MiCA: A Compositional Architecture for Gossip Protocols

Lonnie Princehouse[1], Rakesh Chenchu[1], Zhefu Jiang[1], Kenneth P. Birman[1], Nate Foster[1], and Robert Soulé[2]

[1] Cornell University
{lonnie,rr548,zj46,ken,jnfoster}@cs.cornell.edu
[2] University of Lugano
robert.soule@usi.ch

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.

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.

Keywords: Gossip protocols, fault tolerance, composition, distributed systems, program partitioning, Java.

1 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 [13,6] or any-cast [15,3] 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 [26,22,27], network overlay management [14], and even distributed storage systems [26,8,20,5].

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 characteristics 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 [2]. 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: § 2 and § 3 motivate MiCA's design using intuitive examples and experimental results from a simple simulation; § 4 describes operators for composing protocols and discusses correctness; § 5 discusses state management and an optimization; § 6 describes the MiCA prototype; § 7 presents an evaluation; § 8 discusses related work; and § 9 concludes.

2 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 [14]. 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 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 [24].

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.

2.1 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

```
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

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 [9].

3 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

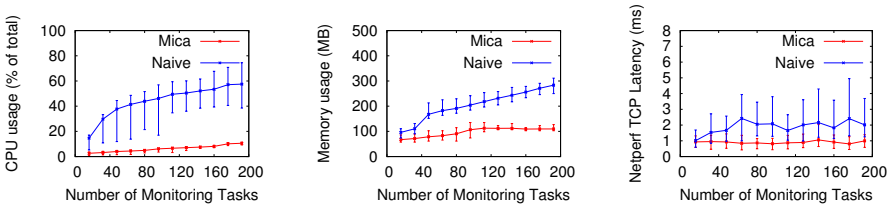

Fig. 2. The average CPU, memory, and network utilization when running an increasing number of monitoring tasks with both naïve composition and MiCA.

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 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.

4 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

		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[30], TIBCO[23]).
	Combined	An application can have many shared subsystems, but each communicates independently (e.g, JXTA[15], Bast[13]).	Composition reduces the overhead of executing multiple monitoring applications simultaneously (e.g., MiCA).

Table 1. Forms of gossip protocol composition.

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 1 presents an overview of various approaches for protocol composition:

- *Isolated state, isolated communication*: This is the naïve multiplexing approach discussed in § 3, 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 [23], and message-storage middleware, such as IBM WebSphere MQ [30]. 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 [7] runs in its own thread and communicates independently, but the overall system state is shared. Examples of this kind of system include JXTA [15] and Bast [13].
- *Shared state, shared communication*: This new approach combines the advantages 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 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.

4.1 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 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.

4.2 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` exchanges, using a boolean `g1Next` to keep track of the next sub-protocol to execute. For reasons

```
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.

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.

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

```
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.

`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 address `a`. 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.

```
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();
    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.

