/\*--------------------------------------------------------

1. Name / Date: Nikhil Komal Kumar Yakkala / 26-02-2023

2. Java version used:

java 19.0.1 2022-10-18

Java(TM) SE Runtime Environment (build 19.0.1+10-21)

Java HotSpot(TM) 64-Bit Server VM (build 19.0.1+10-21, mixed mode, sharing)

3. Compling the program:

> javac -cp "gson-2.8.6.jar" Blockchain.java

4. Running the Program:

Open 3 seperate terminals and type in the command

> java cp ".:gson-2.8.6.jar" Blockchain 0

> java cp ".:gson-2.8.6.jar" Blockchain 1

> java cp ".:gson-2.8.6.jar" Blockchain 2

5. List of files needed for running the program.

i) Blockchain.java

ii) BlockInput0.txt, BlockInput1.txt, BlockInput2.txt

List of files submitted including the above files

iii) checklist-block.html

iv) BlockchainLog.txt

v) BlockchainLedgerSample.json

6. Web Sources Credits:

THANKS to Professor for providing the utitlity code which helped a lot in working with the program!

----------------------------------------------------------\*/

// JSON Libraries

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import com.google.gson.reflect.TypeToken;

// Importing all essentials libraries IO, Util etc..

import java.io.\*;

import java.lang.reflect.Type;

import java.net.ServerSocket;

import java.net.Socket;

import java.nio.charset.StandardCharsets;

import java.security.\*;

import java.security.spec.X509EncodedKeySpec;

import java.util.\*;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.PriorityBlockingQueue;

public class Blockchain {

public static void main(String[] args) {

int process\_id; // declaration of variable process id.

if (args.length < 1)

process\_id = 0;

// using a loop to read the argument given, which are the process ids.

switch (args[0]) {

case "0":

process\_id = 0; // assigning the process id 0 if the given argument is 0.

break;

case "1":

process\_id = 1; // assigning the process id 1 if the given argument is 1.

break;

case "2":

process\_id = 2; // assigning the process id 2 if the given argument is 2.

break;

default:

process\_id = 0; // assigning the default value as 0.

break;

}

/\*Passing the Process id to the constructor of the class blockChainWorker to perform the task for that process id\*/

blockChainWorker bcTtd = new blockChainWorker(process\_id);

}

}

class blockChainRecord implements Serializable {

private String block\_id; // variable block\_id of string type

private String signedBlock\_id; // variable signed block idd of strin type

private String timeStamp; // variable time stamp of type string

private String block\_num; // variable block number of type stirng

private String first\_name; // variable first name of teype stirng

private String last\_name; // variable last name of type string

private String data\_of\_birth; // variable data of birth of string datatype

private String SSN; // variable SSN of string data type

private String diagnosis; // variable diagnosis of string data type

private String treatment; // variable treatment of string data type

private String prescription; // variable precription of string data type

private String hashValue; // variable hash value of stirng data type

private String signedHashValue; // variable signed hash value of string data type

private String hashOfPreviousBlock; // variable hash of previous block of string data type

private String hashOfWinningBlock; // variable hash of winning block of string data type

private String signedHashOfWinningBlock; // variable signed hash of winning block of string data type

private String randomSeed; // variable random seed of string data type

private String processIdVerify; // variable process id verify of string data type

private String process; // variable process of string data type

private UUID uuid; // variable UUID of UUID data type

// set method for block id variable

public void setblock\_id(String block\_id){

this.block\_id = block\_id;

}

// get method for block id variable

public String getblock\_id(){

return block\_id;

}

// set method for signed block id variable

public void setsignedBlock\_id(String signedBlock\_id){

this.signedBlock\_id = signedBlock\_id;

}

// get method for signed block id variable

public String getsignedBlock\_id(){

return signedBlock\_id;

}

// set method for time stamp variable

public void settimeStamp(String timeStamp){

this.timeStamp = timeStamp;

}

// get method for time stamp variable

public String gettimeStamp(){

return timeStamp;

}

// set method for block number variable

public void setblock\_num(String block\_num){

this.block\_num = block\_num;

}

// get method for block number variable

public String getblock\_num(){

return block\_num;

}

// set method for first name variable

public void setfirst\_name(String first\_name){

this.first\_name = first\_name;

}

// get method for first name variable

public String getfirst\_name(){

return first\_name;

}

// set method for last name variable

public void setlast\_name(String last\_name){

this.last\_name = last\_name;

}

// get method for last name variable

public String getlast\_name(){

return last\_name;

}

// set method for DOB variable

public void setdata\_of\_birth(String data\_of\_birth){

this.data\_of\_birth = data\_of\_birth;

}

// get method for DOB variable

public String getdata\_of\_birth(){

return data\_of\_birth;

}

// set method for SSN variable

public void setSSN(String SSN){

this.SSN = SSN;

}

// get method for SSN variable

public String getSSN(){

return SSN;

}

// set method for diagnosis variable

public void setdiagnosis(String diagnosis){

this.diagnosis = diagnosis;

}

// get method for diagnosis variable

public String getdiagnosis(){

return diagnosis;

}

// set method for treatment variable

public void settreatment(String treatment){

this.treatment = treatment;

}

// get method for treatment variable

public String gettreatment(){

return treatment;

}

// set method for prescription variable

public void setPrescription(String prescription){

this.prescription = prescription;

}

// get method for prescription variable

public String getPrescription(){

return prescription;

}

// set method for hash variable

public void setHashValue(String hashValue){

this.hashValue = hashValue;

}

// get method for hash variable

public String getHashValue(){

return hashValue;

}

// set method for signed hash value variable

public void setSignedHashValue(String signedHashValue){

this.signedHashValue = signedHashValue;

}

// get method for signed hash value variable

public String getSignedHashValue(){

return signedHashValue;

}

// set method for hash of previous block variable

public void sethashOfPreviousBlock(String hashOfPreviousBlock){

this.hashOfPreviousBlock = hashOfPreviousBlock;

}

// get method for hash or previous block variable

public String gethashOfPreviousBlock(){

return hashOfPreviousBlock;

}

// set method for hash of winning block variable

public void sethashOfWinningBlock(String hashOfWinningBlock){

this.hashOfWinningBlock = hashOfWinningBlock;

}

// get method for hash of winning block variable

public String gethashOfWinningBlock(){

return hashOfWinningBlock;

