2017/18

Building an Ad Hoc Wireless Sensor Network

17COP531 Coursework

Andy Toward - B419754
Dan Forrester - B422433
Rawa Resul - B729621
Simon Yip - B424047
Matt Beechey - B730374

Contents

Introduction

In this project, a simple Ad hoc wireless sensor network (WSN) has been built through the development of a simple routing protocol. Said protocol has been implemented onto a given hardware platform, consisting of a number of sensor nodes running the Contiki operating system, and 2 PCs running the Tera Term program to display routing information and measurements of each node, including temperature, battery and voltage readings. This report will outline the design of the network, including a system overview and information on the algorithm used. It will also detail portions of the source code used to implement the algorithm, and a series of test procedures along with analysis of these tests and the results they produced.

System Design

Our system comprises of a minimum of 4 sensor nodes; one will act as the source, one as the destination, and at least two nodes behave as intermediate ‘router’ nodes, to retransmit information between the source and destination. The goal of the system is to send a packet of data from the source node to the destination, via the optimal route. The data is contained in a packet, and upon receipt, the destination node should display it on the monitor of a PC to which it is connected. The route that the data takes is decided by a comparison of the battery levels of each intermediate node, with a secondary comparison of their received signal strength indication (RSSI) values in the event that their battery levels are equal.

The establishment of the data’s route is found by broadcasting a route request (RREQ) message from the source node. Said message is received by each intermediate node, it will choose the most optimum previous hop back to the source by comparing the battery levels and RSSI values between the RREQ and store this information into the reverse table. Then, the intermediate node should rebroadcast the message. Each intermediate node will only broadcast a RREQ once for each sequence number, unless it received a sequence number that is higher than the previous sequence number it received from the RREQ.

When a RREQ reaches the destination, this node then sends a unicast route response (RREP) message back to intermediate node it received from. When the intermediate node receives the RREP, it will register the node it received from as the next hop to the destination node in the forwarding table and send a unicast message to the hop specified in the reverse table. RREP is passed around until it reaches the source. The source will compare the RREP messages it receive from intermediate nodes and choose the best hop (intermediate node) with the best battery level and update the forwarding table.
The reason why the destination node sends an RREP for every RREQ message the destination receives because of the specification given, it is specified that the source node decides on the best next hop it should take for the optimum route. The source node will receive multiple RREP messages and able to choose the best next hop by comparing these messages.

After the source has updated the forwarding table, the user can now send a unicast data packet containing the information it wishes to relay to the destination. After travelling via the route based on the forwarding table in each intermediate nodes, the destination node will obtain the source's data and output it on the PC terminal, using the Tera Term program.

There are two different programs that our system is comprised of; one is for the behavior of the source node, whilst the other sets the behavior of all other nodes in the system. There are checks in place to determine whether a node is an intermediate or destination by reading the MAC address of the node. The source MAC address and destination MAC address will be predefined. There should never be a scenario in which the source communicates directly with the destination, this can be achieved by limiting the RSSI and hardcode the behaviour of all other nodes so that it should ignore any broadcast message (RREQ) directly from the source.

Firstly, the source node wishes to determine the route on which it should send its data to reach the destination. To do this, it begins by constructing an RREQ message by creating a buffer. This buffer contains information about itself, and the destination it wishes to reach. This packet buffer is sent via a broadcast channel, able to be picked up by any node in range which is listening for broadcast signals. Intermediate nodes are only capable of receiving one message from a node of each sequence number, to prevent the opportunity for loops to occur. When button 2 is pressed on the source device, the outgoing packet's sequence number is incremented, to once again allow the message to be received and dealt with by the intermediate nodes.

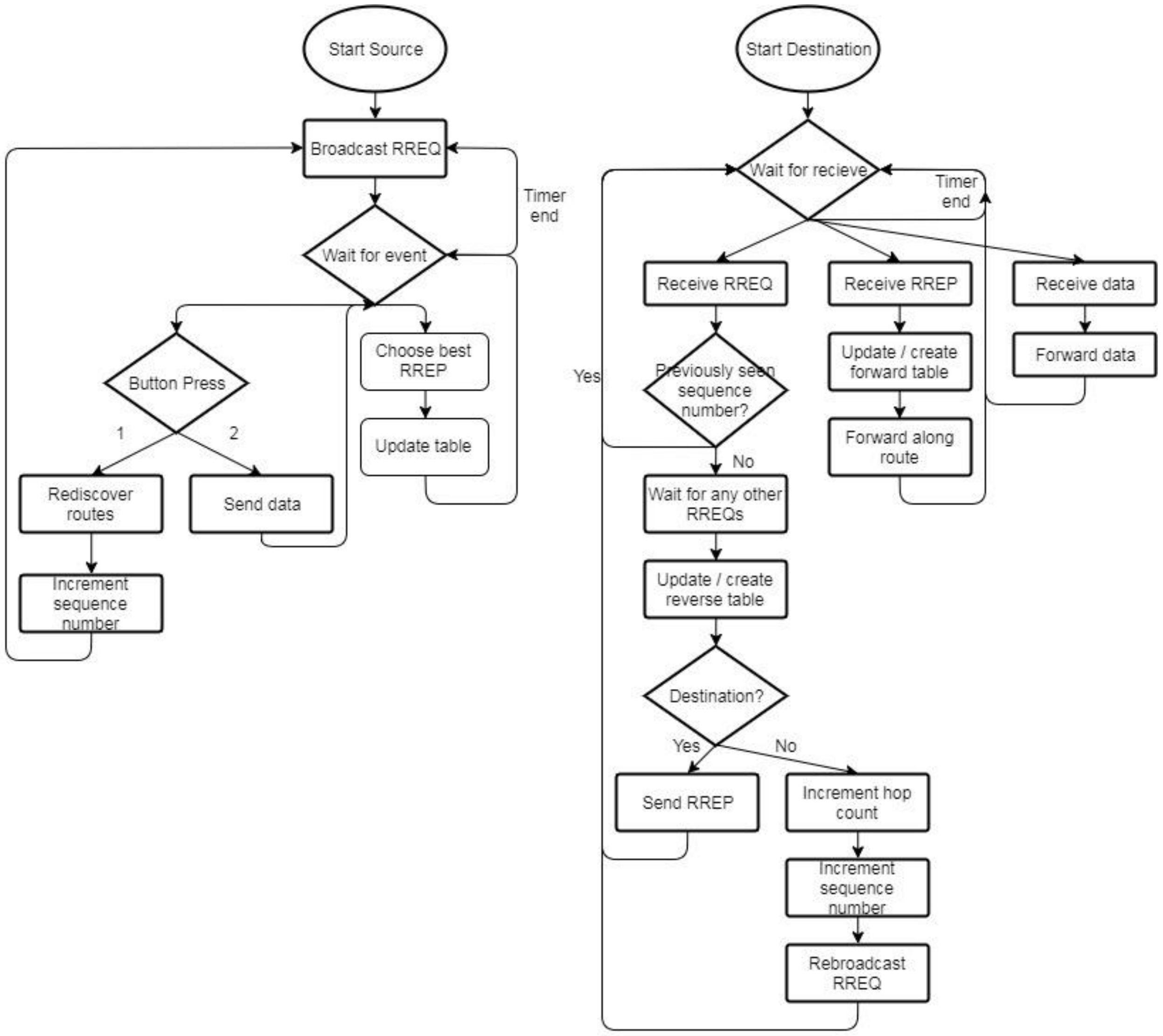
The next stage of the process is having the intermediate devices rebroadcast the message received from the source, passing it through the system on its way to the destination. When a broadcast message is detected, the node confirms that it is of a unique sequence number, and discards it if it not. Assuming the sequence number is a new one, a timer begins to allow for the possibility of more optimal routes to be discovered, rather than simply accepting the first one found. A check is performed to confirm that it is not the intended destination of the packet, and it then reads in the relevant data from the packet buffer, storing in a temporary buffer of its own. This data is added to the node's routing table, before the construction of another packet buffer containing the data which came from the source. This is then rebroadcast in another RREQ message, with the hop count increased by one to indicate that it has travelled through a node on its journey.

Once the destination is reached by the route request, it again confirms that the message is of a new sequence number from each node that broadcasts the request. To confirm to the source that it has received its route request, it prepares and sends back an RREP message via unicast. The route taken by the message is, as previously mentioned, chosen by the node(s) with the highest battery, with RSSI as a secondary comparator. Once an intermediate node receives this route reply, the corresponding information from the packet buffer is added to its forwarding table, to indicate that it is a part of the optimum route. The information is taken and sent along as another unicast RREP message, until reaching the original source node.

After the source receives the unicast RREP, a timer is started before performing any further tasks, to allow for a number of unicast RREPs to be received and the program to determine the best route for the data to traverse. It takes the relevant routing information from the best “battery” (or RSSI level) and constructs its own forwarding table. The data which it wishes to send can now be prepared as a data packet, to be transmitted along the best route and shown on the monitor of the PC to which the destination device is connected. This data is sent via a data transmission message (DATATX) along said path. Any intermediate nodes which receive the transmission confirm that they aren't the intended destination, and retransmit it to the next hop of the route.

Finally, the destination node receives the data originating from the source, and after confirming that it is the intended recipient, prints out the data on screen. Information such as the temperature and voltage of the source node are displayed, and updated with each new sequence broadcast.

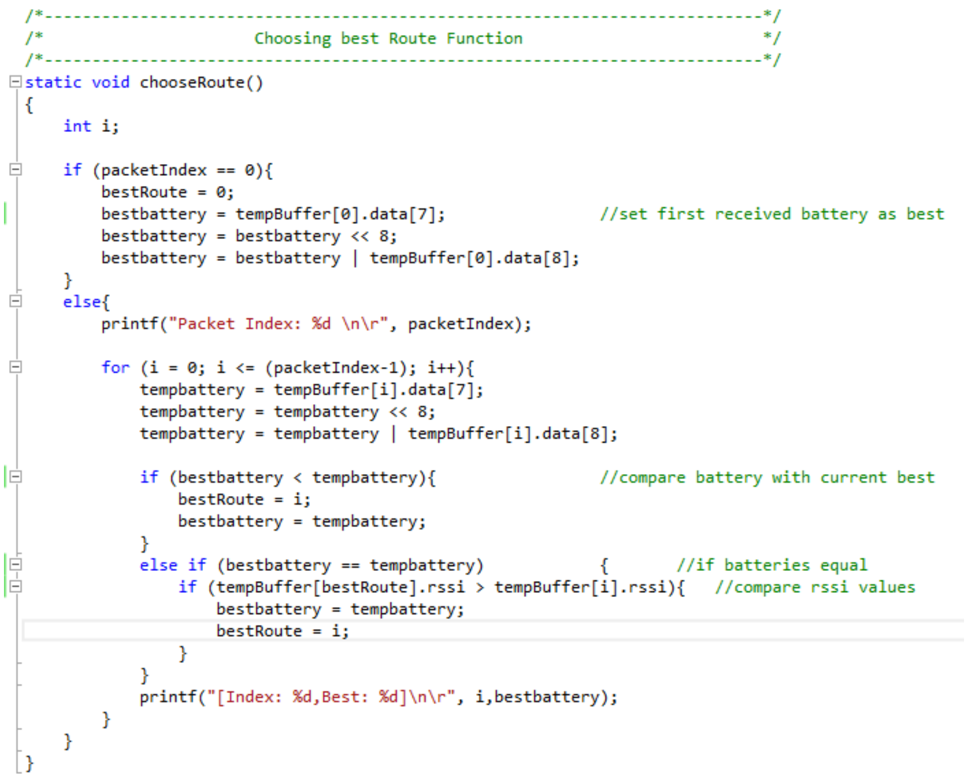
Flowchart


Start Source
Broadcast RREQ
Timer end
Wait for event
Button Press
1
2
Rediscover routes
Send data
Increment sequence number
Choose best RREP
Update table
Start Destination
Wait for recieve
Timer end
Receive RREQ
Receive RREP
Receive data
Previously seen sequence number?
Yes
No
Update / create forward table
Forward data
Forward along route
Wait for any other RREQs
Update / create reverse table
Destination?
Yes
No
Send RREP
Increment hop count
Increment sequence number
Rebroadcast RREQ