– Finally, suppose that `g1` gossips randomly with odd nodes, and `g2` gossips randomly with all nodes. The combined `view` method returns a distribution 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

```
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.

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. [17] 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. [4], 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 previous 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 ex-

tremely 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 protocol 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 [10]. 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.

5 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

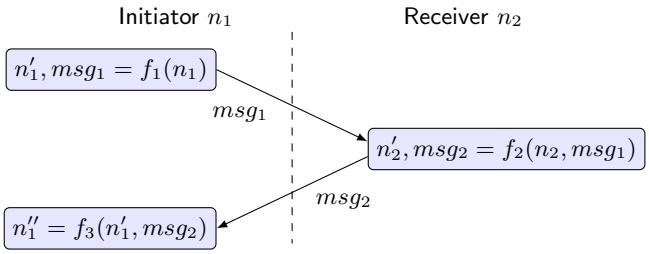

Fig. 7. Execution of a gossip exchange with the explicit messages used by the low-level target of the MiCA compiler. Provided the synthesized functions f_1, f_2, f_3 are correct, the final states of both nodes are guaranteed to be the same as if `update` had executed locally: $(n_1'', n_2') = \texttt{update}(n_1,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

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.

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

MiCA uses the Soot analysis framework [25] 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 [12]. 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.

7 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.

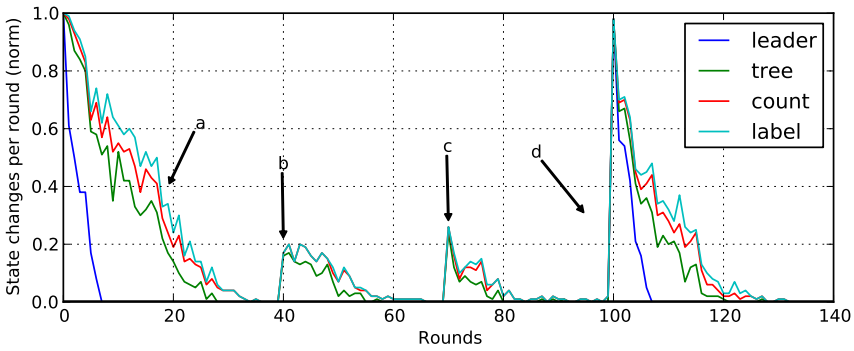

Fig. 8. Convergence of all four layers. Arrows indicate (a) Convergence from arbitrary starting state; (b) a transient fault: 10% of nodes crash; (c) failed nodes recover; (d) a large artificial disruption of the bottom layer's state. Note that the leader election layer was not affected by the transient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

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.

7.1 Layered Protocol

The first case study is based on a four-layer composite protocol originally proposed by Dolev [11]. 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**.

7.2 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 completely 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 mean that 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 proto-

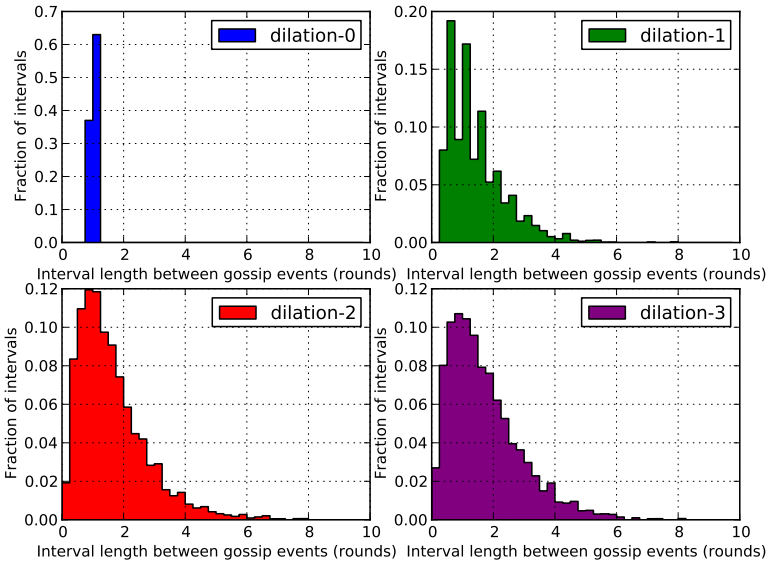

Fig. 9. Effect of dilation for an anti-entry protocol on intervals between gossip exchanges. The labels indicate the degree of dilation: `d0` is no dilation, `d2` is two nested operators, etc.

cols 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.

8 Related Work

Work related to MiCA falls into several general categories: gossip-specific frameworks (Opis [6], Gossip Objects [28]); object-oriented distributed system libraries (Bast [13], Jini [29]); compositional network transport protocol systems (Appia [18], Cactus [31]); and languages and abstractions for distributed programming (P2 [16], MACEDON [21], 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 [19]. 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* [6] 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 ex-

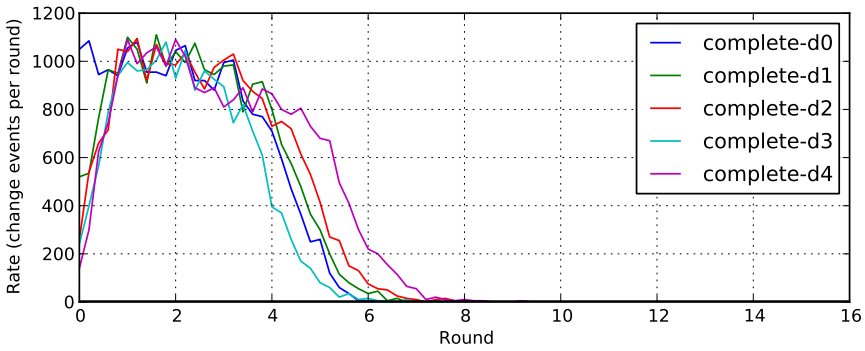

Fig. 10. Effect of dilation for an anti-entropy protocol in a complete topology. The labels indicate the degree of dilation: `d0` is no dilation, `d2` is two nested operators, etc.

ternal 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 [28] 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 works do not share MiCA's gossip-centric world view, but have a common philosophy for protocol composition. *Bast* [13] is an object-oriented library of distributed system components, with goals of modular protocol composition and code reuse. Bast's Java implementation is similar to MiCA's in representing protocol classes and object-oriented composition mechanisms such as inheritance to compose protocol layers, each providing different functionality. MiCA and Bast differ in scope. Bast is a general-purpose distributed systems toolkit, whereas MiCA is a domain-specific framework for gossip systems. MiCA's novelty lies in its gossip-specific abstractions and execution; what makes MiCA interesting for gossip protocols makes it unsuitable for general distributed systems.

Apache River [29] (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 [31] and Appia [18] 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 [16] 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 [21] 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.

9 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 in a manner that preserves their semantics while 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.

