Compositional Gossip Protocols for Infrastructure Management

Lonnie Princehouse
Cornell University
lonnie@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Abstract

The developers of today's cloud computing systems are expected not only to create applications that will work well at scale, but also to create management infrastructures that will monitor run-time conditions and intervene to adapt as conditions evolve. Monitoring and management tasks are generally not performance intensive, but robustness is paramount: when a large application becomes unstable, the management infrastructure must remain robust, fault-tolerant, and predictable, riding out the disruption and quickly restoring stability. MiCA is a new system for creating scalable application and network management services based on gossip that are highly resistant to disruptions and efficient in their use of system resources. This paper introduces MiCA, focusing especially on compositional features and illustrating how these features facilitate building sophisticated applications in a modular style.

1. Introduction

Many management infrastructure tasks can be expressed using *gossip protocols*. including overlay management [4], system monitoring [8–10], and distributed storage [2, 5, 7, 9]. With gossip, each node in a distributed system periodically exchanges information with a randomly-selected peer. Gossip protocols are especially well-suited to the modern cloud data center since they are highly fault-tolerant (a benefit of writing protocols under the expectation of a non-determinism), and they are also well-behaved—periodic communication leads to steady and predictable network load.

This paper presents MiCA (Microprotocol Composition Architecture), a framework for building sophisticated gossip protocols. MiCA has practical and theoretic advantages over ad-hoc coding of these systems:

- High-level abstraction that captures the essence of gossip protocol logic.
- Modular composition operators that enable code reuse, testing of components in isolation, and abstraction over protocol interfaces.
- Optimized implementations of composite protocols that reduce the size and frequency of messages exchanged.

MiCA's vision is to equip system designers with abstractions that capture the essence of gossip protocols, a toolbox of building blocks, and principled composition operators.

2. Motivation for Composition

MiCA's modular architecture supports an lightweight development style. A bare-bones service can be implemented quickly, and then refined by adding extra components to help with scalability, reliability, fault tolerance, and other needs that arise while transitioning a proof-of-concept to a bullet-proof production system. Consider five generations of a hypothetical system:

1. A single master node runs a management protocol. It gossips with a number of workers in a small system. With each gossip exchange, the master records the worker's current load in a table. The master assigns some amount to work to the worker if its load is low.
2. As the service becomes popular, the scalability of this communication pattern becomes a problem. A self-stabilizing spanning tree component is inserted into the protocol stack, and is used to alleviate congestion at the master node. Aggregation and broadcast protocols are added to send and receive data over the spanning tree. The master now receives aggregate data propagated up the tree, and broadcasts task assignments down the tree.
3. Later, the failure of a high-ranking internal node in the spanning tree overlay causes a long system outage. In response, a second spanning tree protocol is

added, creating a redundant and independent spanning tree and making the broadcast/aggregation network more robust. The additional network overhead is deemed acceptable.

4. Next, because the master node is a single point of failure it is replaced with a group of nodes. Each group member is the root of two redundant spanning trees and gossips to other members on a private, small complete graph overlay to maintain a aggregate statistics for the whole system.
5. Finally, A failure-detection and recovery protocol layer is added to the master group, designating group members to step in and assume the responsibility of any master that is detected to have failed.

With each iteration, the system becomes more complex. Although the fifth-generation node is running many logical protocols (two spanning trees for each master node!), MiCA's correlated protocol merging (presented in section 4) will identify and reclaim redundancy in messages sent by different protocols, reducing the effective overhead of the composite protocol to significantly less than the sum of its parts.

3. Gossip Model

MiCA models a gossip protocol in terms of a set of node states S, a set of node identifiers $\mathcal{N}$, and three functions:

$$\begin{array}{ll} \texttt{rate} & \in S \to \mathbb{R}^+ \\ \texttt{view} & \in S \to \Delta(\mathcal{N}) \\ \texttt{update} & \in S^2 \to S^2 \end{array}$$

The execution of each node n with local state s_n is an infinite loop that repeatedly sleeps, selects a peer to gossip with, and exchanges data, as shown in figure 1.

```
do forever:
    sleep(period/rate(s_n))
    m = sample from view(s_n)
    s'_n, s'_m = update(s_n, s_m)
```

Figure 1. Gossip execution on node n. `view` returns a probability-weighted set of neighbors which is sampled by the run-time. `update` computes new states for both nodes in each gossip exchange.

