Project Description

Last Updated: 9/23/2020 12:26 AM

The project logistics are in a separate document on Canvas under 'Files/Project'.

Midpoint check: Nov. 2, 11:59 pm

Final Project: Dec. 9, 11:59 pm

Project Overview

In this project, you are asked to write a P2P file sharing software similar to BitTorrent. You can complete the project in Java or C/C++. There will be no extra credit for C/C++.

BitTorrent is a popular P2P protocol for file distribution. Among its interesting features, you are asked to implement the choking-unchoking mechanism which is one of the most important features of BitTorrent. In the following *Protocol Description* section, you can read the protocol description, which has been modified a little bit from the original BitTorrent protocol. After reading the protocol description carefully, you must follow the implementation specifics shown in the *Implementation Specifics* section.

Protocol Description

This section outlines the protocol used to establish the file management operations between peers. All operations are assumed to be implemented using a reliable transport protocol (i.e. TCP). The interaction between two peers is symmetrical: Messages sent in both directions look the same.

The protocol consists of a handshake followed by a never-ending stream of length-prefixed messages.

Whenever a connection is established between two peers, each of the peers of the connection sends to the other one the handshake message before sending other messages.

handshake message

The handshake consists of three parts: handshake header, zero bits, and peer ID. The length of the handshake message is 32 bytes. The handshake header is 18-byte string 'P2PFILESHARINGPROJ', which is followed by 10-byte zero bits, which is followed by 4-byte peer ID which is the integer representation of the peer ID.

handshake header	zero bits	peer ID

actual messages

After handshaking, each peer can send a stream of actual messages. An actual message consists of 4-byte message length field, 1-byte message type field, and a message payload with variable size.

message length	message type	message payload

The 4-byte message length specifies the message length in bytes. It does not include the length of the message length field itself.
The 1-byte message type field specifies the type of the message.
There are eight types of messages.

message type	value
choke	0
unchoke	1
interested	2
not interested	3
have	4
bitfield	5
request	6
piece	7

Now let me introduce the message payload of the above messages.

choke, unchoke, interested, not interested

'choke', 'unchoke', 'interested' and 'not interested' messages have no payload.

have

'have' messages have a payload that contains a 4-byte piece index field.

bitfield

'bitfield' messages is only sent as the first message right after handshaking is done when a connection is established. 'bitfield' messages have a bitfield as its payload. Each bit in the bitfield payload represents whether the peer has the corresponding piece or not. The first byte of the bitfield corresponds to piece indices 0 – 7 from high bit to low bit, respectively. The next one corresponds to piece indices 8 – 15, etc. Spare bits at the end are set to zero. Peers that don't have anything yet may skip a 'bitfield' message.

request

'request' messages have a payload which consists of a 4-byte piece index field. Note that 'request' message payload defined here is different from that of BitTorrent. We don't divide a piece into smaller subpieces.

piece

'piece' messages have a payload which consists of a 4-byte piece index field and the content of the piece.

behavior of peers

Now let us see how the protocol works.

handshake and bitfield

Suppose that peer A tries to make a TCP connection to peer B. Here we describe thye behavior of peer A, but peer B should also follow the same procedure as peer A. After the TCP connection is established, peer A sends a handshake message to peer B. It also receives a handshake message from peer B and checks whether peer B is the right neighbor. The only thing to do is to check whether the handshake header is right and the peer ID is the expected one.

After handshaking, peer A sends a 'bitfield' message to let peer B know which file pieces it has. Peer B will also send its 'bitfield' message to peer A, unless it has no pieces.

If peer A receives a 'bitfield' message from peer B and it finds out that peer B has pieces that it doesn't have, peer A sends 'interested' message to peer B. Otherwise, it sends 'not interested' message.

choke and unchoke

The number of concurrent connections on which a peer uploads its pieces is limited. At a moment, each peer uploads its pieces to at most k preferred neighbors and 1 optimistically unchoked neighbor. The value of k is given as a parameter when the program starts. Each peer uploads its pieces only to preferred neighbors and an optimistically unchoked neighbor. We say these neighbors are unchoked and all other neighbors are choked.

Each peer determines preferred neighbors every p seconds. Suppose that the unchoking interval is p. Then every p seconds, peer A reselects its preferred neighbors. To make the decision, peer A calculates the downloading rate from each of its neighbors, respectively, during the previous unchoking interval. Among neighbors that are interested in its data, peer A picks k neighbors that has fed its data at the highest rate. If more than two peers have the same rate, the tie should be broken randomly. Then it unchokes those preferred neighbors by sending 'unchoke' messages and it expects to receive 'request' messages from them. If a preferred neighbor is already unchoked, then peer A does not have to send 'unchoke' message to it. All other neighbors previously unchoked but not

selected as preferred neighbors at this time should be choked unless it is an optimistically unchoked neighbor. To choke those neighbors, peer A sends 'choke' messages to them and stop sending pieces.

