

# CS 6685 Programming HW 2, Option 1

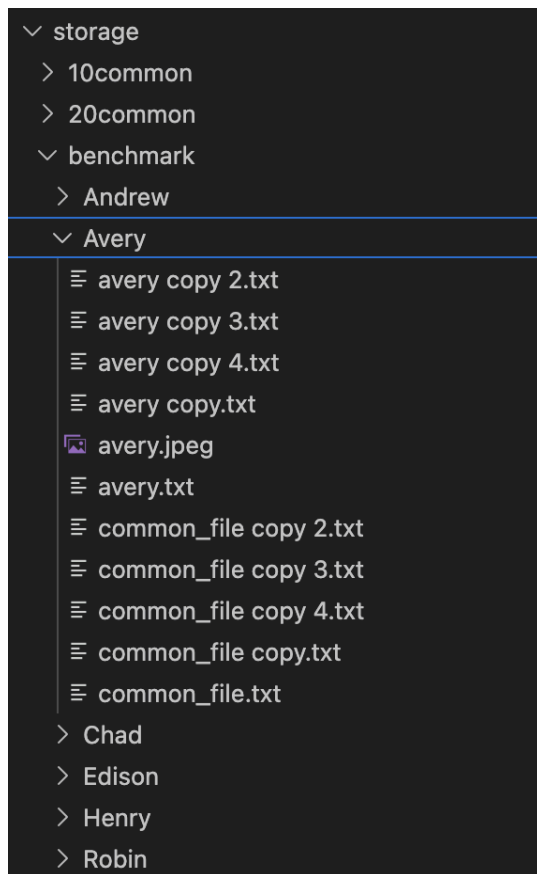
Andrew Hsu

Github: [https://github.com/Andrew920528/p2p\\_playground](https://github.com/Andrew920528/p2p_playground)

## Tests, results, and discussion

Below are screenshots and explanations of all the tests that can be run in my toy p2p system. Please refer to [README.md](#) in homework submission to understand the message protocol of this system.

In my toy system, I created 6 peers: Andrew, Edison, Henry, Avery, Chad, Robin. In all the test cases, Andrew is the sender of the message. Each node has a folder that mimics their “local storage”, as shown in this screenshot.



The storage/benchmark folder is the baseline files. 10common and 20common are for additional comparison. Each member in the benchmark folder has an image file, 5 “unique” files, and 5 “common” files.

## setup & shutdownAll

```
async function main() {  
  const verbose = true;  
  const peers = await setup(verbose);  
  
  await shutdownAll(peers);  
}
```

```
[shutdown] all peers stopped  
(base) → HW_2 git:(main) x node run.js  
[Andrew] started as 12D3KooWAhcKEJjZuPzCwaKdWxmYZjb75VZusptHVox7F8mTRPq3  
[Avery] started as 12D3KooWK7sBxfA1RAhRXoHJAP85HEf3DQYGosYkPX9bCDfKxXFx  
[Robin] started as 12D3KooWDKQEAXsZLjwYLHZmaToykz2z78eeRcXNg5hhhUxtVkyv  
[Henry] started as 12D3KooWJWxu1836b54WytCsS1kn5ax9jRZXRHUeh68RgRwo7QJb  
[Chad] started as 12D3KooWFQxbTkXkSWsd7bYAd3N4sZb17LHKfA6JRKDqBdKTA3fL  
[Edison] started as 12D3KooWCg7FDxV4CNgeHbT9FnBnuAx5MzZugvQH34jVyDcYeZMr  
[connect] Andrew → Avery  
[connect] Andrew → Robin  
[connect] Andrew → Henry  
[connect] Andrew → Chad  
[connect] Andrew → Edison  
[connect] Avery → Robin  
[connect] Avery → Henry  
[connect] Avery → Chad  
[connect] Avery → Edison  
[connect] Robin → Henry  
[connect] Robin → Chad  
[connect] Robin → Edison  
[connect] Henry → Chad  
[connect] Henry → Edison  
[connect] Chad → Edison  
[connect] done  
[setup] done  
[Andrew] stopped  
[Avery] stopped  
[Robin] stopped  
[Henry] stopped  
[Chad] stopped  
[Edison] stopped  
[shutdown] all peers stopped
```

The setup function creates the peers and connects them together. The shutdown function terminates the program. Change verbose to false to hide the messages.

