COMPOSITIONAL GOSSIP SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lonnie J Princehouse

December 2017

COMPOSITIONAL GOSSIP SYSTEMS

Lonnie J Princehouse, Ph.D.

Cornell University 2017

Gossip protocols have a wide range of applications in distributed systems. They offer robust fault tolerance in exchange for probabilistic guarantees and convergence, and are characterized by elegance and simplicity. This body of research considers the problem of gossip protocol representation and composition; that is, how to use simple gossip protocols as building blocks to form more complex and powerful compound protocols. In doing so, we propose a novel formal representation of gossip, and use it to define the essential properties of gossip systems. We propose composition operators that combine protocols, and show how properties of operands protocols are (or are not) transferred to the resulting compound protocols. Choice among composition operators leads to trade-offs of performance and independence, while preserving semantics. The optimization afforded by what we call "correlated merge" operator enables constructions that would be quite difficult to implement on their own by opportunistically combining gossip messages from many constituent protocols. We discuss which practical syntactic features are helpful for gossip system implementation. A proof-of-concept implementation named MiCA is presented, consisting of Java-language runtime for gossip, a library of gossip primitives, a simulator for rapid development, and visualization and analysis tools that can be used to interpret the results of experiments.

BIOGRAPHICAL SKETCH

Lonnie J Princehouse hails from Seattle, Washington, where he attended school and earned a Bachelor of Science degree in Applied and Computational Mathematical Sciences from the University of Washington. Prior to graduate school at Cornell University, Lonnie worked for the Boeing Company's Mathematics and Computing Technology group, where he prototyped CAD software for programmatic geometry design. He has two young children with his wife Haixin.

Dedicated to my wife Haixin, who is even more important than she knows; to Kallista, age 2.5, and Max, age 1.

ACKNOWLEDGEMENTS

Acknowledgements to my advisor, Ken Birman, with his relentless encouragement and saintly patience. To Nate Foster, whose contributions helped sharpen the ideas in these pages. To Dexter Kozen, the voice of the field of mathematics. To my collaborators and co-authors Robert Soulé, Rakesh Chenchu, and Zhefu Jiang. And to the visitors who proffered helpful advice and sounding-boards: Danny Dolev, Dahlia Malkhi, Idit Keidar, and Márk Jelasity.

This research was funded in part by grants from the National Science Foundation and the Defense Advanced Research Projects Agency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The internet has revolutionized almost every aspect of modern life. Using web browsers on mobile devices and conventional desktop and laptop computers, people use the internet to communicate, search for information, order groceries, play games, check their bank accounts, watch movies, find romance, and conduct business.

The distributed systems that run these services are comprised of anywhere from two or three computers to thousands or even millions. Designing distributed programs to run on these distributed systems differs from writing programs for conventional systems that run on one machine. Distributed systems have a more complex, more nuanced programming model, making them difficult to build and debug. In a conventional computer, if a piece of hardware fails, the machine crashes. It's quite clear that the system has crashed and that your program has stopped running.

In a distributed system, when one part breaks, the rest of the system often continues to run. Other parts of the system may not even realize a failure has occurred. The architects of distributed systems must reason about program behavior in the presence of such faults.

Another difference in distributed systems programming is that communication between processes is unreliable, has much higher latency, and has much lower bandwidth relative to the speed of computation. Nodes in a distributed system may be far apart, perhaps distributed around the world. Without a single, fast, reliable, random-access shared memory, placement of state becomes

much more important. Not all processes have access to all data. A process on one node may need to communicate with other nodes to collect pieces of state needed for its own computation.

To illustrate the complexity of modern, web-scale systems, consider what happens when a customer navigates to Amazon's front page: It takes dozens or hundreds of computers in dozens of services to generate this web page. The customer will receive a page customized to their interests, promoting products that are predicted to be of interest to the customer, displaying special promotions and seasonal products that might be relevant, etc. The page will even look up user profile information, such as how many items are in the shopping cart, or whether a user is logged in. Just this single page view is an immensely complex interaction of distributed systems.

The differences in the programming model presented by distributed systems make it inherently difficult. This is compounded by the fact that the languages we use to write distributed systems—C, Java, C#, Ruby, JavaScript, Python, etc.—all operate within the scope of a single machine. These languages' compilers and interpreters target single machine execution. From the perspective of programs written in these languages, communication between processes in a distributed system is an action that happens as communication between two programs, rather than within a single program. There's a disconnect here in how we implement distributed systems and how we want to reason about them. Many distributed programming languages exist. However, no dominant one-size fits all language has emerged for distributed programming. There is no "C for distributed systems"; it may not even be possible. A common feature of distributed programming languages is that their scope is an entire distributed

system, rather than just a single process or a single machine. Most distributed programming languages focus on a specific niche within distributed systems. For example, database-inspired languages such as BLOOM [1] study how to distribute a query across multiple nodes, and work with highly structured data. Other languages for distributed systems are based on SQL or Map/Reduce (not itself a language, but an important programming model for "big data"). Languages like MACEDON [53] and P2 [45] describe the structure overlay networks.

Like these languages, the body of research described in this dissertation proposes a programming model for a category of distributed systems: gossip protocols.

Gossip protocols excel at certain tasks: Health monitoring, group membership, overlay network construction, metrics aggregation, and eventually consistent replication. These are important components of almost all large, production-ready distributed systems. Although gossip is not always the mechanism of choice for these, it is well-suited. We assert that by giving system designers a gossip framework at a high level of abstraction, we will facilitate the building of modular, reusable, robust, and scalable gossip components. Gossip has a reputation for simplicity, which may explain why such a framework doesn't already exist: It's easy enough to "re-implement the gossip wheel" when called to do so. However, the temptation to do so could result in scalability problems: For example, a naive implementation of uniform gossip over a complete network will encounter scalability problems with membership, as every node must know the complete system membership in order to select a random peer for gossip. As the system grows, and as nodes come and go, it

becomes harder to keep membership up to date. In a production system that may already have thousands or millions of users, this is a *latent liability* that will eventually result in crisis. Such occurrences—crashing into forgotten scalability constraints—happen often behind the scenes in the Cloud Computing industry. One of the best ways to avoid such problems is to use battle-tested, well-understood industry standard components when possible. One goal of this research is to move closer towards reusable gossip components.

A detailed definition of gossip and background information are given in chapter 2.

In this dissertation, we propose a new programming model, "pairwise gossip". With this model, we wanted to capture the fundamental elements of gossip protocols. Pairwise gossip is uniquely tailored to gossip protocols, solving the problem described earlier, that conventional languages only have a one-node view of state when writing distributed systems. Pairwise gossip gives pairs of nodes exchanging gossip access to each others' states, and makes both states visible to the programmer.

To the best of our knowledge, this approach is completely novel. There are few other programming frameworks specifically for gossip, and those that are available [44] focus on providing object-oriented classes and utilities as libraries for conventional languages, or on simulation [49]. No other research into gossip protocols has studied composition of protocols, nor have gossip views been represented as discrete probability distributions instead of sets.

This dissertation is organized as follows:

- Chapter 2 gives an overview gossip protocols, their history and applications.
- Chapter 3 describes the pairwise gossip model and our early research into gossip representation and composition, as excerpted from [51].
- Chapter 4 puts forth the the finished version of our gossip framework, MiCA, as it appeared in ECOOP 2014 [52].
- Chapter 5 goes into greater detail on MiCA's implementation and its capabilities.
- Chapter 6 presents further experimental results on gossip protocol composition.
- Chapter 7 contains conclusions and ideas for future directions.

CHAPTER 2

BACKGROUND

This chapter presents definitions and background information on gossip protocols, as well as a literature survey of gossip algorithms. We use the terms "gossip" and "epidemic" interchangeably to describe this class of algorithms; both are used in the literature. First, we discuss the origins of gossip, followed by discussion of the major categories of gossip algorithms.

2.1 Gossip

Gossip protocols are a family of network protocols roughly characterized by the following scenario: One node selects another node at random from a pool of known peers (its *view*). These two nodes exchange information, and one or both update their internal states accordingly. The first node waits for some interval before repeating the process. Nodes gossip concurrently and independently. It is not uncommon to have an upper bound on the size of data exchanged or the amount of computation that may be performed per unit time; this, combined with the regular frequency of gossip exchanges, results in steady, predictable network overhead that scales well as the network grows, and is well-behaved in the presence of network congestion. In general, gossip is well-suited to applications for which probabilistic guarantees are adequate and which do not call for immediate reactions to events.

The canonical gossip uses are:

Anti-entropy. One of the canonical gossip uses; nodes keep versioned sets of

objects (e.g., database rows), and gossip to discover newer versions elsewhere in the system. Anti-entropy is a mechanism for eventual consistency.

Rumor mongering. A probabilistic broadcast mechanism.

Failure detection. Gossiping about who we've talked with recently can help the system notice who we haven't heard from.

Aggregation. Similar to rumor-mongering, but computing a function of data held at different nodes. Examples are: Approximating statistics about node capacity throughout the network[27][29]; computing user-defined aggregate queries[62].

Overlay maintenance. Systems such as distributed hash tables often build dynamic routing overlays that must be constantly updated as nodes enter and leave the system. Gossip can be used to update views in such a manner that the graph of node connectivity either functions as the desired overlay, or can be used to monitor some underlying overlay for purposes of adaptation and repair.

Peer sampling. Randomness of peer selection is important for many gossip protocols. For large networks, however, it is impractical for each node to store the address of all other nodes in the system. Peer sampling algorithms allow gossip nodes to sample values maintained by their peers in a way that approximates true random peer selection given only a fixed-size local view. [47]

```
// Active thread, running on node a with state σ_a
do forever:
  wait t seconds
  b ← selectPeer(σ_a)
  send σ_a to b
  receive σ_b from b
  σ_a ← update_a(σ_a, σ_b)
```

```
// Passive thread, running on node b with state σ_b
do forever:
  a ← awaitConnection
  receive σ_a from a
  send σ_b to a
  σ_b ← update_b(σ_a, σ_b)
```

Figure 1: Active and passive threads

2.2 Literature Survey

Epidemiology as model for information dissemination was proposed by Goffmann and Newill in 1964 [25]. The authors advanced the notion that the spread of ideas and information are analogous to how infections propagate through a population. They laid out the mathematics for what we call uniform gossip—gossip over a complete graph with peers chosen at random—and solved for the rate of convergence of total infection. Kendall and Daley[15] observed that the social phenomenon of a rumor spreading in a social network could also be modeled as an epidemic, and compared several models for the spread of rumors through populations.

The general pattern of a gossip—choose a peer, exchange information, wait, repeat—appeared in distributed systems literature well before it was labeled as "gossip" or as an "epidemic". For example, Usenet's Network News Transfer Protocol (NNTP) [36] uses a network of peered news servers to distribute new messages through individual peer-to-peer interaction. This is cited as an early example of gossip, although RFC 977 did not use that term.

Another early example is proposed by Fischer and Michael [23]. This paper considers the problem of replicating a dictionary across a distributed system, and propose a weakened consistency model that allows high availability and improved partition tolerance. In their discussion of open problems, they propose a communication mechanism that sounds very much like gossip, although it stops short of randomized peer selection:

> We have not yet addressed the problem of finding a good strategy for the nodes to use in deciding when and how to communicate. If each message can be received by only a single process, then various strategies can be imagined. At one extreme, a message transmission from i to j could be attempted periodically for all pairs $i, j, i \neq j$, resulting in a total of $O(N^2)$ messages to propagate information between all pairs of nodes. On the other hand, given a spanning tree in the network and a root, one can propagate information from every node to every other node using only O(N) messages...

This strategy would later by implemented and refined by Ladin et al. [41]. Fischer and Michael consider communication over both a complete graph and a spanning tree topology.

In 1987, Demers et al. introduced *anti-entropy* [19] as a means to keep distributed database replicas consistent. Anti-entropy is a form of gossip wherein database peers compare local replicas data sets of versioned objects. If one peer finds it has a lower versioned object than another, it gets the newer version. Updates to objects propagate through the system as an epidemic, but the point-to-point state exchange is triggered by the detection of stale information rather

than through broadcast of explicit update messages as in [23]. This introduced a weakened consistency model that enabled availability at scale.

Anti-entropy would go on to be influential in the peer-to-peer era, and later in the age of cloud computing and NoSQL database systems. In the late 1990's and early 2000's, the promise of cheap computing devices coupled with pervasive internet connectivity led researchers to envision a world of peer-to-peer computing, where large internet services could be run in a decentralized fashion on nodes scattered around the globe. The Bayou [58] storage system is a shared, distributed data store, intended for a network of mobile devices with intermittent or unreliable internet connectivity. Bayou allows clients to read and write to any replica and uses anti-entropy gossip to make the system *eventually consistent*, a name given to the weakened consistency that arises from anti-entropy's gossip-based update propagation. Eventual consistency in a multi-master system can give rise to write conflicts, which Bayou addressed by allowing applications to specify their own conflict resolution handlers.

Bimodal multicast [7, 6, 5] employs anti-entropy for probabilistic multicast. The authors make the point that, while gossip's probabilistic guarantees are weaker than those offered by other reliable multicast methods, they are more predictable than best-effort systems and offer greater stability. With gossip, although worst-case behavior is possible, the systems are robust in practice and have well-understood bounds. Bimodal multicast's anti-entropy prioritizes recent messages over old messages, giving the system a weak real-time guarantee. In addition to anti-entropy for multicast, the SpinGlass implementation [5] of bimodal multicast also relies on gossip for eventually consistent, self-stabilizing group membership, drawing from [26, 64].

Gossip-based group membership is used widely, typically in conjunction with gossip-based failure detection. Uber's Ringpop [46] service, which provides membership for geospatial coordination within Uber's distributed applications, uses the SWIM [16] gossip protocol for group membership and for failure detection. HashiCorp's Serf, a membership and event delivery platform, is itself based on an extension of SWIM called Lifeguard [13]. Lifeguard extends SWIM with measures to reduce false positive failure detection caused by servers that are temporarily unable to respond, or respond late, due to high CPU or memory utilization. Amazon's S3 uses gossip for group membership, as revealed in the post-mortem of a 2008 outage caused by corrupted gossip messages [57].

In 2007, Amazon's influential Dynamo [18] paper inspired a new generation of eventually consistent database systems. Dynamo utilizes gossip in several ways: anti-entropy, for replication; gossip-based group membership; and failure detection, in the style of [28]. Dynamo's goal was to achieve scalability beyond the capabilities of contemporary ACID-compliant RDBMS, prioritizing availability over consistency when necessary. Although none of Dynamo's gossip techniques were novel, the use of eventual consistency for the Amazon.com shopping cart—an important and highly visible service—fostered a surge in the popularity [67] of eventual consistency and spurred the growth of the NoSQL database market. Dynamo's example would go on to inspire anti-entropy-backed eventual consistency in Facebook's Cassandra [42] (now Apache Cassandra), Riak [40], and more. The popularity of eventual consistency has faded somewhat since, possibly due to the additional complexity it forces on designers. Many current NoSQL databases offer both eventual and strongly consistent operations, but gossip's role is now firmly established.

Another prominent gossip use case is aggregation. Gossip-based aggregation typically runs a distributed algorithm to compute a query on distributed data: Min, Max, Sum, $count$, avg, quantiles, sampling[33, 60], distribution estimation [29], among many others. Research into gossip-based aggregation addresses the challenges of dynamic networks, where nodes leaving or arriving may alter the values of computed aggregates, necessitating recomputation [30]; trading accuracy for time and space efficiency [2]; and analysis of convergence [37, 9]. Aggregation over various topologies has been considered: Uniform gossip on a complete graph [37], arbitrary unstructured graphs [2], spanning trees, and expander graphs [38]. Astrolabe [62] computes aggregate queries on a hierarchy of nodes by gossiping both within and between levels of the hierarchy.

Use cases for gossip-based aggregation include massive peer-to-peer unstructured networks [2] and sensor networks [43, 37]. In both settings, gossip's robustness to dynamic churn and unreliable networks gives it an advantage over other distributed system aggregation techniques, such as spanning-tree-based aggregation [50]. Gossip-based aggregation is also well-suited for applications that need a query result to be present at many nodes, instead of just a spanning tree root, such as Sliver [27] which uses an aggregate at each node to dynamically determine the node's placement relative to other nodes. Spanning tree aggregation is not mutually exclusive with gossip, as demonstrated by Astrolabe, which successfully combines the two.

Gossip is used as a foundational layer for network overlay construction (which can be viewed as a kind of aggregation). For example, for spanning tree overlays for aggregation as discussed above, or for the ring overlays that under-

pin some distributed hash tables [55, 18]. This is a natural extension of gossip-based health monitors, which must be aware of when nodes join and leave the system. Overlay construction is generalized by T-Man [32], which proposes an elegantly simple ranking mechanism that can construct overlays for a class of network topologies. T-Man's algorithm can be viewed as an aggregation, where the primitive data to be aggregated are profiles of nodes participating in the network.

CHAPTER 3

CODE-PARTITIONING GOSSIP*

Code-Partitioning Gossip (CPG) is a novel technique to facilitate implementation and analysis of gossip protocols. A gossip exchange is a pair-wise transaction between two nodes; a gossip system executes an endless sequence of exchanges between nodes chosen by a randomized procedure. Using CPG, the effects of a gossip exchange are succinctly defined by a single function that atomically updates a pair of node states based on their previous values. This function is automatically partitioned via program slicing into executable code for the roles of gossip-initiator and gossip-recipient, and networking code is added automatically. CPG may have concrete benefits for protocol analysis and authoring composite gossip protocols.

In defining code-partitioning gossip, we consider two different perspectives on gossip—that of the programmer, and that of the theorist.

The programmer formulates gossip with implementation in mind. A gossip system uses two threads per node; one active, one passive[31]. The active thread periodically initiates a gossip exchange with a randomly selected peer, and the passive thread awaits and reacts to connections. In this paper, we use the terminology "sender" to refer to the active-thread node that initiates a gossip exchange, and "receiver" to indicate the passive-thread recipient, even though both nodes send and receive data. For brevity, all examples name the sender a and the receiver b. Figure 1 contains pseudo-code for the sender and receiver

*Excerpted from Princehouse, L. and Birman, K. Code-Partitioning Gossip. Fifth Programming Languages and Operating Systems Workshop (PLOS), 2009. Operating Systems Review 2010, Vol. 43

event loops. Note that during an exchange, each node sends its state to the other, and then computes a new state based on the pair of states. In gossip terminology, this is a *push-pull* protocol, and it encompasses the more specific sets of *push* protocols (in which only the sender pushes its state to the receiver) and *pull* protocols, in which state moves only from receiver to sender.

In contrast, the theorist frames gossip in more holistic terms, asking, *"How does the gossip exchange affect the state of the system?"*. Instead of two update functions $\sigma_a \leftarrow \mathsf{update}_a(\sigma_a, \sigma_b)$ and $\sigma_b \leftarrow \mathsf{update}_b(\sigma_a, \sigma_b)$ separated by networking code, the theorist ignores the network and poses the exchange as single *unified update function*, $(\sigma_a, \sigma_b) \leftarrow update(\sigma_a, \sigma_b)$. Using this function, the theorist proves interesting properties about her gossip algorithm. For example, the theorist might prove that update is monotonic with respect to some property of system state, and use this fact in an inductive proof to show that an invariant always holds.