}

// set method for signed hash of winning block variable

public void setsignedHashOfWinningBlock(String signedHashOfWinningBlock){

this.signedHashOfWinningBlock = signedHashOfWinningBlock;

}

// get method for signed hash of winning block variable

public String getsignedHashOfWinningBlock(){

return signedHashOfWinningBlock;

}

// set method for random seed variable

public void setrandomSeed(String randomSeed){

this.randomSeed = randomSeed;

}

// get method for random seed variable

public String getrandomSeed(){

return randomSeed;

}

// set method for process idd verify variable

public void setprocessIdVerify(String processIdVerify){

this.processIdVerify = processIdVerify;

}

// get method for processs id verify variable

public String getprocessIdVerify(){

return processIdVerify;

}

// set method for process variable

public void setprocess(String process){

this.process = process;

}

// get method for process variable

public String getprocess(){

return process;

}

// set method for uuid variable

public void setuuid(UUID uuid){

this.uuid = uuid;

}

// get method for uuid variable

public UUID getuuid(){

return uuid;

}

@Override

public String toString() { // to string method that creates a string using all the variables

return "BlockRecord{" +

"block\_id='" + block\_id + '\'' +

", signedBlock\_id='" + signedBlock\_id + '\'' +

", timeStamp='" + timeStamp + '\'' +

", block\_num='" + block\_num + '\'' +

", first\_name='" + first\_name + '\'' +

", last\_name='" + last\_name + '\'' +

", data\_of\_birth='" + data\_of\_birth + '\'' +

", SSN='" + SSN + '\'' +

", diagnosis='" + diagnosis + '\'' +

", treatment='" + treatment + '\'' +

", prescription='" + prescription + '\'' +

", hashValue='" + hashValue + '\'' +

", signedHashValue='" + signedHashValue + '\'' +

", hashOfPreviousBlock='" + hashOfPreviousBlock + '\'' +

", hashOfWinningBlock='" + hashOfWinningBlock + '\'' +

", signedHashOfWinningBlock='" + signedHashOfWinningBlock + '\'' +

", randomSeed='" + randomSeed + '\'' +

", processIdVerify='" + processIdVerify + '\'' +

", process='" + process + '\'' +

", uuid=" + uuid +

'}';

}

}

class blockChainWorker {

public static int process\_id; // process id variable of int data type.

public static int totalProcesses = 3; // variable declaring the total processes of int data type

public static String serverName = "localhost"; // variable declaring server name of string data type

public static boolean processFlag = false; // vairbale storing the process flag of boolean data type

public static boolean publicKeyFlag = false; // varibale storing the public key flag of boolean data type

public static int pkCounter = 0; // variable storing counter of int data type

public static KeyPair processKeyPair; // variable to store the process key pair of keypain data type

public static PublicKey[] pKeyArray = new PublicKey[totalProcesses]; // variable to store the public key array of type publickey

public static final PriorityBlockingQueue<blockChainRecord> priorityQueue = new PriorityBlockingQueue<>(50, new BRComparator()); // varibale queue of priority blockgn queue

public static LinkedList<blockChainRecord> verifiedBlocks = new LinkedList<>(); // variable verified blocks of linked list data type

public static LinkedList<blockChainRecord> unverifiedBlocks = new LinkedList<>(); // variable unverified block of linked list data type

private static final int iFirstName = 0; // Variable to store index of first name which set to 0.

private static final int iLastName = 1; // Variable to store index of last name which set to 1.

private static final int iDOB = 2; // Variable to store index of DOB which set to 2.

private static final int iSSN = 3; // Variable to store index of SSN which set to 3.

private static final int idiagnosis = 4; // Variable to store index of Diagnosis which set to 4.

private static final int itreatment = 5; // Variable to store index of Treatment which set to 5.

private static final int iprescription = 6; // Variable to store index of prescription which set to 6.

public blockChainWorker(int process\_id) { // Updating teh process id.

blockChainWorker.process\_id = process\_id;

new serverPorts().setPorts(process\_id); // setting the ports for each process.

run();

}

public void run() {

System.out.println("BlockChain Program has started up......\n");

System.out.println(String.format("BlockInput%d.txt", process\_id) + " is being read by the program.");

// Starting all the servers.

new Thread(new mainServer()).start(); // Main Server

new Thread(new PublicKeysServer()).start();// Public Key Server

new Thread(new unverifiedBlockServer(priorityQueue)).start();// Unverified Block Server

new Thread(new updatedBlockServer()).start();// Updated Block Server

try {

Thread.sleep(5000); // Server sleeps for five seconds to start all the processes.

} catch (Exception ex) { //Handing exception

ex.printStackTrace();

}

if (process\_id == 2) // Start all the processes when hte process id is 2.

startUpAllProcesses(); // waking all the processes to start.

try {

processKeyPair = getKeyPair(999); // generating the key value pair

} catch (Exception ex) { // Handing exception

ex.printStackTrace();

}

while (!processFlag) // Calling hte sleep method is the process flag is flase. It waits until all hte processes are called up.

Sleep();

System.out.println("Starting the Program Block Chain");

castPublicKeys(); // Method that shares the public keys wih one and another. Waking it up.

while (!publicKeyFlag) // Calling hte sleep method is the process flag is flase.

Sleep();

if (process\_id == 0) // creating the dummy block as a start block if the process is 0.

writestartBlock();

readBlockInputFile(); // Calling the method to read the given input file accordingly.

multiCastProcesses(); // calling the multicast method to marshall data among the process to get the data verified.

try {

Thread.sleep(2000);

} catch (InterruptedException ex) { // Handling the Exception

ex.printStackTrace();

}

new Thread( new WorkPuzzle(priorityQueue)).start(); // starting the work puzzle thread to solve the puzzle

try {

Thread.sleep(21000);

} catch (Exception ex) { // Handling the Exception

ex.printStackTrace();

}

System.out.println("JSON has been created and the file has been created into the disk");

System.out.println("You can ask the program for the options credits, verify or list.");

Scanner sc = new Scanner(System.in);

String input;

do { // looping to get through the used given commands.

