/\*

Version 1.0 2017-09-03

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 is pedagogical code and should not be considered current for secure applications.

The web sources:

http://www.java2s.com/Code/Java/Security/SignatureSignAndVerify.htm

https://www.mkyong.com/java/java-digital-signatures-example/ (not so clear)

https://javadigest.wordpress.com/2012/08/26/rsa-encryption-example/

https://www.programcreek.com/java-api-examples/index.php?api=java.security.SecureRandom

https://www.mkyong.com/java/java-sha-hashing-example/

https://stackoverflow.com/questions/19818550/java-retrieve-the-actual-value-of-the-public-key-from-the-keypair-object

XML validator:

https://www.w3schools.com/xml/xml\_validator.asp

XML / Object conversion:

https://www.mkyong.com/java/jaxb-hello-world-example/

\*/

/\* CDE: The JAXB libraries: \*/

import javax.xml.bind.JAXBContext;

import javax.xml.bind.JAXBException;

import javax.xml.bind.Marshaller;

import javax.xml.bind.Unmarshaller;

import javax.xml.bind.annotation.XmlAttribute;

import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlRootElement;

import java.io.StringWriter;

import java.io.StringReader;

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

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.SecureRandom;

import java.security.Security;

import java.security.Signature;

import java.security.NoSuchAlgorithmException;

import java.security.spec.PKCS8EncodedKeySpec;

import javax.crypto.Cipher;

/\* CDE Some other uitilities: \*/

import java.util.Date;

import java.util.Random;

import java.util.UUID;

import java.text.\*;

import java.util.Base64;

import java.util.Arrays;

// Produces a 64-bye string representing 256 bits of the hash output. 4 bits per character

import java.security.MessageDigest; // To produce the SHA-256 hash.

@XmlRootElement

class BlockRecord{

/\* Examples of block fields: \*/

String VerificationProcessID;

String PreviousHash;

String BlockID;

String Fname;

String Lname;

String SSNum;

String DOB;

/\* Examples of accessors for the BlockRecord fields: \*/

public String getSSNum() {return SSNum;}

@XmlElement

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

public String getFname() {return Fname;}

@XmlElement

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

public String getLname() {return Lname;}

@XmlElement

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

public String getVerificationProcessID() {return VerificationProcessID;}

@XmlElement

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

public String getBlockID() {return BlockID;}

@XmlElement

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

}

/\* Starting point for the BlockRecord:

<?xml version="1.0" encoding="UTF-8"?>

<BlockRecord>

<SIGNED-SHA256> [B@5f150435 </SIGNED-SHA256> <!-- Verification procees SignedSHA-256-String -->

<SHA-256-String> 63b95d9c17799463acb7d37c85f255a511f23d7588d871375d0119ba4a96a </SHA-256-String>

<!-- Start SHA-256 Data that was hashed -->

<VerificationProcessID> 1 </VerificationProcessID> <!-- Process that is verifying this block, for credit-->

<PreviousHash> From the previous block in the chain </PreviousHash>

<Seed> Your random 256 bit string </Seed> <!-- guess the value to complete the work-->

<BlockNum> 1 </BlockNum> <!-- increment with each block prepended -->

<BlockID> UUID </BlockID> <!-- Unique identifier for this block -->

<SignedBlockID> BlockID signed by creating process </SignedBlockID> <!-- Creating process signature -->

<CreatingProcessID> 0 </CreatingProcessID> <!-- Process that made the ledger entry -->

<TimeStamp> 2017-09-01.10:26:35 </TimeStamp>

<DataHash> The creating process SHA-256 hash of the input data </DataHash> <!-- for auditing if Secret Key exposed -->

<FName> Joseph </FName>

<LName> Ng </LName>

<DOB> 1995.06.22 </DOB> <!-- date of birth -->

<SSNUM> 987-65-4321 </SSNUM>

<Diagnosis> Measels </Diagnosis>

<Treatment> Bedrest </Treatment>

<Rx> aspirin </Rx>

<Notes> Use for debugging and extension </Notes>

<!-- End SHA-256 Data that was hashed -->

</BlockRecord>

\*/

public class BlockH {

public static byte[] signData(byte[] data, PrivateKey key) throws Exception {

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

signer.initSign(key);

signer.update(data);

return (signer.sign());

}

public static boolean verifySig(byte[] data, PublicKey key, byte[] sig) throws Exception {

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

signer.initVerify(key);

signer.update(data);

return (signer.verify(sig));

}

public static KeyPair generateKeyPair(long seed) throws Exception {

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

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

rng.setSeed(seed);

keyGenerator.initialize(1024, rng);

return (keyGenerator.generateKeyPair());

}

public static String CSC435Block =

"We will build this dynamically: <?xml version = \"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>";

public static final String ALGORITHM = "RSA"; /\* Name of encryption algorithm used \*/

/\* Header fields for the block: \*/

public static String SignedSHA256;

/\* CDE NOTE: we do not need this method for the CSC435 blockchain assignment. \*/

public static byte[] encrypt(String text, PublicKey key) {

byte[] cipherText = null;

try {

final Cipher cipher = Cipher.getInstance(ALGORITHM); // Get RSA cipher object

cipher.init(Cipher.ENCRYPT\_MODE, key);

cipherText = cipher.doFinal(text.getBytes());

} catch (Exception e) {

e.printStackTrace();

}

return cipherText;

}

/\* CDE NOTE: we do not need this method for the CSC435 blockchain assignment. \*/

public static String decrypt(byte[] text, PrivateKey key) {

byte[] decryptedText = null;

try {

final Cipher cipher = Cipher.getInstance(ALGORITHM);

cipher.init(Cipher.DECRYPT\_MODE, key);

decryptedText = cipher.doFinal(text);

} catch (Exception ex) {

ex.printStackTrace();

}

return new String(decryptedText);

}

public static void main(String[] args) throws Exception {

/\* CDE: Process numbers and port numbers to be used: \*/

int pnum;

int UnverifiedBlockPort;

int BlockChainPort;

/\* CDE If you want to trigger bragging rights functionality... \*/

if (args.length > 1) System.out.println("Special functionality is present \n");

if (args.length < 1) pnum = 0;

else if (args[0].equals("0")) pnum = 0;

else if (args[0].equals("1")) pnum = 1;

else if (args[0].equals("2")) pnum = 2;

else pnum = 0; /\* Default for badly formed argument \*/

UnverifiedBlockPort = 4710 + pnum;

BlockChainPort = 4810 + pnum;

System.out.println("Process number: " + pnum + " Ports: " + UnverifiedBlockPort + " " +

BlockChainPort + "\n");

/\* CDE: Example of generating a unique blockID. This would also be signed by creating process: \*/

UUID idA = UUID.randomUUID();

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

System.out.println("Unique Block ID: " + suuid + "\n");

/\* CDE For the timestamp in the block entry: \*/

Date date = new Date();

//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 + "." + pnum + "\n"; // No timestamp collisions!

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

/\* CDE: Here is a way for us to simulate computational "work" \*/

System.out.println("How much work we did: ");

int randval;

Random r = new Random();

for (int i=0; i<1000; i++){ // safety upper limit of 1000

Thread.sleep(100); // not really work, but OK for our purposes.

randval = r.nextInt(100); // Higher val = more work

if (randval < 4) { // Lower threshold = more work

System.out.println(i + " tenths of a second.\n");

break;

}

}

try {

/\* CDE put some data into the block record: \*/

BlockRecord blockRecord = new BlockRecord();

blockRecord.setVerificationProcessID("Process2");

blockRecord.setBlockID(suuid);

blockRecord.setSSNum("123-45-6789");

blockRecord.setFname("Joseph");

blockRecord.setLname("Chang");

/\* The XML conversion tools: \*/

JAXBContext jaxbContext = JAXBContext.newInstance(BlockRecord.class);

Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

StringWriter sw = new StringWriter();

// CDE Make the output pretty printed:

jaxbMarshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

/\* CDE We marshal the block object into an XML string so it can be sent over the network: \*/

jaxbMarshaller.marshal(blockRecord, sw);

String stringXML = sw.toString();

CSC435Block = stringXML;

/\* Make the SHA-256 Digest of the block: \*/

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

md.update (CSC435Block.getBytes());

byte byteData[] = md.digest();

// CDE: Convert the byte[] to hex format. THIS IS NOT VERFIED CODE:

StringBuffer sb = new StringBuffer();

for (int i = 0; i < byteData.length; i++) {

sb.append(Integer.toString((byteData[i] & 0xff) + 0x100, 16).substring(1));

}

String SHA256String = sb.toString();

KeyPair keyPair = generateKeyPair(999);

byte[] digitalSignature = signData(SHA256String.getBytes(), keyPair.getPrivate());

boolean verified = verifySig(SHA256String.getBytes(), keyPair.getPublic(), digitalSignature);

System.out.println("Has the signature been verified: " + verified + "\n");

System.out.println("Original SHA256 Hash: " + SHA256String + "\n");

/\* Add the SHA256String to the header for the block. We turn the

byte[] signature into a string so that it can be placed into

the block, but also show how to return the string to a

byte[], which you'll need if you want to use it later.

