward-compatible* capability-like (membrane) wrappers, retrofitting security on vanilla JavaScript—

without any restrictions–by leveraging meta-object capabilities provided by the language itself.

Contract and Metaobject Systems Andromeda's PICs are a special case of contracts—executable partial program specifications—for specifying privilege. Contracts were pioneered by Eiffel's "design-by-contract" methodology [131] and have seen widespread use [161, 65, 234, 103, 97, 217]. PICs fit in *latent* contracts [65], purely dynamic checks that are transparent (and orthogonal) to the type system, as opposed to *manifest* contracts [76], in which types record the most recent check that has been applied to each value. Andromeda restricts the privilege of code in the contracts by leveraging its security-monitor transformation infrastructure. Andromeda's exceptions offer first-order information about which modules are responsible for a violation, contrary to sophisticated (and more expensive) *blame* assignment and propagation in contract systems.

Andromeda's contract-enforcement transformations are made possible by metaobjects [100]—objects that manipulate objects, enabling a program to access its own internal structure. This functionality is inspired by Self's mirrors [215], Smalltalks's method wrappers [29], AmbientTalk's mirages [142], and E's proxies [136]. Andromeda's users do not need to provide their own implementations of metaobjects.

Programmatic Transformations Unsound program transformations [173, 163, 197, 198] attempt to alter the semantics of the original program in *principled* ways, to enable self-healing, failure-recovery, and increased security. While Andromeda alters the semantics of offending components (manifested as `PrivilegeExceptions`), it does not alter those of benign code; transformations do not introduce any form of concurrency.

9.5 Automated Distribution

Automated Parallelization There is a long history of automated parallelization starting from explicit `DOALL` and `DOACROSS` annotations [34, 115] and continuing with compilers that attempt to automatically extract parallelism [160, 82]. These systems operate at a lower level than Ignis (*e.g.*, that of instructions or loops instead of module boundaries) and typically do not exploit runtime information.

More recent work focuses on extracting parallelism from domain-specific programming models [68, 74, 105] and interactive parallelization tools [98, 92]. While these tools simplify the expression of parallelism, programmers are still involved in discovering and exposing parallelism. Moreover, the insights behind these attempts are significantly different from ours, as they extract parallelism statically during compilation instead of dynamically during runtime.

Distributed Environments A plethora of systems assist in the construction of distributed software. At one end of the spectrum, distributed operating systems [169, 236, 159, 143, 166, 178, 56, 14, 189, 182] and programming languages [232, 194, 101, 59] provide a significant amount of flexibility in the resulting application. However, they involve significant manual effort using the provided abstractions, which are strongly coupled with the underlying operating or runtime system. Light-touch distribution does not make assumptions about the underlying operating system, and makes only minimal assumptions about the language runtime.

At the other end of the spectrum, distributed computing frameworks [49, 146, 248, 145, 168] and domain-specific languages [9, 22, 128, 57, 140] simplify certain patterns, but do not offer the flexibility of a full-fledged environment. Developing under these frameworks differs quite significantly from how developers normally compose applications. In Ignis, developers write general programs as they would do normally—only sprinkling them with "control-plane" insights.

Object-based Distribution Several language-based approaches attempt to provide a single system image (SSI), either under new [111, 116, 25, 38] or existing languages [17, 12, 251]. The latter are closer to Ignis, but focus on SSI rather than component replication (except pure-function replication for cJVM [12]), and do not support dynamic profiling-based scale-out. They also impose a cluster-aware version of the JVM, whereas Ignis comes as a third-party module running on a completely unmodified V8.

Taurus' policies [122] and Terracotta's annotations [27] share the same flexibility-automation philosophy as Ignis' recipes, albeit at different levels. Taurus does not transform non-distributed applications, but is complementary to Ignis: using a holistic runtime would have helped coordinate just-in-time compilation, module spawning, and garbage collection across nodes. Terracotta's approach (see AOP below) requires significantly more developer effort, and does not support distributing the standard library (à la `fs` for Ignis).

More generally, distributed operating systems, programming languages, and language-based run-times are closer to Andromeda [225]—the platform upon which Ignis was developed—than Ignis itself.

Application Partitioning Automated application partitioning [89, 247] and mobile code offloading [185, 46, 42, 104, 239, 58] introduce (opaquely) the network into the application. Applications are split into a small number of parts, typically two: one runs on the server with nearly unlimited resources, while the other runs on the client with very constrained resources. There is no runtime profiling, and often no performance-oriented component replication [89, 247], as the goal is to hide the network and offer a continuum to the end-user. Wishbone [148] partitions sensornet programs automatically, but only if written in a custom stream-processing language and with predictable input patterns.

Circus [44] and ISIS [23] exploit development-time module structure, but their replication focuses on fault-tolerance instead of scalability. Circus forwards calls to all replicas, whereas ISIS uses a primary-backup scheme. In contrast to Ignis, they operate in a static environment and without runtime introspection, decomposing applications at compile-time. They also assume knowledge of module requirements and deterministic, idempotent modules whose semantics remain locked; in Ignis, such domain-specific information is expressed via recipes and can change at runtime.

Security-oriented compartmentalization [102, 32, 24] decomposes software into multiple isolated components with the goal of improving its security properties—and often at the boundaries of (third-party) modules [144, 224, 108, 213]. However, it does not leverage runtime profiling, and is usually static, targeting privilege reduction rather than performance increase. DeDoS [52] includes profiling for denial-of-service attacks, but requires users to structure their applications in terms of minimum schedulable units (MSUs).

Component Architectures Lambda [60, 85] and microservice [67, 147] architectures build server applications as sets of loosely-coupled components. While in principle small and light, both are vastly more coarse-grained than language-level modules. Whereas a single multi-hundred-package microservice can scale out independently, light-touch distribution can scale out individual components of a *single* microservice. Moreover, communication between services is request-response style and is made explicit to the application. Most importantly, such decomposition is a manual process that requires careful design, including agreeing on the interfaces, prior to development.

Transformations Aspect-oriented programming (AOP) is a programming model that maps program join-points to advice, actions to be taken at these points [99]. For light-touch distribution, these would be "calls at the module boundary" and "wrap with profiling", respectively. Some cross-cutting aspects around the program could be transformed to their distributed versions (*e.g.*, built-in `fs` module). In contrast to AOP, Ignis does not inject dependencies, therefore control flow is not obscured. Moreover, developers do not need to understand different concerns—that is, program structure is not affected and developers do not alter or introduce code.

More generally, AOP is related to metaobjects [100] that enable a program to access to its own internal structure, including rewriting itself as it executes. They are examples of runtime reflection, of which IGNIS makes extensive use to traverse, understand, and rewrite interfaces; but developers using IGNIS do not need to provide their own metaobjects.

There has been a recent emergence of unsound program transformations [173, 163, 197, 198] that attempt to alter the semantics of the original program in principled ways. Light-touch distribution can be seen as introducing programmer-guided, "control-plane" semantic hints at the module boundaries. Using these hints, programmers effectively guide the principles behind how the semantics of a program change.

Chapter 10

Conclusion

The previous chapters show that it is possible to lower the complexity of employing distributing systems using a combination of techniques. Below, I reflect on some of the design decisions and their limitations (§10.1–10.4), outline possible avenues for further research (§10.5), and close with a few higher-level thoughts (§10.6).

10.1 Limitations

The work presented in Chapters 3–6 was motivated by limitations in ANDROMEDA's design. Several other limitations of the system proper remain, three of which are outlined below.

Asynchronous Programming Continuation-passing call style (CPS), when paired with cooperative scheduling, provides a convenient way of expressing which parts of a program should proceed concurrently and which ones should be sequential. However, writing in CPS-only is too verbose—even a trivial `add(1, 2)` function has to be written as `add(1, 2, κ)` where κ is a continuation function. For this reason, the asynchronous programming model offers a convenient middle ground: CPS for operations involving I/O and need to be concurrent, and conventional direct call style (DPS) for everything else.

