A Middleware for Gossip Protocols

Michael Chow
Cornell University
mcc59@cornell.edu

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

Gossip protocols are known to be highly robust in scenarios with high churn, but if the data that is being gossiped becomes corrupted, a protocol's very robustness can make it hard to fix the problem. All participants need to be taken down, any disk-based data needs to be scrubbed, the cause of the corruption needs to be fixed, and only then can participants be restarted. If even a single participant is skipped in this process, say because it was temporarily unreachable, then it can contaminate the entire system all over again. We describe the design and implementation of a new middleware for gossip protocols that addresses this problem. Our middleware offers the ability to update code dynamically and provides a small resilient core that allows updating code that has failed catastrophically. Our initial PlanetLab-based deployment demonstrates that the middleware is efficient.

1 Introduction

Gossip or epidemic protocols provide state updates in a scalable and reliable manner [4, 5, 7]. However, this very property can make the management of gossip applications cumbersome. Unlike other systems, where components can be updated one at a time, updating a gossip protocol that has gone bad often requires taking down the entire system and redeploying.

For example, the Amazon S3 storage system makes use of a gossip protocol to disseminate server state information. During the course of a system outage on July 20, 2008, some server state information became corrupted, possibly due to a spontaneous bit flip. A corrupted value began disseminating causing servers to fail. The entire system needed to be shut down and on-disk state information fixed on every machine before the system could be brought back. The system was down for more than 6 hours before service could be restored [2].

We often see these kinds of scenarios when developing our own gossip-based applications. Small bugs in the system that cause a local data corruption can create a poison pill that infects the entire system like a malicious virus and results in significant overhead when trying to fix the problem. We desired a solution with which we could robustly repair such problems. Our idea was to gossip new code that fixes the corruption and restores service, but the challenge was how to gossip the code when the very gossiping infrastructure is broken.

In this paper we propose a layered middleware for gossip protocols with the capability of rapid code updating. Our Java-based implementation provides a flexible and resilient framework for developing gossip applications and allows for robust management of gossip protocols. The code updating scheme makes use of gossip to distribute code, inspired by Trickle, an algorithm for propagating code updates in wireless sensor networks [9]. The code updating scheme is managed in the bottommost layer of our middleware. We call this layer the *core*. Since it alone cannot be updated dynamically, many of our design decisions are driven by the desire to keep the core to be small and simple.

In Section 2, we describe how versions of code are described. The architecture of the core is described in Section 3. In Section 4, we look at the performance overhead of our middleware. We discuss related work in Section 5, and conclude in Section 6.

2 Code Versions and Deployments

Our middleware can support a collection of concurrently running gossip-based applications, each of which is implemented by a *module*. Modules can interact, as we will describe later in Section 3.3. Modules can also be updated, and in this section we describe how versions and deployments of modules are described.

Each module has a name that uniquely identifies it to the core. A *version* of the module consists of a set of immutable Java class files that we call a *code archive*. Each class file is stored as an instance of a wrapper class, which contains the byte data of the class as well as the class name and size. One class is designated as the *module class*. The module class implements the Module interface, which describes how the core can interface with the module.

A *deployment* of a module is a tuple consisting of a module version and a *deployment number* that uniquely identifies the deployment. The deployment number itself is a tuple consisting of the time the deployment was initiated and the ID of the node that initiated the deployment (the node that initially loaded the code). Deployment numbers are unique and ordered lexicographically. See Figure 1 for an example. The core maintains a mapping of module name and deployment number to the corresponding code archive: $(\mathit{Deployment\ Number}, \mathit{Module\ Name}) \rightarrow \{\mathit{Class}_1, \mathit{Class}_2, ...\}$.

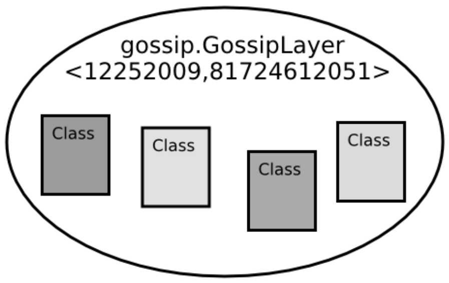

Figure 1: Example of a module. A module is identified by a unique module name, gossip.GossipLayer, and its deployment number <12252009,81724612051>. These two identifiers map to a set of classes.

It is important to distinguish between versions and deployments of a module. For example, suppose our system is initially running version v_1 of a module with deployment number d_1. We update our module with a newer version v_2. The core then proceeds to deployment number d_2. Next we discover a bug in our module and decide to roll back the code to version v_1. We proceed to update our system with code corresponding to v_1, using deployment number d_3. Thus, deployment numbers do not correspond with versions and different deployment numbers may map to the same versions of the code.

