THE UNIVERSITY OF SYDNEY

Due: Wednesday Week 8 by 11:59 AM

COMP2121 Assignment 2: CS Blockchain

The goal of this project is to implement a Multithreaded Client-Server (CS) Blockchain application in Java which can spread messages to multiple servers and accept connections from multiple clients at the same time.

1 Submission Details

The assignment comprises three tasks and each task can be submitted separately. The final version of your assignment should be submitted electronically via PASTA by 11:59AM (before noon) on the Wednesday of Week 8 (Hard Deadline). The project is an individual project, and each student has to submit his/her own version.

1.1 Program structure

For task 1, six java files must be submitted. It is recommended to submit by 11:59 AM on the Wednesday of Week 6 (Soft Deadline).

- A **`Blockchain.java`** file.
- A **`Block.java`** file.
- A **`Transaction.java`** file.
- A **`BlockchainServer.java`** file.
- A **`BlockchainServerRunnable.java`** file.
- A **`PeriodicCommitRunnable.java`** file.

For task 2, ten java files must be submitted. It is recommended to submit by 11:59 AM on the Sunday of Week 6 (Soft Deadline).

- A **`Blockchain.java`** file.
- A **`Block.java`** file.
- A **`Transaction.java`** file.
- A **`BlockchainServer.java`** file.
- A **`BlockchainServerRunnable.java`** file.

- A **BlockchainClient.java** file.
- A **BlockchainClientRunnable.java** file.
- A **ServerInfo.java** file.
- A **ServerInfoList.java** file.
- A **PeriodicCommitRunnable.java** file.

For task 3, ten java files and **ReliabilityReport.pdf** file must be submitted. It is recommended to submit by 11:59 AM on the Sunday of Week 7 (Soft Deadline).

- A **Blockchain.java** file.
- A **Block.java** file.
- A **Transaction.java** file.
- A **BlockchainServer.java** file.
- A **BlockchainServerRunnable.java** file.
- A **BlockchainClient.java** file.
- A **BlockchainClientRunnable.java** file.
- A **ServerInfo.java** file.
- A **ServerInfoList.java** file.
- A **PeriodicCommitRunnable.java** file.
- A **ReliabilityReport.pdf** file.

All classes will be stored in the same default package (no package header in java files), and all files should be located in the same **src** folder with no subfolders. All present **.java** files should be correct and do not forget to remove any dummy files that do not count as source files (E.g. junit test cases, class files). Please zip the **src** folder and submit the resulting archive **src.zip** by the deadline given above. The program should compile with Java 8. No optional packages that are not part of the default Java 8 JDK can be used.

1.2 Submission system

PASTA will be used to assess that your program correctly implements the Multithread Client-Server Blockchain protocol. The archive **src.zip** should be submitted at http://soit-app-pro-10.ucc.usyd.edu.au:8080/PASTA/login/. To access this website, you will have to be connected to the network of the University of Sydney (physically on campus or through VPN).

PASTA stands for "Programming Assessment Submission and Testing Application" and is a web-based application that automates the compilation, execution and testing of your program. When you submit your **`src.zip`** archive in PASTA, the system enqueues your submission in the shared queue of all assignments to be tested. It may take more time close to the deadline as many students will try to submit at the same time.

1.3 Academic Honesty / Plagiarism

By uploading your submission to PASTA you implicitly agree to abide by the University policies regarding academic honesty, and in particular that all the work is original and not plagiarised from the work of others. If you believe that part of your submission is not your work you must bring this to the attention of your tutor or lecturer immediately. See the policy slides released in Week 1 for further details.

In assessing a piece of submitted work, the School of IT may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program. A copy of the assignment may be maintained by the service or the School of IT for the purpose of future plagiarism checking.

2 Marking

This assignment is worth 10% of your final grade for this unit of study.

The first category of tasks, called *Blockchain: Multithreaded Server*, assesses the behaviour of the server with respect to requirement. If you could pass all tests, two marks are given.

- Maximum 1 mark for server accepting multiple client connections at the same time.
- Maximum 1 mark for server processing multiple client requests at the same time with proper synchronisation.

The second category, called *Blockchain: Multithreaded Client*, assesses the behaviour of the client concerning requirement. If you could pass all tests, four marks are given.

- Maximum 1 mark for client being able to read config file properly.
- Maximum 1 mark for client being able to add, remove, update, list servers' info correctly.
- Maximum 1 mark for client unicasting, multicasting and broadcasting messages to servers correctly.
- Maximum 1 mark for client printing replies accurately.