However, mixing the two call styles in the same environment significantly complicates automation—especially since, in both cases, the type of the function remains the same. Consider `.map(`ϕ`)` and `.read(`κ`)` that (in a simply-typed Lambda calculus [165]) have identical types; however, in the first case ϕ is called synchronously, whereas in the second κ is called asynchronously (and its argument is semantically equivalent to a return value). How can transformations know, at runtime, which one is which?

One solution to this problem would be to label functions with identifiers that signal whether a function is synchronous. Fortunately, EcmaScript functions are themselves objects, which allows attaching arbitrary properties—a feature used extensively in Andromeda. An initial label for built-in functions would need to be provided, but then functions calling these functions could automatically label their return values. By tagging functions, transformation wrappers would become call-style-aware, applying the right transformation by checking argument labels at runtime.

Transactional Semantics A feature that is missing from Andromeda today is the ability to invoke multiple operations (on potentially different nodes) in a way that preserves atomicity and isolation. While single-user semantics and cooperative scheduling ameliorate the problem, they do not provide full remedy.

Designing a configurable, "overlay" transaction system for Andromeda combines several interesting challenges. First, it needs to support the execution of general programming language—*i.e.*, arbitrary functions, which might include side-effects outside the boundaries of the system. Second, overlaying transactions on an existing system may needs to support user-defined constructs—*e.g.*, objects, functions—that are not language built-ins and may be expressed through several different means. Third, providing mixed consistency guarantees—at a high-level, configurable way similar to other services—is an open problem that is just starting to be addressed [79, 207, 134]. These challenges compound with a scalable, fully decentralized transaction coordination.

A starting point for a solution would be a combination of a dedicated transaction management service and automated runtime transformations for augmenting operations with transaction identifiers. A pessimistic protocol (or one augmented with a centralized sequencing mechanism) seems more appropriate because reverting certain side-effects such as console output is impossible.

Fault-Tolerance The distributed versions of built-in services provide a degree of fault-tolerance—*e.g.*, `store` for persistent state. However, the current version of Andromeda does not adequately handle replica failures or network partitions in the general sense, which includes handling calls that were scheduled on failed replicas. Call scheduling could be prefixed with a form of check-pointing so that, upon failure, Andromeda reschedules the dropped call on a different replica (potentially jumping the queue, to compensate for the lost time). Moreover, failure-aware scheduling would adapt scale-out towards the unaffected part of the deployment. Andromeda would also need to protect against partial global changes, where only a subset of nodes receives an update (*e.g.*, write to a global variable), using some form of consensus.

Work on provenance (and especially domain-specific lineage, such as the one pioneered by Spark [248]) can be of significant aid here; however, general-purpose failure tolerance at the level of individual (potentially side-effectful) functions will require further insights. Configuration parameters, pervasive in Andromeda, offer a clear starting point for such an endeavor—for example, work on failure-tolerant light-touch distribution will lead to new recipes for tuning failure-related trade-offs.

As a side-note, some of the Andromeda's simplicity seems to be threatened by fault-tolerance: once one starts worrying about fault-tolerance and error handling, complexity may start crippling up towards the user. However, the split between mechanism and policy and, more specifically, the high level semantic annotations of policies should remain robust to the introduction of mechanisms to deal with these concerns. While domain-specific languages for each one of these services may grow, this will not be significantly different from what they already support. The distributed storage system, `store`, shows some evidence: developers only need to declare their failure expectations from the distributed storage system—`replication`, `consistency`, *etc.* The specifics of error handling depend a bit on chosen the language; unfortunately, JavaScript has no (direct) support for option types (*e.g.*, Haskell's `Maybe Int`) but instead allows throwing exceptions. Exceptions bypass the normal control-flow, which significantly complicates transformations (aside from other problems). The hope is that the ideas presented in this dissertation are applicable to languages with more expressive type systems.

10.2 On BreakApp

BreakApp's techniques are not tied to a particular programming language. The use of JavaScript and its ecosystem was primarily due to three reasons—leveraging Andromeda' infrastructure, a large package ecosystem, and a plethora of known security problems.

Interpreted languages are a particularly good fit for runtime compartmentalization. They expose a single function or function-like operator that takes care of locating a module, interpreting it, and exposing its interface in the caller context. Because all of this happens during runtime, the boundary detection that occurs at the import statement is conveniently unified with runtime compartment construction and code transformations. Moreover, the ability to (re-)bind different functions to the same variable names and interpose on object accesses further simplify things. Thus, BreakApp can be ported straightforwardly in languages such as Lua, Python, and Ruby.

Compiled languages do not enjoy these conveniences. The work done at runtime for JavaScript would need to be split across three phases: compile, link, and runtime. An implementation of BreakApp for compiled code would also face further challenges. First, modules may be linked and loaded statically or dynamically, which complicates the choice of how to divide work between the three phases. Second, type information may not be present at either compile time or run time in languages like C, thus complicating object introspection and marshalling. Finally, source code may not be available for all untrusted modules. Without source code, compiler-driven transformations become infeasible.

Some of these individual challenges have been recently addressed in the literature. For example, C-Strider [186] provides type-aware heap traversal for C programs. Concurrent with our work, researchers are just starting to tackle automated module isolation in compiled languages such as C [213] and Rust [108]. This provides evidence that adapting BreakApp-like techniques for compiled languages is feasible, albeit nontrivial.

BreakApp raises many interesting questions regarding module restarting. One line of exploration is related to runtime policy changes, which might lead to changes in the structure of compartmentalization at runtime—and which can be based on dynamic feedback. While virtually all compartmentalization systems in the literature are geared towards static decomposition, BreakApp's (and Andromeda's) dynamic policies can be altered at runtime. A related line of exploration is patching applications by updating individual modules during runtime. Given isolated modules from BreakApp, compartments can be used to enable incremental, restart-free updating of applications, which can benefit security. For example, one could imagine replacing modules with better versions of modules, generated either through existing (better) modules [197, 198, 199] or through active learning and regeneration [174, 196]. Practical challenges include migrating state of individual modules from the old to the new version and the frequency of updates in applications comprising of hundreds of modules.

10.3 On Iris

Similar to BreakApp, Iris's ideas are not tied to a particular programming language; JavaScript and its ecosystem are used for the same illustrative purposes as BreakApp, in addition to the ability to compare Iris with BreakApp.

Other dynamic languages such as Lua, Ruby, and Python conveniently unify module identification with runtime interposition: a single function or function-like operator locates a module, interprets it, and applies transformations before exposing its interface in the caller context. All these languages offer runtime capabilities such as name (re-)binding, value introspection, and access interposition.

Compiled languages—such as Haskell, OCaml, and Rust—would apply transformations using compile-time metaprogramming facilities, such as templates (*e.g.*, Template Haskell) and macro expansions (*e.g.*, Rust's macro system). Such facilities would alleviate the runtime overhead of transformations, improving program startup times; however, they would not affect the overhead of contract evaluation, as this is still a runtime operation. Static type checking could further aid developers by issuing warnings of incompatible privilege settings. Contrary to BreakApp, however, there is no need for support from the runtime linker because libraries do not execute as separate processes.

The combination of BreakApp and Iris raises the question of mixing module-level compartmentalization systems, targeting different threat models, automatically. For example, a more sophisticated combination of analyses can detect (i) native modules (statically), good candidates for SFI-based containment [235, 245], and (ii) DoS-attacked modules (dynamically), good candidates for Ignis-like scale-out [52, 222]. Unutilized performance budget can then be traded carefully for selected security improvements.

10.4 On Ignis

IGNIS' key enablers—*i.e.*, dynamic languages, module ubiquity, and the programming model (§6.2.4)—are also its key limiting barriers for wider applicability.

IGNIS' ability to significantly automate distribution is made possible due to a confluence of trends in today's development environment: (i) many small modules with clean, high-level interfaces enable high-resolution profiling and transformation, (ii) dynamic programming languages simplify value introspection and type rebinding during runtime, and (iii) cooperative, event-driven concurrency models coupled with continuation-passing style (CPS) ensure the application can decide when to release control, (iii) phased program behavior and stateless protocols (*i.e.*, "everything as a server") unlock high-level, program-wide transformations. Making IGNIS applicable more broadly would require lifting some of these barriers—for example, decomposing an application into components even in the absence of explicit modules and providing bolt-on scalability without any of the features offered by dynamic languages.

