Code-Partitioning Gossip

Lonnie Princehouse
Dept. of Computer Science
Cornell University
lonnie@cs.cornell.edu

Ken Birman
Dept. of Computer Science
Cornell University
ken@cs.cornell.edu

ABSTRACT

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

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design,Languages,Algorithms

Keywords

gossip protocol program slicing code partitioning

1. INTRODUCTION

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

Gossip is used for a wide range of tasks. Some general types of gossip protocols are:

Rumor mongering. Nodes seek to disseminate messages by passing the messages to their neighbors who will, in turn, pass them to theirs. As time passes, the probability that every node has a particular message approaches one. A system might use rumor mongering to collect a complete set of something that is originally spread across the network, such as database meta-information[3] or a list of received packets[1].

Aggregation. Similar to rumor-mongering, but computing a function of data held at different nodes. Examples are: Approximating statistics about node capacity throughout the network[5][6]; computing user-defined aggregate queries[15].

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

Peer sampling. Randomness of peer selection is important for many gossip protocols. For large networks, however, it is impractical for each node to store the address of all other nodes in the system. Peer sampling algorithms allow gossip nodes to sample values maintained by their peers in a way that approximates true random peer selection given only a fixed-size local view.[11]

This paper concerns itself with both the analysis and implementation of gossip systems. Accordingly, we consider two different perspectives on gossip—that of the programmer, and that of the theorist.

The programmer formulates gossip with implementation in mind. A gossip system uses two threads per node; one active, one passive[8]. The active thread periodically initiates a gossip exchange with a randomly selected peer, and the passive thread awaits and reacts to connections. In this paper, we use the terminology “sender” to refer to the active-thread node that initiates a gossip exchange, and “receiver”

```
// Active thread, running on node $a$ with state $\sigma_a$
do forever:
  wait $t$ seconds
  $b \leftarrow$ selectPeer($\sigma_a$)
  send $\sigma_a$ to $b$
  receive $\sigma_b$ from $b$
  $\sigma_a \leftarrow$ update$_a(\sigma_a, \sigma_b)$
```

```
// Passive thread, running on node $b$ with state $\sigma_b$
do forever:
  $a \leftarrow$ awaitConnection
  receive $\sigma_a$ from $a$
  send $\sigma_b$ to $a$
  $\sigma_b \leftarrow$ update$_b(\sigma_a, \sigma_b)$
```

Figure 1: Active and passive threads

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

Figure 2: MAXVALUE protocol

to indicate the passive-thread recipient, even though both nodes send and receive data. For brevity, all examples name the sender a and the receiver b. Figure 1 contains pseudocode for the sender and receiver event loops. Note that during an exchange, each node sends its state to the other, and then computes a new state based on the pair of states. In gossip terminology, this is a *push-pull* protocol, and it encompasses the more specific sets of *push* protocols (in which only the sender pushes its state to the receiver) and *pull* protocols, in which state moves only from receiver to sender.

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

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

In this paper, we present Code-Partitioning Gossip (CPG), a Programming Languages-inspired technique for the implementation of gossip protocols. CPG strives to reach a happy medium between the programmer and the theorist. Using CPG, the programmer writes a unified update function that operates on pairs of states. This function is automatically partitioned into update$_a$ and update$_b$, and code for the active and passive threads is synthesized. Networking code is inserted automatically, allowing systems to be easily re-tooled for different network models and transports.

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

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

2. DESIGN

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

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

selectPeer : $\Sigma \rightarrow N$. Chooses a peer to gossip with based on a node's state. Allowed to be non-deterministic.

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

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

Figure 2 contains the actual Java code for MAXVALUE as implemented in our system. MAXVALUE is *mostly ordinary Java code*: The gossip protocol is written as a class, and instances of this class represent individual nodes. The only unusual features are the annotations GossipSelectPeerUniform and GossipExchangeUpdate on lines 12 and 15. These annotations tag elements of the program for special treatment by our runtime system. GossipSelectPeerUniform tells the runtime that the member variable *view* is to be used for uniform random peer selection, and GossipExchangeUpdate marks the function update for automatic partitioning.

```
@GossipExchangeUpdate
public void update(Maxvalue b) {
  value = b.value = max(value, b.value);
}
```

↓

```
public void update_a(Maxvalue b) {
  value = max(value, b.value);
}
public void update_b(Maxvalue a) {
  value = max(a.value, value);
}
```

Figure 3: MAXVALUE automatic partition

We employ static program slicing[16] to accomplish this partition. Briefly, program slicing attempts to solve the following problem: Given a program and a target value (as it appears at some program point), return a subgraph of the program's control flow graph consisting only of statements that contribute to the computation of the target value. This CFG subgraph is called a "slice", and is itself an executable program. When executed, the slice computes the target value exactly as the original program would have. CPG's program slicing is necessarily conservative, omitting statements only if they are proven irrelevant.

