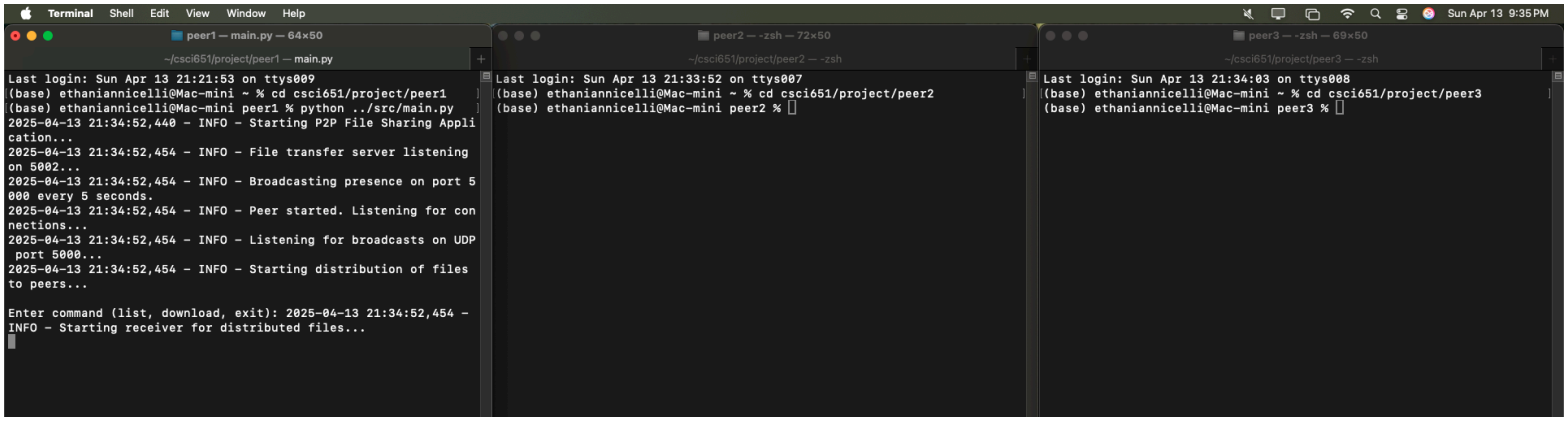
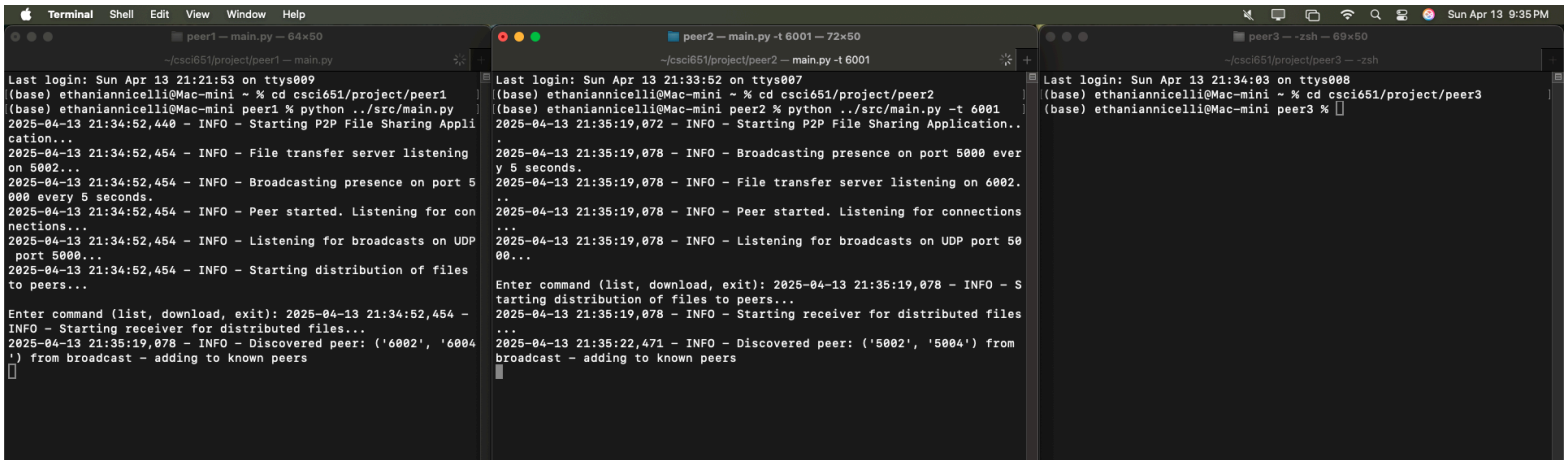
Ethan Iannicelli : eti7075@g.rit.edu

