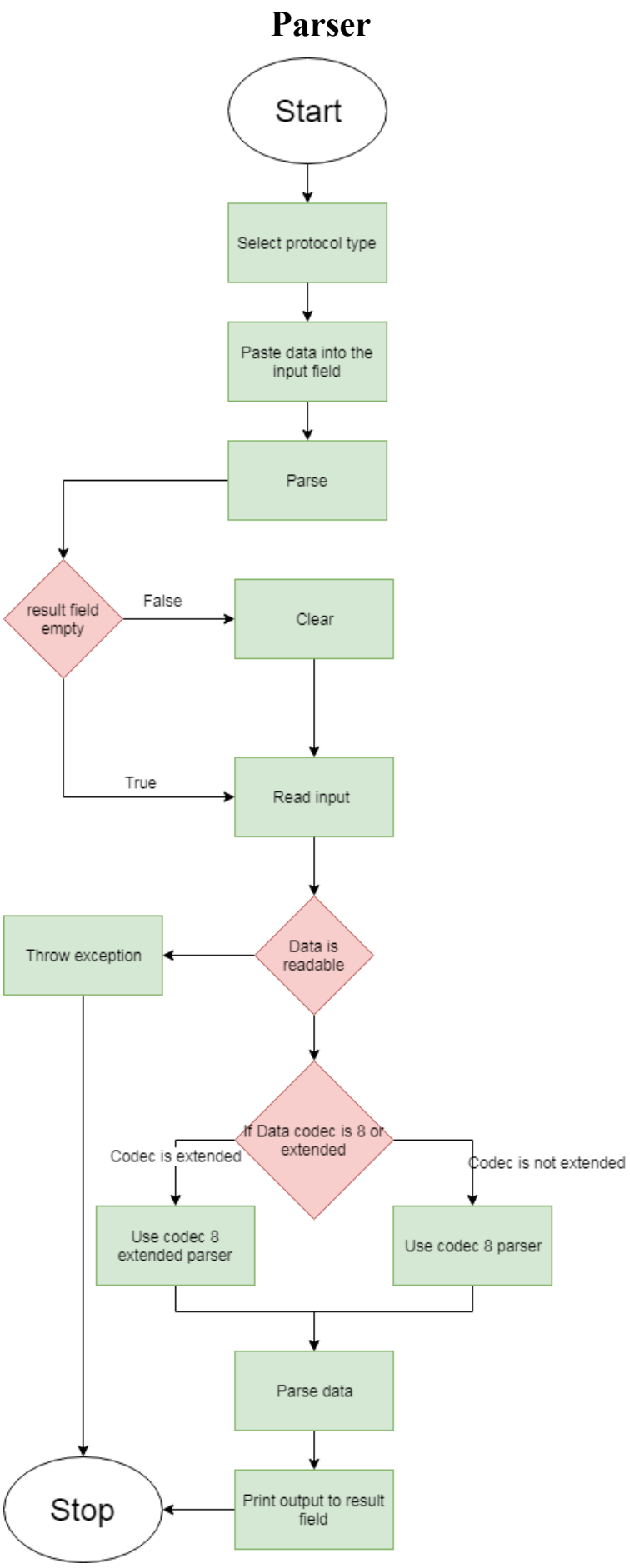

Illustration 1: Flow char of parser

```java
/**
 * <h1>Function to start parsing process</h1>
 * <p>It checks protocol which one is selected and starts parsing according to the selected one.</p>
 * */
public void ParseStart()
{

    System.out.println("selected: "+ ProtocolTypeCb.getValue());
    if(ProtocolTypeCb.getValue()!=null)
    {
        String text = InputArea.getText().trim();
        parser = new Parser( ProtocolType: ""+ ProtocolTypeCb.getValue(),Console);
        parser.ClearAVLRecordCollection();
        Console.setText("");
        parser.ParseData(text);

    }
}
```