Source Code Breakdown

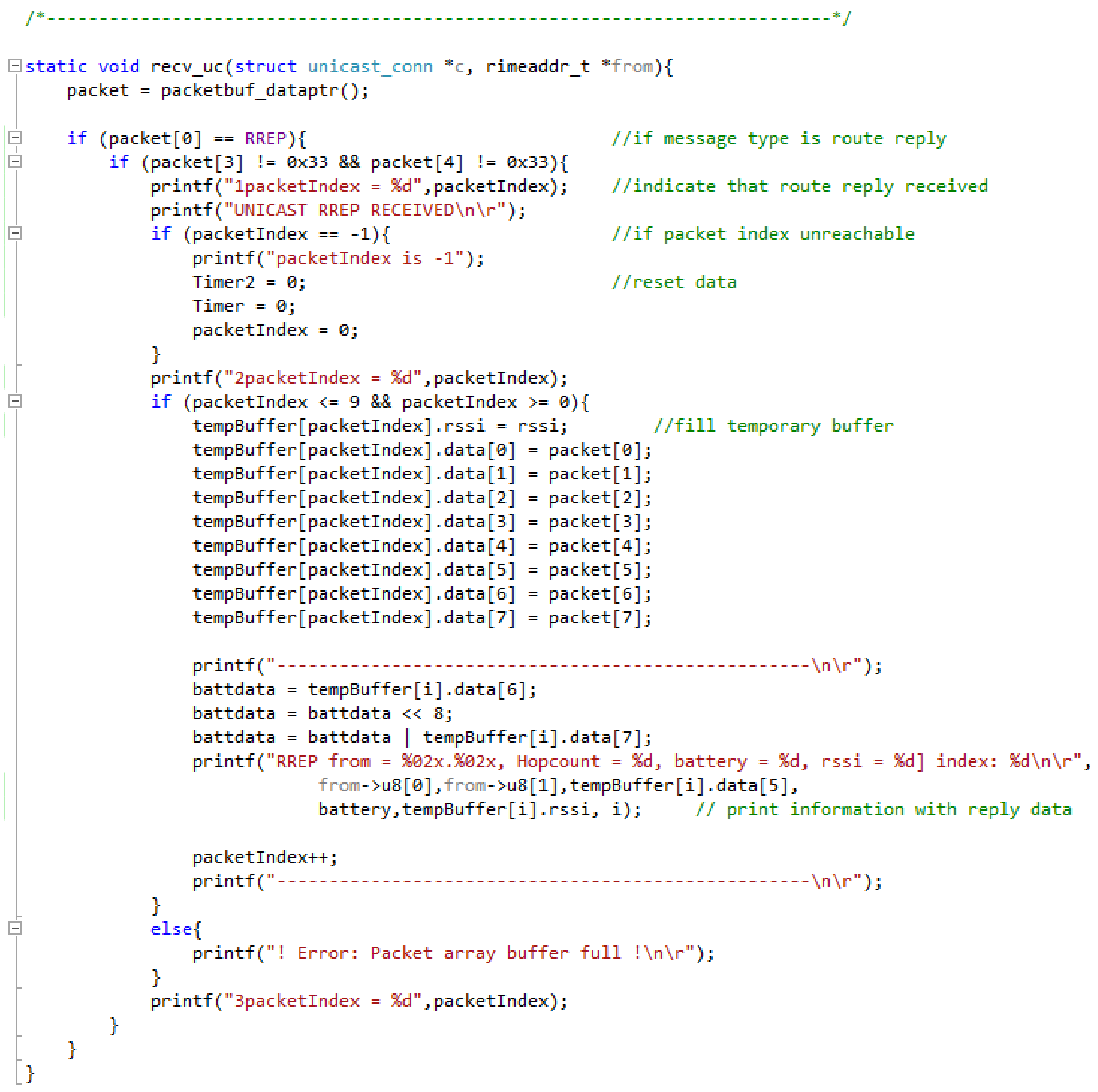
Code for the source node:

- This function is where the source node decides which battery value of its neighbours is the most desirable. If two battery values are found to be equal, the RSSI values are compared to determine which route is the most desirable.



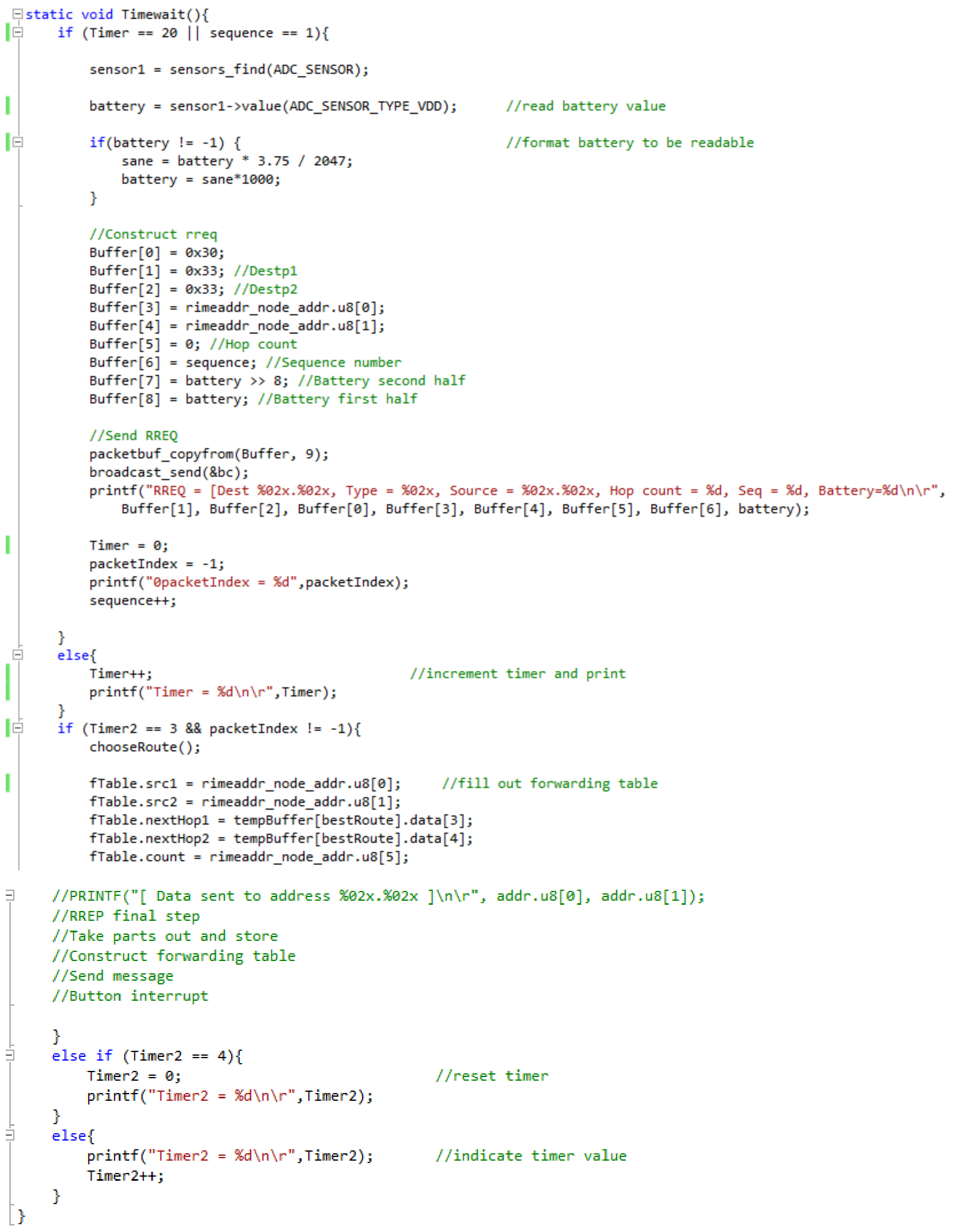
```
/*---------------------------------------------------------------------------*/
/*                    Choosing best Route Function                           */
/*---------------------------------------------------------------------------*/
static void chooseRoute()
{
    int i;

    if (packetIndex == 0){
        bestRoute = 0;
        bestbattery = tempBuffer[0].data[7];                //set first received battery as best
        bestbattery = bestbattery << 8;
        bestbattery = bestbattery | tempBuffer[0].data[8];
    }
    else{
        printf("Packet Index: %d \n\r", packetIndex);

        for (i = 0; i <= (packetIndex-1); i++){
            tempbattery = tempBuffer[i].data[7];
            tempbattery = tempbattery << 8;
            tempbattery = tempbattery | tempBuffer[i].data[8];

            if (bestbattery < tempbattery){                 //compare battery with current best
                bestRoute = i;
                bestbattery = tempbattery;
            }
            else if (bestbattery == tempbattery)            {       //if batteries equal
                if (tempBuffer[bestRoute].rssi > tempBuffer[i].rssi){   //compare rssi values
                    bestbattery = tempbattery;
                    bestRoute = i;
                }
            }
            printf("[Index: %d,Best: %d]\n\r", i,bestbattery);
        }
    }
}
```

- This function is called when a unicast message is received. There is a check to confirm that the message type is an RREP message, and if so fill a temporary buffer and print the data contained in the packet.