Thanks Hugh Thomas for the fix! \*/

SignedSHA256 = Base64.getEncoder().encodeToString(digitalSignature);

System.out.println("The signed SHA-256 string: " + SignedSHA256 + "\n");

byte[] testSignature = Base64.getDecoder().decode(SignedSHA256);

System.out.println("Testing restore of signature: " + Arrays.equals(testSignature, digitalSignature));

verified = verifySig(SHA256String.getBytes(), keyPair.getPublic(), testSignature);

System.out.println("Has the restored signature been verified: " + verified + "\n");

String fullBlock = stringXML.substring(0,stringXML.indexOf("<blockID>")) +

"<SignedSHA256>" + SignedSHA256 + "</SignedSHA256>\n" +

" <SHA256String>" + SHA256String + "</SHA256String>\n " +

stringXML.substring(stringXML.indexOf("<blockID>"));

System.out.println(fullBlock); // Show what it looks like.

/\* CDE Here's how we put the XML back into java object form: \*/

Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();

StringReader reader = new StringReader(stringXML);

BlockRecord blockRecord2 = (BlockRecord) jaxbUnmarshaller.unmarshal(reader);

System.out.println("SSNum: " + blockRecord2.getSSNum()); // Show a piece of the new block object

/\* CDE: In case you want to use it for something, here we encrypt a

string, then decrypt it, using the same public key technology. These

techniques are not needed for the basic CSC435 assignment. Note that this

methocd is intended for 117 bytes or less to pass session keys: \*/

/\* CDE: Encrypt the hash string using the public key. \*/

final byte[] cipherText = encrypt(SHA256String,keyPair.getPublic());

// CDE: Decrypt the ciphertext using the private key:

final String plainText = decrypt(cipherText, keyPair.getPrivate());

System.out.println("\nExtra functionality in case you want it:");

System.out.println("Encrypted Hash string: " + Base64.getEncoder().encodeToString(cipherText));

System.out.println("Original (now decrypted) Hash string: " + plainText);

} catch (Exception e) {

e.printStackTrace();

}

}

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

Name: Tommy Leedberg

Date: October 10, 2018

Java Version: 1.8.0\_181

Command-Line Examples:

Usage: java BlockChain [ProcessNumber]

Instructions:

To Compile:

javac -cp "gson-2.8.4.jar" \*.java

Notes:

An additional port is being used by process 2, this port

is 1524 and it is the port that the KeyManager listens on

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

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import com.google.gson.annotations.SerializedName;

import com.google.gson.reflect.TypeToken;

import javax.xml.bind.DatatypeConverter;

import java.io.\*;

import java.lang.reflect.Type;

import java.net.ServerSocket;

import java.net.Socket;

import java.security.\*;

import java.util.\*;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.PriorityBlockingQueue;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

/\*\*

\* The KeyManager class for generating the public/private key pair and also

\* signing and verifying the blocks of data

\*/

class KeyManager

{

private KeyPair keyPair = null;

private PublicKey publicKey = null;

private PrivateKey privateKey = null;

private Signature signer = null;

/\*\*

\* The KeyManager class used to generate, sign, and verify keys

\*/

public KeyManager(PublicKey publicKey)

{

String signatureAlgorithm = "SHA1withRSA";

try

{

this.signer = Signature.getInstance(signatureAlgorithm);

}

catch (NoSuchAlgorithmException ex)

{

BlockChain.PrintError("Invalid encryption algorithm supplied for signature", ex);

}

// This constructor is for services that will only be using the public key

this.publicKey = publicKey;

}

/\*\*

\* The KeyManager class used to generate, sign, and verify keys

\*/

public KeyManager()

{

String signatureAlgorithm = "SHA1withRSA";

try

{

// A seed was provided so this call is going to create the kv pair and

// keep a copy of the private key

this.signer = Signature.getInstance(signatureAlgorithm);

}

catch (NoSuchAlgorithmException ex)

{

BlockChain.PrintError("Invalid encryption algorithm supplied for key pair generation", ex);

}

}

/\*\*

\* Generate the public/private key pair to be used with signing/verification

\*

\* @param randomSeed

\*/

public void GenerateKeyPair(long randomSeed)

{

String encryptionAlgorithm = "RSA";

String hashingAlgorithm = "SHA1PRNG";

String hashAlgorithmProvider = "SUN";

// NOTE: this method should only be called by the last process in this project

// as I only want to generate a key pair once then the public key will be shared

// and identical for all processes

try

{

// A seed was provided so this call is going to create the kv pair and

// keep a copy of the private key

KeyPairGenerator keyGenerator = KeyPairGenerator.getInstance(encryptionAlgorithm);

SecureRandom rng = SecureRandom.getInstance(hashingAlgorithm, hashAlgorithmProvider);

rng.setSeed(randomSeed);

keyGenerator.initialize(1024, rng);

this.keyPair = keyGenerator.generateKeyPair();

this.publicKey = keyPair.getPublic();

this.privateKey = keyPair.getPrivate();

}

catch (NoSuchAlgorithmException ex)

{

BlockChain.PrintError("Invalid encryption algorithm supplied for keypair generation", ex);

}

catch (NoSuchProviderException ex)

{

BlockChain.PrintError("Invalid hash algorithm provider supplied for keypair generation", ex);

}

}

/\*\*

\* Get the public key for the Key pair

\*

\* @return the public key of the key pair

\*/

public PublicKey GetPublicKey()

{

return this.publicKey;

}

/\*\*

\* Sign the data with the private key

\*

\* @param unsignedData The data to be signed

\* @return the signed data or null if an exception occured

\*/

public byte[] SignData(byte[] unsignedData)

{

try

{

this.signer.initSign(this.privateKey);

this.signer.update(unsignedData);

return this.signer.sign();

}

catch (SignatureException ex)

{

BlockChain.PrintError("Signature error while trying to sign the data block", ex);

return null;

}

catch (InvalidKeyException ex)

{

BlockChain.PrintError("Invalid key exception while trying to sign the data block", ex);

return null;

}

}

/\*\*

\* Validate that the signed data matches the unsigned data once using

\* the public key

\*

\* @param unsignedData The unsigned data to use for verification

\* @param signedData The data that has been signed with the private key

\* @return A value indicating whether or not the data has been signed by

\* the private key

\*/

public boolean VerifySignature(byte[] unsignedData, byte[] signedData)

{

try

{

this.signer.initVerify(this.publicKey);

this.signer.update(unsignedData);

return this.signer.verify(signedData);

}

catch (SignatureException ex)

{

BlockChain.PrintError("Signature error while trying to verify the data block", ex);

return false;

}

catch (InvalidKeyException ex)

{

BlockChain.PrintError("Invalid key exception while trying to verify the data block", ex);

return false;

}

}

}

/\*\*

\* Class representing the ports used for each process

\*/

class Ports

{

public static final int KeyManagerPort = 1524;

private static int KeyServerPortBase = 4710;

private static int UnverifiedBlockServerPortBase = 4820;

private static int BlockChainServerPortBase = 4930;

private static int[] KeyServerPortsInUse = new int[BlockChain.ProcessCount];

private static int[] UnverifiedBlockServerPortsInUse = new int[BlockChain.ProcessCount];

private static int[] BlockChainServerPortsInUse = new int[BlockChain.ProcessCount];

private static int KeyServerPort;

private static int UnverifiedBlockServerPort;

private static int BlockChainServerPort;

/\*\*

\* The ports constructor

\*

\* @param runningProcessCount the number of running processes

\*/

public static void setPortsForAllProcesses(int runningProcessCount)

{

// this is helpful because every process will know all running ports that are in use by the service

for (int i = 0; i < runningProcessCount; ++i)

{

KeyServerPortsInUse[i] = KeyServerPortBase + i;

UnverifiedBlockServerPortsInUse[i] = UnverifiedBlockServerPortBase + i;

BlockChainServerPortsInUse[i] = BlockChainServerPortBase + i;

}

}

/\*\*

\* Set the ports for each of the process types by incrementing the processId to the base value

\*

\* @param processId the process id for this process

\*/

public static void setPortsForCurrentProcess(int processId)

{

KeyServerPort = KeyServerPortBase + processId;

UnverifiedBlockServerPort = UnverifiedBlockServerPortBase + processId;

BlockChainServerPort = BlockChainServerPortBase + processId;

}

/\*\*

\* Get the Key Server Port for this process

\*

\* @return The Key Server Port for the current process

\*/

public static int getKeyServerPort()

{

return KeyServerPort;

}

/\*\*

\* Get the current processes unverified block server port

\*

\* @return The current processes unverified block server port

\*/

public static int getUnverifiedBlockServerPort()

{

return UnverifiedBlockServerPort;

}

/\*\*

\* Get the current processes block chain server port

\*

\* @returnThe current processes block chain server port

\*/

public static int getBlockChainServerPort()

{

return BlockChainServerPort;

}

/\*\*

\* Gets a list of the Key Server Ports that are in use

\*

\* @return A list of the key server ports in use

\*/

public static int[] getKeyServerPortsInUse()

{

return KeyServerPortsInUse;

