Design Documentation

CS550 13F PA3

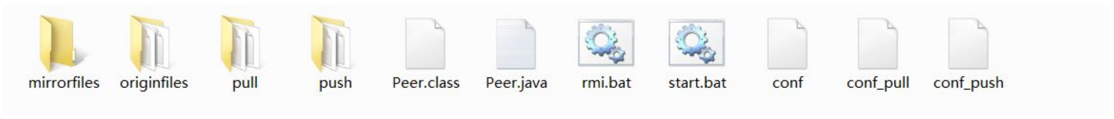
Feng Huang A20281629
Hao Guo A20286403

Introduction

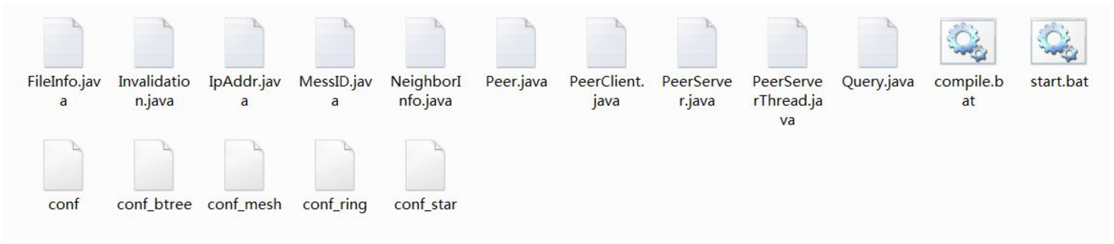
In this project, we refined our PA2, adding file consistency features to it. In our program, a file can only be modified by its original master, which is a unique one to each file. When facing a master modifies its copy, we designed two approaches to invalidate the mirror copies possessed by other peers, which are PUSH and PULL. A Push is initiatively broadcast the invalidate message, however a Pull mode will just periodically ask the owner file current version.

Class Topology

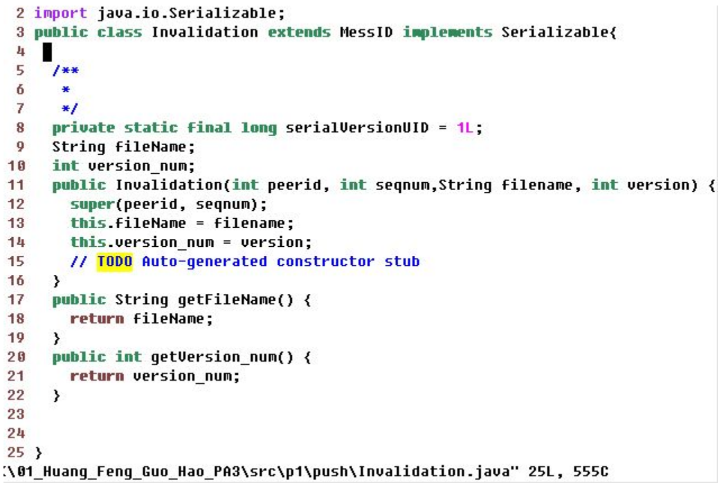
Since we are actually adding features to PA2, the class hierarchy doest change a lot. Most class information are in last design document . What changes is that within one peer, there are two implementations, with a upper class to chose which to call.



Peer.java reads conf and decide which of pull and push to run.
In push:



The only new class here is *Invalidation.java,* it doest similar job as *MessID* does, so, it inherits *MessID*, adding version tracking fields:

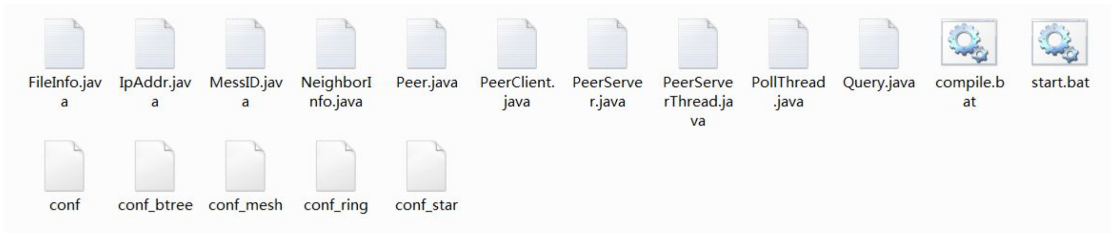


```
import java.io.Serializable;
public class Invalidation extends MessID implements Serializable{

  /**
   *
   */
  private static final long serialVersionUID = 1L;
  String fileName;
  int version_num;
  public Invalidation(int peerid, int seqnum,String filename, int version) {
    super(peerid, seqnum);
    this.fileName = filename;
    this.version_num = version;
    // TODO Auto-generated constructor stub
  }
  public String getFileName() {
    return fileName;
  }
  public int getVersion_num() {
    return version_num;
  }


}
:\01_Huang_Feng_Guo_Hao_PA3\src\p1\push\Invalidation.java" 25L, 555C
```

On the other hand, in pull part, there is no need for Invalidation.java, but it needs a thread class to do polling, which is *PollThread.java*. The basic idea is that every time a file successfully downloaded, a thread is created to poll it to owner in some time intervals.



<u>Design Detail</u>

Push mode:

The push mode needs a message type to be broadcast, in our implementation, we use RMI facility, all message send out is a parameter of remote invoke, and message received is its return value. In the previous project, we had a *MessID* class working as the message parameter. In this push mode, we also need the something as basic, such as counting message number, maintaining visit list. New field is needed to send file version out and in, so we made a class Invalidation, which extends *MessID*, adding String *fileName* and int *version_num* as field. In the root class Peer, we added a remote function *public synchronized void invalidate(Invalidation iv, int ttl).* It works just like query, except that the parameter passed in is new class Invalidation, instead of MessID. We can presume that within a proper ttl, the invalidation message can go through all peers.

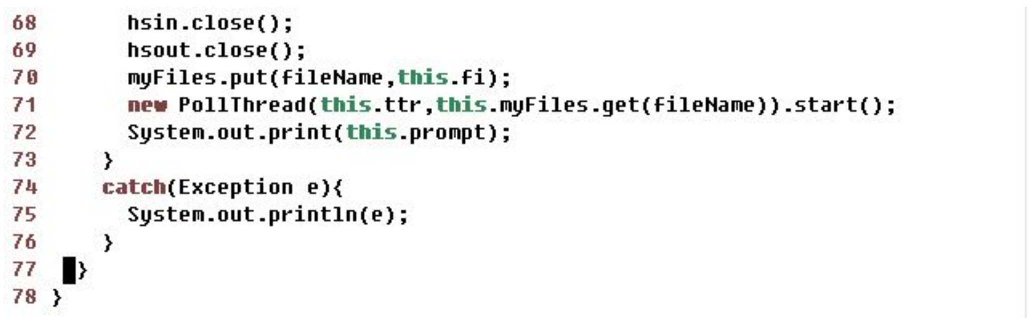
In the *public synchronized void invalidate(Invalidation iv, int ttl),* it first check ttl and decide whether to spread it, then, it check mirrorlist, if the incoming file name matches one of the mirrorfiles, then compare the version number, mostly, the version would be out-dated, so it will mark its copy as *FileInfo.State.INVALID.*

On the other hand, upon a query, a server first check if request file is in origin files, if yes, just reply IpAddr, if no, to check if it is in mirrorfiles, if so, continue checking if it's *FileInfo.State.VALID,* if so send a valid reply, if not sent a invalid reply, the latter will be used to calculate performance statistics.

Pull mode:

The PULL mode is more complex than PUSH, because it maintains more metadata, that is, after a client download from another, it should not only have the file, but also have information about file's origin server and its IP address. In order to do this, we changed *myOriginFiles* from *ArrayList<FileInfo>* to a *Map<String,FileInfo>*. In this way we can easily find a file's info by match a key, instead of iterating list.

