/\* BlockInputG.java

Name : ROHAN DHOYDA

Version 3.0 2020-02-24, combines BlockingPriorityQueue.

To compile and run (may have to update the file name):

javac -cp "gson-2.8.2.jar" BlockInputG.java

java -cp ".;gson-2.8.2.jar" BlockInputG

RunBlockInput.bat:

java -cp ".;gson-2.8.2.jar" m %1

Example for process two:

> RunBlockInput 2

Author: Clark Elliott, with ample help from the below web sources.

You are free to use this code in your assignment, but you MUST add

your own comments. Leave in the web source references.

This utility program shows one method of reading data into a linked list of unverified blocks from an input data file.

The specific data file / Process ID is determined by argment passed to Java at runtime.

The list is shuffled. Blocks are also written into a priority queue with TimeStamp priority which

demonstrates how the priority queue works.

The shuffled list is marshaled (written) to disk in JSON format.

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import java.io.\*;

/\* CDE: The encryption needed for signing the hash: \*/

**import** com.google.gson.Gson;

**import** com.google.gson.GsonBuilder;

**import** java.io.\*;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.security.\*;

/\* CDE Some other uitilities: \*/

**import** java.util.\*;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.PriorityBlockingQueue;

**import** **static** java.nio.charset.StandardCharsets.UTF\_8;

//-------------------------------------------------------------------------------

/\*\*

\* This is how a block structure is created which contains block id, timestamp, pre hash code.

\*/

**class** BlockRecord{

/\* Here I have defined examples of block field\*/

**public** String BlID;

**public** String TSP;

**public** String VPID;

**public** String PrevHash;

**public** String uuid;

String Fname;

String Lname;

String SSNum;

String DOB;

String RandomSeed;

**public** String WinningHash;

String Diag;

String Treat;

String Rx;

**public** **int** blockNumber;

**public** **int** processID;

**public** String hash;

/\*\*

\* Below is the list of getters and setters of the block such as fetching block id and setting it, since the parameter

\* are private we can access it outside other class by using getters and setters.

\*/

**public** String getBlockID() {**return** BlID;}

**public** **void** setBlockID(String BID){**this**.BlID = BID;}

**public** String getTimeStamp() {**return** TSP;}

**public** **void** setTimeStamp(String TS){**this**.TSP = TS;}

**public** String getVerificationProcessID() {**return** VPID;}

**public** **void** setVerificationProcessID(String VID){**this**.VPID = VID;}

**public** String getPreviousHash() {**return** **this**.PrevHash;}

**public** **void** setPreviousHash (String PH){**this**.PrevHash = PH;}

**public** String getUUID() {**return** uuid;}

**public** **void** setUUID (String ud){**this**.uuid = ud;}

**public** String getLname() {**return** Lname;}

**public** **void** setLname (String LN){**this**.Lname = LN;}

**public** String getFname() {**return** Fname;}

**public** **void** setFname (String FN){**this**.Fname = FN;}

**public** String getSSNum() {**return** SSNum;}

**public** **void** setSSNum (String SS){**this**.SSNum = SS;}

**public** String getDOB() {**return** DOB;}

**public** **void** setDOB (String RS){**this**.DOB = RS;}

**public** String getDiag() {**return** Diag;}

**public** **void** setDiag (String D){**this**.Diag = D;}

**public** String getTreat() {**return** Treat;}

**public** **void** setTreat (String Tr){**this**.Treat = Tr;}

**public** String getRx() {**return** Rx;}

**public** **void** setRx (String Rx){**this**.Rx = Rx;}

**public** String getRandomSeed() {**return** RandomSeed;}

**public** **void** setRandomSeed (String RS){**this**.RandomSeed = RS;}

**public** String getWinningHash() {**return** WinningHash;}

**public** **void** setWinningHash (String WH){**this**.WinningHash = WH;}

}

/\*\*

\* This class is blockchain class which consists of process id and blocking queue which takes block record.

