Design Document

Alec Buchanan & Jason Lawrence

October 11, 2018

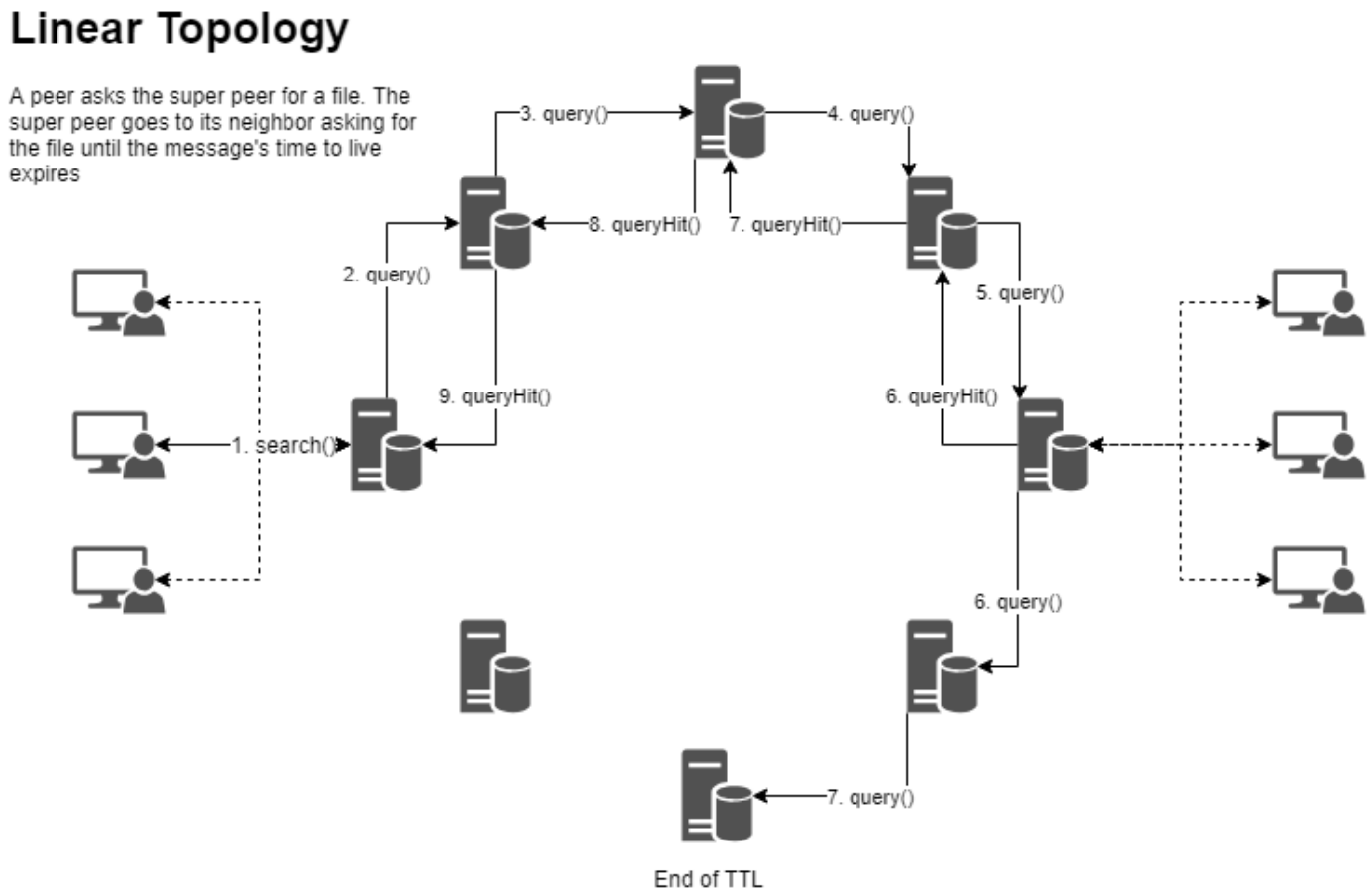
1. Program Design

This programming assignment uses programming assignment one as a base. If you would like to see the documentation for the super peer check the documentation for index server in the PA 1 design document.