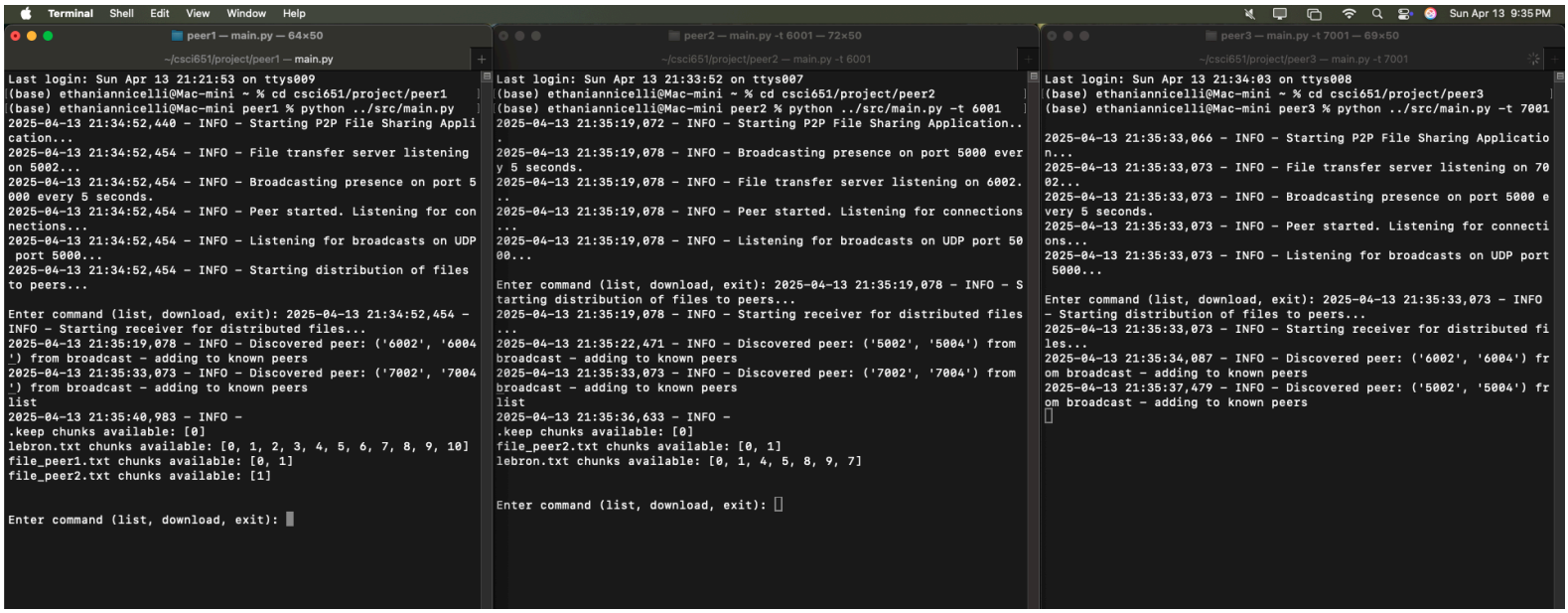
To display this project, I will be showing the status of three peers in different terminals at different points of the peers to peer connections and requests.



