Dynamic Membership in Gossip Protocol Implemented in Golang

Ryan Cain

Aneesh Simha

Erik Trewitt

Goals

In 2017 researcher Alberto Montresor published a paper[1] that proposed a clever solution to manage large membership lists for large clusters by implementing something called a Partial View. Combined with Gossip Protocol this would allow for complete consistency and manage concurrency in a large scale distributed system. For our project, we decided to implement this paper in the Golang programming language developed by Google using the TCP stack. We found many examples of Gossip Protocol being implemented in languages such as C,C++, and Python but not Golang, so we decided that learning this new language as well as the implementation itself would be a great challenge for our team to tackle.

Challenges for Modern Distributed Systems

As today's networks become larger and larger, there is a need for more decentralized protocols that provide robust, fault tolerant forms of communication. Centralized methods of communication such as Multicast and Tree-Based provide a wide variety of problems. In multicast, when a node wants to send a message it will broadcast its message directly to all the nodes in the cluster. Message delivery may fail because nodes may crash during message transmission or packets may be dropped by the network during transmission. The Multicast method by design will not handle these failures to ensure complete transmission. This method is also not viable when clusters begin to scale. Keeping track of all the nodes in the cluster is inefficient when sorting through the list. Another issue is that adding a new node to the cluster, the larger the list gets, the longer it will take for the new node to boot up. Copying the new node's information to all the nodes in the large cluster also slows down performance. The Tree-Based approach is most similar to message forwarding where packets of information are sent from the original node to be forwarded throughout the cluster one by one. This approach also falls prey to the same issues as the Multicast protocol.

A third, more robust approach is the Gossip Protocol. Gossip Protocol is a peer-to-peer communication protocol based on the way epidemics spread virally from host to host. In this protocol, each node in the cluster is able to listen for incoming messages and also send messages to other nodes. Each node has an "infectivity degree" that determines how many messages they can send. For example, if a node has an infectivity degree of five, it can send messages to five other nodes. These five nodes comprise its Partial View. From there each of those nodes sends to five more nodes, and so on and so forth, thus exhibiting exponential growth. This protocol also accounts for node failures and packet losses since the failure of any one node or any message vector would not affect the overall propagation of the message. The failsafe nature of the design is due to the fact that eventually every node is sending the message that is meant to be sent ensuring it gets to its intended recipient quickly. There are three main variants of Gossip Protocol. Push, Pull, and a Hybrid of the two. In Push, once a message is received it is immediately sent out. In Pull, every node is constantly requesting for new multicast messages from random nodes. The Push-Pull hybrid approach has the advantages of both, with the downside of higher bandwidth use and processing time.

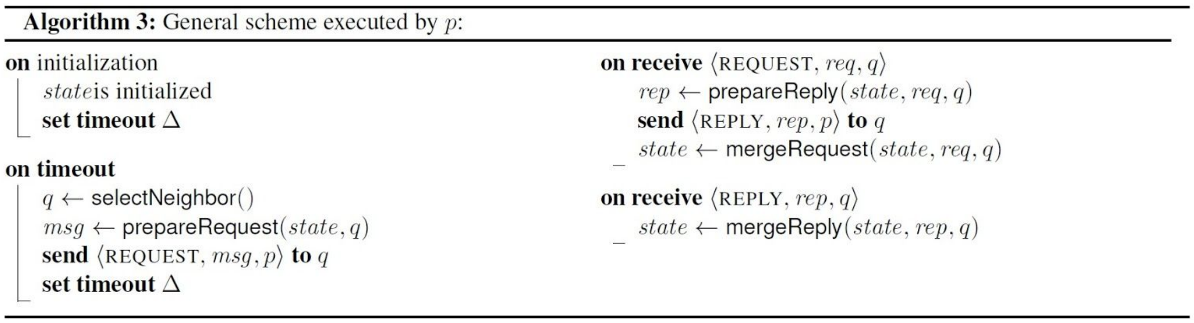
Related Works

There are two main implementations that share the same Partial View based algorithm that our project used. Both of these implementations are discussed in Montressor's paper [1]. The first topic Montressor discusses deals with the topic of distributed aggregation. Distributed aggregation is a set of functions that summarize some global system property. The basic algorithm logic is as follows; each node maintains an estimate of the target global system property, this estimate is then averaged through a time interval, and then gossiped to the network. The algorithm ends when the estimate value for each node converges on some agreed upon averaged value.