Code-Partitioning Gossip generates two slices: one that computes updated state σ'_a for the sender, and one that computes σ'_b for the receiver. These slices are effectively the update_a and update_b functions seen earlier. Figure 3 shows an example of the how MAXVALUE's update function could be partitioned. Code-Partitioning Gossip expects the update function to be deterministic and to halt; the onus to enforce these conditions is on the programmer.

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

3. IMPLEMENTATION

In our prototype implementation, CPG protocols are written in Java, with custom annotations used to designate a protocol's peer selection and exchange update behavior. We considered creating a domain-specific language for gossip, but ultimately decided against it on the grounds that Java provides sufficient extensibility to accomplish our goals, and many programmers are already familiar with Java. When a Java class is loaded, the Code-Partitioning Gossip runtime uses reflection to search for members tagged with one of several special gossip annotations. The annotation GossipExchangeUpdate on a method causes the method to be partitioned and two new methods, representing the two slices of the update method, are dynamically added to the class. These functions are called to perform gossip exchanges by active and passive gossip threads implemented by the Code-Partitioning Gossip runtime.
Our prototype implementation of CPG has two phases of analysis, both operating on the Java bytecode of a protocol class. Running this analysis on bytecode rather than Java source was a pragmatic decision—we felt it would be easier to write a prototype using existing bytecode manipulation tools—but it has some additional benefits, such as the potential to write CPG gossip protocols in any language that targets the JVM (e.g., Scala). For CPG, first the update method is is sliced into active and passive methods that update the states of two local node instances. Second, network code is injected to retrieve state from the remote node when it is needed. Several annotations are provided for peer selection. GossipSelectPeerUniform selects a peer uniformly at random from a set of addresses of other peers. GossipSelectPeerWeighted lets the developer specify probability mass weights (for protocols that require non-uniform random selection, e.g., spatial gossip[9]). GossipSelectMethod designates a method to call directly for peer selection.

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

3.1 Example

We now present a more sophisticated example. Sliver[5] is a slicing protocol. In a network where nodes have varying capacities of some metric, Sliver assigns each node to one of k groups of approximately equal total capacity. Nodes provide a getSlice method that returns an estimate of their current slice; this is computed as follows:

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

Figure 4 shows Sliver as implemented under Code-Partitioning Gossip.

4. RELATED WORK

We are aware of one API framework, GossipKit[10], that uses standard object-oriented programming methodology to furnish the developers of gossip protocols with reusable, modular gossip abstractions. Such a framework serves two purposes: It provides plug-and-play gossip protocols that can

```
public class Sliver {

  private class Rumor {
    public Long timestamp;
    public Double capacity;
    Rumor(Double capacity) {
      timestamp = new Date().getTime();
      this.capacity = capacity;
    }
  }

  private Address address; // This node's address
  private int k; // Number of slices
  private Double capacity;
  private long timeout;

  // Everything we know about other nodes'
  // capacities
  private HashMap<Address, Rumor> rumors;

  public Sliver(int k, Double capacity,
       long timeout, Set<Address> view,
       Address address) {
    this.rumors = new HashMap<Address,Rumor>();
    this.address = address;
    this.k = k;
    this.capacity = capacity;
    this.view = view;
    this.timeout = timeout;
  }

  @GossipSelectPeerUniform
  public Set<Address> view; // known peers

  @GossipExchangeUpdate
  public void update(Sliver b) {
    // Tell the other node about this node's
    // capacity
    b.rumors.put(address, new Rumor(capacity));
  }


  // Called by user to determine this node's slice
  public long getSlice() {
    purgeExpiredRumors();
    long m = rumors.size();
    long B = 0;  // number of known peers with
                 // capacity not greater than
                 // ours
    for(Rumor i : rumors.values() )
      if(i.capacity <= capacity)
        B++;
    return StrictMath.round(k *
        (double) B / (double) m);
  }

  private void purgeExpiredRumors() {
    // Delete expired rumors
    long now = new Date().getTime();
    for(HashMap.Entry<Address,Rumor> e :
          rumors.entrySet() ) {
        Address peer = e.getKey();
        if(now - e.getValue().timestamp < timeout)
          rumors.remove(peer);
    }
  }
}
```

Figure 4: Sliver implementation

be used by developers (e.g., peer sampling), and it facilitates development of gossip protocols by providing a skeletal gossip runtime that can be extended via inheritance. We assert that CPG has an advantage over such a toolkit in that CPG lets the programmer describe a protocol at a higher level of abstraction, namely the pair-wise updates of system state. However, the toolkit approach may be easier to debug since the bytecode run by CPG has been transformed by program slicing.

