CS 550 - Spring 2022
Isaias Rivera
A20442116

P2P File Sharing - Consistency - Tests

A special binary was used which does not include the interactive terminal and helps automate some of the testing. This *test peer* does the following.

- Files are randomly generated where it is ensured each peer has their own unique files.
- Active peers will make requests for files and randomly delete local files.
- After requesting a file, active test peers will record the percentage of invalid files that the request found before it was returned onto a csv file.
- Non active peers will simply randomly make modifications to their own files, invalidating files

Due to the overwhelming number of requests to the filesystem, I had to put a delay between sequential calls to request/delete files. Otherwise, requests seemed to hang and timeout. However, calls between peers, when they happen, are still concurrent. In addition to that, one issue with the design is that the directory the program is watching will auto index files even if they are in the middle of downloading, this means files that take too long to download will actually be indexed in this half downloaded state. There are checks at the time of peer to peer connection to ensure peers actually have the file and this is only really an issue with larger files, but ultimately these requests will fail as the file no longer exists by the time a request is made. Regardless, the indexing server still processes the requests by peers and peers should be ready when this happens.

In every test, each peer aims to run 100 calls, only the actual requesting of a file from super-peers counts towards this. For each client, the response time for each request of this type is stored to a csv. The binaries used are compiled as release, meaning it should run as efficient as they can, however, this also means any issues I ran into were not easily debuggable.

Communication between peer to peer and peer to super-peer indexing server have timeouts, meaning, if there is sufficient traffic it is possible that requests might fail. I did not get around to implementing some method to externally catch and log these errors.

Furthermode, due to time constraints, the TTR value that is used when in pulling mode was very small compared to what the PA suggested. I tested using TTRs of 5s, 15s, and 30s.

Local Tests

These tests were run locally on the same windows machine, for both peers and super-peers.

10 Super-peers and 19 Leaf-nodes

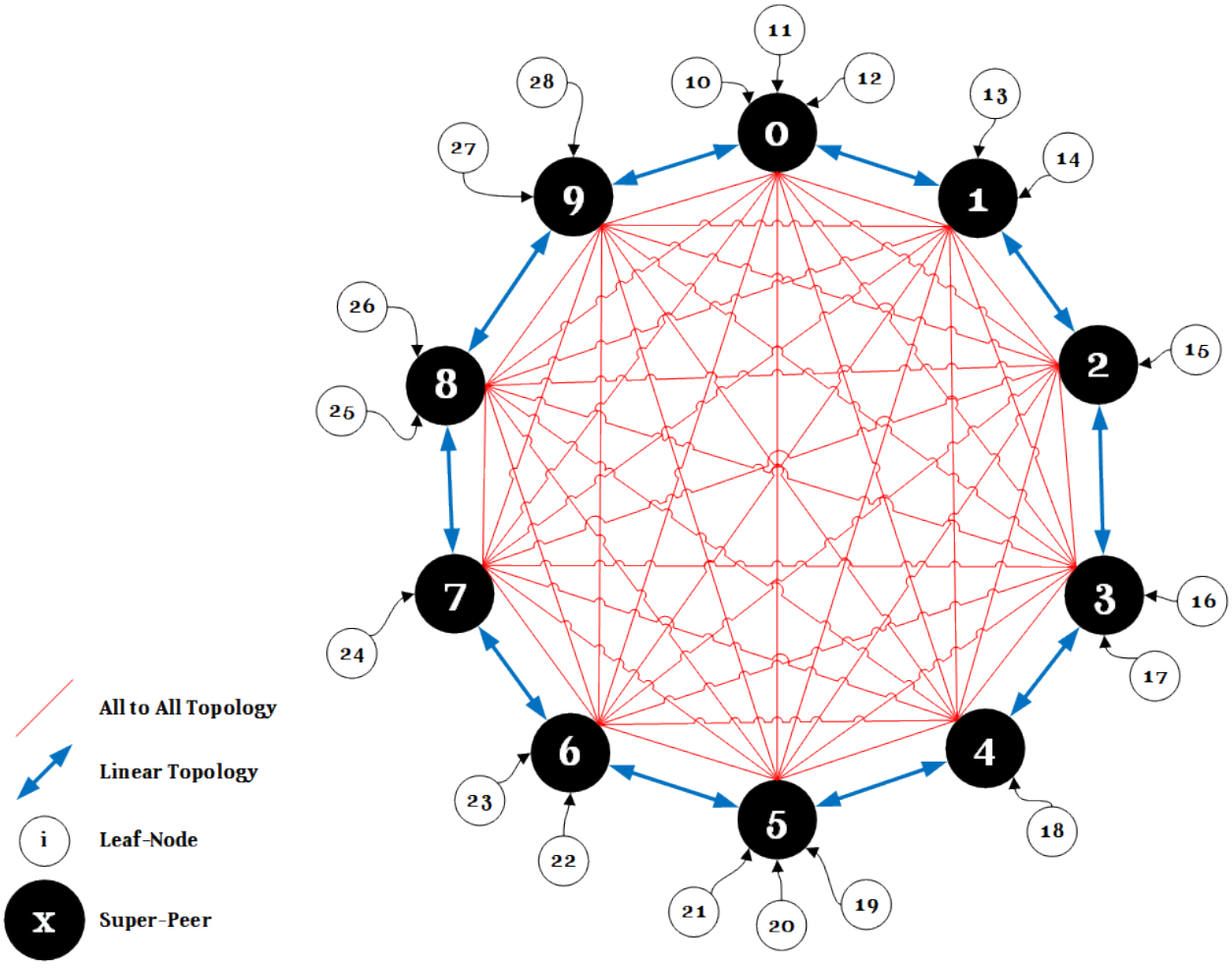
These tests were mostly automated using a python script which read the csv files generated by the test peers and then compiled it into averages. I then took these averages and graphed them in Excel.