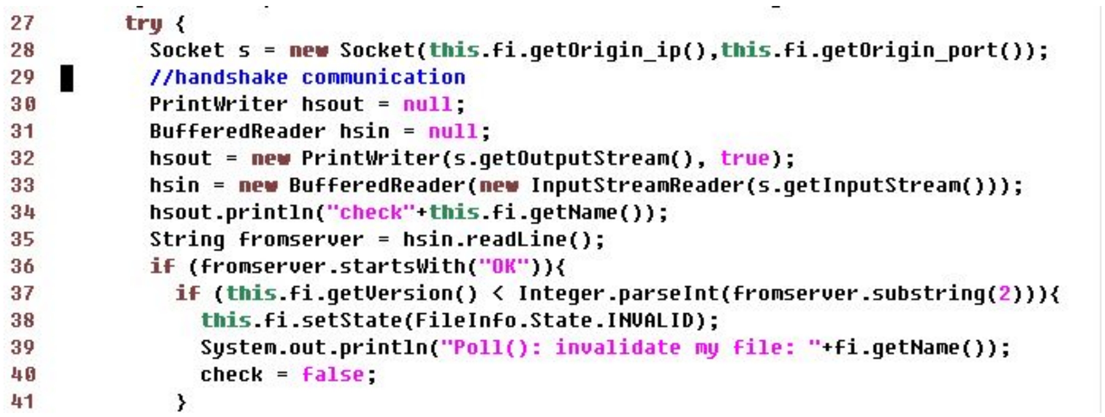
The download command was modified that, in *PeerClient pc = new PeerClient(targetpeer.ia, targetpeer.port,fn,((Peer)p).mirrorPath,((Peer)p).myMirrorFiles,targetpeer.getFi(),((Peer)p).ttr,pr* *ompt);* we put *targetpeer.getFi()* in parameter, and the *PeerClient* later will use if to locate origin server, then fork a thread to tracing its modification. In *PeerClient* class:



```
68        hsin.close();
69        hsout.close();
70        myFiles.put(fileName,this.fi);
71        new PollThread(this.ttr,this.myFiles.get(fileName)).start();
72        System.out.print(this.prompt);
73      }
74      catch(Exception e){
75        System.out.println(e);
76      }
77    }
78 }
```

Line 70 added the origin server info to my newly created mirror file, and line 71 create a thread, given origin server IP address and port, starting polling it.

And it is necessary to explain what the *PollThread* will do: it starts a while loop, within the loop, first sleep for ttr second, then, build a socket connection to origin server(line 28), do simple handshake communication(line 30-36), which is sending "check+filename" and receiving "OK+version_number", finally, it checks with own copy's version number, to invalidate in own number is less than origin number.(line 37-39). Once the file is marked INVALID, the loop breaks and thread exit(line 40).



```
27        try {
28          Socket s = new Socket(this.fi.getOrigin_ip(),this.fi.getOrigin_port());
29          //handshake communication
30          PrintWriter hsout = null;
31          BufferedReader hsin = null;
32          hsout = new PrintWriter(s.getOutputStream(), true);
33          hsin = new BufferedReader(new InputStreamReader(s.getInputStream()));
34          hsout.println("check"+this.fi.getName());
35          String fromserver = hsin.readLine();
36          if (fromserver.startsWith("OK")){
37            if (this.fi.getVersion() < Integer.parseInt(fromserver.substring(2))){
38              this.fi.setState(FileInfo.State.INVALID);
39              System.out.println("Poll(): invalidate my file: "+fi.getName());
40              check = false;
41            }
```