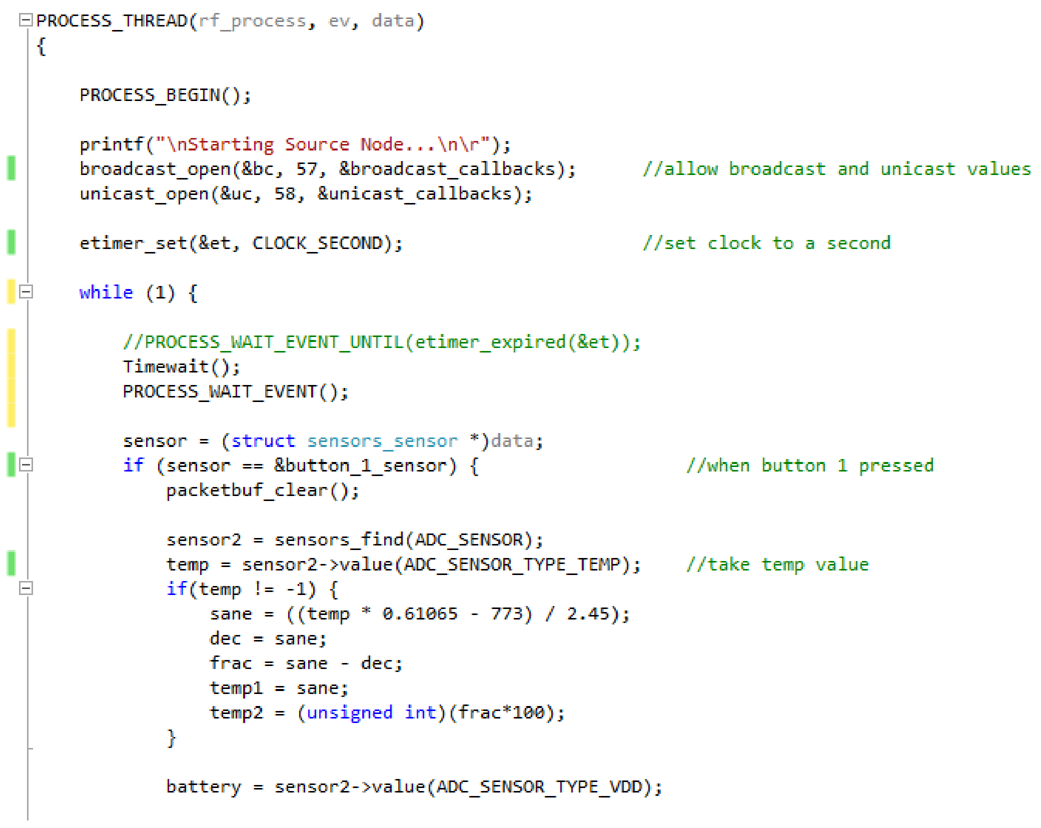
```
/*---------------------------------------------------------------------------*/

static void recv_uc(struct unicast_conn *c, rimeaddr_t *from){
    packet = packetbuf_dataptr();

    if (packet[0] == RREP){                            //if message type is route reply
        if (packet[3] != 0x33 && packet[4] != 0x33){
            printf("1packetIndex = %d",packetIndex);    //indicate that route reply received
            printf("UNICAST RREP RECEIVED\n\r");
            if (packetIndex == -1){                    //if packet index unreachable
                printf("packetIndex is -1");
                Timer2 = 0;                            //reset data
                Timer = 0;
                packetIndex = 0;
            }
            printf("2packetIndex = %d",packetIndex);
            if (packetIndex <= 9 && packetIndex >= 0){
                tempBuffer[packetIndex].rssi = rssi;        //fill temporary buffer
                tempBuffer[packetIndex].data[0] = packet[0];
                tempBuffer[packetIndex].data[1] = packet[1];
                tempBuffer[packetIndex].data[2] = packet[2];
                tempBuffer[packetIndex].data[3] = packet[3];
                tempBuffer[packetIndex].data[4] = packet[4];
                tempBuffer[packetIndex].data[5] = packet[5];
                tempBuffer[packetIndex].data[6] = packet[6];
                tempBuffer[packetIndex].data[7] = packet[7];

                printf("---------------------------------------------------\n\r");
                battdata = tempBuffer[i].data[6];
                battdata = battdata << 8;
                battdata = battdata | tempBuffer[i].data[7];
                printf("RREP from = %02x.%02x, Hopcount = %d, battery = %d, rssi = %d] index: %d\n\r",
                            from->u8[0],from->u8[1],tempBuffer[i].data[5],
                            battery,tempBuffer[i].rssi, i);     // print information with reply data

                packetIndex++;
                printf("---------------------------------------------------\n\r");
            }
            else{
                printf("! Error: Packet array buffer full !\n\r");
            }
            printf("3packetIndex = %d",packetIndex);
        }
    }
}
```

- The source device is told to resend an RREQ packet either every 20 seconds, or when the sequence value of 1 is called. When either of these conditions are met, a new RREQ message is sent, refreshing the route discovery and updating the network when necessary.

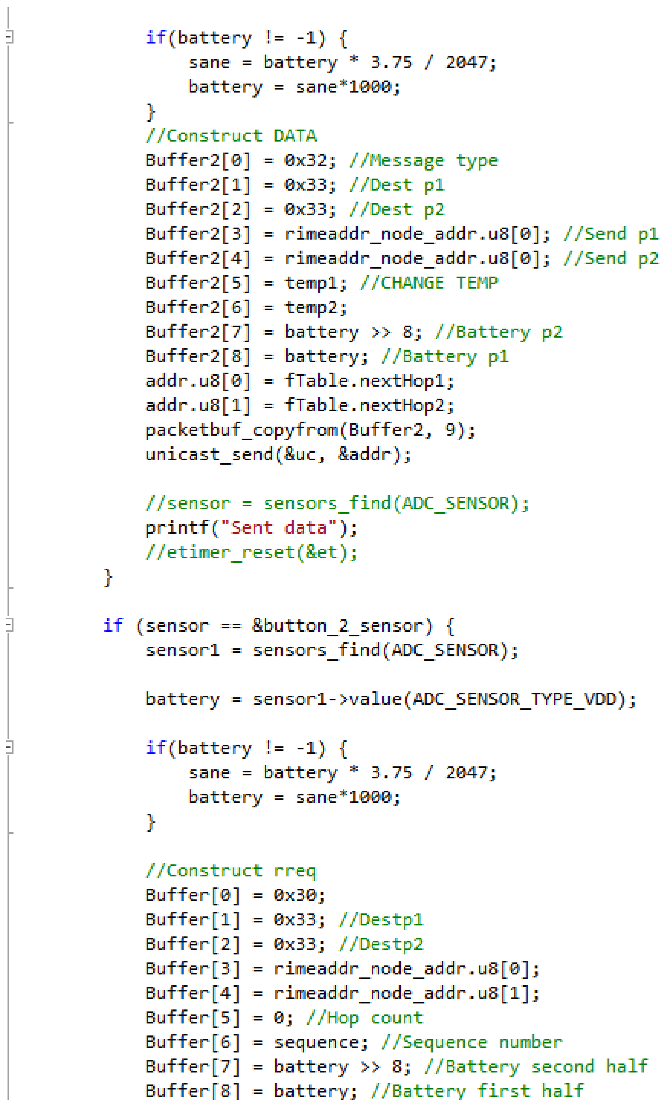


```c
static void Timewait(){
    if (Timer == 20 || sequence == 1){

        sensor1 = sensors_find(ADC_SENSOR);

        battery = sensor1->value(ADC_SENSOR_TYPE_VDD);      //read battery value

        if(battery != -1) {                                 //format battery to be readable
            sane = battery * 3.75 / 2047;
            battery = sane*1000;
        }

        //Construct rreq
        Buffer[0] = 0x30;
        Buffer[1] = 0x33; //Destp1
        Buffer[2] = 0x33; //Destp2
        Buffer[3] = rimeaddr_node_addr.u8[0];
        Buffer[4] = rimeaddr_node_addr.u8[1];
        Buffer[5] = 0; //Hop count
        Buffer[6] = sequence; //Sequence number
        Buffer[7] = battery >> 8; //Battery second half
        Buffer[8] = battery; //Battery first half

        //Send RREQ
        packetbuf_copyfrom(Buffer, 9);
        broadcast_send(&bc);
        printf("RREQ = [Dest %02x.%02x, Type = %02x, Source = %02x.%02x, Hop count = %d, Seq = %d, Battery=%d\n\r",
            Buffer[1], Buffer[2], Buffer[0], Buffer[3], Buffer[4], Buffer[5], Buffer[6], battery);

        Timer = 0;
        packetIndex = -1;
        printf("0packetIndex = %d",packetIndex);
        sequence++;

    }
    else{
        Timer++;                            //increment timer and print
        printf("Timer = %d\n\r",Timer);
    }
    if (Timer2 == 3 && packetIndex != -1){
        chooseRoute();

        fTable.src1 = rimeaddr_node_addr.u8[0];     //fill out forwarding table
        fTable.src2 = rimeaddr_node_addr.u8[1];
        fTable.nextHop1 = tempBuffer[bestRoute].data[3];
        fTable.nextHop2 = tempBuffer[bestRoute].data[4];
        fTable.count = rimeaddr_node_addr.u8[5];

    //PRINTF("[ Data sent to address %02x.%02x ]\n\r", addr.u8[0], addr.u8[1]);
    //RREP final step
    //Take parts out and store
    //Construct forwarding table
    //Send message
    //Button interrupt

    }
    else if (Timer2 == 4){
        Timer2 = 0;                         //reset timer
        printf("Timer2 = %d\n\r",Timer2);
    }
    else{
        printf("Timer2 = %d\n\r",Timer2);   //indicate timer value
        Timer2++;
    }
}
```

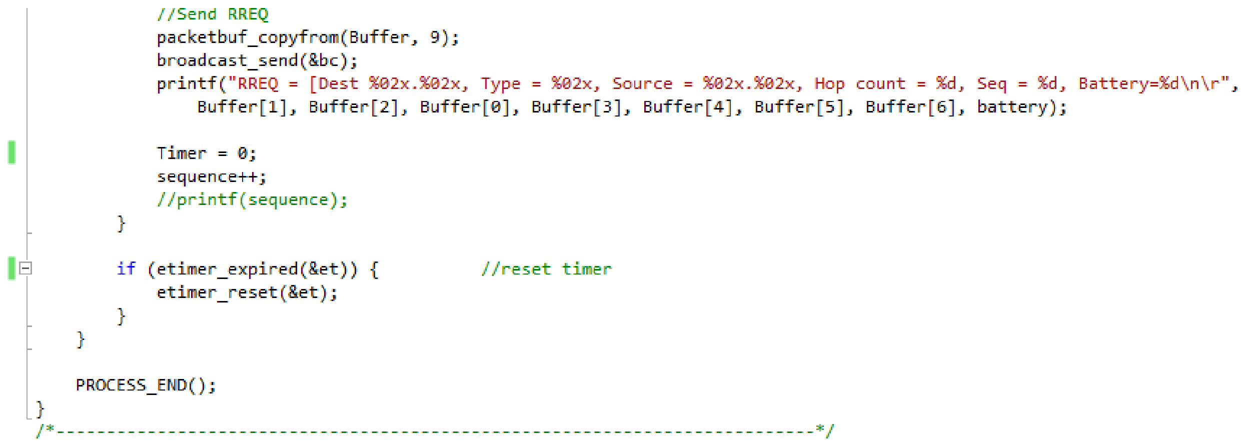
- The following is the process thread of the source node. The 20 second countdown is triggered and repeated, whilst also waiting for a button press on the device itself. Pressing button 1 will send the data packet along the route determined by the network, and button 2 will send another RREQ message out, on demand.



```
PROCESS_THREAD(rf_process, ev, data)
{

    PROCESS_BEGIN();

    printf("\nStarting Source Node...\n\r");
    broadcast_open(&bc, 57, &broadcast_callbacks);      //allow broadcast and unicast values
    unicast_open(&uc, 58, &unicast_callbacks);

    etimer_set(&et, CLOCK_SECOND);                      //set clock to a second

    while (1) {

        //PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        Timewait();
        PROCESS_WAIT_EVENT();

        sensor = (struct sensors_sensor *)data;
        if (sensor == &button_1_sensor) {                   //when button 1 pressed
            packetbuf_clear();

            sensor2 = sensors_find(ADC_SENSOR);
            temp = sensor2->value(ADC_SENSOR_TYPE_TEMP);    //take temp value
            if(temp != -1) {
                sane = ((temp * 0.61065 - 773) / 2.45);
                dec = sane;
                frac = sane - dec;
                temp1 = sane;
                temp2 = (unsigned int)(frac*100);
            }

            battery = sensor2->value(ADC_SENSOR_TYPE_VDD);
```