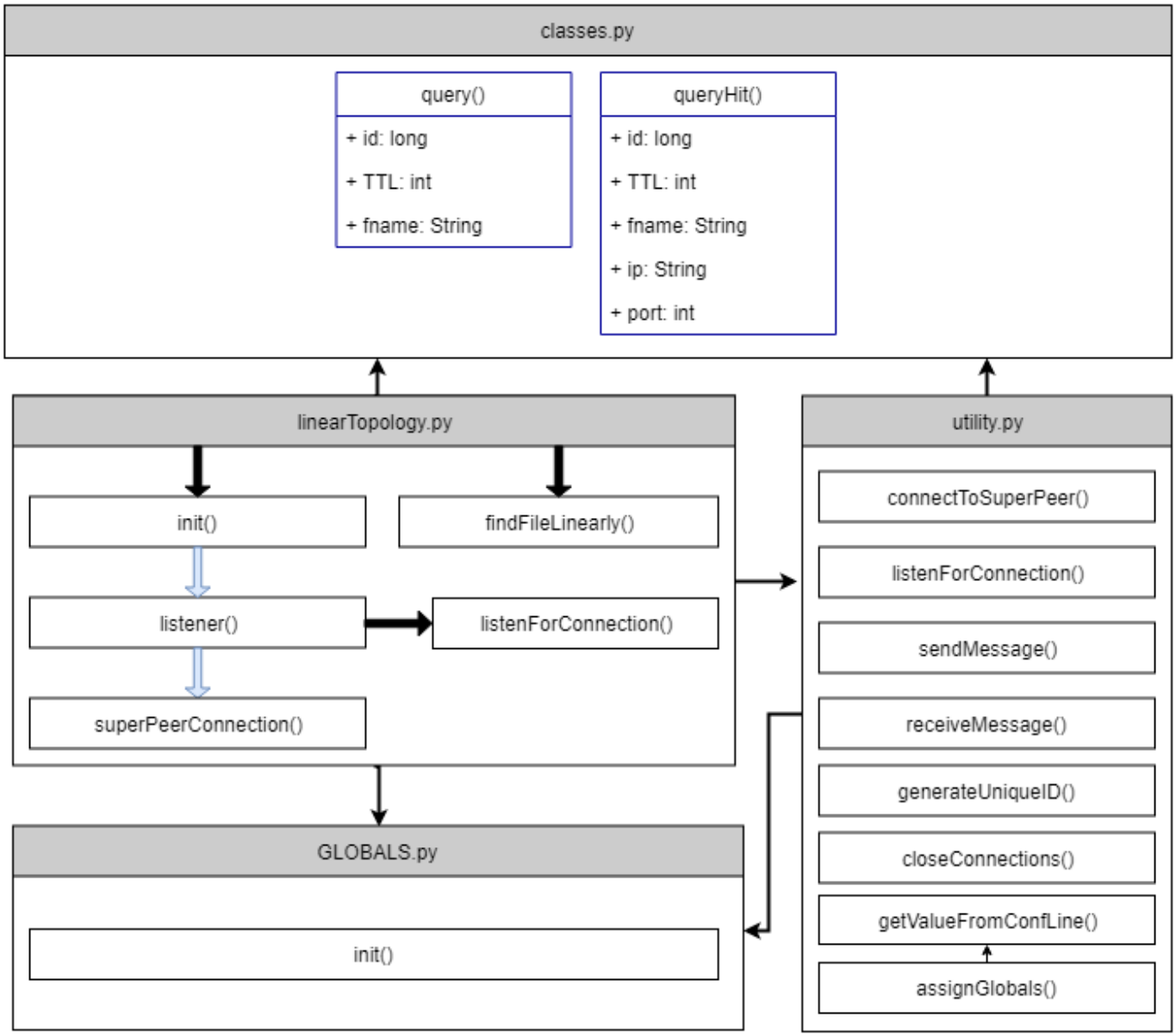
Two different topologies were implemented. The first topology implemented was the linear topology, and the second one implemented was the broadcast topology. The linear topology was implemented like a library that needs to be imported. On the other hand, the broadcast topology was coded in the superpeer.py file.

