

UDP Communication Establishment for Retrieving Temperature Data from Server to a Client

Tanvi Sharma
Swanad Bhawe

Professor Zoltan Safar

ENTS 640 SECTION 0101

December 4, 2016

University of Maryland, College Park

Contents

1. Introduction.....	3
2. Problem statement	
2.1 Client.....	3
2.2 Server	4
3. UML Diagram	
3.1 UML diagram of the client	5
3.2 UML diagram of the server.....	5
4. Block diagram	6
5. Flow chart	
5.1 client.....	7
5.2 server.....	8
6. Implementation of the code	
6.1 Client.....	9
6.2 Server	11
7. Operation and Output.....	13

1. Introduction

The purpose of this project is to introduce the students to the UDP socket programming. As we all know UDP is an unreliable, packet switch data transfer protocol. The project aims to make it more reliable by integrating integrity check in the response and request message and providing a retransmission when time out occurs.

2. Problem Statement:

The project requires us to establish a UDP communication channel between a server and client. UDP is an unreliable data transfer service so, to guarantee reliability few measures are integrated into the application. The first measure is to include an integrity check (using checksum algorithm) in every request and response which is verified at the server and client respectively. The second measure is to ensure retransmission in two scenarios, after a timeout and when the user gives a positive response to retransmission after the check sum fails on the server side.

2.1 Client:

A random measurement ID is fetched from the measurement ID array. The program also generates a random request ID. After the values are generate they are converted into a string format and embedded in the request message format as shown below.

Request Message from the client

```
<request>
  <id>7542</id>
<measurement>2851</measurement> </request>
```

This request message string is trimmed and forwarded as an argument in the checksum function. The checksum string generated from the checksum function is added at the end of the request message.

```
<request>
  <id>7542</id>
<measurement>2851</measurement> </request>
4356
```

Once the request message is send to the server, timer starts and the client waits for the response. If the timeout occur the timer value is doubled and the message is send again. If the timeout occurs more than 4 times the process is terminated. However, if the response message is received its checksum value is verified.

If the checksum value in the message does not match the checksum value calculated by the client, the program breaks the while loop and generates the request message again.

If checksum values match, then the code value is extracted from the response message.

- For code value is 1, the user is asked “if they want to send the message again”. Once the user confirms, the same message is send again else the it breaks the while loop and generates the next request message again.
- For code value 2 and, 3 the error messages of syntax error and nonexistence measurement ID respectively are printed on the output screen and the request message is generated again.
- For coed 0 the temperature value is extracted from the response message and printed on the output screen and the next request message is generated.

This process goes on till the user terminates it externally or it reaches a fourth time out.

2.2 Server:

The server continuously waits for the request message from the client on a specific port.

Once the request message is received, check sum value is compared with the one on the request message and the one calculated on the server. If the checksum values do not match the Code value is set to 1.

If the checksum values matches, then the syntax is verified. It is done by the removing all the value from the request message and compared to the syntax string. If there is a syntax error code value is set to 2.

Syntax string: `<request> <id></id><measurement></measurement></request>`

If the checksum and syntax doesn't have any error the measurement ID is extracted and compared to the measurement ID on the data file on the server. If the measure is not found the Code value is set to 3.

If, however, the measure ID matches then its respective temperature in the data file is saved in a variable and the code value is set to 0.

If the code value is 0: Response message is generated with the response ID and measurement ID same as the Request Id. The temperature value is also inserted as shown below:

Response message code value is 0 (when no error is detected)

```
<response>
  <id>7542</id>
<code>0</code> <measurement>2851</measurement> <value>98.30</value>
</response>
748
```

If the code values are 1, 2 or 3 the below response message format is generated.

Response message when code value is 1,2 or 3 (when there is checksum, Syntax or measurement ID error):

```
<response>
  <id>9999</id>
  <code>3</code>
</response>
4675
```

3. UML diagram

3.1 Server UML diagram:

SERVER
+Syntaxcheck (String Syntax, String id, String mid): Boolean
+Checksumcal (String str): String
+Main():void