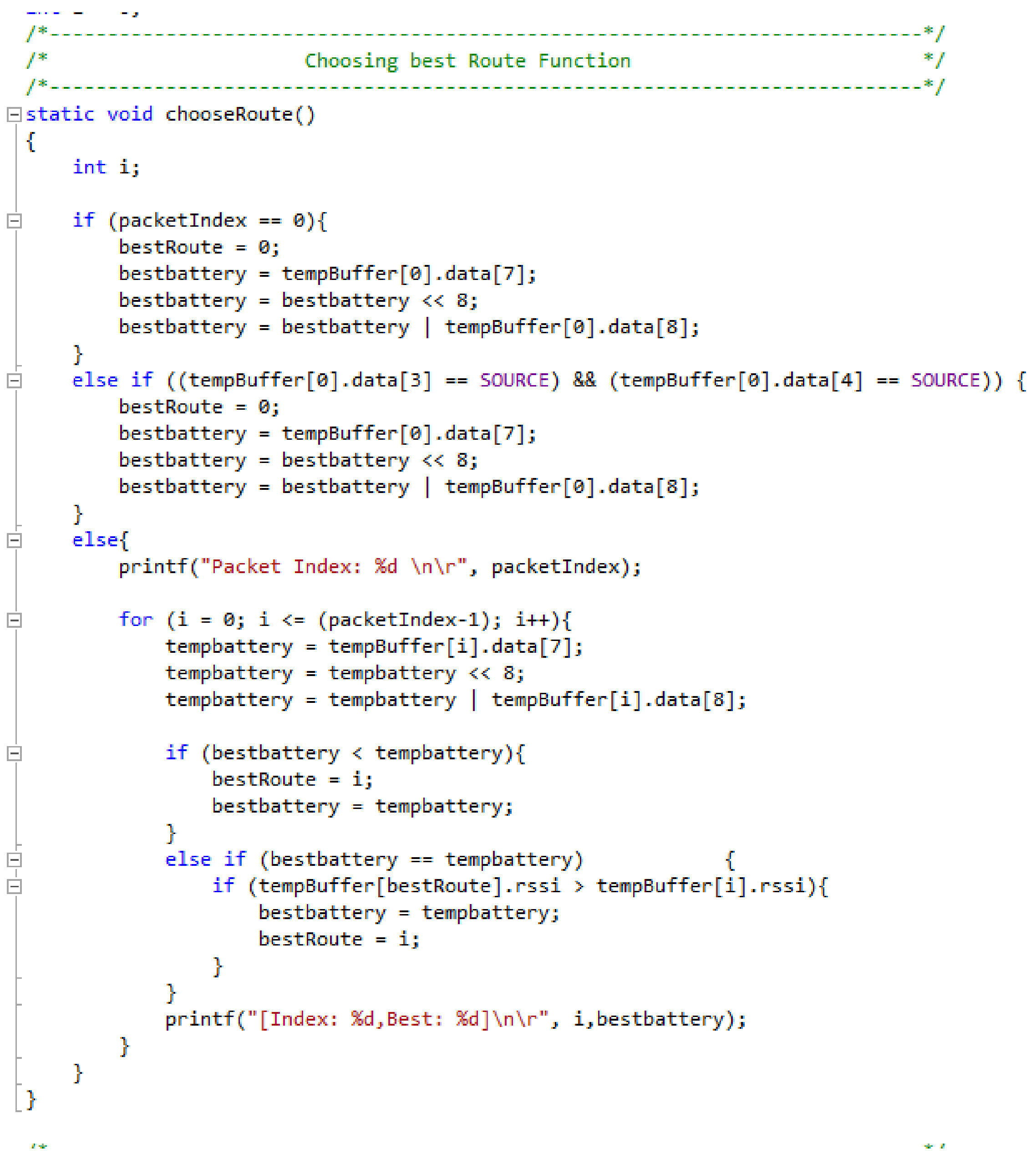
```
            if(battery != -1) {
                sane = battery * 3.75 / 2047;
                battery = sane*1000;
            }
            //Construct DATA
            Buffer2[0] = 0x32; //Message type
            Buffer2[1] = 0x33; //Dest p1
            Buffer2[2] = 0x33; //Dest p2
            Buffer2[3] = rimeaddr_node_addr.u8[0]; //Send p1
            Buffer2[4] = rimeaddr_node_addr.u8[0]; //Send p2
            Buffer2[5] = temp1; //CHANGE TEMP
            Buffer2[6] = temp2;
            Buffer2[7] = battery >> 8; //Battery p2
            Buffer2[8] = battery; //Battery p1
            addr.u8[0] = fTable.nextHop1;
            addr.u8[1] = fTable.nextHop2;
            packetbuf_copyfrom(Buffer2, 9);
            unicast_send(&uc, &addr);

            //sensor = sensors_find(ADC_SENSOR);
            printf("Sent data");
            //etimer_reset(&et);
        }

        if (sensor == &button_2_sensor) {
            sensor1 = sensors_find(ADC_SENSOR);

            battery = sensor1->value(ADC_SENSOR_TYPE_VDD);

            if(battery != -1) {
                sane = battery * 3.75 / 2047;
                battery = sane*1000;
            }

            //Construct rreq
            Buffer[0] = 0x30;
            Buffer[1] = 0x33; //Destp1
            Buffer[2] = 0x33; //Destp2
            Buffer[3] = rimeaddr_node_addr.u8[0];
            Buffer[4] = rimeaddr_node_addr.u8[1];
            Buffer[5] = 0; //Hop count
            Buffer[6] = sequence; //Sequence number
            Buffer[7] = battery >> 8; //Battery second half
            Buffer[8] = battery; //Battery first half
```



```
            //Send RREQ
            packetbuf_copyfrom(Buffer, 9);
            broadcast_send(&bc);
            printf("RREQ = [Dest %02x.%02x, Type = %02x, Source = %02x.%02x, Hop count = %d, Seq = %d, Battery=%d\n\r",
                Buffer[1], Buffer[2], Buffer[0], Buffer[3], Buffer[4], Buffer[5], Buffer[6], battery);

            Timer = 0;
            sequence++;
            //printf(sequence);
        }

        if (etimer_expired(&et)) {          //reset timer
            etimer_reset(&et);
        }
    }

    PROCESS_END();
}
/*---------------------------------------------------------------------------*/
```

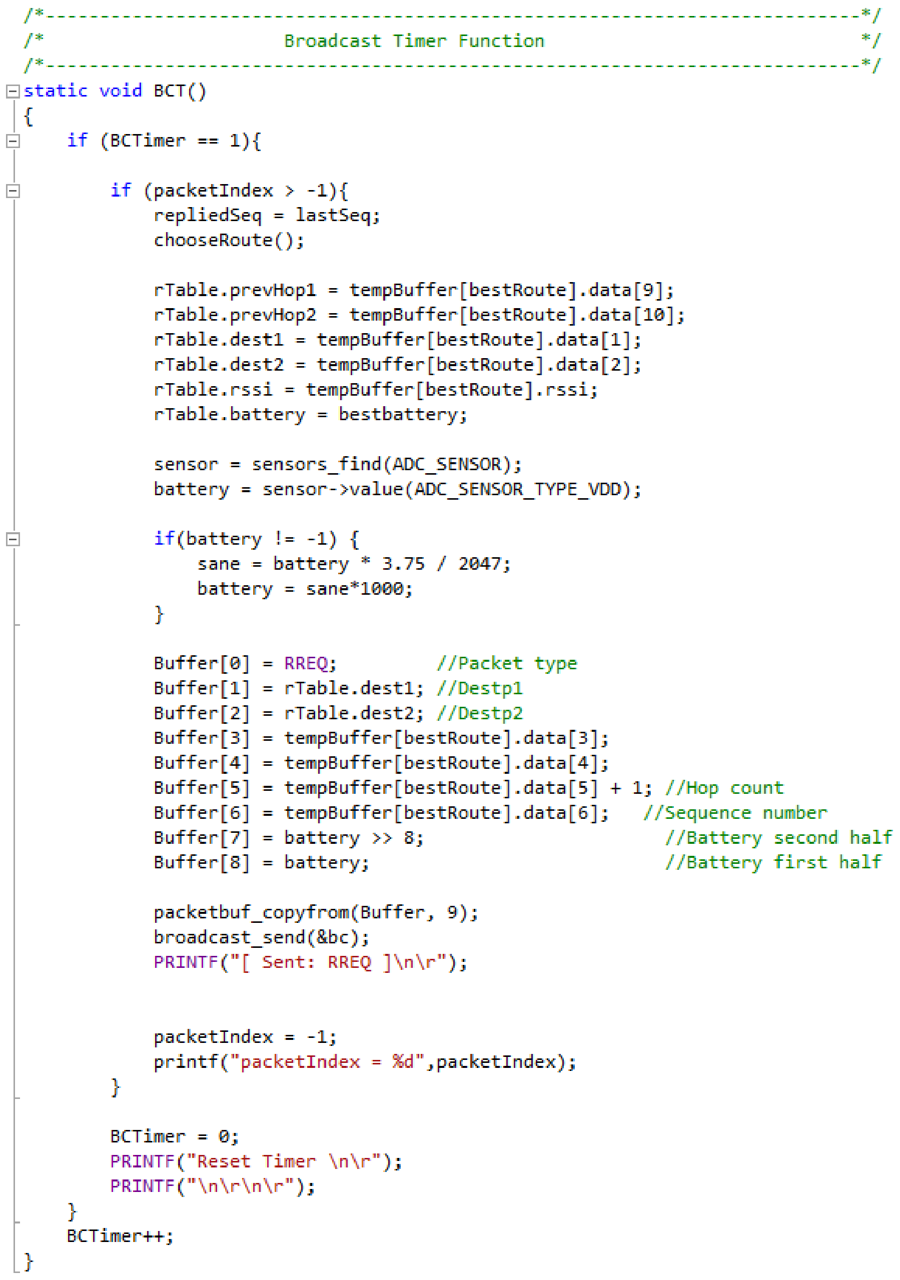
The following code snippets display the behaviour of the other nodes in the system; the intermediate and destination devices.

- This is the same as the function from the sender code, determining the optimal route of the packet being sent.



