



Loughborough  
University

COP531  
WIRELESS NETWORKS

---

System Design Report

---

*Authors:*

AMEER GHAFFORI  
EKOW MENSAH  
HUANG YIWEN  
RIHUI WANG  
ZHENG ZIHENG

February 2, 2018

# Contents

<b>1</b>	<b>Problem Definition</b>	<b>2</b>
1.1	Background . . . . .	2
1.1.1	Protocol stack for wireless sensor networks . . . . .	2
1.2	The Task . . . . .	4
<b>2</b>	<b>System Design</b>	<b>5</b>
2.1	Our protocol stack . . . . .	5
2.2	Routing Principle . . . . .	5
2.3	Embedded Software Design (Flowcharts) . . . . .	7
<b>3</b>	<b>Test Procedure and Evaluation</b>	<b>9</b>
3.1	Description . . . . .	9
3.2	Fundamental Test . . . . .	10
3.2.1	Broadcast Test (RREQ) . . . . .	11
3.2.2	Unicast(RREP) test . . . . .	12
3.2.3	Data transfer test . . . . .	13
3.3	Extra Tests . . . . .	14
3.3.1	All Routers Switched On . . . . .	14
3.3.2	Router1 switched off . . . . .	15
<b>4</b>	<b>Conclusion</b>	<b>16</b>
4.1	Next Steps . . . . .	16
<b>5</b>	<b>Reference List</b>	<b>18</b>
<b>6</b>	<b>Appendix</b>	<b>19</b>
6.1	Code for sender . . . . .	19
6.2	Code for router . . . . .	23
6.3	Code for receiver . . . . .	29

# 1 Problem Definition

Technology has become one of the most important aspects of human life. The need for technology and implementation of new technologies and innovations increases daily and annually. The Internet and networking some of the most vital aspects of technology which directly affect various areas of human activities such as work, education, entertainment etc (Yang,2014). Wireless networks are an advanced and modern technology that been found as an alternative to wired technology which uses cables and wires to connect digital devices. A lot of technologies been found and developed in order to support the wireless network technology, such as 3G, 4G, Wifi, Bluetooth, and Zigbee.

## 1.1 Background

Wireless sensor networks (WSNs) can be explained as a collection of actuators and sensors which are responsible for controlling and monitoring the remote environment. Situated at various locations, each sensor within the network collectively transports the data it senses to the main actuator via the network (Yang, 2014). Sohrabi (2014) explains a mobile ad-hoc wireless network as a type of wireless sensor network in which nodes locate neighbours and establish wireless links in order to facilitate transmission and receipt of data in their own way. Furthermore, the network is self-healing and self-forming as it has the ability to adapt to topological changes (addition and removal of nodes) as well as changes in network traffic. The application of wireless sensor networks can be split into two main categories. These are remote tracking and remote monitoring applications and these categories can be further divided into indoor and outdoor tracking and monitoring. Examples include, tracking the enemy's movement in war, detecting forest fires, tracking animal movement within a zoo, monitoring and sensing chemical levels within a factory environment etc (Yang, 2014).

### 1.1.1 Protocol stack for wireless sensor networks

The protocol stack for wireless sensor networks is similar to that of the Open Systems Interconnection (OSI) model but differs in the fact, it does not implement all seven layers of the OSI protocol stack. Rather, this protocol stack implements five of the seven layers of the OSI protocol stack that is, the application layer, transport layer, network layer, data link layer and physical layer. 5 layers are used because, it is not practical to use seven layers in a real WSN application as it becomes very complex and challenging to implement (Yang, 2014). A block diagram of the WSN protocol stack can be seen below in Figure 1.0.

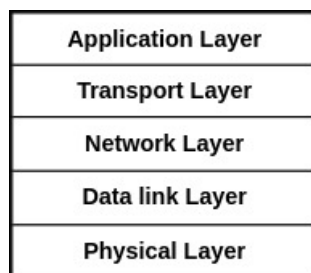


Figure 1: Protocol stack for WSNs

## **Physical Layer**

The physical layer handles the definition and management of connections between nodes, additionally, it handles the selection and creation of the carrier frequency, data encryption and frequency modulation (Yang, 2014). Moreover, the physical layer is responsible for the delivery of messages from node to node (Harish & Sambasivan, 2013).

## **Data Link Layer**

The data link layer ensures the provision of services which enable the access and sharing of a communication medium by several nodes within the network. Examples of these services are media access control (MAC) and error detection and correction (Yang, 2014). Furthermore, Famry (2016) explains the need for a MAC protocol to handle issues pertaining to data-centric routing and power conservation. The MAC protocol aims to establish a network architecture that is, providing communication links between nodes including self organising abilities and to share communication resources amongst all nodes in the most reasonable and effective manner. A well designed MAC protocol enables channel access in a manner which preserves battery life and improves quality of service.

## **Network Layer**

The network layer handles the successful routing of packets between nodes including the creation of communication paths between the nodes. The choice of routing protocol controls the manner in which the communication paths are established as each protocol has its own mechanism of determining routes (Yang, 2014). For instance, some routing protocols may decide to select routing paths which will preserve the lifetime of the network whilst others might select routing paths to provide optimum quality of service or if possible, the routing protocol will aim to achieve both (Yang, 2014). The simplest design is to have each node which receives data repeat broadcasts to every neighbour until the destination node receives the data or the maximum hop lifetime has been attained. This approach is known as flooding (Famry, 2016). Although flooding is advantageous in that it is simple and does not require the maintenance of the topology, it is wasteful as it does not consider efficient energy use and causes overlaps between nodes. A relatively better approach to flooding is gossiping. With gossiping, a node arbitrarily chooses a neighbouring node to transmit data to when it receives the data. In spite of the fact that gossiping does solve problem of implosion (i.e. duplicated data being transmitted to the same node), it causes transmission delays (Famry, 2016). Other alternatives that can be used are flat routing protocols such as the Sensor Protocol for Information via Negotiation (SPIN), Direct Diffusion (DD), hierarchical routing protocols such as Low-Energy Adaptive Clustering Hierarchy (LEACH), location based routing protocols such as Geographic Adaptive Fidelity (GAF) and many more. A routing protocol is selected based on the nature of the application of the wireless sensor network.