The second implementation talks about overlay topology construction. Overlay topology construction is an important tool when creating fully decentralized systems such as a distributed hash table. The main advantage of a gossip-based approach to topology construction is that it allows the overlay topology to be built from scratch, while most other proposed approaches assume the topology is already created [1]. Also, the gossip based approach can be used to merge two existing node networks or to concurrently add a large amount of nodes to a network. Both of these implementations adapt Montresor's generic scheme for a push-pull Gossip algorithm.

Algorithm and Selected Approach

Montresor's membership algorithm[1] takes advantage of the Partial View. The Partial View is the list each node keeps containing information of a small randomly selected cohort of nodes from the cluster. The information stored pertaining to each node is its IP address and a timestamp, together known as the node descriptor. The amount of nodes whose information will be stored in a view is significantly smaller than the overall size of the cluster. When deciding what node to send messages to the node will randomly select a node descriptor from its partial view. In Montresor's algorithm below, we can see that on initialization the node will begin a continuous loop of this random selection to send REQUEST messages.



Algorithm 3: General scheme executed by p:

```
on initialization
    $state$ is initialized
    set timeout $\Delta$
on timeout
    $q \leftarrow$ selectNeighbor()
    $msg \leftarrow$ prepareRequest($state, q$)
    send $\langle$REQUEST, $msg, p\rangle$ to $q$
    set timeout $\Delta$
on receive $\langle$REQUEST, $req, q\rangle$
    $rep \leftarrow$ prepareReply($state, req, q$)
    send $\langle$REPLY, $rep, p\rangle$ to $q$
    $state \leftarrow$ mergeRequest($state, req, q$)
on receive $\langle$REPLY, $rep, q\rangle$
    $state \leftarrow$ mergeReply($state, rep, q$)
```

Fig 1. General algorithm scheme proposed by Montresor

Upon receiving a REQUEST message, the node will send a REPLY back with its partial view before merging the two partial views together. If this new partial view is much larger than the allocated limit, the oldest information in the list gets kicked out. When a node receives a REPLY message it will merge the received partial view with its own, kicking out the oldest information if the limit is exceeded. Thus every time a connection is established, or a message is sent, both partial views are shared and updated.

Features of a Distributed System

The most important features of a Distributed System are fault tolerance, performance, scalability, consistency, and concurrency. Our implementation's main goal is to provide fault tolerance on node failures and packet drops. The gossip protocol concept itself takes care of these issues since every node is multicasting upon receiving a message. If a node fails, there are other nodes that are also sending the same message, ensuring that the node descriptors still reach their target destination. A failed node is quickly forgotten by the network, but once the failed node boots up again, it will make its former neighbors aware of its presence once again, ensuring consistency. The use of TCP, a connection oriented protocol, confirms that messages that are sent have also been received.

Performance is also vastly improved from *O(n)* for regular multicasting to *O(logn)* or *O(logp)* where *p* is the size of the partial view. Each node will propagate *p* messages every round, exponentially spreading the new messages. This reduces the time for full propagation to *O(logn).* The average bandwidth is calculated as the size of the partial view list multiplied by the payload size at any given time.

This implementation is designed to be primarily horizontally scalable, not vertical scalable, in keeping with the goal of our distributed system. By vertical scaling we refer to both altering transmission frequency (interval length between messages) and the frequency at which new messages are generated. Adding more nodes to the system does not decrease performance, since the amount of messages being sent out is the same, capped at the partial view size. A new *goroutine* thread is spun up for each listening and sending function. *Goroutines* are essentially lightweight threads managed by the Golang runtime that allow for easy concurrency, abstracting the work for our team. In the Go paradigm, communication between goroutines is managed by transmitting data over blocking channels rather than with mutexes and semaphores. This is one of the many advantages that came with choosing Golang as the programming language for this project.

