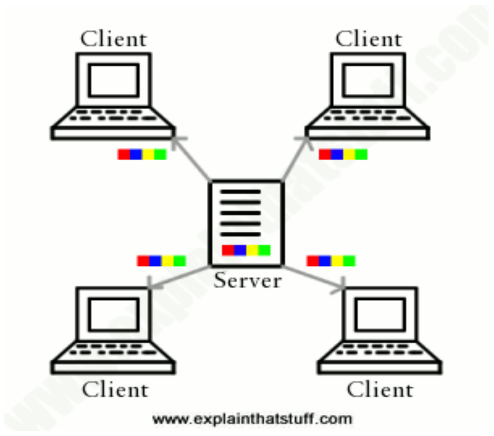
Project Report

CS-835

Shilpa Dhagat

Introduction

As part of this project I have implemented a distributed and highly available peer to peer file transfer cluster, which has single server, serving multiple clients. The server stores only states and has no real file to transfer. File transfer happens only between the clients.



Assumptions Made

- File name is unique. File can be of any type with any extension. Worked fine when transferred doc, image and video (small) files. For file transfer the bytes of the file will be read and transferred so this may not work for huge files, which cannot be contained by java heap memory.
- Multiple instance of the same file across the clients should have the same content.
- Registered file name is same as the actual file name.

Client and server activities from a user’s standpoint

I have one server, serving multiple running clients. Each client has only one point of contact – The Server. The client has its own file system and can do one of the following operations-

- Client can register a globally unique file with the server. Multiple clients can register the same file. Server will store the list of clients against a unique file name. Server stores each client as “host:port”.
- Reach out to server to get the location (client host:port) of a globally unique file.
- De-register the client from the server and gracefully shut down.

Note – Wherever possible ConcurrentHashMap, concurrent collection and data types are used. However, for non-trivial and non-atomic operations I have used explicit locks.

I can have any number of clients registered with the server. Clients have access to files in the file system. Each connected client can register a new file with the server. The server registers this client host:port entry against this file name. Client can request the server for the coordinates of a specific file. Following the coordinates client can request to download the file from the relevant client. If one of the clients fails to be running by the time client request for file download, client will reach out to other running clients, which have the same file. The list of clients for a file download returned by the server is sorted based on the number of successful download by the client. The client can also gracefully deregister from the server.

Server is supposed to maintain only the states in memory. It doesn’t store any file. In other words, server doesn’t need any persistent storage to store the files. Server stores a map of fileName vs the list of clients, which have registered this. P2PServiceImpl implementing P2PServerService is a service implementation, which maintains the server state. All the calls coming via RMI and Socket invoke the same server API. Following is the interface definition-

```
interface P2PServerService
 {
        public String registerFile(Message message);
        public Message queryFile(Message message);
        public String deregisterClient(Message message);
}
```

Server state

Shared state/ Server state and concurrency issues-

A map of files registered with the server. Server maintains a ConcurrentHashMap, which has filename as a key and FileRecord as a value. FileReccord maintains the list of clients for a given filename. Server takes explicit lock only during Register and Deregister process, in which case the server performs a series of non-atomic operations. Query on the other hand involves only get, which is delegated to ConcurrentHashMap and thus an explicit lock is not needed.

Per-client state – Set of registered files by the client and client's host port details. Each client locally maintains the list of registered files with the local file system path to be able to read the bytes when a download request is received from other client.

Data structures-

- **Server will have the following data structures-**
    - Per client state - Map of file name vs FileRecord. Whenever a new register request comes from client to server, server will add a new entry in this map if that file name is not already registered. If the file name is already registered with the server, and a new client register request comes, server will this client to the existing FileRecord in this map. Register operation has possible concurrency issues – 1. If two clients register the same file at the same time, the second request may overwrite the first request and we may be at the risk of losing data.