## Transport Layer

The transport layer oversees steadfast communication between end points. While various transport layer protocols exist, Transmission Control Protocol (TCP) and User Datagram Protocol(UDP) are the most widely used protocols on the transport layer. Transmission control protocol provides reliable data transfer and other services which include congestion control, flow control and error detection (Yang, 2014). Unlike TCP, UDP does not provide reliable data transfer or congestion control, moreover, it does not keep track of connections or the order in which packets arrive.

## Application Layer

The application layer is the first layer of the protocol stack. The application layer handles encryption and processing of the data that was sensed. It is also responsible for the controlling the manner in which data is stored (Yang, 2014). Additionally, before data is transmitted down the protocol stack, the application layer checks each layer to find out whether each layer possesses the requisite resources required to perform the user's request (Yang, 2014).

## 1.2 The Task

This report explains the design of a wireless mobile ad-hoc network which makes use of a simplified version of the ad-hoc distance vector routing protocol (AODV). The network was comprised of four nodes that is, a source node, two intermediate nodes and a destination node. The source node contained both a temperature and voltage sensor and was responsible for transmitting data (temperature  $^{\circ}\text{C}$  and battery level  $V$ ) to the destination node, while the intermediate nodes were responsible for relaying packets between the source and destination nodes. The destination node being the coordinator was connected to a dedicated computer and was responsible for displaying the information sent from the source node. The temperature and voltage readings obtained by the source node was transmitted to the destination node every two seconds via the intermediate nodes. The source node selected the next hop node during transmission of the data based on the battery level and radio signal strength indicator (RSSSI) by analysing the information received by the intermediate nodes. This was done in order to make the ad-hoc network more resilient and long lasting. This will be explained in more detail in the system design section. The functions that the wireless ad-hoc network is required to perform can be described as follows:

- Each node is required to display the battery level and temperature readings that has been transmitted by the source node.
- The temperature readings and battery level should be displayed when it is requested on demand at the press of a button.
- The destination node should be able to display the received message (i.e. temperature and battery level received from sender) on a dedicated computer and the routes it came from.
- When a condition of an intermediate node changes, the sending node must be notified via a route

response and change the next hop appropriately.

## 2 System Design

This section discusses the design of the mobile ad-hoc wireless network(MANET) including the routing algorithm and a flowchart of the system to be implemented.

### 2.1 Our protocol stack

In this system, a simple wireless sensor network consisting of four sensor nodes is designed and implemented using the Contiki Operation System. That is, a sender node S, two router nodes T1 and T2, and one receiver node R as shown in figure 2.

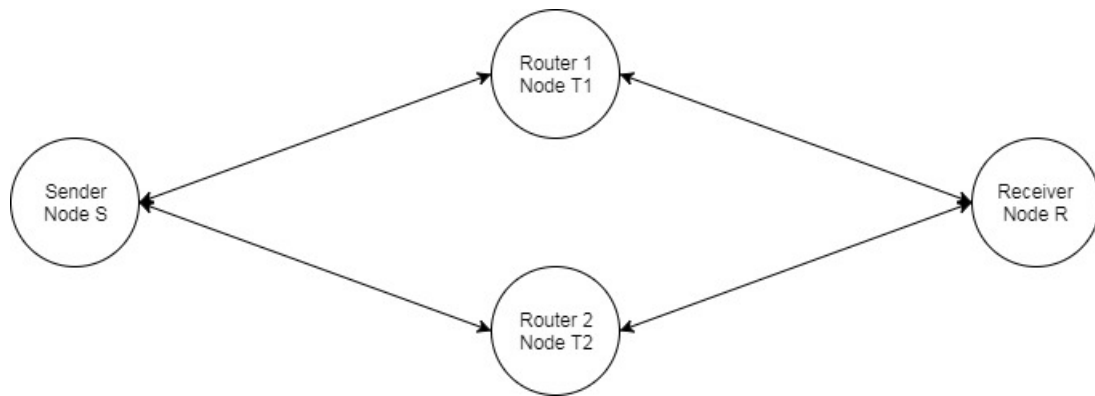


Figure 2: A 4 nodes WNS

When the node S sends data to the destination node R for the first time, there is no available route in the routing table. Then the source node S broadcasts a router request packet (RREQ) across the network and every node within the communication range can receive the request. If an intermediate node receives a RREQ packet, it sets up a reverse route toward to the node where the RREQ packet is received from. Then, the current intermediate node increases the hop count and rebroadcasts RREQ if it has no route to the destination node. Until the RREQ is received by the destination node, the node sends back a route reply packet (RREP) by following the reverse route. When the intermediate node receives the RREP, it sets up a forward table including the next hop to the destination and re-unicast the RREP to the source node. An available route is being built when the source node receives the RREP and then the data packet can be sent according to the forward table in each node. This is the route discovery mechanism to find any possible paths from the source device to the destination device.

### 2.2 Routing Principle

When a route discovery is required, the sender broadcasts RREQ and waits for the responding from the destination. A routing algorithm need to be designed to figure out the best path if the available routes

are more than once. In this system, the battery level and received signal strength indication (RSSI) of the intermediate node will be gathered and send to the source device. The source device compares the battery level with the average level. Then the source node compares the RSSI if the battery level of the intermediate nodes is higher than the average level and choose the route with higher signal strength. If the battery level of the intermediate nodes is lower than the average level, the source node will select the route with higher battery level. The whole process to figure out the best route is shown in figure 3.