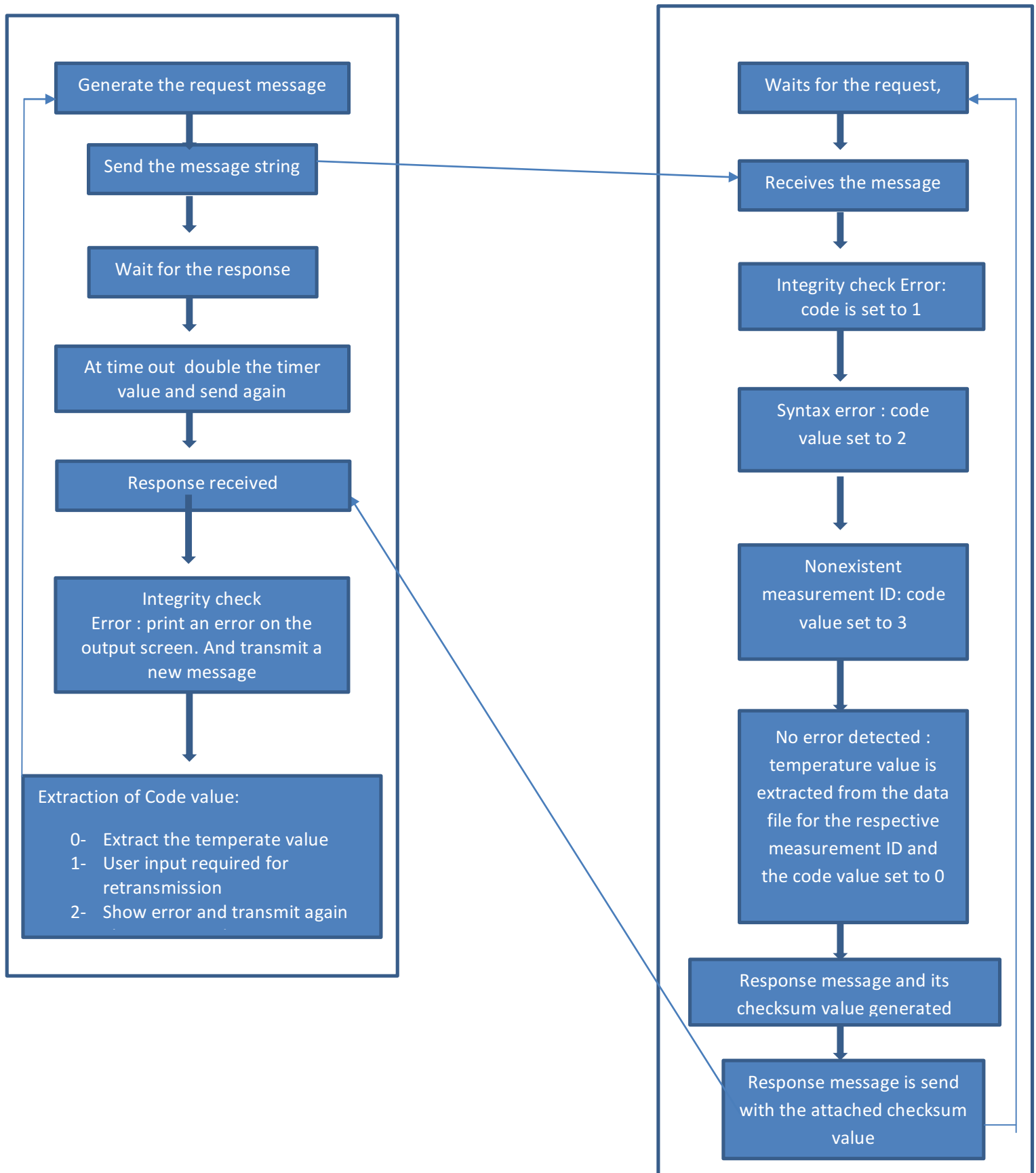
3.2 Client UML diagram:

CLIENT
+Checksumcal (String str): String
+Main():void

4. Block Diagram:

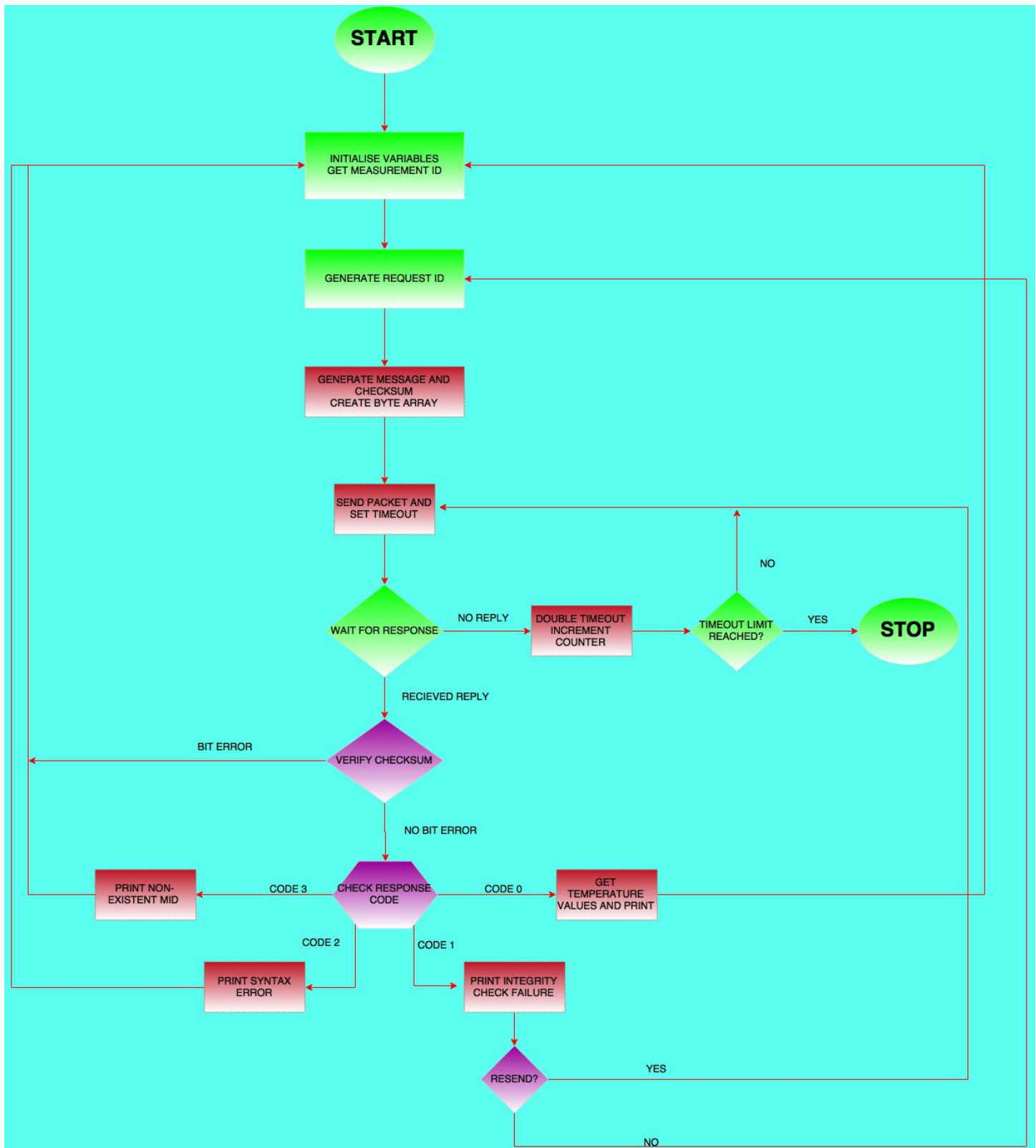
CLIENT

SERVER

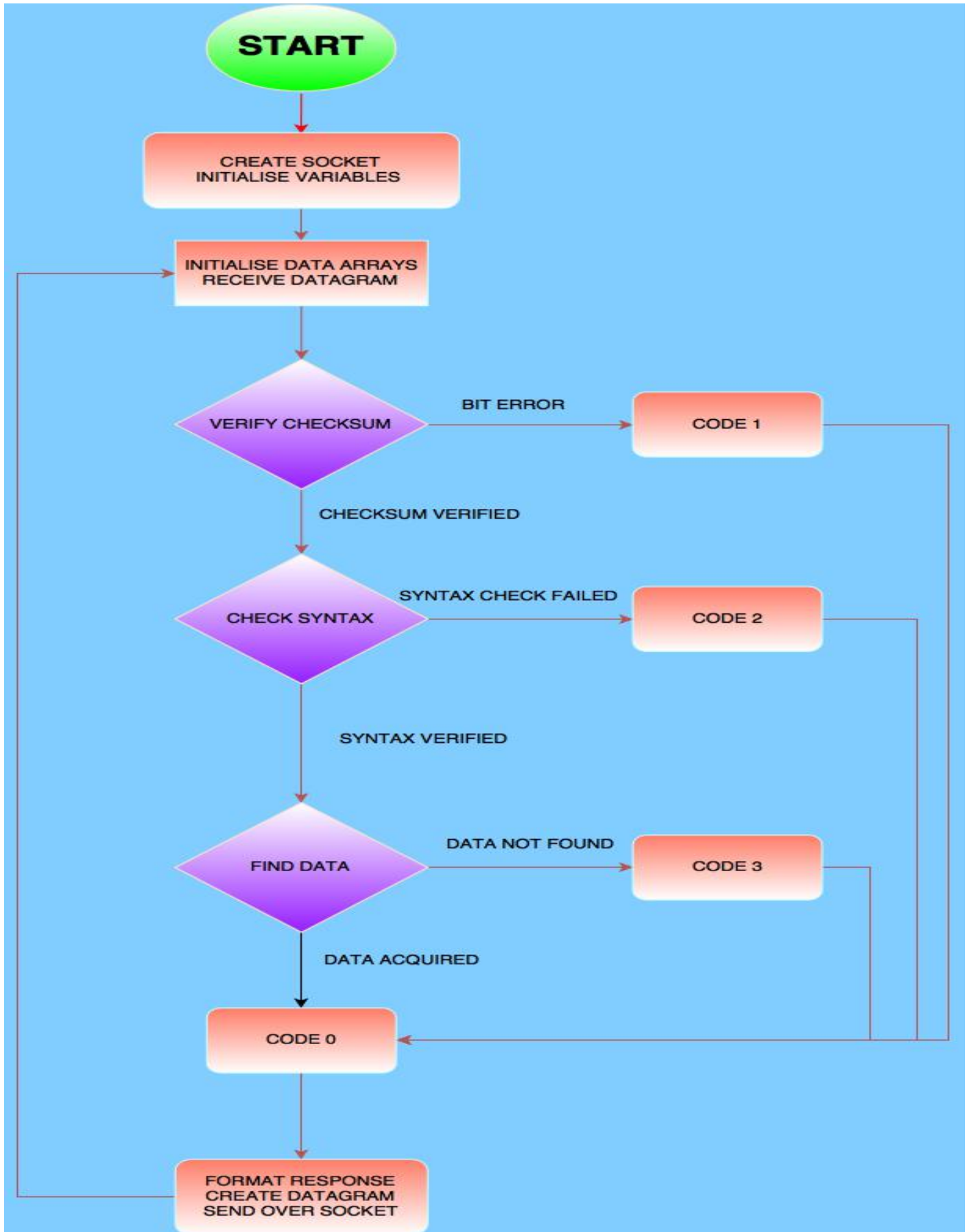


5. Flow chart:

5.1 Client:



5.2 Server:



6. Implementation of the Code:

6.1 Server application:

SERVER
Syntaxcheck(String Syntax, String id, String mid): Boolean
Checksumcal(String str): String
Main():void

The server application includes three methods, Syntaxcheck ,checksumCal and the main method. To implement the server functionality, the following code segments were used.

Main method:

The byte arrays to be used for storing the data for sending in the datagram and processing, which are declared with an arbitrary range of 1024. The datagram packet for storing the received packet from the client is declared and byte arrays are assigned to it. The string object is obtained from the received byte array by passing the array to the string class constructor. We use the split method of the string class to extract the values of the request ID, checksum, and the measurement ID from the thus obtained string.

E.g

```
String[] id1 = receivedData.split("</id>");
String temp = id1[0];
String[] id2 = temp.split("<id>");
String id = id2[1]
```

The first step is to call the *checksumcal* method on the string that is the received string without the checksum value at the end of it. The program proceeds with further processing if the checksum is verified with the value in the received string. If the checksum verification fails the code variable is set to 1. To verify whether the received request is in the proper syntax we first prepare the string for syntax check. The string is stripped of the numeric characters to be tested separately for range.

```
syntax=syntax.replaceAll(cs1[1], "");
syntax=syntax.replaceAll(mid, "");
syntax=syntax.replaceAll(id, "");
```

If the syntax check is passed the program proceeds to extract the measurement value from the file, if not the code value is set to 2. The Buffered Reader class is used to read the file that is stored on the server. The measurement ID obtained by the split method is crossed with values in the file to acquire the corresponding temperature value. The search algorithm is implemented by using the split method again on each line of the file with the splitting substring set to a tab. If after running the search the temperature variables value is still zero, that indicates failure to match the measurement ID. Hence the code is set to 3 i.e. invalid measurement ID. The response string is formulated based on two cases. If the code is 0 that means the response is standard and all tags and the temperature value is included. Otherwise the string is formatted for an error response.

