**Project2Task0Client**

// imports required for UDP/IP  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.SocketException;  
import java.util.Arrays;  
  
public class EchoClientUDP {  
 public static void main(String args[]) {  
 // define a Datagram (UDP style) socket  
 DatagramSocket aSocket = null;  
 try {  
 // show the announcement  
 System.*out*.println("The client is running");  
  
 // build an InetAddress object from a DNS name  
 InetAddress aHost = InetAddress.*getByName*("localhost");  
  
 // client sends a message to port 6789  
 int serverPort = 6789;  
  
 // define the user's input  
 String nextLine;  
  
 // initialize the socket  
 aSocket = new DatagramSocket();  
  
 // read user's input  
 BufferedReader typed = new BufferedReader(new InputStreamReader(System.*in*));  
  
 // loop to receive the input from client  
 while ((nextLine = typed.readLine()) != null) {  
 // build packet contents by input string  
 byte[] m = nextLine.getBytes();  
  
 // build the packet holding the destination address, port and byte array to contact the server  
 DatagramPacket request = new DatagramPacket(m, m.length, aHost, serverPort);  
  
 // send the Datagram on the socket  
 aSocket.send(request);  
  
 // prepare room for the reply  
 byte[] buffer = new byte[1000];  
  
 // build a datagram for the reply  
 DatagramPacket reply = new DatagramPacket(buffer, buffer.length);  
  
 // block and wait  
 aSocket.receive(reply);  
  
 // get bytes arrays from the reply with correct length  
 byte[] reply\_bytes = Arrays.*copyOf*(reply.getData(), reply.getLength());  
  
 // modify the request data with the correct number of bytes  
 reply.setData(reply\_bytes);  
  
 // show the result to the client  
 System.*out*.println("Reply: " + new String(reply.getData()));  
  
 // after hearing the "halt!" message from the server, the client makes an announcement and then exits  
 if (nextLine.equals("halt!")) {  
 System.*out*.println("Client side quitting");  
 break;  
 }  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (aSocket != null) aSocket.close();  
 }  
 }  
}

**Project2Task0Server**

// imports required for UDP/IP  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.SocketException;  
import java.util.Arrays;  
  
public class EchoServerUDP {  
 public static void main(String args[]) {  
 // define a Datagram (UDP style) socket  
 DatagramSocket aSocket = null;  
 try {  
 // show the announcement  
 System.*out*.println("The server is running");  
  
 // initialize the socket using 6789 port number  
 aSocket = new DatagramSocket(6789);  
  
 // loop forever  
 while (true) {  
  
 // prepare room for the request  
 byte[] buffer = new byte[1000];  
  
 // initialize the request  
 DatagramPacket request = new DatagramPacket(buffer, buffer.length);  
  
 // receive request data from the socket  
 aSocket.receive(request);  
  
 // get bytes arrays from the request with correct length  
 byte[] request\_bytes = Arrays.*copyOf*(request.getData(), request.getLength());  
  
 // modify the request data with the correct number of bytes  
 request.setData(request\_bytes);  
  
 // build the reply to client  
 DatagramPacket reply = new DatagramPacket(request.getData(),  
 request.getLength(), request.getAddress(), request.getPort());  
  
 // change request to string format  
 String requestString = new String(request.getData());  
  
 // show the request string  
 System.*out*.println("Echoing: " + requestString);  
  
 // send reply data to socket  
 aSocket.send(reply);  
  
 // after hearing the "halt!" message from the client, the server makes an announcement and then exits  
 if (new String(request.getData()).equals("halt!")) {  
 System.*out*.println("Server side quitting");  
 break;  
 }  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (aSocket != null) aSocket.close();  
 }  
 }  
}

**Project2Task0ClientConsole**

Text

Description automatically generated

**Project2Task0ServerConsole**

Text

Description automatically generated

**Project2Task1EavesdropperUDP**

// imports required for UDP/IP  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.SocketException;  
import java.util.Arrays;  
  
public class EavesdropperUDP {  
 public static void main(String args[]) {  
 // define a Datagram (UDP style) socket to communicate with client  
 DatagramSocket aSocket = null;  
  
 // define another Datagram socket to communicate with server  
 DatagramSocket malSocket = null;  
 try {  
 // malicious player listens on 6798  
 aSocket = new DatagramSocket(6798);  
  
 // initialize another socket to connect with server  
 malSocket = new DatagramSocket();  
  
 // build an InetAddress object from a DNS name  
 InetAddress aHost = InetAddress.*getByName*("localhost");  
  