Class ParserView

When we enter the data in input text area, and select protocol type, we then press "*Parse*" button to start parsing. The first method that initializes parsing process is in *ParserView* class. The method checks if protocol is selected and proceeds to removing white spaces (trim). Clears any leftovers from previous parsing and calls *ParseData* method to begin parsing.

```java
/**
 * <h1>Parse Data</h1>
 * <p>Function uses switch to check what protocol type is selected to start parsing</p>
 * @param hex parameter used for reading and parsing data
 */
public void ParseData(String hex) {
    if (!empty(hex)) {
        HEX = hex;
        switch (ProtocolType) {
            case "TCP":
                TCPType();
                break;
            case "UDP":
                UDPType();
                break;
            case "UNKNOWN":
                UnknownType(hex);
                break;
            default:
                System.out.println("Protocol is invalid");
                break;
        }
        if(avlRecordCollection != null)
        PrintRecordToResultTA();
    } else {
        System.err.println("HEX value is incorrect or null!");
    }
}
```

Parse Data method in Parser class

The method *ParseData* uses switch statement to check which protocol is selected and receives a hex from text area input. Calls out the parser for the right data. If protocol is not selected a simple default will be called and in console will say "*Protocol is invalid*".

```java
/**
 * <h1>TCP type</h1>
 * <p>Function calls to get number of records to create records and finally puts to Collections</p>
 */
private void TCPType() {
    TCPCalcRecords();
    for (int i = 0; i < recordC; i++) {
        CreateRecord();
    }
    System.out.println(" Records :" + recordC);
    AssignCollection();
}
```

TCP type method in Parser class

TCPType method simply calls other few methods, first it calls a method to calculate the number of records in data. After that it runs over a for loop to create records, once its finished it assigns records to collection for further usages.

```java
/**
 * <h1>TCP Calculate Records</h1>
 * <p>A function reads and header of given HEX to get number of records</p>
 */
private void TCPCalcRecords() {
    if (HEX.length() >= 90) {
        String preamble = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 8)));
        HEX = HEX.substring(8, HEX.length());
        String avlDataLength = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 8)));
        HEX = HEX.substring(8, HEX.length());

        Console.appendText("\nPreamble       : " + preamble);
        Console.appendText("\navl Data Length       : " + avlDataLength);
        codec = converter.convertStringToIntHex(HEX.substring(0, 2));
        if (codec <= 142) {
            if (codec == 142){
                n = 4;
            }
            else{
                n = 2;
            }
            System.out.println("Codec: " + codec);
            recordC = converter.convertStringToIntHex(HEX.substring(2, 4));
            HEX = HEX.substring(4, HEX.length());
        }
        else {
            ShowMessage( header: "Parse Error",  text: "Data is corrupted or codec is wrong");
        }
    } else {
        ShowMessage( header: "Parse Error",  text: "Data is corrupted or codec is wrong");
    }
}
```

TCP Calculate Records method in Parser class

The method is called from *TCPType* to get information about data that is about to be parsed. It checks if data is even has at least 90 characters, if validation passes the information gathering begins, first of it starts with *preamble*, later on with *avl data* length. The HEX is being cut down and every time we parse something from it we cut it and make it shorter, we do this until no data to read is left.

After getting header it parses the codec and checks if its less than 10, other than that it will not parse. Once we pass the codec if statement, we collect the records count by simple parsing it from data.

Added if statement to determent if data is codec 8 or codec 8 extended, if codec is extended we simply will be parsing the IO data by twice byte length (instead of 1 byte - 2 bytes etc.). Same goes with UDP.

```java
/**
 * <h1>UDP Calculate Records</h1>
 * <p>A function reads and header of given HEX to get number of records</p>
 */
private void UDPCalcRecords() {
    if (HEX.length() >= 100) {
        String PacketLength = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 4)));
        HEX = HEX.substring(4, HEX.length());
        String PacketIdentification = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 4)));
        HEX = HEX.substring(4, HEX.length());
        String PacketType = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 2)));
        HEX = HEX.substring(2, HEX.length());
        String PacketId = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 2)));
        HEX = HEX.substring(2, HEX.length());
        String ImeiLength = String.format("%d", converter.convertStringToIntHex(HEX.substring(0, 4)));
        HEX = HEX.substring(4, HEX.length());
        if (Integer.parseInt(ImeiLength) < 20) {
            Console.appendText("\nPacketLength      : " + PacketLength);
            Console.appendText("\nPacket Identification      : " + PacketIdentification);
            Console.appendText("\nPacket Type       : " + PacketType);
            Console.appendText("\nPacketId      : " + PacketId);
            Console.appendText("\nImei Length       : " + ImeiLength);

            String Imei = HEX.substring(0, 30);
            System.out.println("IMEI: " + converter.hexToAscii(Imei));
            HEX = HEX.substring(30, HEX.length());

            codec = converter.convertStringToIntHex(HEX.substring(0, 2));
            recordC = converter.convertStringToIntHex(HEX.substring(2, 4));
            HEX = HEX.substring(4, HEX.length());
            if (codec == 142){
                n = 4;
            }
            else{
                n = 2;
            }
        }
        else{
            ShowMessage( header: "Something is wrong",  text: "Imei length is too long.. imei length : " + Integer.parseInt(ImeiLength));
        }

    } else {
        ShowMessage( header: "Parse Error",  text: "Data is corrupted or codec is wrong");
    }
}
```