The behavior of the protocol is determined by the following parameters:

`rate`(s_n) defines how fast n gossips. It is evaluated every round, prior to initiating gossip. A rate of 1.0 indicates the node should wait for *period* seconds, a system-wide constant; a sustained rate of 2.0 indicates the node is gossiping twice per *period*. Because `rate` is a function of current node state, it may change over time. Most primitive (i.e., non-composite) protocols gossip at a constant rate. Varying the rate can violate the well-behaved property of gossip, so it must be done with caution. MiCA's composition operators use dynamic rates, but with strict upper bounds.

`view`(s_n) controls peer selection for node n. It returns a probability mass distribution over the nodes in the system, $\Delta(\mathcal{N})$. Although most conventional gossip protocols choose uniformly at random from a set of peers, MiCA's probability distributions serve three purposes: First, they admit interesting gossip policies, such as "gossip more often with nearby neighbors," and second, they are essential to MiCA's notion of composition, which averages probability weights to build a probability mass distribution that represents the combined preferences of two protocols. MiCA's run-time system samples the distribution. Third, they lift non-determinism out of the protocol definition and give the run-time system access to the entire distribution. As with `rate`, the `view` function is evaluated every time a node gossips, so it may change over time.

`update`(s_n, s_m) is the *gossip exchange function*. When node n gossips with node m, the `update`(s_n, s_m) function computes the updated states of both nodes. The `update` function is written by the programmer as if both node states are local, with no explicit network communication. MiCA's run-time distributes the `update` function to run between two remote nodes. Critically, this is done after protocol composition has occurred, so that MiCA's compiler and run-time have full access to the states of both endpoints when creating composite `update` functions. This pair-of-nodes representation is a key difference between MiCA and other object-oriented gossip frameworks[3][1].

4. Composition

MiCA provides composition operators that combine multiple sub-protocols into a single composite pro-

tocol. We consider only binary operators in this paper. Composition typically has two parts: Synthesizing composite `view` and `rate` functions, and deciding what (if any) state should be shared between sub-protocols.

In this paper, we consider two composition operators for combining `view` and `rates` in a composite protocol: independent merge ($\oplus_{ind}$) and correlated merge ($\oplus_{cor}$). Both operators multiplex sub-protocols, but differ in their philosophies. The correlated merge operator exploits overlapping `views`, piggy-backing messages together whenever possible. This has performance benefits, but makes only one random peer selection for two protocols, which may be problematic if a system depends on independent random selections. The independent merge operator perserves independence of sub-protocol choice, but it lacks the performance benefits of correlation.

A composite protocol formed with these binary operators forms a hierarchy, with the root being the top-level protocol that the MiCA run-time system executes. MiCA only runs one top-level protocol per node, so running two protocols concurrently requires composing them together first.

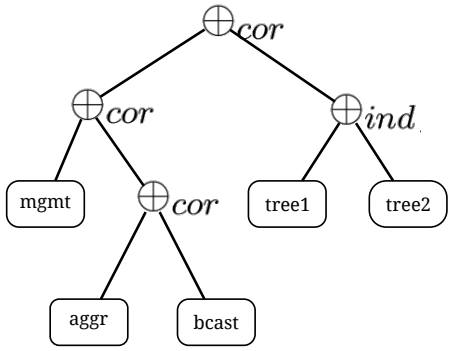

Figure 2. Composition hierarchy used in the third generation of the hypothetical example, showing management, aggregation, broadcast, and two spanning tree building sub-protocols. $\oplus_{cor}$ is used for performance, and $\oplus_{ind}$ is used to combine the two spanning-tree protocols to promote independent network overlays.

MiCA's composition operators also offer guarantees on behavior. We define two properties which, taken together, state that the observable behavior of the composed sub-protocols remains unchanged by composition.

Rate Preservation. A composition operator is rate-preserving if each sub-protocol continues to gossip at its original rate. To accommodate rate preservation, composite protocols increase their rates.

View Preservation. A composition operator preserves the views of its sub-protocols if each sub-protocol gossips, in aggregate, to other nodes in a way that is consistent with their probability masses in the sub-protocol's `view`.

Both independent and correlated merge are rate- and view-preserving, although sharing state between protocols (discussed later) can lead to violations. MiCA's composition operators are associative.