Major design decisions

Montresor's paper and epidemic membership protocol was only published recently, in 2017, presented as a “pure” algorithm, leaving many design decisions to the programmer. This gave us a blank slate when it came to implementation decisions. We chose to implement the algorithm in Golang for two main reasons. Golang was developed by Google with the specific purpose of network programming. There are many libraries and built in functions that are advantageous for network programming. We decided to use TCP rather than UDP for reliability concerns and because it would be a smoother choice for implementing push-pull gossip within a single connection. Montresor actually demonstrates that the push-pull protocol is superior to simple push[1][2]; while our library only uses the push implementation, we plan to add push-pull as one of our immediate next steps (see **Future Works**).

Montresor does not lay out an exact system for determination of the origin node of previously received messages, nor even uniquely identifying messages, so this was another determination we made on our own. We decided to uniquely identify each message with a unique node identifier and count pair; each source node keeps a running count of how many messages (of any type) it has sent, and associates the current count to a message before sending it out. In this way, each message has a unique tuple identifier. Node identifiers are randomly generated 64-bit integers; at higher-node networks, we may need a larger identifier space, but such networks would be extremely large. Using the Birthday Paradox collision approximation, we can see that at one million nodes, the likelihood that a collision exists in the entire network is less than 0.00001% when using 64-bit identifiers.

Unfortunately vector timestamps, despite being popular in other gossip protocol schemes[3], are unable to be practically utilized with dynamic partial views, as each node's neighbors are constantly shifting. We elected to use the system clock at nanosecond precision, but plan to investigate ways in which these timestamps can be efficiently synchronized.

Montresor's exact replacement criteria are underspecified. We elected to replace descriptors only if a newer, unseen descriptor was received, in which case we would eject the oldest stored descriptor. An alternative method, which the paper seemed to be leaning towards, is to eject all descriptors once they reach a certain age, but we decided such a scheme was unnecessary overhead for our purposes. As Golang doesn't have a generalized *O(1)* access, *O(1)* modification data structure, we use a simple array, but this isn't important to the ultimate function. As our partial view descriptor sets always remain small, the *O(n)* access time is not a concern.

Finally, we did identify an alternate replacement algorithm, in which message descriptors have their timestamps refreshed whenever a node sees them for the first time. (Message descriptors, for messages intended to propagate throughout the entire network, are unique from node descriptors, which are exclusively used for membership management and partial views, and should never be refreshed.) This would ensure that each message is far more likely to reach isolated or far-away nodes. While we didn't implement this refresh scheme in our library, it would be desirable in networks in which nodes send messages at higher frequencies.

Implementations Details

There are two main features that our implementation provides, the first being dynamic membership management. This means that at any point during the creation and termination of a given node cluster, the user can expect that after ~log(n) (where n is the number of nodes in the cluster) intervals, the new node will be fully integrated into the cluster[1]. Secondly, our implementation provides concurrent message passing through nodes through the use of Golang channels and procedures. Each node will use one channel and three goroutines each for managing node descriptors and message descriptors, which allows multiple nodes to send messages at the same time to the same node without any bandwidth issues.

Our implementation of the algorithm was based off of Montresor's pure algorithm. Each node is identical to the next in terms of implementation, meaning there is no specially designated host node.

When the first node is initialized it begins with an empty Partial View and begins listening for any incoming connections. It also begins sending keepAlive messages to all the nodes in it's Partial View, currently empty, besides its own information. The second node in the cluster is initialized with the address of the first node, added to its Partial View.

The second cluster then begins transmitting its Partial View to the first node in the cluster, which contains the address of the second node and the first node. Upon receiving the Partial View from the second node, the first node merges that Partial View with its own. Now both nodes have each other's information, a connection is established, and messages can now be sent back and forth to each other. Whenever a new node joins the cluster, its information will be propagated throughout the cluster, and each node will continuously be swapping and updating its Partial View ensuring that every node has a path to another node in the cluster.

We also implemented some customizability in the library. We allowed the Partial View size to be determined by the user. The optimal size of the Partial View to ensure peak performance is dependent on the size of the cluster. We also allowed the ability for the user to pass generic strings between nodes.

Testing Practices

We decided to use Amazon EC2 servers due to our previous experience with them and the ease of setup that they provided. We each created between two to five server instances under the same security group so that connections between them would not be blocked. We would then use these instances for the following tests.