UDP Calculate Records in Parser class

If we choose to parse UDP type then the process would be same except for calculation part. Here it parses the header and checks for imei if its length is not longer than 20 digits. After that it parses the rest of data and returns number of records, the parsing process is the same as the TCP.

```java
private void CreateRecord() {
    AVLRecord AVLRecord;
    RecordHeader recordHeader = GetRecord_Data();
    RecordGPS_Element recordGPS_element = GetRecord_GPS();
    RecordIO_Element recordIOElement = GetRecord_IO();
    AVLRecord = new AVLRecord(recordHeader, recordGPS_element, recordIOElement);
    avlRecords.add(AVLRecord);
}
```

Create Record in Parser class

After getting record count on data, we run through the for loop, the Create Record method simply creates record header, GPS and IO elements objects to be added later on into the AVL Record object. First it calls out the header, then gps and finally the IO.

```java
private RecordHeader GetRecord_Data() {
    HeaderParser headerParser = new HeaderParser(HEX);
    RecordHeader recordHeader = headerParser.ConvertRecord_Data();
    HEX = headerParser.getHEX();
    return recordHeader;
}
```

Get Record Data in Parser class

```
/**
 * <h1>Get Record IO</h1>
 * <p>A function collects IO data from HEX</p>
 *
 * @return returns Record IO element object.
 */
private RecordIO_Element GetRecord_IO() {
    IOParser ioParser = new IOParser(HEX);
    RecordIO_Element recordIOElement;
    recordIOElement = ioParser.GetElement();
    HEX = ioParser.getHEX();
    return recordIOElement;
}
```

Get Record IO in Parser class

```
/**
 * <h1>Get Record GPS</h1>
 * <p>A function collects gps data from HEX</p>
 *
 * @return returns Record GPS element object.
 */
private RecordGPS_Element GetRecord_GPS() {
    GPSParser gpsParser = new GPSParser(HEX);
    RecordGPS_Element recordGPS_element;
    recordGPS_element = gpsParser.ConvertRecord_GPS();
    HEX = gpsParser.getHEX();
    return recordGPS_element;
}
```

Get Record GPS Parser class

```
/**
 * <h1>Assign Collection</h1>
 * <p>Assigns created collection to AVL record collection object</p>
 */
private void AssignCollection() {
    avlRecordCollection = CreateCollection();
}
```