## testBroadcast

```
async function main() {  
  const verbose = true;  
  const peers = await setup(true);  
  await testBroadcast(peers);  
  
  await shutdownAll(peers);  
}
```

```
[connect] done  
[setup] done  
[Andrew] sent: {"type":"broadcast","request_id":"0cb4e902-b509-4f67-835c-5ec39052f737","timestamp":1771023928258,"origin_id":"12D3KooWHfTEppLkaAJeFnskbPpeCVekWhgzrzaNWxe21yy35mzw","origin_name":"Andrew","sender_id":"12D3KooWHfTEppLkaAJeFnskbPpeCVekWhgzrzaNWxe21yy35mzw","sender_name":"Andrew","msg":"Hello friends"}  
[Andrew] stopped  
[Robin] received from Andrew: Type = broadcast  
[Robin] broadcast: Hello friends  
[Henry] received from Andrew: Type = broadcast  
[Henry] broadcast: Hello friends  
[Chad] received from Andrew: Type = broadcast  
[Chad] broadcast: Hello friends  
[Edison] received from Andrew: Type = broadcast  
[Edison] broadcast: Hello friends  
[Avery] stopped
```

When a message is broadcasted, all nearby peers will receive it and print the message in the console.

## testBasicSearch & testNoneExistFile

```
async function main() {
  const verbose = true;
  const peers = await setup(true);
  // await testBroadcast(peers);
  await testBasicSearch(peers);
  await testNoneExistFile(peers);

  await shutdownAll(peers);
}
```

These are basic functionalities that showcase a basic p2psearch, and the behavior when the file does not exist.

```
[Andrew] sent: {"type":"search","request_id":"3353325d-8694-4f9a-8a6c-d65facc62ce","timestamp":1771024097052,"origin_name":"Andrew","sender_id":"12D3KooWPZBRwKs3c5cQBwtEu6KR2ZWoENzcdfpA9iWgqt6mXm2Q","testCase":"benchmark"}
[Robin] received from Andrew: Type = search
[Robin] sent: {"type":"search_result","request_id":"3353325d-8694-4f9a-8a6c-d65facc62ce","timestamp":1771024097052,"origin_name":"Robin","sender_id":"12D3KooWDe4N4ySNzY3wxcZzL7Y9VqD5wzAjwYhB4fNByp43TsVx","responder_id":"12D3KooWDe4N4ySNzY3wxcZzL7Y9VqD5wzAjwYhB4fNByp43TsVx","files":["robin.jpeg"],"testCase":"benchmark"}
[Henry] received from Andrew: Type = search
[Chad] received from Andrew: Type = search
[Edison] received from Andrew: Type = search
[Andrew] received from Robin: Type = search_result
[Andrew] received search result from Robin: robin.jpeg
[Avery] received from Robin: Type = search_result
[Henry] received from Robin: Type = search_result
[Chad] received from Robin: Type = search_result
[Edison] received from Robin: Type = search_result
--- Benchmark results ---
Total keywords queried: 1
Total messages sent (traffic): 1
Files received: 1
[SUCCESS] Keyword "robin.jpeg" latency: 4 ms
Average latency: 4.00 ms
Throughput: 83.33 queries/sec
```

In the above picture, one node (Andrew) will send out a “search” message, and the node that has the file (Robin) will send back a “search\_result”. A callback function is called to calculate the latency.

```
[benchmark] searching 1 keywords
[Andrew] sent: {"type":"search","request_id":"013d7b56-fdfa-422WoENzcdfpA9iWgqt6mXm2Q","origin_name":"Andrew","sender_id":"12D3KooWDe4N4ySNzY3wxcZzL7Y9VqD5wzAjwYhB4fNByp43TsVx","testCase":"benchmark"}
[Robin] received from Andrew: Type = search
[Edison] received from Andrew: Type = search
[Avery] received from Andrew: Type = search
[Chad] received from Andrew: Type = search
[Henry] received from Andrew: Type = search
[benchmark] timed out
--- Benchmark results ---
Total keywords queried: 1
Total messages sent (traffic): 0
Files received: 0
[TIMEOUT] Keyword "edison.mp3" did not receive a result
Average latency: 0.00 ms
Throughput: 0.00 queries/sec
```

In the above screenshot, Andrew requests for “edison.mp3”, which does not exist. The search query timed out after 5 seconds in this test environment. Each node will only process the message once based on the request\_id.