System.out.println("\nEnter C or V or L: ");

input = sc.nextLine().toLowerCase();

if(input.indexOf("quit") < 0) {

if(input.equals("c"))

getCredits();

else if(input.equals("v"))

verifyBlock();

else if(input.equals("l"))

getBlockRecords();

else

System.out.println("Choose only any of the following commands: 'C/V/L'");

}

} while(input.indexOf("quit") < 0);

sc.close();

}

// method to marshall the unverified data among the processes which sloved the puzzle

public void multiCastProcesses() {

Socket MCPSock; // SOcket Initialization

PrintStream dataToServer; // Print stream variable used to marshall data over server.

blockChainRecord temporaryBlockRecord; // variable to store the temp block record.

Iterator<blockChainRecord> blockRecordIterator = unverifiedBlocks.iterator(); // Block record iterator to iterate all the unverified blocks.

try {

while (blockRecordIterator.hasNext()) { // loop to read the records

temporaryBlockRecord = blockRecordIterator.next(); // iterating the blockRecord Iterator.

String blockRecord = buildJson(temporaryBlockRecord); // JSon format is created for the records.

for (int processCounter = 0; processCounter < totalProcesses; processCounter++) {

MCPSock = new Socket(serverName, serverPorts.unverifiedBlockServerPort + processCounter); // starting the socket.

dataToServer = new PrintStream(MCPSock.getOutputStream());

dataToServer.println(blockRecord); // marshalling the data over the network.

dataToServer.flush(); // flush the data,

MCPSock.close(); // ending the connection.

}

}

} catch (Exception ex) { // Handling teh Exception

ex.printStackTrace();

}

}

// duplicate check method to find the duplicate records.

public static boolean duplicateRecordCheck(blockChainRecord blockChainRecordordInd) {

blockChainRecord datarecord = blockChainRecordordInd; // variable block record to assign the passed block to verify the duplicate.

for (blockChainRecord dupicateRecord : verifiedBlocks) { // running the record through the ledger we have stored in the disk and validate if duplicate occurs.

if (datarecord.getblock\_id().equals(dupicateRecord.getblock\_id()))

return true;

}

return false;

}

/\* This method generates the keypairs

I have used the utility code provided by the professor. \*/

public static KeyPair getKeyPair(long randomSeed) throws Exception {

KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance("RSA"); // Getting a key pair with the use of RSA algorithm.

SecureRandom srandom = SecureRandom.getInstance("SHA1PRNG", "SUN"); // re-seeding hte srandom object by setting a random seed value.

srandom.setSeed(randomSeed);

keyGenerator.initialize(1024, srandom); // initializing the keygenerator by sending the size of key.

return (keyGenerator.generateKeyPair()); // calling the key pair generator.

}

/\* cast Public keys is used to pass the public keys among all the processes.

Process id is added to the key \*/

public void castPublicKeys() {

Socket CPSock; // SOcket Initialization

PrintStream datatoServer; // Print stream variable used to marshall data over server.

byte[] pKey = processKeyPair.getPublic().getEncoded(); // getting the byte value of the public keys.

String stringPubKey = Base64.getEncoder().encodeToString(pKey); // storing the variable in string format that stores in our blocks.

System.out.println("Public Key created for Multicasting is : " + stringPubKey);

try {

for (int processCounter = 0; processCounter < totalProcesses; processCounter++) { // casting the public keys for all the processes.

CPSock = new Socket(serverName, serverPorts.serverPublicKeyPort + processCounter); // assigning the socket variable.

datatoServer = new PrintStream(CPSock.getOutputStream()); // processing data to server.

String updated\_PublicKey = process\_id + " " + stringPubKey; // updating the public key appending the process id to it.

datatoServer.println(updated\_PublicKey); // marshalling the public key.

datatoServer.flush();// Flush the object.

CPSock.close(); // ending the connection.

}

} catch (Exception ex) { // Handling exception

ex.printStackTrace();

}

}

// change the flag to true when the process 2 is called by starting all the process.

public boolean startUpAllProcesses() {

Socket SPSock; // Socket initialization

PrintStream datatoServer; // Print stream variable used to marshall data over server.

try {

for (int processCounter = 0; processCounter < totalProcesses; processCounter++) { // assign socket and send signal to server using a loop.

SPSock = new Socket(serverName, serverPorts.startServerPort + processCounter); // assigning the socket variable.

datatoServer = new PrintStream(SPSock.getOutputStream()); // processing data to server.

datatoServer.println("start");// sending message to server.

System.out.println("Starting all the processes");// confirming that the signal was sent to start the process.

datatoServer.flush();// Flush the object.

SPSock.close(); // ending the connection.

}

} catch (Exception ex) {// Handling exception

ex.printStackTrace();

}

return true; // after teh processes start the method turns true.

}

// This method creates a dummy block with the given hard coded values.

public static void writestartBlock() {

String SHA256\_data; // variable to store the SHA256.

blockChainRecord blockRecord = new blockChainRecord(); // object of type block chain record

Date date = new Date(); // variable date to store the date.

long time = date.getTime(); // time variable.

String timeStamp = String.valueOf(time); // time variable converted into string.

String timeStampString = timeStamp + "." + process\_id; // converting time into timestamp by adding process ids.

String UniqueUID = UUID.randomUUID().toString(); // variable UUID.

// setting the values for the first block by hard coding them.

blockRecord.setblock\_id(UniqueUID); // block id variable initialization.

blockRecord.settimeStamp(timeStampString); // timestamp variable initialization.

blockRecord.setfirst\_name("Nikhil"); // first name variable initialization.

blockRecord.setlast\_name("Yakkala"); // last name variable initialization.

blockRecord.setSSN("187-42-1234"); // SSN variable initialization.

blockRecord.setdata\_of\_birth("1999.09.16"); // DOB variable initialization.

blockRecord.setdiagnosis("Meningitis"); // diagnosis variable initialization.

blockRecord.settreatment("PainMedicine"); // treatment variable initialization.

blockRecord.setPrescription("HealthyFood"); // prescription variable initialization.

blockRecord.sethashOfPreviousBlock("1111111111"); // hash of previous block variable initialization.

blockRecord.setblock\_num("1"); // block num variable initialization.

// creating a string variable to store the block data which is used to generate the hash value.

String block\_record = blockRecord.getblock\_id() +

blockRecord.getfirst\_name() +

blockRecord.getlast\_name() +

blockRecord.getSSN() +

blockRecord.getdata\_of\_birth() +

blockRecord.getdiagnosis() +

blockRecord.gettreatment() +

blockRecord.getPrescription();

/\* process of creating a hash we are passing the string we build to MD2StringBuilder first,

the returned hash value is set as winning hash value.\*/

SHA256\_data = MD2StringBuilder(block\_record);

blockRecord.sethashOfWinningBlock(SHA256\_data);

// This dummy record is added into the verified block at the first index.

verifiedBlocks.add(0, blockRecord);

if (process\_id == 0) {// the program writes the first record created to the ledger if process is 0.

buildLedgerJson(blockRecord, "update"); // sending the block to ledger to modify it.

