Performance Analysis of gogossip

Purpose

The purpose of this report is to analyze the convergence time needed for a varying number of nodes in a synchronous network, comparing the three gossip protocols: push, pull, and push-pull.

Background

Gossip is a protocol to spread information through a network. For the remainder of this report, a node that has received the information will be named “infected,” and a node that has yet to receive the information will be named “susceptible.” It involves communication between nodes within the network, with two possible actions: a “push” from an infected node to any other node, to attempt to infect it, and a “pull” from a susceptible node on any other node, to attempt to retrieve infection from it.

The push protocol involves only this push action: each round, every infected node will pick another random node to infect. Picking the target node does not consider whether it is susceptible, but the push will only make a difference if it is susceptible. This continues until every node is infected.

The pull protocol involves only the pull action: each round, every susceptible node will pick another random node to pull from. Again, it does not consider whether the target is infected, but the pull will only matter if a node pulls from an infected node. This continues until every node is infected.

Finally, the push-pull protocol combines both actions in each round. First, in the “push” phase, each infected node pushes to another random node. Then, each still susceptible node will pull infection from another random node.

These three protocols all have the same big-O convergence time. However, we would like to see the real-world differences between these protocols.

Procedures

To run all the benchmarks and collect the data, we wrote a function to test many different configurations and print the results in a table that can be easily imported elsewhere. The configurations include varying network types (asynchronous, synchronous with leader, and synchronous leaderless), protocols (push, pull, push-pull), number of nodes, and number of infected nodes. Each configuration is tested three times to minimize the effect of random error.

To replicate the benchmarks, simply run `go run . bench`.

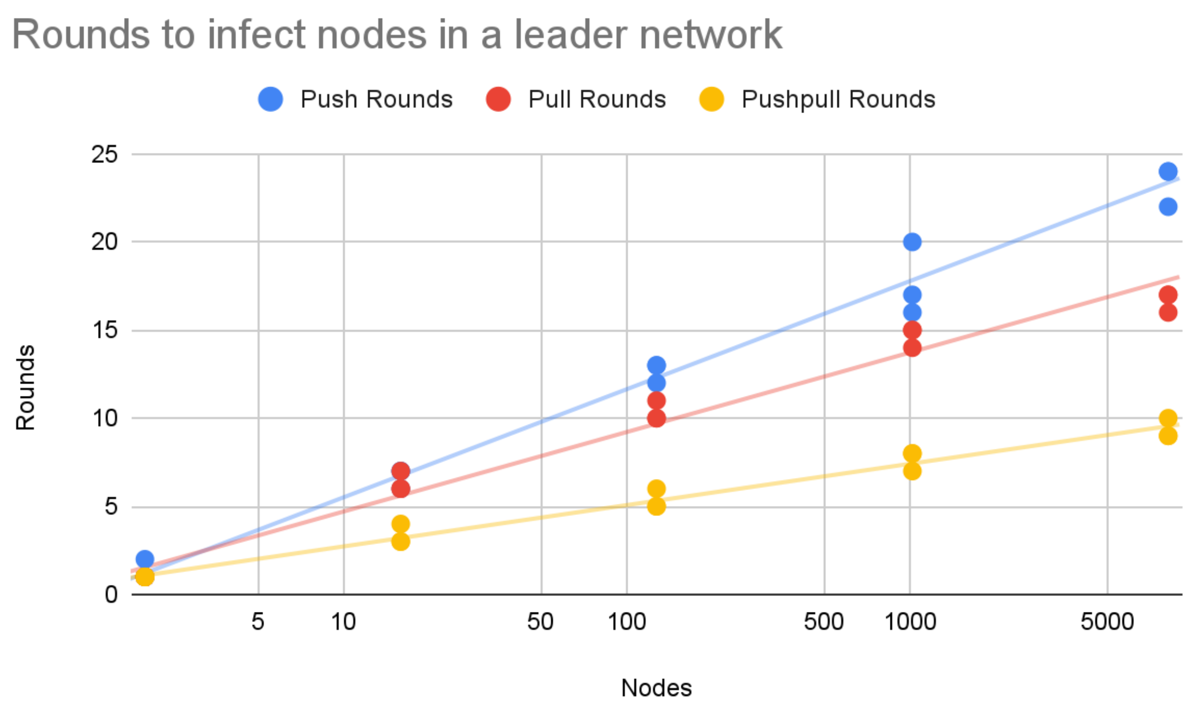
Memory Analysis

In gogossip, we create a network of N nodes. Each node has basic configuration information, as well as channels for communication. The channels are of fixed size and do not vary with the number of nodes, so gogossip has a memory complexity of O(N).

