

REPORT

Computer Network

Assignment2

- BitTorrent Swarming Protocol

2015004239 정성운

Assignment2

SeongWoon Jeong edited this page 8 minutes ago · 7 revisions

Computer Network - ENE4019

Programming Assignment2

Assignment2 (TCP Socket Programming - BitTorrent Swarming Protocol)

professor. 이춘화

1. 개요

- TCP Socket을 이용해 BitTorrent의 Swarming Protocol을 구현한다.
 - TCP(Transmisson Control Protocol) : 통신 규약중의 하나로 신뢰성있는 패킷 전송 기능을 제공한다.
 - Socket : 프로세스가 네트워크를 통해서 데이터를 주고받기 위한 통로
 - BitTorrent Swarming Protocol : 특정 파일을 청크단위로 나눈 뒤, 네트워크의 구성원인 피어가 파일을 다운로드받는 클라이언트이자 파일을 전송해주는 서버가 되는 P2P방식의 프로토콜이다.

2. 과제 설명

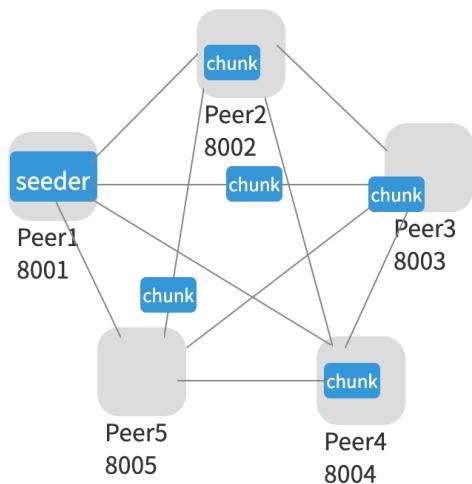
- 요구사항
 - BitTorrent-like 스워밍 프로토콜을 구현한다 내용은 아래의 별도 문서에 기술된 **Swarming Protocol** 을 참고한다.
 - 각 피어는 한 피어로부터 청크를 3개까지 다운로드받은 후, 이 연결을 종료하고 난 후 다른 피어와 새로운 TCP 연결을 맺어 청크를 다운로드받는 과정을 반복한다.
 - Optional Implementation
- Optional Implementation
 - 다음의 선택적 기능을 구현하는 경우에는, 보너스 크레딧이 주어진다. 피어는 3명의 friend를 유지한다. 즉 피어는 TCP 연결을 설정하고 난 후, 다운로드받을 청크가 있는 한 friend관계를 유지한다. 이를 위해 피어는
 - 3개의 download neighbor 쓰레드를 생성하여 수행한다.
 - 피어(upload neighbor thread)는 3개의 청크를 다운로드 받은 후에, 혹은 5초가 지나면 갱신된 bit map을 friend peer에게 보낸다.
 - 10초동안 청크 다운로드가 이루어지지 않으면 해당 피어를 포기하고, 다른 피어와 friend 관계를 맺어 청크 다운로드를 시도한다.
- 주의사항
 - 과제 2의 구현시 클라이언트에서 발생할 수 있는 쓰레드 동기화(thread synchronization) 문제도 고려되어야 한다. 쓰레드 동기화는 두 download neighbor 쓰레드가 동시에 동일한 청크를 다운로드받아 write할 때, 발생할 수 있는 쓰레드간 간섭 문제를 의미한다. 쓰레드 동기화 문제에 대한 이해와 대처기법(즉 자바 프로그램 코드)를 가능한 선에서 제출 문서에 요약해서 정리하도록 한다.
 - 스워밍 프로토콜의 upload neighbor thread와 download neighbor thread의 interaction이 이루어질 수 있도록 command message와 response message가 적절히 정의되어야 한다. 즉 bit map push 메세지와 chunk fetch message 메세지 및 이들의 response 메시지를 정의하고, 그 내용을 제출하는 설계 및 구현 문서에 포함하여야 한다.
 - 프로그램의 설계와 구조의 기술 외에도 제출된 프로그램을 컴파일 및 실행하는 방법을 단계별로 정확하게 기술하여야 한다. 또한 소스코드를 단순히 이 문서에 포함시켜 포함하여 출력하지 않도록 한다. 프로그램 test시 기술된 절차를 그대로 따랐음에도 불구하고 프로그램이 수행되지 않는 경우에는 제출 프로그램이 동작하지 않는 것으로 간주한다.