}

/\*\*

\* Gets a list of the Unverified Block Server Ports that are in use

\*

\* @return A list of the unverified block ports in use

\*/

public static int[] getUnverifiedBlockServerPortsInUse()

{

return UnverifiedBlockServerPortsInUse;

}

/\*\*

\* Gets a list of the Block Chain Ports that are in use

\*

\* @return A list of the block chain ports in use

\*/

public static int[] getBlockChainServerPortsInUse()

{

return BlockChainServerPortsInUse;

}

}

class DataBlock

{

@SerializedName (value = "FirstName")

private String FirstName = "";

@SerializedName (value = "LastName")

private String LastName = "";

@SerializedName (value = "SSN", alternate = {"SocialSecurityNumber"})

private String SocialSecurityNumber = "";

@SerializedName (value = "DOB", alternate = {"DateOfBirth"})

private String DateOfBirth = "";

@SerializedName (value = "Diagnosis")

private String Diagnosis = "";

@SerializedName (value = "Treatment")

private String Treatment = "";

@SerializedName (value = "Medication")

private String Medication = "";

/\*\*

\* SEt the social security number for the person in this record block

\*

\* @socialSecurityNumber The social security number to set

\*/

public void setSocialSecurityNumber(String socialSecurityNumber)

{

this.SocialSecurityNumber = socialSecurityNumber;

}

/\*\*

\* Get the first name for the person in this record block

\*

\* @return The first name for the person in this record block

\*/

public String getFirstName()

{

return this.FirstName;

}

/\*\*

\* Set first name for the person in this record block

\*

\* @firstName The first name for the person in this record block

\*/

public void setFirstName(String firstName)

{

this.FirstName = firstName;

}

/\*\*

\* Get the last name for the person in this record block

\*

\* @return The last name for the person in this record block

\*/

public String getLastName()

{

return LastName;

}

/\*\*

\* Set the last name for the person in this record block

\*

\* @lastName The last name for the person in this record block

\*/

public void setLastName(String lastName)

{

this.LastName = lastName;

}

/\*\*

\* Set the date of birth for the person in this record block

\*

\* @dateOfBirth The date of birth for the person in this record block

\*/

public void setDateOfBirth(String dateOfBirth)

{

this.DateOfBirth = dateOfBirth;

}

/\*\*

\* Set the diagnosis for the person in this record block

\*

\* @diagnosis The diagnosis for the person in this record block

\*/

public void setDiagnosis(String diagnosis)

{

this.Diagnosis = diagnosis;

}

/\*\*

\* Set the treatment for the person in this record block

\*

\* @treatment The treatment for the person in this record block

\*/

public void setTreatment(String treatment)

{

this.Treatment = treatment;

}

/\*\*

\* Set the medication for the person in this record block

\*

\* @medication The medication for the person in this record block

\*/

public void setMedication(String medication)

{

this.Medication = medication;

}

}

/\*\*

\* BlockChain Block object with json serialization

\*/

class BlockRecord implements Comparable<BlockRecord>, Comparator<BlockRecord>

{

@SerializedName (value = "DataBlock")

private DataBlock DataBlock = new DataBlock();

@SerializedName( value = "BlockNumber")

private int BlockNumber = 0;

@SerializedName (value = "SHA256HashedDataBlock")

private String SHA256HashedDataBlock = "";

@SerializedName (value = "SignedSHA256DataBlock")

private String SignedSHA256DataBlock = "";

@SerializedName (value = "CreationTime")

private Date CreationTime = new Date();

@SerializedName (value = "BlockId")

private String BlockId = "";

@SerializedName (value = "SignedBlockId")

private String SignedBlockId = "";

@SerializedName (value = "VerificationProcessId")

private String VerificationProcessId = "";

@SerializedName (value = "CreatingProcess")

private String CreatingProcess = "";

@SerializedName (value = "PreviousHash")

private String PreviousHash = "";

@SerializedName( value = "Seed")

private String Seed = "";

public DataBlock getDataBlock()

{

return this.DataBlock;

}

/\*\*

\* Get the current block number

\* @return The current block number

\*/

public int getBlockNumber()

{

return BlockNumber;

}

/\*\*

\* Set the current block number

\* @param blockNumber The current block number

\*/

public void setBlockNumber(int blockNumber)

{

BlockNumber = blockNumber;

}

/\*\*

\* Get the SHA256Sting

\*

\* @return

\*/

public String getSHA256HashedDataBlock()

{

return SHA256HashedDataBlock;

}

/\*\*

\* Set teh SHA256 String

\*

\* @param sha256String the SHA256HashedDataBlock to set

\*/

public void setSHA256HashedDataBlock(String sha256String)

{

this.SHA256HashedDataBlock = sha256String;

}

/\*\*

\* Get the signed SHA256 string

\*

\* @return The Signed version of the SHA256 string

\*/

public String getSignedSHA256DataBlock()

{

return SignedSHA256DataBlock;

}

/\*\*

\* Set the signed version of the SHA256 signed string

\*

\* @param sha256String the signed SHA256 string

\*/

public void setSignedSHA256DataBlock(String sha256String)

{

this.SignedSHA256DataBlock = sha256String;

}

/\*\*

\* Get the signed previous hash

\*

\* @return The signedPreviousHash

\*/

public String getPreviousHash()

{

return this.PreviousHash;

}

/\*\*

\* Set the signed previous Hash

\*

\* @param previousHash the signed previous hash

\*/

public void setPreviousHash(String previousHash)

{

this.PreviousHash = previousHash;

}

/\*\*

\* Get the random seed value from the block

\* @return An int representing the random seed value

\*/

public String getSeed()

{

return this.Seed;

}

/\*\*

\* Set the Seed value (the value that sovled the puzzle ) for this block

\* @param seed The new seed value

\*/

public void setSeed( String seed)

{

this.Seed = seed;

}

/\*\*

\* Set the process that created this blocks id

\*

\* @param creatingProcess creating processes id

\*/

public void setCreatingProcess(String creatingProcess)

{

this.CreatingProcess = creatingProcess;

}

/\*\*

\* Set teh verifying process's id

\*

\* @param verificationProcessId The id of the verifying process

\*/

public void setVerificationProcessID(String verificationProcessId)

{

this.VerificationProcessId = verificationProcessId;

}

/\*\*

\* Get the block id for this record block

\*

\* @return The block id for this record block

\*/

public String getBlockId()

{

return this.BlockId;

}

/\*\*

\* Set the block Id for this record block

\*

\* @param blockId the block Id to set

\*/

public void setBlockId(String blockId)

{

this.BlockId = blockId;

}

/\*\*

\* Get the signed blockId for this block

\*

\* @return The signed blockId for this block

\*/

public String getSignedBlockId()

{

return this.SignedBlockId;

}

/\*\*

\* Set the signed blockId for this record block

\*

\* @param SignedBlockId The signed block Id for this record block

\*/

public void setSignedBlockId(String SignedBlockId)

{

this.SignedBlockId = SignedBlockId;

}

/\*\*

\* Comparision override for the block record. Compares the creation date's

\*

\* @param other the BlockRecord to compare against

\* @return An int representing the comparison evaluation

\*/

@Override

public int compareTo(BlockRecord other)

{

return this.CreationTime.compareTo(other.CreationTime);

}

@Override

public int compare(BlockRecord a, BlockRecord b)

{

return a.BlockNumber - b.BlockNumber;

}

}

/\*\*

\* A singleton for the Utilities class

\*/

class Utilities

{

private static KeyManager KeyManager = null;

public static KeyManager GetKeyManager()

{

return KeyManager;

}

public static void SetKeyManager(KeyManager keyManager)

{

KeyManager = keyManager;

}

/\*\*

\* Read in the input file and deserialize it into a block record

\*

\* @param fileName the file path to read in

\* @param processId The current processes process Id

\* @return a list of all records in the file

\*/

public static ArrayList<BlockRecord> ReadInputFile(String fileName, int processId)

{

ArrayList<BlockRecord> blockRecords = new ArrayList<BlockRecord>();

String currentPID = Integer.toString(processId);

// read through the input file setting each of the BlockRecord fields

try (BufferedReader br = new BufferedReader(new FileReader(fileName)))

{

String inputLine;

while ((inputLine = br.readLine()) != null)

{

BlockRecord record = new BlockRecord();

/\*\* Header information for the block \*\*/

record.setBlockId(new String(UUID.randomUUID().toString()));

record.setCreatingProcess(currentPID);

// split the input line by 1 or more spaces into a delimited array

String[] inputData = inputLine.split(" +");

/\*\* Patient information for the block \*\*/

// every input file has the same ordering on the data points so we can hard code their index's

record.getDataBlock().setFirstName(inputData[0]);

record.getDataBlock().setLastName(inputData[1]);

record.getDataBlock().setDateOfBirth(inputData[2]);

record.getDataBlock().setSocialSecurityNumber(inputData[3]);

record.getDataBlock().setDiagnosis(inputData[4]);

record.getDataBlock().setTreatment(inputData[5]);

record.getDataBlock().setMedication(inputData[6]);

blockRecords.add(record);

}

}

catch (Exception ex)

{

BlockChain.PrintError("Error while reading input text file", ex);

