A Pairwise Abstraction for Round-Based Protocols

Lonnie Princehouse Nate Foster Ken Birman
Department of Computer Science, Cornell University
{ lonnie, jnfoster, ken }@cs.cornell.edu

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?

Prior work in this area has produced diverse solutions. The DryadLINQ [7] language expresses distributed computations using SQL-like queries. BLOOM [1] also follows a data-centric approach, but assumes an unordered programming model by default. MACEDON [6] provides constructs for describing overlay networks. P2 [5] uses declarative syntax based on Datalog to express network protocols. Bast [4] provides object-oriented, extensible, and composable protocols. Lastly, Jini [2] offers a framework for extensible network services.

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

Our system, Code Partitioning Gossip (CPG), provides an abstraction that lies between the single-node and whole-system perspectives. It is designed specifically for synchronous, fault-tolerant systems—a class that includes many gossip and self-stabilization protocols. These are especially relevant to current computing trends. Because of their passive, round-based nature, they tend to be well-behaved and make predictable use of the network. As such, they are "good neighbors" in massive multi-tenant data centers, such as those that drive Amazon's EC2. Many cloud computing services have relaxed consistency requirements in favor of availability, and this also plays to the strengths of round-based protocols.

Our goal with CPG is to design abstractions for describing these protocols that make it easy to develop richer protocols via composition and code re-use. The fundamental unit seen by programmers in CPG is a *pair* of nodes. A protocol in CPG is defined using a *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 Σ denotes the set of possible node states, the types of these functions can be written as follows:

$$\begin{aligned} \mathit{select} &\in \Sigma^2 \to \text{Address} \\ \mathit{update} &\in \Sigma^2 \to \Sigma^2 \end{aligned}$$

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 langauges 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)

overlay implementations to be easily substituted. Additionally, it is possible to generalize transformations on protocols. For example, [3] outlines a "pipelining" procedure, by which an arbitrary self-stabilizing protocol can be imbued with Byzantine fault tolerance. CPG's abstractions make it possible to implement pipelining as a function on protocols.

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

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

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

References

[1] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. BOOM analytics: Exploring data-centric, declarative programming for the cloud. In *EuroSys*, pages 223–236, 2010.

[2] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[3] Danny Dolev and Ezra N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *SSS*, volume 4838 of *LNCS*, pages 234–252, November 2007.

[4] Benoit Garbinato and Rachid Guerraoui. Flexible protocol composition in Bast. In *ICDCS*, pages 22–29. IEEE Computer Society Press, 1998.

[5] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In Andrew Herbert and Kenneth P. Birman, editors, *SOSP*, pages 75–90. ACM, 2005.

[6] Adolfo Rodriguez, Sooraj Bhat, Charles Killian, Dejan Dostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. Technical report, Duke University, July 2003.

[7] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.