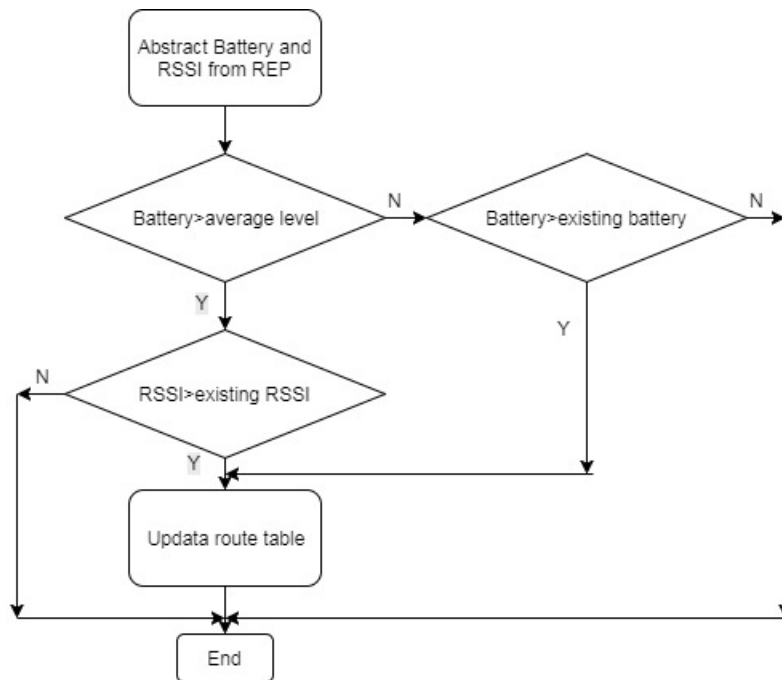


Figure 3: The process to find the best path (Yang, 2014)

For the RREQ, a broadcast count is included to guarantee the data packet sent by the source device takes two or more hops to the destination. The destination node only processes the RREQ with the number of count higher than 1. Meanwhile, the broadcast will be deleted if the number of count is higher than 3 to avoid implosion.

There may be circumstances that the signal strength from the intermediate nodes changes. Therefore, the routing table should be updated in real time, which means the sender node keeps sending RREQ messages all the time to guarantee the best path is always stored in the routing tables.

There may be circumstances that the intermediate node runs out of battery, but the sender still sends data packet to this intermediate node since the routing table is not updated. Therefore, a data identification number is created in the data packet and transmitted along with the data. When the destination node receives the data packet, it identifies the data identification number and sends it back to the source node along with the RREP messages. The routing table will be reset if the source node finds that the identification number received from the destination node is smaller than the current identification number by three, which means the lost data packets are more than three.

## 2.3 Embedded Software Design (Flowcharts)

There are three types of nodes in this system: sender, router and receiver. For the sender, it is capable of sending data packet or RREQ, and receiving the RREP through unicast from the intermediate nodes. To be more detail, in this system, the sender broadcasts the RREQ every two seconds to all its neighbours within the communication range. When the sender receives the unicast of RREP from the router, it updates the routing table according to battery level and RSSI of the intermediate node to guarantee the routing table always keeps the best path. Then the sender unicasts the data packet every two seconds to the next hop stored in the routing table. The flow chart for the sender is shown in figure 4.

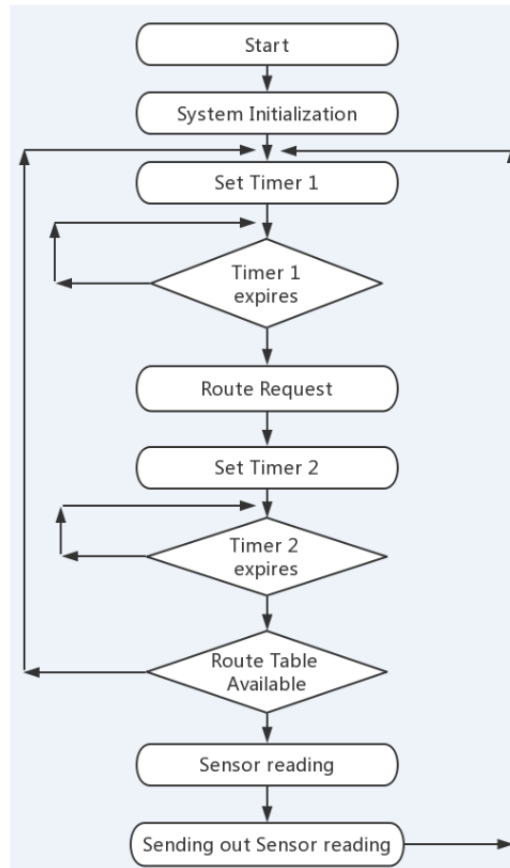


Figure 4: Flowchart of the sender (Yang, 2014)

For the router, it rebroadcasts the RREQ if the count number is lower than three when it receives the RREQ from the source node. If a unicast message is received, then it retransmits the RREP packet or the data packet to the source node and destination respectively. The flow chart for the router is shown in figure 5.

For the receiver, print out the data message when it receives a unicast from the intermediate node and send back a RREP when it receives the broadcast. The flow chart of the receiver is shown in figure 6.