Another limitation is related to IGNIS' statistical profiling models. The decision-making infrastructure outlined in §6.4, including the basic statistical profiling models described in §6.4.1 and the controller logic in §6.4.2 showed the potential of light-touch distribution. More sophisticated prediction logic for profiling and coordination (§6.4) is critical for elasticity. It should combine longer call history with the ability to quickly detect sudden changes in workload characteristics. Moreover, it should attempt to capture resource heterogeneity—even the distinction between locally available processors and distributed hosts. The main challenge is combining sub-linear space growth at the module boundaries with prediction latency low enough to be useful in online decision-making.

10.5 Towards the Future

By exploring this area, Andromeda was fortunate to reveal the existence of several interesting problems that have not yet seen adequate solutions.

Distributed Communication Andromeda's development resurfaced a (mostly known) mismatch between what distributed systems need from the communication layer and what protocols in wide deployment provide. Most distributed systems make extensive use of message-oriented multicast communication patterns in which a node sends messages to several other nodes. Yet, for simplicity, most such systems default to all-or-nothing, one-size-fits-all, point-to-point transport primitives such as TCP—because the alternative is to build a protocol from scratch on top of UDP.

TCP is far from ideal, as in most cases it is either "too much" or "too little". In many cases, TCP is just too costly—setting up and tearing down connections, queueing for ordered delivery, *etc.* are all guarantees that trade performance for reliability. Worse even, as TCP provides a fixed, all-or-nothing, one-size-fits-all semantics—reliable, ordered, and error-checked delivery of connection-oriented byte-streams—it is impossible to recoup some of this cost by "disabling" some of these features.

In other cases, TCP's features alone are insufficient—being point-to-point, it is a poor match for systems whose fundamental *raison d'être* is distribution. Point-to-point reliability is not composable without further higher-level mechanisms, such as atomic broadcast [90], to the point that end-to-end consistency guarantees and transactional semantics end up being handled at a layer above TCP. Thus, TCP's features are not only costly—they are also insufficient.

This mismatch is worsened by the emergence of *mixed*-guarantee systems, in which a single service uses several different distribution guarantees configurable at a very fine granularity. Andromeda's canonical example is `store`, but several other proposals provide mixed consistency [79, 207, 134], replication [63, 233], and indexing [228] guarantees at the granularity of individual objects. In such environments, the combined high cost and insufficient semantics of TCP make little-to-no sense.

A promising solution would be to provide distributed applications (and their developers) tunable transport layer abstractions. Using insights similar to the ones used in Ignis, the proposed solution would combine (i) communication channels configurable at fine-granularity with (ii) a high-level domain-specific language for expressing these configurations. At the back-end, a network stack optimizer could take care of analyzing combinations and generating specialized compositions with performance—of which TCP is an instance.

Configuration Inference A key design principle behind Andromeda (and the frameworks built on top of it) has been the distinction between mechanism and policy. Mechanisms offer powerful automation, but are always parametrizable by some kind of configuration—formulas for `store`, policies for BreakApp, contracts for Iris, and recipes for Ignis. This distinction has multiple benefits, both for the designers and the users of these mechanisms—especially when policies are expressed as functions. Designers do not need to anticipate all possible scenarios, and users can specialize these mechanisms to their needs. As users need to provide only a high-level configuration instead of developing the entire mechanism, configurations fundamentally lower the complexity of employing these systems.

A remaining open problem is the inference of these configuration options—the last barrier to automation. Static analysis is difficult in environments with few-to-zero type annotations and modules written in multiple languages, some of which come compiled. Dynamic analysis is equally challenging, as it requires carefully tracing pre-runs that are still not guaranteed to cover all possible (non-deterministic) interleavings.

A combination of tooling (for space exploration) with some form of learning could lead to conservative recipes that improve performance without breaking semantics. This tractability is due to two key observations. The first observation is that, because complexity is now concentrated in only a narrow space, the space of all possible configurations is smaller and thus more amenable to brute-force search. Appropriately restricting the language further into specific classes would further restrict the search space. For example, IRIS contracts that target a subset of its threat models—say only runtime interface subversion—could be inferred using either static or dynamic analysis.

The second observation is that such search does not need to be brute-force if it starts from conservative models that offer few benefits but without breaking certain invariants. For example, automatically inferring IRIS contracts that offer *some* security improvement without breaking functionality. Similarly, one could start from a high-level security principle—*e.g.*, isolation, non-interference—and generate contracts that collectively enforce this property.

Overlay Type Checking Dynamic languages work well for quick prototyping [158], but as a codebase grows large even small changes can lead to bugs. Such problems have been very common in the development of ANDROMEDA, despite considerable investment in testing, linting, and type checking [64]. While static type checking can quickly rule out certain classes of bugs, it is not compatible with the advanced runtime code transformation and evaluation that ANDROMEDA employs.

The experience of developing several frameworks with ANDROMEDA show clear value in a series of *overlay* type systems that can check types just-in-time [171] and are orthogonal to the semantics of the underlying language. Key benefits include type-checking at load time (*e.g.*, right after a transformation or a message) and the checking of different (and diverse) invariants on the same piece of code.

This approach would be an extension over pluggable type-systems [28], in which multiple type-systems co-exist over the same code-base. A key insight here has been to encode types in the comments, a technique that maintains backward-compatibility, enables the co-existence of multiple type-systems, and avoids having type declarations overload values (this is true for complex types). Example overlay type-systems include (i) one checking purity, useful for massively distributed computations, (ii) one checking `RWX` permissions, useful for quickly checking third-party modules, and (iii) one checking union types, useful for encoding more complex invariants and aiding the development of ANDROMEDA.

10.6 Closing Thoughts

Aside from collecting evidence for my thesis and developing several techniques applicable beyond ANDROMEDA, my dissertation hopefully provides a glimpse of a few broader ideas. Three non-obvious ideas are the following.

First, the use of dynamic programming languages with interpreted semantics, precisely due to the dynamic nature of distribution, brings many interesting benefits to distributed systems. Fundamentally, distributed systems are all about runtime reconfiguration—including interface discovery, (re-)generation, and agreement. Even at a lower level, nodes constantly evaluate and re-evaluate code that is in transit across nodes. This decision represents a significant departure from the existing literature in the field of distributed and operating systems, but makes sense in retrospect: a low-level, statically checked, compiled language seems like a poor fit for such an environment.

Second, leveraging (and providing support for leveraging) the software engineering force of an existing package ecosystem numbering over a million modules is incredibly beneficial. This decision was also unusual for research in distributed and operating systems, especially when a key part of the thesis is to redesign a layer that sits lower in the software stack (operating system abstractions).

Third, the use of powerful languages needs to be paired with equally powerful protection mechanisms. Abstraction misuse—accidental or purposeful—is a general problem, regardless of expressive power; however, power seems to have a multiplicative factor in making misuses catastrophic—for example, a simple assignment affecting runtime resolution in the prototype chain of all objects can be irreversible.

Powerful runtime protection mechanisms, such as the ones presented by Iris and BreakApp, are needed to guard against such misuse.

On a separate note, my hope is that Andromeda can (and will) be used as a basis for both teaching and conducting research on distributed systems. Teaching is aided by the overarching goal, which has been to lower the complexity of employing distributed applications—especially for programming "in the small" [53]. I cannot think of a better consumer for this than students about to embark onto distributed-systems explorations, such as undergraduates and graduate students building projects. In fact, having a distributed environment that students can setup in under a second is what many excellent courses in distributed computing lack. Research prototyping is aided by Andromeda' low-complexity and high-performance characteristics: one can quickly prototype ideas whose single-node performance will not suffer. This is a sharp contrast over older distributed environments written in lower-level languages, exactly because they had not received the same engineering care as Andromeda's internals (*e.g.*, V8, LibUV, libraries, ecosystem, *etc.*). A series of frameworks built atop Andromeda and presented in this dissertation—*e.g.*, BreakApp, Ignis—provide ample evidence for this.