There are, of course, simplifying assumptions. The theorist has presented this gossip exchange as an atomic transaction on system state. In reality, networks are unreliable and nodes sometimes fail. The Two Generals tell us that a node fundamentally has no way of knowing if its counterpart has successfully completed the exchange; the best our nodes can do is to atomically commit changes to their own state, such that the failure of one node halfway through a gossip exchange does not leave the other node with an inconsistent state. Accordingly, the proofs must be expanded to account for the possibility of failure, which may cause a gossip exchange to unpredictably update one, both, or none of the states of its participants.

In this paper, we present Code-Partitioning Gossip (CPG), a Programming

Languages-inspired technique for the implementation of gossip protocols. CPG strives to reach a happy medium between the programmer and the theorist. Using CPG, the programmer writes a unified update function that operates on pairs of states. This function is automatically partitioned into update_a and update_b, and code for the active and passive threads is synthesized. Networking code is inserted automatically, allowing systems to be easily re-tooled for different network models and transports.

Code-Partitioning Gossip offers several possibilities. First, it allows the programmer to create composite gossip protocols using the familiar mechanisms of functional composition and object oriented programming. Second, it affords the theorist the opportunity to bring program analysis tools to bear on the update function. Third, it lets the programmer separate implementation details from protocol semantics.

This paper is organized as follows: Section 2 elaborates on the design of CPG. Section 3 further describes the design of Code-Partitioning Gossip and our prototype implementation. Section 4 discusses existing work as it relates to gossip protocols and code-partitioning. Finally, Section 5 ruminates on the implications and future directions of CPG.

Design

Let the set of all nodes be N. For the purposes of Code-Partitioning Gossip, we define a gossip protocol as the triplet,

State type A datatype. The set of all states is Σ.

selectPeer : $\Sigma \to N$**.** Chooses a peer to gossip with based on a node's state.

Allowed to be non-deterministic.

update : $\Sigma^2 \to \Sigma^2$**.** Deterministic exchange update function. Given a pair of node states, compute an updated pair.

Given such a protocol definition, the CPG runtime automatically partitions update into update_a and update_b. Before explaining exactly how this is done, we present as a simple example the MAXVALUE protocol. In MAXVALUE, each node stores an integer value. During a gossip exchange, both nodes adopt the greater of their two values. MAXVALUE runs on a fixed communication graph. All nodes eventually converge to the maximum value in the system with high probability.

```
1  public class Maxvalue {
2    private Address address;
3    public int value;
4
5    public Maxvalue(Address address, int value,
6        Set<Address> view) {
7      this.address = address;
8      this.value = value;
9      this.view = view;
10   }
11
12   @GossipSelectPeerUniform
13   private Set<Address> view;
14
15   @GossipExchangeUpdate
16   public void update(Maxvalue b) {
17     value = b.value = max(value, b.value);
18   }
19 }
```

Figure 1: MAXVALUE protocol

Figure 1 contains the actual Java code for MAXVALUE as implemented in our system. MAXVALUE is *mostly ordinary Java code*: The gossip protocol is written as a class, and instances of this class represent individual nodes. The only unusual features are the annotations GossipSelectPeerUniform and Gos-

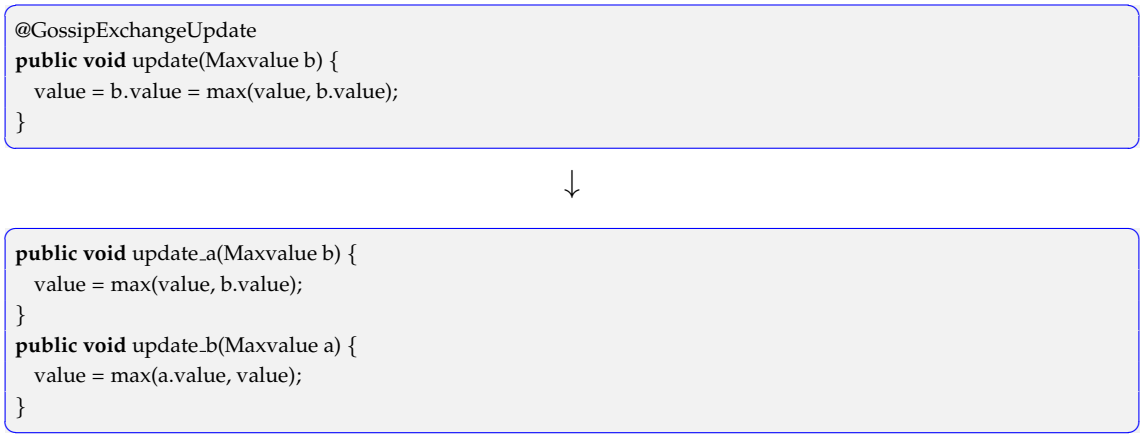

Figure 2: MAXVALUE automatic partition

sipExchangeUpdate on lines 12 and 15. These annotations tag elements of the program for special treatment by our runtime system. GossipSelectPeerUniform tells the runtime that the member variable *view* is to be used for uniform random peer selection, and GossipExchangeUpdate marks the function update for automatic partitioning.

Figure 1 contains the actual Java code for MAXVALUE as implemented in our system. MAXVALUE is *mostly ordinary Java code*: The gossip protocol is written as a class, and instances of this class represent individual nodes. The only unusual features are the annotations GossipSelectPeerUniform and GossipExchangeUpdate on lines 12 and 15. These annotations tag elements of the program for special treatment by our runtime system. GossipSelectPeerUniform tells the runtime that the member variable *view* is to be used for uniform random peer selection, and GossipExchangeUpdate marks the function update for automatic partitioning.

We employ static program slicing[70] to accomplish this partition. Briefly, program slicing attempts to solve the following problem: Given a program and a target value (as it appears at some program point), return a subgraph of the

program's control flow graph consisting only of statements that contribute to the computation of the target value. This CFG subgraph is called a "slice", and is itself an executable program. When executed, the slice computes the target value exactly as the original program would have. CPG's program slicing is necessarily conservative, omitting statements only if they are proven irrelevant.

Code-Partitioning Gossip generates two slices: one that computes updated state σ'_a for the sender, and one that computes σ'_b for the receiver. These slices are effectively the update_a and update_b functions seen earlier. Figure 2 shows an example of the how MAXVALUE's update function could be partitioned. Code-Partitioning Gossip expects the update function to be deterministic and to halt; the onus to enforce these conditions is on the programmer.

This particular brand of program slicing—splitting a function between two nodes—raises some interesting questions. The nodes cooperate initially to share their states, but program slicing may reveal that only pieces of the other node's state are needed to compute update_a or update_b. For example, view and address are part of MAXVALUE's state, but are not needed for the gossip update. Further, which pieces of state are needed may only be known at runtime. Rather than shipping the entire state in a single transaction, our synthesis of the update_a and update_b functions could provide the opportunity to send state between nodes on demand. Such a system might make the additional decisions of whether to send any state speculatively and whether to try to minimize bandwidth used or total number of messages sent between nodes. However, these questions are not our focus.

Implementation

In our prototype implementation, CPG protocols are written in Java, with custom annotations used to designate a protocol's peer selection and exchange update behavior. We considered creating a domain-specific language for gossip, but ultimately decided against it on the grounds that Java provides sufficient extensibility to accomplish our goals, and many programmers are already familiar with Java. When a Java class is loaded, the Code-Partitioning Gossip runtime uses reflection to search for members tagged with one of several special gossip annotations. The annotation GossipExchangeUpdate on a method causes the method to be partitioned and two new methods, representing the two slices of the update method, are dynamically added to the class. These functions are called to perform gossip exchanges by active and passive gossip threads implemented by the Code-Partitioning Gossip runtime.

Our prototype implementation of CPG has two phases of analysis, both operating on the Java bytecode of a protocol class. Running this analysis on bytecode rather than Java source was a pragmatic decision—we felt it would be easier to write a prototype using existing bytecode manipulation tools—but it has some additional benefits, such as the potential to write CPG gossip protocols in any language that targets the JVM (e.g., Scala). For CPG, first the update method is is sliced into active and passive methods that update the states of two local node instances. Second, network code is injected to retrieve state from the remote node when it is needed. Several annotations are provided for peer selection. GossipSelectPeerUniform selects a peer uniformly at random from a set of addresses of other peers. GossipSelectPeerWeighted lets the developer specify probability mass weights (for protocols that require non-uniform random selec-

tion, e.g., spatial gossip[39]). GossipSelectMethod designates a method to call directly for peer selection.

In order to use a gossip protocol, the developer creates an instance of its class and instructs the Code-Partitioning Gossip runtime to begin gossiping. While gossip proceeds quietly in the background, the protocol instance can be used like any other Java object by the encompassing Java program. As a practical matter, nodes in our prototype wait to finish one gossip exchange before engaging in another. This mandates a system-imposed timeout for failed nodes (or else a node would cease to gossip when it fails to receive a response).

Example

We now present a more sophisticated example. Sliver[27] is a slicing protocol. In a network where nodes have varying capacities of some metric, Sliver assigns each node to one of k groups of approximately equal total capacity. Nodes provide a getSlice method that returns an estimate of their current slice; this is computed as follows:

All nodes keep a set of *(node identifier, capacity, timestamp)* triples. During a gossip exchange, the sender transmits its capacity to the receiver, and the receiver records *(sender, capacity, timestamp)*. To compute getSlice, a Sliver node first purges any stale triples (either because they have been superseded by new information about a node, or because their timestamps are too old). It then computes the fraction of known nodes with lesser or equal capacity to itself. The current slice is obtained by multiplying this fraction by the total number of slices k and rounding to the nearest integer.

Figure 3 shows Sliver as implemented under Code-Partitioning Gossip.

Related Work

We are aware of one API framework, GossipKit[44], that uses standard object-oriented programming methodology to furnish the developers of gossip protocols with reusable, modular gossip abstractions. Such a framework serves two purposes: It provides plug-and-play gossip protocols that can be used by developers (e.g., peer sampling), and it facilitates development of gossip protocols by providing a skeletal gossip runtime that can be extended via inheritance. We assert that CPG has an advantage over such a toolkit in that CPG lets the programmer describe a protocol at a higher level of abstraction, namely the pair-wise updates of system state. However, the toolkit approach may be easier to debug since the bytecode run by CPG has been transformed by program slicing.

A second class of related work seeks to generalize specific kinds of gossip protocols. Two such systems are T-Man[32] and Astrolabe[62]. T-Man is a configurable gossip system for the creation and maintenance of structured overlays. T-Man imposes a user-defined sort order $\succ$ over all nodes in the system. Nodes maintain views of fixed size, sorted in this order. When a T-Man node learns of another node x such that $x \succ y$ for some $y \in$ view, x replaces y. The views of each node define the overlay graph. By supplying different sorting functions, T-Man can form a truly surprising variety of overlay topologies.

Astrolabe organizes its nodes into a tree. The tree's inner nodes may contain user-defined aggregation functions that compute some aggregate of the data

stored in the node's children. Users of Astrolabe can then execute database-like queries to evaluate these aggregates. T-Man and Astrolabe do not have the same goals as CPG, so a direct comparison is not possible. However, both T-Man and Astrolabe would make excellent benchmarks if implemented using CPG. Astrolabe, in particular, has a recursive structure that lends itself well to CPG. Implementing these systems using CPG is left for future work.

MACE[53] is a domain-specific language for authoring overlay systems, intended for writing overlays such as Chord[55], Pastry[54], etc. MACE compiles into C++, and claims to save a great deal of programmer effort and attain reasonable performance. While it is not gossip-specific, we see no reason that MACE could not be used to implement gossip systems.

Regarding program slicing and automatic partitioning, Jif/Split[73] and Swift[11] use such a technique to automatically partition programs to run between client and server according to information flow security labels on variables. If anything, Code-Partitioning Gossip is much less ambitious in the scope of its partitioning scheme: Jif/Split and Swift must decide where and when to move data based on a set of hard security constraints, whereas CPG has the locations of variables as a given from the start. CPG differs from these systems in that its pair-wise program slicing implicitly defines the behavior for an n-node system.

3.1 A Pairwise Abstraction for Round-Based Protocols LADA 2012

Prior work in this area has produced diverse solutions. The DryadLINQ [72] language expresses distributed computations using SQL-like queries. BLOOM [1] also follows a data-centric approach, but assumes an unordered programming model by default. MACEDON [53] provides constructs for describing overlay networks. P2 [45] uses declarative syntax based on Datalog to express network protocols. Bast [24] provides object-oriented, extensible, and composable protocols. Lastly, Jini [68] offers a framework for extensible network services.

Two categories of related work differ in the kind of abstraction given to the programmer: Languages based on a *single-node perspective*, including conventional languages like C and Java, only provide programmers with access to a local slice of the global system state. Hence, access to state on remote nodes must be obtained using explicit communication. Writing distributed systems in such languages is difficult, as the language and compiler are unaware that the program is part of a larger system. Languages based on a *whole-system perspective* provide programmers with a broader view of system state. This allows implementations to more closely resemble design, and makes reasoning about the theoretical behavior of distributed systems simpler. However, these languages must often make trade-offs between simplicity and power. Many whole-system languages focus on a particular class of distributed system.

Our system, Code Partitioning Gossip (CPG), provides an abstraction that lies between the single-node and whole-system perspectives. It is designed specifically for synchronous, fault-tolerant systems—a class that includes many

gossip and self-stabilization protocols. These are especially relevant to current computing trends. Because of their passive, round-based nature, they tend to be well-behaved and make predictable use of the network. As such, they are "good neighbors" in massive multi-tenant data centers, such as those that drive Amazon's EC2. Many cloud computing services have relaxed consistency requirements in favor of availability, and this also plays to the strengths of round-based protocols.

Our goal with CPG is to design abstractions for describing these protocols that make it easy to develop richer protocols via composition and code re-use. The fundamental unit seen by programmers in CPG is a *pair* of nodes. A protocol in CPG is defined using a `view` function, which identifies pairs of nodes to communicate in each round, and an `update` function, which takes the states of the selected nodes as input and produces their updated states after communication as output. The global state of the system evolves by the repeated application of the pairwise update function to selected states. If Σ denotes the set of possible node states, the types of these functions can be written as follows:

$$\texttt{view} \in \Sigma^2 \to \text{Address}$$
$$\texttt{update} \in \Sigma^2 \to \Sigma^2$$

Execution proceeds in rounds. In each round, every node uses the `view` function to pick a partner to gossip with, and then executes `update` with the selected node. We do not assume the existence of a central clock; rounds are approximate and each node uses its own clock. We also assume that network communication may time out and that nodes may fail or malfunction at any time. The protocol specified by the programmer must be sufficiently fault tolerant, as many gossip and self-stabilizing protocols are.

CPG provides two operators for composing protocols, *merge* and embed$(., T)$hese operators allow multiple protocols to be written separately and then combined, in the same way that classes in object oriented languages can be composed. In fact, our prototype implementation uses Java as its base, making it literally possible for one protocol to inherit another, or for one protocol to use instances of others. The Java type system can be used to express properties of protocols. For example, protocols implementing the *Overlay* interface are expected to build and maintain a network overlay, and the *TreeOverlay* interface is an extension with the additional constraint of a spanning tree. Protocols that run on an overlay can reference the *Overlay* interface, allowing different overlay implementations to be easily substituted. Additionally, it is possible to generalize transformations on protocols. For example, [3] outlines a "pipelining" procedure, by which an arbitrary self-stabilizing protocol can be imbued with Byzantine fault tolerance. CPG's abstractions make it possible to implement pipelining as a function on protocols.

Our CPG prototype is implemented a Java bytecode post-processor. Protocols are written as Java classes, with special annotations used to denote the `update` and `view` functions. The post processor splits `update` into two functions, one for each node in a communicating pair. Networking code is added automatically. To illustrate, Figure 1 presents the Java definitions of the `view` and `update` functions for a simple gossip protocol that implements leader election for an overlay network. The code on the left side of the figure presents the gossip protocol itself; the code on the right side gives some supporting library definitions. The `view` function chooses randomly from the collection of nodes in the overlay. The `update` function compares the addresses of the leaders on the two nodes being updated, and updates the node whose leader has the larger ad-

dress. When the protocol eventually stabilizes, the overlay node with the least address is elected leader.

Although CPG can express a diversity of gossip, peer-to-peer and self-stabilizing protocols, the language model is inherently probabilistic. For example, the leader election protocol exhibited above converges in logarithmic time to a single leader, but lacks the stronger atomicity semantics of consensus-based leader election solutions. A particularly interesting open problem is this: can CPG be used to simulate the execution of that sort of consensus-based solution, or is there a true separation between the class of programs CPG can express, and the class that includes consensus? We hope to explore this in future work.

In our experience, CPG's pairwise abstraction is not only sufficient to represent a broad range of real-world protocols, but also intuitive for the programmer. The pairwise abstraction helps bridge the gap between implementation and design, and offers benefits through code re-use and composition.

```
public class Sliver {
  private class Rumor {
    public Long timestamp;
    public Double capacity;
    Rumor(Double capacity) {
      timestamp = new Date().getTime();
      this.capacity = capacity;
    }
  }
  private Address address; // This node's address
  private int k; // Number of slices
  private Double capacity;
  private long timeout;
  // Everything we know about other nodes' capacities
  private HashMap<Address, Rumor> rumors;
  public Sliver(int k, Double capacity,
       long timeout, Set<Address> view,
       Address address) {
    this.rumors = new HashMap<Address,Rumor>();
    this.address = address;
    this.k = k;
    this.capacity = capacity;
    this.view = view;
    this.timeout = timeout;
  }
  @GossipSelectPeerUniform
  public Set<Address> view; // known peers
  @GossipExchangeUpdate
  public void update(Sliver b) {
    // Tell the other node about this node's capacity
    b.rumors.put(address, new Rumor(capacity));
  }
  // Called by user to determine this node's slice
  public long getSlice() {
    purgeExpiredRumors();
    long m = rumors.size();
    long B = 0;  // number of known peers with capacity not greater than ours
    for(Rumor i : rumors.values())
      if(i.capacity <= capacity)
  B++;
    return StrictMath.round(k *
        (double) B / (double) m);
  }
  private void purgeExpiredRumors() {
    long now = new Date().getTime(); // Delete expired rumors
    for(HashMap.Entry<Address,Rumor> e :
      rumors.entrySet() ) {
  Address peer = e.getKey();
  if(now - e.getValue().timestamp < timeout)
      rumors.remove(peer);
}}}
```

Figure 3: Sliver implementation

```
public class MinAddressLeader implements Protocol {
  private Address leader;
  public MinAddressLeader(Overlay overlay) {
    Selector s = new RandomSelector(overlay.getView());
    setSelector(s);
  }
  public Address getLeader() {
    if(leader == null) { leader = getAddress(); }
    return leader;
  }
  public void exchange(Protocol other) {
    MinAddressLeader o = (MinAddressLeader) other;
    Address a = getLeader();
    Address b = o.getLeader();
    // Set leader to smallest address
    if(a.compareTo(b) > 0) { leader = b; }
    else { o.leader = a }
  }
  ...
}

public interface Protocol {
  public void setSelector(Selector selector);
  public Selector getSelector();
  public void exchange(Protocol other);
}
public interface Overlay extends Protocol {
  public Collection<Address> getView();
}



public interface Selector {
  public Address selectHost();
}
public class RandomSelector implements Selector {
  private Collection<Address> view;
  public RandomSelector(Collection<Address> view) {
    this.view = view;
  }
  public Address selectHost() { ... }
}
```

Figure 4: Simple leader election protocol in CPG. (Some boilerplate code elided for brevity)

CHAPTER 4

MICA: A COMPOSITIONAL ARCHITECTURE FOR GOSSIP PROTOCOLS (ECOOP 2014)*

Abstract

The developers of today's cloud computing systems are expected to not only create applications that will work well at scale, but also to create management services that will monitor run-time conditions and intervene to address problems as conditions evolve. Management tasks are generally not performance intensive, but robustness is critical: when a large system becomes unstable, the management infrastructure must remain reliable, predictable, and fault-tolerant.