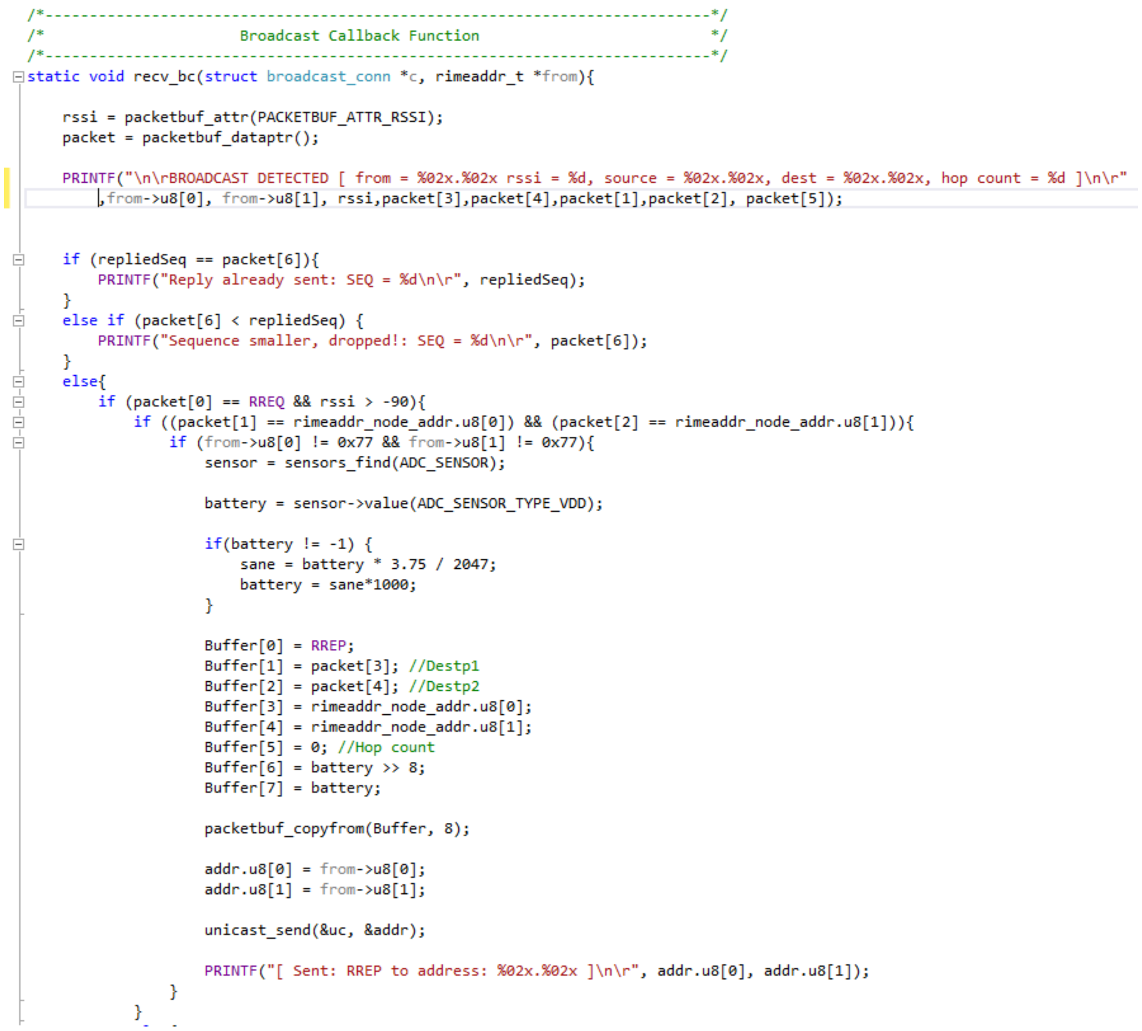
```c
/*-------------------------------------------------------------------------------*/
/*                        Choosing best Route Function                           */
/*-------------------------------------------------------------------------------*/
static void chooseRoute()
{
    int i;

    if (packetIndex == 0){
        bestRoute = 0;
        bestbattery = tempBuffer[0].data[7];
        bestbattery = bestbattery << 8;
        bestbattery = bestbattery | tempBuffer[0].data[8];
    }
    else if ((tempBuffer[0].data[3] == SOURCE) && (tempBuffer[0].data[4] == SOURCE)) {
        bestRoute = 0;
        bestbattery = tempBuffer[0].data[7];
        bestbattery = bestbattery << 8;
        bestbattery = bestbattery | tempBuffer[0].data[8];
    }
    else{
        printf("Packet Index: %d \n\r", packetIndex);

        for (i = 0; i <= (packetIndex-1); i++){
            tempbattery = tempBuffer[i].data[7];
            tempbattery = tempbattery << 8;
            tempbattery = tempbattery | tempBuffer[i].data[8];

            if (bestbattery < tempbattery){
                bestRoute = i;
                bestbattery = tempbattery;
            }
            else if (bestbattery == tempbattery)            {
                if (tempBuffer[bestRoute].rssi > tempBuffer[i].rssi){
                    bestbattery = tempbattery;
                    bestRoute = i;
                }
            }
            printf("[Index: %d,Best: %d]\n\r", i,bestbattery);
        }
    }
}
```

- The following function is triggered once it is called in the process thread, after the one second timer has expired, and then prepares it for retransmission as another RREQ. The information received from the initial message is collated to then be rebroadcast.

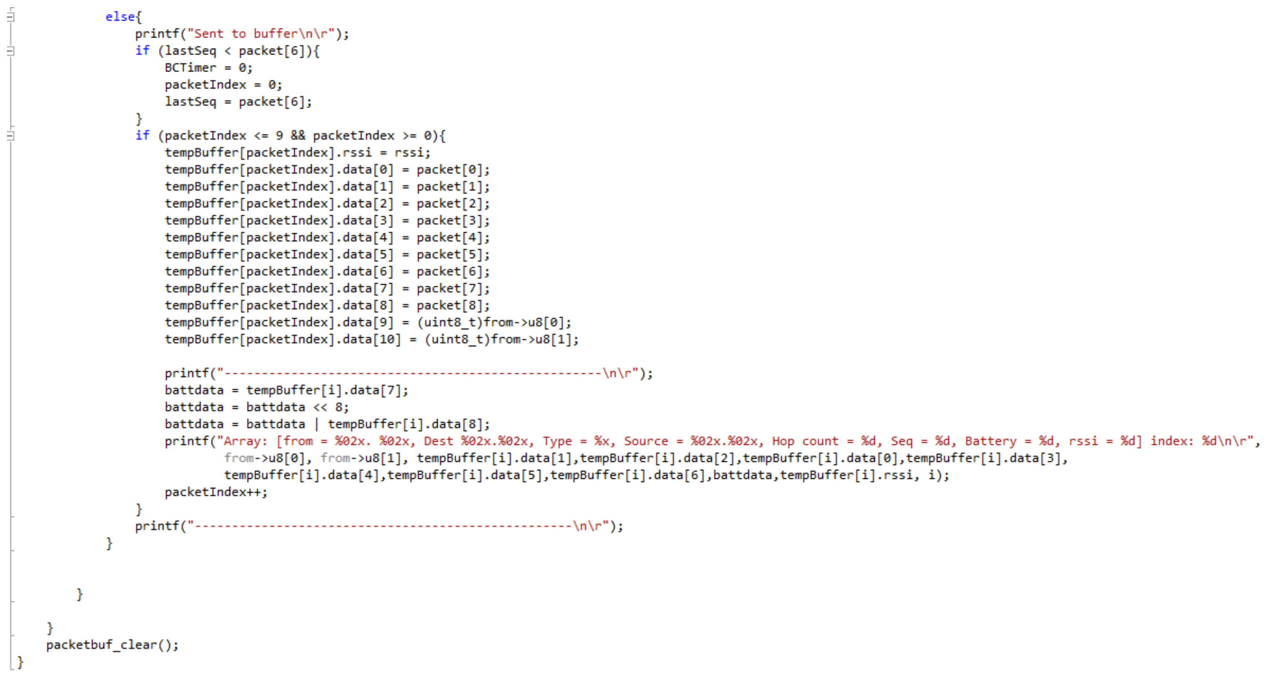


```
/*---------------------------------------------------------------------------*/
/*                      Broadcast Timer Function                             */
/*---------------------------------------------------------------------------*/
static void BCT()
{
    if (BCTimer == 1){

        if (packetIndex > -1){
            repliedSeq = lastSeq;
            chooseRoute();

            rTable.prevHop1 = tempBuffer[bestRoute].data[9];
            rTable.prevHop2 = tempBuffer[bestRoute].data[10];
            rTable.dest1 = tempBuffer[bestRoute].data[1];
            rTable.dest2 = tempBuffer[bestRoute].data[2];
            rTable.rssi = tempBuffer[bestRoute].rssi;
            rTable.battery = bestbattery;

            sensor = sensors_find(ADC_SENSOR);
            battery = sensor->value(ADC_SENSOR_TYPE_VDD);

            if(battery != -1) {
                sane = battery * 3.75 / 2047;
                battery = sane*1000;
            }

            Buffer[0] = RREQ;          //Packet type
            Buffer[1] = rTable.dest1; //Destp1
            Buffer[2] = rTable.dest2; //Destp2
            Buffer[3] = tempBuffer[bestRoute].data[3];
            Buffer[4] = tempBuffer[bestRoute].data[4];
            Buffer[5] = tempBuffer[bestRoute].data[5] + 1; //Hop count
            Buffer[6] = tempBuffer[bestRoute].data[6];   //Sequence number
            Buffer[7] = battery >> 8;                     //Battery second half
            Buffer[8] = battery;                          //Battery first half

            packetbuf_copyfrom(Buffer, 9);
            broadcast_send(&bc);
            PRINTF("[ Sent: RREQ ]\n\r");


            packetIndex = -1;
            printf("packetIndex = %d",packetIndex);
        }

        BCTimer = 0;
        PRINTF("Reset Timer \n\r");
        PRINTF("\n\r\n\r");
    }
    BCTimer++;
}
```

- Upon receipt of a broadcast, the following function is called. A check is performed to confirm whether or not the device receiving the node is the intended destination. If so, an RREP message is returned. If not, the RREQ is rebroadcast in order to head closer to the destination.