3. 사용언어 / 환경

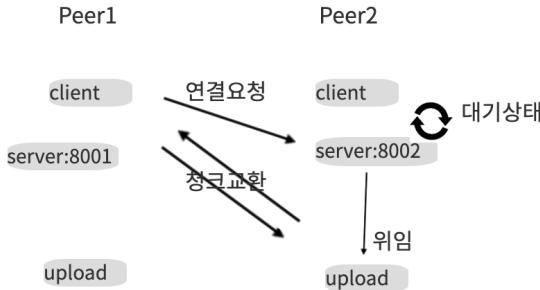
- Java / Mac OS

[프로젝트 설계]

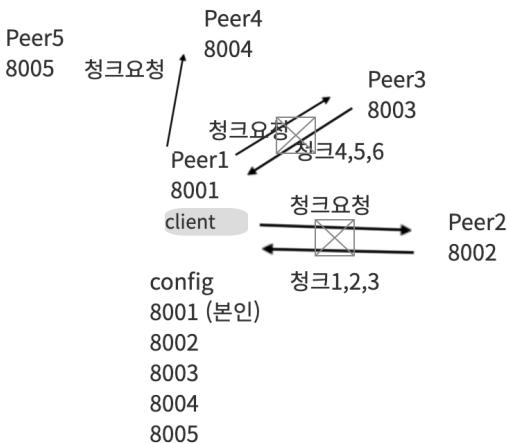
- 기본 프로젝트 설계



해당 프로젝트는 5명의 피어가 10KB로 나누어진 청크를 서로 교환하면서 파일 공유를 하게끔 구현해야 한다. 해당 피어는 파일을 다운로드받는 동시에 다른 피어에게 파일청크를 공유하는 역할을 동시에 수행해야 한다.

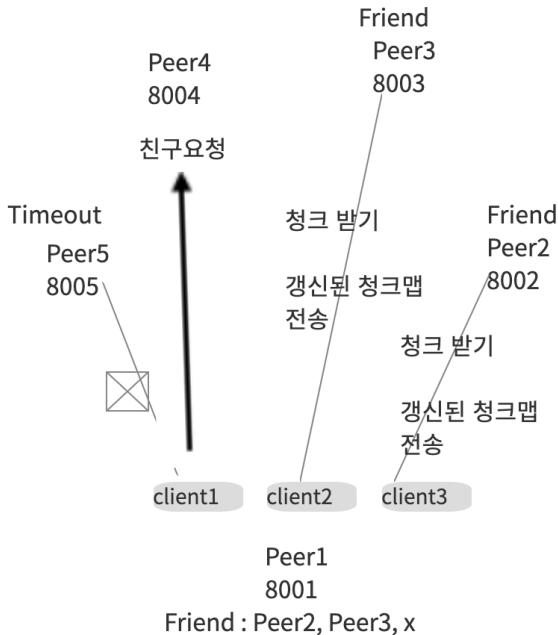


피어는 여러개의 쓰레드를 가지고, 클라이언트 역할을 하는 쓰레드가 서버쓰레드에게 연결을 요청하면 해당 서버쓰레드는 자신의 연결을 Upload 쓰레드에게 위임시키고 자신은 다시 다른 쓰레드의 연결을 대기하는 상태로 바뀐다. 따라서 구현상 [클라이언트 역할을 하는 쓰레드], [server의 역할을 하는 쓰레드], 그리고 [uploader의 역할을 하는 쓰레드], 마지막으로 [해당 쓰레드들을 만들어주는 메인 쓰레드]가 필요할 것이다.



피어는 한 쓰레드에서 청크를 3개 다운로드 받거나, 다운로드 받을 청크가 없으면, 처음에 제공받은 configuration 을 가지고 다른 쓰레드에게 연결 요청을 한다.

- Optional Implement



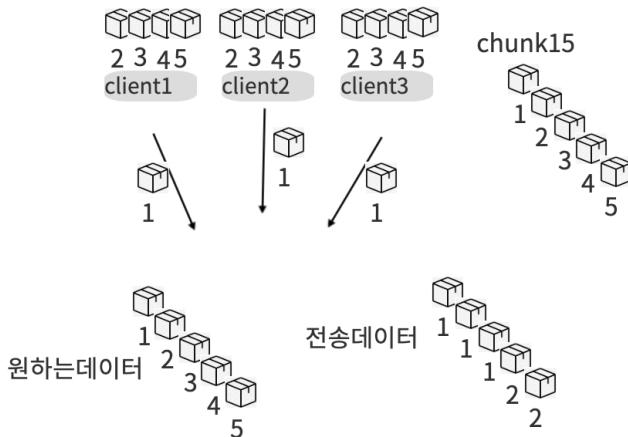
추가 구현은 클라이언트 쓰레드가 3개로 늘어났고, 친구 관계라는 특정 상태가 주어진다. 만약 클라이언트 쓰레드와 해당 피어와 친구가 되면, 해당 피어와 3개의 청크를 받고 난 뒤, 연결을 해제하지 않고 갱신된 청크맵을 재전송하면서 모든 청크에 대한 다운로드가 끝날 때까지 친구관계를 유지한다.