The third category, called *Blockchain: Reliability*, assesses the resilience level of client / server to exceptions and crashes. Maximum four marks are given if all corner cases are handled and documented properly.

- Maximum 1 mark for client resilient to exceptions and crashes.
- Maximum 1 mark for server resilient to exceptions and crashes.
- Maximum 1 mark for implementing test cases and dummy client / server properly.
- Maximum 1 mark for documenting exceptions precisely.

Please make sure previous tasks are implemented correctly, before moving to next task. You may face cascading failures due to incomplete implementation of previous tasks.

2.1 Feedback

PASTA provides feedback about each individual submission, one at a time. It will output a list of tests and outcomes indicating whether your program passed each visible test successfully or failed. The feedback provided is indicative, intentionally high-level and will not precisely identify the nature of any bug of your program. Please write your personal test cases and thoroughly test all your code before submission.

3 Functionalities of a Client-Server Blockchain

The goal of this project is to implement a Reliable Multithreaded Client-Server (CS) Blockchain application. The previous blockchain completed in assignment 1 can only manage one client and one server running at the same time, At this stage, we are going to extend assignment 1 with multithreading support. Many clients can talk to a single server concurrently, and a client can talk to multiple servers concurrently as well. The following tasks indicate the features that should be achieved, one solution for each task is expected to be submitted.

Task 0 Blockchain: Proof-of-Work Simulation

This task is designed to let you familarise yourself with multithreading. No coding is required, and no mark is assigned to this task.

Previously, our method to build a block is pretty simple, where every three transactions generate a new block. In reality, blockchain employs a different way to create a block. Each miner (server) who mines (create) the block, are competing to commit the correct block under some specific rules. An example of such a rule is to build blocks with a hash value starting from character `"A"`. However, the transactions cannot be altered to make hash value beginning with `"A"`. Therefore, we need to introduce an extra random variable called *nonce* when calculating the hash. By changing the nonce value, the hash of the block gets changed as well. The lucky miner who gets the nonce correct is the owner of the block, who can claim the mining reward.

To realise this periodic mining behaviour, we need a separate execution line who mines the block every few seconds. This is the place where multithreading could help us.

A quick recap, *multithreading* enables a single program (process) with multiple lines of execution. This execution unit is named as a thread (In contrast to process, which is the resource unit). In Java, we can quickly implement thread concept by creating a custom class which extends **`Thread`** class or implements the **`Runnable`** interface. To pass the raw inputs into the thread, you could use the constructor. To fetch the processed output back, you could create a getter method. The thread can be started by invoking start() method, and the main thread can wait for the child thread to finish by calling the join() method. Those information was provided in lectures and labs.

Below is a concrete implementation which achieves the functionality described above. Please do not modify this class.

```
1  import java.security.SecureRandom;
2
3  public class PeriodicCommitRunnable implements Runnable{
4
5      private volatile boolean isRunning;
6      private int nonce;
7      private Blockchain blockchain;
8      private SecureRandom randomGenerator;
9
10     public PeriodicCommitRunnable(Blockchain blockchain) {
11         isRunning = true;
12         this.blockchain = blockchain;
13         randomGenerator = new SecureRandom();
14         nonce = randomGenerator.nextInt();
15     }
16
17     public void setRunning(boolean isRunning) { this.isRunning = isRunning; }
18
19     public boolean getRunning() { return isRunning; }
20
21     @Override
22     public void run() {
23         while (isRunning) {
24             blockchain.commit(nonce);
25             nonce = randomGenerator.nextInt();
26             try {
27                 Thread.sleep(400);
28             } catch (InterruptedException e) {
29                 System.err.println("PeriodicCommitterInterrupted.");
30             }
31         }
32     }
33 }
```

Task 1 Blockchain: Multithreaded Server

In task 1, you are required to extend the Blockchain's server, which can process client requests like **tx**<unikey>|<content>, pb and cc| like before. Previously, the blockchain server can only handle one connection at a time. This is not a big deal if there is only one user. However, at this stage, we assume there will be multiple clients accessing your server at the same time. Hence, a multithreaded server is required.

In this case, each client socket being created by **ServerSocket** should be handled by one of the worker thread **BlockchainServerRunnable**. **BlockchainServerRunnable** class's constructor accepts two parameters. The first is the client socket, and the second is the blockchain. Since all worker threads modify the same blockchain, you are also required to improve the blockchain class provided to you with proper synchronisation support as taught in week 2's tutorial.