1.1 Linear Topology

The linear topology works like a ring. Each super peer has a left neighbor and a right neighbor. Messages are sent to the left neighbor and are forwarded around the ring until their time to live expires or they have made a full circle.



1.1.1 Program Structure



add table for each file and document the functions

linearTopology.py

Function	Description
int init ([(char[15],int)] * (char[:]))	The initialization function. It reads the config file, sets globals, and spawns a listener thread. The listener thread listens for incoming connections.
int listener ([(char[15],int)] * (char[:]))	Listens for incoming connections. Takes a function as an argument. The argument function should take a file name and search local leafs for files. It should return the hits with a tupple of (ip,port)
int findFileLinearly (char[:])	Initiates linear search for file. Takes file name as argument.

Function	Description
int superPeerConnection (socket.socket, [(char[15],int)] * (char[:]))	Handles super peer connection and forwards queries. Takes in a connection object and a function to search for files registered on index server.

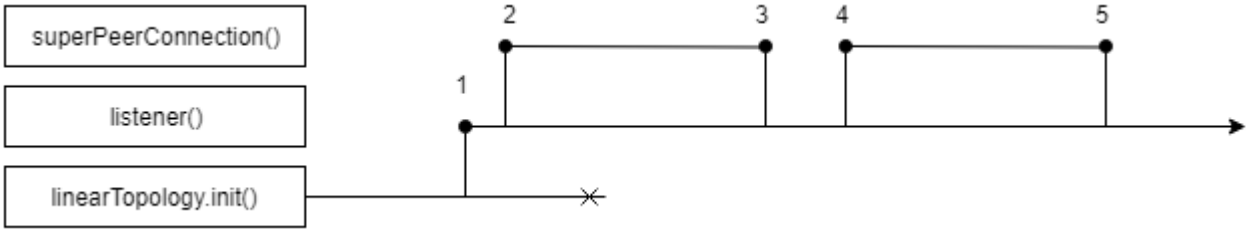
utility.py

Function	Description
int connectToSuperPeer (char[15],int)	Connects to server at given address
generator listenForConnection (int)	Wait for a connection on a given port and yield the socket object when a connection is made.
int sendMessage (socket.socket, void *)	sends the given message over the given socket.
int receiveMessage (socket.socket)	reads from the receive buffer when a message is received
char * getValueFromConfLine (char[:])	Takes a line from the config file and returns the value from that line
void closeConnections (socket.socket, socket.socket)	closes two connections at once
void assignGlobals (file)	Takes config file and assigns globals variables based off of file
long generateUniqueID ()	Creates a unique message ID