System.out.println("Ledger Modified!!! The Dummy Block has been written into the Ledger"); // confirming that we have written the block to ledger.

writeJSONToDisk(); // writing the block to json format which will be stored to disk after written.

}

}

public static void buildLedgerJson(blockChainRecord blockRecord, String value) {

Socket BLJSock; // Socket initialization

PrintStream dataToServer; // Print stream variable to marshall the data to server.

if(value.equals("update")) { // modifying the ledger if the block record is being to written.

try {

for (int processCounter = 0; processCounter < totalProcesses; processCounter++) { // this loop runs for all the processes.

BLJSock = new Socket(serverName, serverPorts.updatedBlockChainPort + processCounter); // Socket declaration.

dataToServer = new PrintStream(BLJSock.getOutputStream()); // Marshalling data over the network.

dataToServer.println(buildJson(blockRecord)); // creating a json format of block record that will be added to the ledger.

System.out.println("Updated block : " + blockRecord.getblock\_id()); // confirming that we have added the block to ledger.

dataToServer.flush(); // Flush the data.

BLJSock.close(); // ending the connection.

}

} catch (IOException IOex) { // Handling Exception

IOex.printStackTrace();

}

}

else if(value.equals("verify")) { // verifying the ledger if asked for verfication of block.

try {

for (int processCounter2 = 0; processCounter2 < totalProcesses; processCounter2++) { // this loop runs for all the processes.

BLJSock = new Socket(serverName, serverPorts.unverifiedBlockServerPort + processCounter2); // Socket declaration.

dataToServer = new PrintStream(BLJSock.getOutputStream()); // Marshalling data over the network.

dataToServer.println(buildJson(blockRecord)); // creating a json format of block record that will be added to the ledger.

System.out.println("Verified block : " + blockRecord.getblock\_id()); // confirming that we have verified hte block in the ledger.

dataToServer.flush(); // Flush the data.

BLJSock.close(); // ending the connection.

}

} catch (IOException IOex) { // Handling Exception

IOex.printStackTrace();

}

}

}

// build Json method that creates a Json format off the data we pass into it using the GSON object.

public static String buildJson(blockChainRecord blockRecord) {

Gson gs = new GsonBuilder().setPrettyPrinting().create(); // creating a GSON object.

String jsonofblock = gs.toJson(blockRecord); // converting the block record to Json using the object.

return jsonofblock; // returning the json record.

}

/\*The input files given to the program BlockInput0.txt, BlockInput1.txt and BlockInput2.txt are processed in this method.

the generated token for each block is colleceted nad constructs an unverified block. \*/

public static void readBlockInputFile() {

String fileinput = String.format("BlockInput%d.txt", process\_id); // The input file is formatted to a string format.

try {

BufferedReader input = new BufferedReader(new FileReader(fileinput)); // getting the data from socket using BufferedReader.

String[] blockData; // variable storing hte block.

String blockDataString; // variable string to store the string.

String UUIDString;

try {

while ((blockDataString = input.readLine()) != null) { // going through the input data.

Date date2 = new Date(); // variable of date datatype.

blockChainRecord blockRecord = new blockChainRecord(); // block record variable.

long time = date2.getTime(); // tiem variable.

String timeStamp = String.valueOf(time); // creating a timestamp variable.

String timeStampString = timeStamp + "." + process\_id; // time stamp string created using timestamp and process id.

UUIDString = UUID.randomUUID().toString(); // UUID String.

blockData = blockDataString.split(" +"); // block data is split by reading the input.

String signedHashBlock = ""; // signed hash block variable.

try {

byte[] digitalSignature1 = signData(UUIDString.getBytes(), processKeyPair.getPrivate()); // getting the byte array using signdata method passing uuid string and key pair.

signedHashBlock = Base64.getEncoder().encodeToString(digitalSignature1); // the byte array is encoded to string and stored int the hash block.

} catch (Exception ex) { // Handlign the Exception

ex.printStackTrace();

}

// setting the values for all the blocks.

blockRecord.setblock\_id(UUIDString); // block id variable initialization

blockRecord.settimeStamp(timeStampString); // timestamp variable initialization.

blockRecord.setsignedBlock\_id(signedHashBlock); // signed block id variable initialization.

blockRecord.setprocess(String.valueOf(process\_id)); // process id variable initialization.

blockRecord.setfirst\_name(blockData[iFirstName]); // first name variable initialization.

blockRecord.setlast\_name(blockData[iLastName]); // last name variable initialization.

blockRecord.setSSN(blockData[iSSN]); // SSN variable initialization.

blockRecord.setdata\_of\_birth(blockData[iDOB]); // DOB variable initialization.

blockRecord.setdiagnosis(blockData[idiagnosis]); // diagnosis variable initialization.

blockRecord.settreatment(blockData[itreatment]); // treatment variable initialization.

blockRecord.setPrescription(blockData[iprescription]); // prescription variable initialization.

unverifiedBlocks.add(blockRecord); // adding the block into the unverified block record.

// creating a string to get an hash value out of by appending all the records of the block.

String stringOfBlockRecord = blockRecord.getblock\_id() + blockRecord.getfirst\_name() + blockRecord.getlast\_name() +

blockRecord.getSSN() + blockRecord.getdata\_of\_birth() + blockRecord.getdiagnosis() +

blockRecord.gettreatment() + blockRecord.getPrescription() + blockRecord.getprocess();

String hexDecimalHashString = MD2StringBuilder(stringOfBlockRecord); // process of creating a hash we are passing the string we build to MD2StringBuilder

String hashSignedString = ""; // variable to store the final signed hash value.

// creating the final unverified signed hash.

try {

byte[] digitalSignature2 = signData(hexDecimalHashString.getBytes(), processKeyPair.getPrivate()); // getting the byte array using signdata method passing uuid string and key pair.

hashSignedString = Base64.getEncoder().encodeToString(digitalSignature2); // the byte array is encoded to string and stored int the hash block.