We ran a variety of different tests in order to properly assess how well our library worked. In order to do this, we limited the number of intervals our nodes would run, and tested various variables under different conditions. The first basic test we did was to get two nodes to talk to each other successfully. Once we got that to work, we moved on to larger node cluster sizes, such as three and five, and eventually ten. At this point when we got this to work, we were able to verify that the nodes were connected but not that they were connected correctly. In order to test for this, we used the generic message passing system as a visual representation of data being passed throughout the cluster. We began with two nodes again, and worked our way up until we again were able to correctly pass messages among our group of ten nodes. Each of these tests also included testing for fault tolerance and bandwidth. For fault tolerance, we stopped a variable amount of nodes early to simulate crashes. We also had nodes join late to make sure they properly integrated into the ongoing network. For testing bandwidth, we measured memory and CPU usage on some of the nodes as they were running in order to make sure there would not be any vertical scaling problems.

Results, Discussion, and Conclusions

While we had no issues scaling our network to ten nodes, this is a great distance from the thousands of nodes that such a network is intended for. Unfortunately, we did not have the ability to scale horizontally (by adding more nodes) on AWS, as running even ten EC2 instances was already stretching our resources in that respect. If we wished to test horizontal scale further, we would have to employ an approach using many processes or threads running on a single machine, which would defeat the purpose of our networked library. We can estimate that performance at horizontal scale would not be impacted, as the average messages a cluster sends and receives are scale proportionately to the frequency at which requests are transmitted and the packet size:

$$processing\ time \sim transmission\ frequency \times messages\ per\ interval \times packet\ size$$ [2]

The average messages sent will always be the same as the average messages received, as networks are entirely self-contained.

Instead of horizontal scaling, we tested vertical scaling, both in altering transmission frequency (interval length) and the frequency at which new messages are generated. We found the former to display a roughly linear scaling on CPU usage, and the latter to have no impact on performance. As our nodes were each an Amazon t2.micro EC2, none were particularly high power, and they didn't start to struggle until they were sending a thousand messages per second.

One issue we did run into is that at small partial views, there is a chance of a network split, in which two subsets of the cluster no longer have connections between them. This chance is exacerbated upon node failures, especially in high numbers. With larger partial views, this chance essentially disappears: Montresor writes that at 80 nodes in a partial view, the likelihood is negligible even for hundreds of thousands of nodes in a single cluster and high failure rates[1].

Conclusions and Future Works

We have many improvements we would like to add in the future. As this is intended to be a general library useful to many users, most of these are aimed at ease of use and performance improvements. To begin with, we view customized callbacks as an absolute necessity. Users would be able to pass a function variable (similar to a C function pointer) to the Gossip Client, and it would be run on every newly received message.

The next immediate improvement would be to fully implement the push-pull gossip schema recommended by Montresor. While the expected number of transmission rounds to reach all nodes is still *O(log n)*, push-pull is faster in practice, as it retains the benefits of push (transmission begins quickly) and pull (transmission converges quickly at *O(log log n)*) [2]. We anticipate implementing this improvement to be largely trivial.

At scale, and when running for extended periods, timestamp synchronization becomes far more important. While other gossip schemes have used vector timestamps to great effect[3], this is not appropriate for our implementation. We have yet to identify a satisfactory protocol, as disseminating timestamp synchronization becomes a dissemination in and of itself and is likely to result in drift, but we dislike the idea of relying on a central source for time info. One solution we wanted to investigate further is electing a smaller group of leader nodes that would coordinate amongst themselves and disseminate time in that way.

The final big improvement we would like to consider is to implement a system of persistent storage and automatic reconnections, so that Gossip Clients could manage reconnection to a network on their own, without users having to implement that logic from scratch.

References

[1] A. Montresor, “Gossip and Epidemic Protocols,” in *Wiley Encyclopedia of Electrical and Electronics Engineering*, American Cancer Society, 2017, pp. 1–15.

[2] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, “Randomized rumor spreading,” *Proceedings 41st Annual Symposium on Foundations of Computer Science, Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pp. 565–574, Jan. 2000, doi: 10.1109/SFCS.2000.892324.

[3] Z. Xiang and N. H. Vaidya, “Timestamps for Partial Replication,” 2016.