As a final note, it seems unfortunate that the distributed (and operating) systems and programming language communities are somewhat separate today. While these communities have considerable shared interests in providing better abstractions, their transfer of ideas is limited.

The programming languages literature has much to offer students of distributed and operating systems. Its community's emphasis on formal specification and understanding enables very useful analyses that can be an ally in any context. Moreover, programming languages are fundamentally a tool for creating new abstractions—the interface between a human and a system—truly unconstrained from the state-of-practice.

Conversely, while the programming-language readership of this dissertation may be more limited, distributed systems have much to offer scholars of programming languages. Perhaps the most important offering is truly a plethora of interesting problems to apply insights and techniques from programming languages. Offering practical, usable abstractions for distributed and operating systems can have widespread impact—even simple ones can make a huge difference in the daily lives of the designers and practitioners of such systems.

Accordingly, this dissertation can be viewed as an attempt to merge these two worlds by revisiting language-based distributed operating system abstractions. The underlying motivating question has been the following: if we were to redesign the abstractions offered by distributed operating systems—possibly by reusing ideas from the past, and given the performance improvements in higher-level languages—how would the solution look like? The constraints of building a practical, usable system within the (admittedly limited, for a project of this scale) scope of a Ph.D. constrained the programming language exploration to augmenting an existing offering rather than designing one from scratch. I am hopeful, however, that the insights and experiences from Andromeda will inform future research in either of these two communities and will aid collaboration.

Appendix A

Tutorial Introduction to Andromeda

This chapter presents a quick introduction to ANDROMEDA, with two key objectives. The first is to show essential elements of the JavaScript language to readers that lack any background in the language—as such, it uses small (but real) programs, without getting too bogged down in details, rules, and exceptions. The second and most important objective is to show the basics of programming with ANDROMEDA. This includes the use of the standard library, how ANDROMEDA interfaces with existing programs, and use of the interactive shell. The goal is not to demonstrate novelty—other chapters achieve this—but to give a sense ANDROMEDA's nature to the reader. Note that ANDROMEDA is at a very early stage of development for practical purposes; still, there are some interesting tasks one can already achieve with it.

The easiest way to try these programs is by typing them into ANDROMEDA's interactive shell. Before going further, note that ANDROMEDA provides interactive documentation using the following shortcut:

```
;?
```

This shortcut will return set of beginner topics, including a fully-interactive version of the following tutorial. Issuing queries on specific names—such as services or methods—can be achieved by typing:

```
1 ;? name
```

Further shortcuts available for interactive use are described later (§A.4).

A.1 First Program: *Hello, Universe!*

Here's a typical first program, adapted for Andromeda:

```
1 andromeda.global.task.exec(() => "Hello,␣Universe!", log)          (p19)
2 //=>{'..b613564047d1..': 'Hello, Universe!',
3 //=> '..b35c5a64519f..': 'Hello, Universe!',
4 //=> '..f74c53b4e063..': 'Hello, Universe!'}
```

Lines 2–4 show a sample output when you type the expression on line 1. Don't worry if your results are somewhat different—either the identifiers on the left or the number of lines. We will now study this program starting from the right end of the program and proceeding to towards its left.

First, the `log` function transforms, beautifies, and prints to the console any value it is provided. This swiss-army knife is useful both for interactive prototyping through the shell and for debugging purposes in programs. To see its simplest use-case, pass `log` as an argument to itself: `log(log)`.

Second, the anonymous function `() => "Hello, Cosmos!"` is syntactic sugar for `function () { return "Hello, Cosmos!" }` that takes no arguments and returns a string. This function and `log` illustrate that functions in Andromeda are first-class citizens: they can be assigned to variables, provided as arguments to calls, and *transmitted across nodes* in the system.

These functions are provided as arguments to `exec`, a method responsible for scheduling tasks to execute on (a set of) nodes. The key point is that `exec` is an asynchronous and non-blocking function; its last argument is a continuation function that will handle the results after the function completes. If it was synchronous and blocking, it would look as `log(exec(t))`. Such a continuation-passing style is preferred, because `exec` can take longer than, say, an addition that executes locally; concurrently, while `exec` is executing remotely, our shell is available for other tasks. This call style is the common case in ANDROMEDA.

Functions such as `exec` are provided by a service—in this case `task`. Services provide the necessary functionality for building distributed applications. Examples of services include `store` for persistent storage and query of objects, `message` for communicating with a remote node, and `nodes` for returning the nodes available in a deployment.

System services are associated with specific node groups, such as `global`. A node group is a collection of nodes, physical or virtual computers that can execute programs and store data. Three node groups come pre-configured in ANDROMEDA: (i) `global`, used here, for addressing all nodes, (ii) `local`, for addressing nodes running on the same host, and (iii) `self`, addressing the current ANDROMEDA node. If you run ANDROMEDA on a single laptop, `global` and `local` refer to the same group.

All ANDROMEDA-related services can be found under the `andromeda` namespace (*i.e.*, the `andromeda.` prefix right before `node.`). Like everything else in ANDROMEDA, the prefix and its semantics are configurable.

A.2 Scaling Up: Standard Library Interfaces

We now know we have a distributed system under our fingertips. Thus, a more interesting example would be to have an *arbitrary* node greet us—*i.e.*, have a remote node execute the "Hello, Universe!" code and send the results back to the current node. This can easily be done by:

```
let h = () => "Hello from " + andromeda.self.config.getNodeId()   (p20)
//=> @base :: function
andromeda.global.tasks.exec(h, {nodes: 'any'}, log)
```

```
//=> 'Hello from ..b613564047d11df47ef7b34f..'
```

This illustrates the use of *optional* configuration arguments, expressed as *objects*. An object is a type of collection that associates string keys with arbitrary values. Values can be booleans, functions, or any other language value—including objects. The structure of objects can change at runtime.

What if we wanted to get a greeting from *all* of the underlying nodes? By changing `task`'s options one can ask all or a subset of nodes to execute something:

```
let h = function (me) {return "Hello␣from␣" + me}                    (p21)
//=> @base :: function
andromeda.global.tasks.exec(h, {args: 'andromeda'}, log)
```

Storage As you might have expected, ANDROMEDA can ship arbitrary objects to remote nodes. Objects represent any state and are used anywhere one would traditionally use a file or a data-base: data, configuration, messages—you name it! The storage system exposes four basic operations, inspired by dataless programming [13]: (i) `put` for insertions, (ii) `get` for retrievals, (iii) `patch` for updates, and (iv) `del` for deletions. For example, consider the following insertion:

```
let galaxy = {name: "Andromeda", magnitude: 3.44}                    (p22)
//=> @base :: object
andromeda.global.store.put(galaxy, 'm31')
```

The system takes care of placing the data in one of the underlying nodes. Although we don't know its precise location (*i.e.*, the node on which it is stored), we know we can always locate it simply by asking ANDROMEDA:

```
andromeda.global.store.get('m31', log)                               (p23)
//=>{ "name": "Andromeda",
//=>  "magnitude": 3.44}
```

Nothing out of the ordinary here: we pass the `log` function so that ANDROMEDA knows what to do when it fetches the object: print it. Later, we will discuss how to tune various state-related properties (*e.g.*, placing, fault-tolerance, consistency *etc.*)

when desired.

Events It would be great if we could be notified about events happening across the system (*e.g.*, failing nodes, newly-joined ones, changes in topology). ANDROMEDA exposes an event channels, where user code can register or generate events.

As an example, consider what happens when nodes detect changes in the underlying topology. When the current node finds out that another node is not available, it will generate a `"peer-down"` notification. Code that depends on all peers being active can then take corrective action. We can instruct the shell to notify us when the underlying topology changes:[1]

```
let f = () => {                                                    (p24)
  andromeda.global.nodes.get(null, (e, d) => {
    log("left␣w/", Object.keys(d).length, "nodes!")
  });
}
//=> @base :: function
andromeda.global.events.on('peer-down', f)
```

Let's jump to the next section to see this one in action!

