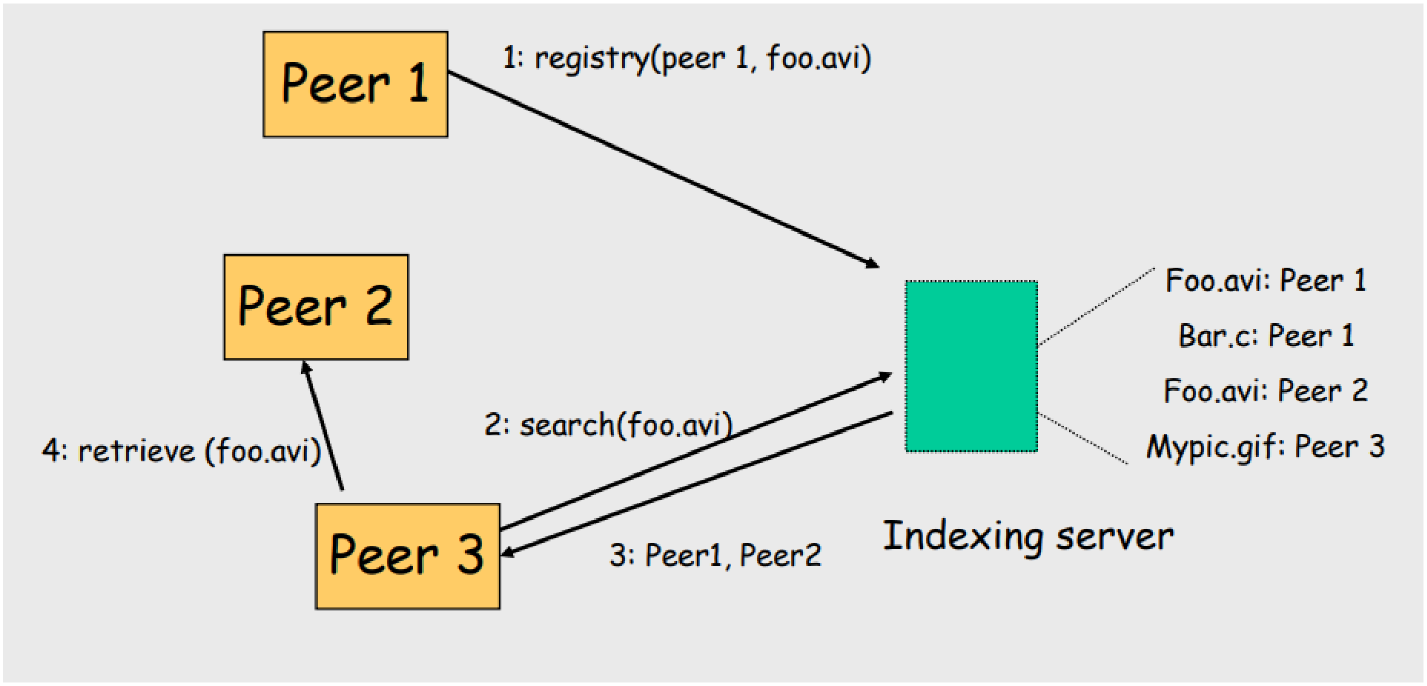
Design Document

Alec Buchanan & Jason Lawrence

September 20, 2018

1. Program Design

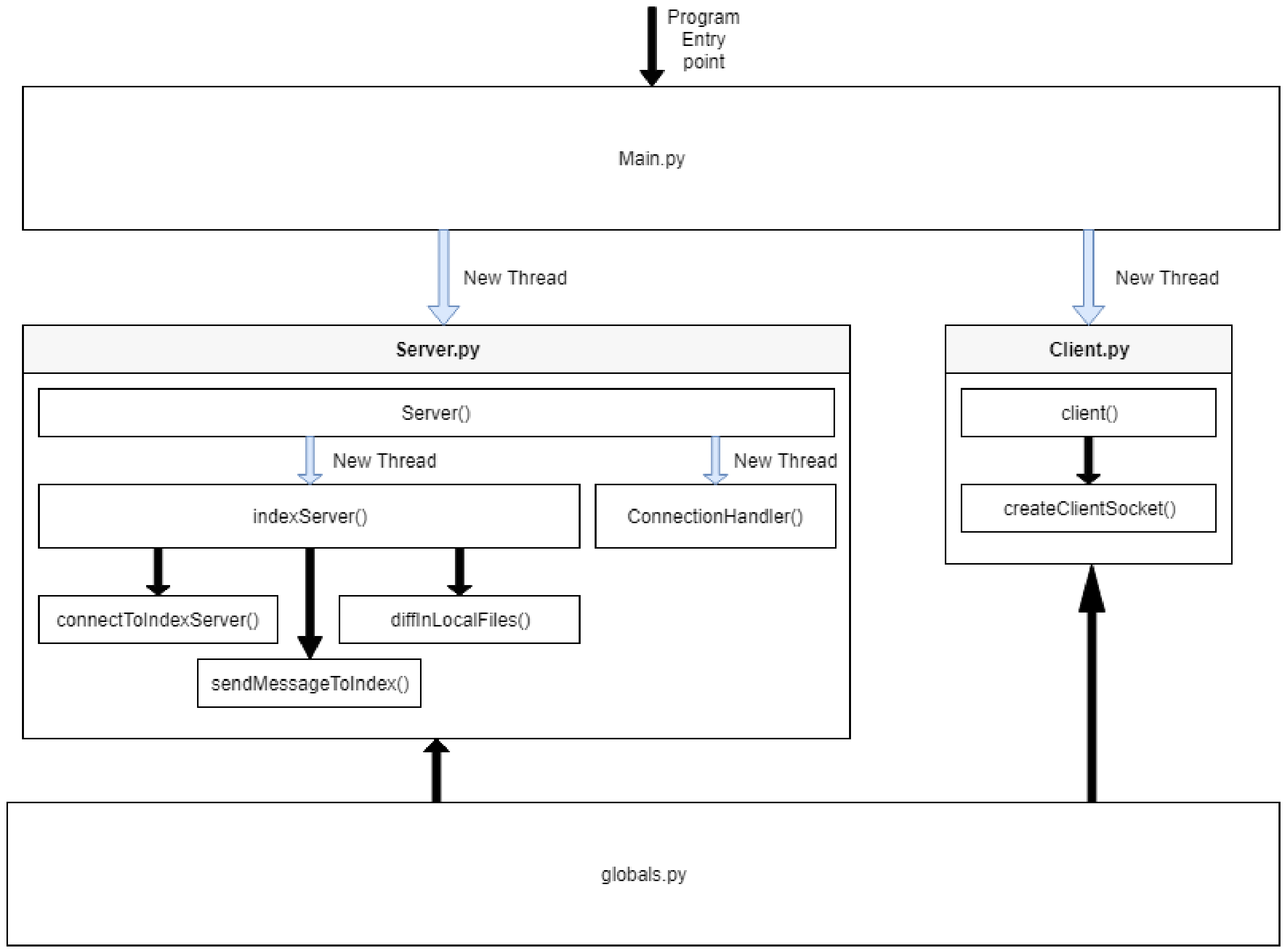


The Whole P2P network consists of two main programs. The first program is the Index server. The Index server keeps a live list of all of the peers connected and which files they are hosting. The second program is the Peer. The peer has two functions: the peer server, and peer client. The Peer server keeps the Index server updated on ther peer's local files, and the peer sends files requested by other peers. The peer client talks to the index server to find which peer has the requested file. Then the peer client will download the requested file from the other peer.

1.1 Peer Design

The peer design is broken down into two categories: program structure, and thread timeline. The program structure is a design of how the different functions in the different files interact. The thread timeline is needed because of the use of threads. Threads are used because of the requirement for the peer to multitask and handle multiple, asynchronus connections. The thread timeline shows a timeline, with events, of how the different threads interact with each other.