만약, 친구관계를 유지하면서 5초간 청크 다운로드가 이루어지지 않으면, 청크맵을 재전송하고, 10초간 청크 다운로드가 이루어지지 않으면 해당 친구 관계를 포기하고, 새로운 Peer에게 친구요청을 한다.

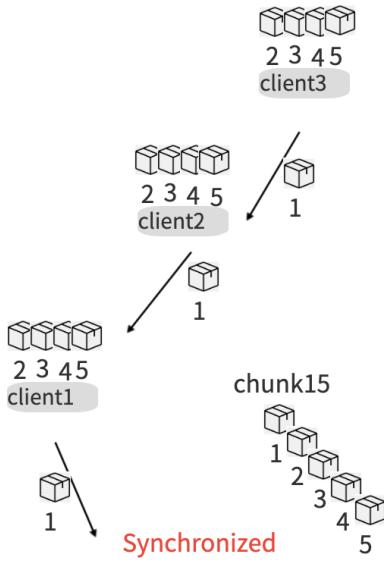
이때 이미 연결되어있는 피어에게 친구요청을 하지 않기 위해서, 쓰레드간에 친구관계를 맺고있는 피어를 판별할 수 있는 FriendArr를 유지해야 한다.

[세부 설계]

데이터동기화



데이터를 쓸때, 클라이언트에서 발생 할 수 있는 쓰레드 동기화(thread synchronization)문제가 고려되어야 한다. 쓰레드 동기화는 두 download neighbor 쓰레드가 동시에 동일한 청크를 다운로드받아 write하게 되면 발생 할 수 있는 쓰레드간 간섭 문제를 의미한다. 이러한 간섭이 일어나면, 원하는 데이터를 제대로 전달받지 못할 수 있다.



[ClientThread.java - line 136]

- 이를 위해 자바 코드에서 Synchronized 를 통해 해당 활동을 수행하는 메서드에 쓰레드 동시접근을 막는 방법이 있다. 이렇게 되면 각 쓰레드가 순차적으로 실행되기 때문에, 같은 청크에 대한 write 요청을 모두 똑같이 처리할 수 있다.

```
// 다른 쓰레드의 동시접근을 막는 synchronized 함수
// friendsIndex를 동시에 증가시키는 경우를 막음.
// 해당 피어의 friendArr의 인덱스를 받아서
synchronized int getFriendArrIndex() {
    Peer.friendsIndex += 1;
    Peer.friendsIndex = Peer.friendsIndex % 3;
    return Peer.friendsIndex;
}

// 다른 쓰레드의 동시접근을 막는 synchronized 함수
// data[][] 배열에 동시에 chunk를 쓰는 경우를 막음.
// uploadThread에서 byte[]를 받아서 Peer의 data[][] 배열에 저장시킨다.
synchronized void readData(int receivedChunknumber) throws IOException {
    Peer.chunkMap[receivedChunknumber] = (byte) 1;
    recvStream.read(Peer.data[receivedChunknumber]);
}
```

프로젝트 내부에서 해당 기법은

[ClientThread.java - line 241]

- ClientThread가 UploadThread로부터 byte[] 데이터를 받아서 데이터 청크를 write 하는 경우에 사용하거나

```
if (response.equals("Received")) {  
    for (int i = 0; i < randomChunk.length; i++) {  
        readData(randomChunk[i]);  
    }  
}  
// 완료 요청.  
sendRequest("Completed");
```

[ClientThread.java - line 178]

- 친구 목록의 같은 인덱스에 동시에 접근해 친구 목록에 값이 중복으로 쓰여지는 경우를 방지하기 위해 사용한다.

```
if (!alreadyConnected) {  
    System.out.println("Client Thread - 친구 리스트에 없습니다. 새로운 연결을 시작합니다.");  
    int friendArrIndex = getFriendArrIndex();  
    Peer.friendsArr[friendArrIndex] = index;
```

Client-Upload Thread간 Command/Response 상세

Command (Client->Upload)	설명	Response (Upload->Client)
Request_Chunks	청크에 대한 요청 (이를 위해 클라이언트는 요청 청크의 총 개수를 전송하 고 random missing chunk Index 전송)	byte[]의 형태로 최대 3개의 청크 데이터 전송
Request_Seeder	해당 피어가 시더인지 여부	Seeder=1, leecher=0
Send_ChunkMap	청크맵을 Client가 Upload에 게로 보냄.	void
Request_Information	해당 피어의 파일정보에 대한 요청	만약 파일정보가 있으면 -> 파일이름, 크기, 청크수, 비트맵 만약 파일정보가 없으면 -> "null"
Already_Friend	해당 피어와 이미 친구관계를 맺고있음	void

[UploadThread.java - line 90]

```
while (true) {  
    response = getResponse();  
    System.out.println("UploadThread 받은요청 : " + response);  
...이하 생략
```

- Upload 쓰레드는 while문을 통해서 계속해서 클라이언트의 request를 기다리고 있음.