This test concurrently ran 29 test peers where 10 of them were super peers. Only N number of these peers are actively making requests, where N ranged from 1 to 15.

This test is performed both in pushing and pulling mode and both the linear and all to all topology, where pulling mode tests also varied the TTR by 5s, 15s, and 30s.

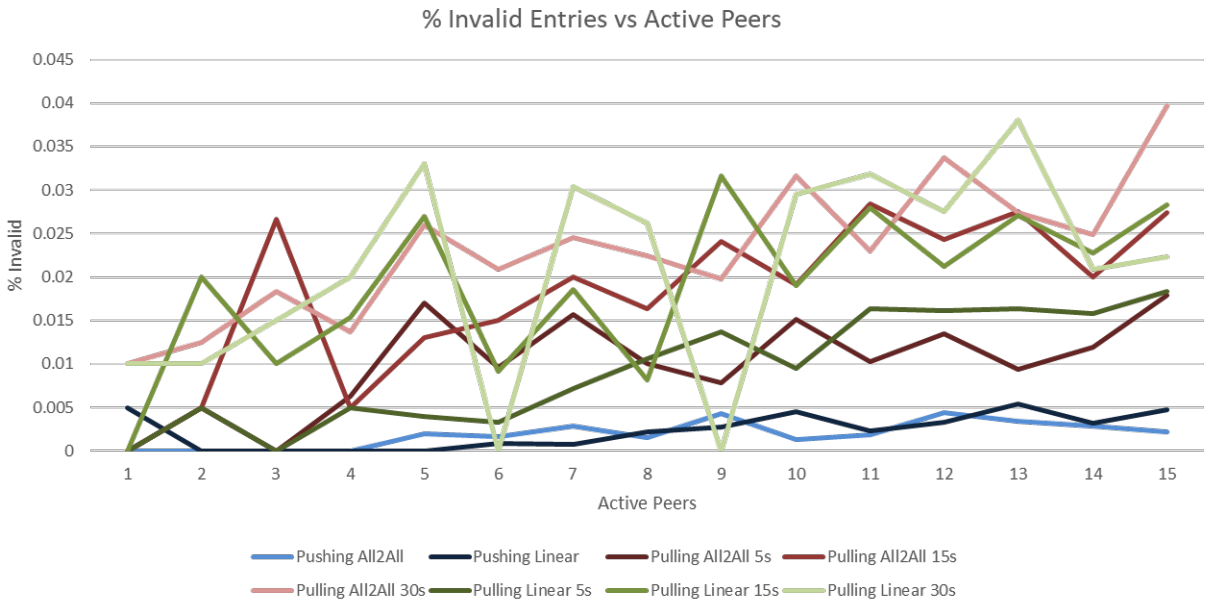
Topology setup

This is a graphic which visualizes how each client is setup using the static configuration.



Results

The following graphs show all the tests



The tests shown were very close to zero percent invalid files found when requesting a file. I believe this was the case as with so many unique files (10 files per peer * 29 peers = 290 unique files) and only requesting a total of 100 files, it is very possible that peers were almost always resorting to connecting to the origin server to pull a file, with no other peers having the file entry. Additionally, because of time constraints, I was unable to run any additional tests. I believe another issue may have been how I collected the data. Unfortunately I ran out of time for testing and was unable to do any further investigation.

Regardless, the application does seem to be behaving as expected, it is just the actual data collection of this metric is an issue. Additionally, the graphs do show a trend of a higher invalidation as the number of active peers increases.

Pushing invalidations seems like the easiest to implement, it also seems to maintain consistency at a higher degree compared to pulling. However, it does create more traffic and it requires, at a minimum, calling every super-peer whenever any modification to any file is made.

Pulling is a bit more complex to implement and requires more work on the client side. However, this does reduce the stress on the network and, in practice, would allows origin server to tell peers how often they should update each file. This would be useful if a server knows, generally, how often certain files take to update.