A second class of related work seeks to generalize specific kinds of gossip protocols. Two such systems are T-Man[7] and Astrolabe[15]. T-Man is a configurable gossip system for the creation and maintenance of structured overlays. T-Man imposes a user-defined sort order $\succ$ over all nodes in the system. Nodes maintain views of fixed size, sorted in this order. When a T-Man node learns of another node x such that $x \succ y$ for some $y \in$ view, x replaces y. The views of each node define the overlay graph. By supplying different sorting functions, T-Man can form a truly surprising variety of overlay toplogies.

Astrolabe organizes its nodes into a tree. The tree's inner nodes may contain user-defined aggregation functions that compute some aggregate of the data stored in the node's children. Users of Astrolabe can then execute database-like queries to evaluate these aggregates. T-Man and Astrolabe do not have the same goals as CPG, so a direct comparison is not possible. However, both T-Man and Astrolabe would make excellent benchmarks if implemented using CPG. Astrolabe, in particular, has a recursive structure that lends itself well to CPG. Implementing these systems using CPG is left for future work.

MACE[12] is a domain-specific language for authoring overlay systems, intended for writing overlays such as Chord[14], Pastry[13], etc. MACE compiles into C++, and claims to save a great deal of programmer effort and attain reasonable performance. While it is not gossip-specific, we see no reason that MACE could not be used to implement gossip systems.

Regarding program slicing and automatic partitioning, Jif/Split[17] and Swift[2] use such a technique to automatically partition programs to run between client and server according to information flow security labels on variables. If anything, Code-Partitioning Gossip is much less ambitious in the scope of its partitioning scheme: Jif/Split and Swift must decide where and when to move data based on a set of hard security constraints, whereas CPG has the locations of variables as a given from the start. CPG differs from these systems in that its pair-wise program slicing implicitly defines the behavior for an n-node system.

5. CONCLUSIONS

The essential idea of Code-Partitioning Gossip is that of writing a gossip exchange as a single atomic function, and then automatically partitioning this function into code for the roles of sender and receiver. We believe this technique offers several advantages.

5.1 Composition

A distributed hash table system might make use of several gossip protocols: A peer sampling protocol to draw adequately random samples from its members, an overlay maintenance protocol to adjust the overlay according to node arrival and departure, a counting protocol to estimate the number of nodes in the system, and an aggregation protocol to estimate the most popular objects to allow nodes to make better caching decisions. The status quo would implement this bundle of protocols in one of two ways: Either as a single monolithic protocol, or as four separate protocols that operate independently, each running its own active and passive threads. In this situation it is difficult to reap any benefit from commonality in communication or computation without significant code rewriting, unless perhaps all four protocols have been written using a middleware layer that abstracts away low-level network code. Using Code-Partitioning Gossip, however, the protocols are composed *prior to partitioning.* Instead of trying to merge multiple active and passive threads, we invoke each protocol's `update` method from within the a single superior `update` method, which is then partitioned CPG allows protocols to be composed in much the same way functions and objects are composed in object oriented programming. This composition can take the form of a top-level udpate function that calls the update functions of sub-protocols, or of extending a protocol by inheriting it. We have a cursory implementation layered self-stabilizing protocols[4] using CPG, but more work is needed to evaluate the real utility of CPG.

5.2 Analysis

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

6. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments, and Danny Dolev for good discussions that influenced the content of this paper. This work was funded in part by grants from NSF and AFRL, with additional support from the Intel Corporation.

7. REFERENCES

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[2] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 31–44, October 2007.

[3] 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 *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[4] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[5] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse. A fast distributed slicing algorithm. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 427–427, New York, NY, USA, 2008. ACM.

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

[7] M. Jelasity and O. Babaoglu. T-man: Fast gossip-based constructions of large-scale overlay topologies. Technical report, 2004.

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

[9] D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. pages 163–172. ACM Press, 2001.

[10] S. Lin, F. Taiani, and G. S. Blair. Facilitating gossip programming with the gossipkit framework. In *DAIS*, pages 238–252, 2008.

[11] L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2006. ACM.

[12] A. Rodriguez, S. Bhat, C. Killian, D. Dostic, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. Technical report, Duke University, July 2003.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

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

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

[17] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning, 2001.