The *checksumcal* method is invoked on the string to generate the checksum and it is appended to the response string. The datagram packet is created with the byte array conversion of the response string and sent over the socket. the port number and address is extracted from the received packet to route it back towards the client.

Checksumcal method:

The *checksumcal* method takes a string input for calculating the checksum. The method returns a string which is the checksum value calculated on the input parameter. The string is first checked for even or odd length. If the string has an odd length, 0 is added to the end of the string. The *toHexString* method of the Integer class is used to obtain the 16-bit unsigned value of the characters at index I and i+1 to form the 16-bit numbers

The 16-bit words are added according the given algorithm to calculate the checksum

```
for(i=0;i<str1.length()-1;i++){
String buffstr=Integer.toHexString((int)str1.charAt(i))+Integer.toHexString((int)str1.charAt(i+1));
buff=Integer.parseInt(buffstr, 16);
s=Integer.parseInt(Integer.toHexString(s), 16);
index=s^buff;
s=(c*index)%d;
```

the calculated checksum is returned as a string using the *toString* method of the Integer class.

Syntaxcheck method:

The *syntaxcheck* method is a Boolean method that accepts three parameters. The strings syntax, request ID and measurement ID. The method checks the string syntax for order and tags by comparing it to a default template that we declare with the correct order and tags.

The request ID and the measurement ID are checked if the are 16-bit unsigned integers. The Boolean value is returned based on the satisfaction of the conditional statements.

```
if(s1.matches(syntax)){
if(id1>=0&&id1<=65535){
if(mid1>=0&&mid1<=65535){
return true;
}}
return false;
```

6.2 Client application:

CLIENT
Checksumcal(String str): String Main():void

The client application has two methods, the main method and the *checksumcal* method.

Main method:

The whole application has to run until the user terminate the application, hence the entire main method is in a while loop that runs infinitely. Generation of random request ID is implemented using *Math.random()* function. We declared an array containing all the measurement IDs to select a measurement ID. After the generation of the measurement and request ID we construct the request message and calculate the checksum of the message. The checksum is appended to the request message.

Datagram packets are created for the request and response messages and a socket for transmission. The next while loop is for implementing retransmission in case of packet loss and other cases. Two counters are used to implement retransmission, *i* and *ret*. The packet is sent over the socket for timeout *t0*.

In the try block we write the receive statement. The counters are set/reset to zero if the packet is received. We check the reception of the packet by examining the byte array received as data. The first step in processing the response packet is checksum verification. Checksum verification is implemented by invoking the *checksumcal* method on the string obtained from the received byte array after stripping the checksum value at the end of the string

If checksum is verified the program proceeds to check the code values, else retransmission is engaged. The code value is obtained by using the split method of the string class. The code value is processed with the switch case statement.

```
String[] temp= responseMessageString.split("</code>");
String[] code=temp[0].split("<code>");
int codeValue=Integer.parseInt(code[1]);

switch(codeValue)
{
    case 0:
        String[] temp1 = responseMessageString.split("</value>");
        String[] temperatureValue=temp1[0].split("<value>");
        System.out.println("the temperature is: "+temperatureValue[1]);
        break;
    case 1: System.out.println("error. integrity check failure");
        Scanner in = new Scanner(System.in);
        System.out.println("Do you want to send the request again(Y/N): ");
        String userResponse = in.next();
        if(userResponse.equals("Y"))
```

```
        {ret++;  
        break;}  
        else {break;}  
    case 2: System.out.println("error. syntax of request is incorrect");  
        break;  
    case 3: System.out.println("error. nonexistent measurement id.");  
        break;  
}
```