 // loop forever  
 while (true) {  
  
 // prepare room for the request  
 byte[] requestBuffer = new byte[1024];  
  
 // initialize the request from the client  
 DatagramPacket request = new DatagramPacket(requestBuffer, requestBuffer.length);  
  
 // receive request data from the client  
 aSocket.receive(request);  
  
 // get bytes arrays from the request with correct length  
 byte[] request\_bytes = Arrays.*copyOf*(request.getData(), request.getLength());  
  
 // modify the request data with the correct number of bytes  
 request.setData(request\_bytes);  
  
 // display the client's messages through eavesdropping  
 System.*out*.println("Eavesdrop: " + new String(request.getData()));  
  
 // malicious player contact the server at 6789  
 DatagramPacket malRequest = new DatagramPacket(request.getData(), request.getData().length, aHost, 6789);  
  
 // malicious player forwards the message to Echo serve  
 malSocket.send(malRequest);  
  
 // prepare room for the reply from server  
 byte[] replyBuffer = new byte[1000];  
  
 // build a datagram for the reply  
 DatagramPacket malReply = new DatagramPacket(replyBuffer, replyBuffer.length);  
  
 // block and wait  
 malSocket.receive(malReply);  
  
 // get bytes arrays from the reply with correct length  
 byte[] malReply\_bytes = Arrays.*copyOf*(malReply.getData(), malReply.getLength());  
  
 // modify the malReply data with the correct number of bytes  
 malReply.setData(malReply\_bytes);  
  
 // build the reply to client  
 DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(), request.getAddress(), request.getPort());  
  
 // send server's reply data to the client  
 aSocket.send(reply);  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (aSocket != null) aSocket.close();  
 }  
 }  
}

**Project2Task1ThreeConsoles**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Project2Task2Client**

// imports required for UDP/IP  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.SocketException;  
import java.nio.ByteBuffer;  
import java.util.Arrays;  
  
public class AddingClientUDP {  
 // create global variables for socket, server port, and destination address  
 static DatagramSocket *aSocket* = null;  
 static int *serverPort* = 0;  
 static InetAddress *aHost* = null;  
  
 public static void main(String args[]) {  
 // define a Datagram (UDP style) socket  
  
 try {  
 // show the announcement  
 System.*out*.println("The client is running.");  
  
 *aHost* = InetAddress.*getByName*("localhost");  
  
 // define the user's input  
 String nextLine;  
  
 // read user's input  
 BufferedReader typed = new BufferedReader(new InputStreamReader(System.*in*));  
  
 // ask user to input a server port and initialize the server port  
 System.*out*.println("Please enter server port:");  
 *serverPort* = Integer.*parseInt*(typed.readLine());  
  
 // prompt the user enter numbers  
 System.*out*.println("Please enter numbers:");  
  
 // loop to receive the input from client  
 while ((nextLine = typed.readLine()) != null) {  
 // the client makes an announcement and then exits when user enter halt!  
 if (nextLine.equals("halt!")) {  
 System.*out*.println("Client side quitting");  
 break;  
 }  
  
 // convert string to int  
 int i = Integer.*parseInt*(nextLine);  
  
 // call the add function related to client server communications  
 int sum = *add*(i);  
  
 // show the result to the client  
 System.*out*.printf("The server returned %d.\n", sum);  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (*aSocket* != null) *aSocket*.close();  
 }  
  
 }  
  
 */\*\*  
 \* client communicates with the server  
 \*  
 \** ***@param*** *i from the user's input  
 \** ***@return*** *sum result from the server  
 \*/* public static int add(int i) {  
 // define and initialize the sum result  
 int sum = 0;  
 try {  
 // initialize the socket  
 *aSocket* = new DatagramSocket();  
  
 // build a ByteBuffer with capacity 4 to store the int and then return a byte array  
 byte[] m = ByteBuffer.*allocate*(4).putInt(i).array();  
  
 // build the packet holding the destination address, port and byte array to contact the server  
 DatagramPacket request = new DatagramPacket(m, m.length, *aHost*, *serverPort*);  
  
 // send the Datagram on the socket  
 *aSocket*.send(request);  
  
 // prepare room for the reply  
 byte[] buffer = new byte[1000];  
  
 // build a datagram for the reply  
 DatagramPacket reply = new DatagramPacket(buffer, buffer.length);  
  
 // block and wait  
 *aSocket*.receive(reply);  
  
 // get bytes arrays from the reply with correct length  
 byte[] reply\_bytes = Arrays.*copyOf*(reply.getData(), reply.getLength());  
  