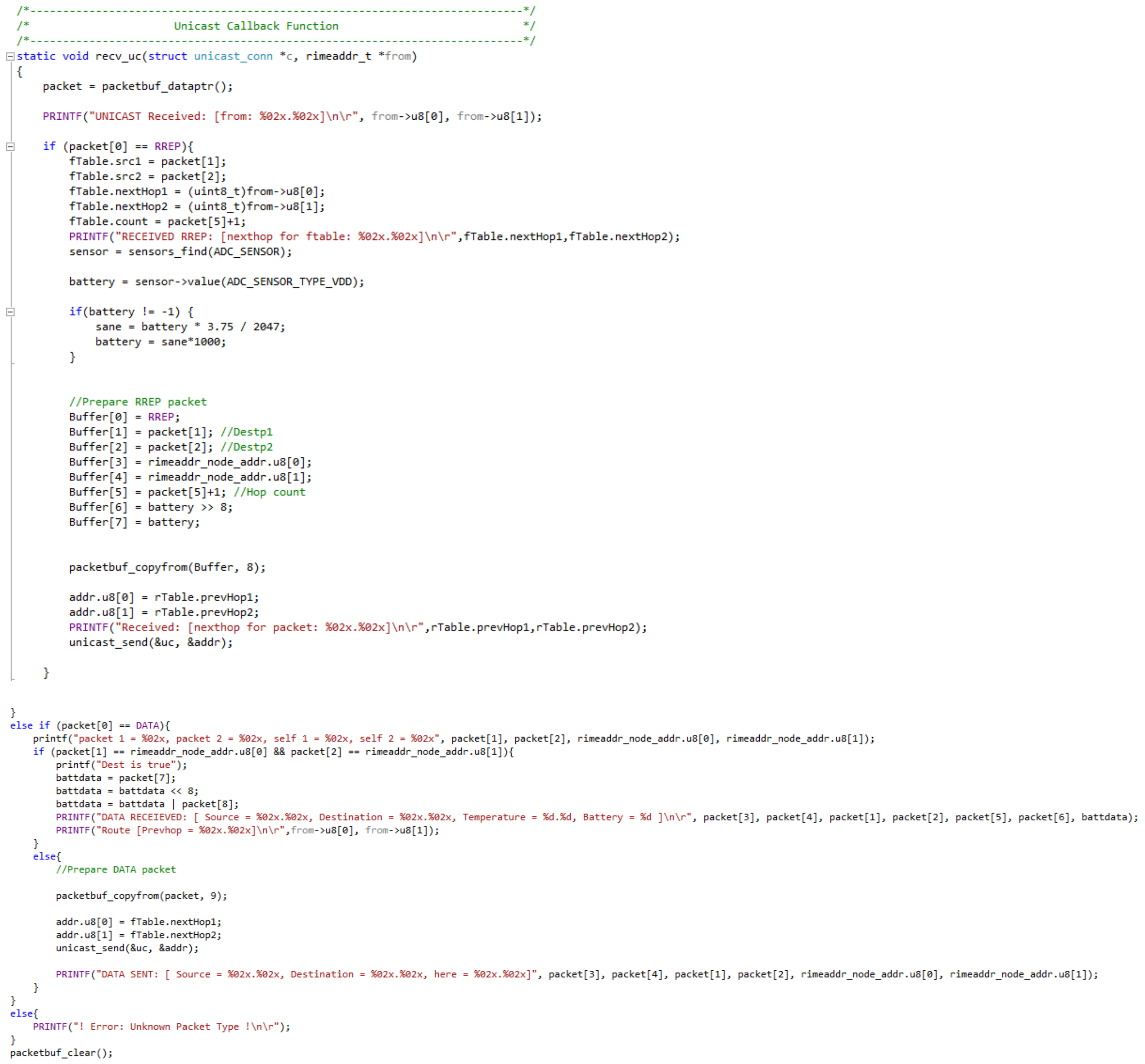


```
/*-------------------------------------------------------------------------*/
/*                      Broadcast Callback Function                        */
/*-------------------------------------------------------------------------*/
static void recv_bc(struct broadcast_conn *c, rimeaddr_t *from){

    rssi = packetbuf_attr(PACKETBUF_ATTR_RSSI);
    packet = packetbuf_dataptr();

    PRINTF("\n\rBROADCAST DETECTED [ from = %02x.%02x rssi = %d, source = %02x.%02x, dest = %02x.%02x, hop count = %d ]\n\r"
        ,from->u8[0], from->u8[1], rssi,packet[3],packet[4],packet[1],packet[2], packet[5]);


    if (repliedSeq == packet[6]){
        PRINTF("Reply already sent: SEQ = %d\n\r", repliedSeq);
    }
    else if (packet[6] < repliedSeq) {
        PRINTF("Sequence smaller, dropped!: SEQ = %d\n\r", packet[6]);
    }
    else{
        if (packet[0] == RREQ && rssi > -90){
            if ((packet[1] == rimeaddr_node_addr.u8[0]) && (packet[2] == rimeaddr_node_addr.u8[1])){
                if (from->u8[0] != 0x77 && from->u8[1] != 0x77){
                    sensor = sensors_find(ADC_SENSOR);

                    battery = sensor->value(ADC_SENSOR_TYPE_VDD);

                    if(battery != -1) {
                        sane = battery * 3.75 / 2047;
                        battery = sane*1000;
                    }

                    Buffer[0] = RREP;
                    Buffer[1] = packet[3]; //Destp1
                    Buffer[2] = packet[4]; //Destp2
                    Buffer[3] = rimeaddr_node_addr.u8[0];
                    Buffer[4] = rimeaddr_node_addr.u8[1];
                    Buffer[5] = 0; //Hop count
                    Buffer[6] = battery >> 8;
                    Buffer[7] = battery;

                    packetbuf_copyfrom(Buffer, 8);

                    addr.u8[0] = from->u8[0];
                    addr.u8[1] = from->u8[1];

                    unicast_send(&uc, &addr);

                    PRINTF("[ Sent: RREP to address: %02x.%02x ]\n\r", addr.u8[0], addr.u8[1]);
                }
            }
```



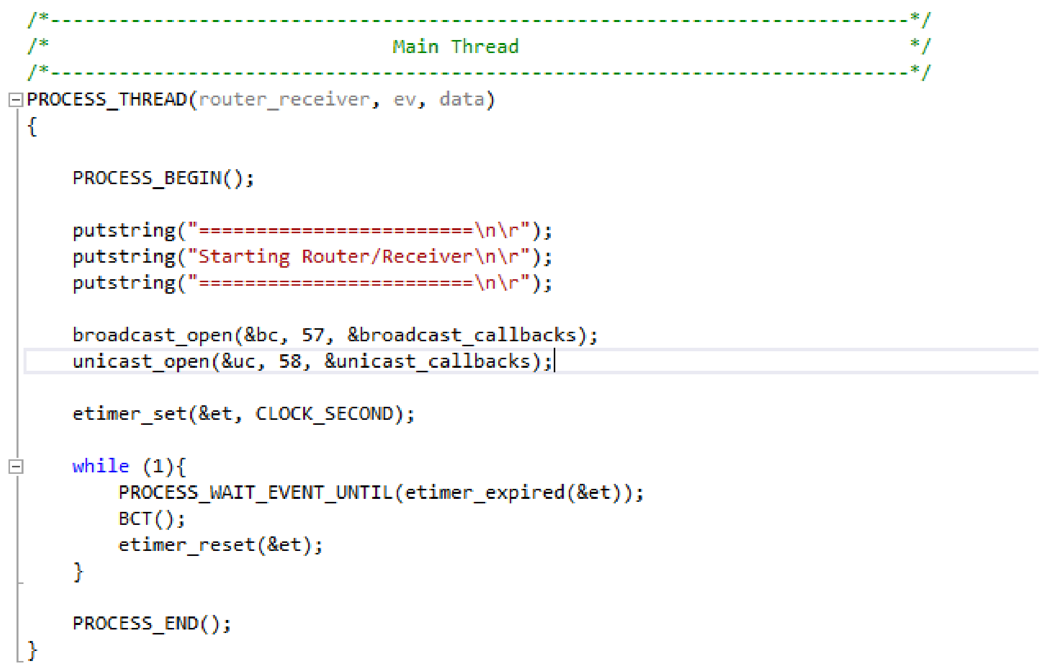
```
            else{
                printf("Sent to buffer\n\r");
                if (lastSeq < packet[6]){
                    BCTimer = 0;
                    packetIndex = 0;
                    lastSeq = packet[6];
                }
                if (packetIndex <= 9 && packetIndex >= 0){
                    tempBuffer[packetIndex].rssi = rssi;
                    tempBuffer[packetIndex].data[0] = packet[0];
                    tempBuffer[packetIndex].data[1] = packet[1];
                    tempBuffer[packetIndex].data[2] = packet[2];
                    tempBuffer[packetIndex].data[3] = packet[3];
                    tempBuffer[packetIndex].data[4] = packet[4];
                    tempBuffer[packetIndex].data[5] = packet[5];
                    tempBuffer[packetIndex].data[6] = packet[6];
                    tempBuffer[packetIndex].data[7] = packet[7];
                    tempBuffer[packetIndex].data[8] = packet[8];
                    tempBuffer[packetIndex].data[9] = (uint8_t)from->u8[0];
                    tempBuffer[packetIndex].data[10] = (uint8_t)from->u8[1];

                    printf("----------------------------------------------------\n\r");
                    battdata = tempBuffer[i].data[7];
                    battdata = battdata << 8;
                    battdata = battdata | tempBuffer[i].data[8];
                    printf("Array: [from = %02x. %02x, Dest %02x.%02x, Type = %x, Source = %02x.%02x, Hop count = %d, Seq = %d, Battery = %d, rssi = %d] index: %d\n\r",
                            from->u8[0], from->u8[1], tempBuffer[i].data[1],tempBuffer[i].data[2],tempBuffer[i].data[0],tempBuffer[i].data[3],
                            tempBuffer[i].data[4],tempBuffer[i].data[5],tempBuffer[i].data[6],battdata,tempBuffer[i].rssi, i);
                    packetIndex++;
                }
                printf("----------------------------------------------------\n\r");
            }


        }

    }
    packetbuf_clear();
}
```

- Upon receipt of a unicast message, the following function is called. The forwarding table of the node is updated to reflect the desired path of the final data packet, and the RREP packet is prepared to be sent along the path towards the original source. If the packet received is instead a DATA packet, the node checks whether it is intended to be the eventual destination of the message. If so, it prints out the data. Otherwise it will retransmit it along the predetermined optimal path, heading one hop closer to the desired destination.



```
/*---------------------------------------------------------------------------*/
/*                     Unicast Callback Function                             */
/*---------------------------------------------------------------------------*/
static void recv_uc(struct unicast_conn *c, rimeaddr_t *from)
{
    packet = packetbuf_dataptr();

    PRINTF("UNICAST Received: [from: %02x.%02x]\n\r", from->u8[0], from->u8[1]);

    if (packet[0] == RREP){
        fTable.src1 = packet[1];
        fTable.src2 = packet[2];
        fTable.nextHop1 = (uint8_t)from->u8[0];
        fTable.nextHop2 = (uint8_t)from->u8[1];
        fTable.count = packet[5]+1;
        PRINTF("RECEIVED RREP: [nexthop for ftable: %02x.%02x]\n\r",fTable.nextHop1,fTable.nextHop2);
        sensor = sensors_find(ADC_SENSOR);

        battery = sensor->value(ADC_SENSOR_TYPE_VDD);

        if(battery != -1) {
            sane = battery * 3.75 / 2047;
            battery = sane*1000;
        }


        //Prepare RREP packet
        Buffer[0] = RREP;
        Buffer[1] = packet[1]; //Destp1
        Buffer[2] = packet[2]; //Destp2
        Buffer[3] = rimeaddr_node_addr.u8[0];
        Buffer[4] = rimeaddr_node_addr.u8[1];
        Buffer[5] = packet[5]+1; //Hop count
        Buffer[6] = battery >> 8;
        Buffer[7] = battery;


        packetbuf_copyfrom(Buffer, 8);

        addr.u8[0] = rTable.prevHop1;
        addr.u8[1] = rTable.prevHop2;
        PRINTF("Received: [nexthop for packet: %02x.%02x]\n\r",rTable.prevHop1,rTable.prevHop2);
        unicast_send(&uc, &addr);

    }


}
else if (packet[0] == DATA){
    printf("packet 1 = %02x, packet 2 = %02x, self 1 = %02x, self 2 = %02x", packet[1], packet[2], rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1]);
    if (packet[1] == rimeaddr_node_addr.u8[0] && packet[2] == rimeaddr_node_addr.u8[1]){
        printf("Dest is true");
        battdata = packet[7];
        battdata = battdata << 8;
        battdata = battdata | packet[8];
        PRINTF("DATA RECEIEVED: [ Source = %02x.%02x, Destination = %02x.%02x, Temperature = %d.%d, Battery = %d ]\n\r", packet[3], packet[4], packet[1], packet[2], packet[5], packet[6], battdata);
        PRINTF("Route [Prevhop = %02x.%02x]\n\r",from->u8[0], from->u8[1]);
    }
    else{
        //Prepare DATA packet

        packetbuf_copyfrom(packet, 9);

        addr.u8[0] = fTable.nextHop1;
        addr.u8[1] = fTable.nextHop2;
        unicast_send(&uc, &addr);

        PRINTF("DATA SENT: [ Source = %02x.%02x, Destination = %02x.%02x, here = %02x.%02x]", packet[3], packet[4], packet[1], packet[2], rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1]);
    }
}
else{
    PRINTF("! Error: Unknown Packet Type !\n\r");
}
packetbuf_clear();
```

- This is the process thread of the intermediate/destination nodes. The broadcast and unicast channels are opened, to allow messages of either type to be received. A 1 second timer is started and once it expires, the BCT() function is called to rebroadcast any RREQ messages received.

