Trung Truong
Comp 112 - Networks
Prof. Dogar

Napster-like p2p network

1. Introduction

   In class, we have covered how a client-server file transfer protocol operates and analyzed the limitations of such a model. For instance, the traditional client-server, while being easy to implement, poses several scaling challenges; it has a single point of failure and linearly decreases the system bandwidth as traffic increases. To compensate for these problems, new peer-to-peer models, where client nodes can coordinate to query files from others, were introduced.

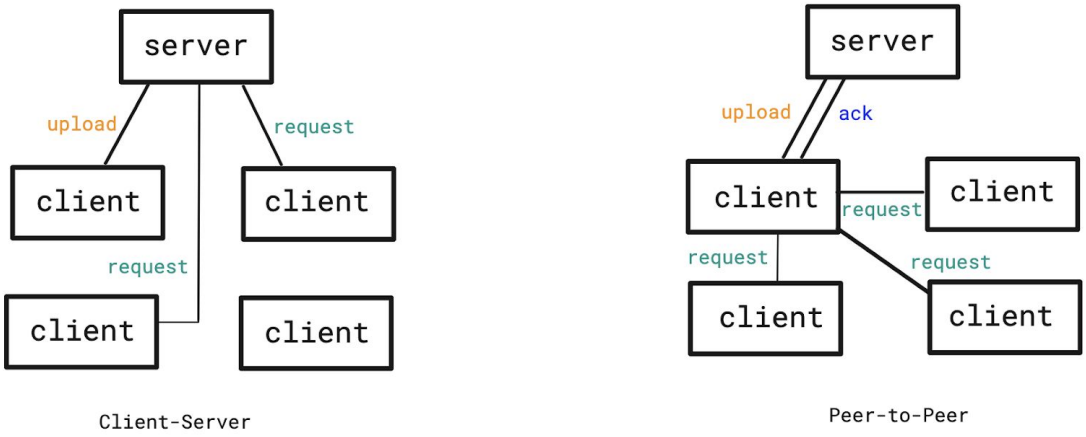

Fig 1. Client-server v.s p2p

Among the popular peer-to-peer file transferring protocol, I implemented a "Napster-like" protocol, named after the popular music sharing platform Napster, for the final project. Unlike the traditional client-server model that uses a central server to handle all clients' file upload and download requests, the Napster architecture uses a central index server only to keep track of all clients in the network. On the other hand, the file transferring process is delegated to nodes in the network. This architecture will also ensure a fair system that equally distributes the heavy-lifting file transfer task among client nodes in the network.

2. Deliverables

   In this project, the deliverables include the following:

   - A working implementation of the index server, which is in charge of monitoring every node in the network and registered file segments.
   - A working implementation of the peer server, that can register files to index server, and perform file transfer to other peer nodes using TCP/IP.

   All invariances from the proposal are achieved, including the following:

   - The index server must know every registered file segment and machine addresses for such segments.
   - The index server must be able to handle if a peer is down.
   - The peer nodes must be in charge of file transferring.

3. Architecture

   The implementation of this project includes 4 main modules, including the TCP module, message protocol, peer server, and index server.

   TCP module

   - This module was implemented to create a layer of abstraction between the index and peer server and the socket API.
   - This module provides wrapper functions that create, close, read from, and write to sockets. This module provides a baseline for the TCP-based peer and index server that require concurrent connection with multiple clients and uncorrupted data transferring.

   Message protocol

   - This module was implemented to create a common communication protocol between the peer servers and the index server. The specifics of each message is listed as below.