Time Analysis

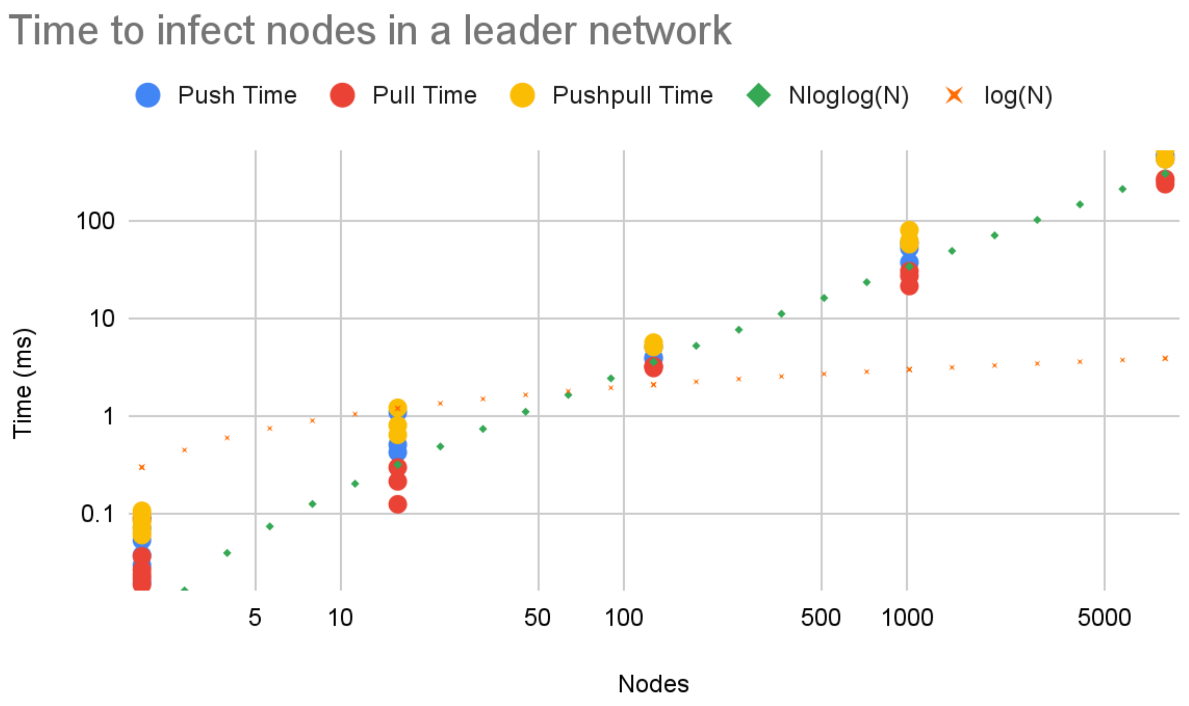
The raw data tables can be accessed at:
https://docs.google.com/spreadsheets/u/1/d/e/2PACX-1vQICkBHnltuPdcpUCj1iQscq8vtjCGssDLqYaa-STOwk0QPNCjQ24lgf0cEzV26lxbA8Gx3w_mTsfF3/pubhtml?gid=0&single=true.

When looking at the rounds needed to infect N nodes in a leader-based synchronous network, we see a clear logarithmic trend:



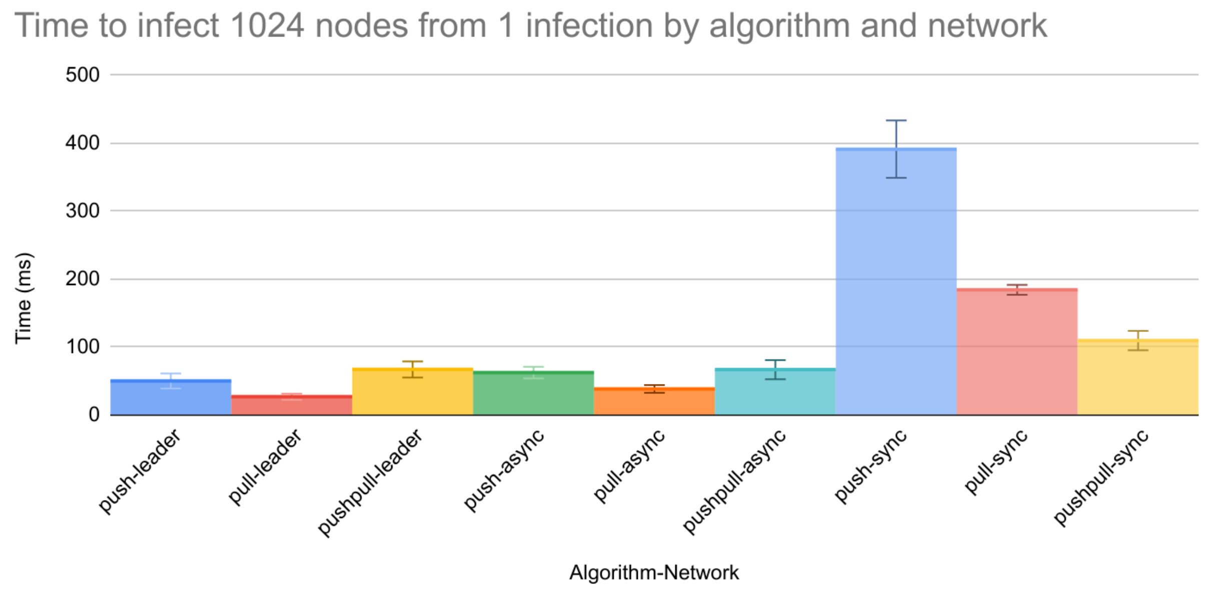
The horizontal axis is using a logarithmic scale, so the trendline appears linear. This more clearly shows the different number of nodes we tested with (2, 16, 128, 1024. 8192). As the number of nodes increases, the number of more rounds needed decreases. This follows the O(log N) complexity that we would expect.