}

// Print out all the names from the records collected

System.out.println(blockRecords.size() + " records read.");

System.out.println("Names from input:");

for (BlockRecord record : blockRecords)

{

System.out.println("\t" + record.getDataBlock().getFirstName() + " " + record.getDataBlock().getLastName());

}

System.out.println("\n");

return blockRecords;

}

/\*\*

\* Serialize a list of block records

\*

\* @param blockRecords The list of block records to serialize

\*/

public static String SerializeRecord(ArrayList<BlockRecord> blockRecords)

{

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

return gson.toJson(blockRecords);

}

/\*\*

\* Serialize a DataBlock

\*

\* @param dataBlock The list of block records to serialize

\*/

public static String SerializeDataBlock(DataBlock dataBlock)

{

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

return gson.toJson(dataBlock);

}

/\*\*

\* Serialize a single block record

\* <p>

\* \* @param blockRecord The block record to serialize

\*/

public static String SerializeRecord(BlockRecord blockRecord)

{

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

return gson.toJson(blockRecord);

}

public static BlockRecord DeserializeRecord(String recordString)

{

return new Gson().fromJson(recordString, BlockRecord.class);

}

/\*\*

\* Deserialize the BlockRecord Ledger

\*

\* @return A deserialized block ledger

\* @param: a serialized block ledger

\*/

public static ArrayList<BlockRecord> DeserializeLedger(String ledgerString)

{

Type listType = new TypeToken<ArrayList<BlockRecord>>(){}.getType();

return new Gson().fromJson(ledgerString, listType);

}

/\*\*

\* Forward the unverified block to the keymanager or create the block and then send it if it's the dummy block

\*

\* @param unverifiedBlock An unverified block

\*/

public static void SendUnverifiedBlocks(BlockRecord unverifiedBlock)

{

try

{

Socket socket = null;

PrintStream out = null;

try

{

// create a socket to the key manager for sending the block to be signed

socket = new Socket(BlockChain.ServerName, Ports.KeyManagerPort);

out = new PrintStream(socket.getOutputStream());

// Special condition for startup so all processes have a dummy block

if (unverifiedBlock == null)

{

BlockRecord record = new BlockRecord();

/\*\* Header information for the block \*\*/

record.setBlockId("0");

record.setCreatingProcess(Integer.toString(BlockChain.PID));

System.out.println("Sending Unverified Block to be Signed");

//Send the record to the key manager to be signed

out.println(SerializeRecord(record));

out.flush();

}

else

{

//Send the record to the key manager to be signed

out.println(SerializeRecord(unverifiedBlock));

out.flush();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Error sending unverified block to key manager to be signed", ex);

}

finally

{

out.close();

socket.close();

}

}

catch (Exception ex)

{

BlockChain.PrintError("Error while sending unverified block", ex);

}

}

/\*\*

\* Send the public key to the Public Key Server port for each process

\*

\* @param keyServerPorts The ports of the key servers

\*/

public static void SendKeys(int[] keyServerPorts)

{

Socket socket;

ObjectOutputStream toServer;

// If we're going to send the keys() then send them to every process and then return

for (int i = 0; i < keyServerPorts.length; i++)

{

// wrapping the try catch inside the for loop will stop the process from failing completely if it cant

// send to a single process for some reason

try

{

// Send the public key to all of the running processes key server ports

socket = new Socket(BlockChain.ServerName, keyServerPorts[i]);

toServer = new ObjectOutputStream(socket.getOutputStream());

System.out.println("Sending public key to process " + i);

toServer.writeObject(KeyManager.GetPublicKey());

toServer.flush();

toServer.close();

socket.close();

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to send public keys to process " + i, ex);

return;

}

}

}

/\*\*

\* Using the given stringToHash generate a new hash string

\* @param stringToHash The string to get a Hash From

\* @return the Hash of the stringToHash

\*/

public static byte[] GetHash( String stringToHash )

{

try

{

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

// Get the hash value

return MD.digest(stringToHash.getBytes("UTF-8"));

}

catch (Exception ex)

{

BlockChain.PrintError("Failed to get hash for string" + stringToHash);

return null;

}

}

}

class UnverifiedBlockConsumer implements Runnable

{

public void run()

{

BlockRecord record;

BlockChain.PrintInformation("Starting the Unverified Block Consumer thread.");

try

{

while (true)

{

boolean blockExists = false;

// Consume from the incoming queue. Do the work to verify. Multi-cast new blockchain

record = BlockChain.Queue.take();

BlockChain.PrintInformation("Unverified block consumer got a new unverified block: " + Utilities.SerializeRecord(record));

for (BlockRecord ledgerRecord : BlockChain.BlockLedger)

{

//If our current ledger already contains a block with this block id that means it's been solved so we dont have to solve it

if (ledgerRecord.getBlockId().compareToIgnoreCase(record.getBlockId()) == 0)

{

blockExists = true;

break;

}

}

if (!isValidBlock(record))

{

continue;

}

try

{

// Set the new block number

int currentBlockNum = BlockChain.BlockLedger.size() + 1;

record.setBlockNumber(currentBlockNum);

// Set the verifiers process id

record.setVerificationProcessID(Integer.toString(BlockChain.ProcessId));

String previousHash;

// If this is the dummy block there isnt anything in the ledger yet so create the previousHash from the current block

if (!record.getBlockId().equals("0") && BlockChain.BlockLedger.size() != 0)

{

int previousBlockNum = BlockChain.BlockLedger.size() - 1;

//Need to validate that this will produce the previous hash

previousHash = BlockChain.BlockLedger.get(previousBlockNum).getPreviousHash();

}

else

{

previousHash = DatatypeConverter.printHexBinary(Utilities.GetHash(record.getSHA256HashedDataBlock()));

}

for (int i = 1; i < 200; i++)

{

// Generate an alpha numeric string and append it to the current serialized block of data

String randString = this.randomAlphaNumeric(10);

String concatString = previousHash + randString;

//Generate the new Hash

String newHash = DatatypeConverter.printHexBinary(Utilities.GetHash(concatString)); // Turn into a string of hex values

if (this.IsValidAnswer(newHash))

{

// The puzzle was solved so update the seed with the answer to the puzzle

record.setSeed(randString);

// Update the previous hash value with the new hash containing the previoius hash and the current blocks

record.setPreviousHash(concatString);

break;

}

// Make sure the block hasn't already been solved while doing work.

for (BlockRecord ledgerRecord : BlockChain.BlockLedger)

{

//If our current ledger already contains a block with this block id that means it's been solved so we dont have to solve it

if (ledgerRecord.getBlockId().compareToIgnoreCase(record.getBlockId()) == 0)

{

BlockChain.PrintInformation("Block already verified so wait for next block");

blockExists = true;

break;

}

}

// There is no reason to continue doing work because the block has been added to the ledger

if (blockExists)

{

break;

}

}

}

catch (Exception ex)

{

BlockChain.PrintError("Error occurred while doing 'work'", ex);

}

if (!blockExists)

{

for (BlockRecord ledgerRecord : BlockChain.BlockLedger)

{

//If our current ledger already contains a block with this block id that means it's been solved so we dont have to solve it

if (ledgerRecord.getBlockId().compareToIgnoreCase(record.getBlockId()) == 0)

{

BlockChain.PrintInformation("Block already verified so wait for next block");

blockExists = true;

break;

}

}

// We made it this far without the puzzle being solved by another process send the new record to be validated and added to the ledger

if (!blockExists)

{

BlockChain.BlockLedger.add(record);

this.SendVerifiedBlock();

}

}

}

}

catch (Exception ex)

{

BlockChain.PrintError("Error while verifying block", ex);

}

}

/\*\*

\* Validate that the "answer" to the puzzle

\* @param answer A string representing our answer

\* @return A value indicating if this is the answer to the puzzle

\*/

private boolean IsValidAnswer(String answer)

{

try

{

// Collect only the first 16 bits from the new hash and get the Base 16 representation of it converted into an int

int workNumber = Integer.parseInt(answer.substring(0, 4), 16);

// if the work # is less that 20000k we've solved the puzzle

if (workNumber < 20000)

{

return true;

}

}

catch (IndexOutOfBoundsException ex)

{

BlockChain.PrintError("Index out of bounds", ex);

return false;

}

return false;

}

/\*\*

\* Code Used from WorkA to Generate a random Alpha Numeric String

\* https://condor.depaul.edu/elliott/435/hw/programs/Blockchain/WorkA.java

\* @param seed The number of randomly chosen characters you want to use

\* @return A random Alpha Numeric string

\*/

private String randomAlphaNumeric(int seed)

{

String alphaNumeric = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

StringBuilder builder = new StringBuilder();

// Loop until the seed value is 0

while (seed-- != 0)

{

// Using Randomly choose a characters index from the A String of all Alpha Numeric values

// repeat until the seed count is 0

int character = (int)(Math.random() \* alphaNumeric.length());

builder.append(alphaNumeric.charAt(character));

}

return builder.toString();

}

/\*\*

\* Send an updated ledger to all processes

\*/

public void SendVerifiedBlock()

{

PrintStream out;