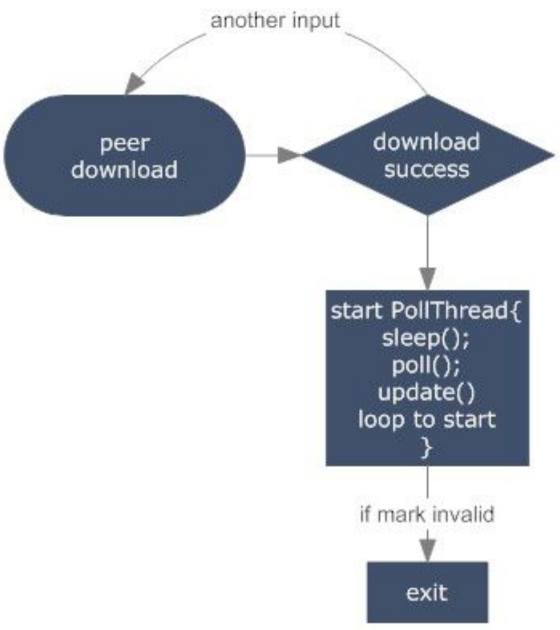
<u>**Implementation Highlight**</u>

<u>Marshaling and Reference Problem</u>

We did encounter a problem that, in order to measure invalidate count, we firstly tried to retrieve *MessID mi* after called remote method, the latter will have *mi* as marshaled parameter and do some change there. However this kind of passing a reference of object as parameter work locally, but not remotely. We got nothing changed in local *mi*. The understanding behind this behavior is that, unlike local object, remote method alway has a copy of you parameter, whatever basic var as int or self-defined object, serialize it and send is what they can do, since they are not sharing memory space, modifying one won't affect another. Finally we solved this by put our information as return value, this works as usual because a copy is what we just needed.

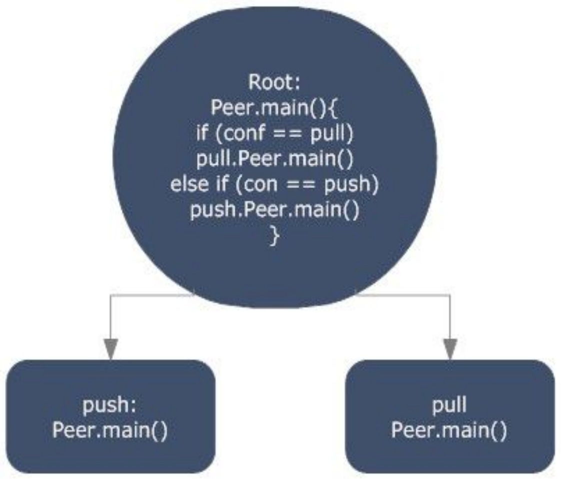
<u>Poll Thread Algorithm</u>

At first, there seems to be two possible solution to do polling. One is to have one thread maintaining all mirror files's versions, the other approach is have one thread each file. We decided to use the former solution, because we would avoid running to many thread which will slow down the system. Sounds reasonable. But the real question is that we never know when a download issued and succeeded, if a thread run and doing nothing for a long time, this is still a waste. On the other hand, we chose the second solution, create a thread for each incoming file. Even if there are a lot more threads, but 99% percent of time they are sleeping, does not need to be scheduled, they are not busy waiting. Moreover, when a file marked invalid, the thread for file exit. So the best case is that there are no polling thread at all.



Combining Two Projects

Our team designed the two approach in parallel, so the question is how to combine them into one as PA instructed. Since both Pull and Push they have difference about data structure, so its very difficult to merge them. We solved this by creating a root class Peer for each peer, and put whole push and pull projects as separate folders, refactor source code, adding package statement. And in the root Peer, read conf file to decide running mode, and finally call the static method to invoke one. Since the are in different packages, having the same code does not interfere. A figure right is showing this:

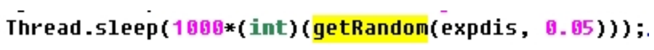


Random and Distribution

The performance analysis asked us to use exponential distribution to simulate modify event occurrence, this is done by having a static method which can calculate a value by giving random double and lambda of exponential expression. The function is like below:

```
public static double getRandom(Random r, double p){
  return -(Math.log(r.nextDouble()) / p);
}
```

And the caller is :



```
Thread.sleep(1000*(int)(getRandom(expdis, 0.05)));
```

which means sleeping as exponential distribution, double param means how many time occur in a time unit (sec)