Fault-Tolerance One reason why people use distributed systems is to tolerate failures: a single computer can be in a state of either working or failed, but a distributed computer can keep working even if a subset of the underlying nodes has stopped working! Suppose that one of our nodes failed unexpectedly. To simulate unexpected node failure:[2]

1. run `andromeda.local.info.get()`

2. note the process ID corresponding to the highest port number, say 3333.

[1] This is somewhat in conflict to ANDROMEDA's philosophy of keeping the user experience identical regardless of changes in the underlying structure. We can think of it in similar terms to a superuser's use of desktop notification when CPU starts overheating.

[2] It's much simpler to shutdown a node from *within* ANDROMEDA. However, we want to simulate an *unexpected* failure, since under normal shutdown a node will take care of some last-minute bookkeeping tasks.

3. in a separate terminal, kill that process using `kill -9 3333`.

Returning to ANDROMEDA, the first thing we should see is the "left w/ ..." message we defined earlier! We can attempt to retrieve the `m31` object:

```
andromeda.global.store.get('m31', log)                              (p25)
//=>{ "name": "Andromeda",
//=>  "magnitude": 3.44}
```

Phew—we can still retrieve our data! Of course, there are cases when such redundant replication is not desirable, or requires several tweaks. This is equally easy, and depends on the originally passed to `store.put`.

Module Management ANDROMEDA is extensible through a simple package manager that allows incorporating code written by other users. Among other things, the package manager takes care of: (i) fetching and bookkeeping third-party code with minimal hassle, (ii) keeping all underlying nodes on the same package version, and (iii) breaking up applications into pluggable components.

Let's import `mdc`, the distributed version of a popular Markdown-to-HTML compiler (the following line requires a network connection):

```
let f = (e, d) => lg('Setup␣complete!')                             (p26)
//=> @base f :: function
andromeda.global.packages.get('@andromeda/mdc', f)
```

Function `f` will notify us when we can start using `mdc`. After setup is complete, we can run program p_4, presented in Chapter 1.

Some packages are already bundled with ANDROMEDA. For instance, `attn.` enables terminal color theming, beautifying source code output from `lg`; `web` is a distributed HTTP server; and `nx` implements a UNIX compatibility layer, allowing users to call UNIX utilities from within ANDROMEDA. In any case, users do not need to go through an installation step and there is no single, centralized package registry.

Wrapping Up This is enough to give a basic idea of what ANDROMEDA feels like. its goal is to allow users to express their intended computation worrying only about the inherent difficulties of their problem and not the difficulties of choosing a distributed model of computation. To shut Andromeda down, do one of the following: type `;q`, press `Ctrl+C`, or call:

```
andromeda.global.nodes.del({halt: true});                    (p27)
//=>  ...Shutting down 3 nodes...
```

A.3 Using Andromeda in Existing Programs

Andromeda is designed to interplay with existing programs; in many cases, these programs are mostly distribution-oblivious—*i.e.*, not written from scratch to exploit distribution. Importing Andromeda into these programs provides access to the library of distributed services.

Importing Andromeda The simplest way to import Andromeda into an existing JavaScript program is the following:

```
1 let andromeda = require("andromeda")                                  (p28)
2 let c = {nodes: 5}
3 let f = () => { andromeda.global.task.exec(() => "Hello!", lg) }
4 andromeda.start(c, f)
```

Program p_{28} first imports Andromeda (1), creates an (optional) configuration (2), and wraps the user program in a function `f` (3). Calling `start` (4) will launch Andromeda by overriding defaults with the values in the configuration; when the startup is complete, it will call the user-defined function `f`.

Alternatively, one might want to launch different functions at different stages of the startup process:

```
let andromeda = require("andromeda")                                  (p29)
andromeda.local.events.on("up", () => log("it's␣up!"))
andromeda.global.events.on("up", () => {
  andromeda.global.task.exec(() => "Hello,␣Universe!", log)
})
```

```
andromeda.start({nodes: 5})
```

The program above will call different functions at different stages of the startup process.

More Complex Programs For a more complicated task, consider calculating a simple proof-of-work function. Proof-of-work functions power modern cryptocurrencies such as bitcoin, and can leverage distribution.

Due to its complexity, we might want to type this program in a file named *e.g.*, `pow.js`. We can then launch it through the UNIX shell, using `andromeda -f "pow.js"`.

```
let andromeda = require("andromeda");                                    (p30)
let pow = (pre, difficulty, start, step) => {
  let zeros = Array(difficulty + 1).join('0');
  let hash = require('crypto').createHash;
  while (true) {
    let s = pre + start.toString(16);
    let digest = hash('sha256').update(s).sha256.digest('hex');
    if (digest.slice(-difficulty) === zeros) {
      cb(null, {nonce: nonce, digest: digest});
    }
    start += step;
  }
};

let dpow = () => {
  andromeda.global.nodes.get(null, (e, nodes) => {
  let max = Object.keys(nodes).length;
  let init = 0;
  let conf = {top: 1, args: () => ['pre', 5, init++, max]};
  andromeda.global.task.exec(pow, conf, out)
});

andromeda.start(null, dpow);
```

This program first imports Andromeda. It then defines two functions: `pow` is responsible for single-node proof-of-work (similar to how it would be written normally); and `dpow` is responsible for calling `pow` on multiple nodes.

A.4 Interactive-Shell Shortcuts

Much of the interaction on Andromeda is achieved through the interactive interpreter (or, shell). The shell exposes a programming interface mostly-identical to the one available for programs. In fact, the `andromeda` instance object available through the shell is *the same* as the one available to programs. Changes to any of the built-in primitives (say, replacing the persistent file system with a non-persistent one, adding a new interface) are reflected immediately in the shell (including the auto-completion methods).

There are two cases in which shell understands code that cannot be part of a valid Andromeda program. The first case is shortcut versions of more verbose function calls to services. This feature is designed to be extensible by both users and packages that add or replace existing shortcuts. Commonly used shortcuts can be seen in Table A.1. The second case is interpreters from programming languages that are different from JavaScript. In these cases, Andromeda forwards parsing and error handling to them. When a user exits the interpreter, they are back into the Andromeda shell.

Programs can be loaded from and saved to the distributed file system. The shell history and the context is also persisted, distributed, and replicated across all the nodes in the global group. Users can start their work in one node and continue their session from any other node. This is another case where consistency is aided by the fact that Andromeda is a single-user system: bits usually travel much faster than users. Users can also process (*e.g.*, run `filter` with regular expressions) their command history and context similar to any other dataset. All of this is exposed via

Table A.1: **Shell Shortcuts.** Examples of (default-defined) shortcuts for interactive use. Imported modules may define their own shortcuts.

Prefix	Calls	To
`;l`	`package.get`	(re-)load module or package and make it available in scope; *e.g.*, `l @andromeda/breakapp` will load the `breakapp` module
`;h`	`repl.hf`	save, load, inspect, or execute shell history; for example, `h /.*global.*/` inspect all commands that match this regexp
`;c`	`repl.cf`	save, load, or print context; for example, `c:s demo` save context as *demo*
`;t`	`utils.typeOf`	output type of expression; for example, `t x` output the type of the value assigned to "x"
`;u`	`unix.exec`	run ¡cmd¿ on Unix and print output; for example, `u:primegen ls -l /` runs `ls -l /` on all nodes of `primegen`
`;?`	`doc.is`	see available interactive shortcuts or documentation for ¡exp¿; *e.g.*, `?  global.store` shows documentation for `store` in `global`.
`;q`	`nodes.del`	quit ANDROMEDA; `q1` quits only the current node
`;v`	`info.get`	display system version and other information

the `shell` service.

Appendix B

The Node.js Module System

As several chapters leverage the module system, whose internals may not be known to the reader. To ensure that the techniques presented in these chapters are appreciated by a wider audience, the following subsections outline how module systems handle modules at runtime—by exemplifying on the internals of the Node.js module system. The chapter starts by sketching out the Node.js runtime (§B.1), then outlining the use of the module system (§B.2), and finally discussing a few technical details of how module loading works (§B.3).