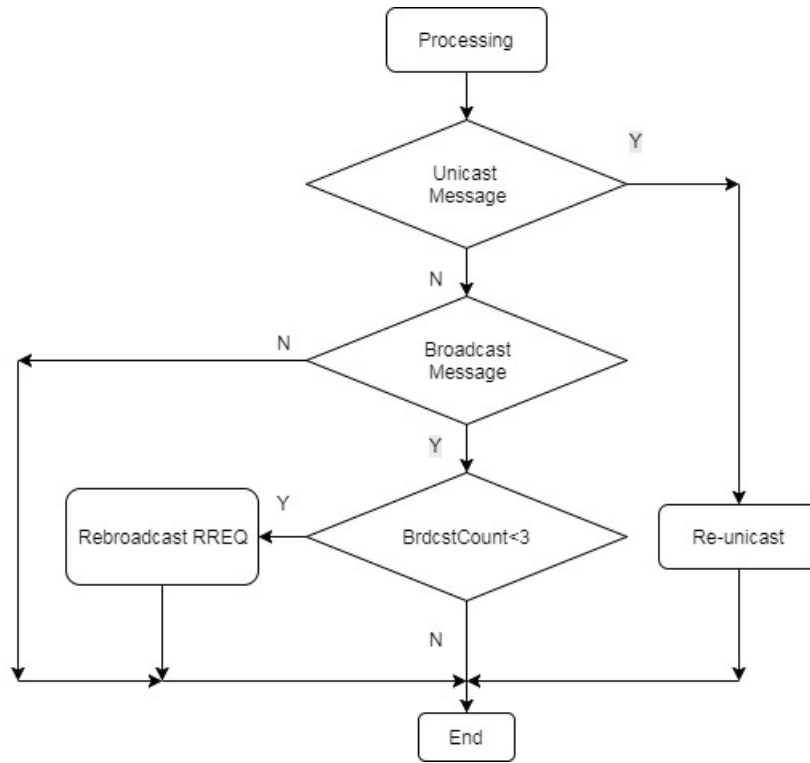


Figure 5: The flowchart of the router

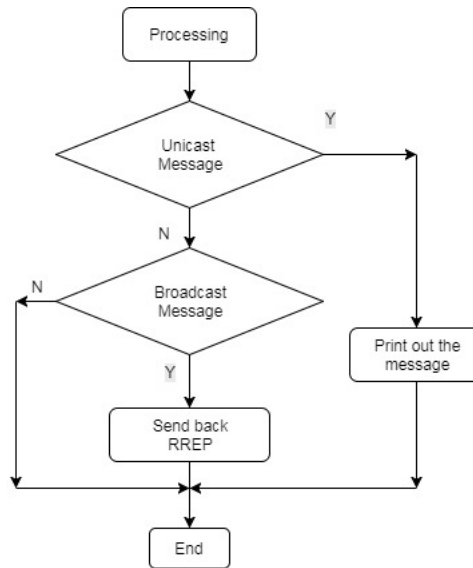


Figure 6: The flowchart of the receiver

### The type of messages: (Yang, 2014)

Route Request Message:

Type (8 bit): identifies the type of message (COMMAND\_ROUTEREQUEST)

Destination address (16 bit): the Rime address of the destination for which a route is required

Broadcast counter (8 bit): specifies how many times the current message has been sent

Broadcast limit (8 bit): the maximum number of times a message can sent by broadcast

Broadcast id (8 bit): identification number of the current broadcast message

Route Reply Message:

Type (8 bit): identifies the type of message (COMMAND\_ROUTERESPONSE)

Destination address (16 bit): the Rime address of the destination for which a route is given

Battery (16 bit): the battery level of the destination node

Nomessage (8 bit): the identification of the data id received in the receiver

Data Transmission Message:

Type (8 bit): identifies the type of message (COMMAND\_DATATX)

Destination address (16 bit): the Rime address of the destination of the sensed data

Source address (16 bit): the Rime address of the source of the transmitted data

Temperature (16 bit): the sensed temperature data to be transmitted

Battery (16 bit): the battery level data to be transmitted

Data id (8 bit): identification number of the current broadcast message

Reverse table:

BrdstID: to identify the broadcast message (8 bit)

From: the source device from which the route request is obtained (16 bit)

Dest: the address of the destination for which a route is required (16 bit)

Routing Table:

Dest: the address of the destination (16 bit)

NextHop: The next hop required to reach the destination (16 bit)

Battery: Battery level of the nodes (16 bit)

RSSI: The received signal strength indication (16 bit)

## 3 Test Procedure and Evaluation

### 3.1 Description

There are mainly two kinds of testing. In the first testing, four sensor nodes are used, one sender, two routers and one receiver respectively. Figure 7 shows the topology of the first test. And this test prove that this wireless system implement the fundamental functions.

Totally, there are three conditions. The first condition is both of router1 and router2 are switched on. The second one is router1 is switched off and router2 is switched on. The last condition is that router2 is switched off and router1 is switched on. In each condition three functions, namely broadcast(RREQ),

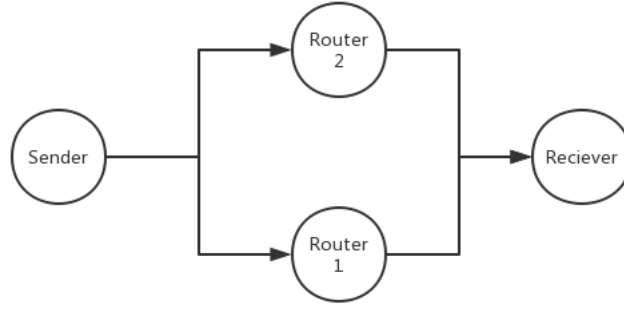


Figure 7: Topology in fundamental test

unicast(RREP) and the send data are tested. Except for the fundamental functions test, more sensor nodes are joined to the system to prove that this system implement some extra functions. The figure shows the topology of the extra test.