[UploadThread.java - line 92]

```
// 요청에 대한 처리。  
switch (response) {  
    // 특정 청크를 요청하는 경우.  
    case ("Request_Chunks"): {  
        // 총 청크 개수 읽기.  
        int total = dis.readInt();  
  
        int[] list = new int[total];  
        // list에 전달받은 랜덤 청크 index 저장시키기.  
        for (int i = 0; i < total; i++) {  
            list[i] = dis.readInt();  
        }  
  
        for (int i = 0; i < total; i++) {  
            // 랜덤 청크 인덱스에 해당하는 청크 데이터를 전송.  
            sendStream.write(Peer.data[list[i]]);  
        }  
        // 잘 전송되었는지 여부 전달받기.  
        response = getResponse();  
        break;  
    }  
...이하 생략
```

- 해당 request가 어떤 경우인지에 따라 처리를 다르게 하기 위해서 switch문을 사용해 case 분류를 함.

[UploadThread.java - line 150]

```
public boolean checkChunkFull() {
    if (this.connectedClientChunkMap == null) {
        return false;
    }
    if (this.connectedClientChunkMap.length == 0) {
        return false;
    }
    for (int i = 0; i < this.connectedClientChunkMap.length; i++) {
        if (this.connectedClientChunkMap[i] == (byte) 0)
            return false;
    }
    return true;
}

... 생략

// 친구 연결을 유지하지 않고 해제하는 경우.
// 1. 시더연결
// 2. 자신이 빙seeder라서 chunkMap null을 전송하는 경우
// 3. 연결된 client의 chunkmap이 가득찬 경우.
// 4. 이미친구인 경우.
if (response.equals("Request_Seeder")
    || (response.equals("Request_Information") && Peer.chunkMap == null)
    || response.equals("Already_Friend")
    || checkChunkFull()) {
    break;
}
... 이하 생략
```

- 해당 조건에 걸리면 친구 연결을 해제하고 쓰레드를 종료시키는데, 해당 조건은 다음과 같다.
 1. 시더판별을 위한 연결이거나
 2. 자신이 파일에 관한 정보가 없는데 파일정보를 요청한 경우
 3. 청크가 이미 가득 찬 경우나
 4. 이미 친구라는 요청이 들어온 경우

- Client Thread는 청크를 다운로드 받기 위해 다음과 같은 로직을 수행함.

- 친구리스트를 보며 해당 피어가 친구인지 확인

- 1-1 친구가 아니면 -> 다음로직 진행

2. 해당 피어에게 파일정보에 대해 요청후 받음.

- 2-1 파일정보가 있다 -> 다음로직 진행.

3. 내 청크맵 리스트 보내기. Request("Send_ChunkMap")

- 3-1 내 청크가 가득차지 않았다 -> 다음로직 진행.

4. 미싱 청크리스트를 찾아 부족한 청크에 대한 요청을 함. Request("Request_Chunks")

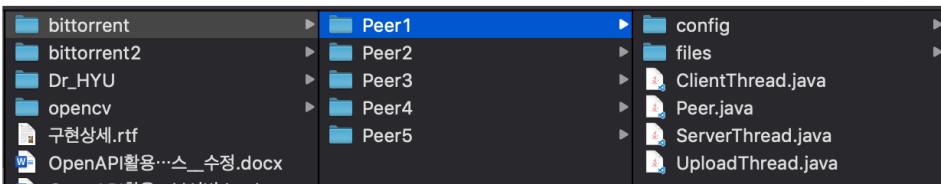
5. 청크를 무사히 받은 뒤 다시 2번으로 돌아감.

- 3-2 내 청크가 가득찼다 -> 연결해제

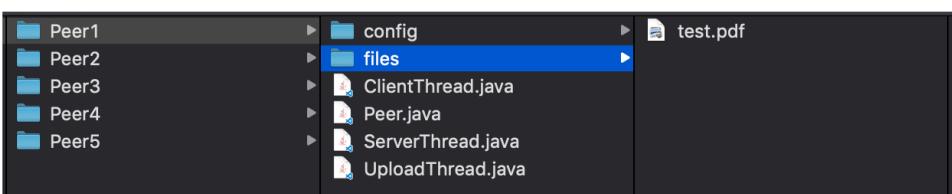
- 2-2 파일정보가 없다 -> 연결해제, 다른 피어 찾기.

- 1-2 친구이면 -> Request("Already_Friend")후 연결해제, 다른 피어 찾기.