 // wraps the reply byte array into a buffer and then return the int value  
 sum = ByteBuffer.*wrap*(reply\_bytes).getInt();  
  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 }  
  
 return sum;  
 }  
  
}

**Project2Task2Server**

// imports required for UDP/IP  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.SocketException;  
import java.nio.ByteBuffer;  
import java.util.Arrays;  
import java.util.Scanner;  
  
public class AddingServerUDP {  
 // create a global variable for socket and sum  
 static DatagramSocket *aSocket* = null;  
 static int *sum* = 0;  
  
 public static void main(String args[]) {  
 try {  
 // show the announcement  
 System.*out*.println("Server started");  
  
 // prompt the user enter a port number for the server to listen to  
 System.*out*.println("Please enter server port:");  
 int serverPort = new Scanner(System.*in*).nextInt();  
  
 // initialize the socket using port number by user's input  
 *aSocket* = new DatagramSocket(serverPort);  
  
 // loop forever  
 while (true) {  
  
 // prepare room for the request  
 byte[] buffer = new byte[1000];  
  
 // initialize the request  
 DatagramPacket request = new DatagramPacket(buffer, buffer.length);  
  
 // receive request data from the socket  
 *aSocket*.receive(request);  
  
 // get bytes arrays from the request with correct length  
 byte[] request\_bytes = Arrays.*copyOf*(request.getData(), request.getLength());  
  
 // modify the request data with the correct number of bytes  
 request.setData(request\_bytes);  
  
 // get add int from the request  
 int add\_num = ByteBuffer.*wrap*(request\_bytes).getInt();  
  
 // show the add process to the server  
 System.*out*.println("Adding: " + add\_num + " to " + *sum*);  
  
 // performs the add operation  
 *sum* = *add*(add\_num);  
  
 // build a ByteBuffer with capacity 4 to store the sum and then return a byte array  
 byte[] sum\_bytes = ByteBuffer.*allocate*(4).putInt(*sum*).array();  
  
 // build the reply to client  
 DatagramPacket reply = new DatagramPacket(sum\_bytes, sum\_bytes.length, request.getAddress(), request.getPort());  
  
 // show the result to the server  
 System.*out*.printf("Returning sum of %d to client\n", *sum*);  
  
 // send reply data to socket  
 *aSocket*.send(reply);  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (*aSocket* != null) *aSocket*.close();  
 }  
 }  
  
 */\*\*  
 \* performs the add operation.  
 \*  
 \** ***@param*** *i from the request  
 \** ***@return*** *sum result  
 \*/* public static int add(int i) {  
 *sum* += i;  
 return *sum*;  
 }  
}

**Project2Task2ClientConsole**

Text

Description automatically generatedText

Description automatically generated

**Project2Task2ServerConsole**

Text

Description automatically generated

**Project2Task3Client**

// imports required for UDP/IP  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.SocketException;  
import java.nio.ByteBuffer;  
import java.util.Arrays;  
  
public class RemoteVariableClientUDP {  
 // create global variables for socket, server port, and destination address  
 static DatagramSocket *aSocket* = null;  
 static int *serverPort* = 0;  
 static InetAddress *aHost* = null;  
  
 public static void main(String args[]) {  
 try {  
 // show the announcement  
 System.*out*.println("The client is running.");  
  
 *aHost* = InetAddress.*getByName*("localhost");  
  
  
 // read user's input  
 BufferedReader typed = new BufferedReader(new InputStreamReader(System.*in*));  
  
 // ask user to input a server port and initialize the server port  
 System.*out*.println("Please enter server port:");  
 *serverPort* = Integer.*parseInt*(typed.readLine());  
  
 // loop to receive the input from client  
 while (typed.readLine() != null) {  
 String operation = null;  
 String value = null;  
 String ID;  
  
 // show the choice to the client  
 System.*out*.println("1. Add a value to your sum.");  
 System.*out*.println("2. Subtract a value from your sum.");  
 System.*out*.println("3. Get your sum.");  
 System.*out*.println("4. Exit client");  
  
 // initialize the operation  
 switch (typed.readLine()) {  
 case "1" -> operation = "add";  
 case "2" -> operation = "subtract";  
 case "3" -> operation = "get";  
 case "4" -> operation = "exit";  
 }  
  