Assign Collection in Parser class

Once we finish our for loop we call method Assign Collection to declare *avl Record* collection.

```java
/**
 * <h1>Print Record to result Text Area</h1>
 * <p>A function that prints all results from Avl Record Collection list.</p>
 */
private void PrintRecordToResultTA() {
    int i = 0;
    Console.appendText("\nCodec: " + avlRecordCollection.getCodecID());
    Console.appendText("\nRecord Count: " + avlRecordCollection.getRecordCount());
    for (AVLRecord record : avlRecordCollection.getRecordList()) {
        i++;
        Console.appendText("\n" + i + " Record==================================================");
        Console.appendText("\nRecord Timestamp:      " + record.getRecordHeader().getTimestamp());
        Console.appendText("\nRecord Priority:       " + record.getRecordHeader().getRecordPriority());
        Console.appendText("\nRecord GPS longitude: " + record.getRecordGPS_elements().getLongitude());
        Console.appendText("\nRecord GPS latitude  : " + record.getRecordGPS_elements().getLatitude());
        Console.appendText("\nRecord GPS altitude  : " + record.getRecordGPS_elements().getAltitude());
        Console.appendText("\nRecord GPS angle     : " + record.getRecordGPS_elements().getAngle());
        Console.appendText("\nRecord GPS satellites: " + record.getRecordGPS_elements().getSatellites());
        Console.appendText("\nRecord GPS Kmh       : " + record.getRecordGPS_elements().getKmh());

        Console.appendText("\nEventID : " + record.getRecordIO_elements().getEventID());
        Console.appendText("\nElement count : " + record.getRecordIO_elements().getElementCount());

        Console.appendText("\n\n1 byte elements ");
        for (RecordIO_Property data : record.getRecordIO_elements().getRecordIO_records().getByteList_1List()) {
            Console.appendText("\nID      : " + data.getID());
            Console.appendText("\n\tValue   : " + data.getValue());
        }

        Console.appendText("\n\n2 byte elements ");
        for (RecordIO_Property data : record.getRecordIO_elements().getRecordIO_records().getByteList_2List()) {
            Console.appendText("\nID      : " + data.getID());
            Console.appendText("\n\tValue   : " + data.getValue());
        }

        Console.appendText("\n\n4 byte elements ");
        for (RecordIO_Property data : record.getRecordIO_elements().getRecordIO_records().getByteList_4List()) {
            Console.appendText("\nID      : " + data.getID());
            Console.appendText("\n\tValue   : " + data.getValue());
        }

        Console.appendText("\n\n8 byte elements ");
        for (RecordIO_Property data : record.getRecordIO_elements().getRecordIO_records().getByteList_8List()) {
            Console.appendText("\nID      : " + data.getID());
            Console.appendText("\n\tValue   : " + data.getValue());
        }

        if(record.getRecordIO_elements().getRecordIO_records().getByteList_XList()!=null)
        {
            Console.appendText("\n\nX byte elements ");
            for (RecordIO_Property data : record.getRecordIO_elements().getRecordIO_records().getByteList_XList()) {
                Console.appendText("\nID      : " + data.getID());
                Console.appendText("\n\tLength   : " + data.getValue());
                Console.appendText("\n\tData   : " + data.getData());
            }
        }

        Console.appendText("\n");
    }
}
```

Print Record To Result Text Area in Parser class

Finally, after parsing process the object of avl Record Collection is ready to be used for printing out the whole results. It runs through every record to print out every record. If parsed data was codec 8 extented then we create X List which contains the data of x length bytes, and we print it out.