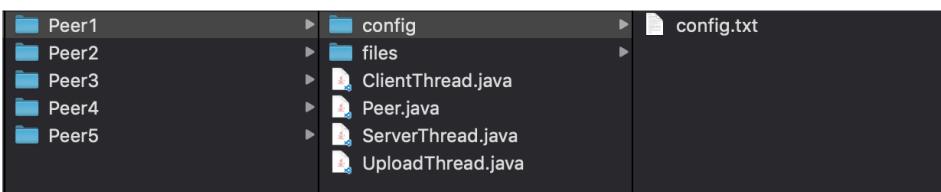
[프로젝트 구조]



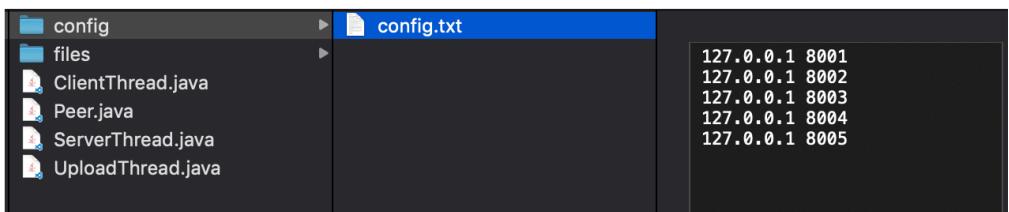
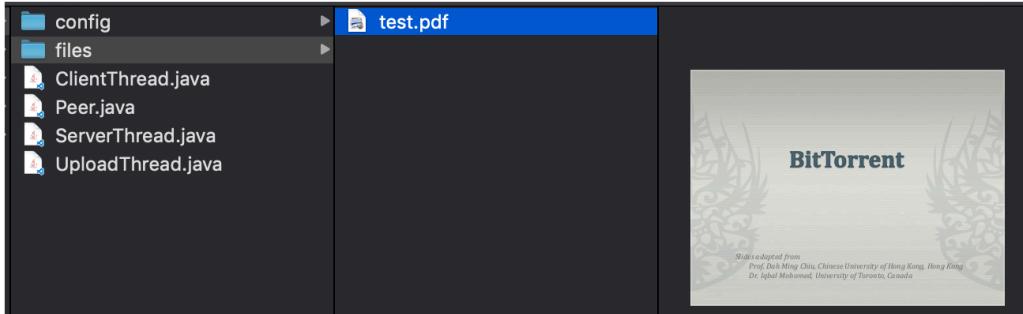
bitTorrent라는 폴더 아래에 Peer1부터 Peer5까지 5개의 폴더가 있고, 각각의 폴더에는 config 폴더와 4개의 java file (ClientThread.java, Peer.java, ServerThread.java, UploadThread.java)가 들어있다.



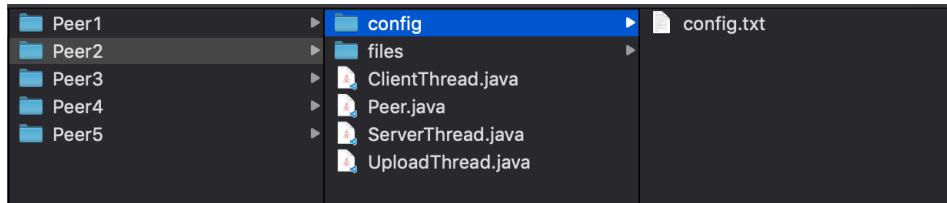
files폴더에는 swarming protocol을 통해 공유하고 싶은 파일이 들어있다. (여기서는 test.pdf) 만약 파일이 들어있다면 해당 피어는 Seeder가 된다.



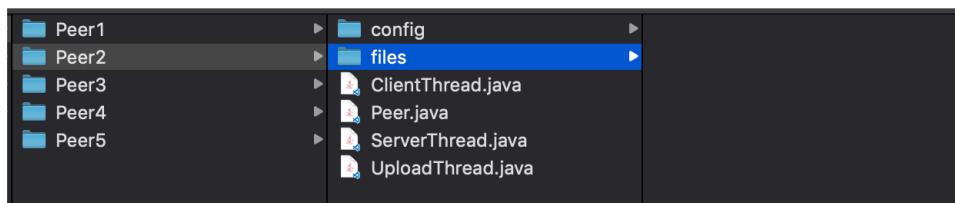
config폴더에는 기본적으로 제공되는 Peer들의 configuartion 정보가 들어있는 config.txt파일이 들어있다.