Correlated Merge

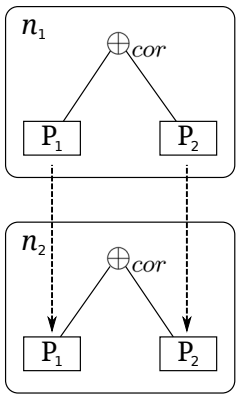

Figure 3. Correlated merge: When node n_1 gossips with n_2, sub-protocols $P1$ and $P2$ gossip. This happens as often as possible without violating view-preservation.

The correlated merge operator attempts to bundle messages together, gossiping both sub-protocols simultaneously to a common neighbor. If sub-protocol views are identical probability mass distributions, then correlated merge will bundle messages together on every round. Conversely, if the distributions are disjoint, then messages will never be bundled because doing so would violate view-preservation by causing a sub-protocol to gossip with a peer that is weighted with zero in its view. When `views` partially overlap, $\oplus_{cor}$ uses a combination of both techniques. Figure 4 shows sample implementation code for $\oplus_{cor}$.

```
class CorrelatedMerge extends Protocol
  Protocol p1, p2;
  Distribution<Address> view() {
    double r1 = p1.rate();
    double r2 = p2.rate();
    double w = r1 / (r1 + r2);
    Distribution<Address> d1 =
        p1.view().scale(w);
    Distribution<Address> d2 =
        p2.view().scale(1-w);
    return Distribution.max(d1,
        d2).normalize();
  }
  double rate() {
    double r1 = p1.rate();
    double r2 = p2.rate();
    Distribution<Address> d1 =
        p1.view().scale(r1);
    Distribution<Address> d2 =
        p2.view().scale(r2);
    return Distribution.max(d1,
        d2).magnitude();
  }
  void update(CorrelatedMerge other) {
    CorrelatedMerge that =
        (CorrelatedMerge) other;
    double r1 = p1.rate();
    double r2 = p2.rate();
    double w = r1 / (r1 + r2);
    double pr1 = p1.view().get(that) * w;
    double pr2 = p2.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 p1 gossips
      p1.update(that.p1); break;
    case 1: // only p2 gossips
      p2.update(that.p2); break;
    case 2: // both p1 and p2 gossip
      p1.update(that.p1);
      p2.update(that.p2);
    }
  }
}
```

Figure 4. Correlated merge implementation. The `Distribution<Address>` class represents a probability distribution over network addresses. Constructors and other boilerplate are omitted for brevity.

In figure 4, `update` gossips sub-protocol $P1$, $P2$, or both, based on a random choice with weights derived from the odds of the run-time having chosen node *other* due to contributions to the composite `view` from each sub-protocol.

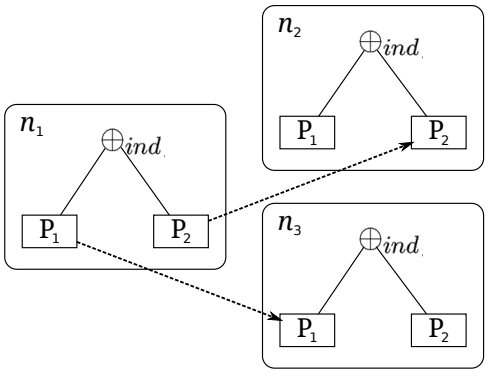

Figure 5. Independent merge: Sub-protocols on node n_1 make independent peer selections and gossip separately. The merge operator increases its rate to keep each sub-protocol running at its requested speed.

Independent Merge

Independent Merge is ideal for situations that call for true independence of gossip. It is unable to exploit overlap in communication, as doing so may sometimes violate implicit assumptions of sub-protocol independence. One such situation is random-walk peer sampling[6]: a random-walk token is passed around the network until a time-to-live counter expires, and information is reported about the peer holding the token when it stopped. If several random walk protocol instances were merged in an attempt to gather multiple samples, $\oplus_{cor}$'s opportunistic bundling would send all tokens along the same random path.

The independent merge operator $\oplus_{ind}$ is defined as follows:

`view` computes a point-wise mean of sub-protocol views, weighted by rate. As with $\oplus_{cor}$, during `update`, the probability of having sampled the designated peer is calculated, and used to weight the choice of which sub-protocol to execute. Because messages are never bundled, `rate` rate is simply the sum of the sub-protocol rates.

State-Sharing

State-sharing refers to the interaction between sub-protocol states during execution. In MiCA's Java implementation, this takes the form of two sub-protocol objects reading and modifying a common reference.

