CS 425 MP2 CP1 Report

I. Submission Information

Group Members: Jianfeng Xia(jxia11), Cheng Hu(chenghu3)
VM Cluster Number: 10
GitLab Repository: https://gitlab.engr.illinois.edu/chenghu3/cs425_mp2

Instructions:

- To build: `go build client.go`
- To run: `./client [PORT]`
- Alternatively, use our Python3 script `experiment.py` to start multiple clients (starting a node every 0.5s) and write stdout to log files.
  Usage of Python script: `python36 experiment.py [NUMBER OF CLIENTS]`
- We have also included scripts to analyze performance and generate plots. To run the scripts:

  ```
  cd logs
  python3 propagation_plot.py
  python3 bandwidth_plot.py
  ```

  Note:
  1. Our plot scripts use matplotlib and numpy, we suggest to run those scripts on machine has those library.
  2. Because the log files are large(we use logs in the case of 100 nodes, 20 mesg/s), please be patient when running scripts.

matplot

- Revision to be graded:
  7c104aa3fa9b8ea4ce9106deba2516e7ce3da8c7

II. Algorithms & System Design

Connectivity

For this assignment, we maintained a full membership list in each node in the system. We developed the following three components to maintain the membership list.

- **Node Join**: After a new node joins the system and receives an INTRODUCE message from the introduction service, it sends a “HELLO” message followed by its server port for receiving messages. For example:
  `HELLO 9000`
  The node receiving the HELLO message sends the full list of membership back. Then, it gossips a message to notify other nodes in the system of the existence of the newly joined node.

- **Failure Detection**: We implemented a simplified SWIM-style failure detection mechanism. That is, for each time interval *T*, each node randomly selects a node in its membership list and tries to establish a TCP connection. If the connection is refused, the other node is considered failed.
  Note: instead of the PING, ACK, and PING-REQ used in the original SWIM paper, we just used a simple TCP dial because: 1. In our experiments, we found that TCP dial is a relatively reliable way to detect failure. 2. Occasional false positives are allowed because our goal is not to keep complete membership lists, but is to keep connectivity and allow efficient message propagation.
- **Dissemination of Failure Updates**: We used the infection-style dissemination described in the SWIM paper. If a node successfully established the TCP connection, it then sends updates (detected node failures + received failure messages) via the TCP connection. The message format is as follows:
  `DEAD [IP 1] [PORT 1],[IP 2] [PORT 2],...`
  As in the original SWIM paper, each node maintains a buffer of recent membership updates and a local count for each buffer elements. The local count specifies the number of times the element has been sent. It is used to prioritize elements that has been sent fewer numbers of times when selecting elements to send. Each element is sent at most a constant number of times by each node.
  We only send at most a constant N number of membership updates at a time (currently N=10). Therefore, this failure detection and dissemination mechanism only adds a constant bandwidth load to each node and is able to scale to handle large amounts of simultaneous node failures (as in the case of “thanos”).

Transaction Broadcast

We used gossip to broadcast transactions. For this check point, we only used push-based gossip. We experimented with two styles of gossip:

1. **A separate gossip goroutine for each message**(function `sendGossipingMsg`). On receiving a gossip message such as `JOIN` and `TRANSACTION`, the node *immediately* start the goroutine `sendGossipingMsg` to start gossiping that *single message*. Each node gossips the message for `round` rounds. Each round, the node randomly selects 2 nodes in its membership list and sends the gossip message, then closes the connection. However, in our experiments, we found that this approach is relatively expensive, because it requires the node to start a goroutine for each new incoming message and constantly dial and close TCP connections, causing a lot of overhead. Although we were able to achieve 100% delivery, there were very few messages that take longer times to reach all nodes. Therefore, **we switched to the second approach.**
2. **In our final submission, we used the same dissemination mechanism described in the previous section for transaction messages.** In the `ping` function, after sending the membership updates, each node also sends the buffered transactions. We created a message buffer with local count for transaction messages as well to ensure that each message is sent by a node no more than a constant N number of times (N=25 in this

case). However, this time we send all the messages in the buffer at once, instead of sending only a constant sized subset of them. The purpose of this change is to ensure fast and reliable message delivery at the cost of higher (but affordable) bandwidth. In our experiments, this approach is more scalable, uses less bandwidth and the message delivery times are more consistent.