3 Core Architecture

The core manages modules and mediates in gossip between modules of the same type at different nodes. The core itself is a module, and indeed can gossip autonomously with cores on peer nodes. Because the core cannot be updated, the services provided by the core should be small and simple. If there is a bug in the core, then the entire system must be taken down so that the bug can be fixed, and we are trying to keep this to a minimum. In our prototype, the core system is just under 2000 lines of Java code (not counting standard Java libraries that we use). This includes code for loading and managing code updates.

A configuration file contains the list of modules, their current versions (identified by hash codes of the corresponding class files), and a deployment number. The configuration file determines which modules and corresponding versions are currently running. The configuration file has its own system-wide deployment number, which maps to a list of modules and their versions. The system-wide deployment number allows nodes to identify whether or not their configuration is up-to-date. The core also maintains some information it needs to gossip autonomously, such as the location of rendezvous nodes for bootstrapping and a list of membership hints (see Section 3.2).

The core provides a small set of functions to the modules that it is running. This includes a function to gossip with another destination. We describe these functions below.

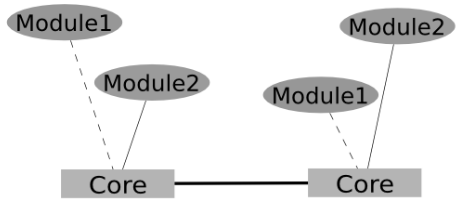

Figure 2: The core system demultiplexes messages to specific modules depending on the unique module name.

3.1 Gossip and Code Update

The core acts as a standard HTTP web server, listening on a configured port number. The URI in each HTTP request starts with a module name, and the core forwards the request to the corresponding module and have it serve the request. The demultiplexing function of the core is illustrated in Figure 2. The core itself acts as a module using the reserved name *core*. The core offers a web interface for access to information such as loaded modules and deployment numbers. An example of an HTTP request for the core's configuration file is: GET /core/config HTTP/1.1

Gossip, too, happens using HTTP GET and POST requests between matching modules on two different nodes. A gossip request contains the deployment number of the source of the request. On receipt, in the common case that the deployment numbers match, the core demultiplexes the message and delivers it to the module

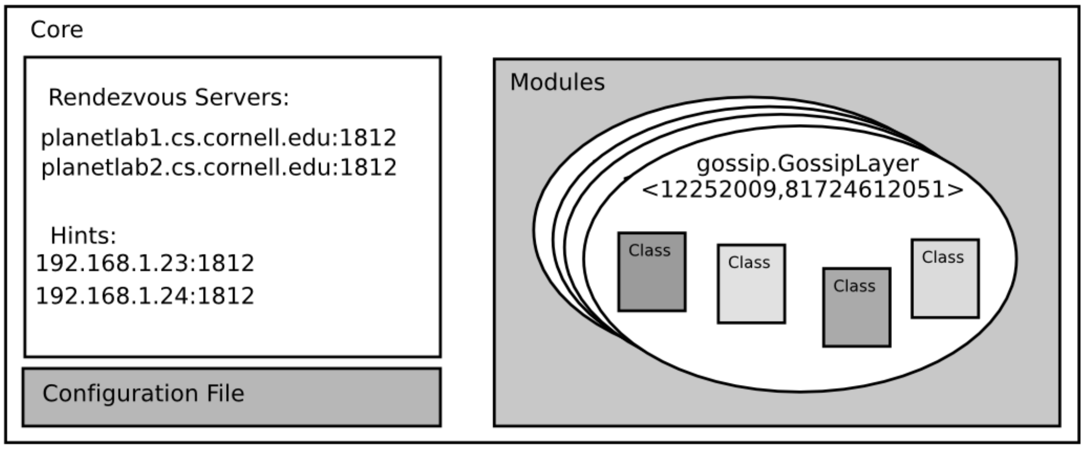

Figure 3: The core maintains a configuration file that lists all of the modules and their versions. It also keeps a list of rendezvous servers and membership hints provided by its modules.

identified in the HTTP request. The module generates a response that is returned to the requesting peer.

If the deployment number in a request does not match the deployment number of the local configuration file, the receiver determines which of the two nodes has the more recent configuration. Using HTTP messages, the more recent configuration file is transferred to the peer, and the missing class files are requested and transferred.

Code updating is implemented through Java class loading. A custom class loader searches the appropriate code archive for the module class, and then loads and instantiates the class. The core provides the ability to transfer module state from an old version of the module to a new one. For example, an application may have built up a membership view over some period of time and for good continuing performance it is important that the new module can pick up from where the old module left off as much as possible. Thus, in our Module interface we provide two methods for transferring state. The method signatures are:

```
public String transferState()

public void acceptState(String state)
```

It is up to the developer to decide what state should be transferred, but we require that it is packed into a Java String object to ensure that there will be no class conflicts. The core explicitly stops the old module before any state is transferred. After the core transfers state information from the old module to the new module, the core starts execution of the new module. This prevents the old module and the new module from simultaneously executing.