 // if input is 1 or 2, then set the value  
 assert operation != null;  
 if (operation.equals("add")) {  
 System.*out*.println("Enter value to add:");  
 value = typed.readLine();  
 } else if (operation.equals("subtract")) {  
 System.*out*.println("Enter value to subtract:");  
 value = typed.readLine();  
 // if input is 4, then exit the client  
 } else if (operation.equals("exit")) {  
 System.*out*.println("Client side quitting. The remote variable server is still running.");  
 break;  
 }  
  
 // get the client ID  
 System.*out*.println("Enter your ID:");  
 ID = typed.readLine();  
  
 String combined;  
 // combine the ID, operation if operation is "get";  
 // otherwise, combine the ID, operation, and value  
 if (operation.equals("get")) {  
 combined = ID + "," + operation;  
 } else {  
 combined = ID + "," + operation + "," + value;  
 }  
  
 // call the operate function related to client server communications  
 int result = *operate*(combined);  
  
 // show the result to the client  
 System.*out*.printf("The result is %d.\n", result);  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (*aSocket* != null) *aSocket*.close();  
 }  
  
 }  
  
 */\*\*  
 \* client communicates with the server  
 \*  
 \** ***@param*** *combined information including ID, operation, and value(if the operation is other than get)  
 \** ***@return*** *result from the server  
 \*/* public static int operate(String combined) {  
 // define and initialize the result  
 int result = 0;  
 try {  
 // initialize the socket  
 *aSocket* = new DatagramSocket();  
  
 // build a ByteBuffer with capacity 4 to store the int and then return a byte array  
 byte[] m = combined.getBytes();  
  
 // build the packet holding the destination address, port and byte array to contact the server  
 DatagramPacket request = new DatagramPacket(m, m.length, *aHost*, *serverPort*);  
  
 // send the Datagram on the socket  
 *aSocket*.send(request);  
  
 // prepare room for the reply  
 byte[] buffer = new byte[1000];  
  
 // build a datagram for the reply  
 DatagramPacket reply = new DatagramPacket(buffer, buffer.length);  
  
 // block and wait  
 *aSocket*.receive(reply);  
  
 // get bytes arrays from the reply with correct length  
 byte[] reply\_bytes = Arrays.*copyOf*(reply.getData(), reply.getLength());  
  
 // wraps the reply byte array into a buffer and then return the int value  
 result = ByteBuffer.*wrap*(reply\_bytes).getInt();  
  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 }  
  
 return result;  
 }  
  
}

**Project2Task3Server**

// imports required for UDP/IP  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.SocketException;  
import java.nio.ByteBuffer;  
import java.util.Arrays;  
import java.util.Scanner;  
  
public class RemoteVariableServerUDP {  
 // create a global variable for socket  
 static DatagramSocket *aSocket* = null;  
 // create a result array with 1000 size  
 static int[] *result* = new int[1000];  
  
 public static void main(String args[]) {  
 try {  
 // show the announcement  
 System.*out*.println("Server started");  
  
 // prompt the user enter a port number for the server to listen to  
 System.*out*.println("Please enter server port:");  
 int serverPort = new Scanner(System.*in*).nextInt();  
  
 // initialize the socket using port number by user's input  
 *aSocket* = new DatagramSocket(serverPort);  
  
 // loop forever  
 while (true) {  
 // prepare room for the request  
 byte[] buffer = new byte[1000];  
  
 // initialize the request  
 DatagramPacket request = new DatagramPacket(buffer, buffer.length);  
  
 // receive request data from the socket  
 *aSocket*.receive(request);  
  
 // get bytes arrays from the request with correct length  
 byte[] request\_bytes = Arrays.*copyOf*(request.getData(), request.getLength());  
  
 // modify the request data with the correct number of bytes  
 request.setData(request\_bytes);  
  
 // get combined string from the request  
 String combined = new String(request.getData());  
  
 // split the combined information into a string array  
 String[] combined\_array = combined.split(",");  
 int ID = Integer.*parseInt*(combined\_array[0]);  
 String operation = combined\_array[1];  
 System.*out*.println("ID: " + ID);  
  
 //if operation is add or subtract, then do the operation;  
 //if operation is get, then do nothing  
 if (operation.equals("add")) {  
 int value = Integer.*parseInt*(combined\_array[2]);  
 // performs the add operation to the result associated with the ID  
 *result*[ID] = *add*(ID, value);  
  
 // show the add process to the server  
 System.*out*.println("Adding: " + value + " to " + *result*[ID]);  
 } else if (operation.equals("subtract")) {  
 int value = Integer.*parseInt*(combined\_array[2]);  
 // performs the subtract operation to the result associated with the ID  
 *result*[ID] = *subtract*(ID, value);  
  