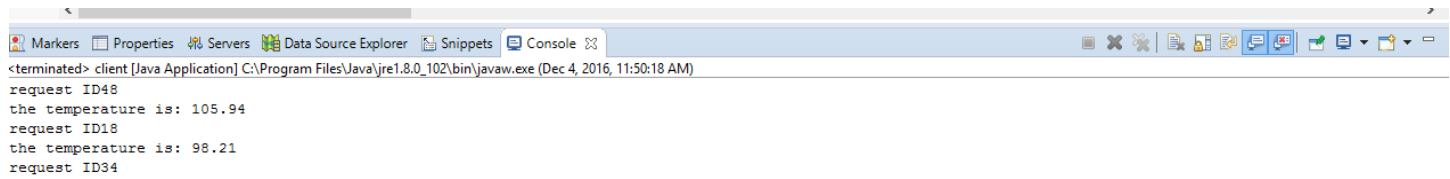
The *ret* counter is incremented in case of positive user input for the case of code 1 so that the same request message is resent. The *ret* variable is used to break out of the retransmission loop when necessary.

The catch block is used to check the timeout limit and modify the counters and timeout value for retransmission

```
catch(IOException e)  
{  
    if(i==3)  
        {System.out.println("timeout reached");}  
        else { to=to*2;  
                ret++;  
            }  
    i+=1;}  
}
```

OPERATION AND OUTPUTS

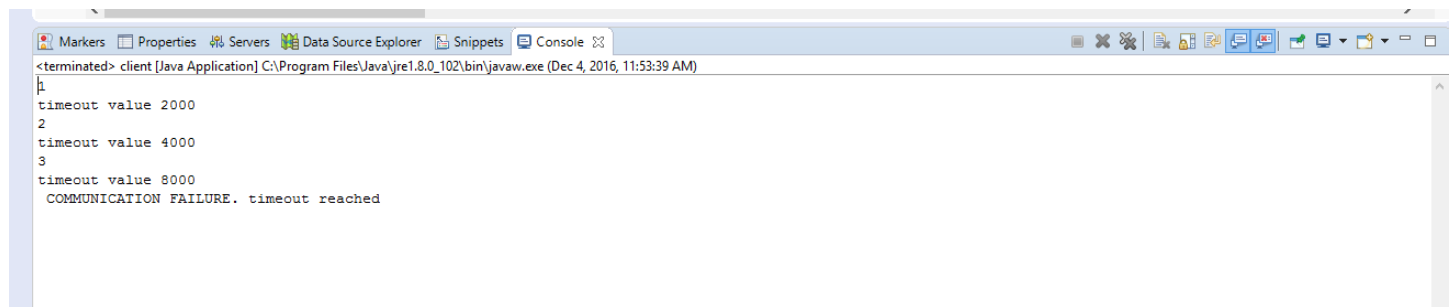
CASE 1: NORMAL OPERATION: NO ERROR



The screenshot shows an IDE console window with the following text:

```
<terminated> client [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Dec 4, 2016, 11:50:18 AM)
request ID48
the temperature is: 105.94
request ID18
the temperature is: 98.21
request ID34
```

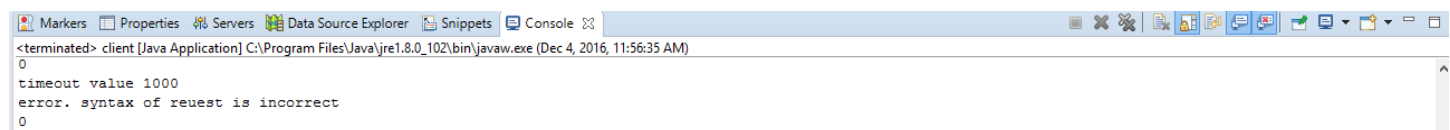
CASE 2: TIMEOUT



The screenshot shows an IDE console window with the following text:

```
<terminated> client [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Dec 4, 2016, 11:53:39 AM)
1
timeout value 2000
2
timeout value 4000
3
timeout value 8000
COMMUNICATION FAILURE. timeout reached
```

CASE 3: SYNTAX ERROR



The screenshot shows an IDE console window with the following text:

```
<terminated> client [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Dec 4, 2016, 11:56:35 AM)
0
timeout value 1000
error. syntax of request is incorrect
0
```

CASE 4: INTEGRITY ERROR



The screenshot shows an IDE console window with the following text:

```
client [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (Dec 4, 2016, 11:59:27 AM)
error. integrity check failure
Do you want to send the request again (Y/N):
N
0
timeout value 1000
```