Listeners flowchart

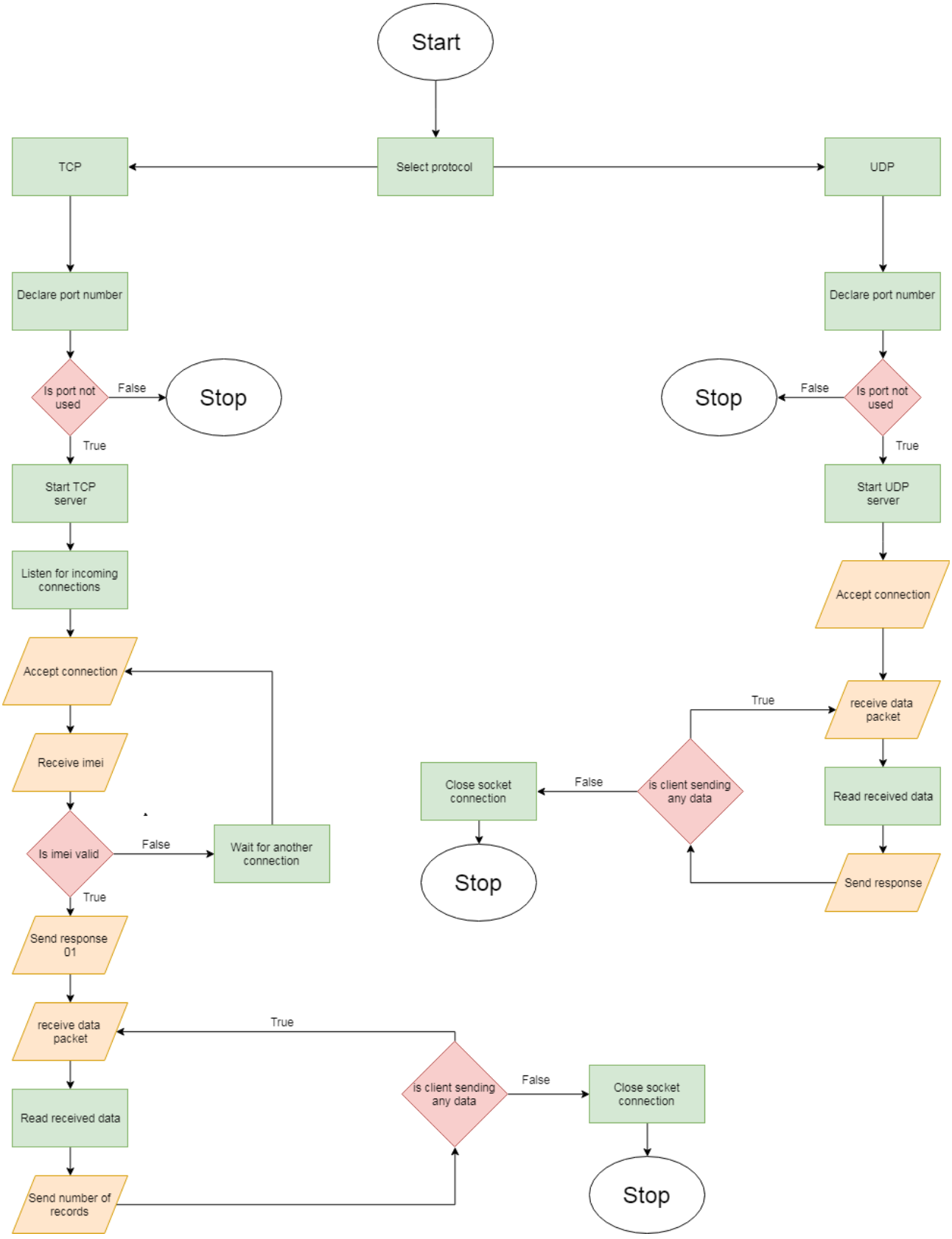

Illustration 2: Flow char of tcp and udp listeners

TCP part

```
public void ListenStart() {
    PortCheckUp checkPortUp = new PortCheckUp();
        if (checkPortUp.CheckPort(PortField.getText())) {
            Platform.runLater(() -> viewModel.setClientMessage(""));
            StartServer();
            StopBtn.setDisable(false);
            ListenBtn.setDisable(true);
            PortField.setDisable(true);
        }
}
```

Listen start method in TCP View class

In the TCP listener when we enter protocol number and press “Listen” we first check if port that we want to use for listening is even available.

```
/**
 * <h1>Check port</h1>
 * <p>Checks if given port number, is reachable and is not in use</p>
 * @param StringPort port is String but it converts to int later on
 * @return true of false
 * */
public boolean CheckPort(String StringPort) {
    if(isInteger(StringPort)) {
        if(isPortInUse(Integer.parseInt(StringPort)))
        {
            CanYouSeeMyPort canYouSeeMyPort = new CanYouSeeMyPort();
             if(canYouSeeMyPort.CheckPortIfAvailable(StringPort) == 200) {

                System.out.println("Port is gud");
                return true;
             }
        }
    }
    else{
        ShowMessage( header: "Port is invalid",  text: "Port is not available.");
        return false;
    }
    return false;
}
```

Check Port method in Port Check Up Port class

The Check port method returns true or false results. If port is in use it will return false and show message box, otherwise it should return true saying that port is not in use and is good to go.

```
private void StartServer() {
    LoadBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);
    TCPServer = new TCPServer(viewModel, Integer.parseInt(PortField.getText()));
    TCPServer.setFlag(true);
    executorService = Executors.newFixedThreadPool( nThreads: 1);
    executorService.submit(TCPServer);
}
```

Start Server method in TCP View class

Later on we initialize server with Start Server method in which we call executor service to run our server.

```java
/**
 * <h1>Run</h1>
 * <p>Runs the runnable thread to listen connections, it accepts a connection, if accept was successful,
 * the connection is added to tcpConnections list and runs the TCPConnection for further listening.
 * The server is running in while loop and stops when Running is set to false,
 * then break is called and shutdowns every connected client.</p>
 * */
public void run() {
    tcpConnections = new ArrayList<>();
    try {
        ss = new ServerSocket(port);
        System.out.println("Listening on port : " + ss.getLocalPort());
        ExecutorService executorService;
        while (true) {
            executorService = Executors.newSingleThreadExecutor();
            socket = ss.accept();
            TCPConnection connection = new TCPConnection(socket, viewModel);
            executorService.submit(connection);
            tcpConnections.add(connection);
            if (!running) {
                StopConnections();
                break;
            }
        }
        executorService.shutdownNow();
        Thread.sleep( millis: 100);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } catch (IOException e) {
        System.out.println("socket is closed");
    }
}
```