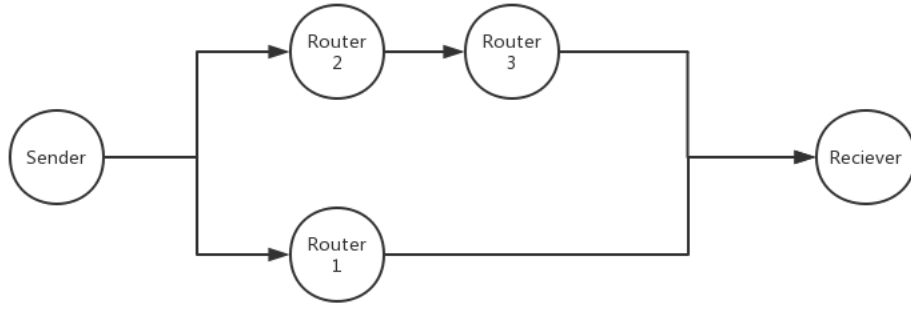


Figure 8: Topology in extra test

It is clear that there are three routers. There are two conditions to be tested. The first one is that all routers are switched on. The second condition is that the router1 is switched off. Similar to the fundamental test, three functions, broadcast(RREQ), unicast(RREP) and the send data are tested. To make the test easier to understand, all the packet information is displayed by all sensor node including sending packet out and receiving packet and will be shown as screenshots.

### 3.2 Fundamental Test

As shown in the figure 7, there are four sensor nodes. MAC address of all sensor nodes are modified. It is shown in the table 1.

Sensor node	MAC address
Sender	00::00:AA:AA
Router1	00::00:BB:BB
Router2	00::00:CC:CC
Receiver	00::00:DD:DD

Table 1: Table showing the sensor node and corresponding MAC address

### 3.2.1 Broadcast Test (RREQ)

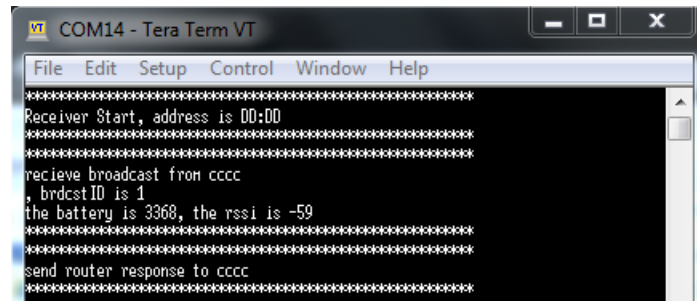
When all sensor nodes are switched on, the broadcast(RREQ) should be sent out by the sender which address is AA:AA. Then the broadcast will be received by two routers and two routers rebroadcast.

Figure 9: Sender transmits broadcast

Figure 10: Router received the broadcast

As the figure 9 shown, two broadcast is sent out and broadcast ID is 1 and 2. In figure 10, it is clear that routers receive the broadcast and the packet is from AA:AA which is the address of sender. When the routers receive the broadcast from sender it need to rebroadcast. And the broadcast ID is unchanged. Therefore, it is proved that the broadcast function is implemented by the sender and routers. What is needed to be explained is the broadcast ID which is used to avoid the broadcast storm. When the

routers receive broadcast, it need to rebroadcast to find the next node. Rebroadcast causes that the routers receive the same broadcast from other routers and rebroadcast again. Obviously it will be an infinite loop. To solve this problem. The broadcast ID is used. When the routers receive the broadcast, it will check the broadcast ID firstly. If it is equal to the old one, routers will update the tables but not broadcast it again. When routers receive the broadcast and rebroadcast. The receiver should receive broadcast. The figure 11 shows that the receiver receives the broadcast from routers.



```

VT COM14 - Tera Term VT
File Edit Setup Control Window Help
*****
Receiver Start, address is 00:00
*****
recieve broadcast from cccc
, brdcstID is 1
the battery is 3368, the rssi is -59
*****
send router response to cccc
*****

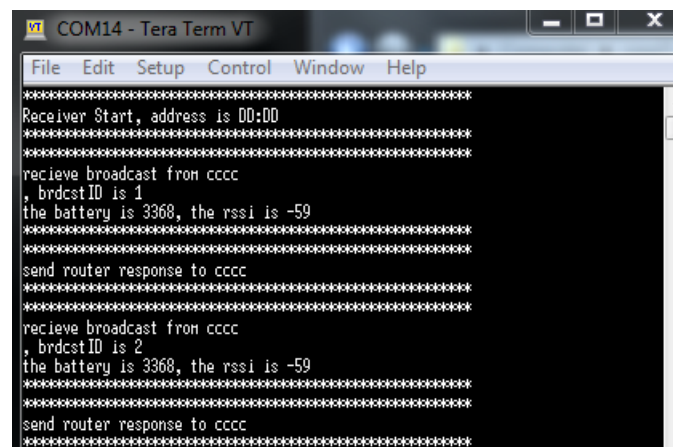
```

Figure 11: Receiver getting broadcast

There is a problem that the receiver only can receive one broadcast from router2(CC:CC). The reason is that the router2 is closer to the receiver and the receiver get the broadcast from router2 first. And the receiver will handle the broadcast which causes receiver cannot receiver other broadcast like the broadcast from router1(BB:BB).

