

Names: George Ulloa
Gourav Jyot Singh Khurana

Date: 12/21/13

Assignment 9: Final Project

- _____ Basic functionality [Max 10 points]
- _____ Search for and reliably retrieve files [Max 30 points]
- _____ Basic congestion control [Max 15 points]
- _____ Support and Utilize Concurrent Transfers [Max 15 points]
- _____ Congestion control corner cases [Max 15 points]
- _____ Robustness: [Max 10 points]:
- _____ Style [Max 5 points]

- _____ Total [Max 100 points]

Total in points _____

*Note: Due to the complexity of the project and the scale we have working in a group. We are submitting one project (source code/documentation/additional files) for both me (George Ulloa and Gourav Jyot Singh Khurana).

Professor's Comments:

Affirmation of my Independent Effort:

Objectives:

- 1. Implement file transfer application that runs on top of UDP and provides a BitTorrent-like protocol to search for peers and download/upload file parts.**
- 2. The application should be able to simultaneously download different file parts, called “chunks”, from different servers.**
- 3. The application should implement a reliability, flow control, and congestion control protocol (similar to TCP) that ensure fair and efficient network utilization.**

For this project we had to implement a simple p2p network. Using the very popular but legally questionable at times Bit torrent as a reference model, have to create a system in which we have a data file, which can be downloaded in small parts or “chunks.” For obvious reasons such, as bandwidth, congestion, and speed, we can not allow a file to be downloaded straight in its entirety. Rather the file will be split into chunks that that can be downloaded via different peers. These chunks are generally identified via hash values, which is the result of computing a well known has function over the data in the chunk. A client will want to download a file; it will grab a file that will contain the values for that chunk. That file will let the client/peer know what chunks to request from other peers in the network.

Taking this in mind we have to first understand how we do we actually get these chunk files? The first thing to have in place is the implementation of our program is having a data file. We will have a “Master Data File.” The master data file, will just be a simple text document that is basically the entire Ebook of the Trooper. To make chunking easier the file will be a multiple of 512 (we could pad it but we could not implement it). At first we tried to use the code that was provided in the sample file and recreate it in C. Due to our inexperience with C as a language we could not understand how using the chunking program. We ended up going with a python program that read the text in our text file, and converted it into the SHA-1 Hash codes. We output this to a file called “master-chunk-file” which list has values for the 512 byte chunks. Since we need our program to be able to actually understand which chunk is which we give it an identifier listing which numbered chunk it is.

id chunk-hash

The first line will specify the file that needs to shared among the peers. The peer will only read the chunks it is provided with in the peer’s has-chunks-file parameter.

Files we need to have:

GetChunks_p1/p2: Chunks peer needs to get

HasChunks_p1/p2: Chunks peer has

MasterChunkFile: List of chunks with ids

MasterDataFile: The Ebook

We are going to first look at the most important java class in our program. Initial state is what the main function of our program is.

Initial state which is located in our “Main” package, is the main java program in our program. It creates Instances of the Host and Receiver Classes. It will be reading A set of

commands that the user inputs:

peer -p <peer-list-file> Hash Code: 1507

-c <has-chunk-files> Hash Code: 1494

-i <peer-identity> Hash Code: 1500

-f <master-chunk-files> Hash Code: 1497

-d <debug-level> Hash Code: 1495

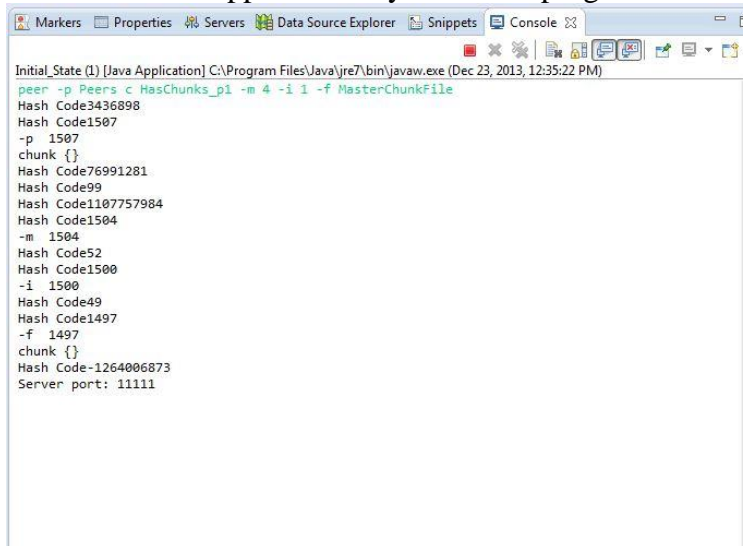
as well as

GET <get-chunk-file><output filename>

GOT <get-chunk-file>

These instructions from the user cause the program to open the specified chunks file and attempt to download all of the chunks listed in it. When it finishes downloading the program it will print GOT <get-chunk-file>. Our code will not handle multiple content requests. In our code reading the command corresponds to a specific hash number which reads the it's designated file, and outputs in the console.

Here it is what happens when you run the program.