A wide range of management tasks can be expressed as *gossip protocols* where nodes in the system periodically interact with random peers and exchange information about their respective states. Although individual gossip protocols are typically very simple, by composing multiple protocols one can create a wide variety of interesting, complex functionality with strong (albeit probabilistic) robustness and convergence guarantees. For example, in a system with a sufficiently dense topology, all nodes will learn the information being disseminated in expected logarithmic time. Unfortunately, programmers today must typically build gossip protocols by hand—an approach that makes their programs more complicated and error-prone, and hinders attempts to optimize gossip implementations to achieve better performance.

*Princehouse, L., Chenchu, R., Jiang, Z., Birman, K., Foster, N., Soulé, R. MiCA: A Compositional Architecture for Gossip Protocols. ECOOP 2014.

MiCA is a new system for building gossip-based management tools that are highly resistant to disruptions and make efficient use of system resources. MiCA provides abstractions that enable expressing gossip protocols in terms of functions on pairs of node states, along with a rich collection of composition operators that facilitates constructing sophisticated protocols in a modular style. The MiCA prototype realizes these abstractions on top of the Java Virtual Machine, and implements optimizations that greatly reduce the number and size of messages used.

Introduction

Monitoring and management infrastructure is critical for ensuring the reliability of modern cloud computing applications. In practice, each application typically has a distinct notion of what constitutes a healthy system state. For example, a scientific computing application might be especially sensitive to CPU utilization, while a database application might depend on the size of buffer queues, and the throughput of a streaming video service might be determined by available network capacity. Other examples include distributed hash tables, which must build and maintain structured overlay networks, and data mining applications, which must ensure the convergence of results produced by iterative computation.

Unfortunately, programmers today typically develop monitoring and management infrastructure by hand—a rudimentary approach that leads to a number of practical problems. First, because they lack tools that provide high-level abstractions, programmers must deal with a host of low-level details such as

setting up and maintaining network connections, serializing and deserializing application data, and dealing with exceptions and failures. Second, because standard infrastructure is not available, they must reimplement conventional algorithms, such as computing the minimum value in the system, from scratch in each new tool. Third, when several different tools are deployed on the same platform, the aggregate behavior can be unpredictable and can produce unexpected errors—nullifying the very properties the tools were designed to ensure!

Clearly, there is a growing need for higher-level frameworks that would enable programmers to rapidly build robust monitoring and management tools. To address this need, this paper presents MiCA (Microprotocol Composition Architecture). Unlike frameworks based on pub-sub [24, 14] or any-cast [35, 8] communication models, MiCA is based on gossip. In a gossip protocol, each node exchanges information with a randomly selected peer at periodic intervals. Because it is based on periodic peer-to-peer communication, gossip's network load tends to be well-behaved, scaling linearly with system size and not prone to reactive feedback. Moreover, because peers are selected randomly, no single node is indispensable, so tools built on gossip are extremely tolerant to disruptions and able to rapidly recover from failures. Accordingly, gossip is an attractive choice for system monitoring tools [62, 56, 63], network overlay management [32], and even distributed storage systems [62, 18, 40, 42].

MiCA enables programmers to describe gossip protocols in terms of three functions: a function `view` that is used to determine peers to gossip with; a function `update` that takes states of gossiping nodes and computes the new states following an exchange; and a function `rate` that determines how frequently exchanges should occur. This abstraction exposes the essential charac-

teristics of gossip protocols, but hides low-level implementation details such as how random numbers are picked, how network connections are managed, and how protocol messages are constructed. Because the MiCA run-time system handles all these details, programmers are free to focus on higher-level issues.

To facilitate building more sophisticated protocols, MiCA also provides a collection of composition operators that combine several smaller protocols into a single larger one. These operators are made possible by MiCA's abstractions, which provide a clean interface for merging protocols while preserving their essential behavior. As examples of protocol composition, a MiCA programmer might develop a layered protocol that first creates a tree overlay on top of an otherwise unstructured network and then aggregates data values up the tree. Or, they might implement a transformation that takes an unreliable protocol and makes it fault-tolerant by running multiple copies of the protocol concurrently in a pipeline [4]. Protocol transformations of these kinds would be extremely tedious to implement by hand but are easy to express in MiCA.

Describing gossip protocols using higher-level abstractions provides the MiCA system with opportunities for optimizing implementations of protocols automatically. For example, although the `update` function is defined on pairs of node states, the compiler can often determine that only a portion of the state of each node actually needs to be serialized and sent over the network using program analysis. In composite protocols, the run-time system can often bundle messages from different sub-protocols together, thereby reducing the communication cost of running those protocols simultaneously. Consequently, MiCA programs can provide correct behavior and predictable performance, while substantially reducing overhead compared to hand-written code.

We have built a prototype implementation of MiCA and used it to implement a wide range of standard protocols. To evaluate the performance of our system, we have performed experiments using MiCA on a collection of micro-benchmarks and simulations. Overall, these experiments demonstrate the effectiveness and robustness of our approach—in particular, that MiCA effectively bounds the costs of monitoring applications with hundreds of distinct components.

In summary, the main contributions of this paper are as follows:

1. We design a novel framework for building gossip protocols that captures their essential features while eliding tedious low-level implementation details.
2. We develop a collection of primitive gossip protocols and well-behaved protocol composition operators that satisfy natural correctness criteria.
3. We present our implementation and results from experiments illustrating the expressiveness and robustness of our framework.

The rest of the paper is structured as follows: § 4 and § 4 motivate MiCA's design using intuitive examples and experimental results from a simple simulation; § 4 describes operators for composing protocols and discusses correctness; § 4 discusses state management and an optimization; § 4 describes the MiCA prototype; § 4 presents an evaluation; § 4 discusses related work; and § 4 concludes.

Overview

This section introduces MiCA, using an epidemic protocol as a running example.

Assumptions. MiCA is based on a model of gossip in which the behavior of the system emerges from frequent pairwise interactions between nodes in the system. We call each interaction an *exchange*, and the nodes participating in an exchange a *gossip pair*. The state of the system evolves as the result of repeated, concurrent exchanges.

This model reflects several assumptions that hold in real-world cloud computing and data center environments: messages may be reordered or lost by the network, and the local clocks on each node all run at the same rate (though the clocks need not be synchronized). The evolution of the system state proceeds in loose rounds, with each correctly functioning node initiating a gossip exchange once every unit of time. Although the probabilistic nature of this model means that gossip protocols do not provide firm guarantees at fine-grained time scales, the expected behavior of the system over time can be reasoned about accurately.

Failures are inevitable in any real-world system, and systems based on gossip protocols are no exception. MiCA uses a failure model that includes both fail-stop and Byzantine nodes: nodes may crash and messages may be forged or lost, either due to network faults or malicious code executing on some of the nodes in the system. We do assume, however, that all messages are well formed and that malfunctioning nodes do not overwhelm the system by sending messages at arbitrary rates (an assumption that could be enforced by the network

itself).

These assumptions mean that failures can prevent an otherwise correct node from gossiping in any particular round, but over time, such failures are likely to be vastly outnumbered by successful exchanges. Primitive gossip protocols are expected to tolerate transient failures—*e.g.*, selecting sufficiently long rounds to prevent endemic timeouts—and programmers are expected to avoid pathological topologies and communication patterns that could lead to partitions or bottlenecks. In practice, most gossip protocols are designed to overcome transient faults and achieve convergence under less than ideal network conditions.

Programming model. The programming abstraction provided in MiCA closely follows the informal model of gossip protocols just described. With MiCA, programmers write gossip protocols by specifying the implementation for one participant node. Each participant in a protocol is a Java object implementing the following interface:

```
interface GossipParticipant {
  ProbMassFunc<Address> view();
  double rate();
  void update(GossipParticipant other);
}
```

The first method, `view`, controls peer selection during gossip exchanges. Unlike other gossip systems, which assume uniform random selection from a set of neighboring nodes or the global set of nodes, MiCA allows the programmer to specify the view as a discrete probability distribution on the set of network addresses. The MiCA run-time samples this distribution to select a gossip peer. The `view` method returns a probability mass function object (i.e.,

`ProbMassFunc`), which supports a `sample` method. As we will discuss in § 4, MiCA composition operators ensure that the probability mass function is scaled to provide a proper distribution over gossip nodes.

This approach has several advantages. First, working with probability distributions allows greater flexibility than uniform random selection. For example, probabilities can be used to encode notions of locality ("gossip more frequently with nearby neighbors") and capacity ("gossip more frequently with super-peers"), and even to encode overlay topologies [32]. Second, it allows developers to implement their protocols as if they were deterministic. Sources of non-determinism (e.g., peer-selection) are abstracted away and handled by the MiCA runtime. This makes programs simpler and eliminates a potential source of bugs. Third, it retains precise information about distributions and makes them available for analysis and manipulation by other operators. In particular, these distributions are used heavily by MiCA's composition operators—*e.g.*, composing two protocols with uniform random peer selection over different sets of nodes yields a non-uniform distribution over the union of those sets—unlike other systems, where views are sampled and discarded prior to composition, losing opportunities for optimization.

The `view` function also serves as a way to delegate overlay topology maintenance to another software component. When populating the view, developers often need to pay attention to the structure of the selected nodes: correctness and convergence are usually tied to particular topological properties, which may not hold for ad-hoc topologies. The MiCA programmer can use Java's type system to declare these requirements; for example, a protocol that outsources its view to an overlay maintenance layer might accept this layer as an instance

```
class MinFinder implements GossipParticipant {
  int value;
  ProbMassFunc<Address> view;
  MinFinder(int value, ProbMassFunc<Address> view) {
    this.value = value;
    this.view = view;
  }
  ProbMassFunc<Address> view() { return view; }
  double rate() { return 1.0 }
  void update(GossipParticipant other) {
    MinFinder that = (MinFinder) other;
    this.value = min(this.value, that.value);
    that.value = this.value;
  }
}
```

Figure 1: Anti-entropy protocol in MiCA

of the interface `ExpanderGraphOverlay`.

The second method, `rate`, specifies the local node's gossip rate relative to the basic unit of time. A constant rate such as 1.0 is usually sufficient for non-composite protocols, but variable rates are used by composition to multiplex sub-protocols without slowing down their overall convergence rates against wall-clock time. Per-node variable rates are also used by some gossip protocols, for example, as a mechanism to compensate for dropped packets [60].

The third method, `update`, takes the state of the gossip peer as input and performs an exchange, potentially modifying the states of the initiating node and the peer. Due to failures, one or both of the nodes may not actually be updated—modifications are not guaranteed to be atomic. However, the widespread success of gossip protocols testifies to the utility of this abstraction, and its simplicity: programmers are able to work with pairs of node states rather than having to explicitly send and receive messages, and the tedious logic needed to manually deal with timeouts and failures is subsumed by the model.

Example

As an example, consider the MiCA program in Figure 1. `MinFinder` nodes implement a simple epidemic protocol that, given a system in which nodes initially contain arbitrary integer values, eventually converges to a global system state where every (correctly functioning) node contains the minimum value in the system. The `view` method returns a probability distribution on network addresses. For the purpose of this example, we assume the view is known in advance and is supplied as a parameter to the constructor. The `rate` method returns a constant indicating that 1.0 gossip exchanges should occur every round. The `update` method implements a push-pull anti-entropy protocol: it compares the values stored on the initiating node and the receiving node, and updates both values to the minimum. It is worth pointing out that while the `update` method allows developers to transmit data between nodes, it is ultimately the MiCA runtime that determines which data is sent. As a result, the runtime can optimize the exchange. For example, if it can determine that some data will not be used by an update, it will only send the relevant subset of the data. It is straightforward to show that `MinFinder` participants converge to the minimum value in expected logarithmic time (in the absence of failures) on a complete graph [19].

Naïve Composition

Cloud computing platforms such as Amazon EC2, Microsoft Azure, IBM Websphere, Google Compute Engine, and Facebook consist of tens or even hundreds of thousands of individual components that must be monitored to ensure

the health of the platform. Gossip protocols provide a simple way to ensure that monitoring tools will behave predictably and have bounded communication costs. However, while it is not difficult to monitor multiple components of a system simultaneously—one can fork a new process for each component—combining tasks naïvely leads to increasing demands on system resources such as CPU, memory, and network bandwidth. In large systems, these demands can cause the cost of monitoring to rapidly dominate the very system being monitored. Addressing this issue is one of the primary motivations for MiCA.

To quantify the cost of naïve composition (and the potential for optimization) we conducted an experiment in which we executed several monitoring tasks simultaneously. We executed an increasing number of copies of an anti-entropy protocol and measured CPU utilization, memory utilization, and network latency. Intuitively, this experiment can be thought of as modeling the situation where an administrator must monitor an aggregate value for each of a large number of components. We ran the experiment on a testbed consisting of 32 virtual machines on a Eucalyptus cluster. Each VM was configured with an emulated 2.9GHz CPU, 4GB memory, 10GB ATA disk, and 1Gb/s NIC. The physical nodes hosting the VMs were 15 Dell-R720 servers with two 8-core 2.9GHz E5-2690 CPUs, 96GB RAM, 2×900GB disks, and two 10Gb/s Ethernet NICs each.

The results of the experiment are given in Figure 2. They show that CPU, memory, and network utilization rapidly increased under naïve composition, whereas MiCA was able to scale out to hundreds of monitoring tasks with only a little additional cost compared to running a single copy of the epidemic protocol. For example, with 200 monitoring components, CPU utilization on each

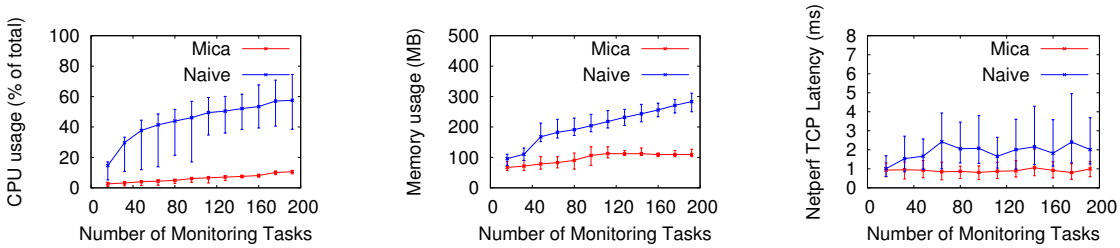

Figure 2: The average CPU, memory, and network utilization when running an increasing number of monitoring tasks with both naïve composition and MiCA.

instance exceeded 50% and required 250MB of memory, and network latency for other traffic was increased by a factor of two. Overall, this experiment demonstrates how interactions between monitoring components can incur substantial costs, and highlights the benefits that can be gained using optimized implementations of higher-level abstractions provided in systems such as MiCA.

Protocol Combinators

MiCA not only helps developers build complex monitoring tools out of simpler reusable components—it also provides operators that combine protocols while preserving semantics and guaranteeing predictable performance. As motivation for these operators, suppose that we want to execute two copies of the `MinFinder` protocol: one copy to compute the minimum address in the system, and a second copy to compute the smallest amount of free memory of any node in the system. Why might we want to do this? Perhaps the first copy implements leader election and the second implements a monitoring application. Using the abstractions described in the last section, it would not be difficult to construct a new `MinFinderTwo` protocol that implements both tasks. This protocol would maintain a pair of values, and would update both components

of the pair on each exchange. Of course, it would be even better if we could simply reuse our existing implementation of `MinFinder` instead of building a whole new protocol from scratch. This section presents composition operators that do just this—merging one or more gossip protocols into a single protocol that implements the behaviors of each sub-protocol.

There are many different ways of combining protocols. MiCA compositional operators can be categorized along two axes: whether the *state* and *communication* of the composed protocols are *isolated* or *shared*. Table 4.1 presents an overview of various approaches for protocol composition:

- *Isolated state, isolated communication*: This is the naïve multiplexing approach discussed in § 4, in which each protocol executes completely independently. As demonstrated by our simulations, this approach does not scale.
- *Isolated state, shared communication*: This approach provides communication primitives that can combine messages with the goal of reducing network congestion. This approach is used in pub-sub message buses, like TIBCO [59], and message-storage middleware, such as IBM WebSphere MQ [69]. POSIX streams also provide a similar style of message multiplexing.
- *Shared state, isolated communication*: This approach enables a single application to have many subsystems, each of which is monitored independently. For example, each job in MapReduce [17] runs in its own thread and communicates independently, but the overall system state is shared. Examples of this kind of system include JXTA [35] and Bast [24].
- *Shared state, shared communication*: This new approach combines the advan-

		Communication	
		Isolated	Combined
State	Isolated	With this naïve implementation strategy, each application is completely independent.	Subsystems cannot share state, but can multiplex messages (e.g., MQ[69], TIBCO[59]).
	Combined	An application can have many shared subsystems, but each communicates independently (e.g, JXTA[35], Bast[24]).	Composition reduces the overhead of executing multiple monitoring applications simultaneously (e.g., MiCA).

Table 4.1: Forms of gossip protocol composition.

tages of the previous two, allowing a single application to be expressed in terms of several sub-protocols whose state depends on each other, while reducing communication overhead by bundling messages together.

Note that although Table 4.1 locates MiCA in the quadrant for shared-state and shared-communication, MiCA actually provides a comprehensive suite of composition operators that capture each of these forms of composition. The rest of this section discusses correctness criteria for protocol composition operators, and then presents the operators that we find most useful in applications in detail.