} catch (Exception ex) { // Handling teh Exception

ex.printStackTrace();

}

blockRecord.setHashValue(hexDecimalHashString); // updating the value of hash.

blockRecord.setSignedHashValue(hashSignedString); // updating the value of signed hash.

Sleep();

}

} catch (IOException IOex) { // Handling the Exception

IOex.printStackTrace();

}

} catch (FileNotFoundException FNFex) { // Handlgin the Exception

FNFex.printStackTrace();

}

}

// Creating hash value and converting it into hexadecimal before returing.

private static String MD2StringBuilder(String block\_record) {

StringBuffer hexDecimalHash = new StringBuffer(); // variable to store the hexadecimal hash.

String hexDecimalHashString = ""; // variable to store the hash in form of string.

try {

MessageDigest MD = MessageDigest.getInstance("SHA-256"); // message digest object initializations.

MD.update(block\_record.getBytes()); // getting the block records in form of bytes.

byte[] recordsInBytes = MD.digest(); // variable to read the bytes from message digest.

hexDecimalHash = new StringBuffer();

for (byte bt : recordsInBytes) // reading the bytes we stored in the variable.

hexDecimalHash.append(Integer.toString((bt & 0xff) + 0x100, 16).substring(1)); // appending bytes into hexdecimal hash by converting each byte into hexa decimal form.

hexDecimalHashString = hexDecimalHash.toString(); // converting the hash into string.

} catch (NoSuchAlgorithmException NSAex) { // Handling Exception

NSAex.printStackTrace();

}

return hexDecimalHashString; // return the hash value.

}

// MEthod to list all teh data of the block chain records.

private void getBlockRecords() {

Gson gson = new Gson(); // creating a GSON Object

LinkedList<blockChainRecord> blockRecordList; // variable block record of type linkedlist

try {

Reader fileInput = new FileReader("BlockChainLedger.json"); // Reading the ledger from the disk and displaying to the user as request.

Type tokenType = new TypeToken<LinkedList<blockChainRecord>>() {}.getType(); // Type token of block record linked list.

blockRecordList = gson.fromJson(fileInput, tokenType); // making a list from the json file.

System.out.println("The verified Blocks are displayed below:\n");

Iterator<blockChainRecord> blockchainIterator = blockRecordList.iterator(); // makibgn a iterator varibale to go through the records in the list.

int blockRecordListLength = blockRecordList.size(); // variable to store the lenght of the list.

while (blockchainIterator.hasNext()) {

blockChainRecord blockIterator = blockchainIterator.next(); // variable block iterator to iterate through teh block chain iterator.

System.out.printf("%d. " + "%s " + "%s " + "%s " + "%s " + "%s " + "%s " + "%s " + "%s \n", blockRecordListLength,blockIterator.gettimeStamp(), blockIterator.getfirst\_name(), blockIterator.getlast\_name(),

blockIterator.getdata\_of\_birth(), blockIterator.getSSN(), blockIterator.getdiagnosis(),blockIterator.gettreatment(), blockIterator.getPrescription());

blockRecordListLength--;

}

} catch (IOException IOex) { // Handling the Exceprtion

IOex.printStackTrace();

}

}

// MEthods to verfiy the block again when the user asked to

private void verifyBlock() {

boolean checkedFlag = false; // creating a variable to store the checked flag wihch determines if the verificaton is done.

Gson gson = new Gson(); // creating a GSON Object

LinkedList<blockChainRecord> blockChainList; // variable block record of type linkedlist

try {

Reader fileInput = new FileReader("BlockChainLedger.json"); // Reading the ledger from the disk and displaying to the user as request.

Type tokenType = new TypeToken<LinkedList<blockChainRecord>>() {}.getType(); // Type token of block record linked list.

blockChainList = gson.fromJson(fileInput, tokenType); // making a list from the json file.

for (blockChainRecord record : blockChainList) {

String block\_Number = record.getblock\_num(); // variable storing block number of string type.

if(!block\_Number.equals("1"))

{

String dataString; // variable data string of string data type.

String recordConcat = record.getblock\_id() + record.getfirst\_name() + record.getlast\_name() + record.getSSN() + record.getdata\_of\_birth() + record.getdiagnosis() + record.gettreatment() + record.getPrescription() + record.getprocess(); // concatning the block records into a string.

try {

String unverifiedBlock = recordConcat; // creating a string variable unverified block and concating the record string.

unverifiedBlock = unverifiedBlock + record.gethashOfPreviousBlock(); // the previous hash is also concatinated to the unverified block.

String completeData = unverifiedBlock + record.getrandomSeed(); // unverified block is added with random seed and created another string.

MessageDigest MD = MessageDigest.getInstance("SHA-256"); // creating a message digest instance of SHA-256 algo.

byte[] hashValueBytes = MD.digest(completeData.getBytes(StandardCharsets.UTF\_8)); // creating bytes variable.

dataString = WorkPuzzle.byteArray2Str(hashValueBytes); // the data string is assigned the workpuzzle hash value.

if (!dataString.equals(record.gethashOfWinningBlock())) {

System.out.println("SHa-256 verificatio n is failed.\n");

checkedFlag = true;

}

int workingNumber = Integer.parseInt(dataString.substring(0, 4), 16); // getting the first 16 bits of the hash value.

// verifyign if the puzzle was being able to solve or not

if (!(workingNumber < 20000)) {

System.out.println("Puzzle was not solved");

checkedFlag = true;

}

// verifying the hash is checked or not

try {

boolean hash\_verification = verifySignature(record.gethashOfWinningBlock().getBytes(),pKeyArray[Integer.parseInt(record.getprocessIdVerify())],Base64.getDecoder().decode(record.getsignedHashOfWinningBlock()));

if(hash\_verification) // confirming that the hash sign is verified or not

System.out.println("SHa-256 hash sign is verified for Block: " + block\_Number);

else { // confirming that the hash sign is verified or not

System.out.println("SHa-256 hash sign is verified failed for Block: " + block\_Number);

checkedFlag = true;

}

boolean signedBlock = verifySignature(record.getblock\_id().getBytes(),pKeyArray[Integer.parseInt(record.getprocess())],Base64.getDecoder().decode(record.getsignedBlock\_id())); // varible to store the signed block of the verifed block

if(signedBlock) // confirming that the block is verified or not

System.out.println(block\_Number + " Block has been verified");

else { // confirming that the block is verified or not

System.out.println(block\_Number + " Block has not verified");

checkedFlag = true;

}

} catch (Exception ex) { // Handling the Exception

ex.printStackTrace();

}

} catch (NoSuchAlgorithmException NSAex) { // Handling the Exception

NSAex.printStackTrace();

}

}

}

}

catch (Exception ex) { // Handling the Exception

ex.printStackTrace();

}

// Once the entire blockchain is verified we display the message saying that the complete blockchain is verified successfully again.

