Blockchain: Reliability

Nahian-Al Hasan - nhas9102@uni.sydney.edu.au

Part I: White Box Testing -

For white box testing, I have prepared several edge cases and noted down the behaviour of my BlockchainClient and BlockchainServer applications. I believe that by gracefully handling such cases, I can prevent my applications from crashing, and give the user tangible feedback.

BlockchainServer

I. Catching NumberFormatException for args[0]
Input: $ java -cp bin BlockchainServer wrong_port
Expected Output: Server replies that port is not a number.
Actual Output: Error: Port is not a number.

II. Catching NumberFormatException for args[0]
Input: $ java –cp bin BlockchainServer 10000000000000
Expected Output: Server replies that overflow has occurred for args[0].
Actual Output: Error: Port number caused overflow.

III. Catching IllegalArgumentException for args[0]
Input: $ java –cp bin BlockchainServer 1000
Expected Output: Server would send error that 1000 is less than 1024
Actual Output:
Error: Illegal port number.
Please try values between 1024 and 65535.

BlockchainClient

I. Catching FileNotFoundException for config file
Input: $ java –cp bin BlockchainClient wrong_config
Expected Output: Client would reply that file wasn't found *but would not terminate*.
Actual Output: Error: File wrong_config was not found.

II. Catching SocketTimeoutException when reading from InputReader
Code:

```
outWriter.println(message);
clientSocket.setSoTimeout(2000);
Thread.sleep(5);
while(inputReader.ready() && (temp = inputReader.readLine()) != null)
{
   output += temp + "\n";
}
```

Expected: Sends "Server not available" when no input has been received by the client's InputReader for over 2 seconds. This was achieved by not replying through the server upon

reception of message.
Actual: Prints "Server not available" as expected.

III. Catching Exception by trying to connect to non-existing server with valid IP-address and port
Expected: Client would throw exception upon trying to create new a new client socket.
Actual: Throws error only when host is localhost, but crashes in the case of foreign IP address.

Fix:

```
try {
   if(!InetAddress.getByName(this.serverName).isReachable(2000)) {
      throw new Exception();
   }
} catch (Exception e) {
   this.reply += "Server is not available\n\n";
   return;
}
```

The isReachable() function terminates if the server does not reply within 2 seconds and appends the error message to the reply. The client does not break after implementing this code.

Part II: Grey Box Testing -

In this part, I communicated a series of malicious commands and requests to the client and server respectively. My findings were quite fruitful and I was able to find plenty of bugs and fix them. However, most of my standard client functions were infallible as I implemented ***strict*** Regular Expressions (RegExp) for ***all*** of my commands to the client.

I. Trying to broadcast tx|nhas9102|| to the servers
Expected: Server would reject the empty message.
Actual: Server rejected empty message and replied "Rejected".

I used my own function for validating **tx** messages to the server, which strictly covered all corner cases for malicious tx commands.

II. Unicasting or Multicasting with pb
Expected: Undefined behaviour when using server indices greater than the size of the ServerInfoList.
Actual: IndexOutOfBoundException in the ArrayList.

This had occurred due to an error in my code, when I decided to multicast or unicast messages to indices greater than the size of the ArayList. This error made me realise I needed to print nothing and continue in the case the index provided did not exist.

III. Writing mixed up pb transactions, e.g. pb|abc|1 to the client
Expected: Prints a blank line.
Actual: Prints a blank line on the terminal.

This occurs due to the fact that my RegExp for pb, `"^pb(?:\\|\\d+)+"` disregards the entire message if it provides an unspecified index for the serverInfoList.

IV. Writing ad|x.y.z|6333 or up|localhost|1000000 to the client. These messages check the validity of my *isHostVerified(String)* and *isPortVerified(int)* functions.
Expected: Client would return "Failed".
Actual: Client returns "Failed".

While checking the validity of ports are certainly straightforward, checking that of IP addresses are not so easy. Similar to my previous solutions, I used RegExps to validate IPv4 and IPv6 addresses before creating an entry in my serverInfoList. Owing to the RegExps being very large, I have not included the code for those.
These methods are as same as the ones implemented when checking the config file. Therefore, my config file parsing is also quite reliable.

Part III: Black Box Testing

I. Broadcasting or Multicasting after having deleted a server.

> ls
> Server0: fe80::c9f1:420a:4a0b:29ba 6333
> Server1: 124.190.141.170 4444
> Server2: localhost 8333
>
> rm|1
> Succeeded
>
> pb

My friend, Shenin Faizah (sfai4579) ran these series of commands from my client. To my surprise, I received a NullPointerException because I'd forgotten to check for nulls.
Solution: A simple check for nulls before unicasting/multicasting/broadcasting the message solved the problem.

Additionally, she created her own testConfig file to stretch the capabilities of my server. However, his is the only Exception I could obtain from Black Box testing. Otherwise, the rest of the functionalities seemed to be working fine.