Run method in TCP Server class

Run method runs the server thread in which creates a server socket with declared port number. In While loop the listening starts and waits for the connections, once connection is established it will call out the tcp connection object to communicate with the device, it will put into the executor service and add to connections list. The server runs until we press “Stop” which calls to set running to false and causes to interrupt the server thread and all connections that are active.

```java
/**
 * <h1>Run function to start listener</h1>
 * <p>Simply runs the runnable thread to listen everything from client</p>
 * */
public void run() {
    try {
        inputStream = new DataInputStream(socket.getInputStream());
        outputStream = new DataOutputStream(socket.getOutputStream());
        Listen();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Run method in TCP Connection class

The TCP connection run method declares input and output streams and has one method call Listen.

```java
/**
 * <h1>Listen</h1>
 * <p>Function for listening connected client</p>
 * @throws IOException throws exception if input stream is interrupted
 * */
private void Listen() throws IOException {
    while (flag) {
        System.out.println("listening...");
        while (!socket.isClosed() && inputStream.available() == 0) {
            try {
                Thread.sleep( millis: 1);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
        Communicate();
    }
    inputStream.close();
    outputStream.close();
    socket.close();
}
```

Listen method in TCP Connection class

The Listen method is responsible for listening. The while loop runs until we set flag to false, the while loop can also be break if device disconnects from the server. While loop is running we call our method Communicate.

```java
/**
 * <h1>Communicate</h1>
 * <p>A reader and sender with client, first it reads imei, then sends back 01.
 * It receives data, as soon it receives it sends back number of records.
 * The while loop initializes and runs until it get interrupted or client disconnects.</p>
 * */
private void Communicate()  {
    imei =  Objects.requireNonNull(ReadInput()).substring(4);
    imei = converter.ReadImei(imei);
    String path = System.getProperty("user.home") + "/Desktop";
    logger = new Logger( path: path+"/Logs/TCPLogs/"+imei);
    logger.PrintToLOG( text: GetTime()+" IMEI: " +imei);
    SendOutput("01");
    logger.PrintToLOG( text: "\tResponse: [0" + 1 + "]");
    String input = ReadInput();
    System.out.println("Crc: " + Integer.toHexString(CRC(input)));
    Log(Objects.requireNonNull(input));
    while(flag){
        String recordsCount = GetNumberOfRecords(input);
        SendOutput("000000" + recordsCount);
        logger.PrintToLOG( text: "\tResponse: [000000" + recordsCount + "]");
        input = ReadInput();
        System.out.println("Crc: " + Integer.toHexString(CRC(input)));
        Log(Objects.requireNonNull(input));
    }
}
```

Communicate method in TCP Connection class

This method is mainly used to communicate with the device, as soon it receives imei it responds with byte 01, later on once it receives data packet we start another loop to send and receive data packets by receiving it and sending number of records. The loop is also breakable by setting flag to false. It logs at the same time as it communicates.

```java
/**
 * <h1>Read Input</h1>
 * <p>Reads the input from client. Currently maximum message byte is set up to 8192,
 * if message is bigger then message will not be properly readable and displayed.</p>
 * @return String of received data
 * */
private String ReadInput() {
    byte[] messageByte = new byte[8192];
    int dataSize;
    try {
        dataSize = inputStream.read(messageByte);
        String finalInput = converter.BytesArrayToHex(messageByte, dataSize);
        SendToConsole(finalInput);
        return finalInput;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

Read Input method in TCP Connection class

Currently it can receive up to 8192 bytes of data, it puts the received data to byte array and converts to hex for readability and further usages.

```java
/**
 * <h1>Send Output</h1>
 * <p>Sends output to the client</p>
 * @param message the parameter is a received hex data
 * */
private void SendOutput(String message)  {
    try {
        outputStream.write(converter.StringToByteArray(message));
        outputStream.flush();
    } catch (IOException e) {
        System.out.println("Output stream was interrupted");
    }
}
```

Send Output method in TCP Connection class

Send Output method simply converts string to byte array and sends it to device.