 // show the subtract process to the server  
 System.*out*.println("Subtracting: " + value + " to " + *result*[ID]);  
 }  
  
 // build a ByteBuffer with capacity 4 to store the result and then return a byte array  
 byte[] sum\_bytes = ByteBuffer.*allocate*(4).putInt(*result*[ID]).array();  
  
 // build the reply to client  
 DatagramPacket reply = new DatagramPacket(sum\_bytes, sum\_bytes.length, request.getAddress(), request.getPort());  
  
 // show the result to the server  
 System.*out*.printf("Returning result of %d to client\n", *result*[ID]);  
  
 // send reply data to socket  
 *aSocket*.send(reply);  
 }  
 // handle socket exceptions  
 } catch (SocketException e) {  
 System.*out*.println("Socket: " + e.getMessage());  
 // handle general I/O exceptions  
 } catch (IOException e) {  
 System.*out*.println("IO: " + e.getMessage());  
 } finally {  
 // always close the socket  
 if (*aSocket* != null) *aSocket*.close();  
 }  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *ID for choosing the result array's index  
 \** ***@param*** *value for add  
 \** ***@return*** *result after operating  
 \*/* public static int add(int ID, int value) {  
 *result*[ID] += value;  
 return *result*[ID];  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *ID for choosing the result array's index  
 \** ***@param*** *value for subtract  
 \** ***@return*** *result after operating  
 \*/* public static int subtract(int ID, int value) {  
 *result*[ID] -= value;  
 return *result*[ID];  
 }  
  
}

**Project2Task3ClientConsole**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Project2Task3ServerConsole**

Text

Description automatically generated with low confidence

**Project2Task4Client**

import java.net.\*;  
import java.io.\*;  
import java.nio.ByteBuffer;  
import java.util.Arrays;  
  
public class RemoteVariableClientTCP {  
 // create global variables for read from and write to the socket  
 static Socket *clientSocket*;  
 static BufferedReader *in* = null;  
 static PrintWriter *out* = null;  
  
 public static void main(String args[]) {  
 try {  
 // show the announcement  
 System.*out*.println("The client is running.");  
  
 InetAddress aHost = InetAddress.*getByName*("localhost");  
  
 // read user's input  
 BufferedReader typed = new BufferedReader(new InputStreamReader(System.*in*));  
  
 // ask user to input a server port and initialize the server port  
 System.*out*.println("Please enter server port:");  
 int serverPort = Integer.*parseInt*(typed.readLine());  
  
 // initialize the socket  
 *clientSocket* = new Socket(aHost, serverPort);  
  
 // loop to receive the input from client  
 while (typed.readLine() != null) {  
 String operation = null;  
 String value = null;  
 String ID;  
  
 // show the choice to the client  
 System.*out*.println("1. Add a value to your sum.");  
 System.*out*.println("2. Subtract a value from your sum.");  
 System.*out*.println("3. Get your sum.");  
 System.*out*.println("4. Exit client");  
  
 // initialize the operation  
 switch (typed.readLine()) {  
 case "1" -> operation = "add";  
 case "2" -> operation = "subtract";  
 case "3" -> operation = "get";  
 case "4" -> operation = "exit";  
 }  
  
 // if input is 1 or 2, then set the value  
 assert operation != null;  
 if (operation.equals("add")) {  
 System.*out*.println("Enter value to add:");  
 value = typed.readLine();  
 } else if (operation.equals("subtract")) {  
 System.*out*.println("Enter value to subtract:");  
 value = typed.readLine();  
 // if input is 4, then exit the client  
 } else if (operation.equals("exit")) {  
 System.*out*.println("Client side quitting. The remote variable server is still running.");  
 break;  
 }  
  
 // get the client ID  
 System.*out*.println("Enter your ID:");  
 ID = typed.readLine();  
  
 String combined;  
 // combine the ID, operation if operation is "get";  
 // otherwise, combine the ID, operation, and value  
 if (operation.equals("get")) {  
 combined = ID + "," + operation;  
 } else {  
 combined = ID + "," + operation + "," + value;  
 }  
  
 // call the operate function related to client server communications  
 int result = *operate*(combined);  
  
 // show the result to the client  
 System.*out*.printf("The result is %d.\n", result);  
 }  
 } catch (IOException e) {  
 System.*out*.println("IO Exception:" + e.getMessage());  
 } finally {  
 try {  
 if (*clientSocket* != null) {  
 *clientSocket*.close();  
 }  
 } catch (IOException e) {  
 // ignore exception on close  
 }  
 }  
 }  
  