MiCA does not enforce any state-sharing rules, but neither do its stock composition operators introduce any state-sharing. The simplest case is *isolation*, in which the two protocols do not affect one another, effectively reducing composition to multiplexing. Under isolation, the states of the sub-protocols remain disjoint. Formalization of state-sharing schemes is beyond the scope of this short paper, but for composition operators, proving isolation-preservation would entail showing that running two subprotocols under composition is indistinguishable from running them as two separate top-level protocols in two separate MiCA runtimes on the same node. More interesting is *embedding*, where the state of one sub-protocol state is embedded within the state of the other. This pattern appears when one protocol uses another as a service. For instance, sub-protocol P_1 could build a network overlay topology, and P_2 could use that overlay to define its `view`. We say P_1 is be *embedded* in P_2, indicating a subordinate relationship—P_1 is unaware of P_2, but meanwhile P_2 is reading from P_1's state. The embedding protocol P_2 should not modify the embedded P_1 state, as doing so may disrupt the correctness of P_1. Embedding is useful for stacking together layers of protocols, where each layer depends on the one before it. Figure 7 shows the construction of such an embedding. Embedding admits a range of useful subprotocol interaction, but its one-way state dependence (P_1 is independent of P_2) is valuable for proving properties of the embedding protocol based on the embedded; for example, "If P_1 stabilizes, then P_2 also stabilizes.". Unrestricted state sharing where two subprotocols are allowed to mutate each others' state can invalidate correctness properties of individual protocols.

5. Conclusion

The MiCA gossip framework contributes a novel formulation of gossip protocols, amenable to composition that preserves key properties of sub-protocols. It offers composition operators for increased performance or reliability. MiCA aims to supply standardized microprotocol components which can be used to implement a wide variety of systems with real-world applications in the data centers that power the cloud.

```
class BasicLeaderElection extends
    Protocol implements LeaderElection {
  Address leader;
  Distribution<Address> view;
  LeaderElection(Distribution<Address>
      view) {
    view = view;
    leader = getAddress();
  }
  void update(LeaderElection that) {
    if(this.leader.compareTo(that.leader)>0){
      this.leader = that.leader;
    } else {
      that.leader = this.leader;
    }
  }
  @Override
  boolean isLeader() {
    return leader.equals(getAddress());
  }
  Distribution<Address> getView() {
    return view;
  }
}
```

Figure 6. Basic leader election protocol: Nodes sort each other by their network address, electing the smallest address as leader. Other protocols will call the isLeader() method, which is part of the LeaderElection interface.

```
Protocol
    createComposite(Distribution<Address>
    view) {
  // Leader election layer
  LeaderElection leaderElection = new
      BasicLeaderElection(view);
  // Leader serves as the root of a
      spanning tree
  SpanningTreeOverlay tree = new
      SpanningTreeOverlay(leaderElection,
      view);
  // Aggregation protocol gossips with
      children in tree, counts nodes in
      subtree
  TreeCountNodes counting = new
      TreeCountNodes(tree);
  // Compose the three protocols
  return new CorrelatedMerge(
    new CorrelatedMerge(leaderElection,
        tree),
    counting);
}
```

Figure 7. Composition example: A spanning tree is built and used for an aggregation. An example leader election protocol is shown in Figure 6. The other two sub-protocols are not included for lack of space.

References

[1] P.-E. Dagand, D. Kostić, and V. Kuncak. Opis: Reliable distributed systems in OCaml. In *International Workshop on Types in Language Design and Implementation (TLDI), Savannah, GA*, pages 65–78, Jan. 2009.

[2] 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 *ACM Symposium on Operating Systems Principles (SOSP), Stevenson, WA*, pages 205–220, Oct. 2007.

[3] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *International Conference on Distributed Computing Systems (ICDCS), Amsterdam, Netherlands*, pages 22–29. IEEE Computer Society Press, May 1998.

[4] M. Jelasity, A. Montresor, and Ö. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[5] A. Lakshman and P. Malik. Cassandra: Structured storage system on a P2P network. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Calgary, Canada*, page 5, Aug. 2009.

[6] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Denver, Colorado*, pages 123–132, Aug. 2006.

[7] Riak: An open source scalable data store, Nov. 2010. Available at `http://docs.basho.com/index.html`.

[8] 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, 2006.

[9] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computing Systems*, 21(2):164–206, 2003.

[10] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-based failure detection service. In *International Middleware Conference (Middleware), The Lake District, England*, pages 55–70, Sept. 1998.