GLOBALS.py

Function	Description
void init (void)	Defines and initializes global variables

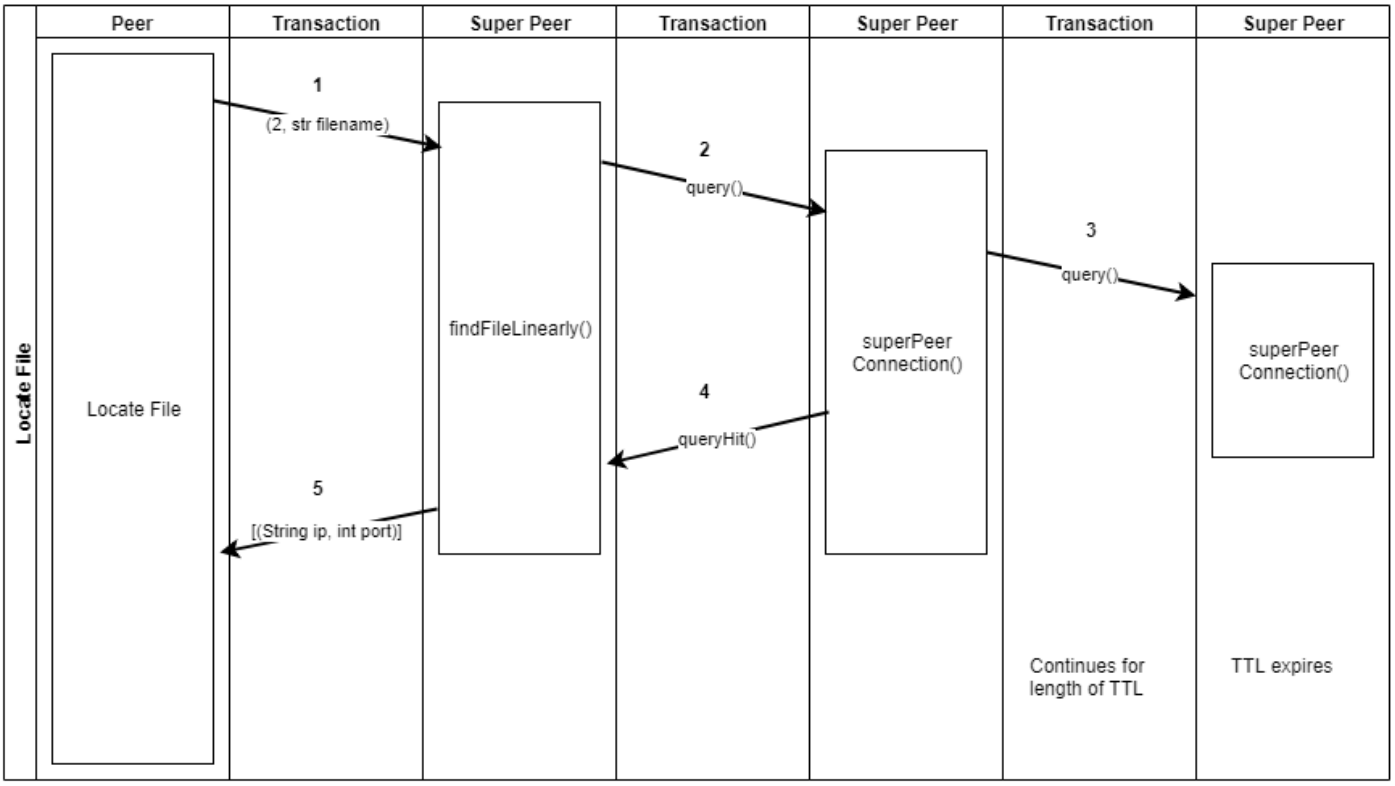
1.1.2 Thread Timeline



Events

1. init is done reading config and setting global variables, so it starts the listener thread.
2. listener receives a connection from another super peer
3. Connection is closed for various reason (error, expired TTL, loop detected)
4. listener receives a connection from another super peer
5. Connection is closed for various reason (error, expired TTL, loop detected)