Socket sock;

try

{

// Forward the new block to all of the block chain server so it can be added to the ledger

int[] blockChainServerPorts = Ports.getBlockChainServerPortsInUse();

for (int i = 0; i < blockChainServerPorts.length; i++)

{

// send to each process in group, including us:

sock = new Socket(BlockChain.ServerName, blockChainServerPorts[i]);

out = new PrintStream(sock.getOutputStream());

BlockChain.PrintInformation("Sending updated block ledger");

out.println(Utilities.SerializeRecord(BlockChain.BlockLedger));

out.flush();

out.close();

sock.close();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to send verified block to block chain server", ex);

}

}

/\*\*

\* Validate the block checking the signed blockId, the signed data block, and the previous hash

\* @param record The record block to validate

\* @return

\*/

private boolean isValidBlock(BlockRecord record)

{

// Get the signed block id to verify it

try

{

byte[] signedBlockId = Base64.getDecoder().decode(record.getSignedBlockId());

// Validate the signature of the signed block id if it's invalid dont bother processing it

if (!Utilities.GetKeyManager().VerifySignature(record.getBlockId().getBytes(), signedBlockId))

{

BlockChain.PrintError("Record's blockId has an been signed by an invalid private key");

return false;

}

}

catch (IllegalArgumentException ex)

{

BlockChain.PrintError("Failed to base64 decode the signed block id", ex);

return false;

}

// Get the signed data block to verify it

try

{

byte[] signedDataBlock = Base64.getDecoder().decode(record.getSignedSHA256DataBlock());

// Validate the signature of the signed data id if it's invalid dont bother processing it

if (!Utilities.GetKeyManager().VerifySignature(record.getSHA256HashedDataBlock().getBytes(), signedDataBlock))

{

BlockChain.PrintError("Record's data block has an been signed by an invalid private key");

//return false;

}

}

catch (IllegalArgumentException ex)

{

BlockChain.PrintError("Failed to base64 decode the signed dat block", ex);

return false;

}

return true;

}

}

/\*\*

\* The Public Key worker class

\*/

class UnverifiedBlockWorker extends Thread

{

private Socket socket;

UnverifiedBlockWorker(Socket s)

{

this.socket = s;

}

public void run()

{

BlockChain.PrintInformation("Unverified Block Client Connected");

BufferedReader in = null;

try

{

// create an input stream on the specified socket

in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));

try

{

BlockChain.PrintInformation("Received a new Unverified Block");

String newBlock = "";

String incomingBlock;

while ((incomingBlock = in.readLine()) != null)

{

newBlock += incomingBlock;

}

BlockChain.Queue.put(Utilities.DeserializeRecord(newBlock));

}

catch (Exception ex)

{

BlockChain.PrintError("Server error", ex);

}

finally

{

in.close();

this.socket.close();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Error opening i/o pipe on the specified socket", ex);

}

}

}

/\*\*

\* The Public Key worker class listens for the public key and then creates the KeyManager if it doesnt already have a

\* publicKey

\*/

class PublicKeyWorker extends Thread

{

private Socket socket;

PublicKeyWorker(Socket s)

{

this.socket = s;

}

public void run()

{

BlockChain.PrintInformation("Public Key Client Connected");

ObjectInputStream in;

// If this is process 2 we've already established the public key so we dont need to do anything else here

if (BlockChain.ProcessId == 2)

{

return;

}

try

{

// create an input stream on the specified socket

in = new ObjectInputStream(this.socket.getInputStream());

try

{

PublicKey publicKey = (PublicKey) in.readObject();

BlockChain.PrintInformation("Process " + BlockChain.ProcessId + " got a new key: " + publicKey.toString());

if (Utilities.GetKeyManager() == null)

{

BlockChain.PrintInformation("Setting public key");

Utilities.SetKeyManager(new KeyManager(publicKey));

}

}

catch (Exception ex)

{

BlockChain.PrintError("Server error", ex);

}

finally

{

in.close();

this.socket.close();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Error opening i/o pipe on the specified socket: ", ex);

}

}

}

/\*\*

\* The Key Manager Worker listens for record blocks when a block comes in it uses the secret key to sign the

\* SHA256 signed string.

\*/

class KeyManagerWorker extends Thread

{

private Socket socket;

private KeyManager keyManager;

KeyManagerWorker(Socket s, KeyManager keyManager)

{

this.socket = s;

this.keyManager = keyManager;

}

public void run()

{

BlockChain.PrintInformation("Key Manager Client Connected");

PrintStream out = null;

BufferedReader in = null;

try

{

// create an input stream on the specified socket

in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));

try

{

// Listen for a new block to be completed and then add it to the current ledger

String recordBlock = "";

String incomingBlock;

while ((incomingBlock = in.readLine()) != null)

{

recordBlock += incomingBlock;

}

// Get the new block, sign the SHA256 string and the blockId with the private key, and send the block out to the unverified block process

BlockRecord blockToSend = Utilities.DeserializeRecord(recordBlock);

// Generate a hash of the data block

byte[] blockHash = Utilities.GetHash(Utilities.SerializeDataBlock(blockToSend.getDataBlock()));

// This code was taken from Blockh.java @ https://condor.depaul.edu/elliott/435/hw/programs/Blockchain/BlockH.java

// I actually had this working without building the hex string and then when I switched to using more complex data in the hash it broke everything and

// i ran out of time to figure out why

StringBuffer sb = new StringBuffer();

for (int i = 0; i < blockHash.length; i++)

{

sb.append(Integer.toString((blockHash[i] & 0xff) + 0x100, 16).substring(1));

}

String SHA256String = sb.toString();

blockToSend.setSHA256HashedDataBlock(SHA256String);

// Sign the data block and base 64 encode it

byte[] signedDataBlock = this.keyManager.SignData(blockToSend.getSHA256HashedDataBlock().getBytes("UTF-8"));

blockToSend.setSignedSHA256DataBlock(Base64.getEncoder().encodeToString(signedDataBlock));

// Sign the block and then base 64 encode it

byte[] signedBlockId = this.keyManager.SignData(blockToSend.getBlockId().getBytes("UTF-8"));

blockToSend.setSignedBlockId(Base64.getEncoder().encodeToString(signedBlockId));

int[] unverifiedBlockPorts = Ports.getUnverifiedBlockServerPortsInUse();

// send the generated block to each process

for (int i = 0; i < unverifiedBlockPorts.length; i++)

{

if( blockToSend.getBlockId().equals("0"))

{

BlockChain.PrintInformation("Sending DummyBlock Block to Process " + i);

}

else

{

BlockChain.PrintInformation("Sending Unverified Block to Process " + i);

}

Socket unverifiedBlockServerSocket = new Socket(BlockChain.ServerName, unverifiedBlockPorts[i]);

out = new PrintStream(unverifiedBlockServerSocket.getOutputStream());

out.println(Utilities.SerializeRecord(blockToSend));

out.flush();

out.close();

unverifiedBlockServerSocket.close();

}

}

catch (Exception ex)

{

BlockChain.PrintError("Server error", ex);

}

finally

{

this.socket.close();

in.close();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Error opening i/o pipe on the specified socket: ", ex);

}

}

}

/\*\*

\* The BlockChain worker class

\*/

class BlockChainWorker extends Thread

{

private Socket socket;

private static final Lock serializeLock = new ReentrantLock();

private static boolean IsSerializing = false;

BlockChainWorker(Socket s)

{

this.socket = s;

}

public void run()

{

BlockChain.PrintInformation("Block Chain Client Connected");

BufferedReader in = null;

try

{

// create an input stream on the specified socket

in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));

try

{

// Listen for a new ledger to come in

String newLedger = "";

String incomingBlock;

while ((incomingBlock = in.readLine()) != null)

{

newLedger += incomingBlock;

}

BlockChain.BlockLedger = Utilities.DeserializeLedger(newLedger);

if (BlockChain.ProcessId == 0)

{

this.ExportLedger();

}

}

catch (Exception ex)

{

BlockChain.PrintError("Block Chain Server error", ex);

}

finally

{

this.socket.close();

in.close();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Error opening i/o pipe on the specified socket", ex);

}

}

private void ExportLedger()

{

BlockChain.PrintInformation("Exporting updated ledger");

try

{

BufferedWriter bw = null;

String serializedBlock;

serializeLock.lock();

try

{

serializedBlock = this.SerializeRecord(BlockChain.BlockLedger);

}

finally

{

serializeLock.unlock();

}

try

{

BlockChain.PrintError("Current Ledger Size: " + BlockChain.BlockLedger.size());

//BlockChain.PrintInformation("--NEW BLOCKCHAIN--\n" + serializedBlock);

bw = new BufferedWriter(new FileWriter("BlockChainLedger.json", false));

bw.write(serializedBlock);

bw.flush();

}

catch (IOException ex)

{

BlockChain.PrintError("Error while exporting the blockchain ledger", ex);

}

finally

{

bw.close();

}

}

catch (Exception ex)

{

BlockChain.PrintError("Error while exporting the blockchain ledger", ex);

}

}

/\*\*

\* Serialize a list of block records

\*

\* @param blockRecords The list of block records to serialize

\*/

private String SerializeRecord(ArrayList<BlockRecord> blockRecords)