String mes = (!checkedFlag) ? "Verification of entre Blockchain completed sucessfully!" : "Errors " + "!";

System.out.println(mes);

}

// Methods to keep a count of the verified processes .

private void getCredits() {

Gson gson = new Gson(); // creating a GSON Object

LinkedList<blockChainRecord> blockChainList; // variable block record of type linkedlist

int[] credits = new int[totalProcesses]; // variable to store the credits the credits of array type.

try {

Reader fileInput = new FileReader("BlockChainLedger.json"); // Reading the ledger from the disk and displaying to the user as request.

Type tokenType = new TypeToken<LinkedList<blockChainRecord>>() {}.getType(); // Type token of block record linked list.

blockChainList = gson.fromJson(fileInput, tokenType); // making a list from the json file.

for (blockChainRecord records : blockChainList) { // going thorugh the block records

if (records.getprocessIdVerify() != null) {

int numberOftheProcess = Integer.parseInt(records.getprocessIdVerify()); // variable to store the num of processess.

credits[numberOftheProcess]++; // increasfing the counter after verifying.

}

}

// showing the credits user requested.

System.out.printf("Credit for process 'Zero': %d" + "\n" + "Credit for process 'One': %d" + "\n" + "Credit for process 'Two': %d", credits[0],credits[1], credits[2]);

} catch (IOException IOex) { // Handling the Exception

IOex.printStackTrace();

}

}

/\* I have used hte code provided by the professor to sign data. \*/

public static byte[] signData(byte[] byteData, PrivateKey PKKey)

throws SignatureException, InvalidKeyException, NoSuchAlgorithmException {

Signature signaturewithSHA = Signature.getInstance("SHA1withRSA");

signaturewithSHA.initSign(PKKey);

signaturewithSHA.update(byteData);

return (signaturewithSHA.sign());

}

/\* I have used hte code provided by the professor to sign data.

verifyiing if the data has been signed \*/

public static boolean verifySignature(byte[] byteData, PublicKey PKKey, byte[] dec)

throws NoSuchAlgorithmException, InvalidKeyException, SignatureException {

Signature verifySignature = Signature.getInstance("SHA1withRSA");

verifySignature.initVerify(PKKey);

verifySignature.update(byteData);

return (verifySignature.verify(dec));

}

// Writing hte JSon into the disk. This code has been provided by professor in the utility code

public static void writeJSONToDisk() {

System.out.println("Writing the JSON Ledger Block into disk\n");

Gson gs = new GsonBuilder().setPrettyPrinting().create(); // creating a GSON object.

String jsonofblock = gs.toJson(blockChainWorker.verifiedBlocks); // converting the block record to Json using the object.

try (FileWriter dataToFile = new FileWriter("BlockChainLedger.json")) {

gs.toJson(blockChainWorker.verifiedBlocks, dataToFile); // converting the block record to Json using the object.

} catch (IOException IOex) { // Handlign teh Exception

IOex.printStackTrace();

}

}

// Sleep method to put program on hold until all the processes start

public static void Sleep() {

try {

Thread.sleep(1000);

} catch (Exception ex) { // Handling exception

ex.printStackTrace();

}

}

}

// method used for defining the ports used int hte program.

class serverPorts {

public static int startServerPort = 4600; // variable to store the default port of hte main server.

public static int serverPublicKeyPort = 4710; // variable to store the default port of the publickey server.

public static int unverifiedBlockServerPort = 4820; // variable to store the default port of unverified block server.

public static int updatedBlockChainPort = 4930; // variable to store the default port of updated block chain server.

public static int MServerPort; // variable to store the updated main server port.

public static int PKServerPort; // variable to store hte updated public key server port.

public static int UBServerPort; // variable to store the unverified server port.

public static int UpBServerPort; // variable to store the updated update block server port.

// updating the ports of the server according to the process id given

public void setPorts(int process\_id) {

MServerPort = startServerPort + process\_id; // main server port.

PKServerPort = serverPublicKeyPort + process\_id; // public key server port.

UBServerPort = unverifiedBlockServerPort + process\_id; // unverified server port.

UpBServerPort = updatedBlockChainPort + process\_id; // updated block server port.

}

}

/\* Method compared the time stamps and adds the blocks acc to time stamps

I have used hte code provided by the professor to sign data. \*/

class BRComparator implements Comparator<blockChainRecord> {

@Override

public int compare(blockChainRecord blockrec1, blockChainRecord blockrec2) {

String date1 = blockrec1.gettimeStamp();

String date2 = blockrec2.gettimeStamp();

if (date1.equals(date2))

return 0;

if (date1 == null)

return -1;

if (date2 == null)

return 1;

return date1.compareTo(date2);

}

}

// MainServer class to start the main server and invoke the worker class.

class mainServer implements Runnable {

public void run() {

Socket MSSock;// Socket initialization

System.out.println("The Main Server started and is at port : " + serverPorts.MServerPort);// Displaying a message showing which port the Main Server is connected.

try {

ServerSocket MServerSock = new ServerSocket(serverPorts.MServerPort);

while (true) {

MSSock = MServerSock.accept();// Listening for requests

new MSWorker(MSSock).start();// Starting up the worker

}

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// MainServer Worker Class

class MSWorker extends Thread {

Socket MSWorkerSock;// Socket initialization

public MSWorker(Socket MSWorkerSock) {// Constructer used to assign the socket

this.MSWorkerSock = MSWorkerSock;

}

public void run() {

try {

BufferedReader input = new BufferedReader(new InputStreamReader(MSWorkerSock.getInputStream()));// getting the data from socket using BufferedReader.