B.1 JavaScript Implementation

Node.js combines a number of components to provide a high-performance JavaScript runtime decoupled from the web browser: (i) the V8 engine, Google's performance-oriented JavaScript implementation used in Chrome; (ii) libuv, a cross-platform library for event-driven, asynchronous I/O; (iii) a number of utility libraries (*e.g.*, a C library for HTTP parsing, zlib for compression, OpenSSL for SSL and TLS); and (iv) a JavaScript standard library (*e.g.*, `module` for loading modules, `url` for URL processing).

V8 compiles and executes JavaScript source code. It provides mechanisms for creating objects, calling functions, allocating and collecting memory, and isolating code within a single runtime.

Libuv implements the I/O subsystem (*e.g.*, file-system, network) with an emphasis on providing abstractions that (i) are identical between different platforms, and (ii) follow an asynchronous, event-driven style. Its core functionality is comprised of the event loop, its worker threads, and callback-based notifications of I/O and other events (*e.g.*, signals). Moreover, it provides a unified, platform independent API by wrapping such platform-specific APIs for system-level functionality (*e.g.*, POSIX timers, sockets, file operations, signals, system events, execution of children processes *etc.*) with platform-independent, asynchronous equivalents. It gathers events from the operating system (*e.g.*, epoll, kqueue, IOCP, event ports) and invokes callbacks defined by the users specifically as handlers for these events.

B.2 Interface of the Module System

Node.js' module system is implemented entirely in JavaScript. It exposes `require`, a global-looking function for importing modules. The following example demonstrates the use of `require`:

```
// -------------- [./main.js] -------------                    (p31)
// importing point
let Point = require("./point.js");
Point.create(1, 1).print(); // => [1, 1]

// -------------- [./point.js] ------------
let Point = function Point (x, y) {
    this.x = x; this.y = y;
};
Point.prototype.print = function print () {
  console.log([this.x, this.y]);
};
```

```
13 module.exports = {
14     create:  function create (x, y) {
15         return new Point(x, y);
16     }
17 };
```

In the example above, the main module (`main.js`) imports the `point.js` module using the `require` statement (line 3) Functionality from the exporting module (`point.js`) that is expected to become available to the importing module (`main.js`) is assigned to a special `module.exports` object (line 13); the rest is module-private functionality. Files and modules are in one-to-one correspondence (each file is treated as a separate module). Method `require` is synchronous (*i.e.*, blocking): it will block execution until the module specified is loaded. The module system is implemented in the `module` built-in module (§B.3), which locates, wraps, compiles, and executes the specified file.

Andromeda hooks into `module` and alters the return value of the `require` call that imports `point.js` (line 3).

B.3 Implementation of the Module System

At a very high level, loading a fresh module with `require("foo");` corresponds to the following five stages:

1. Resolution: identify the file to which the module specified corresponds, and locate it in the filesystem.

2. Loading: depending on the file type, use the corresponding loader (*e.g.*, V8 compiler for `js`, `JSON.parse` for `json` *etc.*).

3. Wrapping: wrap the module so that module-globals get encapsulated and Node.js globals (*e.g.*, `require`) get resolved.

4. Evaluation: evaluate the wrapped module in the current context, so that global names and top-level objects get resolved correctly.

5. Caching: add the module to a handful of module-related cache structures, for purposes of consistency and performance.

Andromeda interposes on all of these steps to facilitate transformations. Wrapping (3) and evaluation (4) are particularly interesting, because they allow Andromeda to interpose at the module boundary during runtime. Before a module's code is evaluated, the Node.js module loader wraps the module so that (i) it keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object; and (ii) it provides some global-looking variables that are actually specific to the module, such as the `module` and `exports` objects that the implementor can use to export values from the module and convenience variables—such as `__filename` and `__dirname` containing the module's absolute filename and directory path, respectively. True globals remaining are (i) the global objects as defined by the EcmaScript standard (*e.g.*, `Object`, `Function`, `Math`); and (ii) Node.js-specific globals (*e.g.*, `console`, `process`, `timer`). These globals require further interposition.

```
// Node.js will wrap a module with a function,                (p32)
// so as to bring certain names into scope
// before compiling/evaluating code.
let wrapped = "function␣(" +
        "exports,␣require,␣module,␣" +
        "__filename,␣__dirname,␣CTX)␣{" +
    "let␣Math␣=␣CTX.Math" +
    "let␣console␣=␣CTX.console" +
    //...[more definitions]
    moduleSource +
  "});"
```

Andromeda hooks into the wrapper function (the last variable in the function definition, `CTX`). This trivial source-to-source transformation re-defines global variables as module-locals and initializes them with ANDROMEDA-augmented values. For example, `console` in the context of the module will be an ANDROMEDA-created object that allows ANDROMEDA to interpose on it. Evaluation of the module passes an additional value to this function, which is the modified context. As a result, any changes to the top-level objects and any global variables are accessible from the within the module.

```
 1 let load = function (mID) {                                        (p33)
 2   if (cache[mID]) {
 3      return cache[mID];
 4   }
 5   let m = {
 6     exports: {},
 7     id: mID,
 8     dir: path.resolve(mID)
 9   };
10   let cm = v8.compile(wrapped);
11   let ctx = andromeda.utils.freshContext(implicits[mID]);
12   cm(m.exports, this.require, m, mID, m.dir, ctx);
13   m.exports = andromeda.utils.wrap(m.exports, explicits[ID]);
14   cache[mID] = m;
15   return m.exports;
16 }
```

The `load` method in the `Module` module combines evaluation (line 10) and caching (line 14) of the wrapped module. After evaluation, invoking the compiled function generates the value that is assigned to `module.exports` from within the module (line 12). ANDROMEDA passes a freshly constructed context at that invocation, modified according to the implicit segment of the PIC corresponding to the module being loaded. Before returning the value of `module.exports`, ANDROMEDA transforms it according to the explicit segment of the PIC corresponding to the module being loaded. Finally, the results of the entire process are placed into the module cache for later use.

Bibliography

[1] *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] CVE-2016-2537., 2016.

[3] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[4] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, 1986.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[6] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[7] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 1–10, New York, NY, USA, 2012. ACM.

[8] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[9] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[10] Amazon.com, Inc. Amazon web services, 2006. Accessed: 2017-11-11.

[11] Apache Software Foundation. hadoop tutorial, 2016.

[12] Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: A single system image of a jvm on a cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 4–, Washington, DC, USA, 1999. IEEE Computer Society.

[13] R. M. Balzer. Dataless programming. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 535–544, New York, NY, USA, 1967. ACM.

[14] Amnon Barak and Oren La'adan. The mosix multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4):361–372, 1998.

[15] Mick Bauer. Paranoid penguin: Apparmor in ubuntu 9. *Linux Journal*, 2009(185):9, 2009.

[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[17] John K. Bennett. The design and implementation of distributed smalltalk. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 318–330, New York, NY, USA, 1987. ACM.

[18] Jon Bentley. Programming pearls: A spelling checker. *Commun. ACM*, 28(5):456–462, May 1985.

[19] Jon Bentley, Don Knuth, and Doug McIlroy. Programming pearls: A literate program. *Commun. ACM*, 29(6):471–483, June 1986.

[20] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[21] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer, 2014.

[22] Martin Biely, Pamela Delgado, Zarko Milosevic, and Andre Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

[23] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[24] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[25] Andrew P Black, Norman C Hutchinson, Eric Jul, and Henry M Levy. The development of the emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 11–1. ACM, 2007.

[26] Oscar Bolmsten. Malicious package: crossenv and other 36 malicious packages, 2017. Accessed: 2019-03-19.

[27] Jonas Bonér and Eugene Kuleshov. Clustering the java virtual machine using aspect-oriented programming. In *AOSD'07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[28] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.

[29] John Brant, Brian Foote, Ralph E. Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 396–417, Berlin, Heidelberg, 1998. Springer-Verlag.

[30] Frederick P. Brooks, Jr. No silver bullet—essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[31] Fraser Brown, Ariana Mirian, Atyansh Jaiswal, Andres Notzli, and Deian Stefan. SPAM: a Secure Package Manager, 2017.

[32] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[33] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, New York, NY, USA, 1997. ACM.

[34] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM.