If peer A has a complete file, it determines preferred neighbors **randomly** among those that are interested in its data rather than comparing downloading rates.

Each peer determines an optimistically unchoked neighbor every *m* seconds. We say *m* is the optimistic unchoking interval. Every *m* seconds, peer A reselects an optimistically unchoked neighbor **randomly** among neighbors that are choked at that moment but are interested in its data. Then peer A sends 'unchoke' message to the selected neighbor and it expects to receive 'request' messages from it.

Suppose that peer C is randomly chosen as the optimistically unchoked neighbor of peer A. Because peer A is sending data to peer C, peer A may become one of peer C's preferred neighbors, in which case peer C would start to send data to peer A. If the rate at which peer C sends data to peer A is high enough, peer C could then, in turn, become one of peer A's preferred neighbors. Note that in this case, peer C may be a preferred neighbor and optimistically unchoked neighbor at the same time. This kind of situation is allowed. In the next optimistic unchoking interval, another peer will be selected as an optimistically unchoked neighbor.

interested and not interested

Regardless of the connection state of choked or unchoked, if a neighbor has some interesting pieces, then a peer sends 'interested' message to the neighbor. Whenever a peer receives a 'bitfield' or 'have' message from a neighbor, it determines whether it should send an 'interested' message to the neighbor. For example, suppose that peer A makes a connection to peer B and receives a 'bitfield' message that shows peer B has some pieces not in peer A. Then, peer A sends an 'interested' message to peer B. In another example, suppose that peer A receives a 'have' message from peer C that contains the index of a piece not in peer A. Then peer A sends an 'interested' message to peer C.

Each peer maintains bitfields for all neighbors and updates them whenever it receives 'have' messages from its neighbors. If a neighbor does not have any interesting pieces, then the peer sends a 'not interested' message to the neighbor. Whenever a peer receives

a piece completely, it checks the bitfields of its neighbors and decides whether it should send ‘not interested’ messages to some neighbors.

request and piece

When a connection is unchoked by a neighbor, a peer sends a ‘request’ message for requesting a piece that it does not have and has not requested from other neighbors. Suppose that peer A receives an ‘unchoke’ message from peer B. Peer A selects a piece **randomly** among the pieces that peer B has, and peer A does not have, and peer A has not requested yet. Note that we use a **random selection strategy**, which is **not** the rarest first strategy usually used in BitTorrent. On receiving peer A’s ‘request’ message, peer B sends a ‘piece’ message that contains the actual piece. After completely downloading the piece, peer A sends another ‘request’ message to peer B. The exchange of request/piece messages continues until peer A is choked by peer B or peer B does not have any more interesting pieces. The next ‘request’ message should be sent after the peer receives the piece message for the previous ‘request’ message. Note that this behavior is different from the pipelining approach of BitTorrent. This is less efficient but simpler to implement. Note also that you don’t have to implement the ‘endgame mode’ used in BitTorrent. So we don’t have the ‘cancel’ message.

Even though peer A sends a ‘request’ message to peer B, it may not receive a ‘piece’ message corresponding to it. This situation happens when peer B re-determines preferred neighbors or optimistically unchoked a neighbor and peer A is choked as the result before peer B responds to peer A. Your program should consider this case.

Implementation Specifics

Configuration files

In the project, there are two configuration files which the peer process should read. The common properties used by all peers are specified in the file *Common.cfg* as follows:

NumberOfPreferredNeighbors 2
UnchokingInterval 5
OptimisticUnchokingInterval 15
FileName TheFile.dat
FileSize 10000232
PieceSize 32768

The meanings of the first three properties can be understood by their names. The unit of UnchokingInterval and OptimisticUnchokingInterval is in seconds. The FileName property specifies the name of a file in which all peers are interested. FileSize specifies the size of the file in bytes. PieceSize specifies the size of a piece in bytes. In the above example, the file size is 10,000,232 bytes and the piece size is 32,768 bytes. Then the number of pieces of this file is 306. Note that the size of the last piece is only 5,992 bytes. Note that the file *Common.cfg* serves like the metainfo file in BitTorrent. Whenever a peer starts, it should read the file *Common.cfg* and set up the corresponding variables.

The peer information is specified in the file *PeerInfo.cfg* in the following format:
[peer ID] [host name] [listening port] [has file or not]

The following is an example of file *PeerInfo.cfg*.