1.1.1 Program Structure

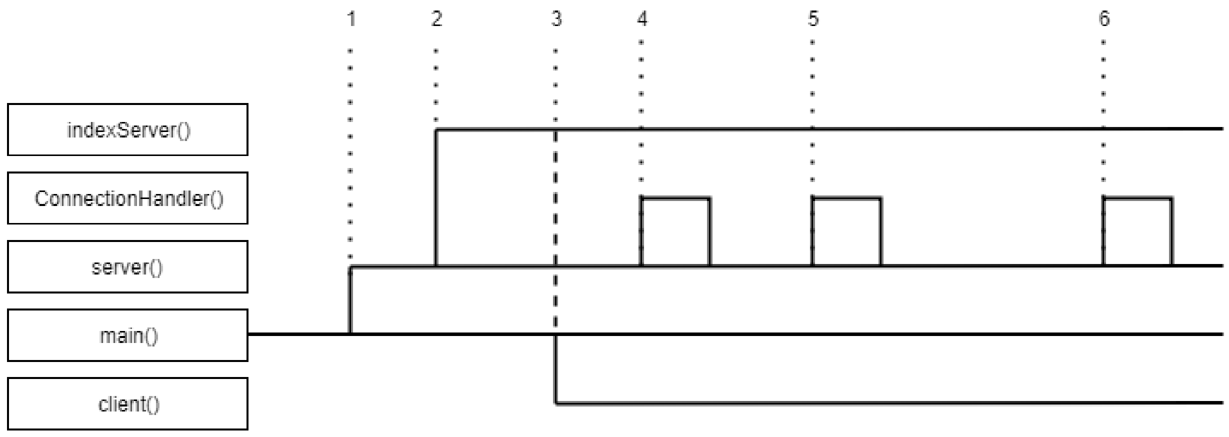


Main.py
void main(args[], argc): spawns peer server and peer client thread

Server.py
int server(void): spawns index server thread and handles incomming connections from clients
int connectionHandler(socket.socket, str): spawned by server() and handles the protocol and file transfer with clients
int indexServer(void): Keeps the index server up to date on local files
socket.socket connectToIndexServer(void): returns socket with connection to index server
[(int, str)] diffInLocalFiles(void): Determines which files need to be registered or removed from index server
int sendMessagesToIndex(socket.socket, [(int,str)]): Updates index server on local files

client.py
int client(void): Asks user for input and downloads files
socket.socket createClientSocket(void): creates socket for client to connect to server with

1.1.2 Thread Timeline



Events

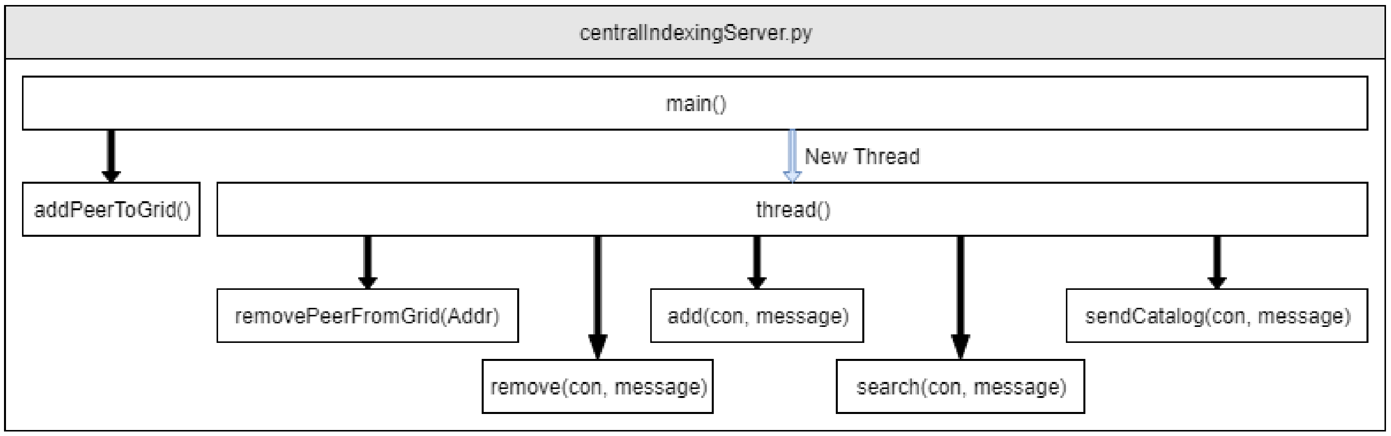
1. *main()* starts peer server
2. *server()* creates connection with index server and registers files
3. *server()* is fully started, so *client()* now starts
4. *client()* makes connection with *server()* and asks for file
5. *client()* makes connection with *server()* and asks for file
6. *client()* makes connection with *server()* and asks for file