      This same map is used to look up for the list of clients, which have registered this file with the server. However, it may be obvious to use a readLock here so that we get the correct list of client, it is not essential. The reason of not using a readLock is because we get a list of clients from the server and incase if one of the clients gets deregistered by the time this client reaches out to the deregistered client for file download, the invoker will have other clients to download the file from.

During deregistration process all the entries belonging to a deregistered client should be removed or updated. Multiple clients may request to deregister at the same time, I need to use explicit Write lock to perform this operation.

```
class FileRecord
 {
   private String fileName;
   private String filePath;
   private Set<String> hostPortSet;
}
```

- Per client state - Map of client <host:port> vs Integer numberOfRegisteredFiles. During register and deregister request, this map will be updated. I need to use a WRITE lock to perform the operation.
- Shared state - Integer totalNumberOfRegisteredFiles. During register and deregister request, this map will be updated. I need to use a WRITE lock to perform the operation.

- **Client will have the following data structures-**
  - String clientListeningHostPort – Non static client variable. clientListeningHostPort stores client's external IP and port separated by colon such as "localhost:5561".
  - Map of file name vs FileRecord. This will be a client local file record map and it will act as a cache for client process to check before client reaches out to server to query for a file. This is Threadsafe as client operations are single threaded.
  - Boolean isStopped flag has been used to know if the client has been stopped.
  - String homeDir is used to specify the home directory for each client.

- **Socket based service-**

  - The server process can handle multiple client requests simultaneously. Server listens for new connection requests on a server socket.

  - Whenever a new client connects to the server, it starts a new server sockets also in a new thread to listen for new download requests from other clients.

  - For all the communications between server and client and between client and client the Message class will be used-

    ```
    public class Message

    {
       // host:port,server or host:port,client
       private String from;
       // Different types of messages such as register, query
       private String type;
       // Message format for each type of message will be different.
       private String message;
    }
    ```

  - For file download response, a different class FileDownloadResponse will be used-

    ```
    public class FileDownloadResponse
    {
       private String response;
       private byte[] data;
    }
    ```

  - In this implementation a single file can be registered with the server by multiple clients. In an event if one of the clients goes down the same file can be seamlessly downloaded from other available clients. This feature makes the file server cluster highly available.

- **RMI based Interface-**
  - **Registration**
    - **Server side registration**- Server initializes RMI server for a name, creates a registry on a port and binds this server instance with the registry against the same name.

```
private void startRMI()
{
        String name = "1";
        try
    {
        BankRMIServer bankRmiServer = new BankRMIServer(name);
        Registry registry = LocateRegistry.createRegistry(12345);
        registry.bind(name, bankRmiServer);
    }
catch (Exception e)
   {
        System.err.println(e);
   }

 }
```

    - **Client side initialization**- Client gets access to the registry, passing server host and rmi port and gets an instance to the RMI remote interface, passing the same name against which the instance was registered by the server.

```
private void getRemoteInstance()
        {
            Registry registry;
            try
              {
                 registry = LocateRegistry.getRegistry("localhost", 12345);
                  this.serverService = (P2PServer) registry.lookup("1");
              }
                catch(Exception e)
              {
                    System.err.println(e);
              }
        }
```

- RMI based method invocation takes place between two java processes across the wire. It may also take place across the physical hosts.

- RMI stands for Remote Method Invocation, which implies that a java process can invoke a remote method call on another java process.

- The way it works is the invoker serializes the method invocation and sends the serialized bytes across the network to the other process. The receiver deserializes the method invocation, invokes the method and sends the response back to the invoker.