[35] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[36] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 516–519. IEEE, 2015.

[37] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 609–633, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[39] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

[40] David Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[41] Kristina Chodorow. *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[42] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[43] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 65–82, Berkeley, CA, USA, 2018. USENIX Association.

[44] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 63–78, New York, NY, USA, 1985. ACM.

[45] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, volume 2, 2003.

[46] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[47] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a posix kernel in a high-level language. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 89–105, Berkeley, CA, USA, 2018. USENIX Association.

[48] Ryan Dahl and the Node.js Foundation. Node.js, 2009. Accessed: 2017-06-11.

[49] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[50] Erik DeBill. Module counts, 2009. Accessed: 2018-11-18.

[51] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[52] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Nik Sultana, Bowen Wang, Jingyu Qian, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. A demonstration of the dedos platform for defusing asymmetric ddos attacks in data centers. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 71–73, New York, NY, USA, 2017. ACM.

[53] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. *SIGPLAN Not.*, 10(6):114–121, April 1975.

[54] Udit Dhawan, Albert Kwon, Edin Kadric, Catalin Hritcu, Benjamin C Pierce, Jonathan M Smith, André DeHon, Gregory Malecha, Greg Morrisett, Thomas F Knight, et al. Hardware support for safety interlocks and introspection. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on*, pages 1–8. IEEE, 2012.

[55] Edsger W Dijkstra. On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer, 1982.

[56] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[57] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 400–415, New York, NY, USA, 2016. ACM.

[58] Janick Edinger, Dominik Schäfer, Martin Breitbach, and Christian Becker. Developing distributed computing applications with tasklets. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*, pages 94–96. IEEE, 2017.

[59] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[60] Marius Eriksen. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 5:1–5:7, New York, NY, USA, 2013. ACM.

[61] Robert Escriva. Why consus? what happened to hyperdex?, 2016. Accessed: 2017-06-11.

[62] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[63] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, August 2012.

[64] Facebook. Flow – a static type checker for javascript, 2014.

[65] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[66] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. *ACM SIGPLAN Notices*, 48(1):371–384, 2013.

[67] Martin Fowler and James Lewis. Microservices, 2014. Accessed: 2015-02-17.

[68] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[69] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[70] Nicolas Giard and Wiki.js Contributors. wiki.js, 2018. Accessed: 2018-09-18.

[71] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[72] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[73] Google LLC. Google cloud platform, 2011. Accessed: 2017-11-11.

[74] Michael I Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S Meli, Andrew A Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002.

[75] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 154–169, New York, NY, USA, 1999. ACM.

[76] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM.

[77] Brendan Gregg. *Systems Performance: Enterprise and the Cloud.* Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2013.

[78] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1031. ACM, 2015.

[79] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA, 2016. USENIX Association.

[80] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 147–158, New York, NY, USA, 2008. ACM.

[81] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[82] Mary W Hall, Jennifer M Anderson, Saman P. Amarasinghe, Brian R Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[83] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[84] Michael Hart. *Project Gutenberg.* 1971.

[85] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[86] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[87] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[88] TJ Holowaychuk and the Koa.js Developers. Koajs examples: Blog, 2009. Accessed: 2019-03-01.

[89] Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 187–200, Berkeley, CA, USA, 1999. USENIX Association.

[90] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[91] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 618–625, New York, NY, USA, 1997. ACM.

[92] Makoto Ishihara, Hiroki Honda, and Mitsuhisa Sato. Development and implementation of an interactive parallelization assistance tool for openmp: ipat/omp. *IEICE transactions on information and systems*, 89(2):399–407, 2006.

[93] Wolfgang John, Joacim Halén, Xuejun Cai, Chunyan Fu, Torgny Holmberg, Vladimir Katardjiev, Tomas Mecklin, Mina Sedaghat, Pontus Sköldström, Daniel Turull, Vinay Yadhav, and James Kempf. Making cloud easy: Design considerations and first components of a distributed operating system for cloud. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'18, pages 12–12, Berkeley, CA, USA, 2018. USENIX Association.

[94] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[95] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[96] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[97] Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 28–51, 2015.

[98] Ken Kennedy, Kathryn S Mckinley, and C-W Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, 1991.

[99] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[100] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[101] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[102] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[103] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010.

[104] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[105] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.

[106] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 567–581, 2016.

[107] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[108] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 51–57, New York, NY, USA, 2017. ACM.

[109] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[110] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. 2017.

[111] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the eden system. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 148–159, New York, NY, USA, 1981. ACM.

[112] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, New York, NY, USA, 1975. ACM.

[113] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[114] Jochen Liedtke, Kevin Elphinstone, Sebastian Schonberg, Hermann Hartig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 28–31. IEEE, 1997.

[115] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[116] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 111–122, New York, NY, USA, 1987. ACM.

[117] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[118] SS Jeremy Long. Owasp dependency check. 2015.

[119] Cristina Videira Lopes and Gregor Kiczales. *D: A language framework for distributed programming*. PhD thesis, 1997.

[120] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The snow theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[121] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. The case for the holistic language runtime system. In *Proceedings of the 1st International Workshop on Rack-scale Computing*, 2014.

[122] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 457–471, New York, NY, USA, 2016. ACM.

[123] Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively*. PhD thesis, Carnegie Mellon University, 2016.

[124] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the eighteenth international conference on architectural support for programming languages and operating systems*, pages 461–472. ACM, 2013.

[125] ESLint Maintainers. Postmortem for malicious packages published on july 12th, 2018, 2018. Accessed: 2018-07-12.

[126] Neil McAllister. Twitter survives election after ruby-to-java move, 2012. Accessed: 2019-04-11.

[127] Steve McConnell. *Code complete*. Pearson Education, 2004.

[128] Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, eventually consistent computations with crdts. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[129] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[130] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Networked and Distributed Systems Security*, volume 10 of *NDSS'10*, pages 357–374, 2010.

[131] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[132] James Mickens. Pivot: Fast, synchronous mashup isolation using generator chains. In *2014 IEEE Symposium on Security and Privacy*, pages 261–275. IEEE, 2014.

[133] Microsoft Corporation. Microsoft azure, 2010. Accessed: 2017-11-11.

[134] Matthew Milano and Andrew C. Myers. Mixt: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM.

[135] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, 2008. *Google white paper*, 2009.

[136] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag.

[137] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[138] CVE MITRE. Common vulnerability enumeration. 2006. Accessed: 2018-11-18.

[139] CWE MITRE. Common weakness enumeration. 2006. Accessed: 2018-11-18.

[140] Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan: An embedded dsl framework for distributed embedded systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, pages 2:1–2:9, New York, NY, USA, 2018. ACM.

[141] Bongki Moon, H. v. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, January 2001.

[142] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, and Éric Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 89–100, New York, NY, USA, 2007. ACM.

[143] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[144] Derek G. Murray and Steven Hand. Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes. In *Proceedings of the 1st European Workshop on System Security*, EUROSEC '08, pages 40–46, New York, NY, USA, 2008. ACM.

[145] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[146] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[147] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[148] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. Wishbone: Profile-based partitioning for sensornet applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 395–408, Berkeley, CA, USA, 2009. USENIX Association.

[149] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, March 1984.

[150] Ben Noordhuis. Ctrl-c does not quit repl when in infinte loop, 2016. Accessed: 2018-09-11.

[151] Inc." "npm. npm-shrinkwrap: Lock down dependency versions, 2012.

[152] Inc. npm. Compromised version of eslint-scope published, 2018. Accessed: 2018-07-12.

[153] npm, Inc. Details about the event-stream incident, 2018. Accessed: 2018-12-18.

[154] npm, Inc. Malicious Package: stream-combine, 2019. Accessed: 2019-01-25.

[155] npm, Inc. Malicious Package: stream-combine, 2019. Accessed: 2019-01-25.

[156] Erlend Oftedal et al. Retirejs, 2016.

[157] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 326–336, New York, NY, USA, 1986. ACM.

[158] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, March 1998.

[159] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, 1988.

[160] David A Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. In *In CSRD Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, 1993.

[161] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, May 1972.

[162] Craig Partridge and Robert Hinden. Version 2 of the reliable data protocol (RDP), 4 1990.