1001 lin114-00.cise.ufl.edu 6008 1
1002 lin114-01.cise.ufl.edu 6008 0
1003 lin114-02.cise.ufl.edu 6008 0
1004 lin114-03.cise.ufl.edu 6008 0
1005 lin114-04.cise.ufl.edu 6008 0
1006 lin114-05.cise.ufl.edu 6008 0

Each line in file *PeerInfo.cfg* represents a peer. The first column is the peer ID, which is a positive integer number. The second column is the host name where the peer is. The third column is the port number at which the peer listens. The port numbers of the peers may be different from each other. The fourth column specifies whether it has the file or not. We only have two options here. '1' means that the peer has the complete file and '0' means that the peer does not have the file. We do not consider the case where a peer has only some pieces of the file. Note, however, that more than one peer may have the file. Note also that the file *PeerInfo.cfg* serves like a tracker in BitTorrent.

Suppose that your working directory for the project is '~/project/'. All the executables and configuration files needed for running the peer processes should be in this directory. However, the files specific to each peer should be in the subdirectory 'peer_[peerID]' of the working directory. We will explain which files are in the subdirectory later.

Peer process

The name of the peer process should be **'peerProcess'** and it takes the peer ID as its parameter. You need to start the peer processes in the order specified in the file *PeerInfo.cfg* on the machine specified in the file. You should specify the peer ID as a parameter.

For example, given the above *PeerInfo.cfg* file, to start the first peer process, you should log into the machine lin114-01.cise.ufl.edu, go to the working directory, and then enter the following command:
> peerProcess 1001

If you implement the peer process in Java, the command will be
> java peerProcess 1001

(The above procedure may be unpleasant. A more effective way to do the job will be introduced in the section '*How to start processes on remote machines*'.)
Then the peer process starts and reads the file *Common.cfg* to set the corresponding variables. The peer process also reads the file *PeerInfo.cfg*. It will find that the [has file or not] field is 1, which means it has the complete file, it sets all the bits of its bitfield to be 1. (On the other hand, if the [has file or not] field is 0, it sets all the bits of its bitfield to 0.) Here, the bitfield is a data structure where your peer process manages the pieces. You have the freedom in how to implement it. This peer also finds out that it is the first peer; it will just listen on the port 6008 as specified in the file. Being the first peer, there are no other peers to make connections to.

The peer that has just started should make TCP connections to all peers that started before it. For example, the peer process with peer ID 1003 in the above example should make TCP connections to the peer processes with peer ID 1001 and peer ID 1002. In the same way, the peer process with peer ID 1004 should make TCP connections to the peer processes with peer ID 1001, 1002, and 1003. When all processes have started, all peers are connected with all other peers. Note that this behavior is different from that of BitTorrent. It is just for simplifying the project.

When a peer is connected to at least one other peer, it starts to exchange pieces as described in the protocol description section. A peer terminates when it finds out that all the peers, **not just itself**, have downloaded the complete file.

How to start processes on remote machines

Note: *Depending on where you work from, the following has been reported to have problems due to SSH. But, you may be able to use the provided information to find a customized solution. Some student groups have been able to get around the problem and I will provide their solutions on Canvas. If you cannot get around the difficulties, you can always start the peer processes manually in your demo.*

For convenience, you will be given a Java program for starting your peer processes on remote machines.

For your information, let us see how the Java program starts peer processes on remote machines. To start a peer process on a remote machine, we can use the remote login facility **ssh** in UNIX/LINUX systems. To execute the **ssh** command from a Java program, we use the exec() method of the Runtime class.

First, you need to get the current working directory, which is the directory where you start the Java program, as follows:

```
String workingDir = System.getProperty("user.dir");
```

Second, you invoke exec() method as in the following:

```
Runtime.getRuntime().exec("ssh " + hostname + " cd " + workingDir + " ; " +
peerProcessName + "  " + peerProcessArguments );
```

Here *ssh* is the command to start a process that will run on the host given by the program variable hostname. Immediately after the host name, you need to specify the *cd* command to make the current working directory of the new remote process be the same as the directory where you start the Java program so that the peer process can read the configuration files. The variable *workingDir* specifies the full path name of the directory where you invoked the Java program. After the semicolon, you should specify the name of the peer process to run on the remote machine. You need to modify the given Java program so that *peerProcessName* contains the correct string. For example, if you implement the peer process in C or C++, the variable should contain '*peerProcess*'. If you implement the peer process in Java, it should contain '*java peerProcess*'. The variable *peerProcessArguments* should contain the necessary arguments to the peer process. In our case, it should contain the peer ID.

Now, you run the Java program, *startRemotePeers*, at the working directory as follows:
> java startRemotePeers

Then the program reads file *PeerInfo.cfg* and starts peers specified in the file one by one. It terminates after starting all peers.