## testSelfContains

```
async function main() {  
  const verbose = true;  
  const peers = await setup(true);  
  // await testBroadcast(peers);  
  // await testBasicSearch(peers);  
  // await testNoneExistFile(peers);  
  await testSelfContains(peers);  
  
  await shutdownAll(peers);  
}
```

An edge case I realized while testing was when the caller himself stores the file, it timed out because the peers can't find the associated file. To mitigate this, I made it so the sender of a search message looks for its local storage first before sending the message to its peers.

```
[benchmark] searching 1 keywords  
[Andrew] sent: {"type":"search_result","request_id":"b9fba1e1-f626-4ab8-W8TkcGrxjtY7kombUgjkHChqZJakxH6","origin_name":"Andrew","sender_id":"12D3KooWnyJmWzeKaNLVPW8TkcGrxjtY7kombUgjkHChqZJakxH6","responder_id":"12D3KooWnyJmWzeKaNLVPW8TkcGrxjtY7kombUgjkHChqZJakxH6"}  
[Andrew] file found locally  
{  
  uuid: 'b9fba1e1-f626-4ab8-a172-eb86bf4aca66',  
  timestamp: 1771025141654,  
  local: true  
}  
--- Benchmark results ---  
Total keywords queried: 1  
Total messages sent (traffic): 0  
Files received: 1  
[SUCCESS] Keyword "andrew.txt" latency: 2 ms  
Average latency: 2.00 ms  
Throughput: 500.00 queries/sec
```

## Testing Baseline Measurement (HW Option 1.1)

```
async function main() {
  const verbose = false;
  const peers = await setup(verbose);
  // await testBroadcast(peers);
  // await testBasicSearch(peers);
  // await testNoneExistFile(peers);
  // await testSelfContains(peers);
  await Option1_1TestBaselineMeasurement(peers);
  await shutdownAll(peers);
}
```

Because the latency of shared files and unique files are different, I tested search on 10 unique files and 10 shared files.

===== Benchmark Comparison of Option 1.1 - Baseline Measurement =====

(index)	Test	Total Time (ms)	Avg Latency (ms)	Throughput (q/s)	Traffic (msgs)	Files Received
0	'Search 10 common files in benchmark'	108	'21.60'	'46.30'	50	5
1	'Search 10 unique files in benchmark'	99	'27.20'	'101.01'	10	10

===== Per-Keyword Latency =====

(index)	Keyword	Search 10 common files in benchmark	Search 10 unique files in benchmark
0	'common_file.txt'	'22 ms'	'NaN'
1	'common_file copy.txt'	'22 ms'	'NaN'
2	'common_file copy 2.txt'	'22 ms'	'NaN'
3	'common_file copy 3.txt'	'21 ms'	'NaN'
4	'common_file copy 4.txt'	'21 ms'	'NaN'
5	'avery.txt'	'NaN'	'32 ms'
6	'avery.jpeg'	'NaN'	'32 ms'
7	'robin.jpeg'	'NaN'	'17 ms'
8	'robin.txt'	'NaN'	'17 ms'
9	'henry.jpeg'	'NaN'	'36 ms'
10	'henry.txt'	'NaN'	'35 ms'
11	'chad.txt'	'NaN'	'17 ms'
12	'chad.jpeg'	'NaN'	'17 ms'
13	'edison.txt'	'NaN'	'34 ms'
14	'edison.jpeg'	'NaN'	'35 ms'

The latency is similar, but after multiple testing I find common files usually having a higher latency. My reasoning is that when searching for common files, more search\_result msg are sent (50 compared to 10). This higher traffic causes a longer latency. One way to mitigate this is for all peers to drop a search process when the respective search\_id is marked completed.

## Testing Shared Files with Same Query (HW Option 1.2.1)

```
windson: Refactor | Explain | Generate JSDoc | X
async function main() {
  const verbose = false;
  const peers = await setup(verbose);
  // await testBroadcast(peers);
  // await testBasicSearch(peers);
  // await testNoneExistFile(peers);
  // await testSelfContains(peers);
  // await Option1_1TestBaselineMeasurement(peers);
  await Option1_2_1TestSharedFileWithSameQuery(peers);
  await shutdownAll(peers);
}
```

===== Benchmark Comparison of Option 1.2.1 - Different number of shared files with fixed query =====

(index)	Test	Total Time (ms)	Avg Latency (ms)	Throughput (q/s)	Traffic (msgs)	Files Received
0	'Search benchmark storage'	77	'18.80'	'64.94'	50	5
1	'Search 10common storage'	111	'31.40'	'45.05'	50	5
2	'Search 20common storage'	112	'32.40'	'44.64'	50	5