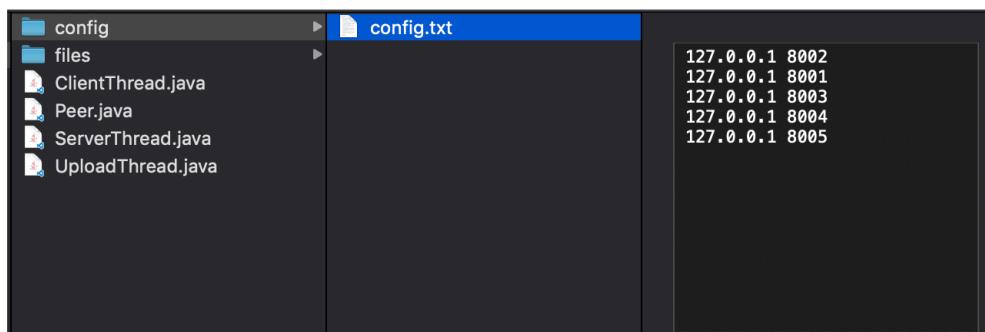
현재 나는 Peer1을 Seeder로 설정하기 위해 files 폴더에 공유하기를 원하는 파일 test.pdf를 넣어두었고, 해당 Peer1의 config.txt파일을 위와같이 설정했다. 파일은 IP와 TCP port number의 쌍으로 이루어져 있으며, 첫번째줄은 자기 자신의 정보이고, 나머지 4줄은 다른 피어들의 정보들이다.



마찬가지로 Peer2폴더 또한 같은 구조로 되어있다.



다만 Peer1만 Seeder로 설정할 것이기 때문에, 해당 files 폴더에 test.pdf파일을 따로 넣어주지 않아 비어있는 상태이다.

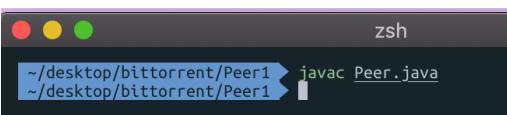


해당 Peer2의 config.txt를 읽어보면 마찬가지로 첫번째줄에 자기 자신의 정보를 담고 있고, 나머지 4줄에 다른 Peer들의 정보를 담고 있는 것을 볼 수 있다.

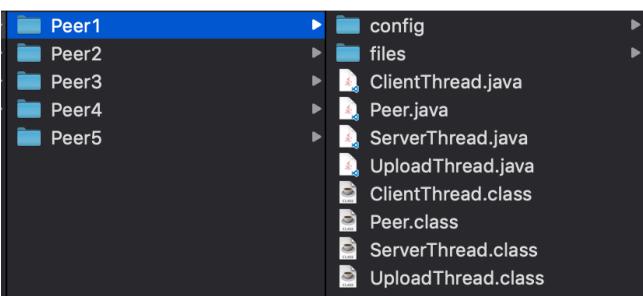
[테스트 시나리오 - MAC]

```
~/desktop/bittorrent/Peer1 ➜
```

터미널을 켜서 Peer1 폴더로 이동한다.



```
~/desktop/bittorrent/Peer1 ➜ javac Peer.java  
~/desktop/bittorrent/Peer1 ➜
```



javac Peer.java를 console에 입력하면, 해당 폴더에 클래스 파일이 생긴다.

```
java
~/desktop/bittorrent/Peer1  javac Peer.java
~/desktop/bittorrent/Peer1  java -Xmx1g Peer config.txt test.pdf
=====
    Peer Start
=====
    Server Thread Start
=====
        Client Thread Start
=====
        Client Thread Start
=====
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
=====
        Peer End
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
=====
        Client Thread Start
=====
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
```

java HeapMemoryOption ClassFileName ConfigFileName TorrentFileName 을 입력한다.

예를들어 나의 테스트 예제 java -Xmx1g Peer config.txt test.pdf는

1gb의 힙을 가진 Peer라는 클래스파일을 실행시키고, config.txt와 test.pdf는 각각 config, files폴더에 있는 config파일과 공유하고자 하는 파일의 이름이다. 힙은 최소한 전송하려는 파일의 크기보다 커야만 하는데, 그 이유는 전송하려는 데이터를 모두 static byte[]안에 저장시키기 때문이다.

이후 해당 콘솔이 아닌 새로운 콘솔을 띄운다.

```
java
~/desktop/bittorrent/Peer2  javac Peer.java
~/desktop/bittorrent/Peer2  java -Xmx1g Peer config.txt test.pdf
=====
    Peer Start
=====
There is no such file in files folder
File Download Start
=====
        Server Thread Start
=====
        Client Thread Start
=====
        Client Thread Start
=====
Client - 소켓 연결을 시도합니다.
Client - 소켓 연결을 시도합니다.
=====
        Peer End
=====
        Client Thread Start
=====
Client - 소켓 연결을 시도합니다.
```

새로운 콘솔에서는 Peer2로 이동한 뒤, 1번 과정에서 했던

javac Peer.java

java -Xmx1g Peer config.txt test.pdf를 진행해주면 된다.