Correctness Properties

To reason effectively about a composite protocol, programmers need assurance that the semantics of the combined protocol faithfully encodes the behavior of each sub-protocol. This section identifies essential properties for gossip composition:

- *View preservation:* A *view-preserving* operator ensures that the ratio of the frequencies with which it initiates gossip exchanges that update sub-protocols are identical to the ratio (calculated pointwise) of the distributions generated by each sub-protocol's `view` method. In other words, the rate of events where the composite chooses to execute P_i.`update` may be reduced or increased, but must be done so uniformly for all nodes in P_i's view.

- *Rate preservation:*

  A *rate-preserving* operator ensures that each sub-protocol continues to run at the same wall-clock rate as it would if run in isolation. Of course, there is a tension between view preservation and rate preservation: to ensure the former, a composite protocol must only execute each sub-protocol on certain exchanges, while to ensure the latter, it must not delay the rate at which the sub-protocol gossips.

- *State preservation:* A *state-preserving* operator ensures that the effect on the state of each sub-protocol is either the outcome of executing the `update` method of that sub-protocol or a no-op. In other words, composition does not introduce any co-mingling of sub-protocol states. Note that deliberate state sharing is still allowed—indeed, it is vital for building layered protocols where a lower-level protocol computes some form of state (such as a mesh-overlay), which is imported as a read-only input by one or more higher-level protocols layered over it. In the context of MiCA, state corresponds to an instance of a `GossipParticipant`, and everything reachable from it.

Together, these properties facilitate reasoning about composite protocols in

```
class RoundRobinMerger implements GossipParticipant {
  GossipParticipant g1, g2;
  boolean g1Next; // if true, g1 gossips next
   ...
  ProbMassFunc<Address> view() {
    if (g1Next) return g1.view();
    else return g2.view();
  }
  double rate() { return g1.rate() + g2.rate(); }
  void update(GossipParticipant other) {
    RoundRobinMerger that = (RoundRobinMerger) other;
    if (g1Next) g1.update(that.g1);
    else g2.update(that.g2);
    g1Next = !g1Next;
  }
```

Figure 3: Round-robin merging. Note: assumes `g1` and `g2` to gossip at the same rate.

a modular way: the programmer can write, reason about, and deploy a smaller protocol within a larger composite, and understand the way that it will behave without having to consider the entire program. They serve as guides while designing and debugging the operators presented in the rest of this section.

Operators

We now define a few useful MiCA composition operators. We begin with an obvious operator, round-robin merging, whose behavior is intuitive but restrictive and inefficient, before moving on to more sophisticated probabilistic operators.

Round-robin merging. Arguably the most obvious way to merge multiple protocols into a single protocol is to interleave their operations in round-robin fashion. Figure 3 defines a simple composition operator that does exactly this: given sub-protocols `g1` and `g2`, it alternates between `g1` exchanges and `g2` ex-

changes, using a boolean `g1Next` to keep track of the next sub-protocol to execute. For reasons discussed below, this operator assumes that the `rate` methods of `g1` and `g2` are equivalent. The `view` method branches on `g1Next` and dispatches the `view` method from `g1` or `g2`. The `update` method is similar, but also updates `g1Next` so that the other protocol will execute on the next exchange. The `rate` method is slightly different: it returns the *sum* of the rates for `g1` and `g2`. This is correct since doubling the rate of the combined protocol compensates for the fact that each sub-protocol is only able to initiate an exchange every other round. Hence, the rate at which each sub-protocol converges will be preserved in the composite protocol. Note that if `g1` and `g2` have different rates, then it would be incorrect to combine them using round-robin merging—a more sophisticated strategy would be needed to account for the rate disparity. The next operator provides a possible approach.

Correlated merging. Another way to combine several protocols into one is to do so probabilistically. That is, instead of alternating between the sub-protocols in sequence, we can invoke the `view` methods to compute the probability distributions for each sub-protocol and construct a composite distribution that represents the peer selection preferences of both. This approach takes advantage of the fact that both sub-protocols may sometimes be willing to gossip with the same peer, allowing execution of both `update` methods to be *bundled* into a single exchange and reducing the overall number of messages sent without degrading performance. The correlated merge operator (Figure 4) is aggressive in trying to exploit this form of overlap—it bundles messages as often as possible while still satisfying the view-preservation and rate-preservation properties. Because this operator is somewhat involved, we step through each of its methods in detail.

```
class CorrelatedMerger implements GossipParticipant
  GossipParticipant g1, g2;
   ...
  ProbMassFunc<Address> view() {
    double r1 = g1.rate();
    double r2 = g2.rate();
    double w = r1 / (r1 + r2);
    ProbMassFunc<Address> d1 = g1.view().scale(w);
    ProbMassFunc<Address> d2 = g2.view().scale(1−w);
    return ProbMassFunc.max(d1, d2).normalize();
  }
  double rate() {
    double r1 = g1.rate();
    double r2 = g2.rate();
    ProbMassFunc<Address> d1 = g1.view().scale(r1);
    ProbMassFunc<Address> d2 = g2.view().scale(r2);
    return ProbMassFunc.max(d1, d2).magnitude();
  }
  void update(CorrelatedMerger other) {
    CorrelatedMerger that = (CorrelatedMerger) other;
    double r1 = g1.rate();
    double r2 = g2.rate();
    double w = r1 / (r1 + r2);
    double pr1 = g1.view().get(that) * w;
    double pr2 = g2.view().get(that) * (1−w);
    double pmin = Math.min(pr1,pr2);
    double pmax = Math.max(pr1,pr2);
    double alpha = (pr1 − pmin) / pmax;
    double beta = (pr2 − pmin) / pmax;
    double gamma = pmin / pmax;
    switch (weightedChoice({ alpha, beta, gamma })) {
    case 0: // only g1 gossips
      g1.update(that.g1); break;
    case 1: // only g2 gossips
      g2.update(that.g2); break;
    case 2: // both g1 and g2 gossip
      g1.update(that.g1);
      g2.update(that.g2);
    }
  }
}
```

Figure 4: Correlated merging.

The `view` method works more or less in the way just described: it computes the views for `g1` and `g2` and scales them by `w` and `(1-w)` respectively, where `w` is the relative weight of `g1`'s rate with respect to `g2`. It then computes the pointwise `max` of the scaled distributions and normalizes the result. This produces a distribution that reflects the peer selection preferences of `g1`

and `g2` with respect to their relative rates. This is equivalent to summing the two rate-scaled views and then subtracting their intersection, where the area of the intersection represents the fraction of correlation between views that can be exploited by bundling—two sub-protocols with identical views intersect completely, whereas two disjoint views have none. The `rate` method calculates the views for `g1` and `g2`, scales them by `r1` and `r2`, and then takes the area under the pointwise maximum of the resulting distributions. This calculation determines the rate needed to correctly execute both sub-protocols while preserving their rates, and anticipating opportunistic bundling of messages. The `update` method must decide whether to gossip `g1`, `g2`, or both. To do this, it uses the sub-protocol views to compute three probabilities: given that a particular peer was sampled from the composite view, let `alpha` be the probability that only `g1` chose to gossip with that peer, `beta` be the same for `g2`, and `gamma` be the probability that both nodes choose to gossip—i.e., the view intersection for the selected peer's address. A pseudo-random choice selects one of these three possibilities and executes the respective `update` methods.

Correlated merge has two significant advantages over simple round-robin. First, it is completely general, in that it does not make any assumptions about the protocols being combined. This is unlike round-robin merge, which assumes that the two sub-protocols gossip at the same rate. Second, it can greatly reduce the number of messages needed to implement the composite protocol; this is advantageous because it amortizes overheads over the messages in the bundle. The degree to which the operator is able to bundle messages depends on the amount of overlap in the peer selection preferences of `g1` and `g2`—the greater the overlap of their distributions, the greater the benefit.

To illustrate correlated merging, consider the following abstract examples.

- Suppose that `g1` gossips by selecting randomly from nodes with odd addresses, and `g2` by selecting randomly from nodes with even addresses. That is, if there are n nodes in total, `g1`'s `view` method returns a distribution where odd nodes have probability mass $2/n$ and even nodes have probability mass 0, and symmetrically for `g2`. Because these distributions are disjoint, the `view` method for the merged protocol returns the uniform distribution on all n addresses. For a given gossip partner b, the distribution computed by `g1` assigns probability mass 0 to b if b's address is even, and the distribution computed by `g2` assigns probability mass 0 to b if b's address is odd. The combined `update` method invokes `g1`'s `update` method when called with a partner b whose address is odd and otherwise invokes `g2`'s `update` method. Importantly, it never invokes both `update` functions as the peer selection preferences are disjoint. In a sense, probabilistic merge operator subsumes round-robin merging when the sub-protocol distributions are disjoint.
- Suppose instead that both `g1` and `g2` gossip by selecting randomly from all nodes—i.e., the `view` method for both sub-protocols returns a uniform distribution where every node has probability mass $1/n$. The combined `view` method returns the same uniform distribution and the `update` method evaluates `g1` and `g2` every round, where round length is a system-wide constant. This example shows how probabilistic merge allows protocols with equivalent `view` methods to be combined without additional messages or rate increases.
- Finally, suppose that `g1` gossips randomly with odd nodes, and `g2` gossips randomly with all nodes. The combined `view` method returns a distri-

```
class IndependentMerger implements GossipParticipant
  GossipParticipant g1, g2;
   ...
  ProbMassFunc<Address> view() {
    double r1 = g1.rate();
    double r2 = g2.rate();
    double w = r1 / (r1 + r2);
    ProbMassFunc<Address> d1 = g1.view().scale(w);
    ProbMassFunc<Address> d2 = g2.view().scale(1−w);
    return d1.add(d2).normalize();
  }
  double rate() { return g1.rate() + g2.rate(); }
  void update(IndependentMerger other) {
    IndependentMerger that = (IndependentMerger) other;
    double r1 = this.g1.rate();x
    double r2 = this.g2.rate();
    double w = r1 / (r1 + r2);
    double pr1 = g1.view().get(that) * w;
    double pr2 = g2.view().get(that) * (1−w);
    double alpha = pr1 / (pr1 + pr2);
    double beta = pr2 / (pr1 + pr2);
    switch (weightedChoice({ alpha, beta })) {
    case 0: // Only g1 gossips
      g1.update(that.g1); break;
    case 1: // Only g2 gossips
      g2.update(that.g2); break;
  }
}
```

Figure 5: Independent merging.

bution in which nodes with odd addresses are assigned probability mass $4/(3 \cdot n)$ and nodes with even addresses are assigned probability mass $2/(3 \cdot n)$. Hence, the run-time chooses peers with odd addresses twice as often as it chooses peers with even addresses. The combined `update` method has two cases: if the node has an odd address, it always invokes `g1`'s `update` method and additionally invokes `g2`'s `update` method with probability $1/2$. Or, if the node has an even address, then it only invokes `g2`'s `update` method. Hence, the merged protocol distributes exchanges evenly between `g1` and `g2`, allowing many exchanges with odd peers to execute both sub-protocols.

Independent merging. Although it is often advantageous to bundle messages from multiple sub-protocols together, there is also a downside to the correlated merge operator: the peer selection preferences of the sub-protocols are no longer independent. This could violate assumptions in a program that depends on independence. For example, the correctness of the random walk protocol developed by Massoulié et al. [47] depends on randomly sampling locations in the system. If we mistakenly composed two copies of this protocol using the correlated merging operator just defined, believing that this would yield samples from two distinct random walks, both instances would actually generate the same walks. Such problems could have dire consequences in systems whose robustness assumes independent peer selection. Another example involving random walks comes from Broder et al. [10], who solve the problem of generating independent paths between pairs of nodes with a random walk approach. More generally, any system relying on the independence of concurrent gossip protocols could be inadvertently sabotaged by the correlated merge operator. To address this concern, we present an independent probabilistic merge operator (Figure 5). Like correlated merge, independent merge makes probabilistic gossip choices, and combines sub-protocol `view` and `rate` methods. However, the independent merge ensures that the probabilistic decisions made by each sub-protocol are independent.

Epoch pipelining. The final operator presented in this section implements a completely different kind of composition. Rather than composing multiple sub-protocols in parallel, it composes a single protocol with itself, running two instances in a primary-backup configuration for enhanced fault tolerance.

As a motivating example, recall the `MinFinder` example from the previ-

```
class EpochPipeliner<G extends GossipParticipant> extends CorrelatedMerger {
  GossipParticipantFactory<G> factory = null;
  int epochLength = 0;
  int currentEpochStart = 0;
  EpochPipeliner(GossipParticipantFactory<G> factory, int epochLength) {
    super(factory.create(), factory.create());
    ...
  }
  void update(EpochPipeliner<G> other) {
    int now = getRuntimeState().getSystemClockRounds();
    if (now − currentEpochStart >= epochLength) {
      g1 = g2; // promote backup to primary
      g2 = factory.create();
      currentEpochStart = now;
    }
    super.update(other);
  }
}
```

Figure 6: Epoch-based "pipelining" operator.

ous section, which gossips the minimum value in the system using a simple anti-entropy protocol. This protocol converges rapidly to a stable state and is extremely robust—a small number of lost messages or transient failures have little affect on overall convergence. However, it is susceptible to a particular failure that can easily lead to unintuitive behavior. To illustrate, consider a system in which each node executes `MinFinder`. Next, suppose that after running the protocol for a while, the node that originally contained the minimum value crashes. What should happen? We might want the system to converge to the next smallest value in the system. But, assuming the crashed node successfully communicated with at least one other node, this is not what will happen. Instead, the system will continue gossiping the old minimum value even though none of the nodes in the system still have that value.

To address this problem, we can execute two copies of `MinFinder` side by side. The primary protocol, by convention `g1`, contains the definitive copy of the protocol while the backup protocol, `g2`, executes a second copy of the proto-

col from a fresh state. The composite protocol executes the two copies in parallel until a certain number of rounds have elapsed—sufficiently many to ensure that the backup copy has converged to a stable value. At that point, the composite protocol replaces the primary with the backup and resets the backup to a fresh copy of the protocol. It is easy to see that this "pipelined" protocol does not suffer from the anomaly described above, since the minimum value is recomputed from scratch in each epoch. Note that this implementation of pipeline parallelism requires system-wide clock drift to be less than one half of a round, to prevent possible contamination from the primary layer to the backup layer. This is a reasonable constraint in a data center, where round-trip communication times between nodes are no more than a few milliseconds.

We can define pipelining on top of any of the merging operators just defined. Figure 6 gives a definition using correlated merge operator. Note that the `view` and `rate` functions are inherited from the super class. The definition of a pipelining operator based on independent merge is similar, and preferable in many scenarios since it makes completely independent choices when selecting a peer. On the downside, however, it requires extra messages and an increased rate, whereas the operator based on correlated merge only requires larger messages since it can always bundle messages from each pipeline stage. A more general `EpochPipeliner` implementation might admit other implementations of epoch-switching, for example, triggered by a consensus threshold instead of a clock [20]. Finally, although we do not develop it here, one can define pipelining of k protocol copies at a time for higher levels of fault tolerance.

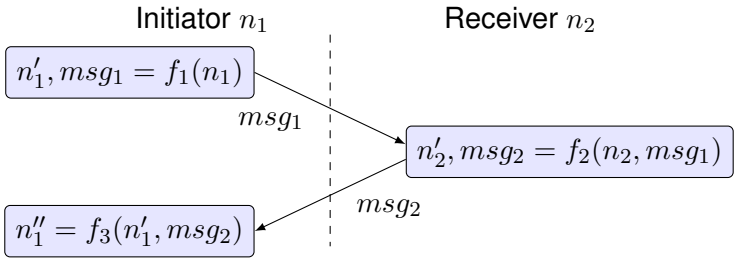

Figure 7: Execution of a gossip exchange with the explicit messages used by the low-level target of the MiCA compiler. Provided the synthesized functions f_1, f_2, f_3 are correct, the final states of both nodes are guaranteed to be the same as if `update` had executed locally: $(n_1'', n_2') = \texttt{update}(n_1, n_2)$.

State Management and Data Movement

MiCA is designed to abstract away the details of handling distributed state. In particular, developers write the `update` function with the illusion that each participating node is able to access the other's state as if it were local. In actuality, the `update` function is a distributed program that exchanges messages using the communication pattern illustrated in Figure 7. The MiCA compiler transforms the `update` function into the distributed implementation, and the MiCA runtime manages the exchange of state between the nodes.

To transform `update` into the distributed equivalent, MiCA partitions the function into three fragments, f_1, f_2, and f_3, that cooperate to execute the gossip exchange. First, the initiator of the exchange updates its own state by applying f_1, and sends its updated state to the receiver node in message msg_1. Next, the receiver executes its fragment, f_2, using the initiator state and its own state, and then returns its new state in msg_2. Finally, the initiator updates its state, using f_3, with the data from the receiver. Note that when partitioning the function into fragments, the compiler must ensure that the fragments obey the constraints imposed by the program dependence graph (PDG). So, f_1 cannot execute code

that may read state from n_2, and f_3 cannot execute code that may modify the state of n_2. This can be expressed as two cuts in the PDG, breaking `update` into three regions corresponding to f_1, f_2, and f_3.

Consistency Model. A key challenge for maintaining MiCA's local state abstraction is handling failures during the execution of `update`. Ideally, MiCA would provide guarantees about an exchange, even if failures occur. Unfortunately, it is impossible to guarantee the obvious property—transactional atomicity—because when a network fault is detected on a given node, that node has no way of determining whether the remote node has successfully completed its last phase. This means that the node cannot decide whether or not to roll back its local state or not (this is an instance of the classic Two Generals' problem).

To avoid these issues, MiCA employs a relaxed consistency model. MiCA saves node state before executing calls to `update`. If a network error is detected (including timeouts, which do not necessarily mean the message failed to reach its destination), the state is rolled back. All state changes that occurred during the unsuccessful update are erased by the rollback. This leaves four possible outcomes for each gossip exchange: each node completes successfully, or one or both revert to their original state. However, it precludes the possibility of corrupting state by interrupting `update` in the middle of its execution.

Communication Optimization. The simplest strategy to exchange state between the participants would be to send the entire state of each node. In contrast, MiCA uses an optimization to reduce the communication overhead. Rather than send the entire state, the compiler performs a static analysis that determines conservative sets of objects that may be read and may be modified

by f_1, f_2, and f_3. MiCA then generates custom serializers that send the relevant objects in messages msg_1 and msg_2. This analysis is currently performed at the granularity of fields of the root protocol objects. While coarse, this is a significant improvement over the naïve strategy, in that fields that will definitely not be used are not exchanged. It would be natural to duplicate the execution of side-effect-free code to further reduce the amount of state that needs to be transmitted, but MiCA does not currently implement this extension.

Implementation

We have built a full working prototype of MiCA, implemented as an extension to Java, and made it available under an open-source license. Our implementation can be obtained at: `https://github.com/mica-gossip/mica`. It includes the compiler and runtime, as well as a library of primitive protocols and implementations of the composition operators presented in this paper.

The MiCA compiler is implemented as a bytecode post-processor. Post-processing allows MiCA to partition the `update` function into methods for each node participating in the gossip exchange, and perform the static analysis for the communication optimization.

