Distributed systems are hard to program, for a number of reasons. Some of these reasons are inherent. System state is scattered across multiple nodes, and is not random access. Bandwidth is an ever-present concern. Fault tolerance is much more complicated in a partial-failure model. This presents both a challenge and an opportunity for the programming languages community: What should languages for distributed systems look like? What are the shortcomings of conventional languages for describing systems that extend beyond a single machine?


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-stabilizing 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 select 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 $\Sigma$ denotes the set of possible node states, the types of these functions can be written as follows:

$$select \in \Sigma^2 \rightarrow \text{Address}$$

$$update \in \Sigma^2 \rightarrow \Sigma^2$$

Execution proceeds in rounds. In each round, every node uses the select 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. These 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
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 1. Simple leader election protocol in CPG. (Some boilerplate code elided for brevity)