Contributors:

Jason Trapp (jwt9552@g.rit.edu)
Jose Estevez (jae9307@rit.edu)

Introduction:

We built a simple peer-to-peer file sharing simulation with two files: **peer.py** and **server.py**. The project is built with python, using sockets for peer-to-peer communication and Flask for peer-to-server communication.

We built a simple peer-to-peer file sharing simulation using Python. The project consists of two files: **peer.py** and **server.py**. Sockets were used for direct peer-to-peer communication and Flask was used for peer-to-server communication.

Architecture Overview:

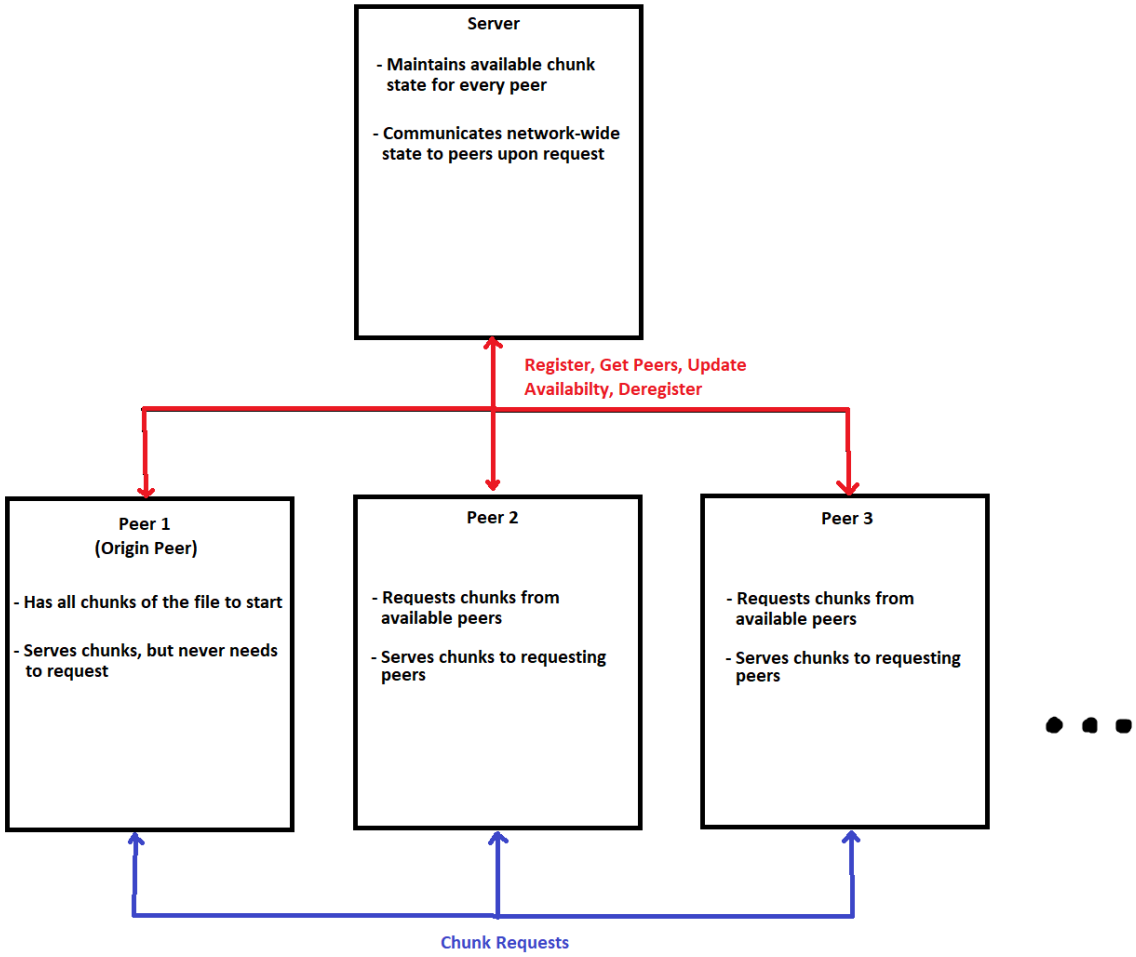
peer.py is responsible for initializing an origin peer (which starts with the complete file) and 9 other peers, which begin with no file chunks.