A combination of static and dynamic threads are used. static threads are used for *server()*, *client()*, and *indexServer()*. It is hardcoded that there will be three threads for the listed functions, whereas, *connectionHandler()* threads are created dynamically. Everytime a client connects to the peer server a new thread is spawned.

1.2 Index Server Design

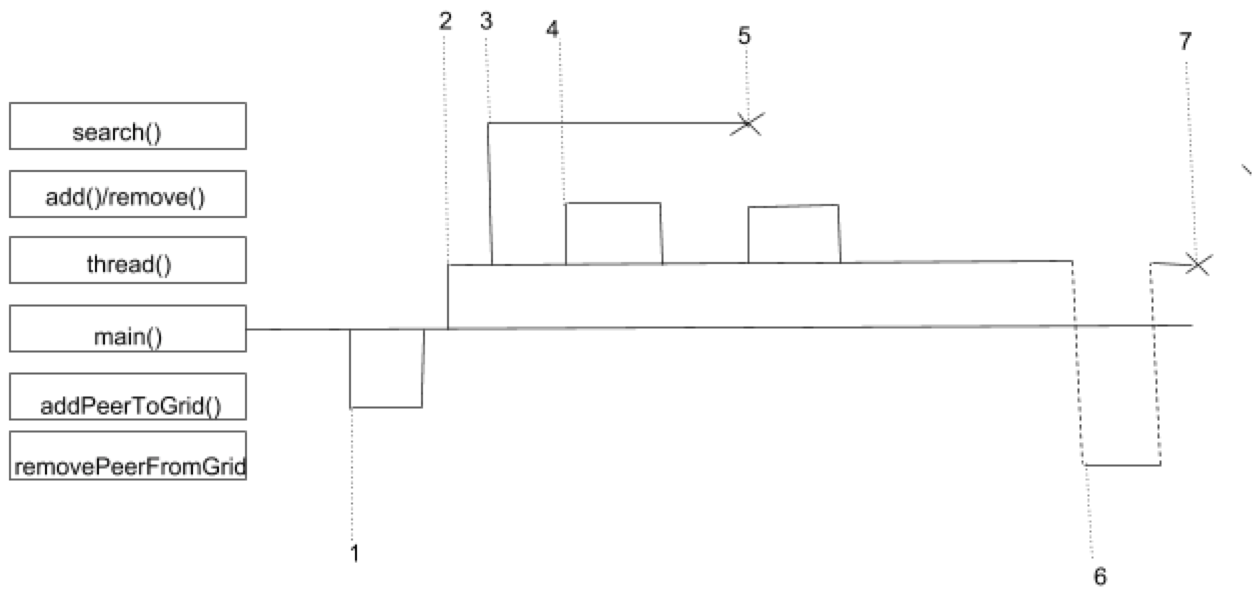
The Index Server's design will follow the same format as the Peer design, where we will first discuss the program structure and then explain the thread timeline. The program structure details how each function interacts with the other functions if they do so. The thread timeline is needed to help understand how the Index server handles the different connections and operations. Using threads in the Index Server allows it to accept multiple connections at the same time as well as concurrently handle requests from the different clients/servers.

1.2.1 Program Structure



Void main(args[], argc): Spawns threads for both server and clients and sets up the grid data structure to store all of the files and IP addresses.
thread(connection): handles receiving messages from both clients and servers and then does work on the data. If the connection came from the server side of the peer it keeps the connection alive so it is quicker to update the file list. Client connections are closed after the request they made are handled.
add(con, filename): adds a filename to the list of files for that peer.
remove(con, filename): removes the file from the peer's file list.
search(con, filename): looks through all of the files on all of the peers and returns the IP addresses of the peers that have that file.