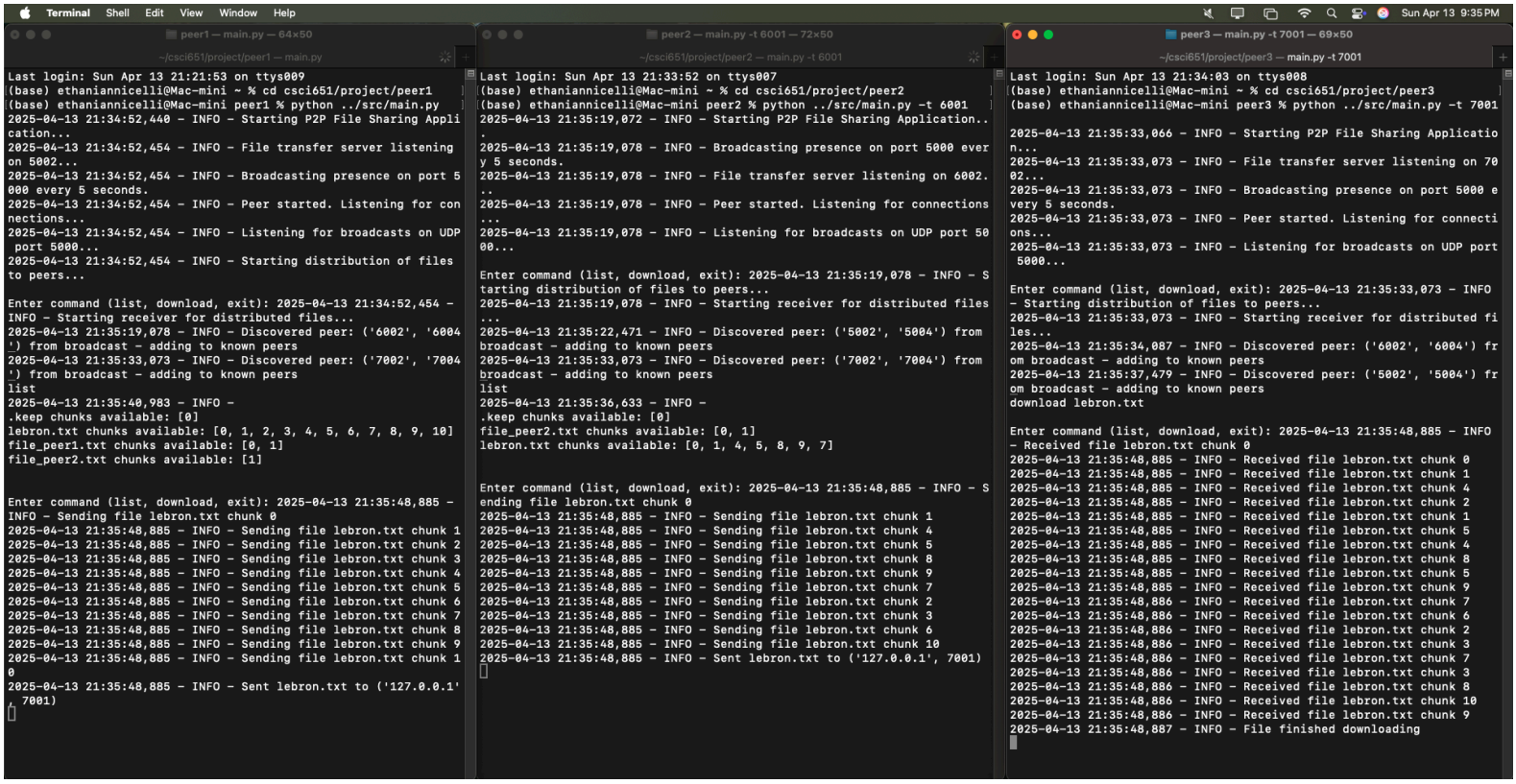
First, we connect to the peer pool with peer 1, operating on ports 5001, 2, 3, 4. We see the startup logs for the server.