Logging & Evaluation

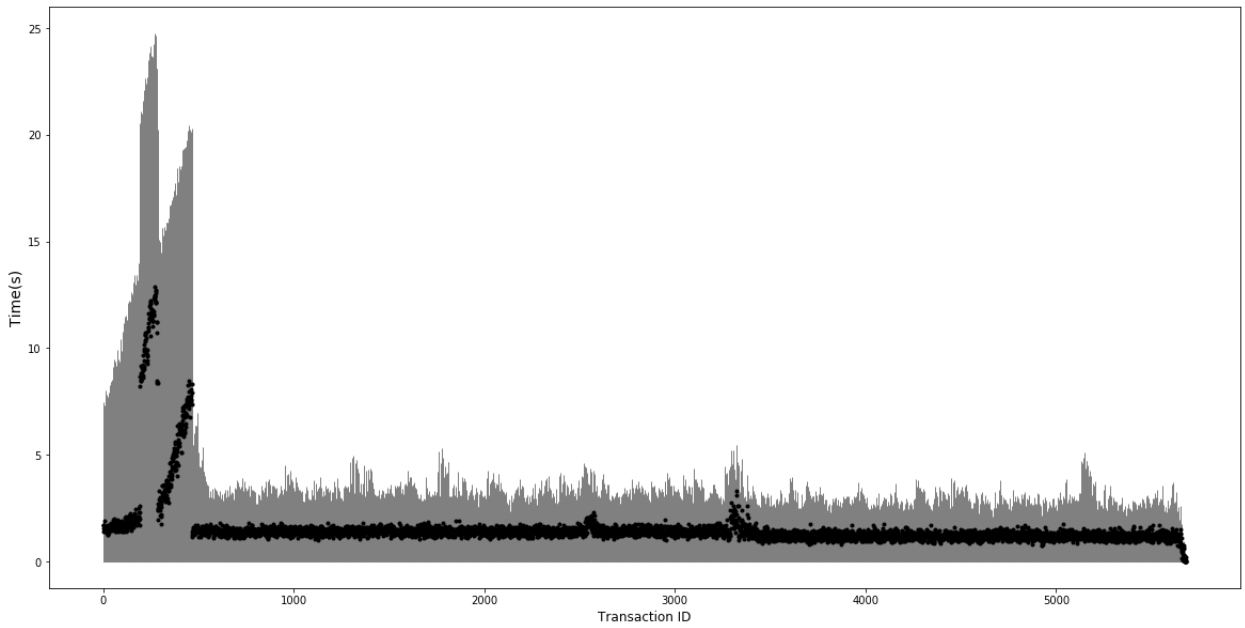
In this section, our plots are generated using data from an experiment with 100 nodes initially and 20 transactions per second. After the system runs for a while, a `thanos` command was sent and half of the nodes died.

Propagation Delay

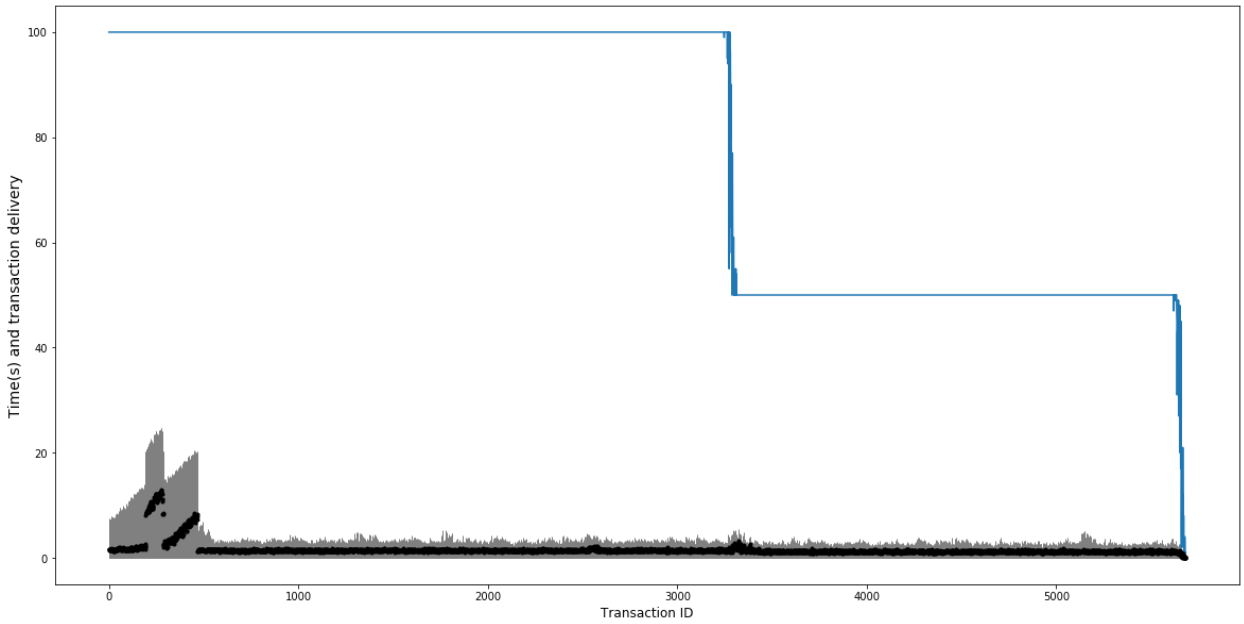
To calculate propagation delay, we logged the timestamp and the transaction message when a node first receives a transaction. Here is an example message:

```
LOG 2019-03-14 21:09:39.893160 TRANSACTION 1552615779.890393
bd6552fc14403304167359 85e057720b 68070 51642 651
```