Alternatively, we could have provided a small data store in the core, but we decided against this as we wanted to keep the size and functionality of the core small.

3.2 Gossip between Cores

As described, the code update functionality piggybacks on existing gossips between modules. We designed it this way to keep the core small. However, we do not want to depend on existing modules for configuration and code updates to work correctly, as modules may fail. Also, initially the core will be running without any modules other than itself.

The core implements a very rudimentary but robust gossip protocol. For this it has a static list of rendezvous nodes for bootstrapping purposes, and it maintains a small list of membership hints. (Deployments typically originate at rendezvous nodes although they can originate from any node.) The list of membership hints is initialized with the addresses of the rendezvous nodes. During normal operation, the core monitors successful gossip exchanges between modules and adds addresses to its membership hints. The core keeps this list of hints relatively small (configured, but typically on the order of two dozen addresses) and maintains the most recent addresses of successful gossip exchanges in there, as well as the addresses of the rendezvous nodes.

Periodically, the core selects a random membership hint and attempts to gossip with the core at the corresponding address. If this fails, then the core removes the hint from its list of hints. Note that if all else fails, eventually the core will be gossiping with only the rendezvous nodes. This ensures that the core can continue to obtain configuration and code updates even if all its modules fail, but will be inefficient as long as there is no reasonably working gossiping module loaded.

3.3 Layers

It is often useful for different modules to make use of each other's services. For example, one module could gossip membership, while another module could gossip failure information, and yet another module could implement an aggregation service. These modules could make use of each other's services rather than duplicate functionality. In order to support this, our middleware provides a layer abstraction in addition to the module abstraction. Modules can register, with the core, a Layer interface that other modules can access. Indeed, the core itself exports such a Layer interface. A module may use multiple layers. See Figure 4 for an example.

The Layer interface is non-blocking and upcall-based. For example, a Layer that implements membership invokes an upcall to all interested modules whenever membership changes. The interface calls are mediated through the core so that individual modules can be updated without affecting other modules, even dependent ones.

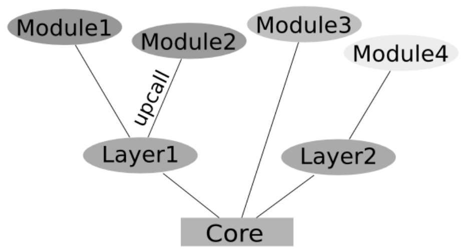

Figure 4: Layers are simply modules, but they can provide an additional service to other modules. Layers must also register with the core in order for the core to provide code updating services.

4 Performance

We performed a series of tests in order to determine the amount of additional overhead associated with providing automatic code updating. Using a prototype of our middleware, we recorded all of the messages sent and received by the core as well as all of the messages sent and received by an application built on the core. The application we are running is a simple membership protocol that gossips membership views containing 30 members and merges them (taking the union and each selecting 30 members at random, using self-reinforcement [1]). The amount of overhead is the number of messages that the core adds to the total message count from both the core and the application. In a series of tests, we ran 100 instances of our middleware on a local machine with 10 rendezvous nodes containing the loaded code to be distributed. The interval for gossips for the core was set to 10 seconds and 5 seconds for the membership module. We started recording messages as soon as the node was initialized so that we could measure the number of messages sent when code is being sent to the node.

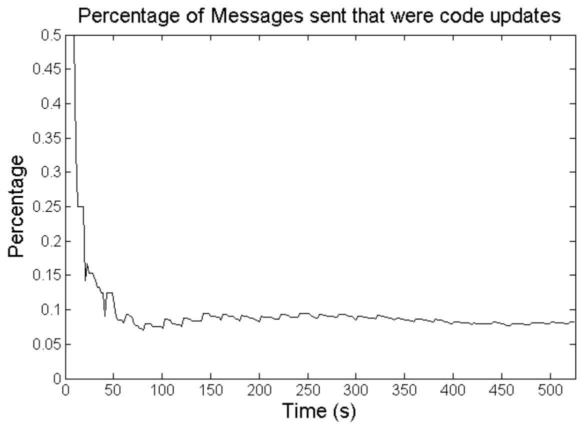

Figure 5: The percentage of messages sent by the core for code updates in a node.

In Figure 5, initially when there is an exchange of code and when the application is first starting up, the messages from the core represent a large portion of traffic. However, after the core has finished loading, the application begins to dominate the traffic, as expected. The core no longer needs to perform code updates and is only sending out messages to check its configuration file periodically.