1.2.2 Thread Timeline



Events:

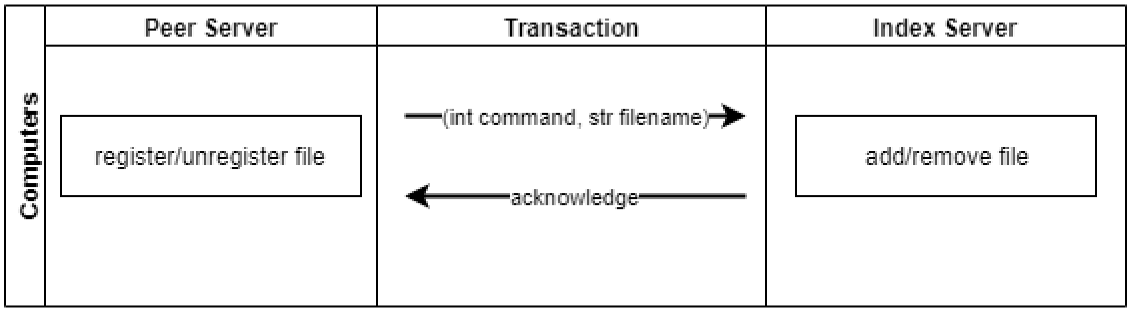
1. *main()* calls *addPeerToGrid()* if the peer is not registered
2. *main()* spawns a thread to handle the connection
3. If the connection is from the client handle the search request by calling *search()*
4. If the connection is from the server then add and remove files from the file list as needed
5. After the search request is handled send results back to the client and then close the connection and kill the thread
6. When the server connection is no longer active unregister the peer.
7. Close the connection and kill the thread.

The index server utilizes both static and dynamic threads. Static threads are used for connecting to peer server and dynamic threads are used for peer client connections

1.3 Protocols

There are a couple of different protocols that we had to define and use. The first was a protocol for file registration. The second was the protocol for file transfer.

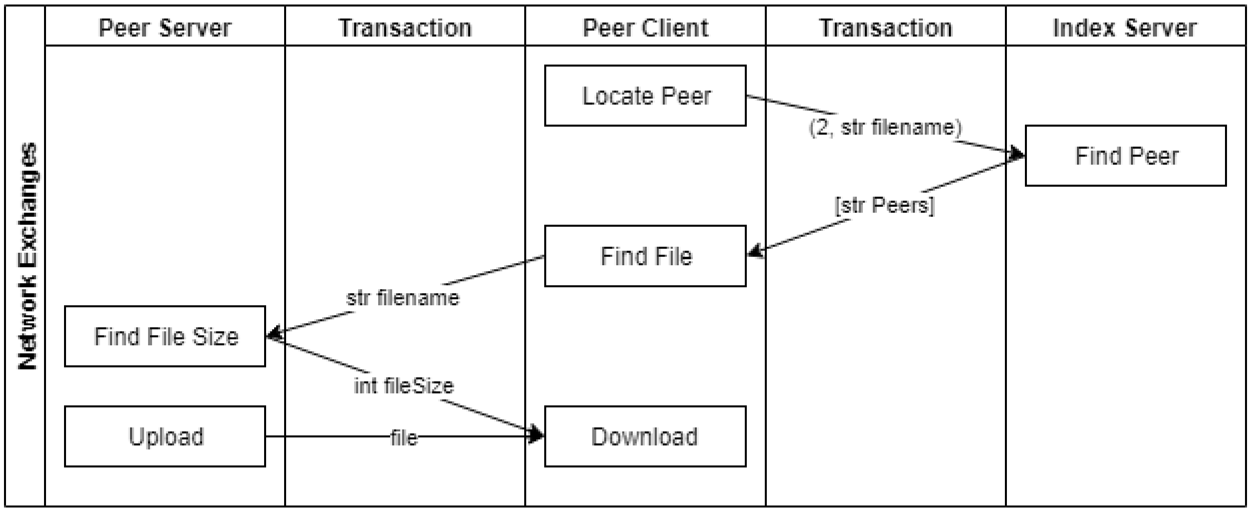
1.3.1 File Registration



Transactions

1. *(int command, str fileName)*: A tuple with a command and file name is sent. The command can be 0 for unregister/remove file or 1 for register/add file. The filename is the name of the file that needs to be added
2. *acknowledge*: The index server sends an acknowledge back to the peer confirming it got the command and that the peer is ok to send another command.