Then, we wrote a Python script to process the log files and get the propagation delays for each message. We calculated and visualized the minimum, maximum, average and median of the propagation delays, along with the number of nodes reached for each message:



Propagation delay of each transaction

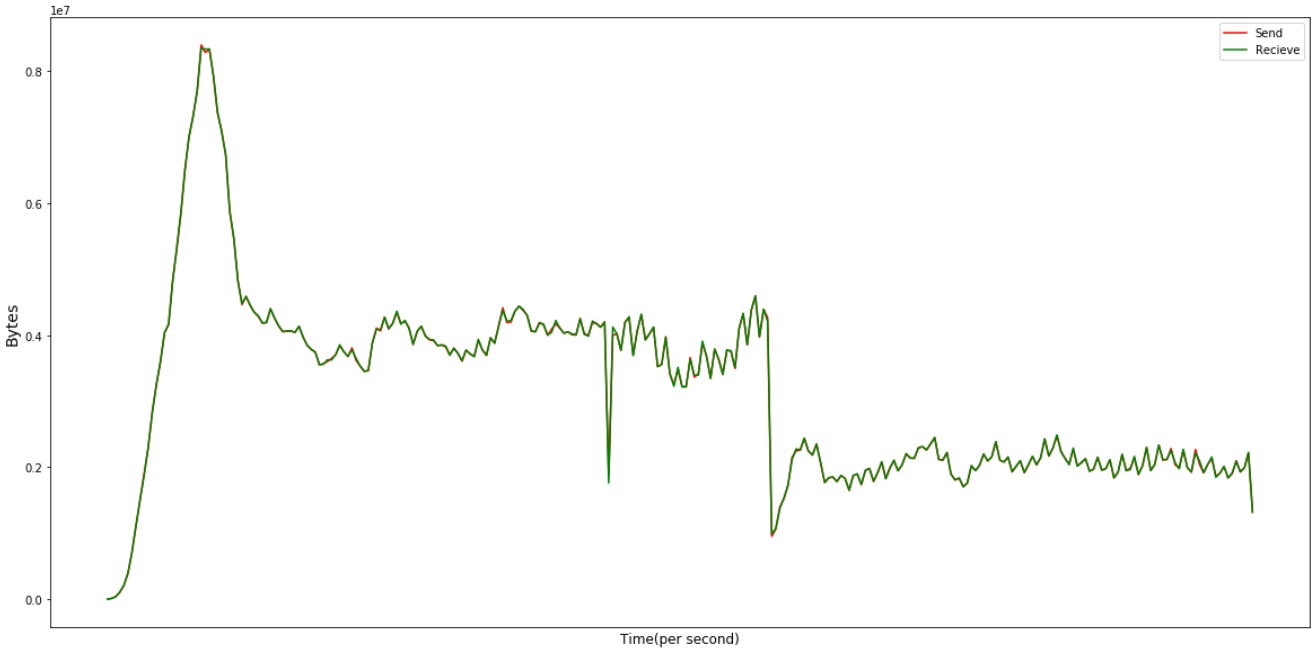


Propagation delay and delivery count of each transaction

The grey area is the range of propagation delays, and the black dots are medians of propagation delays. The blue line indicates the delivery count.
It is clear from the graph that we have achieved **100%** delivery and consistent, short propagation delays. The longer delays at the start are due to node joins happening after the transactions are created.

Bandwidth

We logged the number of bytes for each message and calculated the bytes per second to measure bandwidth. The experiment results are as follows:



Aggregated bandwidth of all nodes(per second)

Note that the bandwidth numbers are not important here, because we can adjust the parameters such as gossip intervals, gossip rounds for each message to easily reduce the bandwidth. In general, there are trade offs between bandwidth and reliable delivery/propagation delays.

So, although the system already works well with 100 nodes and 20 msg/sec, here are the things we could do to improve the performance *if performance is our primary concern:*

Possible performance improvements:

- Encode/compress the messages instead of using plain text to reduce bandwidth.
- Tune the parameters such as gossip intervals/rounds of gossip for each message, to find a sweet spot such that the system uses least bandwidth to ensure 100% delivery and short propagation delays.
- Use the same dissemination approach for node join messages as well. Currently we use the first gossip approach described in the Transaction Broadcast section for node join updates. In our experiments, each node join is separated by 0.5 seconds. If we want to handle large amounts of simultaneous joins, we can also use the dissemination approach for failure updates, at the cost of a slower membership lists updates.