 */\*\*  
 \* client communicates with the server  
 \*  
 \** ***@param*** *data information including ID, operation, and value(if the operation is other than get)  
 \** ***@return*** *result from the server  
 \*/* public static int operate(String data) {  
 // define and initialize the result  
 int result = 0;  
 try {  
 // initialize the read and write for socket  
 *in* = new BufferedReader(new InputStreamReader(*clientSocket*.getInputStream()));  
 *out* = new PrintWriter(new BufferedWriter(new OutputStreamWriter(*clientSocket*.getOutputStream())));  
  
 // send message to the server  
 *out*.println(data);  
 *out*.flush();  
  
 // read a line of data from the stream  
 result = Integer.*parseInt*(*in*.readLine());  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return result;  
 }  
}

**Project2Task4Server**

import java.net.\*;  
import java.io.\*;  
import java.util.Scanner;  
  
public class RemoteVariableServerTCP {  
 // create a result array with 1000 size  
 static int[] *result* = new int[1000];  
  
 public static void main(String args[]) {  
 Socket clientSocket = null;  
 try {  
 // show the announcement  
 System.*out*.println("Server started");  
  
 // prompt the user enter a port number for the server to listen to  
 System.*out*.println("Please enter server port:");  
 int serverPort = new Scanner(System.*in*).nextInt();  
  
 // Create a new server socket  
 ServerSocket listenSocket = new ServerSocket(serverPort);  
  
 /\*  
 \* Block waiting for a new connection request from a client.  
 \* When the request is received, "accept" it, and the rest  
 \* the tcp protocol handshake will then take place, making  
 \* the socket ready for reading and writing.  
 \*/  
 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())));  
  
 // loop forever to accept the request and return the result to the client  
 while (true) {  
 if (in.hasNextLine()) {  
 // read the data from the client  
 String data = in.nextLine();  
 System.*out*.println("Echoing: " + data);  
  
 // split the combined information into a string array  
 String[] combined\_array = data.split(",");  
 int ID = Integer.*parseInt*(combined\_array[0]);  
 String operation = combined\_array[1];  
 System.*out*.println("ID: " + ID);  
  
 //if operation is add or subtract, then do the operation;  
 //if operation is get, then do nothing  
 if (operation.equals("add")) {  
 int value = Integer.*parseInt*(combined\_array[2]);  
 // performs the add operation to the result associated with the ID  
 *result*[ID] = *add*(ID, value);  
  
 // show the add process to the server  
 System.*out*.println("Adding: " + value + " to " + *result*[ID]);  
 } else if (operation.equals("subtract")) {  
 int value = Integer.*parseInt*(combined\_array[2]);  
 // performs the subtract operation to the result associated with the ID  
 *result*[ID] = *subtract*(ID, value);  
  
 // show the subtract process to the server  
 System.*out*.println("Subtracting: " + value + " to " + *result*[ID]);  
 }  
  
 // show the result to the server  
 System.*out*.printf("Returning result of %d to client\n", *result*[ID]);  
  
  
 // send message to the client  
 out.println(*result*[ID]);  
 out.flush();  
 } else {  
 // continue to accept requests after previous connection is closed  
 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  
 }  
 }  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *ID for choosing the result array's index  
 \** ***@param*** *value for add  
 \** ***@return*** *result after operating  
 \*/* public static int add(int ID, int value) {  
 *result*[ID] += value;  
 return *result*[ID];  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *ID for choosing the result array's index  
 \** ***@param*** *value for subtract  
 \** ***@return*** *result after operating  
 \*/* public static int subtract(int ID, int value) {  
 *result*[ID] -= value;  
 return *result*[ID];  
 }  
  
}

**Project2Task4ClientConsole**

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Project2Task4ServerConsole**

Text

Description automatically generated with medium confidence

**Project2Task5Client**

import java.io.\*;  
import java.lang.reflect.Array;  
import java.math.BigInteger;  
import java.net.InetAddress;  
import java.net.Socket;  
import java.security.MessageDigest;  
import java.security.NoSuchAlgorithmException;  
import java.util.Arrays;  
  
public class SigningClientTCP {  
 // create global variables for read from and write to the socket  
 static Socket *clientSocket*;  
 static BufferedReader *in* = null;  
 static PrintWriter *out* = null;  
  
 public static void main(String args[]) {  
 // define the client's ID  
 BigInteger ID;  
 try {  
 // show the announcement  
 System.*out*.println("The client is running.");  
  
 InetAddress aHost = InetAddress.*getByName*("localhost");  
  