{

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

return gson.toJson(blockRecords);

}

}

/\*\*

\* The BlockChain thread

\*/

class BlockChainThread implements Runnable

{

private int port;

BlockChainThread(int port)

{

this.port = port;

}

public void run()

{

int q\_len = 6;

Socket socket;

try

{

ServerSocket serverSocket = new ServerSocket(port, q\_len);

BlockChain.PrintInformation(String.format("BlockChain Process listening on the port %s.", port));

while (true)

{

// wait for the next ADMIN client connection:

socket = serverSocket.accept();

// Once a connection has come in start the admin worker

new BlockChainWorker(socket).start();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to start block chain worker", ex);

}

}

}

/\*\*

\* The UnverifiedBlock thread

\*/

class UnverifiedBlockThread implements Runnable

{

private int port;

UnverifiedBlockThread(int port)

{

this.port = port;

}

public void run()

{

int q\_len = 6;

Socket socket;

try

{

ServerSocket serverSocket = new ServerSocket(port, q\_len);

BlockChain.PrintInformation(String.format("Unverified Block Process listening on the port %s.", port));

while (true)

{

// wait for the next unverified block to be sent

socket = serverSocket.accept();

// Once a block has come in start the admin worker

new UnverifiedBlockWorker(socket).start();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to start block chain worker with exception: ", ex);

}

}

}

/\*\*

\* The BlockChain thread

\*/

class PublicKeyThread implements Runnable

{

private int port;

PublicKeyThread(int port)

{

this.port = port;

}

public void run()

{

int q\_len = 6;

Socket socket;

try

{

ServerSocket serverSocket = new ServerSocket(port, q\_len);

BlockChain.PrintInformation(String.format("PublicKey Process listening on the port %s.", port));

while (true)

{

// wait for the next public key to come in

socket = serverSocket.accept();

// Once a key has come in start the public key worker

new PublicKeyWorker(socket).start();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to start public key worker", ex);

}

}

}

/\*\*

\* The Key Manager thread

\*/

class KeyManagerThread implements Runnable

{

private int port = 1524;

public void run()

{

int q\_len = 6;

Socket socket;

try

{

ServerSocket serverSocket = new ServerSocket(port, q\_len);

BlockChain.PrintInformation(String.format("Key Manager Process listening on the port %s.", port));

BlockChain.PrintInformation("Creating private/public key pair");

// Create the key manager and the public/private key pair

KeyManager keyManager = new KeyManager();

keyManager.GenerateKeyPair(1000);

// Set the Utility classes KeyManager and then send out the public key to all processes

Utilities.SetKeyManager(keyManager);

Utilities.SendKeys(Ports.getKeyServerPortsInUse());

while (true)

{

// wait for the next public key to come in

socket = serverSocket.accept();

// Once a key has come in start the public key worker

new KeyManagerWorker(socket, keyManager).start();

}

}

catch (IOException ex)

{

BlockChain.PrintError("Failed to start Key Manager Worker", ex);

}

}

}

/\*\*

\* BlockChain class

\*/

public class BlockChain

{

public static final boolean DEBUGMODE = true;

public static final int ProcessCount = 3;

public static final int PID = 0;

public static final String ServerName = "localhost";

public static final BlockingQueue<BlockRecord> Queue = new PriorityBlockingQueue<BlockRecord>();

public static ArrayList<BlockRecord> BlockLedger = new ArrayList<BlockRecord>();

public static int ProcessId = 0;

/\*\*

\* The main entry point of the block chain program

\*

\* @param args The arguments

\*/

public static void main(String args[])

{

String inputFileName;

// Create the ports object to store the port information about all ports in use as well as the ports per process

Ports.setPortsForAllProcesses(ProcessCount);

if (args.length == 0)

{

System.out.println("\n-------------------------------------------------------");

System.out.println("Usage: java BlockChain [ProcessNumber]");

System.out.println("Missing ProcessNumber parameter so defaulting to 0\n");

System.out.println("-------------------------------------------------------\n");

}

else

{

ProcessId = Integer.parseInt(args[0]);

}

switch (ProcessId)

{

case 1:

{

inputFileName = "BlockInput1.txt";

break;

}

case 2:

{

// Process 2 is the last process and is a little special, it will multi-cast the public keys

// which means it also needs to generate the key pair

inputFileName = "BlockInput2.txt";

// Process 2 hosts the key manager service so start it up

new Thread(new KeyManagerThread()).start();

break;

}

default:

{

inputFileName = "BlockInput0.txt";

break;

}

}

// Set the ports for this process

Ports.setPortsForCurrentProcess(ProcessId);

BlockChain.PrintInformation("Ports:");

System.out.println("\t\t Public Keys Port: " + Ports.getKeyServerPort());

System.out.println("\t\t UnverifiedBlocksPort: " + Ports.getUnverifiedBlockServerPort());

System.out.println("\t\t BlockChainPort: " + Ports.getBlockChainServerPort());

BlockChain.PrintInformation("\nUsing input file: " + inputFileName + "\n");

// Start the block chain, unverified blocks, and public keys servers as well as the UnverifiedBlock consumer

try

{

new Thread(new PublicKeyThread(Ports.getKeyServerPort())).start();

new Thread(new UnverifiedBlockThread(Ports.getUnverifiedBlockServerPort())).start();

new Thread(new UnverifiedBlockConsumer()).start();

new Thread(new BlockChainThread(Ports.getBlockChainServerPort())).start();

}

catch (Exception ex)

{

BlockChain.PrintError("Failed to start server threads", ex);

}

try

{

// Block each process until a public key is available this should signify all the servers are up and running

BlockChain.PrintInformation("waiting on PublicKey");

while (Utilities.GetKeyManager() == null)

{

Thread.sleep(1000);

}

// A little extra sleep to make sure the key is set up

Thread.sleep(1000);

}

catch (Exception ex)

{

BlockChain.PrintError("Error while sleeping", ex);

}

// Now that all the servers on each process should be running let process 2 send out the initial unverified block

if (ProcessId == 2)

{

Utilities.SendUnverifiedBlocks(null);

}

//All dummy blocks have been sent so get all the blocks from the file and send them out 1 by 1

ArrayList<BlockRecord> recordBlocks = Utilities.ReadInputFile(inputFileName, ProcessId);

for (BlockRecord record : recordBlocks)

{

Utilities.SendUnverifiedBlocks(record);

}

}

/\*\*

\* Write the log line to System.out

\*

\* @param logString The string to log

\*/

public static void PrintInformation(String logString)

{

System.out.println("Process " + BlockChain.ProcessId + ": " + logString);

}

/\*\*

\* Writes the error string and exception to System.err

\*

\* @param logString the log line

\* @param ex The exception

\*/

public static void PrintError(String logString, Exception ex)

{

String errorStr = "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ERROR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

errorStr += "Process " + BlockChain.ProcessId + ": " + logString + " with exception: " + ex + "\n";

System.err.println(errorStr);

ex.printStackTrace();

System.err.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ERROR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

/\*\*

\* Writes the error string and exception to System.err

\*

\* @param logString the log line

\*/

public static void PrintError(String logString)

{

if (BlockChain.DEBUGMODE)

{

String errorStr = "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ERROR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

errorStr += "Process " + BlockChain.ProcessId + ": " + logString + "\n";

errorStr += "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ERROR\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";

System.err.println(errorStr);

}

}

}

[

{

"DataBlock": {

"FirstName": "Julie",

"LastName": "Wilson",

"SSN": "123-45-6999",

"DOB": "1996.03.07",

"Diagnosis": "Insomnia",

"Treatment": "Exercise",

"Medication": "HotPeppers"

},

"BlockNumber": 1,

"SHA256HashedDataBlock": "28ce2cc53e5149603ab2f0997ee43cca9d9172d3bd437e7774e3cee9e587a7b3",

"SignedSHA256DataBlock": "Vy/XC6a5wTRZ3F0hSqMdrp4H+oQuT1iXWKHutJjiCushgYBtCEqzGzkX0A79xM0K+Lrmvc6uBNMZnQij4NOcRH01FFskmHfihJisZJVF85GsD77FU3vZ4eva+MVsADEOvFdJfAwLvqWCQ9PvbUn/T+RnSymWS8IiMYXuwp+ENSg=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "44c1e462-3065-4802-8d2c-8be3f7b61b5d",

"SignedBlockId": "ITW+/nw0a4Zj2Ic+7Pf9wncGyiOLVnQQZhrzZiHg7t53c4slTMukiaGYchFac1dDUGcYsTbhhSAviDhLuFrKHZ2Ul17G6ahC/H8eSBhShphKuN36KQU8rKmvafSyubCAwUs92akwf8WPncAadbSO4XyKZEKDDRp41uGP789H+jo=",

"VerificationProcessId": "2",

"CreatingProcess": "0",

"PreviousHash": "73318A38B61FB62FD5BC30A74986E2D0C8E6CB00D39C9AD22005FBAE20CB0E7CC14T9CXNGT",

"Seed": "C14T9CXNGT"