The current implementation uses TCP/IP for network communication. One connection is kept alive for the duration of the gossip exchange. However, the communication layer of MiCA does not depend on this particular implementation choice. In ongoing work, we are exploring an alternative implementation that uses UDP. Because gossip protocols are tolerant of failures, the unreliable communication mechanism seems like a natural choice if some performance

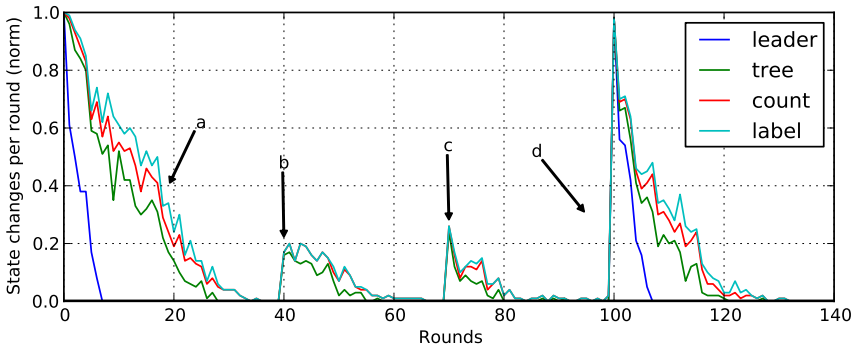

Figure 8: Convergence of all four layers. Arrows indicate (a) Convergence from arbitrary starting state; (b) a transient fault: 10% of nodes crash; (c) failed nodes recover; (d) a large artificial disruption of the bottom layer's state. Note that the leader election layer was not affected by the transient fault because the leader did not crash.

benefit can be gained due to smaller packet headers, reduced connection state, etc.

MiCA uses the Soot analysis framework [61] for analysis and transformation, and relies on Soot for computing the program dependence graph, points-to sets, and call graph. For functions f_1, f_2, and f_3, the remote node (either n_1 or n_2) is replaced with a custom-generated proxy class, inspired by the Uniform Proxies of Eugster [22]. An instance of this proxy class may represent a local or remote GossipParticipant object; in the case of a remote object, the proxy acts as a container for the subset of fields that may be necessary for remote execution.

Experience and Case Studies

To evaluate our design and implementation of MiCA, we asked volunteers in an undergraduate course to use MiCA for developing distributed applications. To explore how MiCA performs in real-world scenarios, we performed two case

studies in a simulated environment.

In the undergrad course, a number of students who had no connection to our research efforts used MiCA to develop their projects. Using MiCA, they developed a data replication protocol for use in coherent distributed caching, a probabilistic consensus protocol, a scalable distributed denial-of-service (DDoS) detection application, and a storage backend for a peer-to-peer social network.

The case studies were performed in a simulated runtime. This runtime simulates a gossip network of many logical nodes with a discrete event simulation passing messages via message queues on a single machine. All of the MiCA logic and state serialization is the same as in the TCP/IP runtime. The simulated runtime allowed us to perform experiments faster than realtime. For the first case study, we implemented a four-layer composite protocol that builds a tree over an otherwise unstructured topology and then labels the nodes of the tree according to a depth-first traversal. During execution, we introduced several disruptions, and measured the time needed for each layer to converge back to a stable state. This experiment demonstrates how MiCA facilitates building sophisticated protocols out of simple components, as well as the resilience of such composite protocols to various kinds of failures. For the second case study, we studied the effect on convergence times for protocols built using probabilistic merge. Because this operator changes the gossip rate for each sub-protocol from a deterministic to an probabilistic value, the expected convergence time is increased in certain topologies. This experiment illustrates this effect, which we call dilation, using another simulation.

Layered Protocol

The first case study is based on a four-layer composite protocol originally proposed by Dolev [21]. The layers represent several standard varieties of gossip, all working together: overlay maintenance, aggregation, and dissemination. The lowest layer, **leader**, gossips on a fixed topology and executes a standard leader election protocol. The leader selected by the lowest layer is then used by the second layer, **tree**, to construct a spanning tree overlay. The third and fourth layers, **count** and **label**, gossip over the tree overlay. The **count** layer recursively counts the number of nodes in each sub-tree and aggregates the results up the tree to the root, while **label** assigns a numeric label to each node, resulting in a depth-first traversal ordering. The labeling is achieved using a dissemination protocol: a parent assigns labels to its children based on its own label plus an offset calculated from the sizes of the children's sub-trees.

Unlike all the composite protocols we have seen so far, this layered protocol requires sharing state between the sub-protocols. For example, the protocol for the **tree** layer depends on the state maintained by the **leader** layer. It is straightforward to encode this behavior in MiCA—the programmer simply creates references between the sub-protocols using ordinary Java references. For example the following code creates the layers needed for the case study:

```
LeaderElection leader = new LeaderElection(topology);
Tree  tree  = new Tree(leader, topology);
Count count = new Count(tree);
Label label  = new Label(tree, count);
GossipParticipant g = new IndependentMerger(leader,
                                  new IndependentMerger(label,
                                    new IndependentMerger (tree, count)));
```

Note that sharing state between sub-protocols using references obviously breaks the state preservation property, albeit in a fairly innocuous way.

After implementing the layered protocol, we then executed it on a random topology in a simulated environment and measured the amount of time needed for each layer to converge under various disruptions. Figure 8 present the convergence results for all four layers on a 100-node random graph of degree four, starting from arbitrary initial states. To model failures, we introduced a transient disruption by crashing 10% of the nodes at $t = 40$ and restarting them at $t = 70$. At $t = 100$, we introduced a major disruption by clobbering the state of the **leader** layer with arbitrary values. We measured convergence as the normalized per-round rate of change: a value of 1.0 indicates that 100% of the nodes were changing in a given round while a value of 0.0 indicates the protocol has converged. As these graphs show, MiCA can be used to implement protocols that will recover rapidly from transient failures, even major ones, and even when several protocols are combined together.

We also ran the experiment using correlated merge instead of independent merge. This resulted in similar convergence times, but each gossip exchange bundled together the messages for 2.3 layers on average, dramatically reducing the total number of gossip exchanges by 56%. Note, however, that this is not a general result: this particular layered protocol is amenable to correlation because **count** and **label** always gossip together, as do **leader** and **tree**.

Dilation

The second case study illustrates an effect that we call *dilation*, and that can arise when protocols running at different rates are merged probabilistically. Recall that the rate of a gossip protocol controls the frequency at which the node initiates exchanges with another node. When a protocol runs in isolation, rate is deterministic: the node sleeps until the appropriate time, initiates an exchange with that node, and then sleeps again. However, in a composite protocol implemented using the probabilistic merge operator, a given sub-protocol will only be able to initiate gossip at an *expected* rate. In particular, although the average rate will faithfully track the value specified by the `rate` method for that sub-protocol, the variance of the distribution of the interval between gossip exchanges increases as sub-protocols are added to the composite.

To demonstrate this effect, we simulated the anti-entropy protocol from Figure 1, obtaining the results seen in Figure 9. The graph in the upper left corner gives the baseline: the protocol executes deterministically, and the distribution of intervals between exchanges is tightly clustered around 1.0 (because no packet loss occurs in this experiment, it would be exactly 1.0 were it not for measurement artifacts). The next graph, on the upper right, shows the effect when the protocol is composed with another protocol using probabilistic merge. Now the distribution contains values ranging from less than 1.0 all the way up to 5.0. That is, some exchanges occur faster than the stated rate, and some occur slower, even though the average exactly matches the target rate. As additional sub-protocols are added to the composite, shown by the graphs on the bottom row, the dilation becomes increasingly evident.

A natural question to ask is whether this phenomenon affects important properties of a protocol, such as convergence. The answer is that it can, depending on the protocol and topology, but significant consequences are seen only in somewhat artificial situations. Figure 10 depicts the convergence rate for the anti-entropy protocol with various degrees of dilation on a system whose topology is a complete graph.

The x-axis contains the number of gossip rounds and the y-axis contains the number of changes induced on that round. A protocol converges when the number of changes reaches 0. In a complete graph topology, the effect of dilation is minimal: because we are executing an anti-entropy protocol and every node is connected to every other node, overall convergence does not hinge on specific nodes being able to gossip at particular moments. We believe that this would be the most common case in real uses of MiCA.

Note that dilation does not imply probabilistic merge is incorrect—on the contrary, all our operations correctly produce protocols that faithfully implement the sub-protocol, and faithfully run them at the correct average rate. The point is somewhat more subtle: what we see here is that turning a deterministic behavior into a probabilistic one can sometimes slow convergence if the underlying topology has a slow information-dissemination time, but would not have this impact when running on a topology with the properties of an *expander graph*, of which the complete graph is an extreme example. We plan to continue studying dilation in the future, with the goal of fully characterizing the classes of protocols and topologies that are guaranteed to be immune to this effect. We are also exploring other ways to implement the composition operators that incorporate mechanisms for limiting or otherwise bounding the effects of dilation.

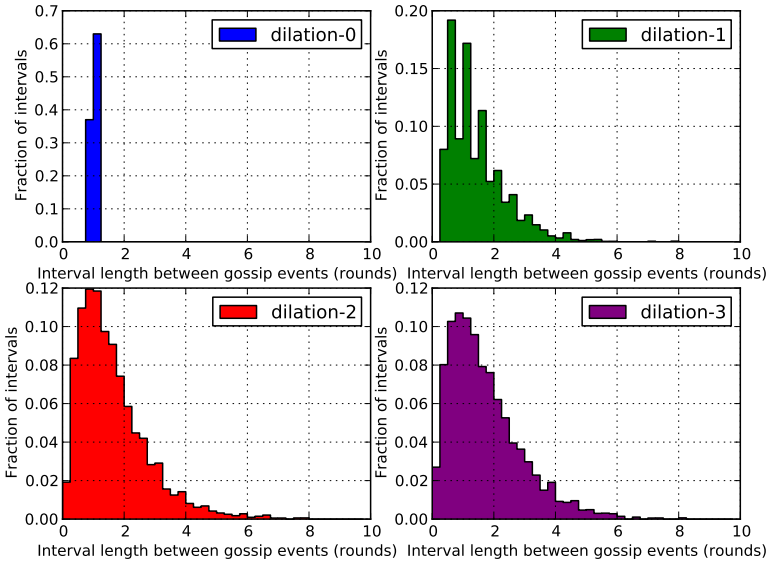

Figure 9: Effect of dilation for an anti-entry protocol on intervals between gossip exchanges. The labels indicate the degree of dilation: `d0` is no dilation, `d2` is two nested operators, etc.

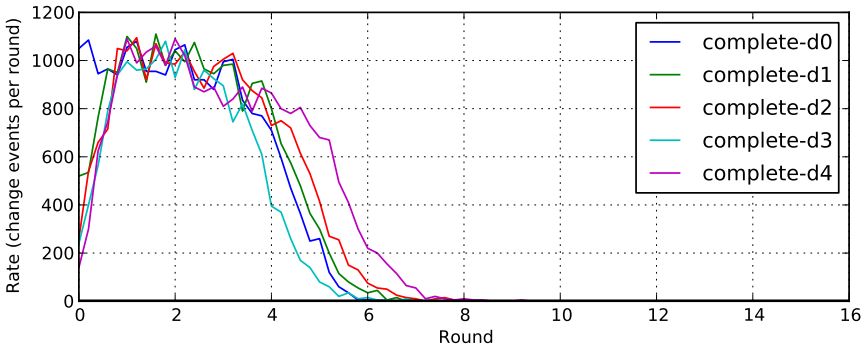

Figure 10: Effect of dilation for an anti-entropy protocol in a complete topology. The labels indicate the degree of dilation: `d0` is no dilation, `d2` is two nested operators, etc.

Related Work

Work related to MiCA falls into several general categories: gossip-specific frameworks (Opis [14], Gossip Objects [66]); object-oriented distributed system libraries (Bast [24], Jini [68]); compositional network transport protocol systems

(Appia [48], Cactus [71]); and languages and abstractions for distributed programming (P2 [45], MACEDON [53], BLOOM [1]). In this section, we discuss each of these in turn. It should also be noted that MiCA's core abstraction—the pairwise representation of gossip protocols—was originally presented in a short workshop paper [51]. This earlier work did not define gossip protocols precisely and did not include an implementation or experiments.

The first of these categories contains systems closest to MiCA, namely, those concerned specifically with gossip. *Opis* [14] is an OCaml-based framework for gossip. It offers a formal definition of gossip similar to that used in MiCA. In Opis, gossip protocols are event-driven programs that react to user-defined external network events and internal timer events. This is an interesting contrast to MiCA's protocol representation, which could also be regarded as using events to drive state changes, but has only a small, fixed number of state transitions exposed to the programmer. Like MiCA, Opis leverages object-oriented composition for protocols, but with added benefit from OCaml's rich type system. However, Opis offers no analog to MiCA's compositions, which consider not only the object-oriented composition of classes, but also explore strategies for semantic-preserving combination of protocol views.

The *Gossip Objects* framework [66] offers a compositional infrastructure for publish-subscribe gossip protocols. Unlike MiCA, Gossip Objects is an implementation specifically for publish-subscribe gossip, and not a general framework. Like MiCA, Gossip Objects has optimizations for running many concurrent systems. Composition takes the form of speculative message delivery, bundling messages to non-subscribers in an effort to have them delivered indirectly and accelerate the overall gossip rate. Gossip Objects does not preserve

the relative rates of protocols being combined. This is a design decision, not a bug: Gossip Objects' purpose is to improve the efficiency of message delivery.

The next category of related work consists of general-purpose, object-oriented approaches to building distributed systems. These frameworks do not provide MiCA's gossip-centric world view, but do share a common philosophy for protocol composition. *Bast* [24] is an object-oriented library of distributed system components, whose main goals were modular composition and code reuse. The platform introduced a primitive group type and allowed developers to define subtypes supporting additional properties. The primary focus in Bast was on atomic broadcast with various levels of ordering and durability. For example, a database built using Bast might obtain ACID guarantees by exploiting ordering and other atomicity properties of the underlying groups (e.g., in implementations of locking or propagation of updates to replicas). However, while Bast's Java implementation is similar to MiCA's in that both represent protocols as classes and use object-oriented composition mechanisms such as inheritance, MiCA focuses on gossip protocols, and on optimizations that reduce communication while preserving semantics. To the best of our knowledge, Bast never explored gossip protocols, and generally avoided transformations where knowledge of protocol semantics would be needed.

Apache River [68] (originally Jini) is a Java framework for client-server distributed services, originally created by Sun Microsystems. It provides extensible components for service registration and discovery for distributed systems, and other utilities to facilitate distributed systems programming such as remote method invocation and mobile code. Less broad than Bast, it is a good example of an off-the-shelf component available to Java developers building distributed

systems. River's services are good examples of the protocol layers that could be implemented in a MiCA stack.

Cactus [71] and Appia [48] both undertake the challenge of transport protocol composition. Recognizing that transports like TCP and UDP are not ideal for all situations, these two systems provide ways to modularly compose a transport protocol that has desired properties; for example, Cactus could be used to satisfy the statement "I need a transport protocol with congestion control, but I don't need reliable ordering". Cactus includes a library of "micro-protocols", each of which implements a particular functionality; the philosophy of composition is similar to MiCA's. Although MiCA gossip protocols run at a layer above the transport, some functionality, such as quality-of-service, could be implemented either in transport or as a MiCA gossip layer.

Finally, there are languages designed for directly programming an entire distributed system. Although MiCA is not a language, its distribution of the update function onto a pair of nodes is similar to what these whole-system languages accomplish. P2 [45] and Bloom [1] are declarative languages that approach distributed systems programming from a databases perspective. P2 allows programmers to specify properties of distributed system state and compiles to a dataflow-oriented runtime system. Bloom is a Ruby-like language, designed for efficient and concise query execution on distributed data tables. MACEDON [53] is a language for building P2P-style overlay networks. Like MiCA, it uses a domain-specific language extension to describe its systems; unlike MiCA, its domain is not gossip, but overlay networks. The programmer writes from a single-node perspective, but MACEDON includes tools for analyzing whole-system behavior.

Future Work

Today's data center operators lack tools for creating new services to manage networks and applications, both within enterprise networks and even in the new class of wide-area enterprise VLANs that span between today's massive cloud-computing data center systems. This paper presents MiCA, a new compositional architecture and system for building network management protocols. The system assists developers in creating applications from micro-protocols implemented using gossip or self-stabilization mechanisms, which can then be composed in a property-preserving manner to build sophisticated functionalities. Unlike protocols built in a more classical manner, which have been known to misbehave in unexpected and disruptive ways when deployed on a very large scale, MiCA yields scalable solutions with absolutely predictable, operator-controlled, worst-case message rates and sizes. Using the techniques of the gossip and self-stabilization communities, the developer creates components that are provably convergent under the MiCA run-time model. Moreover, the framework provides abstractions for composing protocols while preserving semantics and optimizing across components to make the best possible use of available communication resources. In this manner, MiCA makes it easy to build the massively scalable applications needed to efficiently operate today's data centers.

Conclusions

The essential idea of Code-Partitioning Gossip is that of writing a gossip exchange as a single atomic function, and then automatically partitioning this

function into code for the roles of sender and receiver. We believe this technique offers several advantages.

Composition

A distributed hash table system might make use of several gossip protocols: A peer sampling protocol to draw adequately random samples from its members, an overlay maintenance protocol to adjust the overlay according to node arrival and departure, a counting protocol to estimate the number of nodes in the system, and an aggregation protocol to estimate the most popular objects to allow nodes to make better caching decisions. The status quo would implement this bundle of protocols in one of two ways: Either as a single monolithic protocol, or as four separate protocols that operate independently, each running its own active and passive threads. In this situation it is difficult to reap any benefit from commonality in communication or computation without significant code rewriting, unless perhaps all four protocols have been written using a middleware layer that abstracts away low-level network code. Using Code-Partitioning Gossip, however, the protocols are composed *prior to partitioning*. Instead of trying to merge multiple active and passive threads, we invoke each protocol's update method from within the a single superior update method, which is then partitioned CPG allows protocols to be composed in much the same way functions and objects are composed in object oriented programming. This composition can take the form of a top-level update function that calls the update functions of sub-protocols, or of extending a protocol by inheriting it. We have a cursory implementation layered self-stabilizing protocols[21] using CPG, but more work is needed to evaluate the real utility of CPG.

Analysis

Code-Partitioning Gossip also offers the possibility of using program analysis tools to analyze the behavior of gossip protocols. Gossip protocols in the literature are often presented with dual representations: One as a low-level implementation proof-of-concept, and one high-level theoretical representation used for analysis. Code-Partitioning Gossip unifies these two representations by providing a representation that can be partitioned into a working implementation, but also is abstract enough to facilitate formal reasoning, notably by containing all stateful effects of a gossip exchange within a single deterministic function. Program analysis tools could be used to prove, for example, that if some predicate P holds for an pair of node states before a gossip exchange, it also holds afterwards, where predicate P would be written in the same language as the protocol's implementation. We leave this as future work.