또다시 새로운 콘솔을 3개 더 켜서, 3개의 Peer3, Peer4, Peer5에 관해서도 같은 과정을 진행해 주면 된다.

```
java
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
127.0.0.1:8003 와 연결되었음
127.0.0.1:8005 와 연결되었음
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
```

```
java
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
127.0.0.1:8004 와 연결되었음
127.0.0.1:8001 와 연결되었음
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
```

```
java
=====
Client Thread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
127.0.0.1:8004 와 연결되었음
127.0.0.1:8004 와 연결되었음
127.0.0.1:8001 와 연결되었음
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
```

```
java
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
127.0.0.1:8004 와 연결되었음
127.0.0.1:8003 와 연결되었음
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
Client Thread End
=====
```

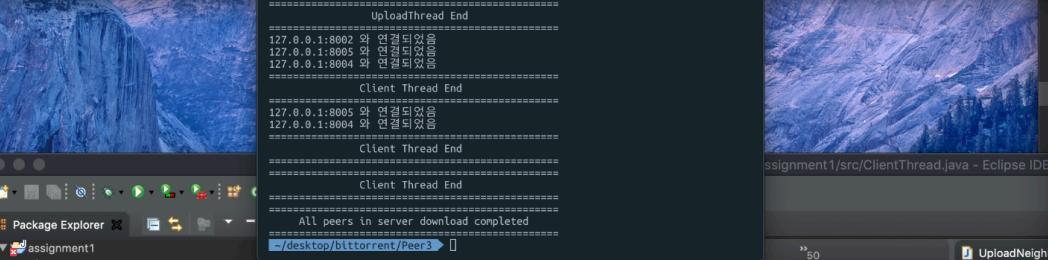
최종적으로, 위와 같은 실행 환경이 구성될 것이다.

```
zsh
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
127.0.0.1:8003 와 연결되었음
127.0.0.1:8005 와 연결되었음
=====
Client Thread End
=====
Client Thread End
=====
=====
All peers in server download completed
=====
-/desktop/bittorrent/Peer1 ➤
```

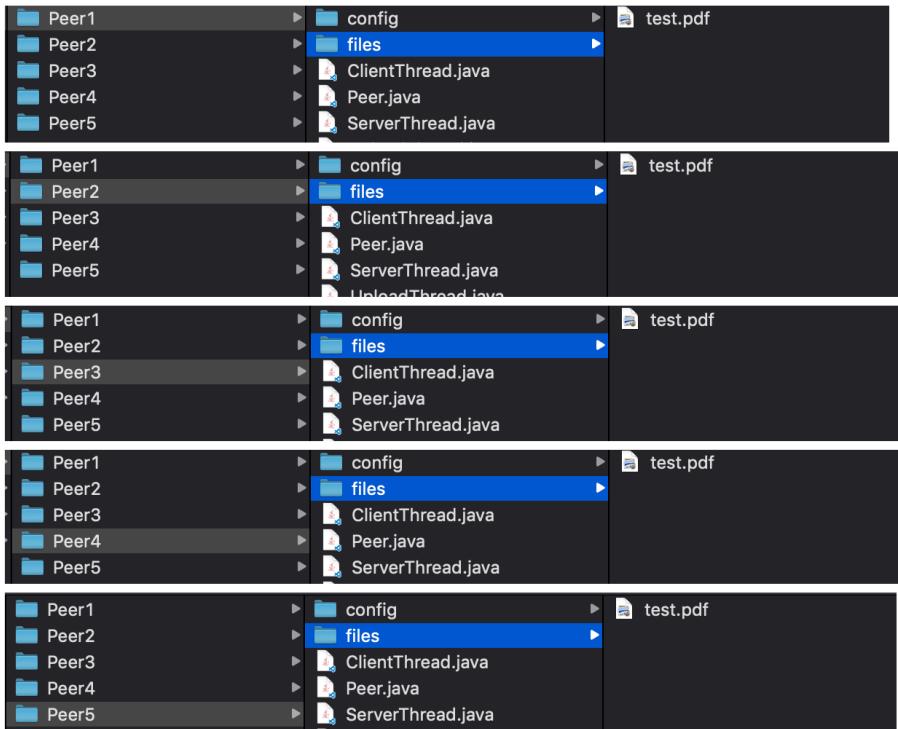
```
=====
Client Thread End
=====
127.0.0.1:8001 와 연결되었음
=====
Client Thread End
=====
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
All peers in server download completed
=====
~/desktop/bittorrent/Peer2 ▶
```

```
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
All peers in server download complete =====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
~/desktop/bittorrent/Peer4 □
```

```
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
completed
```



전송하려는 파일의 크기에 따라 걸리는 시간이 다르겠지만, 파일을 다운로드 받게 되면 서버와 클라이언트, 업로드 쓰레드가 모두 종료되게 되고, 프로세스가 자동으로 종료되면서 콘솔 화면으로 나온다.



