Project Description

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++, preferably Java. 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 for file distribution among the peers. All operations should be implemented using a reliable transport protocol (i.e. TCP).

The protocol is for file sharing/distribution (a.k.a. collaborative download). There is a single file to be downloaded by all the participating peers. The file is initially possessed by one or more peers. The file is logically divided into *pieces* of pre-defined size, and each piece has an index. The first piece has the index 0, the second piece has index 1, and so on. The peers exchange information about who have what pieces, make requests for and receive missing pieces.

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 a handshake message before sending other messages.

handshake message

The handshake message consists of three fields: handshake header, zero bits, and peer ID. The length of the handshake message is 32 bytes. The handshake header is a 5-byte string 'HELLO', which is followed by 23-byte zero bits. The last field is a 4-byte peer ID, which is an integer.

handshake header	zero bits	peer ID

normal messages

After the handshake, each peer can send a stream of normal messages. A normal message consists of a 4-byte message length field, 1-byte message type field, and a message payload of a 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

We now describe the message payload of the above messages.

choke, unchoke, interested, not interested

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

have

Each 'have' message has a payload that contains a 4-byte index of a file piece. It indicates that the sender has the piece referred to by the index.

bitfield

A 'bitfield' message is only sent as the first message right after the connection is established and the handshake is completed. Each 'bitfield' message has a bitfield as its payload. Each bit in the bitfield represents whether the sending peer has the corresponding piece or not. The first byte of the bitfield corresponds to the 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 do not have any pieces yet may skip a 'bitfield' message.

request

Each 'request' message has a 4-byte payload, which contains the index of the piece

being requested. (Note: the 'request' message payload defined here is different from that in BitTorrent. We do not divide a piece into smaller subpieces.)

piece

Each 'piece' message has a payload consisting of a 4-byte piece index and the content of the piece. It is used to send a piece from the sender to a receiving peer.

Behavior of the 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. After the TCP connection is established, peer A sends a handshake message to peer B. It should receive a handshake message from peer B. Peer A should check whether the received handshake message header is correct and the peer ID is the expected one.

After the handshake, 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 finds out that peer B has pieces that it doesn't have, peer A sends an 'interested' message to peer B. Otherwise, it sends a 'not interested' message.

choke and unchoke

The number of concurrent connections through which a peer uploads its pieces is limited. At any 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 may upload its pieces only to the preferred neighbors and an optimistically-unchoked neighbor. We say these neighbors are unchoked and all other neighbors are choked.

Each peer determines the preferred neighbors every p seconds. In other words, peer A reselects its preferred neighbors every p seconds. To make the decision, peer A calculates the downloading rate from each of its neighbors during the previous p-second unchoking interval. Among the neighbors *that are interested in* its data, peer A picks the k neighbors that have transmitted data to A at the highest rates. Then, peer A unchokes those selected neighbors (which become the preferred neighbors) by sending 'unchoke' messages. It expects to receive 'request' messages from them. If a preferred

neighbor is already unchoked, then peer A does not have to send an 'unchoke' message to it. All other neighbors previously unchoked but not selected as preferred neighbors at this moment should be choked. To choke those neighbors, peer A sends 'choke' messages to them and stops sending pieces. An exception to this is the optimistically-unchoked neighbor.

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

Optimistic Unchoking

Each peer determines an optimistically-unchoked neighbor every *m* seconds. Every *m* second is considered an optimistic-unchoking interval. Every *m* seconds, peer A selects an optimistically-unchoked neighbor *randomly* among the neighbors that are choked at that moment but are interested in A's data. Then, peer A sends an 'unchoke' message to the selected neighbor, and expects to receive a 'request' messages from it. Peer A then sends the requested data.

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

interested and not interested

Regardless of the connection state being choked or unchoked, if a neighbor has some pieces that peer A does not have, then peer A sends an '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’ or ‘bitfield’ messages from the 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 unchoked by a neighbor, a peer sends a ‘request’ message to request 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, peer A does not have, and peer A has not made a request yet. Note that, here, 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, our protocol does not have the ‘cancel’ message.

Even if 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-selects its preferred neighbors or optimistically unchokes another neighbor and, as a result, peer A is choked before peer B responds to peer A’s request. Your program should be able to handle this case.

have

Whenever peer A receives a piece completely from peer B, peer A notifies all its neighbors (including choked, unchoked neigbhors and peer B) about the availability of the newly received piece by a ‘have’ message. The ‘bitfield’ messages also serve the purpose of informing the neighbors about the availability of pieces, but are used less frequently. A peer sends a ‘bitfield’ message after the initial handshake.

Implementation Specifics

Configuration files

In the project, there are two configuration files that 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 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 each 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 sun114-11.cise.ufl.edu 6008 1
1002 sun114-12.cise.ufl.edu 6008 0
1003 sun114-13.cise.ufl.edu 6008 0
1004 sun114-14.cise.ufl.edu 6008 0
1005 sun114-21.cise.ufl.edu 6008 0
1006 sun114-22.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 some but not all the pieces of the file. Note that more than one peer may have the file. 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 processes

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 sun114-11.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 that your peer process manages to keep track of the pieces. You have the freedom in how to implement it. This peer also finds that it is the first peer; it will just listen on the port 6008 as specified in the

configuration file. Being the first peer, there are no other peers to make connections to.

Each other peer, after starting, should make TCP connections to all peers that have 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. After all processes have started, all peers are connected with each other. Note that this behavior is different from that of BitTorrent. It simplifies 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

You will be given a Java program that can be used for starting your peer processes on remote machines. The program and its usage will be posted later.

For your information, let us describe how the Java program works. To start a peer process on a remote machine, we can use the remote login facility **ssh** in Unix 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 store 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 another 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 that generates the log. [peer_ID 2] is the remote peer which [peer_ID 1] initiates a connection with. 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 that generates the log, [peer_ID 2] is the peer that has initiated a 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 the 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 the 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 a ‘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 an ‘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 a ‘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 finishes downloading the complete file, it generates the following log:
[Time]: Peer [peer_ID] has downloaded the complete file.

Protocol robustness

Remember that you are building a distributed system and your implementation should be robust against contingencies and un-specified behaviors. In particular, during message exchange, you always should consider the possibility that the other peer is not responding as expected and handle that possibility gracefully. For instance, when peer A sends an 'unchoke' message to peer B, it expects to receive a 'request' message for peer B. However, there are situations where peer B does not send a 'request' message; it may even send a 'not interested' message'. Peer A should have a timeout mechanism after sending the 'unchoke' message. If the timer expires, peer A may move on to unchoke another peer, if that makes sense. In another example, peer A sends a 'request' message to peer B and expects a 'piece' message. What if peer B does not send a 'piece' message? Your code should consider this and other similar issues.

Important Note

Your program should run on the CISE machines. You are required to demonstrate your program on the 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 the port number 6008, which is used in our examples. Use your own port number to reduce the chance of collision with other students' choices.

How to submit your project

Please submit on E-Learning.