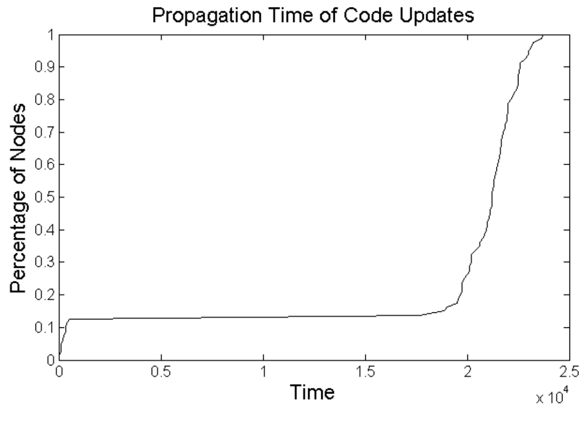

Figure 6: The figure above shows the propagation time of code updates. The rendezvous nodes are shown all the way to the left.

Next we look at propagation time of code updates and how long it takes for code to propagate through the system. Using the same gossip and code intervals from the previous test, we measured how long it took for each node to receive the code after creating a new deployment.

The times were recorded and measured. In the figure, the rendezvous nodes loaded the new code separately at the beginning. The graph shows the typical S-shaped propagation behavior of gossip, starting slowly, then progressing rapidly to reach the remaining participants.

5 Related Work

The code update mechanism that we use is similar to that of Trickle [9], an algorithm used for propagating code updates among wireless sensor nodes. Trickle gossips metadata that describes the version of the running code. Using the gossip, the wireless sensor nodes are then able to detect whether they should broadcast code or request code.

Our work is also related to mobile code and mobile agents, although in our work we disseminate code separately from state (both using gossip). The objective in mobile code systems is usually to bring code to the data, rather than vice versa, in order to avoid moving large amounts of data across the Internet [8]. In active networks, programs can be sent to a node in the network to be executed or processed [3], so that the behavior of a network can be controlled in a dynamic manner.

Gossip applications have been used in data aggregation, overlay maintenance, and resource allocation. Previous work has focused on trying to characterize a model for gossip applications [5, 7].

Other gossip middlewares include GossipKit [10] and T-Man [6]. GossipKit is an event-driven framework and seeks to support a wide array of gossip protocols and applications on various communication layers. It makes use of a modular and inheritance approach to provide extensibility for different gossip applications [10]. In comparison, our work focuses on providing reliable code updating with a layered-upcall architecture. T-Man is a specific gossip middleware for creating and managing different network overlays. Our work can be used to supplement such specific gossip applications by providing a reliable code updating service.

6 Conclusion and Future Work

Our modularized middleware provides the ability to update code dynamically and allows for updating and rolling back of different versions of gossip-based services without having to take the system down, even under catastrophic scenarios where the services have gone sour.

Future work includes offering NAT traversal through a layer service. Another important issue is security, as code may be disseminated from any node, but we can address this by only accepting configuration files signed by trusted authorities and using cryptographically secure hashes to identify class files. We are also looking into ways in which we can support updating the core module itself, although it will be clear that doing so will have to be done with utmost care, as roll-back may not be possible.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions. This work was supported in part by NSF TRUST, AFRL Castor, and AFOSR grants.

References

[1] A. Allavena, A. Demers, and J.E. Hopcroft. Correctness of a gossip based membership protocol. In PODC ’05: Proceedings of the twenty-fourth annual ACM symposium on Principles of Distributed Computing, 2005. ACM Press.

[2] The Amazon S3 Team. Amazon S3 Availability Event: July 20, 2008 [online]. Available from: http://status.aws.amazon.com/s3-20080720.html

[3] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in Active Networks. IEEE Communications Magazine, Special Issue on Programmable Networks, October 1988.

[4] 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, 1987. ACM Press.

[5] P. Eugster, P. Felber, and F. Le Fessant. The “Art” of Programming Gossip-based Systems. ACM SIGOPS Operating Systems Review. Volume 41, Issue 5, October 2007. pp 37-42.

[6] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: Gossip-based fast overlay topology construction. Computer Networks. Volume 53, Issue 13, 28 August 2009, Pages 2321-2339.

[7] A-M. Kermarrec and M. van Steen. Gossiping in Distributed Systems. ACM SIGOPS Operating Systems Review, Volume 41, Issue 5, October 2007. pp 2-7.

[8] D. Lange and M. Oshima. Seven good reasons for mobile agents. Communications of the ACM. Volume 42, Issue 3, March 1999. Pages 88-89.

[9] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI), 2004.

[10] S. Lin, T. Francois, and G. Blair. GossipKit: A Framework of Gossip Protocol Family. 5th MiNEMA Workshop, 2007.