===== Per-Keyword Latency =====

(index)	Keyword	Search benchmark storage	Search 10common storage	Search 20common storage
0	'common_file.txt'	'19 ms'	'32 ms'	'33 ms'
1	'common_file copy.txt'	'19 ms'	'32 ms'	'32 ms'
2	'common_file copy 2.txt'	'19 ms'	'31 ms'	'32 ms'
3	'common_file copy 3.txt'	'19 ms'	'31 ms'	'33 ms'
4	'common_file copy 4.txt'	'18 ms'	'31 ms'	'32 ms'

Varying the number of files each peer is shared from 5 to 10, 20 files among 5 peers with the same 10 test queries. In general, having more files causes a higher latency. I think this is because there are more files to search through. However, perhaps because the system is too small, the result is relatively inconsistent and insignificant. There are times where 10 common storage results in higher latency than 20 common storage. Perhaps the system is too small that minor connection fluctuation causes a difference.

## Testing Files with different number of query (HW Option 1.2.1)

```
async function main() {
  const verbose = false;
  const peers = await setup(verbose);
  // await testBroadcast(peers);
  // await testBasicSearch(peers);
  // await testNoneExistFile(peers);
  // await testSelfContains(peers);
  // await Option1_1TestBaselineMeasurement(peers);
  // await Option1_2_1TestSharedFileWithSameQuery(peers);
  await Option1_2_2TestDifferentNumberOfQueries(peers);
  await shutdownAll(peers);
}
```

===== Benchmark Comparison of Option 1.2.2 - Different Number of Queries =====

(index)	Test	Total Time (ms)	Avg Latency (ms)	Throughput (q/s)	Traffic (msgs)	Files Received
0	'Search 10'	86	'22.00'	'116.28'	50	10
1	'Search 20'	177	'46.75'	'112.99'	100	20
2	'Search 40'	346	'91.15'	'57.80'	200	20

===== Per-Keyword Latency =====

(index)	Keyword	Search 10	Search 20	Search 40
0	'common_file.txt'	'23 ms'	'48 ms'	'92 ms'
1	'common_file copy.txt'	'23 ms'	'48 ms'	'93 ms'
2	'common_file copy 2.txt'	'22 ms'	'48 ms'	'92 ms'
3	'common_file copy 3.txt'	'23 ms'	'48 ms'	'92 ms'
4	'common_file copy 4.txt'	'22 ms'	'47 ms'	'92 ms'
5	'common_file copy 5.txt'	'22 ms'	'47 ms'	'92 ms'
6	'common_file copy 6.txt'	'22 ms'	'47 ms'	'91 ms'
7	'common_file copy 7.txt'	'21 ms'	'48 ms'	'91 ms'
8	'common_file copy 8.txt'	'21 ms'	'47 ms'	'92 ms'
9	'common_file copy 9.txt'	'21 ms'	'47 ms'	'92 ms'
10	'common_file copy 10.txt'	'NaN'	'47 ms'	'91 ms'
11	'common_file copy 11.txt'	'NaN'	'46 ms'	'91 ms'
12	'common_file copy 12.txt'	'NaN'	'46 ms'	'91 ms'
13	'common_file copy 13.txt'	'NaN'	'46 ms'	'90 ms'
14	'common_file copy 14.txt'	'NaN'	'47 ms'	'90 ms'
15	'common_file copy 15.txt'	'NaN'	'46 ms'	'91 ms'
16	'common_file copy 16.txt'	'NaN'	'46 ms'	'91 ms'
17	'common_file copy 17.txt'	'NaN'	'46 ms'	'90 ms'
18	'common_file copy 18.txt'	'NaN'	'45 ms'	'90 ms'
19	'common_file copy 19.txt'	'NaN'	'45 ms'	'89 ms'

This test shows a strong positive correlation with traffic (the number of messages sent) and latency. When there are more queries to handle, more messages are sent and causes the network to slow down. Throughput also decreases as more queries are introduced, possibly showing that the system is using up more resources and some queries need to be in queue.

## Conclusion and reflection

Overall, this is a solid system for gossip based messaging. I treated this project as a playground for my class project, which is a social interaction p2p based app. I think the modularized approach of libp2p makes a lot of sense to me, and I am able to build something scalable with the incorporation of OOP. With [node.js](#) being the “backend”, I can easily hook this project to a frontend using react, making it more than just a terminal based experiment.

A noteworthy point is during the development of this project, I tried to implement the traditional flooding method myself. I removed the code in the end because I think it defeats the purpose of pubsub protocol (it was less scalable and efficient). It was however a good opportunity to put what I learned in class into perspective.