- I am having the following RMI method invocations between client and server-

  ```
  public String registerFile(Message message);
  public Message queryFile(Message message);
  public String deregisterClient(Message message);
  ```

- And the following between client and client

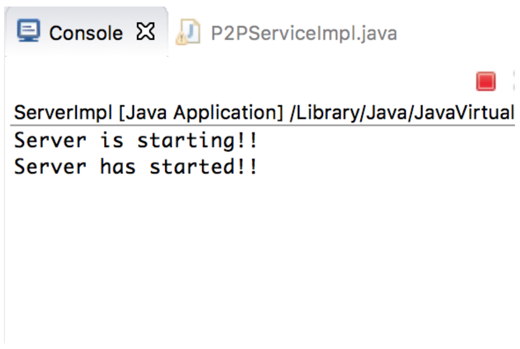
  ```
  public FileDownloadResponse transmitFile(Message message);
  ```

- **Test Case Scenario-**

  - **Scenario1 :**
    Connect 2 different clients with the server socket. Client1 registers a file with the server. Client2 requests the coordinates of the same file from the server and downloads this file from the coordinates given by the server.

    A sample run is described below by using a series of screenshots.
    File transfer from client 1 to client 2.

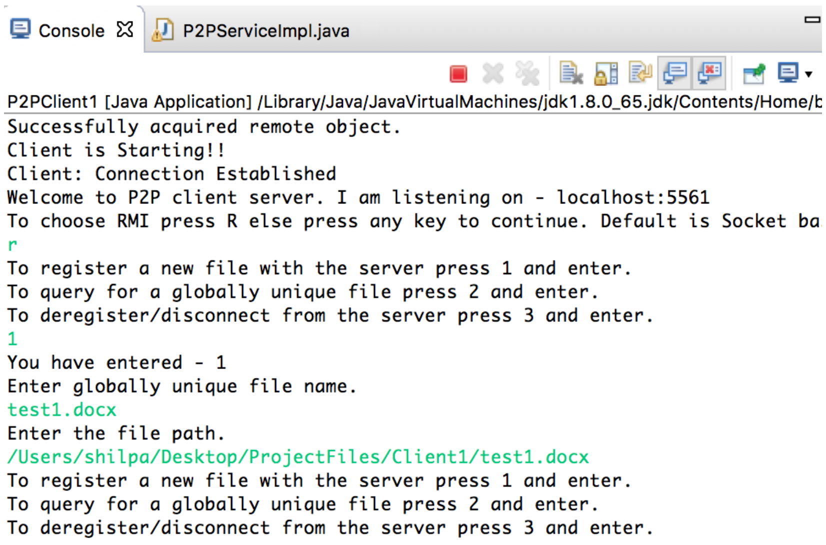
    1. Firstly, ServerImpl is run to initialize the RMI server. Server starts the RMI server and it then initializes itself. The below screenshot shows that the server has started running.



Console ☒ P2PServiceImpl.java

```
ServerImpl [Java Application] /Library/Java/JavaVirtual
Server is starting!!
Server has started!!
```

2. New is I am running the P2PClient1 to launch Client1.
Choosing between the two connections RMI or socket based. For registering a file to the server I am pressing 1. It askes for the file name to be registered. And then it asks for the file path where the file is located in the file system. After this the file is registered on the server.