### 3.2.2 Unicast(RREP) test

When the receiver get the unicast, it should be able to solve the broadcast and choose a better router according to hop counts battery and RSSI. In this case the hop counts are the same in the two path. Therefore only battery and RSSI need to be considered.



```

VT COM14 - Tera Term VT
File Edit Setup Control Window Help
*****
Receiver Start, address is 00:00
*****
recieve broadcast from cccc
, brdcstID is 1
the battery is 3368, the rssi is -59
*****
send router response to cccc
*****
recieve broadcast from cccc
, brdcstID is 2
the battery is 3368, the rssi is -59
*****
send router response to cccc
*****

```

Figure 12: Unicast to router

From figure 12 it is clear that the receiver chooses the router2(CC:CC) and send unicast to it. What is needed to be declared is that the hop-counts has the best priority. Battery has the second priority and the RSSI is the last. In this case the hop-counts are the same and the battery need to be compared. The

router2 has higher battery so it is chosen. After that the router2 need to receive the unicast and send another unicast to the sender.

```

COM4 - Tera Term VT
File Edit Setup Control Window Help
Welcome to Contiki 2.4.
Running on: N740 NanoSensor (CC2431-F128).

*****
Router Start, address is BB:BB
*****
recieve broadcast from aaaa,
brdcstID is 1
*****
send broadcast, brdcstID is 1
*****
recive router response from dddd
send router repsonse to a new router aaaa
*****
recieve broadcast from aaaa

```

Figure 13: Unicast to router

From figure 13 it is clear that the router2 receive the RREP from the router (DD:DD) and send it to the last hop which is the receiver (AA:AA). After the router send the RREP to the sender, the path from sender and receiver is established which means the routing process is finished.

### 3.2.3 Data transfer test

After the path is established, it is supposed that the sender can send to the router using the path. In this test, sender sent data packet continuously to the receiver by push the button2.

```

COM13 - Tera Term VT
File Edit Setup Control Window Help
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****
send the data to cccc
temperature1 is 28
*****

```

Figure 14: Data packet from sender

From figure 14 it is clear that the data is sent to router2(CC:CC) first. Because it was chosen in the routing process.

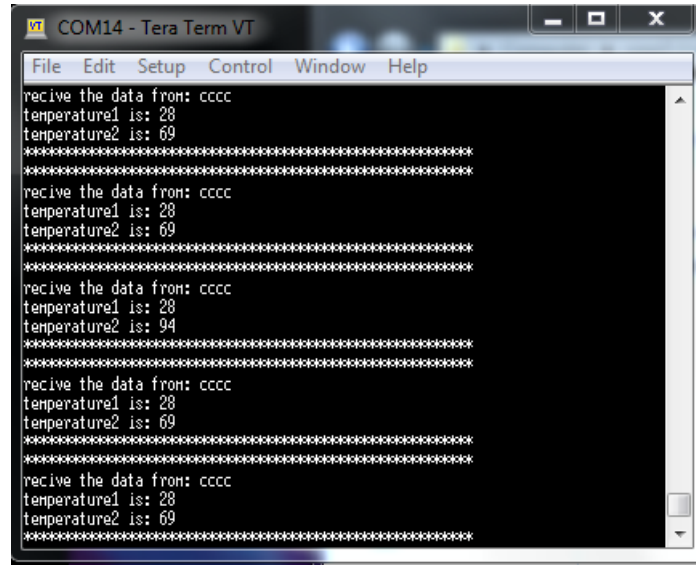


Figure 15: Receiver receives data

From figure 15, it is clear that the content of the data is the same as what is sent out from the sender. In this test, only the temperature is chosen as the testing feature. And the data is from the router2(CC:CC). It proves that there is only one path in the topology which means the path is established successfully.

### 3.3 Extra Tests

In the extra test, there are five routers in the topology. From figure 8, it can be seen that there are three routers, one receiver and one sender. In this part, two conditions will be tested. The first one is all router are switched on and second one is that the router1 is switched off. This test aims to prove that router can choose shorter path and when the path is broken it can establish new path. Details in the test will not be discussed and some main parts will be introduced. Address of each sensor node is displayed in the table2.

Sensor node	MAC address
Sender	00::00:AA:AA
Router1	00::00:BB:BB
Router2	00::00:CC:CC
Router3	00::00:FF:FF
Receiver	00::00:DD:DD

Table 2: Address of each sensor node in the test

#### 3.3.1 All Routers Switched On

In this test, when the path is established, the sender need to send data to router1 first.

```

COM4 - Tera Term VT
File Edit Setup Control Window Help
*****
Router Start, address is BB:BB
*****
recieve data from aaaa
send data to dddd
*****
recieve data from aaaa
send data to dddd
*****
recieve data from aaaa
send data to dddd
*****
recieve data from aaaa
send data to dddd
*****
recieve data from aaaa
send data to dddd
*****

```

Figure 16: router1 receive the data

### 3.3.2 Router1 switched off

If the router1 is switched off, when sender send data to receiver, it need to use router2 and router3. So router 2 and router3 are all supposed to receive the data and finally the receiver receive data from router3. The topology is as shown in figure 17.

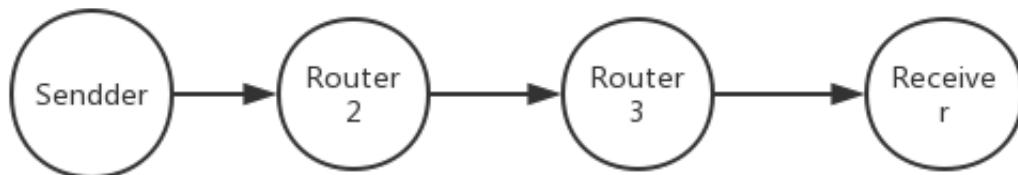


Figure 17: Topology of two routers in one path

```

Welcome to Contiki 2.4.
Running on: N740 NanoSensor (CC2431-F128).
*****
Sender Start, address is AA:AA
*****
send the broadcast, brdcstId is: 2
*****
send the data to CC:CC
temperatural is 30
*****