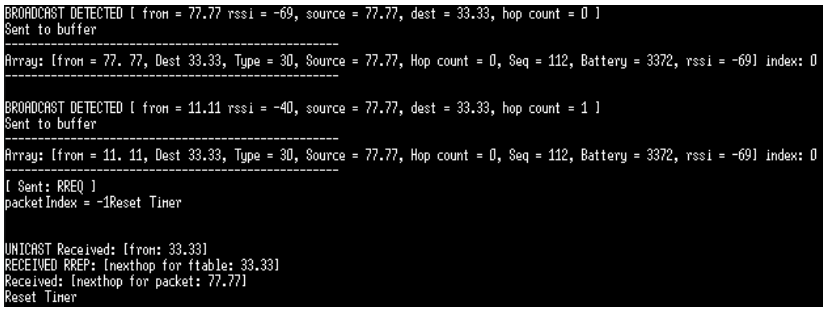


```c
/*---------------------------------------------------------------------------*/
/*                              Main Thread                                  */
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(router_receiver, ev, data)
{

    PROCESS_BEGIN();

    putstring("========================\n\r");
    putstring("Starting Router/Receiver\n\r");
    putstring("========================\n\r");

    broadcast_open(&bc, 57, &broadcast_callbacks);
    unicast_open(&uc, 58, &unicast_callbacks);

    etimer_set(&et, CLOCK_SECOND);

    while (1){
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        BCT();
        etimer_reset(&et);
    }

    PROCESS_END();
}
```

Test Procedure/Results Analysis

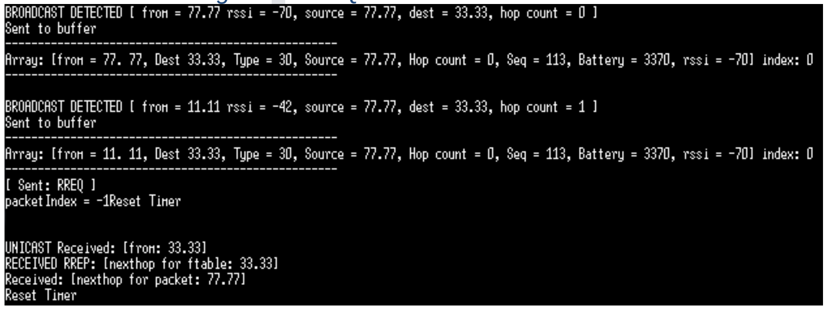
For testing purposes four nodes have been used. One node acts as a sender, two nodes act as routers or relay mediums and the last node acts as the receiver. Although, the routers and the receiver share the same code. For displaying the messages each node was connected to a separate computer using USB and the software called Tera Term. As required, four different functions have been tested below.

1. The sender node reports regularly its battery values at a set interval of 20 seconds to the other nodes in its vicinity via RREQ messages. This is also to ensure that the network structure is up-to-date. The other nodes received the packets correctly and displayed the results. The sequence number has been increased with each new RREQ request. The following pictures shows a time interval of 60 seconds whereby three RREQ packets have been sent from the sender. Noticeably, the battery level of the sender drops by 2 units every time a new RREQ packet was sent, starting from 3372 and ending with 3368.



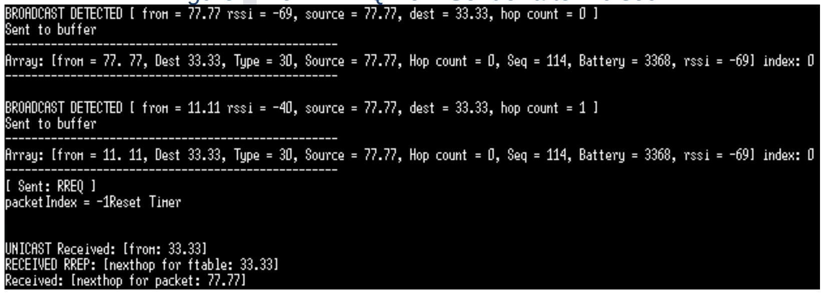
```
BROADCAST DETECTED [ from = 77.77 rssi = -69, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
--------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 112, Battery = 3372, rssi = -69] index: 0
--------------------------------------------------

BROADCAST DETECTED [ from = 11.11 rssi = -40, source = 77.77, dest = 33.33, hop count = 1 ]
Sent to buffer
--------------------------------------------------
Array: [from = 11. 11, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 112, Battery = 3372, rssi = -69] index: 0
--------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]
Reset Timer
```

Figure 1 RREQ from Sender after 20 sec



```
BROADCAST DETECTED [ from = 77.77 rssi = -70, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
--------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 113, Battery = 3370, rssi = -70] index: 0
--------------------------------------------------

BROADCAST DETECTED [ from = 11.11 rssi = -42, source = 77.77, dest = 33.33, hop count = 1 ]
Sent to buffer
--------------------------------------------------
Array: [from = 11. 11, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 113, Battery = 3370, rssi = -70] index: 0
--------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]
Reset Timer
```

Figure 2 New RREQ from Sender after 40 sec

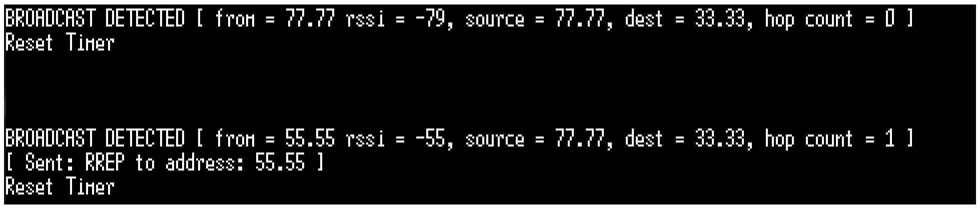


```
BROADCAST DETECTED [ from = 77.77 rssi = -69, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
--------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 114, Battery = 3368, rssi = -69] index: 0
--------------------------------------------------

BROADCAST DETECTED [ from = 11.11 rssi = -40, source = 77.77, dest = 33.33, hop count = 1 ]
Sent to buffer
--------------------------------------------------
Array: [from = 11. 11, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 114, Battery = 3368, rssi = -69] index: 0
--------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]
```

Figure 3 New RREQ from Sender after 60 sec

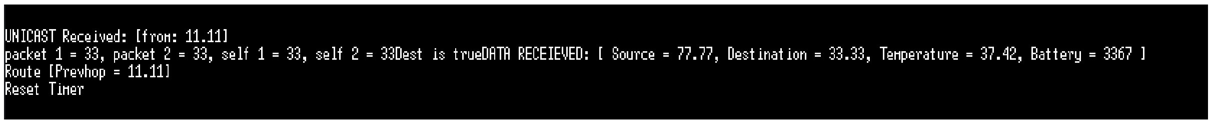
2. The buttons on the sender node were assigned to perform a new RREQ with a new sequence number and the other to send a data packet via unicast. Both buttons operate successfully performing the desired outcome. If a router goes down, then a new RREQ can be produced by pressing the B2 button instead of waiting for the 20 seconds timer to expire.

3. The destination node successfully received the packets from the nodes including the relayed data packet from the sender and displays its information in Tera Term. The following screenshots will demonstrate this.



```
BROADCAST DETECTED [ from = 77.77 rssi = -79, source = 77.77, dest = 33.33, hop count = 0 ]
Reset Timer


BROADCAST DETECTED [ from = 55.55 rssi = -55, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 55.55 ]
Reset Timer
```

Figure 4 Receiver responding to RREQ



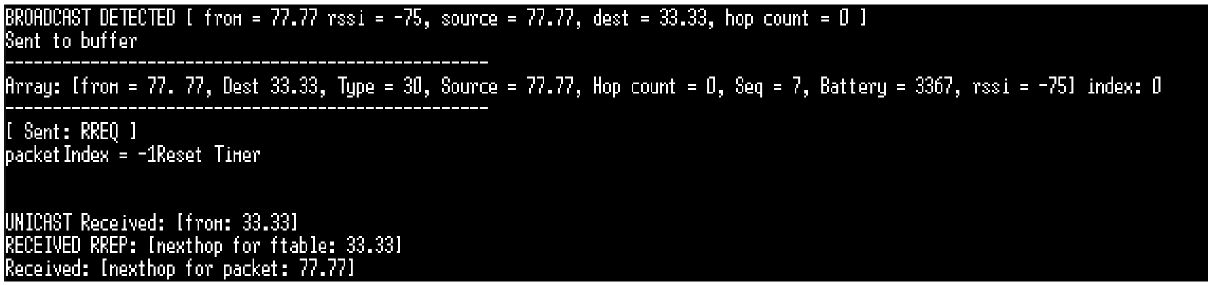
```
UNICAST Received: [from: 11.11]
packet 1 = 33, packet 2 = 33, self 1 = 33, self 2 = 33Dest is trueDATA RECEIEVED: [ Source = 77.77, Destination = 33.33, Temperature = 37.42, Battery = 3367 ]
Route [Prevhop = 11.11]
Reset Timer
```

Figure 5 Receiver displaying received data packet

From figure 4 it can be seen that the receiver ignores the RREQ from the receiver directly in order to achieve multi-hop routing. The arrival of a unicast data packet is shown in figure 5 with all of the relevant information including temperature and battery of the sender.

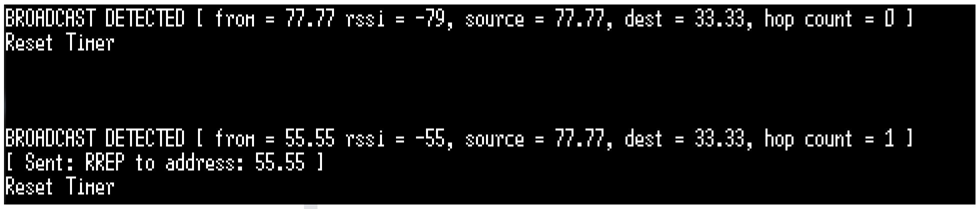
4. Multiple scenarios have been run and documented to show the outcome if the dynamic and structure of the network undergoes changes.