To test it, try to connect to your server with multiple clients developed in the previous assignment at the same time, you should make sure all your clients can see the same result, and no race condition occurs.

Please use the skeleton code below to implement your **BlockchainServer** class.

```
1  public class BlockchainServer {
2
3      public static void main(String[] args) {
4
5          if (args.length != 1) {
6              return;
7          }
8
9          int portNumber = Integer.parseInt(args[0]);
10         Blockchain blockchain = new Blockchain();
11
12         PeriodicCommitRunnable pcr = new PeriodicCommitRunnable(blockchain);
13         Thread pct = new Thread(pcr);
14         pct.start();
15
16         // implement your code here.
17
18         pcr.setRunning(false);
19         pct.join();
20     }
21
22     // implement any helper method here if you need any
23 }
```

please use the skeleton code below to implement your **BlockchainServerRunnable** class.

```
1  public class BlockchainServerRunnable implements Runnable{
2
3      public BlockchainServerRunnable(Socket clientSocket, Blockchain blockchain) {
4          // implement your code here
5      }
```

```
6
7       public void run() {
8           // implement your code here
9       }
10
11      // implement any helper method here if you need any
12  }
```

Task 2 Blockchain: Client

Previously, our client just worked as a simple repeater sitting between the user and server. Whatever the user types in, the client directly forward it to the server. Whatever the server replies, it just gets popped back to the client. At this stage, we redesign our client to make it support more routing mechanisms. So it can provide multicast and broadcast functionalities.

Here is a list of commands your client must handle properly.

- ls command. "ls"
- ad command. "ad|<host name>|<port number>"
- rm command. "rm|<server index>"
- up command. "up|<server index>|<host name>|<port number>"
- tx command. "tx|<sender>|<content>"
- pb command. "pb"
- pb command. "pb|<server index>"
- pb command. "pb|<server index>|<server index>|<server index>..."
- sd command. "sd"

To maintain a list of known server, we create two separate classes to store server information. The first is the **`ServerInfo`** class which records individual server information like host name and port number. Another is the **`ServerInfoList`** class, which stores a list of **`ServerInfo`** objects. Some methods of those two classes are already implemented for you. Your first job here is to implement the **`InitialiseFromFile()`** method. In this method, your client should be able to read a config file where each line (entry) follows the <key>=<value> form. At this stage, only three types of entries are allowed. The first type has key **`servers.num`**, and the value should be an integer. For example, **`servers.num=2`**. The second type has key **`server<server index>.host`**, and the value should either be a host name like **`localhost`** or an IP address like **`127.0.0.1`**, This value field cannot be an empty string. E.g. **`server2.host=localhost`**. The third type has key **`server<server index>.port`**, and the value should be an integer between 1024 and 65535 inclusive. E.g. **`server1.port=8333`**. If **`servers.num`** entry is missing in the config file, server info list will be initialised to empty. **`server<server index>.host`** and **`server<server index>.port`** must come in pairs. If does not come in pairs or does not follow the description above, you should place a null

value in the server info list. Any entries with server index above or equal to **`servers.num`** should be ignored. If there are multiple entries with the same key, the latest value is used.

In the following example. The ServerInfoList will be initialised to size three, with elements null, null, serverinfo(localhost, 8333).

```
servers.num=2 # same key, first entry will be ingored.
servers.num=3

server0.host=localhost # port number is missing, null value will be added to the list.

server1.host=localhost
server1.port=123456 # port number is greater than 65535, null value will be added to the list.

server2.host=localhost

server2.port=8333

server3.host=localhost # server index equals to servers.num, this entry will be ignored.
server3.port=8334
```

please use the skeleton code below to implement your **`ServerInfo`** class.

```
public class ServerInfo {

        private String host;
        private int port;

        public ServerInfo(String host, int port) {
            this.host = host;
            this.port = port;
        }
        public String getHost() { return host; }
        public int getPort() { return port; }
        public void setHost(String host) { this.host = host; }
        public void setPort(int port) { this.port = port; }

        // implement any helper method here if you need any
    }
```