},

{

"DataBlock": {

"FirstName": "Sally",

"LastName": "McCutty",

"SSN": "123-456-999",

"DOB": "1970.01.01",

"Diagnosis": "Migraine",

"Treatment": "IcePack",

"Medication": "Almotriptan"

},

"BlockNumber": 2,

"SHA256HashedDataBlock": "aff6c698b1af261c3fc6b343138b4540e2ae2ef5c19a996705f9d7cc4da67b87",

"SignedSHA256DataBlock": "AUrSnq50nLamQY6FX+jcKzEJMH0BQ9I7S1OBeJrr5sNeZVl/T/0t4oGVkVseOPzImbFb/Yg0p54NxGpNeKkkiVheIKCEXqWousrAvG09pIa/SWhQQ2Ze4qcC07XV9YarLgvTdqck8lM2PWUEI8ixkkApGZHIrJLiaYQOP2Oa5Aw=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "b916cd22-1910-4f29-9e93-af7298a83c79",

"SignedBlockId": "VaGup67idgUOc7ws4nLo9h6WMHti3aRBP4+WveYxCfnCwUyRGKu8h7mmNB6enPJu5mQhJprOEqJa/H/RnKqMoRljSmhRktPOzOgJAJUNhm6JTS7RwezuiA7BWeKXoMTg21+w3EFyxktcVbAXmDGdtWv32BLBV496cGHMAj08S6E=",

"VerificationProcessId": "2",

"CreatingProcess": "1",

"PreviousHash": "73318A38B61FB62FD5BC30A74986E2D0C8E6CB00D39C9AD22005FBAE20CB0E7CC14T9CXNGTY480ABX14O",

"Seed": "Y480ABX14O"

},

{

"DataBlock": {

"FirstName": "Wayne",

"LastName": "Blaine",

"SSN": "123-45-6777",

"DOB": "1942.07.07",

"Diagnosis": "Measles",

"Treatment": "WaitToGetBetter",

"Medication": "CodLiverOil"

},

"BlockNumber": 3,

"SHA256HashedDataBlock": "4c61939ac627aeb0b89b8e0e4649498b16b4a58a469c5dffb54cded6daaacf1b",

"SignedSHA256DataBlock": "Os2RK5xR/eWDt7I3tScMvLAN9KG5X4YMhkE+mx6962o7CCKvSgeg4rjwmW33SYtsNGm4vIsG+/PhxqI9/bRitb/bRlXiBpjIF4WgNtBukYXlY1FcyizKqO1HPmJsDjQTZ5tPgepjU5FUhipggUnFENpgwTYamF/6Mym2eP9sAwE=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "dfc551a0-cdd6-4946-8787-a2da7ca3d1ad",

"SignedBlockId": "FodbbZg08qYJQiP0qSm49TbJLXOhmYw12ToMyQ/fhFSpjzdidQTM2O0BqEl1yDgM4tWm/+xVN/KqRoFDYJ42N7wMSkA0pV52dNC0vze4AsgA6jBnoiDzpsUkjXI7T0RRUJLxxFSe9r1Q/k02jOobwrwT+13VnBzu4lmgONbVcvQ=",

"VerificationProcessId": "2",

"CreatingProcess": "0",

"PreviousHash": "73318A38B61FB62FD5BC30A74986E2D0C8E6CB00D39C9AD22005FBAE20CB0E7CC14T9CXNGTY480ABX14OH3E5Z2N4FM",

"Seed": "H3E5Z2N4FM"

},

{

"DataBlock": {

"FirstName": "John",

"LastName": "Smith",

"SSN": "123-45-6789",

"DOB": "1996.03.07",

"Diagnosis": "Chickenpox",

"Treatment": "BedRest",

"Medication": "aspirin"

},

"BlockNumber": 4,

"SHA256HashedDataBlock": "0d5be93aaeabb3499828bdef13f2cdbdf5d5a40191c43bdd918097cadf595c89",

"SignedSHA256DataBlock": "W/QIkFiF1PbkkaYo5zGG1CjWUEKRDv1wpGQYfqr2lAyG/V3GOa7a8EyvuqM9dSZKKMdfLb7XORa0Wp++BCgN3tnnfydD5P4oNX1b0QifHPe4fSLiqj8HwrMYscIoB+9+0lDrjN+1XWUvV81sA2u9N/l+HFGkkrUiJkSWar6HPow=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "4ee170c7-bc98-44fb-85c3-2e6dc47465d9",

"SignedBlockId": "QWBCN+BHwSlOwKSPcjWwC7D58k0IXzffmvZKphYBlPu1XO+YotsdTGpu8N1UmiVI9RGgc8dVXEnqX4Um7Lp21qLrGXXF2K298Nl26l2qM1Ib4hurm+IB1CU31jKH6/m4/4Emr7Zv2daGfvKB7qVFnkzghZiDEbJs37q04ndwxhQ=",

"VerificationProcessId": "2",

"CreatingProcess": "0",

"PreviousHash": "73318A38B61FB62FD5BC30A74986E2D0C8E6CB00D39C9AD22005FBAE20CB0E7CC14T9CXNGTY480ABX14OH3E5Z2N4FMEEBSCAIXS3",

"Seed": "EEBSCAIXS3"

},

{

"DataBlock": {

"FirstName": "Helen",

"LastName": "Keller",

"SSN": "666-45-6789",

"DOB": "1880.06.27",

"Diagnosis": "Arthritis",

"Treatment": "WarmCloths",

"Medication": "Aspirin"

},

"BlockNumber": 5,

"SHA256HashedDataBlock": "c669b5ca867249c640704b2e5a89a5dfcc67d8023c651f88f82047401ed4ea04",

"SignedSHA256DataBlock": "QGigrx/W1zdFwCXd/flNmW3buNbfpRnvUT1jiM9U1U774n7Cgc+esMFgnJ67by1ReLIV0IRBLKSIGU058Ru34PF/Uu5NfNWIoaJQW01D5o3Tt2GBE/T9fYsg1Q5esKbhLs5gRNXP/GXnGOESqBrfTmfHafjkNgoc8mOLwe4zgyc=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "bd9fceae-a952-4958-a340-3b2a6f6cf61f",

"SignedBlockId": "OyQxk3vxwWXfrvdWO02GXQYgyvieVDRJreHAqeKu9tPrny6MTN05pDvep27+6RmaxJgzC9yctDdQ+sacpWRDHYmlqFgaAGKCNPFbY5xo4Zf3s7KiCHvqxKafaPnZCkqnOUUcL0H3YU8FvRlmxVixfuwQsck6Ux2NiC7Cgs4Qds8=",

"VerificationProcessId": "2",

"CreatingProcess": "2",

"PreviousHash": "73318A38B61FB62FD5BC30A74986E2D0C8E6CB00D39C9AD22005FBAE20CB0E7CC14T9CXNGTY480ABX14OH3E5Z2N4FMEEBSCAIXS3LHC5VGRI5D",

"Seed": "LHC5VGRI5D"

},

{

"DataBlock": {

"FirstName": "",

"LastName": "",

"SSN": "",

"DOB": "",

"Diagnosis": "",

"Treatment": "",

"Medication": ""

},

"BlockNumber": 6,

"SHA256HashedDataBlock": "e864138cae72e7954d150baa3313be7622ed31e271fdee26f53219c67e954b5d",

"SignedSHA256DataBlock": "WEfX7NA535UgK3EwyDHCyy2aW1y2mKX+yPl9oism46Qu0RwJ/KK4ajZTzY5sunnu4mdrZlS2+e5NvmZg8HOLEa829TR48eq5mz0ZPmF1X9CWw0QprGmFlZZlG4fhA4PlNH8zK8UP0FlZeIKIqV9iGtzAno8tMwC8wSxG/9Wx9eY=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "0",

"SignedBlockId": "MasMlS3rKupjqhvH93fkQW2UjwbN5vZMZsIKjTVij0EAky8V9LA+ZL5BK5VNmXKAAAV9QrEsg0Os78Yl04TH7qvu8FDU6kXmHk3Dl6OtcVO+IIcKGExVBky+C93Jwr8bGrO8Yu1hb6kaUgDWsLosXLa+VGdIy9hOMdXjRm2VWUc=",

"VerificationProcessId": "2",

"CreatingProcess": "0",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7",

"Seed": "BLLIDAUQV7"

},

{

"DataBlock": {

"FirstName": "Joe",

"LastName": "Blow",

"SSN": "123-45-6888",

"DOB": "1996.03.07",

"Diagnosis": "Smallpox",

"Treatment": "BedRest",

"Medication": "Whiskey"

},

"BlockNumber": 7,

"SHA256HashedDataBlock": "dcfe29bb2c75178d1884030542dcbbbff7f840eb2df6ccd626ef0d2473416c3c",

"SignedSHA256DataBlock": "Os2RK5xR/eWDt7I3tScMvLAN9KG5X4YMhkE+mx6962o7CCKvSgeg4rjwmW33SYtsNGm4vIsG+/PhxqI9/bRitb/bRlXiBpjIF4WgNtBukYXlY1FcyizKqO1HPmJsDjQTZ5tPgepjU5FUhipggUnFENpgwTYamF/6Mym2eP9sAwE=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "505cd721-0482-4c7c-af18-08a5415d723e",