String data = input.readLine();// reading the data.

blockChainWorker.processFlag = true;

// Closing the connection of the socket

MSWorkerSock.close();

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// Public Key class gets the keys from the processes

class PublicKeysServer implements Runnable {

public void run() {

Socket PKSock;// Socket initialization

System.out.println("Public Keys Server started and is at the port: " + serverPorts.PKServerPort);// Displaying a message showing which port the Public Key Server is connected

try {

ServerSocket PKServerSock = new ServerSocket(serverPorts.PKServerPort);

while (true) {

PKSock = PKServerSock.accept();// Listening for requests

new PublicKeyWorker(PKSock).start();// Starting up the worker

}

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// Public Key Worker class

class PublicKeyWorker extends Thread {

Socket PKWorkerSock;// Socket initialization

public PublicKeyWorker(Socket PKWorkerSock) {// Constructer used to assign the socket

this.PKWorkerSock = PKWorkerSock;

}

public void run() {

try {

BufferedReader input = new BufferedReader(new InputStreamReader(PKWorkerSock.getInputStream()));// getting the data from socket using BufferedReader.

String[] data = input.readLine().split(" "); // reading the data and splitting it.

int process\_id = Integer.parseInt(data[0]);

byte[] PKBytes = Base64.getDecoder().decode(data[1]); // storing the public key into the local variable.

X509EncodedKeySpec PKEncoded = new X509EncodedKeySpec(PKBytes);

KeyFactory PKfact = KeyFactory.getInstance("RSA"); // getting RSA Algorithm Instance.

PublicKey res\_key = PKfact.generatePublic(PKEncoded); // Assigning a public key variable.

blockChainWorker.pKeyArray[process\_id] = res\_key; // Adding the public key into the list.

blockChainWorker.pkCounter++; // Counter increment.

if (blockChainWorker.pkCounter == 3) // checking if got all the public keys.

blockChainWorker.publicKeyFlag = true; // chaning the publickeyflag.

System.out.println("Publickey received for Process: " + process\_id); // Confirming that the program recieved the publickey.

PKWorkerSock.close(); // ending the socket connection.

} catch (Exception ex) {// Handling Exception

ex.printStackTrace();

}

}

}

// Unverified Class verifies the blocks that are read from the input file.

// It invokes its respective worker class.

class unverifiedBlockServer implements Runnable {

BlockingQueue<blockChainRecord> blockRecordQueue;// Initializing queue to store blocks of records.

public unverifiedBlockServer(BlockingQueue<blockChainRecord> blockRecordQueue) { // Constructor which initializes the block queue.

this.blockRecordQueue = blockRecordQueue;

}

@Override

public void run() { // method which runs after receiving the blocks.

Socket UBSSock;// Socket initialization

System.out.println("Unverified Block Server started and is at port " + serverPorts.UBServerPort);// Displaying a message showing which port the Unverified Server is connected

try {

ServerSocket UBSServerSock = new ServerSocket(serverPorts.UBServerPort);

while (true) {

UBSSock = UBSServerSock.accept();// Listening for requests

new unverifiedBlockServerWorker(UBSSock).start();// Starting up the worker

}

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// UBServer Worker class

class unverifiedBlockServerWorker extends Thread {

Socket UBSWorkerSock;// Socket initialization

public unverifiedBlockServerWorker(Socket UBSWorkerSock) {// Constructer used to assign the socket

this.UBSWorkerSock = UBSWorkerSock;

}

@Override

public void run() {

try {

BufferedReader input = new BufferedReader(new InputStreamReader(UBSWorkerSock.getInputStream()));// getting the data from socket using BufferedReader.

String block\_input;// Varible initialization storing the input.

Gson gson = new Gson();// Initialising hte gson object.

StringBuffer sb = new StringBuffer(); // string buffer object.

while ((block\_input = input.readLine()) != null) { // storing the data into json format

sb.append(block\_input);

}

blockChainRecord blockreInput = gson.fromJson(sb.toString(), blockChainRecord.class);// Marshalling hte block records data

System.out.println("Added " + blockreInput.getblock\_id() + " into the Priority Blocking Queue" + "\n");// Confirming that we have updated the priority queue.

blockChainWorker.priorityQueue.put(blockreInput);

UBSWorkerSock.close(); // ending the socket connection.

} catch (Exception ex) {// Handling Exception

ex.printStackTrace();

}

}

}

/\* this is updated block server which receives the blocks after verification

Json object reads it and transmitts \*/

class updatedBlockServer implements Runnable {

@Override

public void run() {

Socket UpBSSock; // Socket initialization

System.out.println("Updated Block chain Server started and is at port: " + serverPorts.UpBServerPort);// Displaying a message showing which port the Block Chain Server is connected

try {

ServerSocket UpBServerSock = new ServerSocket(serverPorts.UpBServerPort);

while (true) {

UpBSSock = UpBServerSock.accept();// Listening for requests

new UpdatedBlockchainWorker(UpBSSock).start();// Starting up the worker

}

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// Worker class that adds the blocks into blockchain ledger.

class UpdatedBlockchainWorker extends Thread {

Socket UpBWorkerPort; // Socket initialization

public UpdatedBlockchainWorker(Socket UpBWorkerPort) {// Constructer used to assign the socket

this.UpBWorkerPort = UpBWorkerPort;

}

@Override

public void run() {

try {

BufferedReader input = new BufferedReader(new InputStreamReader(UpBWorkerPort.getInputStream()));// getting the data from socket using BufferedReader.

Gson gson = new Gson();// Initialising hte gson object.

String block\_input;// Varible initialization storing the input.

StringBuffer sb = new StringBuffer(); // string buffer object.

while ((block\_input = input.readLine()) != null) { // storing the data into json format

sb.append(block\_input);

}

blockChainRecord blockreInput = gson.fromJson(sb.toString(), blockChainRecord.class);

if (!blockChainWorker.duplicateRecordCheck(blockreInput)) { // checking if the ledger already contains the block.

blockChainWorker.verifiedBlocks.add(0, blockreInput); // adding hte block to verified blocks.

System.out.println("Added the Block into Legder"); // confirming that the program added the block into hte legder.

// System.out.println("Block Count after verifying in the Ledger is: " + blockChainWorker.verifiedBlocks.size());

}

if (blockChainWorker.process\_id == 0) // if the process is 0 then the whole blockchain ledger is loaded into disk.

blockChainWorker.writeJSONToDisk();

UpBWorkerPort.close(); // ending the socket connection.

} catch (IOException IOex) {// Handling Exception

IOex.printStackTrace();

}

}

}

// The puzsle is sovled in the work method.

class WorkPuzzle implements Runnable {

BlockingQueue<blockChainRecord> blockRecordQueue; // variable to store the blocks in the queue

private static final String alpNumChar = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"; // radnom string is created which is used to solve the puzzle.

public WorkPuzzle(PriorityBlockingQueue<blockChainRecord> blockRecordQueue) {

this.blockRecordQueue = blockRecordQueue; // assigning the block record queue.