In the first scenario there are three nodes with one intermediate node (router). Sender with address 77.77 intends to find out route to receiver with address 33.33. Router with address 55.55 receives RREQ from sender and knows it is not the destination thus creates another RREQ. Receiver gets the RREQ from the Router and replies with a RREP which then eventually will arrive at the sender.



```
BROADCAST DETECTED [ from = 77.77 rssi = -75, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
--------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 7, Battery = 3367, rssi = -75] index: 0
--------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]
```

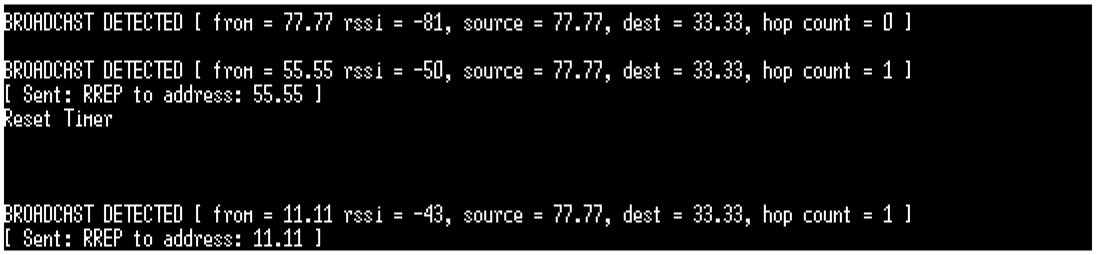
Figure 6 Router receives RREQ, sends RREQ



```
BROADCAST DETECTED [ from = 77.77 rssi = -79, source = 77.77, dest = 33.33, hop count = 0 ]
Reset Timer


BROADCAST DETECTED [ from = 55.55 rssi = -55, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 55.55 ]
Reset Timer
```

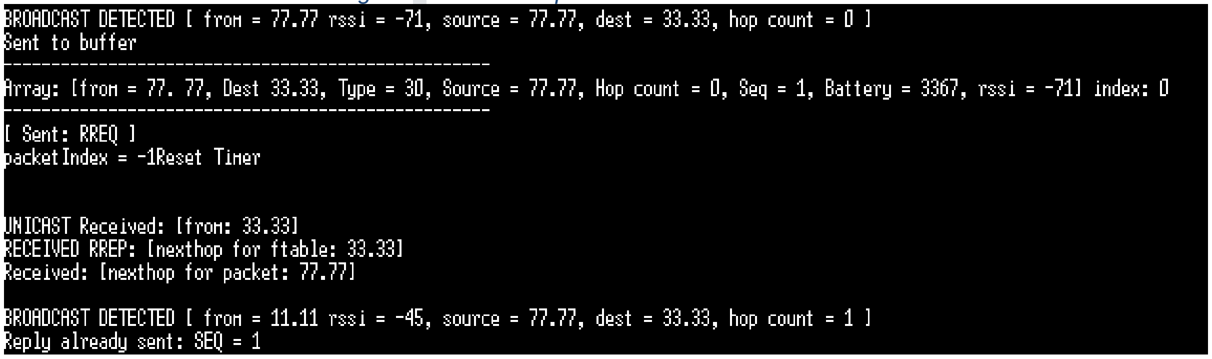
Figure 7 Receiver replies with RREP to Router 1

In the second scenario there are four nodes with two routers. Sender with address 77.77 initiates RREQ to receiver with address 33.33. Both routers receive the RREQ and send out a new RREQ. The receiver gets both RREQs and replies to both of them with RREP. The routers then forward the RREP to the sender. The sender chooses the best path based on battery level and RSSI and sends the data packet back to the receiver.



```
BROADCAST DETECTED [ from = 77.77 rssi = -81, source = 77.77, dest = 33.33, hop count = 0 ]

BROADCAST DETECTED [ from = 55.55 rssi = -50, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 55.55 ]
Reset Timer


BROADCAST DETECTED [ from = 11.11 rssi = -43, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 11.11 ]
```

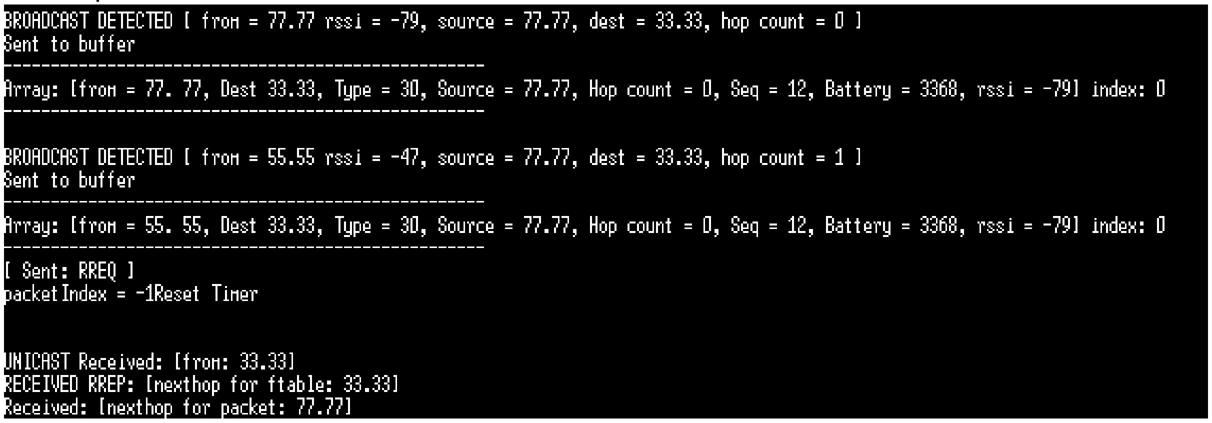
Figure 8 Receiver replies to Router 1 and Router 2



```
BROADCAST DETECTED [ from = 77.77 rssi = -71, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
---------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 1, Battery = 3367, rssi = -71] index: 0
---------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]

BROADCAST DETECTED [ from = 11.11 rssi = -45, source = 77.77, dest = 33.33, hop count = 1 ]
Reply already sent: SEQ = 1
```

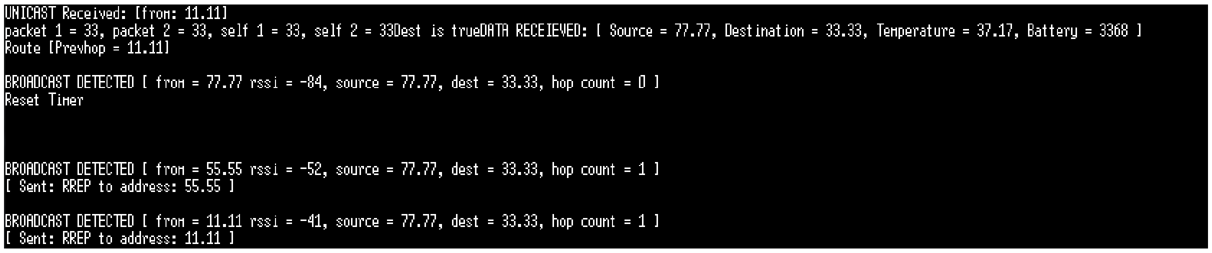
Figure 9 Router 1 response

Now the sender has received both replies from both routers it chooses the best path to send the data packet to. In this case it chose the router with the address 11.11.



```
BROADCAST DETECTED [ from = 77.77 rssi = -79, source = 77.77, dest = 33.33, hop count = 0 ]
Sent to buffer
---------------------------------------------------
Array: [from = 77. 77, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 12, Battery = 3368, rssi = -79] index: 0
---------------------------------------------------

BROADCAST DETECTED [ from = 55.55 rssi = -47, source = 77.77, dest = 33.33, hop count = 1 ]
Sent to buffer
---------------------------------------------------
Array: [from = 55. 55, Dest 33.33, Type = 30, Source = 77.77, Hop count = 0, Seq = 12, Battery = 3368, rssi = -79] index: 0
---------------------------------------------------
[ Sent: RREQ ]
packetIndex = -1Reset Timer


UNICAST Received: [from: 33.33]
RECEIVED RREP: [nexthop for ftable: 33.33]
Received: [nexthop for packet: 77.77]
```

Figure 10 Router 1 forwards data packet to receiver



```
UNICAST Received: [from: 11.11]
packet 1 = 33, packet 2 = 33, self 1 = 33, self 2 = 33Dest is trueDATA RECEIEVED: [ Source = 77.77, Destination = 33.33, Temperature = 37.17, Battery = 3368 ]
Route [Prevhop = 11.11]

BROADCAST DETECTED [ from = 77.77 rssi = -84, source = 77.77, dest = 33.33, hop count = 0 ]
Reset Timer



BROADCAST DETECTED [ from = 55.55 rssi = -52, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 55.55 ]

BROADCAST DETECTED [ from = 11.11 rssi = -41, source = 77.77, dest = 33.33, hop count = 1 ]
[ Sent: RREP to address: 11.11 ]
```

Figure 11 Receiver receives unicast data packet and prints it

Commentary

Initially, the dataset was based solely around structs and the system which comes with transferring a type of struct and creating an array of structs (being the array of RREQ's). Eventually it was decided that not only was this system needlessly difficult to manage, but it was also much worse with memory management. The decision was taken to change the system at the very beginning and instead of transmitting structs that would need to be assembled into arrays of structs, the system would instead utilise (in effect) a buffer system, but is in reality an array.

The first implementation consisted of a system which allowed the Destination node to decide the route taken for the data packet, in this vain of thought, we proposed that the Sender node would send out a Route Request (RREQ) to all of the intermediate nodes in range. The intermediate nodes that received that RREQ would then put the Sender in their reverse table. The intermediate nodes would then relay that RREQ (crucially, with the same sequence number) to the nodes that they could also reach. Any node that received that RREQ would put the RREQs into an array and choose the best previous hop based on firstly the battery life, but if more than one node had the same battery life remaining, it would decide on RSSI, which we deemed was volatile enough to never be the same, if battery life was ever the same, which the group also doubted would ever be the case.

From there, the intermediate node would broadcast the RREQ (incrementing the hop count) to their neighbours, assuming one of those was the Destination, it would act the same way as a normal router. It would wait a period of four seconds, receive any RREQs, look at the battery life of each and choose the best, otherwise it would rely on utilising the RRSI reading, to put into its forwarding table.

If an intermediate node broadcasted the RREQ to another intermediate node that had already received the same sequence numbered RREQ, then that node would not drop that packet, but once again perform all of the relevant checks on the packet and node before changing (or not) the reverse table. This caused issues such as infinite loops, as two nodes might have each other in their reverse tables.

The Destination would then reply with a Route Reply (RREP), and would send it using its own forwarding table. Any node it reached would then utilise the reverse table it had and send it on that route. As soon as the RREP reached the Sender node, the Sender node would then send the data along that very same route. This meant that there could only ever be a single route, and the Destination (effectively) decided which route to take.

The second major implementation attempt focused around solving the loop issues, and also around solving the issue with the intermediate nodes re-broadcasting an RREQ even if they had already received and sent one. For these two major issues, there was a single fix that remedied both. The group implemented a system whereby if a sequence number had passed through a node even once, any other RREQ's with the same sequence number would be immediately

dropped and that intermediate node would take no other actions. This meant that in a single edit, the group had solved the two pressing issues with the second implementation.
The third, and final implementation moved the mechanism for the Destination choosing the route, to the Sender determining the route. This was required because the coursework specification demanded it. To this end, the changes that were made meant that as the Sender sent out the initial RREQ, the nodes that received that very first RREQ would reply with an RREP themselves with their battery and RSSI information, and as such, the Sender would then decide the destination for the very first hop.

If the group had the time to create the program again, it is clear that it would settle on a set design for the implementation of the packet structure (such as using an array buffer structure instead of a struct structure), as this wasted valuable time which could have otherwise been spent doing more important tasks.