Message	Definition	Description
`REGISTER`	2 bytes: type 4 bytes: file_size 20 bytes: file_name 16 bytes: seeder_ip 2 bytes: seeder_portno 64 bytes: file_hash	This message is sent from peer node to index server to register a file name. The SHA-256 file hash is also calculated in order to confirm the integrity of the file later.
`REGISTER_CONFIRM`	2 bytes: type	After a `REGISTER` message is received by the index

	4 bytes: file_size 20 bytes: file_name 16 bytes: index_ip 2 bytes: index_portno	server, the index server sends a `REGISTER_CONFIRM` message to the seeder address embedded in the `REGISTER` message to confirm if the seeder actually contains the file. If a `REGISTER_ACK` message is received, it means the file is valid and the seeder information is registered in the index server. Else, the file information is discarded.
`REGISTER_ACK`	2 bytes: type 4 bytes: file_size 20 bytes: file_name 16 bytes: seeder_ip 2 bytes: seeder_portno 64 bytes: file_hash	When a `REGISTER_CONFIRM` message is received, the peer server responds back with a `REGISTER_ACK` message if the listed file exists. Included in the `REGISTER_ACK` message will be the computed SHA-256 hash of the listed file on the peer server, and this hash will be compared with the original hash from the `REGISTER` message to validate the file integrity.
`FILE_FOUND`	2 bytes: type 20 bytes: file_name 16 bytes: seeder_ip 2 bytes: seeder_portno 64 bytes: file_hash	When a `REQ_FILE` message is sent to the index server, the index server responds back with a `FILE_FOUND` message if the requested file has been registered.
`ERR_FILE_NOT_FOUND`	2 bytes: type	When a `REQ_FILE` message is sent to the index server, the index server responds back with a `ERR_FILE_NOT_FOUND` message if the requested file has not been registered.
`REQ_FILE`	2 bytes: type 20 bytes: file_name 16 bytes: leecher_ip 2 bytes: leecher_portno	A peer server can send a `REQ_FILE` message to the known index server to get seeders information. If the index server responds with a `FILE_FOUND` message, the same peer can send another `REQ_FILE` message to the embedded seeder to begin downloading the file.
`DATA`	2 bytes: type 4 bytes: file_size 20 bytes: file_name 4 bytes: segment_size 4 bytes: segno 16 bytes: seeder_ip 2 bytes: seeder_portno 64 bytes: file_hash 8 kb: data	After a peer server sends a `REQ_FILE` message to the seeder peer server, the seeder peer will respond with a series of `DATA` messages if the listed file is found. The binary file is divided into 8kb segments for each `DATA` message. On the other end, if a peer server receives `DATA` messages, it will start collecting them and arrange the data in a correct order (as listed by embedded segment number). Once the file is fully downloaded, the original seeder sends a `REGISTER` message with the leecher peer IP address to add such peer to the list of seeders.

Peer server and Index server

- These modules both have a server loop that can concurrently connect with multiple clients, and their behaviors are defined above based on the message received.

4. Demo

   A live demo can be viewed here. Link to the source code.

5. Future improvements and discussion

   While the a functional implementation of the project has been achieved, this project can be improved in the following ways

   - Better seeder-picking algorithm. At the moment, whenever a leecher finishes downloading a file, it becomes a seeder for future downloads, and the seeder for future downloads is chosen randomly. This implementation, while ensuring that the heavy-lifting file transfer is equally distributed among seeders, offers no reward for the original seeder. A rate limiting implementation where the seeders get rewarded with higher bandwidth was being implemented, but unfortunately I didn't have a chance to finish it.
   - Like the original Napster site, this architecture still poses a single point of failure (the index server). An algorithm to go around this issue is using query-flood along with the index server, which was also being implemented but not finished.

6. Conclusion and reflection

   While this project is not fully scaled, it provides a functional, and light-weight tool for efficient file distribution in a computer network (this project was tested with multiple machines in Halligan servers). Possible applications include group programming during labs, and rapid file distribution in small groups. Throughout this project, I investigated multiple peer-to-peer networking algorithms, and implemented multiple ways to ensure the file integrity is not corrupted. Overall, while it was a large project for an individual in a short amount of time, its base-line goal was achieved and it was a good learning experience all throughout.