A screenshot of a Java IDE's console window. The title bar shows 'Initial_State (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Dec 23, 2013, 12:35:22 PM)'. The console output shows the execution of a command: 'peer -p Peers c HasChunks_p1 -m 4 -i 1 -f MasterChunkFile'. The output then lists several 'Hash Code' entries: 'Hash Code3436898', 'Hash Code1507', '-p 1507', 'chunk {}', 'Hash Code76991281', 'Hash Code99', 'Hash Code1107757984', 'Hash Code1504', '-m 1504', 'Hash Code52', 'Hash Code1500', '-i 1500', 'Hash Code49', 'Hash Code1497', '-f 1497', 'chunk {}', 'Hash Code-1264006873', and finally 'Server port: 11111'.

```
Initial_State (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Dec 23, 2013, 12:35:22 PM)
peer -p Peers c HasChunks_p1 -m 4 -i 1 -f MasterChunkFile
Hash Code3436898
Hash Code1507
-p 1507
chunk {}
Hash Code76991281
Hash Code99
Hash Code1107757984
Hash Code1504
-m 1504
Hash Code52
Hash Code1500
-i 1500
Hash Code49
Hash Code1497
-f 1497
chunk {}
Hash Code-1264006873
Server port: 11111
```

Markers Properties Servers Data Source Explorer Snippets Console

Initial_State (1) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Dec 23, 2013, 12:36:54 PM)

```
peer -p Peers -c HasChunks_p1 -m 4 -i 1 -f MasterChunkFile
```

```
Hash Code3436898
Hash Code1507
-p 1507
chunk {}
Hash Code76991281
Hash Code99
Hash Code1107757984
Hash Code1504
-m 1504
Hash Code52
Hash Code1500
-i 1500
Hash Code49
Hash Code1497
-f 1497
chunk {}
Hash Code-1264006873
Server port: 11111
```

```
peer -p Peers -c HasChunks_p2 -m 4 -i 2 -f MasterChunkFile
```

```
Hash Code3436898
Hash Code1507
-p 1507
Hash Code76991281
Hash Code1494
-c 1494
Hash Code1107757985
Hash Code1504
-m 1504
Hash Code52
Hash Code1500
-i 1500
Hash Code50
Hash Code1497
-f 1497
chunk {}
Hash Code-1264006873
Server port: 22222
```

```
peer -p Peers -c HasChunks_p2 -m 4 -i 2 -f MasterChunkFile
```

```
Hash Code3436898
Hash Code1507
-p 1507
Hash Code76991281
Hash Code1494
-c 1494
Hash Code1107757985
Hash Code1504
-m 1504
Hash Code52
Hash Code1500
-i 1500
Hash Code50
Hash Code1497
-f 1497
chunk {}
Hash Code-1264006873
Server port: 22222
Server receive:
WHOHAS 2b2f1b56c100e1b5d03cd1d2a889201373887825 886151b7898e46facb4f8af567d347348a3e6580
Server send:
IHAVE efc711f33ea198a5e97d26ea55fc4cd5349ad924 dd56180640440dcb9a71f8ce5a61e6f999b8ed78
```

To find hosts to download from. The requesting peer sends a “WHOHAS<list>” (list is a chunk hashes it wants to download) request to all peers. The list specifies the SHA-1 hashes of the chunks it wants to retrieve. The maximum packet size for UDP in this assignment is 1500 bytes. So it will be split in multiple WHOHAS queries if the list is too large for a single packet. Chunk hashes have a fixed length of 20 bytes. When a WHOHAS query has been received a peer sends back the list of chunks it contains using IHAVE<list> replay. The list will contain the list of hashes for the chunks it has.

Packet Type	Code
WHOHAS	0
IHAVE	1
GET	2
DATA	3
ACK	4
DENIED	5

The requesting peer looks at all IHAVE replies and decides which remote peer to fetch each of the chunks from. It then downloads it using GET<chunk-hash> requests. When the peer receives the Get requests it will send back multiple DATA packets to the requesting peer. It will then send back an ACK packet to the sent to notification that it has received the packet. All packets have a common header which we will need to use in order to be understood.

1. Magic Number [2 bytes]
2. Version Number [1 byte]
3. Packet Type [1 byte]
4. Header Length [2 bytes]
5. Total Packet Length [2 bytes]
6. Sequence Number [4 bytes]
7. Acknowledgment Number [4 bytes]

Our Host and Receiver classes reflect these actions, reading datagrams, and allowing the transfer of data.

We implemented a model package which contains a Packet_Info, Peer, ReadMasterChunkList classes. Packet_info helps to establish to read and interpret the packet header as mentioned earlier, by filling in the packet’s information by reading the header. The Peer class creates a “Peer” the peer declares an id, string address, and a port number. The id we set as 1,2 (we could do more but that’s a little too ambitious). The ip address we set as 127.0.01, and for the Port number we just set it as 11111,22222 .

ReadMasterChunkList class, by using the HashMap java utility allows up to read the chunks in our file by interpreting the SHA-1 hash values.

The service package contains some of the most complex parts of our program at least initially to create. FileService verifies the file’s SHA1 checksum. It checks the name of the file that has to be verified and the expected check sum. Its true if the expected SHA1 checksum matches the files SHA1 checksum; false otherwise. In the converttoHex method we return the

Hexadecimal Code. The ReadMasterDataFile/ReadHasChunkFile method reads the Master Data/Chunk file and does this by comparing the hash values. ReadPeerList file reads the list of peers in the program. The JoinFile method puts all of the chunks together and writes this data out. Packet Service is much less complex; this class reads out packet info, byte by byte.

The State Package contains the Instance class which creates the data for the instance of connection (peer node, peer list path, etc). The Thread package contains CheckList, HostThread, and ReceiveThread classes. These classes create threads for the Chunklist, running host, and receiver.

Putting all of this together is how we implement a p2p program at a basic level. Please remember to read the README file, and enjoy the reading “The Trooper” over the holidays!