please use the skeleton code below to implement your **`ServerInfoList`** class.

```
public class ServerInfoList {

        ArrayList<ServerInfo> serverInfos;

        public ServerInfoList() {
            serverInfos = new ArrayList<>();
        }

        public void initialiseFromFile(String filename) {
            // implement your code here
```

```
11          }
12
13          public ArrayList<ServerInfo> getServerInfos() { return serverInfos; }
14          public void setServerInfos(ArrayList<ServerInfo> serverInfos) {
15            this.serverInfos = serverInfos;
16          }
17
18          public boolean addServerInfo(ServerInfo newServerInfo) {
19              // implement your code here
20          }
21
22          public boolean updateServerInfo(int index, ServerInfo newServer) {
23              // implement your code here
24          }
25
26          public boolean removeServerInfo(int index) {
27              // implement your code here
28          }
29
30          public boolean clearServerInfo() {
31            // implement your code here
32          }
33
34          public String toString() {
35              String s = "";
36              for (int i = 0; i < serverInfos.size(); i++) {
37                  if (serverInfos.get(i) != null) {
38                      s += "Server" + i + ": " + serverInfos.get(i).getHost()
39                        + " " + serverInfos.get(i).getPort() + "\n";
40                  }
41              }
42              return s;
43          }
44
45          // implement any helper method here if you need any
46      }
```

Before moving to the next stage, we suggest everyone implement three helper methods first. **`unicast(), broadcast() and multicast()`**. In computer networking, these are the three most common ways to route messages. Typically, those three casting methods are implemented in the network layer supported by the IP protocol. Here we are trying to do a simulation at the application layer, where we define unicast as a one-to-one communication with one thread, broadcast as a one-to-all communication with multiple threads, and multicast is a one-to-many communication with multiple threads. The **`BlockchainClientRunnable`** class will send the message to the server, and store the reply to variable **`reply`**. The socket connection should close immediately after the response comes back by sending **`cc`** to the server. At this stage, Socket connects and the following reads and writes operation should have a timeout of 2 seconds. Once all threads terminate, all replies should be printed on the screen. If the connected server does not reply or any exception has been raised, it should print **`"Server is not available\n\n"`** to the screen. All replies should prepend a header

like **"Server<server index>: <host> <port>\n"**.

Alright, now you can start implementing all commands.

- ls (List) command, print the server info list, using **toString()** method provided. **(toString() + "\n")**
- ad (Add) command, append the new **ServerInfo** to the **ServerInfoList** if the **ServerInfo** is valid.
- rm (Remove) command, remove the **ServerInfo** indexed by the server index from the list (set to null) if the server index is a valid index.
- up (Update) command, update the **ServerInfo** indexed by server index to be the new **ServerInfo** provided if the new **ServerInfo** is valid and the server index is valid as well.
- cl (Clear) command, remove all null values from the **ServerInfoList** if null value exists.
- tx (Transact) command, broadcast the transaction message to all servers and print the reply one by one in the terminal.
- pb (PrintBlockchain) command, broadcast **pb** request to all servers, if no server index specified. Unicast to that specific server, if there is one server index given. Multicast to those specified server, if there is more than one server index given.
- sd (ShutDown) command, shut down your client.

For ad, rm, up and cl command, if the operation succeeded, please print **"Succeeded\n\n"** to the terminal. Otherwise, please print **"Failed\n\n"** to the terminal. For pb command, if the index specified is invalid, or if the value at that index is null, simply ignore that index and print nothing. For any other undefined commands, please print **"Unknown Command\n\n"** to the terminal.

To test it, firstly, test it with two servers running on the same machine, then try to migrate you server to different machines in the same local network (same LAN or same WLAN), finally move to the WAN network.

Make sure you can reproduce the following transcript locally. Assume the config file is the example above, and there are two servers running at localhost port 8333, 8334.

```
ls
Server2: localhost 8333

ad|localhost|8334
Succeeded

ls
Server2: localhost 8333
Server3: localhost 8334

```

```
rm|3
Succeeded

ls
Server2: localhost 8333

rm|-1
Failed

ls
Server2: localhost 8333

up|1|localhost|8334
Succeeded

up|3|localhost|8335
Succeeded

ls
Server1: localhost 8334
Server2: localhost 8333
Server3: localhost 8335

cl
Succeeded

ls
Server0: localhost 8334
Server1: localhost 8333
Server2: localhost 8335

clear
Unknown Command

tx|test0001|1
Server0: localhost 8334
Accepted

Server1: localhost 8333
Accepted

Server2: localhost 8335
Server is not available

pb
Server0: localhost 8334
Pool:
--------------------------------------------------------------------------------
|test0001|                                                                    1|
--------------------------------------------------------------------------------
```

```
61
62  Server1: localhost 8333
63  Pool:
64  ------------------------------------------------------------------------------
65  |test0001|                                                                  1|
66  ------------------------------------------------------------------------------
67
68  Server2: localhost 8335
69  Server is not available
70
71  pb|0|1
72  Server0: localhost 8334
73  Pool:
74  ------------------------------------------------------------------------------
75  |test0001|                                                                  1|
76  ------------------------------------------------------------------------------
77
78  Server1: localhost 8333
79  Pool:
80  ------------------------------------------------------------------------------
81  |test0001|                                                                  1|
82  ------------------------------------------------------------------------------
83
84  pb|0
85  Server0: localhost 8334
86  Pool:
87  ------------------------------------------------------------------------------
88  |test0001|                                                                  1|
89  ------------------------------------------------------------------------------
90
91  sd
```

