**Project2Task5Server**

import java.math.BigInteger;  
import java.net.\*;  
import java.io.\*;  
import java.security.MessageDigest;  
import java.util.Map;  
import java.util.Scanner;  
import java.util.TreeMap;  
// Name: Leo Lin  
// Andrew ID: hungfanl  
public class VerifyingServerTCP {  
 static BigInteger *n*, *e*, *id*;  
 static int *operator*;  
 static Map<BigInteger, Integer> *database* = new TreeMap<>();  
 public static void main(String args[]) {  
 Socket clientSocket = null;  
 try {  
 int serverPort = 7777; // the server port we are using  
 // Create a new server socket  
 ServerSocket listenSocket = new ServerSocket(serverPort);  
 clientSocket = listenSocket.accept();  
 // If we get here, then we are now connected to a client.  
 // Set up "in" to read from the client socket  
 Scanner in;  
 in = new Scanner(clientSocket.getInputStream());  
 // Set up "out" to write to the client socket  
 PrintWriter out;  
 out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream())));  
 // In order for the server to run forever, we need to handle the situation when client shut down  
 while (true) {  
 String data;  
 // When the client is connected, there will be string pass to in.  
 // Under the situation, we can safely conduct the calculation and return the value.  
 if(in.hasNextLine()){  
 data = in.nextLine();  
 // The string passed by the client is id, e, n, operator, number seperated by ","  
 // and the hash message seperated by " "  
 String[] message = data.split(" ");  
 String[] operation = message[0].split(",");  
 *id* = new BigInteger(operation[0]);  
 *e* = new BigInteger(operation[1]);  
 *n* = new BigInteger(operation[2]);  
 *operator* = Integer.*parseInt*(operation[3]);  
 int number = Integer.*parseInt*(operation[4]);  
 // conduct corresponding arithmetic according to the three index  
 // verify the hash message  
 if(*idMatch*(message[0], message[1])){  
 int outcome = *arithmetic*(*operator*, *id*, number);  
 // return the outcome of calculation to the client  
 System.*out*.println("Returning sum of " + outcome + " to client");  
 out.println(outcome);  
 }  
 else out.println("The request is wrongly encrypted");  
 out.flush();  
 }  
 // However, when the client is shut down, there will not be next line pass to in.  
 // In such cases, we will need to have clientSocket to accept another socket and renew the in/out  
 else {  
 clientSocket = listenSocket.accept();  
 in = new Scanner(clientSocket.getInputStream());  
 out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream())));  
 }  
 }  
 // Handle exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO Exception:" + e.getMessage());  
  
 // If quitting (typically by you sending quit signal) clean up sockets  
 } finally {  
 try {  
 if (clientSocket != null) {  
 clientSocket.close();  
 }  
 } catch (IOException e) {  
 // ignore exception on close  
 }  
 }  
 }  
 // Doing corresponding arithmetic according to the user's request  
 public static int arithmetic(int operator, BigInteger id, int number){  
 String[] operation = {"Addition", "Subtraction", "Check"};  
 System.*out*.println("The visitor's ID is: " + id);  
 System.*out*.println("Operator: " + operation[operator-1]);  
 if(operator == 1){  
 *database*.put(id, *database*.getOrDefault(id,0) + number);  
 }  
 else if (operator == 2){  
 *database*.put(id, *database*.getOrDefault(id,0) - number);  
 }  
 // return the number after calculation  
 return *database*.getOrDefault(id,0);  
 }  
  
 // Check if the hash of messageToCheck is same as the decryption if encryptedHashStr  
 // Exploit from ShortMessageVerify.java provided by the handout  
 public static boolean idMatch(String messageToCheck, String encryptedHashStr) {  
 BigInteger decryptedHash = null;  
 BigInteger bigIntegerToCheck = null;  
 try{  
 // Decrypt it  
 decryptedHash = new BigInteger(encryptedHashStr).modPow(*e*, *n*);  
 // Get the bytes from messageToCheck  
 byte[] bytesOfMessageToCheck = messageToCheck.getBytes("UTF-8");  
 // compute the digest of the message with SHA-256  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
  
 byte[] bigDigest = md.digest(bytesOfMessageToCheck);  
  
 // messageToCheckDigest is a full SHA-256 digest  
 // add a zero byte in front of bigDigest  
 byte[] messageToCheckDigest = new byte[bigDigest.length + 1];  
 messageToCheckDigest [0] = 0; // most significant set to 0  
 for(int i = 0; i < bigDigest.length; i++){  
 messageToCheckDigest [i+1] = bigDigest[i];  
 }  
 bigIntegerToCheck = new BigInteger(messageToCheckDigest);  
 } catch (Exception e){  
 e.printStackTrace();  
 }  
 System.*out*.println("The hash message provided by the client: " + decryptedHash);  
 System.*out*.println("The hash value of the message:" + bigIntegerToCheck);  
 System.*out*.println("Verify result: " + bigIntegerToCheck.equals(decryptedHash));  
 return bigIntegerToCheck.equals(decryptedHash);  
 }  
}