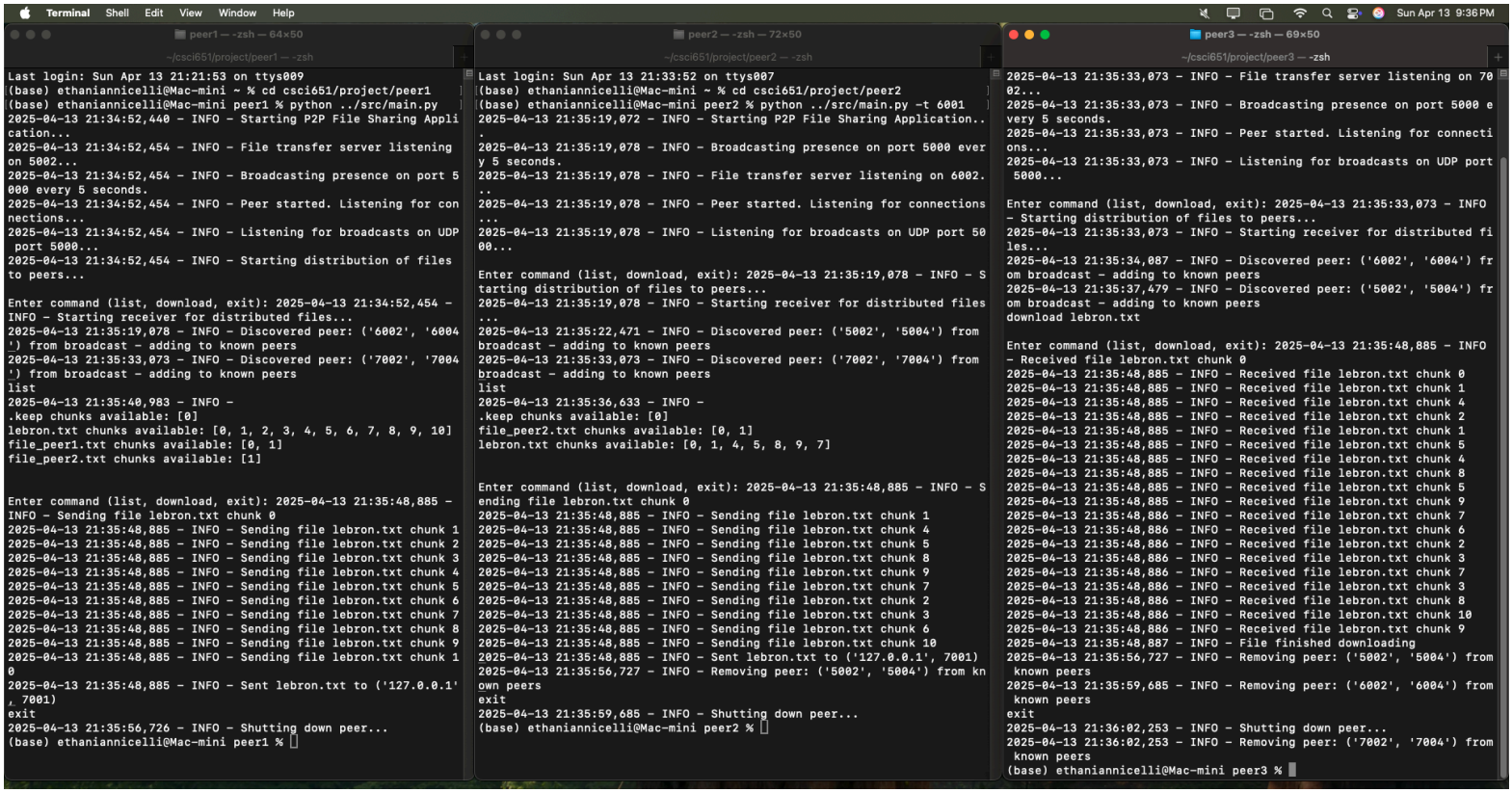
Next, we startup peer 2 operating on ports 6001, 2, 3, 4. We see the startup messages in the peer 2 terminal, and we also see that the peers have discovered each other. Note that neither peer discovered itself, we prevented this on purpose for a clearer log.



In this next image, we have done two things: first, we started peer 3 on ports 7001, 2, 3, 4. We see the peer's startup logs and that the peer has discovered the other 2 peers. Next, we performed the `list` command on both peer 1 and peer 2. We see in peer 1 that it has all chunks for `lebron.txt`, and peer 2 has only *sum* of the chunks for `lebron.txt`. This is because peer 1 owns the file, and distributes it to other peers prematurely, anticipating a download request.



The next image displays the logs for a download request from peer 3. Peer 3 entered the command `download lebron.txt`, and received it in part from peers 1 and 2. Once peer 3 obtained all the files required, it wrote it to a file as downloaded. We also see in peer 1 and peer 2 logs that each peer sent these chunks to peer 3.



The last step in our test is to close down each peer. We start with peer 1, followed by peer 2, followed by peer 3. We can see in the respective peers that each peer recognizes this shutdown, and removes that peer from the pool of known peers.

Meeting the requirements:

- Peer discovery: this property can be viewed in the logs as peers broadcast themselves and also discover other broadcasting peers
- Indexing and searching: each peer can list the chunks of information for each file it has access to. This is visible in the logs
- Data Transfer Mechanism: We can see in the logs the transfer of data. While I did not perform a diff in the terminal at the time, I can attest to the order and validity of the data downloaded.
- Concurrency and threading: Each request to each peer uses a different thread, as does each operation of any peer. This maintains the ability to perform multiple jobs, and each job has very low overhead on a local machine, which provides strong guarantees with respect to resource allocation. We also have a mechanism to handle the state of the

received chunks, and end requests when a file's chunks are satisfied. This works by checking if the last file is an EOF chunk (b'') and if every chunk up to that numbered chunk has been received.

- Chunk file transfer: Chunk size is defined in our configuration file at 256 bytes, which is small on purpose to showcase the chunked file transfer. We can see in the logs that each chunk is sent separately, as well as stored in session memory separately.
- Verifying File integrity: while there is no true opposition in this project, we do implement a basic checksum function that verifies the downloaded chunks are valid. We do not print logs on success, but if there would be a corrupted packet, we would output an error in the logs for the user.
- User interface: we opted for a basic command line interface with the options to list your chunks or download a file from other peers. We also provide thorough logging to enable the user to know what is happening while the program is running.