이후 해당 폴더로 이동해보면 files 폴더에 시더(Peer1)의 test.pdf파일이 Peer2, Peer3, Peer4, Peer5에게도 다운로드 되어 있는 것을 볼 수 있다.

[콘솔 로그에 대한 분석]

Seeder

```
~/desktop/bittorrent/Peer1 ➤ java -Xmx1g Peer config.txt test.pdf
=====
Peer Start
=====
Server Thread Start
=====
Peer End
=====
Client Thread Start
=====
Client Thread Start
=====
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
=====
Client Thread Start
=====
Client Thread - 파일 다운로드 종료
Client Thread - 파일 만들기.
```

- 코드를 실행시키면 피어 쓰레드가 시작되고, 피어 쓰레드는 서버와 클라이언트 쓰레들을 시작시킨 후 종료된다. 이후 Client Thread가 파일을 찾는데, 파일의 정보가 완전하므로 바로 파일 다운로드를 종료하고 파일을 만들어 덮어씌운다.

```

=====
        UploadThread Start
=====
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
=====
        UploadThread Start
=====
127.0.0.1:8003 와 연결되었음
127.0.0.1:8004 와 연결되었음
127.0.0.1:8002 와 연결되었음
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
=====
        UploadThread Start
=====
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
UploadThread 받은요청 : Request_Information

```

- Client Thread들은 다른 피어의 서버쓰레드와 연결되어 Seeder여부를 체크하는 Request를 요청하고 종료 한다.
- Server Thread에 요청이 들어오면 UploadThread가 start되는데, Upload Thread는 받은 요청에 대해서 연결을 해제하지 않은 상태로 계속해서 Information 요청과 Send_ChunkMap, Request_Chunks요청을 수행한다.

```

=====
        UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
        UploadThread End
=====
=====
        UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
        UploadThread End
=====
=====
        UploadThread Start
=====
UploadThread 받은요청 : Request_Seeder
=====
        UploadThread End
=====
=====
        All peers in server download completed
=====
~/desktop/bittorrent/Peer1▶ []

```

- 이후, 모든 피어가 다운로드가 끝나면 All peers in server download completed를 띄운 후 모든 쓰레드를 종료시켜 프로세스가 끝나게 한다.

leechers

```
~/desktop/bittorrent/Peer2 ➤ java -Xmx1g Peer config.txt test.pdf
=====
          Peer Start
=====
There is no such file in files folder
File Download Start
=====
          Server Thread Start
=====
=====
          Client Thread Start
=====
Client - 소켓 연결을 시도합니다.
=====
          Peer End
=====
=====
          Client Thread Start
=====
Client - 소켓 연결을 시도합니다.
Client Thread Start
=====
Client - 소켓 연결을 시도합니다.
청크다운로드: 5초간 응답이 없어 127.0.0.1:8004으로 연결에 실패했습니다. 다음 주
```

- 코드를 실행시키면 Peer Thread가 실행되고, 이번에는 시더가 아니므로 폴더에 file이 없어서 file download start가 시작된다. Client Thread들은 다른 서버에 연결 요청을 시도하는데, 이때 연결에 5초간 응답이 없으면 다음 피어에게 연결요청을 한다.

```
=====
          UploadThread Start
=====
UploadThread 받은요청 : Already_Friend
=====
          UploadThread End
=====
Client - 부족한 청크 개수 : 5개
Client - 부족한 청크 개수 : 5개
Client - 부족한 청크 개수 : 252개
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
Client - 부족한 청크 개수 : 97개
Client - 부족한 청크 개수 : 244개
UploadThread 받은요청 : Request_Information
UploadThread 받은요청 : Send_ChunkMap
UploadThread 받은요청 : Request_Chunks
=====
          UploadThread Start
```

- ServerThread가 요청을 받으면 UploadThread가 실행되는데, Client Thread의 요청에 대한 응답을 진행 한다.
- Client Thread는 부족한 청크의 개수를 파악해서 연결된 friend에게 3초마다 부족한 청크를 요청한다.

```
=====
UpLoadThread End
=====
127.0.0.1:8005 와 연결되었음
=====
Client Thread End
=====
=====
UploadThread Start
=====
UpLoadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
UploadThread Start
=====
UpLoadThread 받은요청 : Request_Seeder
=====
UploadThread End
=====
=====
All peers in server download completed
=====
~/desktop/bittorrent/Peer2 ➤ |
```

- 마찬가지로 모든 피어가 다운로드가 끝나면, All peers in server download completed를 띄운 후 모든 쓰레드가 종료되고 프로세스가 종료된다.
-