However, when examining the time needed to infect the same configurations, the data no longer appears to follow a logarithmic function:



When we plot the data on logarithmic scales for both the horizontal and vertical axes, the data appear linear. Thus, it seems like the data follow O(N log log N), and plotting this, it indeed predicts the data with greater accuracy. It seems like as the number of nodes increased, the time needed for each round increased exponentially. We believe this is because of the number of goroutines waiting to read or write values from/to the various channels, limited by the number of goroutines that can run concurrently, defined by GOMAXPROCS and the number of threads on the computer.

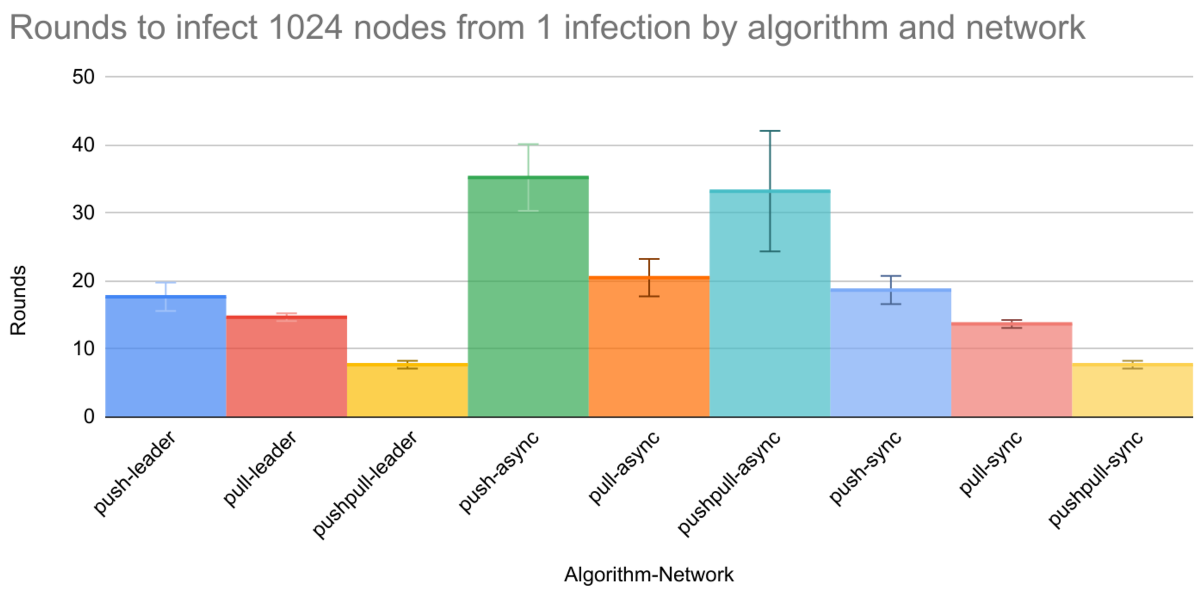
We also compared the various protocols and network types:



The pull-based gossip in the leader network was consistently the fastest compared to the other protocols and networks, with the pull gossip in an asynchronous network close behind. The leader-based pull gossip protocol was the simplest and most efficient in the code as well, because it does not require a separate goroutine to write and/or read values in the background. The pull runs in constant time: write current infection status to the channel, read another node's infection status, and write that infection status back to the channel in case another node selected the same target.

The synchronous, leader-less gossips were the slowest by a significant margin, which is likely because we had to create a stronger WaitGroup to better handle concurrency with each goroutine waiting at different times. This allows us to synchronize the goroutines at the cost of performance.

The push and push-pull protocols were slower than the pull protocols. We expected the push-pull to be the fastest, but with 1024 nodes this was not the case. However, this was likely because as the number of nodes increased, the time needed per round also increased. When plotting with the number of rounds, we see that push-pull has the smallest number of rounds for the synchronous networks:



This makes it clear that push-pull was not faster than pull because of time inefficiencies in the push algorithm.