```

Figure 18: Sender sent the data

From the images, the sender sends out the data out. And router2 receive the data and transfer it to router3. Finally, router3 send data to receiver.



```

*****
recieve data from AA:AA
*****
Send data to FF:FF
*****

```

Figure 19: Router received the data

```

*****
recieve data from CC:CC
*****
Send data to DD:DD
*****

```

Figure 20: Router receive the data and sent it to another router

```

*****
recieve data from FF:FF
temperature1 is: 30
temperature2 is: 19
*****

```

Figure 21: Receiver received the data

## 4 Conclusion

Overall, it can be concluded that a good attempt was made to complete all the four functions that was stated in the introduction and that the implementation of the wireless sensor node is good. The strengths of this design is that, the wireless ad-hoc network is scalable as more nodes can be added to the network. Furthermore, the network is self-healing. This is because when nodes are taken out, the network initiates a route discovery to find an alternative route. This is shown in section 3.3.2. The weakness in our design is that, route requests are always broadcast regardless of whether a viable route is established. This is a problem because it wastes power. After a route is established, there need to keep broadcasting route requests no longer exists. Route requests should be sent out if a link is broken or node becomes inactive. Another weakness with our design is that, there is no way to check if a route is still active. This can be achieved using *hello* messages and is discussed in the section 4.1 below.

### 4.1 Next Steps

As part of the implementation of the wireless sensor network, it was desired to include *hello* messages as a means of ensuring that routes are still active. This could not be implemented due to time constraints. This will be included in the future design of the wireless ad-hoc network. This will be implemented as follows:

The transmitter (i.e. sender) will send a *hello* message via unicast to the router that it chose as its next hop to the destination to check whether that route is still active. The router upon receiving the *hello* message replies to the *hello* message that was transmitted by the sender via unicast. The sender then continues to send the data to that router as the route is confirmed to be active. The intermediate nodes (routers) also send *hello* messages to their next hops until this reaches the destination. If replies

are obtained by from each of the nodes, the route can then be confirmed as still being active. A time interval of two seconds is given to reply to the *hello* message. After this two second interval has elapsed, if no reply is obtained for the *hello* message, the node then the link is considered to be broken. As a result, the current node then initiates a route request to find an alternative path. This method avoids the need to keep sending route request broadcasts after a viable route has been established and limits the re-initiation of route discoveries to only when a link is considered to be inactive or broken.

## 5 Reference List

Famry, H.,A.,M. (2016). *Wireless Sensor Networks. Concepts, Applications, Experimentation and Analysis*. Singapore: Springer Science and Business Media

Harish, I., Sambasivan, S., I. (2013). “A protocol stack design and implementation of wireless sensor network for emerging application”. *IEEE International Conference on Emerging Trends in Computing, Communication and Nanotechnology*. Tirunelveli, India.

Sohrabi, K. (2007). “Introduction to Wireless Ad-Hoc Networks”. *Vehicular Technology Conference*

Yang, S.,H. (2014). *Wireless Sensor Networks Principles, Design and Applications*. London : Springer.

## 6 Appendix

### 6.1 Code for sender

```
#include "contiki.h"
#include "net/rime.h"
#include <string.h>
#include "dev/button-sensor.h"
#include "dev/sensinode-sensors.h"
#include "dev/leds.h"

#include <stdio.h>
#define TABLELENGTH 10

#define COMMANDROUTEREQUEST 0x20 //Command for requesting route
#define COMMANDROUTERESPONSE 0x21 //Command for route response
#define COMMANDDTATTX 0x22 //Command for sending data through unicast
#define COMMANDROUTEREQUESTS 0x23 //Command for requesting route
struct
{
    uint16_t destination; //The destination address
    uint16_t nextHop; //The next hop which point to the destination address
} fwdTable[TABLELENGTH];

static rimeaddr_t address;
static uint16_t destination = 0xDDDD;
static struct unicast_conn uc;
static struct broadcast_conn bc;
static uint8_t dataBuffer[50];
static uint8_t i = 0;
static uint16_t nextHop = 0;
static uint8_t found = 0;
static const struct broadcast_callbacks broadcast_callbacks = { recv_bc };
static const struct unicast_callbacks unicast_callbacks = { recv_uc };
static uint8_t Hello = 1;
static int rv;
static struct sensors_sensor * sensor;
static float sane = 0;
static uint16_t battery;
static int dec;
static float frac;
static uint8_t temperature1 = 0;
static uint8_t temperature2 = 0;
```

```

static struct etimer et;
static uint8_t brdcstCounter = 0;
static uint8_t brdcstID = 1;

/*-----*/
PROCESS(routenode_process, "Example unicast");
AUTOSTART_PROCESSES(&routenode_process);
/*-----*/

static void recv_uc(struct unicast_conn *c, const rimeaddr_t *from) {
    uint8_t * data;
    data = packetbuf_dataptr();

    //get the destination of packet
    destination = data[1];
    destination = destination << 8;
    destination = destination | data[2];

    //get the address of last node
    nextHop = from->u8[1];
    nextHop = nextHop << 8;
    nextHop = nextHop | from->u8[0];
    if (data[0] == COMMANDROUTERESPONSE) {
        for (i = 0; i < TABLELENGTH; i++) {
            if (fwdTable[i].destination == destination) {
                found = 1;
                break;
            }
        }

        if (found) {
            fwdTable[i].nextHop = nextHop;
            fwdTable[i].destination = destination;
        }
        else {
            for (i = 0; i < TABLELENGTH; i++) {
                if (fwdTable[i].destination == 0x0000) {
                    break;
                }
            }
            fwdTable[i].nextHop = nextHop;
            fwdTable[i].destination = destination;
        }
    }
}