"SignedBlockId": "XZWgaB8DPmrTQOWBp58jlSq0NoDHuUx1nDuIjK1A6ZodAaKlAjh67iQgaEbW/V94xk/UUbsK9VGGj0hC7h5RO/UbaHU0o7pgxM5vEZc+vNRWG1ChFpHhre7xhCoJG/eKZKHy5HA5FP47VdI9CpJYjLrIwYYHsNBRU2malVL97nY=",

"VerificationProcessId": "1",

"CreatingProcess": "0",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3L",

"Seed": "US59IK9M3L"

},

{

"DataBlock": {

"FirstName": "Bruce",

"LastName": "Lee",

"SSN": "456-789-123",

"DOB": "1940.11.27",

"Diagnosis": "SoreStomach",

"Treatment": "LessCombat",

"Medication": "Vicodine"

},

"BlockNumber": 8,

"SHA256HashedDataBlock": "59a52c8b5004dd11c16c114cdd28b20113465658b1f7b0427b1e82dd5c253c48",

"SignedSHA256DataBlock": "AUrSnq50nLamQY6FX+jcKzEJMH0BQ9I7S1OBeJrr5sNeZVl/T/0t4oGVkVseOPzImbFb/Yg0p54NxGpNeKkkiVheIKCEXqWousrAvG09pIa/SWhQQ2Ze4qcC07XV9YarLgvTdqck8lM2PWUEI8ixkkApGZHIrJLiaYQOP2Oa5Aw=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "9f44be6a-fc98-4234-9522-655de0bcf0ef",

"SignedBlockId": "cLyXmSDHwsZBiKComUA8uz3BBYGXniAoo8w2fvZDPvxxkG/dUR0CPIx30eROq/AdHvJpEWPoBBc707wYwzPA2wzptwlw5FgS9oZqG/+5R9lGOYVBi9JHk0sRHj4sXO3FDTLfvWdTOzau2Jpc2jq6d/xn2/Ku6lK2RoEWGZvC+aE=",

"VerificationProcessId": "1",

"CreatingProcess": "1",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3LLRKU0NQOWK",

"Seed": "LRKU0NQOWK"

},

{

"DataBlock": {

"FirstName": "Rita",

"LastName": "Vita",

"SSN": "999-456-789",

"DOB": "1992.01.31",

"Diagnosis": "ObessivePersonality",

"Treatment": "TryToRelax",

"Medication": "Ibuprofen"

},

"BlockNumber": 9,

"SHA256HashedDataBlock": "6e8dec1cc250835597b8a0284b92c0798ecdf92e5de4be75678eb84f1dc7ae54",

"SignedSHA256DataBlock": "FIGIvS7tUL+DHPzy+VhXidtr/8/aZHLHUcfuVVYvdRjmW0Lm4W5SnNrTpg0XeIPxejTcoTylQcITeWZ5a5zxzkCF8tKwanwSYzaXerAg3isBOgGHxZxlNI0OaWljCtpnGQWtMTsRwDMW3mlzKDj54kmIfM0QjAXzOFzqKiuZrPo=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "4ce25fc8-844c-454d-8c9c-b162f8e3bdc4",

"SignedBlockId": "GRsjlSOuew3vmPBstHNoK5tnYvAl/Weo6LBvwgeupieCToVzBXNZnjR27CFV+forU0y9Toxgq63aE0g34iWR/7r5N/q7wwfkUnXgmpEIt3IriB1eXqIow8A8xXtJYiVVO4KsN0JMVcMoOYHKQlHdHXhZndNj0WBgz9WB9KzvOXY=",

"VerificationProcessId": "1",

"CreatingProcess": "1",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3LLRKU0NQOWKWWPR0ODB9S",

"Seed": "WWPR0ODB9S"

},

{

"DataBlock": {

"FirstName": "Wei",

"LastName": "Xu",

"SSN": "123-456-333",

"DOB": "1996.03.22",

"Diagnosis": "Shingles",

"Treatment": "WaitForRelief",

"Medication": "Zovirax"

},

"BlockNumber": 10,

"SHA256HashedDataBlock": "81d4307e3955716185582bb0562f4fc86e330422f3c81ac8ccd73f8dcfcda9a1",

"SignedSHA256DataBlock": "d1sSsj4rb3YJNkaizYRugl+hC2lqjI5kiHVyYs1CE7r2HPCii/iaXloRv6zwAOnI9QJ1/N8Sdfld3oCw4SlrmXI5CRvE6cQ1+RiOhYxRI/hP09Q+TA2XYjVXFgbf1nk2JHA6LUig2B3vZ52qX68FRvmCbqqMr5byxMXXSt2Bj7M=",

"CreationTime": "Oct 23, 2018 10:23:45 PM",

"BlockId": "701e3ddb-3171-4989-aa2d-b322250c1c8f",

"SignedBlockId": "GO0lEeXz+2k0TPiJLbMZOhMROLUCfSODqjcFQXhqmIZhWi+DdD0hAYnJ+grM6FBeD3md9Mu0+gq1jCHKLvu3Y7GYb7QanGJY1Zq+TKBhCq+3cbidilELQrlDHzNjuB0maLlA3bE6+Vp5J2Az/74QslWKhxZkDelaSFT8XlYtNzA=",

"VerificationProcessId": "1",

"CreatingProcess": "1",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3LLRKU0NQOWKWWPR0ODB9SH7SDGN01LL",

"Seed": "H7SDGN01LL"

},

{

"DataBlock": {

"FirstName": "John",

"LastName": "Kennedy",

"SSN": "333-45-6999",

"DOB": "1917.05.29",

"Diagnosis": "AddisonsDisease",

"Treatment": "DrugTherapy",

"Medication": "Steroids"

},

"BlockNumber": 11,

"SHA256HashedDataBlock": "ab9fc5ea88fca8a01ae676f902d55393e8940aee608944cca5d367fdb0b5ca76",

"SignedSHA256DataBlock": "MLdnV5UNhZxFh4ZgindNwNM7cSCkUfvP3vbF5lHITDcJT7u7lCH9B3U7roncBKPdK0KoLerA/E6iJF2A1xu7p4fFHqmP07u4uLkCWOI8/pDWy8Bpnwzkr/cUQuc2vpT1KzzNnWSFcnfyjtuCAk2Q20voo+d8z/qrg1fbBVLxOCc=",

"CreationTime": "Oct 23, 2018 10:23:46 PM",

"BlockId": "8ad0098d-2769-4d5b-a6e8-49fd5ca6dba8",

"SignedBlockId": "ESNUrYf+IXUYrwLTNuOXNzou7ReqfQvfquNPxSRYF5VIgJl1Vy0DtNp0eou5tGRxFIKMBv+mczMTbysZjcWrSOnzqSBo6/xp7Ib1+SseU+2j7Krn5hsV+uiqDbm1u4pMDgHBGXW8YczKtWgxYCqKZMPErMgzT4ZHFKK28YGGeXQ=",

"VerificationProcessId": "1",

"CreatingProcess": "2",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3LLRKU0NQOWKWWPR0ODB9SH7SDGN01LL89OA828BMX",

"Seed": "89OA828BMX"

},

{

"DataBlock": {

"FirstName": "Joe",

"LastName": "DiMaggio",

"SSN": "111-22-3333",

"DOB": "1914.11.25",

"Diagnosis": "SoreKnees",

"Treatment": "RestFromSports",

"Medication": "Aspirin"

},

"BlockNumber": 12,

"SHA256HashedDataBlock": "670bb01430e160db7d1ecdf38da49610e512ce235b2ac7380042c9187d185326",

"SignedSHA256DataBlock": "YhJRn8DiiIotlrD2hms59AjZ1B4wJGWopMDjbWFsw4N6WE1jeYr0P/TtboVb5lmyg9bYE1eT8AFeNELXCWh5MAfcV4Y+X9TtBQrfEJnhD2PDy4yUquIuen77Q8DLumCYoDqolyhz9kAmcgy5sx6LXQx9XsPorR6JH8vJhRg9yt8=",

"CreationTime": "Oct 23, 2018 10:23:46 PM",

"BlockId": "69310bfc-c4e3-4cfc-accc-2870ba585bcf",

"SignedBlockId": "Va/mAIzDtPPVUkHoKRxG7Y0pjeM0GYDEQ4KGMuGjrYBahdVlqHVMdyEWfUq3+N5U03zQ3YCjOpIPR0HnQL9NBTOJ25FfTkp/BEy7c550jg+HpFza0Wf8jqXTH3c5cX9s2FkD5IKuzdiULs7oV2Qol6AkEKoCJYlO0cN58yg/ySo=",

"VerificationProcessId": "1",

"CreatingProcess": "2",

"PreviousHash": "2DCAC1EC9FAEA4B40289243FEFC132C876FD6DD5981F8062DB0D75371DAE6262BLLIDAUQV7US59IK9M3LLRKU0NQOWKWWPR0ODB9SH7SDGN01LL89OA828BMXUHZVMBB8B4",

"Seed": "UHZVMBB8B4"

}

]