CHAPTER 5

IMPLEMENTATION

Chapter chapter 4 gives a summary of the MiCA proof-of-concept implementation. This chapter goes into greater detail.

The proof-of-concept MiCA implementation consists of more than 13,000 lines of Java code, implementing a runtime and simulator for MiCA, and over 3,000 lines of Python analysis tools.

The MiCA Runtime

Each MiCA session instantiates a *Runtime* class. MiCA protocol instances are created via this runtime, which stores the state for all MiCA instances on a node. In a true TCP/IP experiment with many real or virtual servers, each runtime will typically only host one MiCA instance, but it is convenient to host more than one in a single runtime for experiments and simulation. The MiCA runtime is the consumer of the functions that define a MiCA protocol: `update`, `view`, `rate`. The runtime is responsible for creating the illusion that MiCA's compiled JVM bytecode is being executed as a conversation between two nodes.

There are three Runtime back-end implementations:

- Code Partitioning Runtime, which employs Soot [61] as describe earlier to statically determine a conservative set of protocol instance members that may be needed to compute `update` on the other node in a gossip exchange pair. Code partitioning seeks to use static analysis to reduce the amount of state that must be communicated in order to compute a protocol's `update`

function during a gossip exchange. The MiCA implementation performs this analysis on the members of gossip classes. It is unsound, in that certain Java idioms (such as use of static members) will cause it to execute `update` incorrectly.

- Simple Runtime. For testing, this runtime does not perform and optimizations; it simply serializes every instance that gossips and sends the serialized object to the remote node so the remote can compute the `update`.
- Simulated Runtime. The simulation can simulate MiCA either with or without code partitioning. Network communication and time are both simulated, allowing simulations to execute much faster than real time.

MiCA nodes write log files detailing who they gossiped with, the various states of gossip they undergo, and documenting their own state changes. Graphs in this dissertation are produced by analyzing these logs.

Visualization and Analysis

MiCA's Python tools include a GUI event visualizer for MiCA's copious log files, and a set of Jupyter modules that let a researcher quickly query and plot information derived from MiCA logs. The visualizer, named "Micavis", operates on event traces represented in logs collected from a gossip system execution. It is able to scroll backwards and forwards through time, replaying the log traces as appropriate to show some facet of global system behavior. It makes the assumption that node clocks are synchronized, or at least synchronized "well enough"—to a much closer value than the time between gossips.

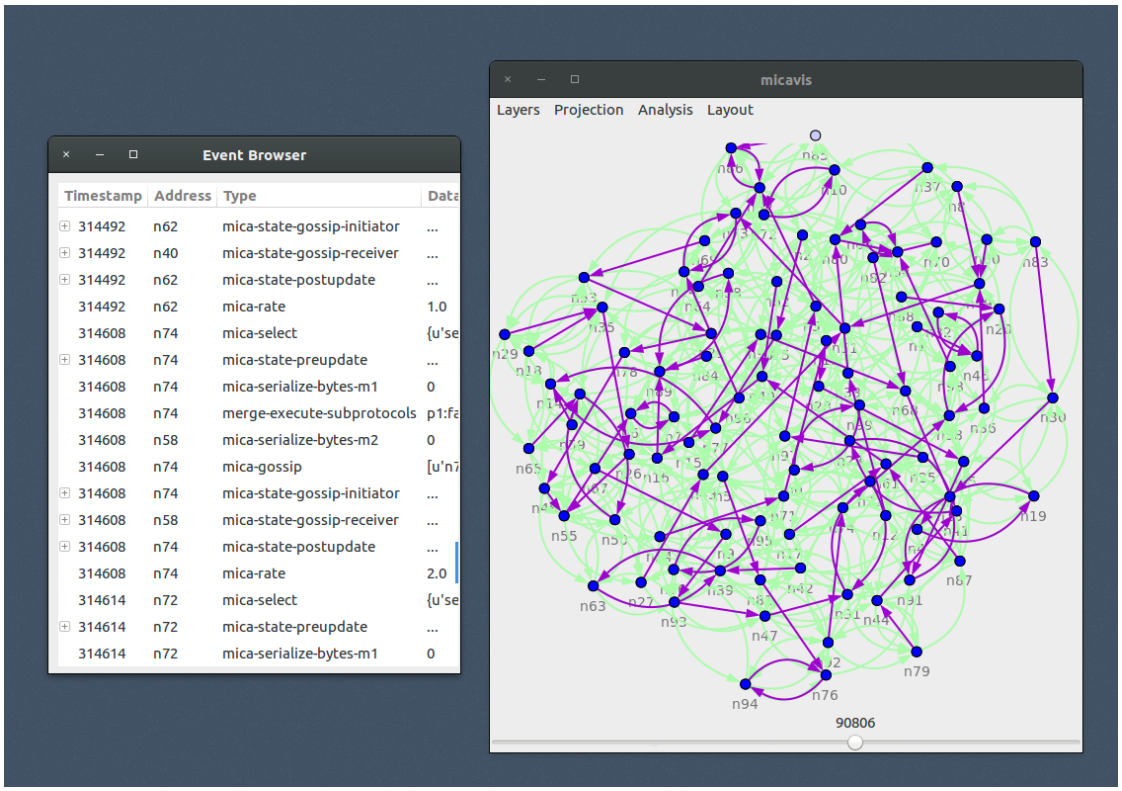

Figure 1: Micavis log visualization tool showing the gossip exchanges (purple) occurring at a moment in time. The current view graph is showing below in green.

The micavis visualizer is able to generate graphs of convergence, visually graph nodes according to their view at a point in time, and zoom in on protocol state before or after any kind of gossip event.

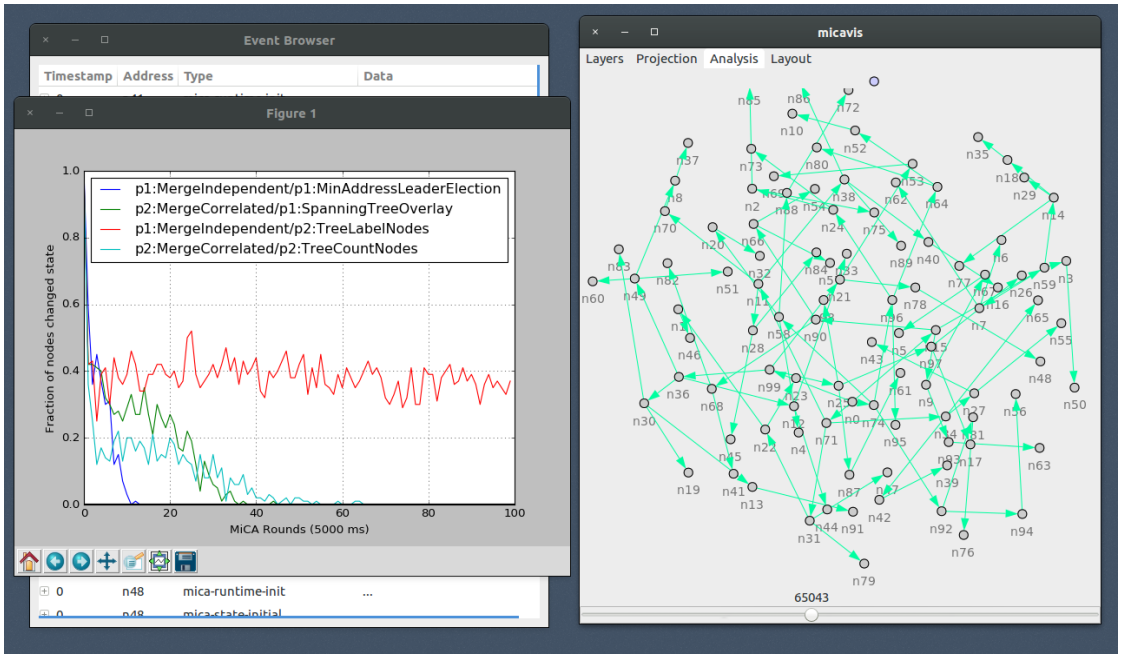

Figure 2: The Micavis log visualization tool plots convergence rates for subprotocols of a composite protocol.

CHAPTER 6

EVALUATION

In addition to the results presented in chapter 4, this chapter shares some additional results related to topology management, composition, and a phenomenon we call "dilation".

6.1 Topology Experiments

One feature that distinguishes MiCA's model of gossip is that a MiCA protocol defines how to derive view from node state, rather than keeping the view separate from other "payload" data. This makes it natural to build protocols that modify their own views. In practice, MiCA compound protocols with self-modifying views generally take the form of hierarchies of subprotocols, where each level consumes a view from the previous one, computes a new view, and exports it to the next level. We refer to these protocols as overlays.

In this section, we construct a compound protocol with several layers of topology overlays. The objective of the protocol is this: starting from an unstructured, connected communication graph, divide nodes into k groups of equal size. Then build a ring overlay for each group.

Peer sampling overlay Constructs an ever-changing, random topology, as described by Massoulié et al [47]. Nodes gossip about other known nodes in the extended network, retaining a constantly changing, fixed-size view that approximates random selection from the whole graph. The peer sampling overlay never stabilizes.

Distributed slicing algorithm Distributed slicing is the problem of dividing a group of nodes into equally-sized subgroups, without central coordination. We implement the Sliver[27] distributed slicing algorithm with MiCA. It gossips over the peer sampling overlay, computing a node's slice membership is by measuring where it sits relative to its peers in some predetermined sort order.

Filter overlay A filter overlay is simple MiCA primitive that imports another overlay (in this case, the peer sampler) and exports a filtered view that excludes some nodes. Here, we have each filter overlay node gossip about its peers' Sliver slice ID. The exported view for a node contains only the peers that occupy the same slice.

We use T-Man[32]'s topology construction to build a ring overlay for each group. T-Man is a simple, but powerful, gossip protocol: Every round, each node adds random peers to its view (sourced from the filter overlay). Then a node-specific ranking function is used to sort the view, and the top c peers are kept as the new view. To build a ring in this way, T-man uses $c = 2$ and requires each node to have a unique integer ID, where all N nodes participating in the ring are represented by N contiguous IDs. The ranking function computes the ring distance between two nodes' IDs. The T-Man ring becomes stable when all nodes know about their peers with ring IDs that immediately precede and supersede their own. Some observations about T-Man: First, the distance function needs to know N, the number of nodes in the network. Second, any gap in a node's view will break the ring. For example, suppose node id 3 tries to rank a view of $\{1, 2, 6\}$. The closest $c = 2$ nodes both precede 3 in the ring, causing the node to have two pointers in the id-descending direction and none ascending. As a result, we must ensure that every node knows the network size, and that

there are no gaps in the ring address space. These two conditions are difficult to achieve in the presence of churn. Fortunately, the self-stabilizing spanning tree protocol stack from earlier chapters can achieve these goals. To recap, the constituents of that stack are:

Leader election Using an intrinsic, totally ordered property (like node address), choose the minimum node as the leader.

Spanning tree overlay Using the chosen leader as the root of a spanning tree, each node gossips with its peers to find the peer closest to the root. That peer then becomes the node's parent in the tree.

Tree count nodes Gossip over the spanning tree, using the designated leader as the root and counting how many nodes exist in each subtree.

Label nodes Using subtree node counts, assign integer labels to each node, with the tree's root assigned 0 and the rest of the tree assigned ascending labels in pre-fix depth first search order.

T-Man nodes need to know the size of the network in order to compute the ring distance function. The *tree count nodes* component ensures that each node knows the size of its own subtree within the spanning tree, so the root node will know the size of the network. We add a *tree size broadcast* component that propagates this number among peers reachable via the filter overlay. The combination of the tree size broadcast and node labeling enables nodes to construct a T-Man ring. The final two protocol layers are:

Tree size broadcast The spanning tree root node disseminates its current tree size count.

T-Man Ring Nodes use the *tree size broadcast* and *label nodes* layers to identify peers with adjacent ring IDs.

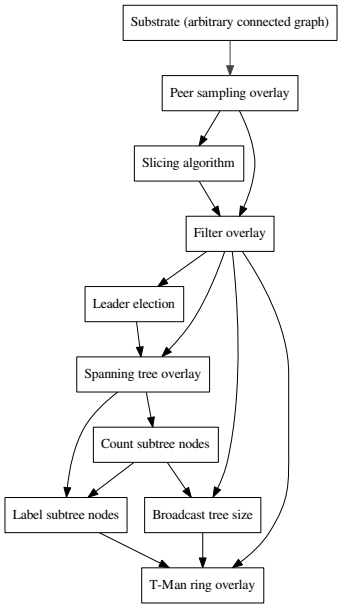

Figure 1: Dependencies between protocol layers for the topology demo stack.

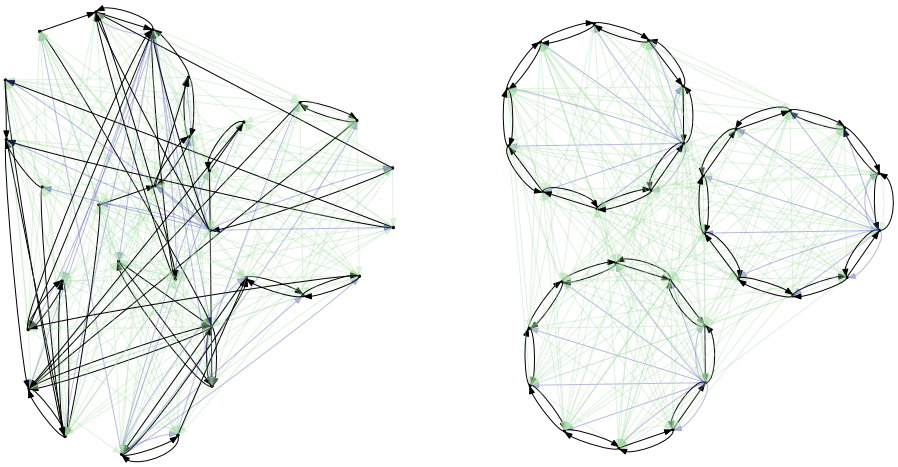

Figure 2: Overlays over 30 nodes before and after self-stabilization (30 and 100 rounds). T-Man ring overlay (black), spanning tree overlay (light blue), peer sampling overlay (light green). Not shown: Filter overlay, substrate graph.

6.1.1 Information Retention in Layered Gossip Protocols

A simple leader election protocol is used as a running example throughout this paper. In this example, every node has a unique descriptor. Nodes use a predetermined sort order over descriptors to agree that the least node is the leader. When nodes gossip, they tell each other about the least node they know of, updating their leader belief as necessary when they learn about a new least node. This simple protocol fails spectacularly when the leader leaves the system: other nodes continue to believe in the vanished leader because they have no way to forget.

This can be addressed by introducing information retention timestamps. Every piece of information shared in the system is given a timestamp by its originator. Other nodes discard or discount information with stale timestamps. In the case of the leader election example, when a node shares its own descriptor with a peer, the descriptor is timestamped. Because a node that is active in the system is constantly broadcasting its own descriptor, the timestamp attached to this descriptor with other peers stays fresh, as they keep the most recent timestamp they've seen for this information. However, if a node leaves the system, the timestamp attached to its descriptor as known by its peers will grow stale.

This strategy is used by Sliver and others to handle network churn. In the MiCA layered topology protocol, we use retention timestamps for all of the self-stabilizing constituent protocols: slicing (Sliver), filter overlay, leader election, spanning tree overlay, subtree node counting and labeling, tree size broadcast, and the T-Man ring overlay. Whenever a self-stabilizing protocol's input protocol is *not* self-stabilizing (e.g., our slicing algorithm's dependency the random peer sampling overlay), a fine balance exists between retention time and con-

vergence. This is especially true for what we will call *curator protocols*.

A curator protocol is one that must wait to receive a certain amount of consistent information before it converges; for example, an individual node of the T-Man Ring protocol reaches its converged state only after it has gossiped with the two peers that should be adjacent to it in the ring overlay. With the introduction of retention timestamps, it must have gossiped with these two peers within the maximum retention time, or it will forget them. This condition must be met by all of the nodes in the system for a curator protocol to converge. If retention time is too short, the protocol will fail to stabilize. On the other hand, if retention time is too long, stabilization will be delayed. In the case of our leader election protocol, a too-long retention window causes the system to retain an incorrect leader belief long after the old leader has left the system.

6.2 Experiments at Scale

This chapter details results from experiments at scale with our compositional gossip model. Recall that we have a choice of two binary operators to compose protocols: *correlated merge*, which gossips its operand protocols in tandem whenever possible while still respecting their gossip rate and view preferences, and *independent merge*, which multiplexes its operands.

Our demonstration composite protocol is built from five inter-dependent subprotocols, intended to represent common gossip use cases. The subprotocols are:

MinAddressLeaderElection A simple leader election protocol that leads all

nodes in the system to agree on a leader based on an intrinsic property such as address.

SpanningTreeOverlay Using the elected leader as root, this protocol constructs a spanning tree overlay that can be used by downstream protocols. It demonstrates overlay construction.

TreeCountNodes A typical aggregation protocol. Gossiping over the spanning tree overlay, each node aggregates the number of nodes in its subtree.

TreeLabelNodes A broadcast protocol; recursively assigns unique labels to all nodes using the spanning tree.

RandomWalkCoinCollector Nodes constantly exchange random walk tokens with their neighbors. This protocol never stabilizes, but a snapshot of system state indicatives well-connected nodes (where tokens accumulate) and poorly-connected nodes (where tokens are scarce).

The first four subprotocols listed above are all self-stabilizing. We expect them to eventually converge to a steady state in the absence of externally triggered system state changes. Because each depends on the state of the previous, they should converge in a cascade. We refer to these protocols collectively as the "self-stabilizing stack". The fifth protocol, `RandomWalkCoinCollector`, does not stabilize; it endlessly circulates random walk tokens.

The correlated and independent composition operators are associative with respect to correctness and convergence, but not performance. A correlated merge is more effective when used on operands that have significant overlap in their views.

Our experimental setup uses two composition structures, each run twice:

once exclusively with correlated merge, and once with independent, for a total of four variations. Of the five subprotocols, `MinAddressLeaderElection` and *Tree* gossip over a bootstrapped static overlay, and the remaining three (*Count*, *Label*, *Walk*) gossip over the constructed spanning tree. The first structure, shown in Figure 3, merges subprotocols that do not share a common gossip substrate. The second structure, Figure 4, merges those that do. We expect the second structure to perform better for correlated merges.