1.1.3 Protocol

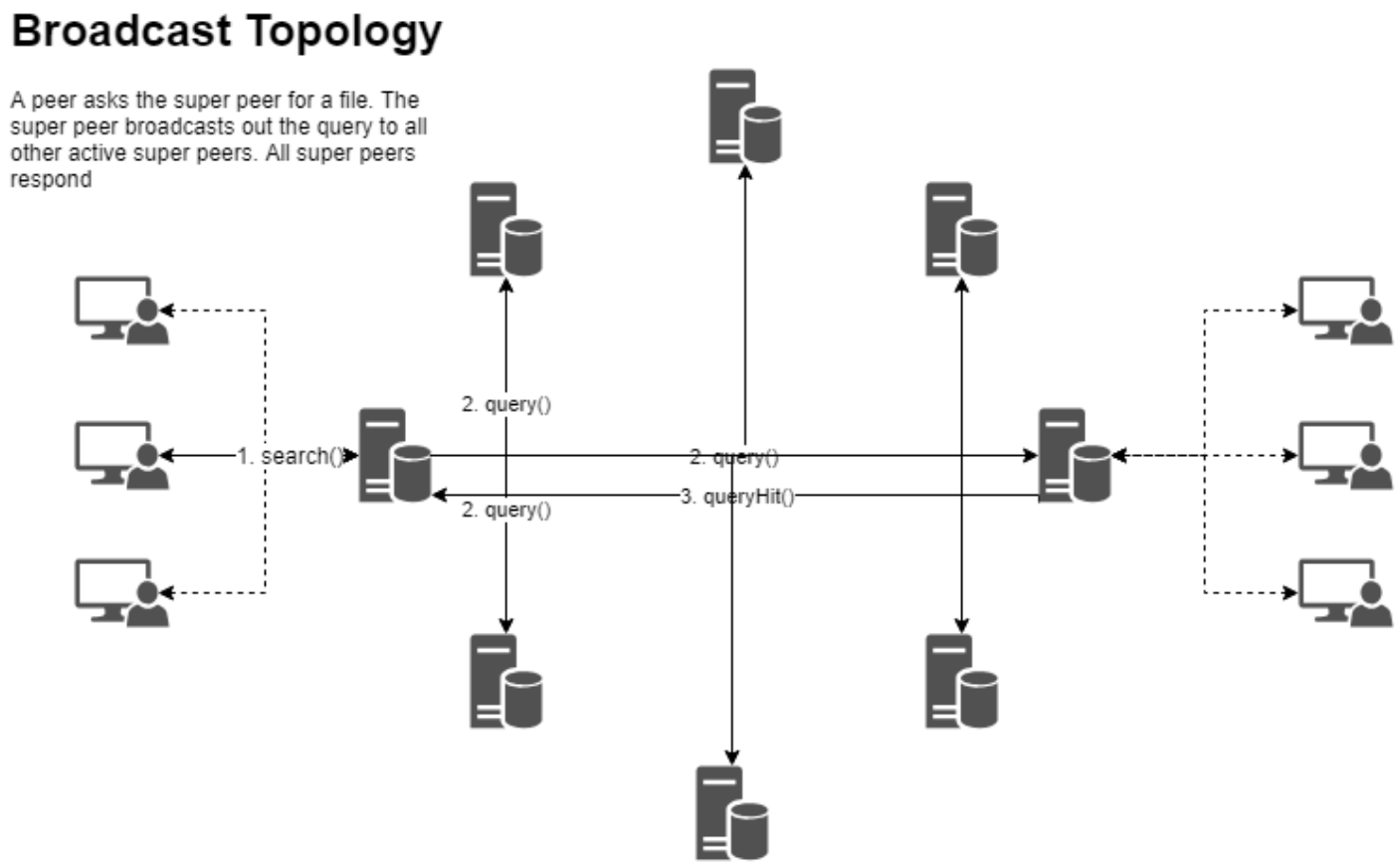


Transactions

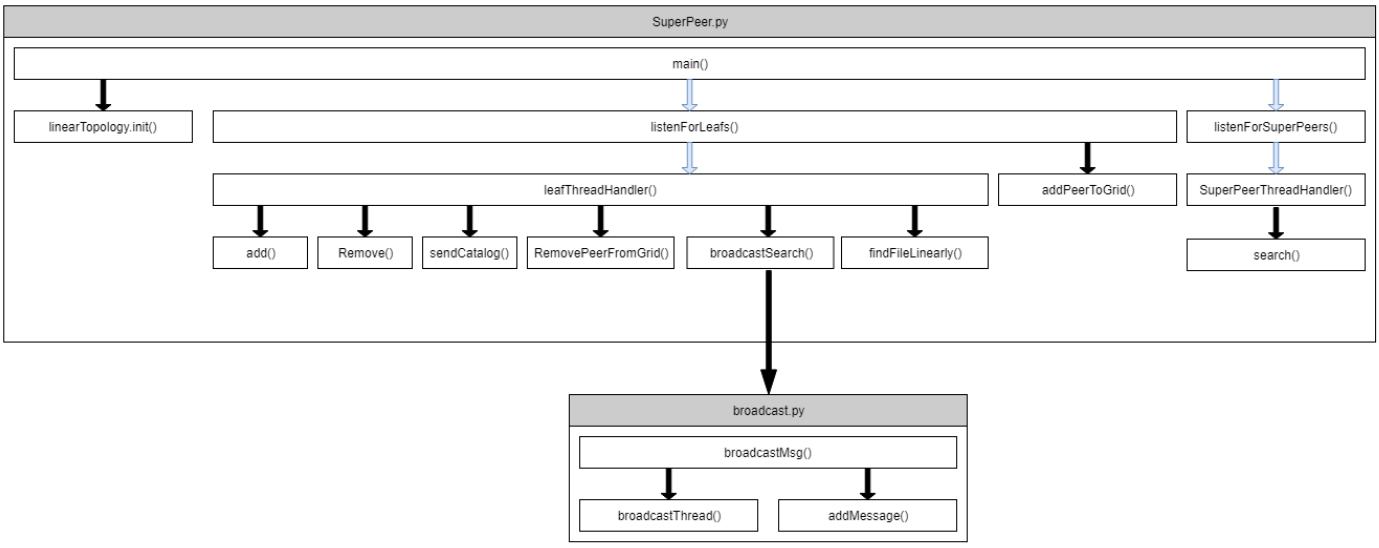
1. *peer* asks *super peer* to search for file
2. *super peer* sends out a query to its neighboring super peer
3. *super peer* forwards query until TTL expires. The last peer is in charge of closing the connection. The closing connection makes a ripple affect going all the way back to the originating function.
4. *super peer* has a peer with the file, so a queryhit is sent
5. *super peer* sends the peer a list of other peers that have the file

1.2 Broadcast Topology

Broadcast topology is pretty self explanatory. When a super peer queries a file, it sends the query to all other super peers at once. The super peers with the file respond with a query hit.