 // read user's input  
 BufferedReader typed = new BufferedReader(new InputStreamReader(System.*in*));  
  
 // ask user to input a server port and initialize the server port  
 System.*out*.println("Please enter server port:");  
 int serverPort = Integer.*parseInt*(typed.readLine());  
  
 // initialize the socket  
 *clientSocket* = new Socket(aHost, serverPort);  
  
 // create new RSA public and private keys, and initialize e, d, n  
 BigInteger[] pairs = RSAAlgorithm.*createKeys*();  
 BigInteger e = pairs[0];  
 BigInteger d = pairs[1];  
 BigInteger n = pairs[2];  
 String public\_key = e + "," + n;  
 String private\_key = d+ "," + n;  
 // display these keys to the user  
 System.*out*.println("public key: " + public\_key);  
 System.*out*.println("private key: " + private\_key);  
  
 // concatenate e and n to generate a public key  
 // and then hash the client's public key  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
 md.update(public\_key.getBytes());  
  
 // take the last 20 bytes of the hash to generate client's ID  
 ID = new BigInteger(Arrays.*copyOfRange*(md.digest(), md.digest().length-20, md.digest().length));  
  
 System.*out*.println("ID: " + ID);  
  
 // loop to receive the input from client  
 while (typed.readLine() != null) {  
 String operation = null;  
 String operand = null;  
  
 // show the choice to the client  
 System.*out*.println("1. Add a value to your sum.");  
 System.*out*.println("2. Subtract a value from your sum.");  
 System.*out*.println("3. Get your sum.");  
 System.*out*.println("4. Exit client");  
  
 // initialize the operation  
 switch (typed.readLine()) {  
 case "1" -> operation = "add";  
 case "2" -> operation = "subtract";  
 case "3" -> operation = "get";  
 case "4" -> operation = "exit";  
 }  
  
 // if input is 1 or 2, then set the value  
 if (operation.equals("add")) {  
 System.*out*.println("Enter value to add:");  
 operand = typed.readLine();  
 } else if (operation.equals("subtract")) {  
 System.*out*.println("Enter value to subtract:");  
 operand = typed.readLine();  
 // if input is 4, then exit the client  
 } else if (operation.equals("exit")) {  
 System.*out*.println("Client side quitting. The remote variable server is still running.");  
 break;  
 }  
  
 // define message, signature, and data sent to the server  
 String data;  
 String message;  
 String signature;  
  
 // combine message m = id, the public key (e and n), the operation, the operand  
 message = ID + "," + public\_key + "," + operation + "," + operand;  
// System.out.println("Message :" + message);  
  
 // use MessageSign function  
 MessageSign sov = new MessageSign(e, d, n);  
  
 // set the leftmost byte as 0 makes the value positive.  
 // RSA only works with positive values  
  
 signature = sov.sign(message);  
// System.out.println("Signature :" + signature);  
  
 // combine message and signature  
 data = message + "," + signature;  
  
 // call the operate function related to client server communications  
 String result = *operate*(data);  
  
 // show the result to the client  
 if (!result.equals("Error in request")) {  
 System.*out*.printf("The result is %s.\n", result);  
 } else {  
 System.*out*.println(result);  
 }  
  
 }  
 } catch (IOException e) {  
 System.*out*.println("IO Exception:" + e.getMessage());  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (*clientSocket* != null) {  
 *clientSocket*.close();  
 }  
 } catch (IOException e) {  
 // ignore exception on close  
 }  
 }  
 }  
  
 */\*\*  
 \* client communicates with the server  
 \*  
 \** ***@param*** *data information including ID, operation, and value(if the operation is other than get)  
 \** ***@return*** *result from the server  
 \*/* public static String operate(String data) {  
 // define and initialize the result  
 String result = null;  
 try {  
 // initialize the read and write for socket  
 *in* = new BufferedReader(new InputStreamReader(*clientSocket*.getInputStream()));  
 *out* = new PrintWriter(new BufferedWriter(new OutputStreamWriter(*clientSocket*.getOutputStream())));  
  
 // send message to the server  
 *out*.println(data);  
 *out*.flush();  
  
 // read a line of data from the stream  
  
 result = *in*.readLine();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 return result;  
 }  
}

**Project2Task5Server**

import java.math.BigInteger;  
import java.net.\*;  
import java.io.\*;  
import java.security.MessageDigest;  
import java.util.Arrays;  
import java.util.Scanner;  
  
public class VerifyingServerTCP {  
 // create a result array with 1000 size  
 static int *result* = 0;  
  
 public static void main(String args[]) {  
 Socket clientSocket = null;  
 try {  
 // show the announcement  
 System.*out*.println("Server started");  
  