\*/

**public** **class** Blockchain {

**public** **static** **int** PRCID; //we have defined integer variable for process ID

**public** **static** List<BlockRecord> blockchainList = **new** ArrayList<>(); //blockchainList is an instance of the ArrayList class that can hold objects of type BlockRecord

**public** **final** BlockingQueue<BlockRecord> queue = **new** PriorityBlockingQueue<>(100,BlockTSComparator);

/\*\*

\* Uses the comparator to compare two block records using timestamp. If both are equal, return 0 or else return other status codes

\*/

**public** **static** Comparator<BlockRecord> BlockTSComparator = **new** Comparator<BlockRecord>() {

@Override

**public** **int** compare(BlockRecord b1, BlockRecord b2) {

//comparing two block with their timestamps

//block1 timestamp

String s1 = b1.getTimeStamp();

//block 2 timestamp

String s2 = b2.getTimeStamp();

//if two timestamps are equal, return 0

**if** (s1 == s2) {

**return** 0;

}

//if one timestamp is null, return -1.

**if** (s1 == **null**) {

**return** -1;

}

**if** (s2 == **null**) {

**return** 1;

}

//this return an int value after comparing

**return** s1.compareTo(s2);

}

};

/\*\*

\* Setting all the values based on their final index

\*/

**private** **static** **final** **int** iFNAME = 0; //We have now set all this variables with an index value

**private** **static** **final** **int** iLNAME = 1; //We have set the index LastName Variable to 1

**private** **static** **final** **int** iDOB = 2; //We have set the index Date of Birth Variable to 2

**private** **static** **final** **int** iSSNUM = 3;//We have set the index Social Security Number Variable to 3

**private** **static** **final** **int** iDIAG = 4; //We have set the index DIAG Variable to 4

**private** **static** **final** **int** iTREAT = 5; //We have set the index TREAT Variable to 5

**private** **static** **final** **int** iRX = 6; //We have set the index RX Variable to 6

**private** **static** String FILENAME; // We have defined string variable for FILENAME

**public** **static** **void** main(String[] argv) {

/\*\*

\* Taking first argument as process id.

\*/

Blockchain.PRCID = Integer.parseInt(argv[0]);

System.out.println("Hello from Process " + Blockchain.PRCID); //Whenever the code will run it will firstly display this message

Ports.setPorts();

//if process id is 0, then we will add to artificial block process id.

**if** (PRCID == 0) {

blockchainList.add(ArtificialBlock(PRCID));

}

//otherwise we will start block chain and start its server.

Blockchain bc = **new** Blockchain();

bc.startBlockchainServers();

//sending key value

KeySend();

//adding process ID to blockchain list.

blockchainList.addAll(Blockchain.createBlocks(Blockchain.PRCID, argv));

//adding gson builder to do pretty printing

Gson gson = **new** GsonBuilder().setPrettyPrinting().create();

/\*\*

\* Converting the Json objects.

\*/

String json = gson.toJson(blockchainList);

System.out.println("\n JSON String list is: " + json);

/\*\*

\* Writing the json to a file

\*/

**try** (FileWriter writer = **new** FileWriter("myList.json")) {

//adding file writer to myList-Json

gson.toJson(blockchainList, writer);

} **catch** (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* Start the blockchain servers based on the public key and unverified servers

\*/

**public** **void** startBlockchainServers(){

//Start servers

PublicKeyServer PKS = **new** PublicKeyServer();

//starting the public key server

PKS.start();

//starting the queue

UnverifiedBlockServer UBS = **new** UnverifiedBlockServer(queue);

//starting the server

UBS.start();

}

**public** **void** startingTheBlockServer(**int** processIdOfServer) {

String processIDOfServer;

**int** serverCommunication = 0;

**long** communicateStringOfProcess = (**long**) 0.0;

}

**public** **void** checkTheServerParameters(**int** id){

String name = "ServerID";

}

/\*\*

\* Setting the block as artificial takes parameteres id

\* **@param** id

\* **@return**

\*

\*/

**private** **static** BlockRecord ArtificialBlock(**int** id) {

Date date = **new** Date(); //Class Date is a part of the Java standard library

// Whenever a new Date object is created with no arguments, it represents the current date and time.