[163] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[164] NSA Peter Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track:... USENIX Annual Technical Conference*, page 29. The Association, 2001.

[165] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[166] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*, pages 1–9, 1990.

[167] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[168] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 275–288. USENIX Association, 2014.

[169] Richard F Rashid and George G Robertson. *Accent: A communication oriented network operating system kernel*, volume 15. ACM, 1981.

[170] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[171] Brianna M. Ren and Jeffrey S. Foster. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 462–476, New York, NY, USA, 2016. ACM.

[172] Matt Richardson and Shawn Wallace. *Getting started with raspberry PI.* " O'Reilly Media, Inc.", 2012.

[173] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[174] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. Active learning for inference and regeneration of computer programs that store and retrieve data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 12–28, New York, NY, USA, 2018. ACM.

[175] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[176] Timothy Roscoe. *The structure of a multi-service operating system.* PhD thesis, University of Cambridge, Computer Laboratory, 1995.

[177] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[178] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre

Léonard, et al. Overview of the chorus distributed operating systems. In *Computing Systems*, 1991.

[179] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[180] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[181] Sam Saccone. npm fails to restrict the actions of malicious npm packages, 2016.

[182] Jan Sacha, Henning Schild, Jeff Napper, Noah Evans, and Sape Mullender. Message passing and scheduling in Osprey. 2013.

[183] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[184] Salvatore Sanfilippo and Pieter Noordhuis. Redis, 2009. Accessed: 2016-09-30.

[185] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[186] Karla Saur, Michael Hicks, and Jeffrey S. Foster. C-strider: Type-aware heap traversal for c. *Softw. Pract. Exper.*, 46(6):767–788, June 2016.

[187] Isaac Z. Schlueter et al. Node package manager, 2010. Accessed: 2017-02-17.

[188] Malte Schwarzkopf. *Operating system support for warehouse-scale computing.* PhD thesis, University of Cambridge, Computer Laboratory, 2015.

[189] Malte Schwarzkopf, Matthew P Grosvenor, and Steven Hand. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 9. ACM, 2013.

[190] Node Security. Continuous security monitoring for your node apps, 2016.

[191] Thomas Hunter II (Intrinsic Security). Compromised npm package: event-stream, 2018. Accessed: 2019-03-19.

[192] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[193] Nicolas Seriot. Parsing JSON is a minefield, 2016.

[194] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 15–26, New York, NY, USA, 2005. ACM.

[195] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. *EROS: a fast capability system*, volume 33. ACM, 1999.

[196] Jiasi Shen and Martin C. Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 269–285, New York, NY, USA, 2019. ACM.

[197] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 37–48, New York, NY, USA, 2009. ACM.

[198] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

[199] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 43–54, New York, NY, USA, 2015. ACM.

[200] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages, 2016.

[201] Ayrton Sparling et al. Event-stream, github issue 116: I don't know what to say., 2018. Accessed: 2018-12-18.

[202] Joel Spolsky. Things you should never do, part i, 2000. Accessed: 2017-12-11.

[203] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining javascript with cowl. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 131–146, Broomfield, CO, 2014. USENIX Association.

[204] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[205] Gerald Jay Sussman and Guy L. Steele, Jr. The first report on scheme revisited. *Higher Order Symbol. Comput.*, 11(4):399–404, December 1998.

[206] Cloudius Systems. Osv, 2013.

[207] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 337–350, Denver, CO, USA, 2016. USENIX Association.

[208] Liran Tal. The state of open source security—2019, 2019. Accessed: 2019-03-19.

[209] Gil Tene. wrk2. 2015. Accessed: 2018-11-18.

[210] Jeff Terrace, Stephen R Beard, and Naga Praveen Kumar Katta. Javascript in javascript (js. js): sandboxing third-party scripts. In *Presented as part of the 3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, 2012.

[211] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb 1998.

[212] The gRPC Authors. grpc, 2018. Accessed: 2019-04-16.

[213] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Towards automatic compartmentalization of c programs on capability machines. In *Workshop on Foundations of Computer Security 2017*, FCS'17, pages 1–14, 2017.

[214] Nikolai Philipp Tschacher. Typosquatting in programming language package managers. Bachelor thesis, University of Hamburg, March 2016.

[215] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[216] Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

[217] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. A server-side javascript security architecture for secure integration of third-party libraries. *Security and Communication Networks*, 2019, 2019.

[218] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical report, Ithaca, NY, USA, 1995.

[219] Blesson Varghese and Rajkumar Buyya. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79:849–861, 2018.

[220] Nikos Vasilakis, Pranjal Goel, Henri Maxime Demoulin, and Jonathan M. Smith. The web as a distributed computing platform. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, EdgeSys'18, pages 7–12, New York, NY, USA, 2018. ACM.

[221] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, Henri Maxime Demoulin, Konstantinos Kallas, André DeHon, and Jonathan M. Smith. Andromeda: A distributed userspace. In *Submitted to the 27th ACM Symposium on Operating Systems Principles*, SOSP 2019, New York, NY, USA, 2019. ACM.

[222] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. Ignis: Scaling distribution-oblivious systems with light-touch distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 1010–1026, New York, NY, USA, 2019. ACM.

[223] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Towards fine-grained, automated application compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 43–50, New York, NY, USA, 2017. ACM.

[224] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Networked and Distributed Systems Security*, NDSS'18, 2018.

[225] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. From lone dwarfs to giant superclusters: Rethinking operating system abstractions for the cloud. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[226] Nikos Vasilakis, Yash Palkhiwala, and Jonathan M. Smith. Query-efficient partitions for dynamic data. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 23:1–23:8, New York, NY, USA, 2017. ACM.

[227] Nikos Vasilakis, Cristian-Alexandru Staicu, Ben Karel, André DeHon, Jonathan M. Smith, and Michael Pradel. Iris: Language-based module-level compartmentalization. In *Submitted to the 26th ACM Conference on Computer and Communications Security*, CCS 2019, New York, NY, USA, 2019. ACM.

[228] Dimitrios Vasilas, Marc Shapiro, and Bradley King. A modular design for geo-distributed querying: Work in progress report. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '18, pages 4:1–4:4, New York, NY, USA, 2018. ACM.

[229] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine- to coarse-grained dynamic information flow control and back. *Proc. ACM Program. Lang.*, 3(POPL):76:1–76:31, January 2019.

[230] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[231] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 154–170, Berlin, Heidelberg, 2008. Springer-Verlag.

[232] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[233] VMware, Inc. VMware Xenon: A Decentralized Control Plane, 2016. Accessed: 2017-06-11.

[234] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

[235] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[236] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 17, pages 49–70. ACM, 1983.

[237] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. The risc-v instruction set manual volume ii: Privileged architecture version 1.7. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49*, 2015.

[238] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[239] Xiaojuan Wei, Shangguang Wang, Ao Zhou, Jinliang Xu, Sen Su, Sathish Kumar, and Fangchun Yang. Mvr: An architecture for computation offloading in mobile edge computing. In *Edge Computing (EDGE), 2017 IEEE International Conference on*, pages 232–235. IEEE, 2017.

[240] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, October 2018.

[241] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[242] Ashley G Williams. Changes to npm's unpublish policy, 2016.

[243] Phil Winterbottom and Rob Pike. The design of the inferno virtual machine, 1997.

[244] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proc. VLDB Endow.*, 9(5):420–431, January 2016.

[245] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[246] Serdar Yegulalp. How one yanked javascript package wreaked havoc, 2016.

[247] Cliff Young, Yagati N Lakshman, Tom Szymanski, John Reppy, David Presotto, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse. Protium, an infrastructure for partitioned applications. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 47–52. IEEE, 2001.

[248] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[249] Carlo Zapponi. Githut: A small place to discover languages on github, 2014.

[250] Jamie Zawinski. Resignation and postmortem, 1999. Accessed: 2018-02-12.

[251] John N Zigman, Ramesh Sankaranarayana, et al. djvm-a distributed jvm on a cluster. 2002.

[252] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. *CoRR*, abs/1902.09217, 2019.