File handling

The file handling method is up to you except each peer should use the corresponding subdirectory 'peer_[peerID]' to contain the complete files or partial files. For example, if the working directory is '~/project/', then the peer process with peer ID 1001 should use the subdirectory '~/project/peer_1001/', the peer process with peer ID 1002 should use the subdirectory '~/project/peer_1002/', and so on. You should maintain the complete files or partial files for a peer in the corresponding subdirectory.
For those peer processes specified in the file *PeerInfo.cfg* that have the complete file, you need to make sure that the corresponding subdirectories for those peers actually contain the file before you start them.

Writing log

Each peer should write its log into the log file 'log_peer_[peerID].log' at the working directory. For example, the peer with peer ID 1001 should write its log into the file '~/project/log_peer_1001.log'.
Each peer process should generate logs for a peer as follows.

TCP connection

Whenever a peer makes a TCP connection to other peer, it generates the following log:
[Time]: Peer [peer_ID 1] makes a connection to Peer [peer_ID 2].

[peer_ID 1] is the ID of peer who generates the log, [peer_ID 2] is the peer connected from [peer_ID 1]. The [Time] field represents the current time, which contains the date, hour, minute, and second. The format of [Time] is up to you.

Whenever a peer is connected from another peer, it generates the following log:
[Time]: Peer [peer_ID 1] is connected from Peer [peer_ID 2].

[peer_ID 1] is the ID of peer who generates the log, [peer_ID 2] is the peer who has

made TCP connection to [peer_ID 1].

change of preferred neighbors

Whenever a peer changes its preferred neighbors, it generates the following log:
[Time]: Peer [peer_ID] has the preferred neighbors [preferred neighbor ID list].

[preferred neighbor list] is the list of peer IDs separated by comma ‘,’.

change of optimistically unchoked neighbor

Whenever a peer changes its optimistically unchoked neighbor, it generates the following log:
[Time]: Peer [peer_ID] has the optimistically unchoked neighbor [optimistically unchoked neighbor ID].

[optimistically unchoked neighbor ID] is the peer ID of the optimistically unchoked neighbor.

unchoking

Whenever a peer is unchoked by a neighbor (which means when a peer receives an unchoking message from a neighbor), it generates the following log:
[Time]: Peer [peer_ID 1] is unchoked by [peer_ID 2].

[peer_ID 1] represents the peer who is unchoked and [peer_ID 2] represents the peer who unchokes [peer_ID 1].

choking

Whenever a peer is choked by a neighbor (which means when a peer receives a choking message from a neighbor), it generates the following log:
[Time]: Peer [peer_ID 1] is choked by [peer_ID 2].

[peer_ID 1] represents the peer who is choked and [peer_ID 2] represents the peer who chokes [peer_ID 1].

receiving ‘have’ message

Whenever a peer receives a ‘have’ message, it generates the following log:
[Time]: Peer [peer_ID 1] received the ‘have’ message from [peer_ID 2] for the piece

[piece index].

[peer_ID 1] represents the peer who received the ‘have’ message and [peer_ID 2] represents the peer who sent the message. [piece index] is the piece index contained in the message.

receiving ‘interested’ message

Whenever a peer receives an ‘interested’ message, it generates the following log:
[Time]: Peer [peer_ID 1] received the ‘interested’ message from [peer_ID 2].

[peer_ID 1] represents the peer who received the ‘interested’ message and [peer_ID 2] represents the peer who sent the message.

receiving ‘not interested’ message

Whenever a peer receives a ‘not interested’ message, it generates the following log:
[Time]: Peer [peer_ID 1] received the ‘not interested’ message from [peer_ID 2].

[peer_ID 1] represents the peer who received the ‘not interested’ message and [peer_ID 2] represents the peer who sent the message.

downloading a piece

Whenever a peer finishes downloading a piece, it generates the following log:
[Time]: Peer [peer_ID 1] has downloaded the piece [piece index] from [peer_ID 2]. Now the number of pieces it has is [number of pieces].

[peer_ID 1] represents the peer who downloaded the piece and [peer_ID 2] represents the peer who sent the piece. [piece index] is the piece index the peer has downloaded. [number of pieces] represents the number of pieces the peer currently has.

completion of download

Whenever a peer downloads the complete file, it generates the following log:
[Time]: Peer [peer_ID] has downloaded the complete file.

Important Note

For your demo, it is preferable that you demonstrate your program on the CISE machines. For initial development and testing, you may find it more convenient to work on your own machine.

When using CISE machines, please make sure that there are no run-away processes when debugging and testing your program. Your peer processes must terminate after all peers have downloaded the file.

When you are testing your program, do not use port number 6008, which is shown in this example. Use your own port number so that it does not conflict with other students' choices.