UDP part

```
public void ListenStart() {

        if(portCheckUp.CheckPort((PortField.getText())))
        {
            viewModel.setClientMessage("");
            StartServer();
            StopBtn.setDisable(false);
            PortField.setDisable(true);
            ListenBtn.setDisable(true);
        }
}
```

Listen start method in UDP View class

```
private void StartServer()
{
    LoadBar.setProgress(ProgressIndicator.INDETERMINATE_PROGRESS);
    server = new UDP_ListenerServer(viewModel,Integer.parseInt(PortField.getText()));
    server.setFlag(true);
    executorService = Executors.newFixedThreadPool( nThreads: 1);
    executorService.submit(server);
}
```

Start server method in UDP View class

The UDP start server starts the same way as TCP. It checks the port for availability and starts the server if its available.

```
/**
 * <h1>Run function to start listener</h1>
 * <p>Simply runs the runnable thread to listen everything from client</p>
 * */
public void run() {
        try {
            setDs(new DatagramSocket(port));
        } catch (SocketException e) {
            e.printStackTrace();
        }
        System.out.println("Datagram Socket initialized");
        listen();
}
```

Run method in UDP Listener server class

The run method initializes DatagramSocket with declared port number, later on it calls listen method to do the listening.

```java
/**
 * <h1>Listen</h1>
 * <p>Function for listening connected client and communicates with it.
 * The while loop is running until we set flag to false.</p>
 * */
private void listen()
{
    while (flag) {
        if(!flag)
        {
            ds.close();
            break;
        }
        try {
            byte[] bb = new byte[4096];

            DatagramPacket dp = new DatagramPacket(bb, bb.length);
            System.out.println("Packet created");

            System.out.println("Waiting for data from client");

            getDs().receive(dp);
            String str = new String(dp.getData(),  offset: 0, dp.getLength());
            int i = dp.getLength();
            byte[] data;
            data = dp.getData();
            InetAddress IPAddress = dp.getAddress();

            int port = dp.getPort();
            System.out.println("ByteArray: " + str);
            System.out.println();
            String msg = converter.BytesArrayToHex(data);

            msg = msg.replaceAll( regex: " ",  replacement: "");
            msg = msg.substring(0, i * 2);
            System.out.println("RESULT : " + msg);
            UDPPacket packet = FillPacket(msg.substring(0,50));
            PrintPacket(packet);
            logger.PrintToLOG( text: "received : " + msg);
            SendResponse(packet, IPAddress, port);

            String finalData = msg;
            Platform.runLater(() -> getViewModel().setClientMessage(viewModel.getClientMessage() + "\r\n"+IPAddress+" : " + finalData));

        } catch (IOException e) {
            System.out.println("Socket is closed");
        }
    }
}
```

Listen method in UDP Listener server class

This method is the main method for listening UDP connections. The while loop runs until we change flag to false. It creates a datagram packet byte array of 4096 length and waits for client to receive the data. Once client sends the data packet we store it in DatagramPacket (dp) and use it for reading. Secondly we convert data packet to data byte array to convert it to hex. Then we read the hex and send feedback to client.

```java
/**
 * <h1>Send Response</h1>
 * <p>Function for sending output back to the client</p>
 * @param packet UDPPacket object
 * @param IPAddress InetAddress for sending to specific address
 * @param port receiver port is required.
 * */
private void SendResponse(UDPPacket packet, InetAddress IPAddress, int port) {
    byte[] concatBytes = ArrayUtils.addAll(converter.toHexBytes( text: "0005"),converter.toHexBytes(packet.getPacketIdentification()));
    concatBytes = ArrayUtils.addAll(concatBytes,converter.toHexBytes( text: "01"));
    concatBytes = ArrayUtils.addAll(concatBytes,converter.toHexBytes(packet.getPacketID()));
    concatBytes = ArrayUtils.addAll(concatBytes,converter.toHexBytes(packet.getNumberOfData()));
    DatagramPacket sendPacket = new DatagramPacket(concatBytes, concatBytes.length, IPAddress, port);
    try {
        getDs().send(sendPacket);
    } catch (IOException e) {
        System.out.println("IO exception sending response");
    }
    logger.PrintToLOG( text: "Response send: ["+converter.BytesArrayToHex(concatBytes)+"]\n");
}
```

Send Response method in UDP Listener server class

The send response method for sending the feedback, we concat multiple arrays into one. So we would generate a single byte array to send it.