BlockRecord BR = **new** BlockRecord(); // This will create a new block record

String T1 = String.format("%1$s %2$tF.%2$tT", "", date);

String TimeStampString = T1 + "." + id; // There will be no timestamp collisions!

BR.setTimeStamp(TimeStampString); // Will be able to priority sort by TimeStamp

String[] tokens;

String suuid = UUID.randomUUID().toString();

//uuid to be set as block id.

BR.setBlockID(suuid);

// In below code we have defined data for dummy block but in our case its an Artificial block

BR.setFname("ROHAN"); // The FNAME will be ROHAN in the dummy block

BR.setLname("DHOYDA"); // The LNAME will be DHOYDA in the dummy block

BR.setSSNum("668-775-869"); // The serial Number will be 668-775-869 in the dummy block

BR.setDOB("03-19-1995"); // The DOB will be 03-19-1995 in the dummy block

BR.setDiag(" Captain America"); // The Diag will be Captain America in the dummy block

BR.setTreat("Avengers"); // The Treat will be Avengers in the dummy block

BR.setRx("scd");

BR.setWinningHash("1721"); // The winning hash will be 1721 in the dummy block

**return** BR;

}

**public** **void** checkFilesParameters(String fileName){

String fileCheckIfEmptyOrNot;

String fileGrouping;

**int** fileID = 0;

}

/\*\*

\* This is the block creation and adding all the block records in a list

\* **@param** id

\* **@param** args

\* **@return**

\*/

**public** **static** List<BlockRecord> createBlocks(**int** id, String[] args) {

List<BlockRecord> list = **new** ArrayList<>(); // This code will create a array list of block record objects named list. The parameter block record specifies

// The parameter block record specifies that it will only specify block record object

**try** {

//based on the value of process id , we can set the file name.

**switch** (Blockchain.PRCID) {

**case** 1:

FILENAME = "BlockInput1.txt";

**break**;

**case** 2:

FILENAME = "BlockInput2.txt";

**break**;

**default**:

FILENAME = "BlockInput0.txt";

**break**;

}

//using buffered reader to read the filename based on switch case result.

System.out.println("Using input file: " + FILENAME);

BufferedReader br = **new** BufferedReader(**new** FileReader(FILENAME));

String[] tokens = **new** String[10];

String InputLineStr;

String suuid;

UUID idA;

BlockRecord tempRec;

**while** ((InputLineStr = br.readLine()) != **null**) {

Date date = **new** Date(); //Class Date is a part of the Java standard library

// Whenever a new Date object is created with no arguments, it represents the current date and time.

BlockRecord BR = **new** BlockRecord(); // This will create a block record

//String T1 = String.format("%1$s %2$tF.%2$tT", "Timestamp:", date);

String T1 = String.format("%1$s %2$tF.%2$tT", "", date);

String TimeStampString = T1 + "." + PRCID; // No timestamp collisions!

// System.out.println("Timestamp: " + TimeStampString);

BR.setTimeStamp(TimeStampString); // Will be able to priority sort by TimeStamp

suuid = **new** String(UUID.randomUUID().toString()); // A unique block ID will be generated

BR.setBlockID(suuid);

/\* Put the file data into the block record: \*/

tokens = InputLineStr.split(" +"); // Tokenize the input

BR.setFname(tokens[iFNAME]);

BR.setLname(tokens[iLNAME]);

BR.setSSNum(tokens[iSSNUM]);

BR.setDOB(tokens[iDOB]);

BR.setDiag(tokens[iDIAG]);

BR.setTreat(tokens[iTREAT]);

BR.setRx(tokens[iRX]);

String blockRecStr = BR.getBlockID() + BR.getFname() + BR.getLname() +

BR.getSSNum() + BR.getDOB() + BR.getDiag() +

BR.getTreat() + BR.getRx();

String hash = calculateBlockHash(blockRecStr);

BR.setWinningHash(hash);

BR.setTimeStamp(String.valueOf(**new** Date().getTime()));

list.add(BR);

Socket sock; // Creating a Server so that the processes can communicate with clients

PrintStream toServer;

**try**{

Gson gson = **new** GsonBuilder().setPrettyPrinting().create();

// We will firstly convert the Java object to a JSON String:

// for(int i=0; i< 2; i++){// Send our public key to all servers.

sock = **new** Socket("localhost", Ports.UnverifiedBlockServerPortBase + Blockchain.PRCID);

toServer = **new** PrintStream(sock.getOutputStream());

String json = gson.toJson(BR);

toServer.println(json );

System.out.println(String.format("Unverified Block sent to process %s",Blockchain.PRCID));

// toServer.println("FakeKeyProcess" + Blockchain.PRCID); toServer.flush();

sock.close();

// }

}**catch** (Exception x) {x.printStackTrace ();}

}

br.close();

// fr.close();

**return** list;

} **catch** (IOException e) {

System.out.println("An error occurred while reading the file: " + e.getMessage());

}

**return** **null**;

}

String value;

// The block hash calculates the previous blocks SHA256, the current data and a random string that solves the puzzle and that particular block is verified

**public** **void** checkingParameters(**long** id){

String checkTheParameters = "";

**int** valueOFId =0 ;

}

/\*\*

\* This method is to used to calculate the block hash using the value provided as an argument.

\* **@param** value

\* **@return** String value of hash

\*/

**public** **static** String calculateBlockHash(String value) {

MessageDigest digest = **null**;

**byte**[] bytes = **null**;

**try** {

digest = MessageDigest.getInstance("SHA-256");

bytes = digest.digest(value.getBytes(UTF\_8));

} **catch** (NoSuchAlgorithmException ex) {

ex.printStackTrace();

}

StringBuffer buffer = **new** StringBuffer();

**for** (**byte** b : bytes) {

buffer.append(String.format("%02x", b));

}

**return** buffer.toString();

}

**public** **void** changingBlockChainServersToVerifyCondition(**int** serverID){

**int** serverConfig = 0;

String ServerClientName;

**float** clientPID = 0.0f;

}

/\*\*

\* This method is used to generate a key value pair based on seeding of the server

\* **@param** seed

\* **@return**

\* **@throws** Exception

\*/

**public** **static** KeyPair generateKeyPair(**long** seed) **throws** Exception {

KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA");

// The KeyPairGenerator generate pairs for public and private keys

SecureRandom rng = SecureRandom.getInstance("SHA1PRNG", "SUN");

rng.setSeed(seed);

keyGenerator.initialize(1024, rng);

**return** (keyGenerator.generateKeyPair());

}

**public** **void** derivingComparatorValue(**int** processIDWhichDenotesProcessesCommunication){

**int** processesValue = 0;

String blockRecordUnderstandingParameters;

**float** id = 0.0f;

}

/\*\*

\* This method is used to send the value of the key , it can be multiCast to other processes.

\*/

**public** **static** **void** KeySend () { // Multicast our public key to the other processes

Socket sock;

PrintStream toServer;

**try**{

KeyPair key = generateKeyPair(0);

// for(int i=0; i< 2; i++){// Send our public key to all servers.

sock = **new** Socket("localhost", Ports.KeyServerPortBase + Blockchain.PRCID);

toServer = **new** PrintStream(sock.getOutputStream());

toServer.println(key.getPublic());

System.out.println(String.format("Public key sent to process %s",Blockchain.PRCID));

// toServer.println("FakeKeyProcess" + Blockchain.PRCID); toServer.flush();

sock.close();

// }

}**catch** (Exception x) {x.printStackTrace ();}

}

**public** **int** keyCheck(**int** key1, **int** key2){

**return** 0;

}

}

/\*\*

\* This class is used to derive public key and extends all the threads parameter based on socket configuration

\*/

**class** PublicKeyWorker **extends** Thread { // Worker thread to process incoming public keys

Socket keySock; // Class member, socket, local to Worker.

PublicKeyWorker (Socket s) {keySock = s;} // Constructor, assign arg s to local sock

**public** **void** run(){

**try**{

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(keySock.getInputStream()));

String data = in.readLine ();

System.out.println("Got key: " + data);

keySock.close();

} **catch** (IOException x){x.printStackTrace();}

}

**public** **void** checkThreadParameters(**int** threadID){

**int** threadIDOfServer =0 ;

**int** processId = 1;

}

}

/\*\*

\* This class is used to create a public key server which extends the parameters of a thread.