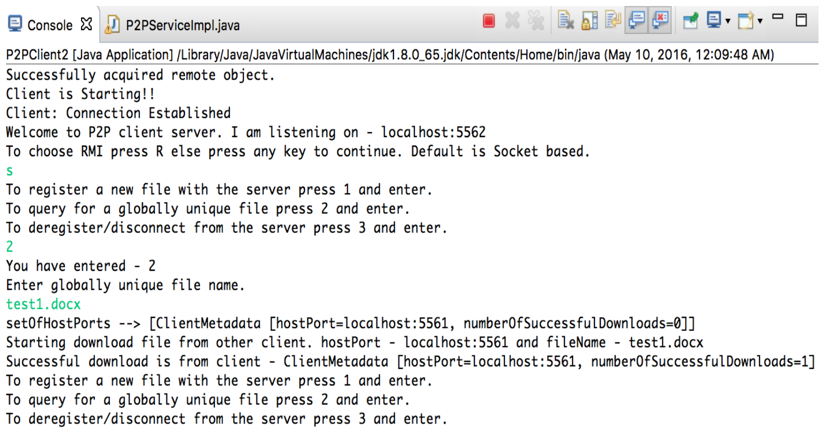


Console ☒ P2PServiceImpl.java

```
P2PClient1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/b
Successfully acquired remote object.
Client is Starting!!
Client: Connection Established
Welcome to P2P client server. I am listening on - localhost:5561
To choose RMI press R else press any key to continue. Default is Socket ba
r
To register a new file with the server press 1 and enter.
To query for a globally unique file press 2 and enter.
To deregister/disconnect from the server press 3 and enter.
1
You have entered - 1
Enter globally unique file name.
test1.docx
Enter the file path.
/Users/shilpa/Desktop/ProjectFiles/Client1/test1.docx
To register a new file with the server press 1 and enter.
To query for a globally unique file press 2 and enter.
To deregister/disconnect from the server press 3 and enter.
```

3. Now after registering the file, I am running P2Pclient2 for launching client2. Again I have to select between RMI or socket based connection. Now for querying and downloading the file from client 1 to client2, I should select option 2. And then I have to provide the filename. After providing the information, it tells the client2 that who all registered the same file on the server with the following details:

Hostport – localhost:5561 and filename – test1.docx
And then it downloads the file to the requested client file system.
If the file has been registered by 2 or more clients, then on querying it will show all the clients who has registered that particular file and downloads from the one which has the greater number of successful download history.



```
P2PClient2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (May 10, 2016, 12:09:48 AM)
Successfully acquired remote object.
Client is Starting!!
Client: Connection Established
Welcome to P2P client server. I am listening on - localhost:5562
To choose RMI press R else press any key to continue. Default is Socket based.
s
To register a new file with the server press 1 and enter.
To query for a globally unique file press 2 and enter.
To deregister/disconnect from the server press 3 and enter.
2
You have entered - 2
Enter globally unique file name.
test1.docx
setOfHostPorts --> [ClientMetadata [hostPort=localhost:5561, numberOfSuccessfulDownloads=0]]
Starting download file from other client. hostPort - localhost:5561 and fileName - test1.docx
Successful download is from client - ClientMetadata [hostPort=localhost:5561, numberOfSuccessfulDownloads=1]
To register a new file with the server press 1 and enter.
To query for a globally unique file press 2 and enter.
To deregister/disconnect from the server press 3 and enter.
```

At any time, the client can de-register itself from the server and gracefully shut down.

- **Scenario2 :**
  Connect 3 different clients with the server socket. Client1 registers a file with the server. Client2 requests a different file with the server. Client3 requests for both the files and is able to download these files from each client.

- **Scenario3 :**
  Connect 3 different clients with the server socket. Client1 registers a file with the server. Client2 registers the same file with the server. Client1 deregisters from the server and shuts down. Client3 requests for the same file from the server and is able to download it from client2. Repeat above test cases again but instead of client1 shut down client2 this time. Client3 should still be able to download the file from Client1. This test case proves that the peer to peer file transfer cluster is highly available.

- **Scenario4 :**
  Deregister a client and make sure that the client deregisters itself from the server successfully, has no dirty data left in the server and is able to terminate the java process.

- ## Ghost Client implementation-

  - The ghost client implementation will mimic the behavior of an actual client. While launching the ghost client, we can pass the number of ghost clients we intend to launch and each client will run on a single host. These clients will have access to the common folder, which has few files and a private folder, to which the downloaded files will be stored. Each client will randomly either register or query for the registered file from the server.

- ## Correctness and Performance of server-

  - To measure the performance and load on server I have used GhostClientManager. I have used 100 clients. All these clients are running on the same JVM and of course same host. Server is running on a different host. Between RMI/Socket based choices for each client, I have generated a Random Boolean and based on the random generated value in the code I have toggled between TCP and RMI.

  - I have measured time to download files between clients. Since all the clients are running on same host, the download was very fast.