1.2.1 Program Structure

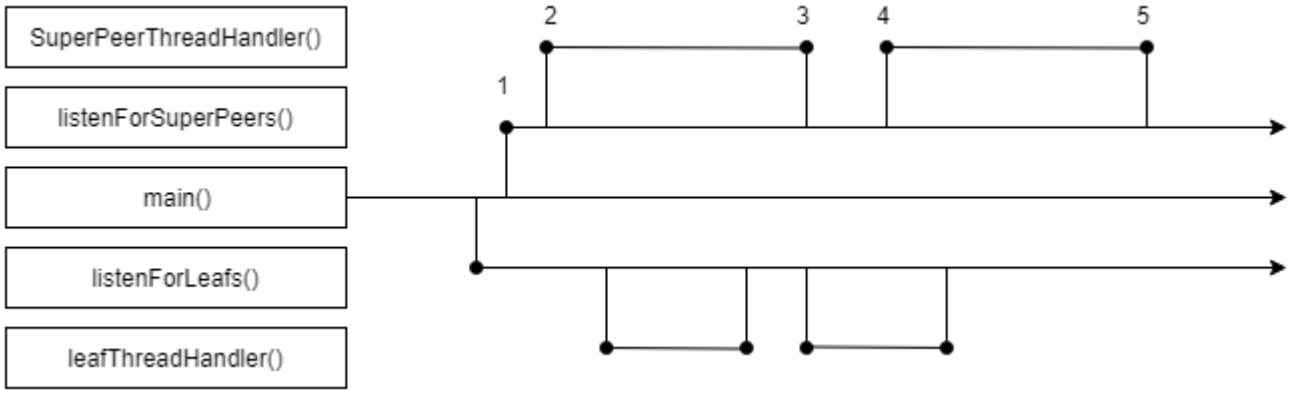


SuperPeer.py

Function	Description
todo BroadCastMsg (todo)	Takes query and broadcasts it to all other super peers
int finFileLinearly (char[:])	Sends out a query to the super peer's neighbor
void listenForSuperPeer (void)	Listens for new super peer connections and dispatches a new thread on connection

Function	Description
void SuperPeerThreadHandler (socket.socket)	Handles incoming connection with another super peer
void addMessage (void)	Initializes the Reponse list to make sure no old responses were left in it.
void broadcastThread (char[:], [(char[:], int)], int)	Sends a message to all other Super Peers.
[(char[:], int)] broadcastMsg (char[:])	Handles incoming connection with another super peer