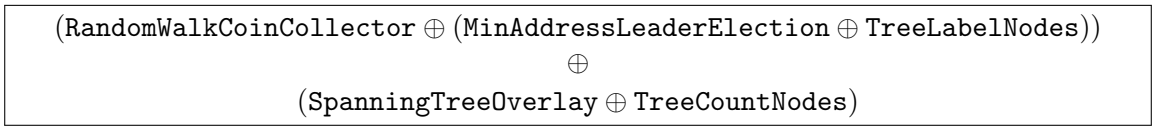

Figure 3: *Composition Structure 1*

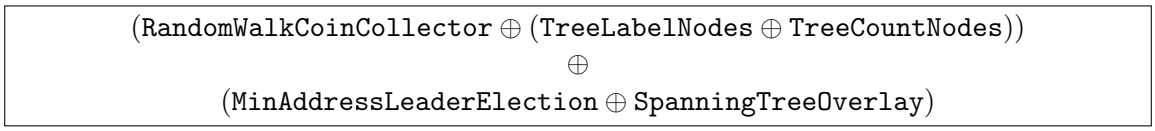

Figure 4: *Composition Structure 2*

We refer to the four combinations of merge operator and structure as *correlated-struc1*, *correlated-struc2*, *independent-struc1*, and *independent-struc2*.

Experimental setup

Experiments were run on on Amazon EC2, using 45 m3.medium instances, all within the same datacenter. Each EC2 virtual machine hosted ten virtual MiCA gossip nodes, for a total of 450 effective nodes.

Each of the four experimental variants was run for 300 rounds of gossip*.

*Recall that a MiCA gossip round is no more than a unit of time. Unlike synchronous formulations of gossip, there is no guarantee such as "all nodes gossip exactly once during each

This is long enough for the self-stabilizing stack to converge. Rounds were set to one second; a conservative interval that allowed gossip exchanges to complete without timeouts[†] due to serialization or compute time.

Each node writes a detailed log of its activities: gossip exchanges, merge operator decisions, state changes, etc. After each run, logs were aggregated and sorted by host timestamp. Although this is not generally a sound way to order events in a distributed system, our particular analyses are tolerant of a small amount of error. Clock skew among our fleet of recently-launched EC2 hosts is orders of magnitude smaller than our gossip round length, so misordered events should be rare.

Results: Convergence and Gossip Rates

Convergence of the self-stabilizing stack is measured by counting the frequency of state changes for subprotocol state. Although nodes continue to gossip after they have stabilized, their state has reached a fixed point. 5 shows the rate of state change for the five subprotocols for *correlated-struc1*. The cascading convergence of the four self-stabilizing protocols is visually obvious: First `MinAddressLeaderElection` stabilizes, then `SpanningTreeOverlay`, followed by `TreeCountNodes` and `TreeLabelNodes`. `RandomWalkCoinCollector` maintains a steady rate after its overlay `SpanningTreeOverlay` has stabilized.

Figure 6 shows the convergence of all four variants side by side. Only a small

round". A protocol that specifies its rate as 1.0 will gossip, on average, once per round.

[†]MiCA serializes gossip exchanges on each node, leaving it susceptible to gossip backups if gossip requests arrive too quickly.

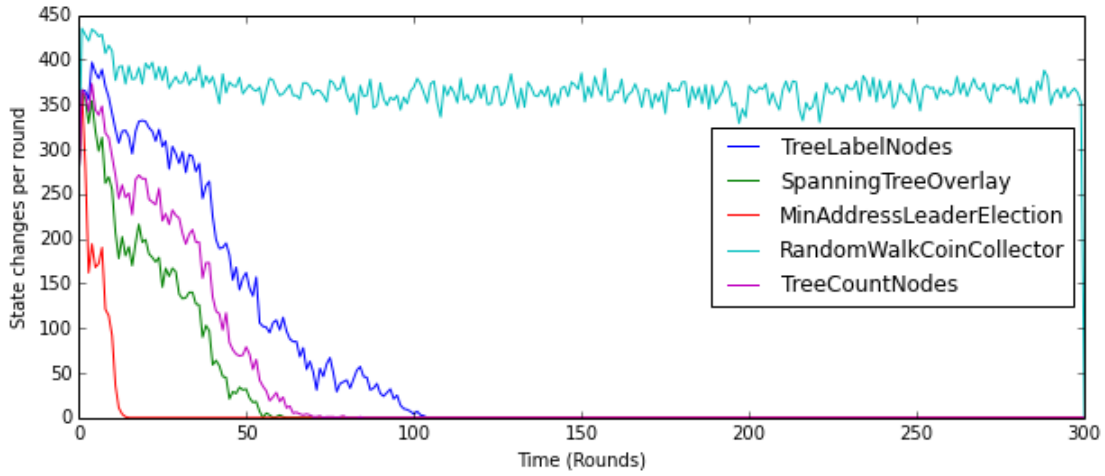

Figure 5: *correlated-struc1* subprotocol convergence tracks the number of nodes with changed subprotocol state in each round. When this reaches and stays at zero for a self-stabilizing protocol, the protocol has converged.

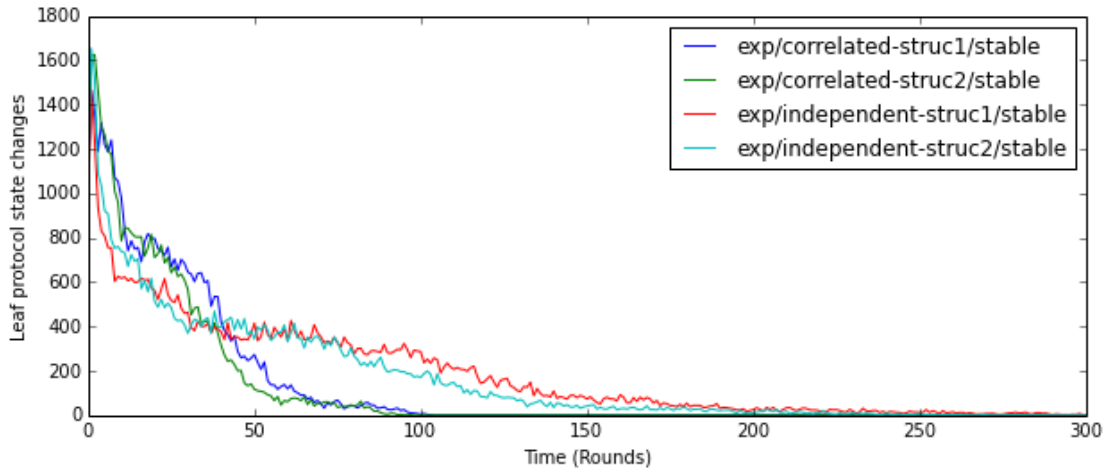

Figure 6: Self-stabilizing stack convergence comparison.

difference in convergence rate is evident, with correlated variants converging slightly slower than independent variants. This seems like a modest result until we see that the *actual* gossip rates (Figure 7)[‡] of the compound protocols differed by substantially.

[‡]Gossip rate plots are smoothed with LOWESS[12] unless otherwise noted, to make trends more easily discernible to readers. Figure 9 is an example of an un-smoothed rate graph.

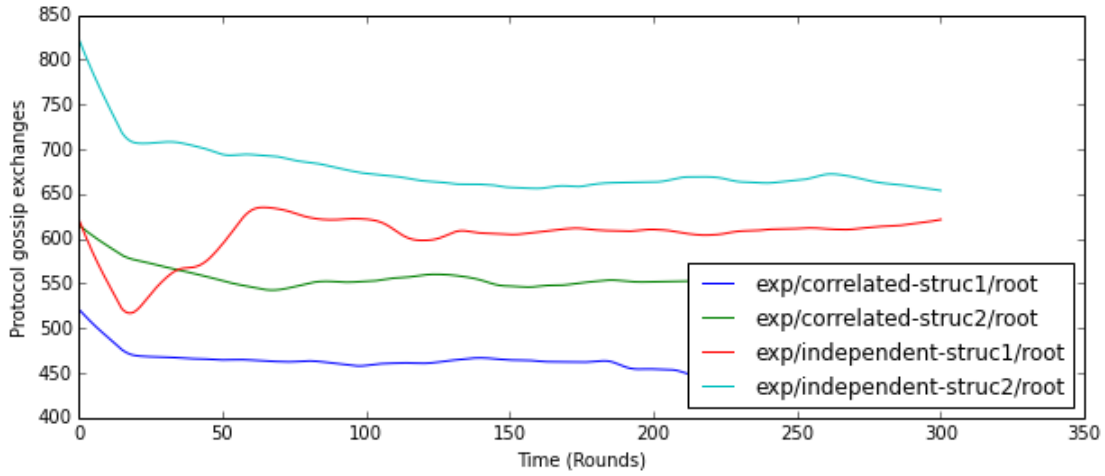

Figure 7: Gossip rate of the compound protocol using correlated vs. independent merge. The correlated merge operator uses significantly fewer messages than independent merge, but still converges more rapidly.

Here, *actual* gossip rate refers to the number of messages sent by the compound protocol, and *effective* rate is the number of subprotocols represented by a message. E.g., if a compound merge operator gossips two subprotocols simultaneously every round, this is an actual rate of one and an effective rate of two. Both *correlated-struc1* and *correlated-struc2* achieved the same convergence as their independent counterparts, but did so with nearly a 70% reduction in the number of messages used.

Figure 8 highlights the difference between actual and effective rates for correlated and independent merges. Comparison of effective gossip exchange rates for each variant, defined as the sum of gossip rates of subprotocols, confirms that correlated and independent runs had *nearly* identical effective rates. The difference is explained by a kind of joint failure scenario: The MiCA runtime occasionally drops a gossip exchange because it times out on the receiver queue.

With correlated gossip, dropping an exchange impacts more subprotocols than it does with an independent merge. Compounding the problem, under

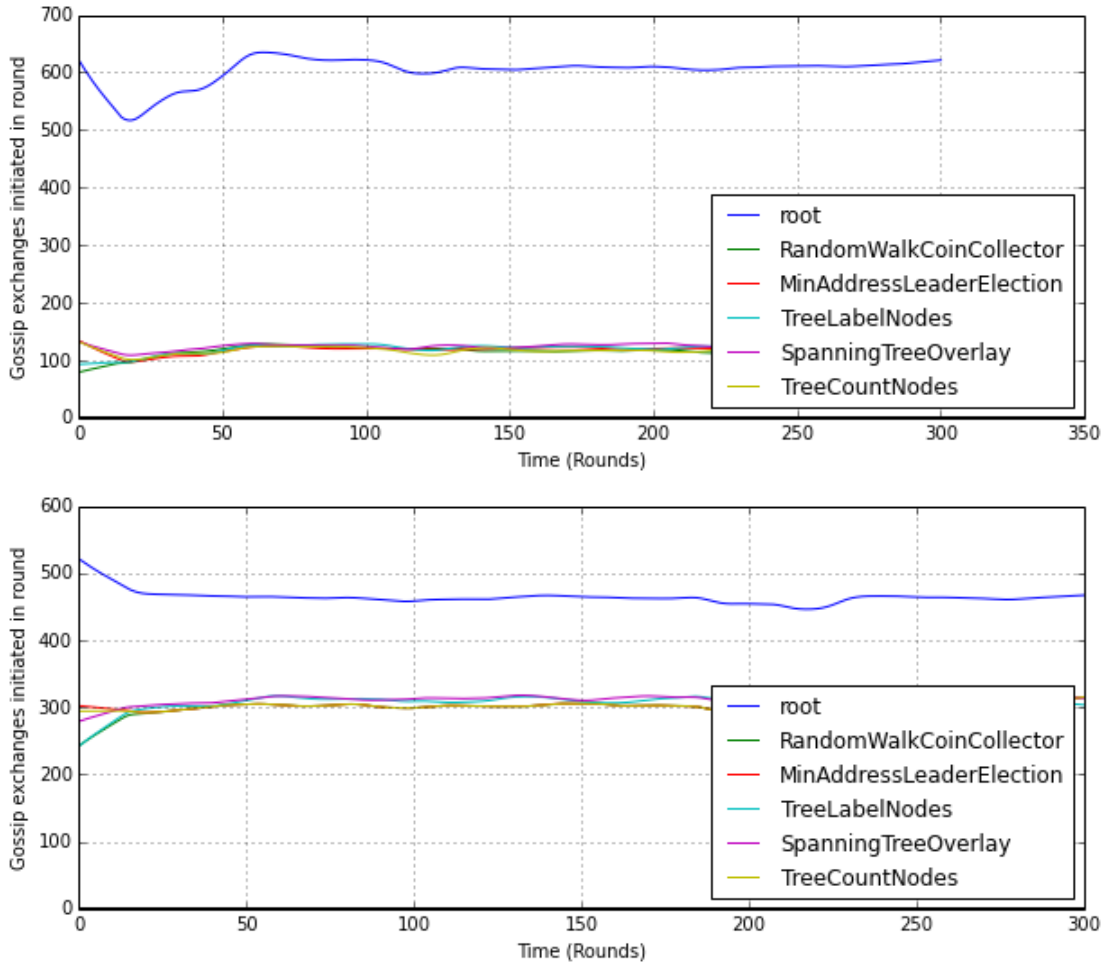

Figure 8: Difference between actual gossip rate ("root") and effective subprotocol rates for *independent-struc1* (top) and *correlated-struc1* (bottom). In the bottom graph, the three correlated tree-gossiping protocols are perfectly aligned by correlated gossip.

correlated merge, these events are more likely to happen due to higher variance in gossip rate; see Figure 9 and compare the spread between highs and lows for correlated versus independent trials. MiCA could compensate for this by attempting to gossip slightly faster than the nominal rate when it notices exchanges being dropped, although this is not implemented. Doing so would also come with a risk of creating feedback.

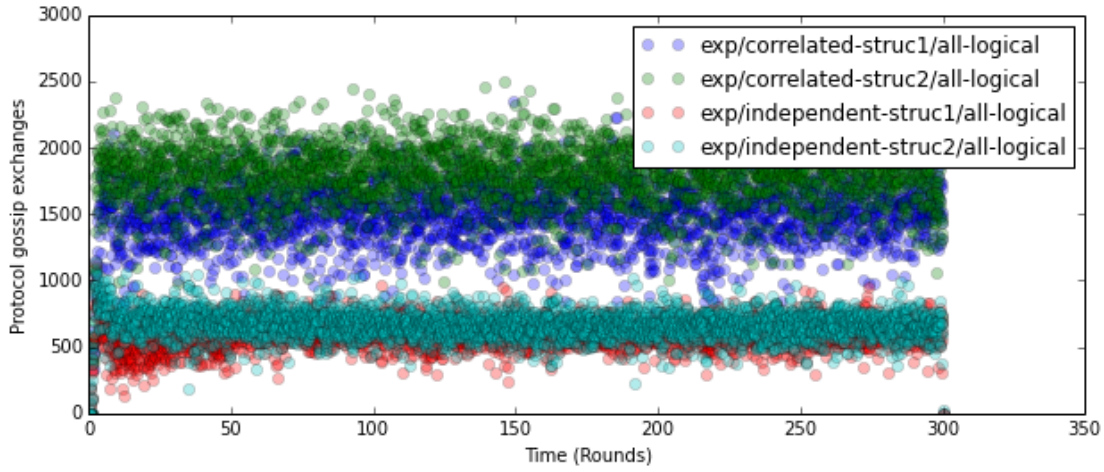

Figure 9: Effective gossip rates of all variants. Not smoothed. Greater variance is evident for correlated merges.

6.3 Dilation Experiments

The below graphs show the effect of varying degrees of dilation on two topological extremes, a ring and a complete graph. The sample protocol used was a basic PUSH/PULL find-minimum-value protocol. The degree of dilation (written "dilation-D") indicates the number of coin flips that must all be successful for the protocol to gossip when its *update* method is executed: "dilation-0" indicates no dilation, and "dilation-2" means that a node has only a 1/4 chance of doing anything when it gossips. The rate of dilated protocols is adjusted accordingly, to preserve the average rate of gossip.

Both experiments were run on a MiCA simulator with 1000 nodes, numbered 0 through 999. The find-min protocol has converged when every node has learned the minimal node number.

The gossip round length was set to a large value, 100 seconds, to prevent high-dilation (hence, high-rate) trials from experiencing backups waiting for

the protocol update function (which takes several milliseconds) to complete. No such backups occurred during the experiments; this is important because such backups would muddy the performance effects caused directly by dilation.

Convergence is measured by the rate of "change" events, which are generated whenever a node's minimum-value belief changes.

Dilation histograms measure the time between gossiping on a per-node basis; for example, if node $n1$ gossips at time 1.3 seconds and next at 11.3 seconds, then a 10-second interval data point is included in the histogram calculation.

Complete Graph Topology

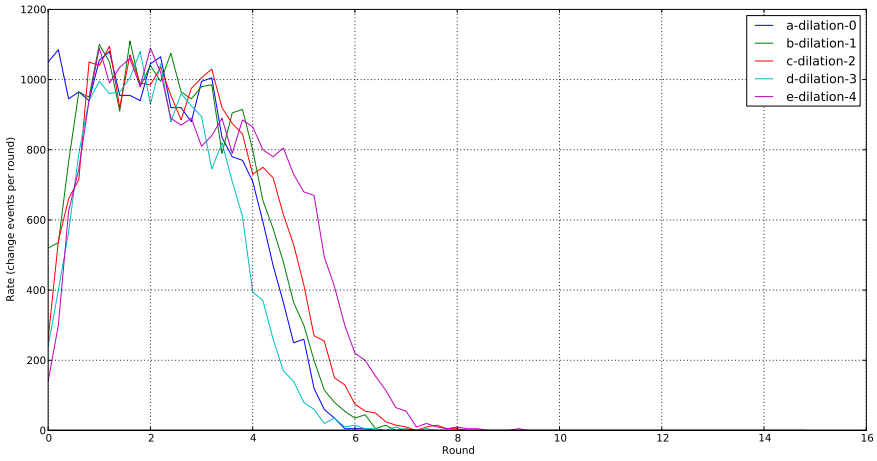

Figure 10: Convergence on a complete graph. The effect of dilation is minimal — in fact, dilation-3 converges before dilation-2 (although not faster than dilation-1 or dilation-0, but this is difficult to see), making us suspect that different random seeds could produce different convergence orderings, and that all of these convergence rates are essentially the same.

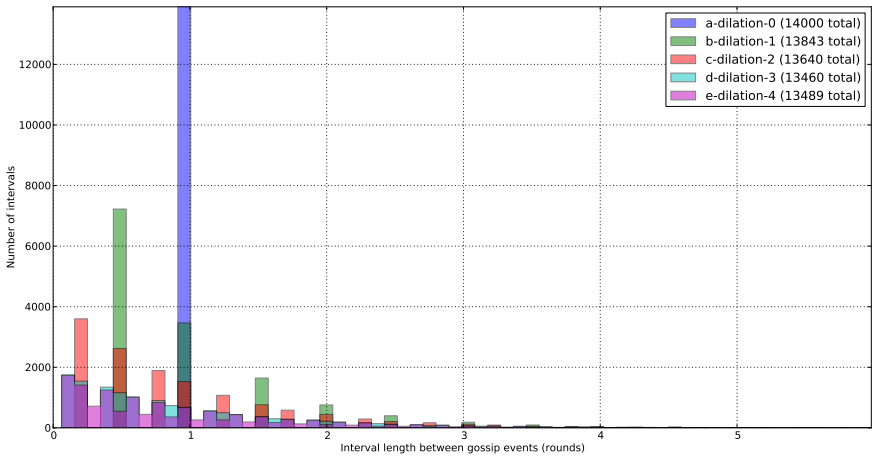

Figure 11: A histogram of the interval between successive gossips shows the degree of dilation the complete graph and confirms that the total number of gossip events is roughly unchanged by dilation.