 // prompt the user enter a port number for the server to listen to  
 System.*out*.println("Please enter server port:");  
 int serverPort = new Scanner(System.*in*).nextInt();  
  
 // Create a new server socket  
 ServerSocket listenSocket = new ServerSocket(serverPort);  
  
 /\*  
 \* Block waiting for a new connection request from a client.  
 \* When the request is received, "accept" it, and the rest  
 \* the tcp protocol handshake will then take place, making  
 \* the socket ready for reading and writing.  
 \*/  
 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())));  
  
 // loop forever to accept the request and return the result to the client  
 while (true) {  
 if (in.hasNextLine()) {  
 // read the data from the client  
 String data = in.nextLine();  
 System.*out*.println("Echoing: " + data);  
  
 // split the combined information into a string array  
 String[] data\_array = data.split(",");  
  
 // extract information  
 BigInteger ID = new BigInteger(data\_array[0]);  
 BigInteger e = new BigInteger(data\_array[1]);  
 BigInteger n = new BigInteger(data\_array[2]);  
 String public\_key = e + "," + n;  
 System.*out*.println("public key: " + public\_key);  
 String operation = data\_array[3];  
 int operand = 0;  
 String ori\_message;  
 if (! operation.equals("get")) {  
 operand = Integer.*parseInt*(data\_array[4]);  
 // get the original message sent by the client  
 ori\_message = ID + "," + e + "," + n + "," + operation + "," + operand;  
 } else {  
 ori\_message = ID + "," + e + "," + n + "," + operation + "," + null;  
 }  
  
 // get the signature  
 String signature = data\_array[5];  
  
 // check if the public key hash to the ID  
 boolean isSameID;  
 // hash the client's public key  
 MessageDigest md = MessageDigest.*getInstance*("SHA-256");  
 md.update(public\_key.getBytes());  
  
 // take the last 20 bytes of the hash of public ket to generate client's ID  
 BigInteger check\_ID = new BigInteger(Arrays.*copyOfRange*(md.digest(), md.digest().length-20, md.digest().length));  
 isSameID = check\_ID.equals(ID);  
 System.*out*.println("Is ID verified: " + isSameID);  
  
 // check if the request properly signed  
 boolean isSignedValid;  
 MessageVerify verifySig = new MessageVerify(e, n);  
 isSignedValid = verifySig.verify(ori\_message, signature);  
 System.*out*.println("Is signature verified: " + isSignedValid);  
  
  
 // if verify correctly, then the server do the operation;  
 // otherwise, the server returns the message "Error in request".  
 if (isSameID && isSignedValid) {  
 //if operation is add or subtract, then do the operation;  
 //if operation is get, then do nothing  
 if (operation.equals("add")) {  
 // performs the add operation to the result associated with the ID  
 *result* = *add*(operand);  
 // show the add process to the server  
 System.*out*.println("Adding: " + operand + " to " + *result*);  
 } else if (operation.equals("subtract")) {  
 // performs the subtract operation to the result associated with the ID  
 *result* = *subtract*(operand);  
 // show the subtract process to the server  
 System.*out*.println("Subtracting: " + operand + " to " + *result*);  
 } else if (operation.equals("get")) {  
 // show the subtract process to the server  
 System.*out*.println("Getting: " + *result*);  
 }  
  
 // show the result to the server  
 System.*out*.printf("Returning result of %d to client\n", *result*);  
  
 // send message to the client  
 out.println(*result*);  
 out.flush();  
 } else {  
 // send error message to the client  
 out.println("Error in request");  
 out.flush();  
 }  
 } else {  
 // continue to accept requests after previous connection is closed  
 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  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (clientSocket != null) {  
 clientSocket.close();  
 }  
 } catch (IOException e) {  
 // ignore exception on close  
 }  
 }  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *value for add  
 \** ***@return*** *result after operating  
 \*/* public static int add(int value) {  
 *result* += value;  
 return *result*;  
 }  
  
 */\*\*  
 \* performs the add or subtract operation.  
 \*  
 \** ***@param*** *value for subtract  
 \** ***@return*** *result after operating  
 \*/* public static int subtract(int value) {  
 *result* -= value;  
 return *result*;  
 }  
  
}

**Project2Task5ClientConsole**

Text

Description automatically generated

**Project2Task5ServerConsole**

Text

Description automatically generated