Each peer is running on a separate thread and is responsible for reporting its available chunks to the server, discovering compatible peers to match with, and serving chunks to other requesting peers.

server.py acts as the tracker. Its primary responsibility is to maintain the current state of each peer's chunk availability. It has 4 API endpoints:

1. **POST /register:** This endpoint receives requests from new peers to join the network and initializes their ip, port, and available chunks
2. **GET /get_peers:** This endpoint is called by peers that are looking for a new peer to match with. It returns the current state of all peers in the network (what chunks they have)
3. **POST /update:** After exchanging with a peer, this endpoint is reached to update which chunks it has
4. **POST /deregister:** When a peer no longer needs to be in the network, this endpoint is used to remove it from server managed state

Architecture Visualization:



Peer Matching Algorithm:

Each peer follows a simple strategy to complete its file:

1. Check which chunks it's still missing
2. Ask the server for a list of peers and what chunks they have
3. Randomly pick a peer
4. If that peer has any chunks it needs, grab up to 5 of them
5. After downloading, update tracker with new available chunk list
6. Repeat until peer has the full file

Screenshots:

Selecting a starting file:

```
D:\peertopeer>py peer.py
--------------------------------------------------------------------------------
Enter the filename to share (e.g., file1.txt): file2.txt
```

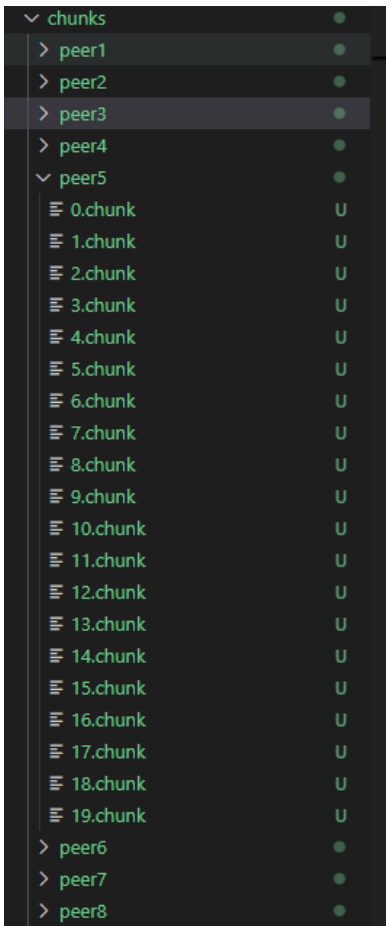
Server printing available chunks for each peer:

```
Available peers for 'file2.txt':
  peer1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
  peer2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer3: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer4: [0, 1, 2, 3, 4]
  peer5: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer6: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer7: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer8: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer9: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  peer10: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Server registering a peer:

```
Peer registered: peer2 at 192.168.1.136:6001 with files: {}
Current peers: {'peer1': {'ip': '192.168.1.136', 'port': 6000, 'files': {'file2.txt': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
3, 14, 15, 16, 17, 18, 19]}}, 'peer2': {'ip': '192.168.1.136', 'port': 6001, 'files': {}}}
```

Directory view of chunks after every peer is finished:



Peer requesting a chunk and request being fulfilled:

```
Peer peer7 requesting chunks [15, 16, 17, 18, 19] from peer4 (192.168.1.136:6003)
Peer peer7 requesting chunk 15 of file2.txt from 192.168.1.136:6003.
Peer peer4 accepted connection from ('192.168.1.136', 62947)
Received request: file2.txt 15 from ('192.168.1.136', 62947)
```

Confirmation of file share completion:

```
Peer peer8 has downloaded all chunks. Done!
Peer peer4 has downloaded all chunks. Done!
Peer peer10 has downloaded all chunks. Done!
Peer peer7 has downloaded all chunks. Done!
Peer peer9 has downloaded all chunks. Done!
```