Ring Topology

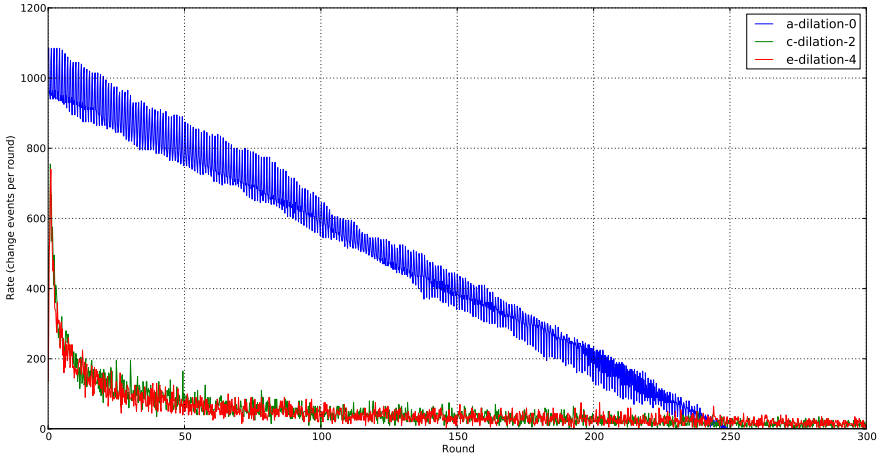

Figure 12: Convergence on a ring topology. The effect of dilation is dramatic, although there appears to be little difference between dilation-2 and dilation-4.

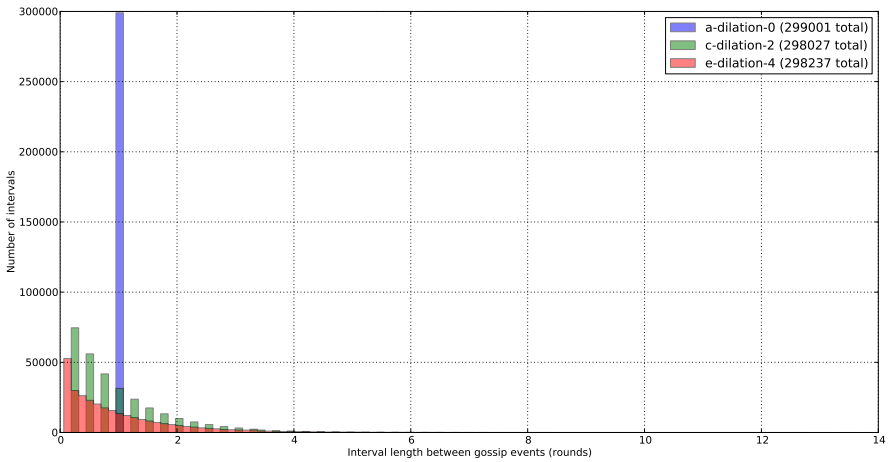

Figure 13: A histogram of the interval between successive gossips shows the degree of dilation the ring, and confirms that the total number of gossip events is roughly unchanged by dilation.

CHAPTER 7

CONCLUSION

The research represented in this dissertation began with a question: Can we build a programming language for gossip? Gossip is a powerful mechanism, but it is built from simple elements: Periodic communication and stochastic peer selection. The first step was to decide on the right gossip model.

Early analysis of the spread of epidemics used what we would now call uniform gossip over a complete graph. That is, peers chose their communication partners uniformly at random from the entire population. With this simple model, it's trivial to show that epidemics spread in logarithmic time with respect to the number of nodes [25]. Based on this, one model of gossip might look like this: A gossip system consists of a set of nodes and a function that defines what happens when two peers gossip. However, if we tried to implement this model in a practical system, we would quickly discover a limitation: Real systems need to accommodate node churn. If all pairs of nodes are potential gossip partners, this means that every change in system membership needs to be broadcast to all nodes. When a new node joins, it must be bootstrapped with a complete membership set. This is fine for small systems, but does not transfer well to internet-scale, in which the burden of maintaining correct membership will overwhelm other activity of the system. This can be avoided by gossiping over sparser topologies, where every node needs to keep only a subset of group membership in its local state. Thus, if we want our gossip model to admit large-scale systems, it will need to handle diverse topologies. Further, alternate topologies are required for some gossip algorithms [39, 62], and others are intended for arbitrary or even dynamic topologies [33]. Convergence rates

in some cases may even be improved over those of uniform gossip [38].

Two subsequent design decisions about the gossip model heavily influenced the direction that MiCA would take. First, that a node's view (e.g., set of peers to choose from) should not be separate from the rest of its protocol state. MiCA protocols are asked at each round what their view is, as a function of their state. They can use this to compute any view they wish, including dynamic views that change over time. Second, that views would be represented not as a set of peer addresses, but as a discrete probability distribution. This allows a protocol to weight some peers more heavily than others. Not only is this useful for concepts like spatial gossip, but it would turn out to be crucial to our goal of protocol composition— two overlapping but non-identical views can be combined into one, as described in chapter 3, with adjusted probability weights preserving the gossip frequency preferences of constituent protocols.

Consideration of the language itself settled on a DSL embedded into Java. The gossip model could *mostly* be represented using ordinary Java classes and semantics, but the `update` function member of a gossip protocol was designed to operate on system state spanning two nodes. The DSL took the form of an annotation to the update function that indicated its special semantics to a byte-code post-processor, which would weave in network code to give the effect of a distributed program. Other annotations provided syntactic sugar for common features, for example, uniform gossip on a Java collection could be specified through an annotation instead of forcing the programmer to write a boilerplate `view` function.

The choice of Java as a base for this implementation was, in retrospect, not ideal. The implementation has always had to have caveats attached; for exam-

ple, "Don't access static class members from inside a gossip `update` function." Bytecode transformation proved to be somewhat awkward, and maybe too low-level to achieve best results. JVM type erasure caused headaches. `update` functions were implemented as overrides of a base class method, and as such had to perform some unsightly type-casting to cast arguments as the correct classes. Erlang might have been a better choice.

Early versions of this research focused on the compiler problem of how to distribute the pair-wise `update` function. Although the gossip model remained unchanged into later versions, that focus gradually gave way to more concentration on composition and the gossip model itself. In retrospect, I feel these are greater contributions than clever program partitioning. a No other work in the literature survey has treated a gossip protocol as an abstract entity that can be used as a building block to build more sophisticated gossip systems.

When thinking of distributed programming applied to gossip, it's natural to think about composition and encapsulation; gossip algorithms serve precise purposes, and if you want to use more than one in an application, it's logical to think you should be able to instantiate one gossip runtime and then run both gossip protocols side by side. One success of MiCA is that it really does allow the programmer to use gossip protocols as building blocks to assemble more complex systems. For example, gossip-based group membership protocols [16, 63] typically need to implement failure detection. MiCA makes it natural to implement failure detection and group membership separately, in such a way that the group membership protocol can instantiate and use failure detection.

The contributions of this body of research are as follows:

A novel model for gossip. Capable of admitting most, possibly all, gossip algorithms from the literature. Functions on local node state to determine rate, pair-wise update, and peer selection; view represented as a discrete probability distribution offers new possibilities.

Pair-wise distributed programming model. A novel system model for distributed programming that captures precisely the nature of gossip and allows new gossip protocols to be written without any explicit network communication, which is handled automatically by the post-processor runtime.

Proof of concept implementation and simulator. MiCA experiments were conducted with a Java runtime implementation capable of running MiCA protocols on a local simulator and on a real network of machines. In simulation, time is simulated as well, leading to results of experiments much faster than realtime.

Library of gossip components. MiCA's prototype includes Java interfaces and implementations of rumor mongering, topological overlay construction, aggregation, anti-entropy, and random peer selection, among others. Examples demonstrate the easy of building complex protocols from these basic building blocks.

MiCA represents a new point in the design space of distributed programming. Nothing similar precedes it; it is the first distributed programming attempt specifically aimed at gossip systems, and the first to target a pair-wise programming model. Other gossip frameworks exist, but with more conventional programming models.

Future work and open questions

MiCA's examples are, admittedly, contrived. Implementing a realistic system could bring strong validation of our approach, either by building a system from scratch using MiCA, or by substituting MiCA-based gossip for the gossip functionality of an existing system. For example, Cassandra [42] is widely used, open source, uses gossip for anti-entropy, and happens to be written in Java. These facts make it an excellent candidate. After duplicating Cassandra's existing anti-entropy functionality, MiCA could be used to enhance it in a number of ways. Replication messages between replicas hosted on the same node could be bundled together for network savings. Gossip rates and view probability weights could be adjusted to prioritize gossip to replicas that lag behind.

MiCA could also be explored as a general platform for gossip research. One advantage it offers is that it can implement platform effects as adapter classes, such as Gossip Objects [65]'s speculative message bundling, in addition to conventional gossip protocols.

Further research could enhance MiCA's code partitioning for its `update` functions, as well. The static analysis currently used is naive, and does not match well with MiCA's composition. Improved code partitioning might choose to break each gossip exchange into variable numbers of messages sent between a gossip pair, instead of only one send and one receive message, or simply implement a MiCA runtime that runs an `update` function by transparently proxying one peer.

Gossip composition has many open questions remaining. We have proposed two composition operators: a correlated merge that saves space, but increases

the change of correlated failure; and an independent merge that approximates running two independent, concurrent protocols. Both of these operators are binary: they take two protocols and combine them into one. Because of this, any large compound protocol is achieved through an entire binary tree of composition operators. Although the correctness of the operators is not affected by the order of composition, does. In other words, the operators do not commute with respect to performance. Two merged protocols will see a higher degree of network savings if their respective views overlap substantially. Implementation of an $n - ary$ composition operator could achieve greater efficiency that binary operators cannot, because truly computing optimal overlap sets for gossip bundling is only possible as a global problem (and, to be fair, arriving at an optimal solution is probably NP-complete, so approximations would be more practical for systems with large numbers of protocols and nodes).

A fourth open research idea is to further explore the idea of gossip adapters and transformers. These are MiCA gossip protocols that wrap another protocol and subtly change its behavior. For example, one currently implemented transformer type is an EpochDelimiter. This takes a protocol factory as its input. The transformer runs k concurrent instances of the input protocol, periodically deleting the oldest and instantiating a new one. Normal operations of the target protocol are served by the oldest copy. When consumers of the protocol state interact with it, they see information from a protocol that has only been running for, at most, a known, finite length of time. This can be used to implement forgetfulness, e.g., of self-stabilizing protocols that do not have a built-in mechanism for forgetting. The Epoch interface could also be used to implement group membership epochs as called for by [7]. In the future, these transformers could include mechanisms to add new properties to existing protocols, such as

Byzantine Fault Tolerance as is added in [34]; or speculative rumor delivery, as implemented by [66]; or adding checksums to gossiped messages, such as could have prevented [57].

Finally, one last idea for future work would be to extend our MiCA gossip model with a concept of "amnesia". Gossip protocols learn about information from the wider system in two ways. The first is *affirmation*, where a peer communicates a belief; for example, a heartbeat "ping" that indicates the peer is still alive. The second is *silence*, where something about the system can be inferred because nothing has been received; for example, if a heartbeat is not received after a grace period, then it may be inferred that the missing peer has either failed or a network partition has occurred. Absence is often used to expire stale information. In a group membership protocol, nodes might rely on affirmation to learn about new group members, but rely on silence to purge stale members. A common pattern emerged while writing example protocols for MiCA: timestamps must be attached to every piece of information that might become stale according to the protocol logic. Policies for marking something stale must be defined, and may be parametric with the size of the network, the rate of incoming updates, or other factors. Convergence and correctness may hinge on having reasonable expiration policy. A system that never forgets risks being always out of date; one that forgets too quickly will not be correct. As an example, self-stabilizing protocols will fail to stabilize if they expire data too quickly. With MiCA, this "amnesia sensitivity" is compounded by composition. Multiple layers of self-stabilizing protocols, composed together, are extremely sensitive to having reasonable expiration policies. In practice, for MiCA's examples, we tuned these parameters manually until the compound protocols worked correctly. This is not a satisfying approach, and the first step to a better solution is

to step back and consider adding amnesia as a core concept to MiCA's model of gossip.

BIBLIOGRAPHY

[1] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems*, pages 223–236, April 2010.

[2] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab, April 2003.

[3] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 385–394, New York, NY, USA, 2008. ACM.

[4] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *Symposium on Principles of Distributed Computing*, pages 385–394, August 2008.

[5] K. P. Birman, R. van Renesse, and W. Vogels. Spinglass: secure and scalable communication tools for mission-critical computing. In *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 85–99 vol.2, 2001.

[6] Kenneth P. Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert Van Renesse, Ohad Rodeh, and Werner Vogels. The horus and ensemble projects: Accomplishments and limitations. Technical report, Ithaca, NY, USA, 1999.

[7] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *Transactions on Computing Systems*, 17(2):41–88, 1999.

[8] Bonjour. Available at `http://www.apple.com/support/bonjour/`.

[9] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.

[10] Andrei Z. Broder, Alan M. Frieze, and Eli Upfal. Static and dynamic path selection on expander graphs: A random walk approach. *Random Structures and Algorithms*, 14(1):87–109, August 1999.

[11] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, October 2007.

[12] W.S. Cleveland. Lowess: A program for smoothing scatterplots by robust locally weighted regression. *The American Statistician*, 35:54, 1981.

[13] Jon Currey. Making gossip more robust with lifeguard.

[14] Pierre-Évariste Dagand, Dejan Kostić, and Viktor Kuncak. Opis: Reliable distributed systems in OCaml. In *International Workshop on Types in Language Design and Implementation*, pages 65–78, January 2009.

[15] D. J. DALEY and D. G. KENDALL. Epidemics and rumours. *Nature*, 204(4963):1118–1118, Dec 1964.

[16] Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 303–312, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[18] 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 *Symposium on Operating Systems Principles*, pages 205–220, October 2007.

[19] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[20] Danny Dolev and Ezra N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *International Conference on Stabilization, Safety, and Security of Distributed Systems*, pages 234–252, November 2007.

[21] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[22] Patrick Eugster. Uniform proxies for java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 139–152, October 2006.

[23] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, pages 70–75, New York, NY, USA, 1982. ACM.

[24] Benoit Garbinato and Rachid Guerraoui. Flexible protocol composition in Bast. In *International Conference on Distributed Computing Systems*, pages 22–29, May 1998.

[25] WILLIAM GOFFMAN and VAUN A. NEWILL. Generalization of epidemic theory: An application to the transmission of ideas. *Nature*, 204(4955):225–228, Oct 1964.

[26] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical report, Santa Cruz, CA, USA, 1992.

[27] Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, and Robbert van Renesse. A fast distributed slicing algorithm. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Toronto, Canada*, pages 427–427, August 2008. Brief announcement.

[28] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM.

[29] Maya Haridasan and Robbert van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *IPTPS 2008: Proceedings of the 7th International Workshop on Peer-to-Peer Systems*, 2008.

[30] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. *Gossip-based aggregation in large dynamic networks*, volume 23(3) of *ACM Transactions on Computer Systems*, pages 219–252. August 2005.

[31] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based ag-

gregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, August 2005.

[32] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, January 2009.

[33] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25, August 2007.

[34] Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM Trans. Comput. Syst.*, 33(2):5:1–5:32, May 2015.

[35] JXTA The Language and Platform Independent Protocol for P2P Networking. Available at https://jxta.kenai.com.

[36] Brian Kantor and Phil Lapsley. RFC 977: Network News Transfer Protocol. RFC 977, RFC Editor, February 1986.

[37] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 482–491, Oct 2003.

[38] D. Kempe and J. Kleinberg. Protocols and impossibility results for gossip-based communication mechanisms. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 471–480, 2002.

[39] David Kempe, Jon Kleinberg, and Alan Demers. Spatial gossip and resource location protocols. pages 163–172. ACM Press, 2001.

[40] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[41] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

[42] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[43] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[44] Shen Lin, Francois Taiani, and Gordon S. Blair. Facilitating gossip programming with the gossipkit framework. In *DAIS*, pages 238–252, 2008.

[45] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Symposium on Operating Systems Principles*, pages 75–90, October 2005.

[46] Lucie Lozinski. Uber engineering's ringpop, Feb 2016.

[47] Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer counting and sampling in overlay networks: Random walk methods. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 123–132, New York, NY, USA, 2006. ACM.

[48] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *International Conference on Distributed Computing Systems*, pages 707–710, April 2001.

[49] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 99–100, Sept 2009.

[50] Lehel Nyers and Mrk Jelasity. A comparative study of spanning tree and gossip protocols for aggregation. *Concurrency and Computation: Practice and Experience*, 27(16):4091–4106, 2015. cpe.3549.

[51] Lonnie Princehouse and Ken Birman. Code-partitioning gossip. *Operating Systems Review*, 43:40–44, January 2010.

[52] Lonnie Princehouse, Rakesh Chenchu, Zhefu Jiang, Kenneth P. Birman, Nate Foster, and Robert Soulé. *MiCA: A Compositional Architecture for Gossip Protocols*, pages 644–669. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[53] Adolfo Rodriguez, Charles Edwin Killian, Sooraj Bhat, Dejan Kostic, and

Amin Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Symposium on Networked Systems Design and Implementation*, pages 267–280, March 2004.

[54] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[55] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[56] Rajagopal Subramaniyan, Pirabhu Raman, Alan D. George, Matthew A. Radlinski, and Matthew A. Radlinski. GEMS: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. *Cluster Computing*, 9(1):101–120, January 2006.

[57] Amazon S3 Team. Amazon s3 availability event: July 20, 2008.

[58] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[59] Tibco message bus. http://www.tibco.com/products/automation/messaging/default.jsp.

[60] Norbert Tölgyesi and Márk Jelasity. Adaptive peer sampling with newscast. In *European Conference on Parallel Computing*, pages 523–534, August 2009.

[61] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, November 1999.

[62] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[63] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-based

failure detection service. In *International Middleware Conference*, pages 55–70, September 1998.

[64] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, London, UK, UK, 1998. Springer-Verlag.

[65] Ymir Vigfusson, Ken Birman, Qi Huang, and Deepak Nataraj. Go: Platform support for gossip application. In *IEEE P2P*, pages 222–231, 2009.

[66] Ymir Vigfusson, Ken Birman, Qi Huang, and Deepak P. Nataraj. Optimizing information flow in the gossip objects platform. *Operating Systems Review*, 44(2):71–76, December 2010.

[67] Werner Vogels. Eventually consistent, Dec 2007.

[68] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[69] WebSphere MQ. http://www-03.ibm.com/software/products/en/wmq/.

[70] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society, March 1981.

[71] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *International Conference on Computer Communications*, pages 319–328, April 2001.

[72] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[73] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning, 2001.