Acknowledgements. This work was supported by grants from the National Science Foundation, DARPA, and ARPA-e, and a Sloan Research Fellowship.

References

1. P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *European Conference on Computer Systems*, pages 223–236, Apr. 2010.
2. M. Ben-Or, D. Dolev, and E. N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *Symposium on Principles of Distributed Computing*, pages 385–394, Aug. 2008.
3. Bonjour. Available at http://www.apple.com/support/bonjour/.
4. A. Z. Broder, A. M. Frieze, and E. Upfal. Static and dynamic path selection on expander graphs: A random walk approach. *Random Structures and Algorithms*, 14(1):87–109, Aug. 1999.
5. Apache Cassandra. Available at http://cassandra.apache.org.
6. P.-E. Dagand, D. Kostić, and V. Kuncak. Opis: Reliable distributed systems in OCaml. In *International Workshop on Types in Language Design and Implementation*, pages 65–78, Jan. 2009.
7. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.
8. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.
9. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Symposium on Principles of Distributed Computing*, pages 1–12, Aug. 1987.
10. D. Dolev and E. 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, Nov. 2007.
11. S. Dolev. *Self-Stabilization.* MIT Press, 2000.
12. P. Eugster. Uniform proxies for java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 139–152, Oct. 2006.
13. B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *International Conference on Distributed Computing Systems*, pages 22–29, May 1998.
14. M. Jelasity, A. Montresor, and Ö. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, Jan. 2009.
15. JXTA The Language and Platform Independent Protocol for P2P Networking. Available at https://jxta.kenai.com.

16. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Symposium on Operating Systems Principles*, pages 75–90, Oct. 2005.
17. L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Symposium on Principles of Distributed Computing*, pages 123–132, Aug. 2006.
18. H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *International Conference on Distributed Computing Systems*, pages 707–710, Apr. 2001.
19. L. Princehouse and K. Birman. Code-partitioning gossip. *Operating Systems Review*, 43:40–44, January 2010.
20. Riak. Available at http://basho.com/riak/.
21. A. Rodriguez, C. E. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Symposium on Networked Systems Design and Implementation*, pages 267–280, Mar. 2004.
22. R. Subramaniyan, P. Raman, A. D. George, M. A. Radlinski, and M. A. Radlinski. GEMS: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. *Cluster Computing*, 9(1):101–120, Jan. 2006.
23. Tibco message bus. http://www.tibco.com/products/automation/messaging/default.jsp.
24. N. Tölgyesi and M. Jelasity. Adaptive peer sampling with newscast. In *European Conference on Parallel Computing*, pages 523–534, Aug. 2009.
25. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, Nov. 1999.
26. R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *Transactions on Computing Systems*, 21(2):164–206, Feb. 2003.
27. R. van Renesse, Y. Minsky, and M. Hayden. A gossip-based failure detection service. In *International Middleware Conference*, pages 55–70, Sept. 1998.
28. Y. Vigfusson, K. Birman, Q. Huang, and D. P. Nataraj. Optimizing information flow in the gossip objects platform. *Operating Systems Review*, 44(2):71–76, Dec. 2010.
29. J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.
30. WebSphere MQ. http://www-03.ibm.com/software/products/en/wmq/.
31. G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *International Conference on Computer Communications*, pages 319–328, Apr. 2001.

A Artifact Description

Authors of the artifact. Lonnie Princehouse

Summary. The artifact is a prototype implementation of the MiCA gossip framework. It includes the runtime and libraries used to develop and experiment with MiCA. It also includes implementations of the protocol composition operators and examples given in the paper. The implementation is able to run MiCA protocols on a real network or on a simulated network with a variety of network topologies.

Content. The artifact package includes:

- A runnable jar with bundled dependencies
- Source code
- Documentation and examples

Start with index.html

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of our code is available at: https://github.com/mica-gossip/mica.

Tested platforms. The artifact requires Java 6 or greater to run the compiled jar, or a recent version of Eclipse to use the pre-built Eclipse workspace.

License. BSD 3-Clause License (http://opensource.org/licenses/BSD-3-Clause)

MD5 sum of the artifact. 68988b8c4623a529366a01d89113ec66

Size of the artifact. 26521350