1.3.2 File Transfer



Transactions

1. *(2, str fileName)*: A tuple containing the command 2 (search), and a string of the filename.
2. *[str Peers]*: A list of peer servers that host the requested file. If the list is empty, then the file can not be found.
3. *str fileName*: The file name requested from the peer client.
4. *int fileSize*: The file size of the requested download. A file size of 0 means the file can not be found.
5. *file*: The file is then sent over the network

2. Design Tradeoffs

2.1 Considerations

- What programming language to use?
- To use blocking I/O or non blocking I/O?
- Whether to use FTP or use sockets to send files?
- What type of data structure to use to store list and files of peers on index server?

- How should the protocol between the peer server and index server work? Should the peer server register each file seperatly (send an individual message for each file) or send one message with all of the files that need to be registered?
- Should the peer decide what their peer number is or should the index server assign a peer number?
- Should the search command from the peer client connect to the same port on the index server that the peer server does? Or, should the peer client connect to a seperate port that is just for clients to search for files?

2.2 Implimented

What programming language to use?
We settled on python for a couple reasons: Python allows for quick development; it is very widely used with a lot of support; it is an easy language to conceptualize and understand.

To use blocking I/O or non blocking I/O?
We chose to use blocking I/O communication with our sockets. All of the protocols we designed and implimented in our script revolve around synchronus communication and execution. Every place we used *socket.recv()* there was either nothing else to do without the received information or the received information was necessary to continue.

Whether to use FTP and RPC or use sockets to send files?
Sockets were used over RPC or FTP because of the simplicity and familiarity.

What type of data structure to use to store list of peers and files on index server?
We used two different data structures to implement the index server. The first data structure that was used is a python list. We stored the peer's Ip address in it so we could check to see if the peer was registered with the index server. The other data structure was a list of lists. the first index of each list is the IP address for the peer and all subsequent indexes are the file names stored on the peer. this made it really simple to append and remove filenames from the peer's file list. searching was O(N) because we have to iterate through each file list. We did not duplicate filenames or IP addresses in the data structure.

How should the protocol between the peer server and index server work?
After debating, we decided to update the index server with individual register or unregister commands as opposed to one giant register/unregister command with multiple filenames. This will create more network traffic, but is easier to package (from the sender side) and easier to parse (from the receiver side).

Should the peer decide what their peer number is or should the index server assign a peer number?
The index server assigns the peer a peer number. This way peer numbers can be assigned dynamically and based on availability. The alternative would either require hard coding or require the peers to communicate and negotiate peer numbers.

Should the search command from the peer client connect to the same port on the index server that the peer server does?
The port for the connection to search the index server is the same as the port to register/unregister files. This makes the protocol more complicated, but it makes the index server a lot simpler. So, the commands to send to the index server are 0 (remove), 1 (add), 2 (search).

3. Improvements (TODO)

- Set up better load balancing for downloading files from peers. One way to do it would be to let the client randomly pick a peer from the list sent by the index server. A second way of doing it would be to let the index server order the list by the least popular peers in the list.

- Have client try another peer server if the first download attempt fails. This will automate failure recovery and prevent the client from trying to connect to the same peer server a second time.
- Allow the client to ask the index server for a list of all files availabe for download. This could be done by adding another command to the command list. By connecting to the index server on the regular port, one could get a list of available files by sending something like *(3, “”)*. This would be an easy way to use current infrastructure to list available files.
- Let the user create a config file with useful variables. Currently, config settings (like index server ip and local shared directory are hard coded). Ideally we would have a file *settings.config* that would store all useful, computer specific information.