}

@Override

public void run() {

try {

while (true) { // loop to check all the blocks

blockChainRecord blockRecord = blockChainWorker.priorityQueue.take(); // the blocks are iterated one by one.

// String variable to store the blockrecord

String blockRecordString = blockRecord.getblock\_id() + blockRecord.getfirst\_name() + blockRecord.getlast\_name() + blockRecord.getSSN() +

blockRecord.getdata\_of\_birth() + blockRecord.getdiagnosis() + blockRecord.gettreatment() + blockRecord.getPrescription() + blockRecord.getprocess(); // creating a string by appending all the blocks records.

String randSeedString; // ranodm seed string variable of string data type

String combinedString; // concatinated stirng variable of stirng data type.

String hashValue; // hash variable

boolean signedHash; // variable to check if the hash is signed.

boolean signedBlock; // variable to check if the block is signed.

if (blockChainWorker.duplicateRecordCheck(blockRecord) && blockRecord != null) // checkin if any duplicate records are present

continue;

signedBlock = blockChainWorker.verifySignature(blockRecord.getblock\_id().getBytes(),blockChainWorker.pKeyArray[Integer.parseInt(blockRecord.getprocess())],Base64.getDecoder().decode(blockRecord.getsignedBlock\_id()));

System.out.println(signedBlock ? "Block ID Signed" : "Block ID not Signed"); // comfirming that the program has signed or not signed hte ID

signedHash = blockChainWorker.verifySignature(blockRecord.getHashValue().getBytes(),

blockChainWorker.pKeyArray[Integer.parseInt(blockRecord.getprocess())],

Base64.getDecoder().decode(blockRecord.getSignedHashValue()));

System.out.println(signedHash ? "Hash is Signed" : "Hash not Signed"); // Confirming that the program has signed the hash.

String previousBlockID = blockChainWorker.verifiedBlocks.get(0).getblock\_id(); // variable to store the previous block id of type string

int hashsscount; // variable to store the count of hash substring

String blockRecordUpdate = blockRecordString; // variblae to store th solved blocks

blockRecordUpdate = blockRecordUpdate + blockChainWorker.verifiedBlocks.get(0).gethashOfWinningBlock(); // the updated variable is appended with the winning hash of previous block

if (!blockChainWorker.duplicateRecordCheck(blockRecord)) { // condition to check if the block is present in the ledger and slove the puzzle

try {

for (int count = 1; count < 20; count++) {

randSeedString = alphaNumericGenerator(8); // get a random number

combinedString = blockRecordUpdate + randSeedString; // making a string with updated blocks n=adn random number

MessageDigest MD = MessageDigest.getInstance("SHA-256"); // getting hte hash value

byte[] bytecomString = MD.digest(combinedString.getBytes(StandardCharsets.UTF\_8));

hashValue = byteArray2Str(bytecomString); // converting the hexa decimal hashsed value to string

System.out.println("Hash Value for Block Record is: " + hashValue);

hashsscount = Integer.parseInt(hashValue.substring(0, 4), 16); // we are taking the first 16 bits of the hash as a substring value and compare

System.out.println("16 bits of the hash in Hex: " + hashValue.substring(0, 4));

System.out.println("16 bits of the hash in Decimal: " + hashsscount);

if (!(hashsscount < 20000)) // checkinig if the hash count is less than 20000 if not we solve puzzle again.

System.out.format("Hash Count is greater than 20000. Need to solve the puzzle again\n\n", hashsscount);

if (hashsscount < 20000) { // working on the puzzle

if (!previousBlockID.equals(blockChainWorker.verifiedBlocks.get(0).getblock\_id()))

blockChainWorker.buildLedgerJson(blockRecord, "verify"); // getting teh data of the block

else { // the ledger is updated if it is not updated

blockRecord.sethashOfWinningBlock(hashValue); // setting the winning hash.

blockRecord.setrandomSeed(randSeedString); // setting the random seed string.

System.out.format("Hash count is less than 20000. Solved the puzzle.\n", hashsscount);

blockRecord.sethashOfPreviousBlock(blockChainWorker.verifiedBlocks.get(0).gethashOfWinningBlock()); // updating the winning hash of the block

int previousBlocknum = Integer.parseInt(blockChainWorker.verifiedBlocks.get(0).getblock\_num()); // assigning a variable to get the previous block nuumber

blockRecord.setblock\_num(String.valueOf(++previousBlocknum)); // updating the block number.

blockRecord.setprocessIdVerify(String.valueOf(blockChainWorker.process\_id)); // updating the process id .

String hashSignVerify; // variable to store the signed hash of string data type

byte[] digiSign = blockChainWorker.signData(hashValue.getBytes(),blockChainWorker.processKeyPair.getPrivate()); // getting the digital signature.

hashSignVerify = Base64.getEncoder().encodeToString(digiSign); // updating the signed hash value/

blockRecord.setsignedHashOfWinningBlock(hashSignVerify); // updating the value of signed hash data.

blockChainWorker.verifiedBlocks.add(0, blockRecord); // updating the ledger with the signed hash value.

System.out.println("Updated the Ledger..");

blockChainWorker.buildLedgerJson(blockRecord, "update");

continue;

}

break;

}

if (blockChainWorker.duplicateRecordCheck(blockRecord)) // if the block record is already verified we skip

break;

blockChainWorker.Sleep();

}

} catch (Exception ex) { // Handling the excepteion

ex.printStackTrace();

}

}

}

} catch (Exception ex) { // Handling the excepteion

ex.printStackTrace();

}

}

/\* Method to get ranodm seed value using aplha numeric characters

I have used hte code provided by the professor to sign data.\*/

public static String alphaNumericGenerator(int counter) {

StringBuilder sb = new StringBuilder();

while (counter-- != 0) {

int ch = (int) (Math.random() \* alpNumChar.length());

sb.append(alpNumChar.charAt(ch));

}

return sb.toString();

}

// Method to convert the byter array to Stirng

public static String byteArray2Str(byte[] barray) {

StringBuilder hexStrg = new StringBuilder(barray.length \* 2);

for (byte bytes : barray) {

hexStrg.append(String.format("%02X", bytes));

}

return hexStrg.toString();

}

}