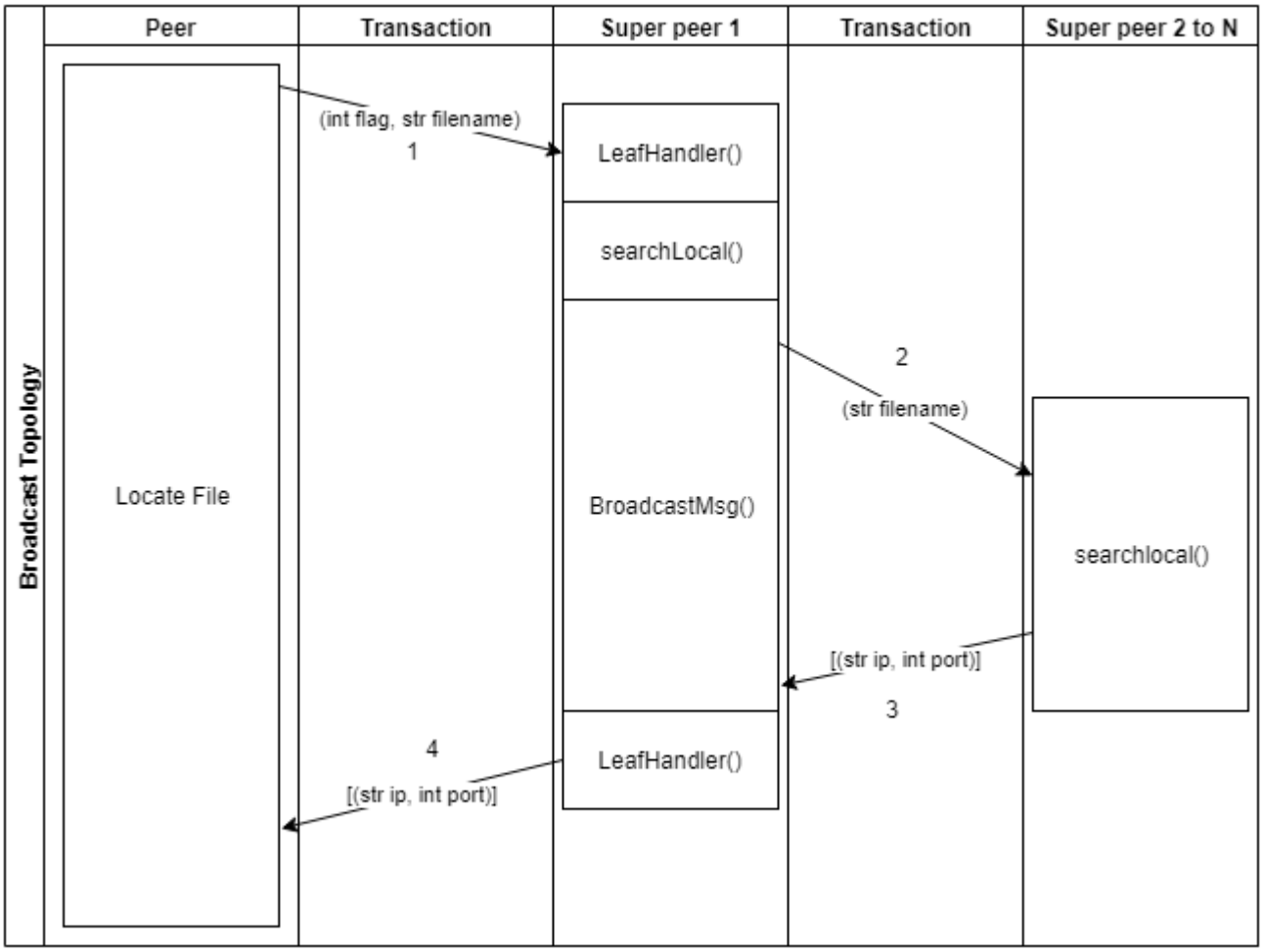
1.2.2 Thread Timeline



Events

1. *main()* spawns thread to listen for super peer connections under the broadcast port
2. *listenForSuperPeers()* gets a connection from another super peer
3. *superPeerThreadHandler()* sees that the connection has been closed
4. *listenForSuperPeers()* gets a connection from another super peer
5. *superPeerThreadHandler()* sees that the connection has been closed

1.2.3 Protocol



Transactions

1. *peer* asks *super peer* to search for file
2. *super peer 1* broad casts out queries to all super peers
3. *super peer i* sends back a queryhit if it has the file registered
4. *super peer 1* responds to the peer with a list of peer servers

2. Design Tradeoffs

2.1 Considerations

- (Linear topology) Should neighbor connections be kept open or should a new connection be made for every query?
- How does the system determine which topology to use?
- Should the super peers listen for other super peers on the same port, or should it be based on topology?
- Which data structure should the linear topology use to store the data structures in?

2.2 Implimented

(Linear topology) Should neighbor connections be kept open or should a new connection be made for every query?
We decided that a new connection should be made for every query. Creating and closing sockets is not very

resource or time intensive, so there is not much to lose. Also, this method allows for easier multitasking. It is difficult to have multiple threads operate on the same socket. However, it is a lot easier to have multiple threads work on multiple sockets. Whenever a new query or connection comes in, a new thread is spawned.

How does the system determine which topology to use?
The topology used is determined by the client. The alternative would be to have the super peer determine it, but that would make things more complex than it has to be. In the current configuration, the peer remembers what topology the user wants. When a file is requested from the super peer, a flag is sent with the filename to tell the super peer which topology to use.

Should the super peers listen for other super peers on the same port, or should it be based on topology?
The ports are based off of the topology. Since we have been testing on the same machine, we can not have a “standard” port for each topology. Because of this, we designated the 8000 port range for the broadcast topology and the 7000 port range for the linear topology. Separating the ports made it easier to work on the project since there are two of us. There was less conflict. It also made it easy to distinguish which queries were meant to be broadcasted and which were meant to be forwarded (linear).

Which data structure should the linear topology use to store the data structures in?
Currently the linear topology records the message IDs in a list. It is not the ideal solution but given the time we had it was our best option. The preferable data structure would be something that has a quick insert and read time complexity.

3. Improvements (TODO)

1. The data structure for recording message IDs should be changed. The current data structure used is a list which has an insertion time of O(n) and a lookup time of O(n). Using a tree based data structure would allow for much faster lookup and insert times, O(logn).
2. The code in the linear topology that forwards queries could have a bug in it. When it reads a query hit it assumes that there is only one queryhit object in the receive buffer. If there are multiple queryhits in the receive buffer, then there might be an error or some queryhits might be ignored. Because of the nature of the linear topology, the messages are pretty spaced out and this has never been an issue. To ensure it does not become and issue, the socket.recv(size) size could be set to sizeof(queryhit)