Please use the skeleton code below to implement your **`BlockchainClient`** class.

```
1   public class BlockchainClient {
2
3       public static void main(String[] args) {
4
5           if (args.length != 1) {
6               return;
7           }
8           String configFileName = args[0];
9
10          ServerInfoList sil = new ServerInfoList();
11          sil.initialiseFromFile(configFileName);
12
13          Scanner sc = new Scanner(System.in);
14
15          while (true) {
16              String message = sc.nextLine();
17
```

```
                // implement your code here
            }
        }

        public static void unicast (int serverIndex, ServerInfo serverInfo, String message) {
          // implement your code here
        }

        public static void broadcast (ServerInfoList serverInfoList, String message) {
          // implement your code here
        }

        public static void multicast (ServerInfoList serverInfoList,
          ArrayList<Integer> serverIndices, String message) {
          // implement your code here
        }

        // implement any helper method here if you need any
    }
```

Please use the skeleton code below to implement your **`BlockchainClientRunnable`** class.

```

    public class BlockchainClientRunnable implements Runnable {

        private String reply;

        public BlockchainClientThread(int serverIndex, String serverName,
          int portNumber, String message) {
            // header string
            this.reply = "Server" + serverIndex + ": " + serverName
              + " " + portNumber + "\n";
        }

        public void run() {
            // implement your code here
        }

        public String getReply() {
            return reply;
        }

        // implement any helper method here if you need any

    }
```

Task 3 Blockchain: Reliability

For this task, we are going to test the reliability of your blockchain by acting as an attacker. We would like our blockchain to be as robust as possible, so to do this we should try to identify all possible vulnerabilities where our server might crash or hang forever.

Evaluation Procedure:

To find vulnerabilities, we are going to test our code as much as possible to look specifically for corner cases. The ultimate goal here is to let our client / server resilient to all kinds of exceptions (which lead to crushes). Exceptions should be handled properly by your code with meaningful error message, rather than throwing it to the public or simply print the stack trace. We propose a three phase testing procedure as the guidance.

Step 1. (White Box Testing) Write general unit tests that should generate exceptions in your code. You should document which exception the test case is targeting on, what is the input, what is the expected output, and what is the actual output, you should demonstrate it with code and comments.

Step 2. (Grey Box Testing) For this step, we should write dummy clients / servers that runs the blockchain. However, we are writing malicious clients and servers, their intention is to misbehave and cause havoc. The goal of this test is to illustrate how your blockchain copes with malicious clients / servers. Note: you are encouraged to test your malicious servers and clients with other people's clients and servers. Document how your clients / servers break other people's client / server or how your client / server broke from other people's. Please note, if your client / server manages to break someone else's code, DO NOT FIX THEIR CODE FOR THEM, they should be able to fix is based on the input you gave them. You should document which exception your dummy client / server is trying to produce, and how do you achieve it, please demonstrate it with code and comments. You should also document your testing procedure, E.g. when, with whom, under which testing environment, do you successfully breaks other people's client / server, do your peer successfully breaks your client / server, etc.

Step 3. (Black Box Testing) Now, it's time to turn to actual people. Since people break things quite easily, your task is to go and find a friend that is outside of this course and has little to no knowledge of networking. Let your friend write a config file and use your client and server (with no help from you). You should document your testing procedure, E.g. when, with whom, under which testing environment, do you find any new exceptions, etc.

For marking, we will mark on how many exceptions have you found, how many exceptions have been covered by your test cases, how sophisticated your attack is (0-1 mark). The quality of the documentation (0-1 mark).

Tips for good marks:

- Set up IDEs to help you easily find errors and debug.
- Try not to throw exceptions across multiple methods, catch and handle them as soon as possible.
- Write the most specific exception. For example: Don't just catch IOException, try to catch ConnectException which is a subclass of IOException.