\*/

**class** PublicKeyServer **extends** Thread {

//public ProcessBlock[] PBlock = new ProcessBlock[3]; // Typical would be: One block to store info for each process.

/\*\*

\* This method starts the input server using the input port

\* Also it creates the public key to based on server configuration to access the thread.

\*/

**public** **void** run(){

**int** q\_len = 6;

Socket keySock;

System.out.println("Starting Key Server input thread using " + Integer.toString(Ports.KeyServerPort));

**try**{

ServerSocket servsock = **new** ServerSocket(Ports.KeyServerPort, q\_len);

**while** (**true**) {

keySock = servsock.accept();

//creating new public key and starting the server

**new** PublicKeyWorker (keySock).start();

}

}**catch** (IOException ioe) {System.out.println(ioe);}

}

}

**class** UnverifiedBlockServer **extends** Thread {

BlockingQueue<BlockRecord> queue;

/\*\*

\* This is a constructor accepting a queue.

\* **@param** queue

\*/

UnverifiedBlockServer(BlockingQueue<BlockRecord> queue) {

**this**.queue = queue;

}

/\*\*

\* The unverified block receives messages from the server using the port configuration.

\* Stores the requests in the queue and executes a socket based on queue length and unverified block

\*/

**public** **void** run() {

**int** q\_len = 6;

Socket sock;

System.out.println("Starting the Unverified Block Server input thread using " +

Integer.toString(Ports.UnverifiedBlockServerPort));

**try** {

ServerSocket UVBServer = **new** ServerSocket(Ports.UnverifiedBlockServerPort, q\_len);

**while** (**true**) {

//Executes a new unverified block

sock = UVBServer.accept();

//Starting the process of a thread

**new** UnverifiedBlockWorker(sock, queue).start();

}

} **catch** (IOException ioe) {

System.out.println(ioe);

}

}

}

/\*\*

\* This class share the priority queue and places the blocks in this but can be retrieved randomly by the clients based on their timestamps

\*/

**class** UnverifiedBlockWorker **extends** Thread {

BlockingQueue<BlockRecord> queue;

Socket sock;

/\*\*

\* This constructor creates a socket and queue of an unverified block

\* **@param** s

\* **@param** queue

\* Also the queue is initialised in this constructor

\*/

UnverifiedBlockWorker (Socket s, BlockingQueue<BlockRecord> queue) {sock = s;

**this**.queue = queue;

}

/\*\*

\* This method creates an input stream and reads the data from the block

\* and converts it into json format.

\* Also, the block record is put in the queue based on the priority

\*/

**public** **void** run(){

**try**{

//ObjectInputStream unverifiedIn = new ObjectInputStream(sock.getInputStream());

BufferedInputStream breader = **new** BufferedInputStream(sock.getInputStream());

String block = **new** String(breader.readAllBytes());

Gson gson = **new** Gson();

BlockRecord BR = gson.fromJson(block,BlockRecord.**class**);

// System.out.println("Received UVB: " + br.getTimeStamp() + " " + br.);

queue.put(BR); // Note: make sure you have a large enough blocking priority queue to accept all the puts

sock.close();

} **catch** (Exception x){x.printStackTrace();}

}

}

/\*\*

\* This class is used to configure port configuration based on key server and unverified block.

\*/

**class** Ports{

**public** **static** **int** KeyServerPortBase = 4710;

**public** **static** **int** UnverifiedBlockServerPortBase = 4820;

**public** **static** **int** BlockchainServerPortBase = 4930;

**public** **static** **int** KeyServerPort;

**public** **static** **int** UnverifiedBlockServerPort;

**public** **static** **int** BlockchainServerPort;

/\*\*

\* This method is used to set the port using process id and blockchain id.

\*/

**public** **static** **void** setPorts(){

KeyServerPort = KeyServerPortBase + (Blockchain.PRCID);

UnverifiedBlockServerPort = UnverifiedBlockServerPortBase + (Blockchain.PRCID );

BlockchainServerPort = BlockchainServerPortBase + (Blockchain.PRCID );

}

}