```

```

static void recv_bc(struct broadcast_conn *c, rimeaddr_t *from) {

    packetbuf_clear();

}

PROCESS_THREAD(routenode_process, ev, data) {
    PROCESS_BEGIN();

    printf("*****\n\r");
    printf("Sender Start, address is AA:AA\n\r");
    printf("*****\n\r");
    for (i = 0; i < TABLELENGTH; i++) {
        fwdTable[i].destination = 0x0000;
        fwdTable[i].nextHop = 0xffff;
    }

    broadcast_open(&bc, 128, &broadcast_callbacks);
    unicast_open(&uc, 129, &unicast_callbacks);

    etimer_set(&et, CLOCK_SECOND * 5);

    while (1) {

        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et) || ev == sensors_event);

        sensor = (struct sensors_sensor *)data;
        if (sensor == &button_1_sensor) {
            //calculate the battery
            sensor = sensors_find(ADC_SENSOR);
            rv = sensor->value(ADC_SENSOR_TYPE_VDD);
            if (rv != -1) {
                sane = rv * 3.75 / 2047;
                battery = sane * 1000;
            }

            //prepare the broadcast data
            dataBuffer[0] = COMMAND_ROUTEREQUESTS;
            dataBuffer[1] = destination >> 8;
            dataBuffer[2] = destination;
            dataBuffer[3] = brdcstCounter;
            dataBuffer[4] = battery >> 8;
            dataBuffer[5] = battery;
            dataBuffer[6] = brdcstID;
            brdcstID = brdcstID + 1;

            //send the broadcast data
            packetbuf_copyfrom(dataBuffer, 7);

```

```

        broadcast_send(&bc);
        sensor = (struct sensors_sensor *)data;
        printf("*****\n");
        printf("send the broadcast, brdstId is: %d\n\r", dataBuffer[6]);
        printf("*****\n");
    } if (sensor == &button_2_sensor) {
        sensor = sensors_find(ADC_SENSOR);
        rv = sensor->value(ADC_SENSOR_TYPE_TEMP);
        if (rv != -1) {
            sane = ((rv * 0.61065 - 773) / 2.45);
            dec = sane;
            temperature1 = dec;
            frac = sane - dec;
            temperature2 = (unsigned int)(frac * 100);
        }

        dataBuffer[0] = COMMAND_DTATIX;
        dataBuffer[1] = destination >> 8;
        dataBuffer[2] = destination;
        dataBuffer[3] = temperature1;
        dataBuffer[4] = temperature2;

        for (i = 0; i < TABLELENGTH; i++) {
            if (fwdTable[i].destination == destination) {
                break;
            }
        }
        address.u8[0] = fwdTable[i].nextHop;
        address.u8[1] = fwdTable[i].nextHop >> 8;

        sensor = sensors_find(ADC_SENSOR);
        rv = sensor->value(ADC_SENSOR_TYPE_VDD);
        sane = rv * 3.75 / 2047;
        battery = sane * 1000;

        packetbuf_copyfrom(dataBuffer, 5);
        unicast_send(&uc, &address);
        printf("*****\n");
        printf("send the data to %02x\n\r temperature1 is %d\n\r", fwdT
        printf("The battery is %d\n\r", battery);
        printf("*****\n");
        packetbuf_clear();
        sensor = (struct sensors_sensor *)data;
    }

    etimer_reset(&et);
}

```

```

PROCESS_END();
}

```

## 6.2 Code for router

```

#include "contiki.h"
#include "net/rime.h"
#include <string.h>
#include "dev/button-sensor.h"
#include "dev/sensinode-sensors.h"
#include "dev/leds.h"

#include <stdio.h>
#define TABLELENGTH 10

#define COMMANDROUTEREQUEST 0x20 //Command for requesting route
#define COMMANDROUTERESPONSE 0x21 //Command for route response
#define COMMANDDTATTX 0x22 //Command for sending data through unicast
#define COMMANDROUTEREQUESTS 0x23 //Command for requesting route
#define BATTERY_AVERAGE_LVL 3000

struct
{
    uint16_t destination; //The destination address
    uint16_t nextHop; //The next hop which point to the destination address
} fwdTable[TABLELENGTH];

struct
{
    uint16_t destination; //The destination address
    uint16_t nextHop; //The next hop which point to the destination address
    uint16_t battery;
    uint16_t rssi;
    uint8_t hopCounts;
} rvsTable[TABLELENGTH];

static rimeaddr_t address;
static uint16_t destination;
static struct unicast_conn uc;
static struct broadcast_conn bc;
static uint8_t dataBuffer[50];
static uint16_t nextHop = 0;
static uint16_t rssi = 0;

```



```

static uint8_t brdcstIDLH = 0;
static uint8_t i = 0;
static uint8_t found = 0;

static const struct broadcast_callbacks broadcast_callbacks = { recv_bc };
static const struct unicast_callbacks unicast_callbacks = { recv_uc };

static int rv;
static struct sensors_sensor * sensor;
static float sane = 0;

static uint16_t battery = 0;

/*static uint8_t temperature1 = 0;
static uint8_t temperature2 = 0;*/

static uint8_t brdcstCounter = 0;
static uint8_t brdcstLimit = 4;
static uint8_t brdcstID = 0;

/*-----*/
PROCESS(routenode_process, "Example unicast");
AUTOSTART_PROCESSES(&routenode_process);
/*-----*/

static void recv_bc(struct broadcast_conn *c, rimeaddr_t *from) {
    uint8_t * data;
    data = packetbuf_dataptr();

    found = 0;

    //get the address of last hop
    nextHop = from->u8[1];
    nextHop = nextHop << 8;
    nextHop = nextHop | from->u8[0];

    //get the destination of the packet
    destination = data[1];
    destination = destination << 8;
    destination = destination | data[2];

    //get hop count of the packet
    brdcstCounter = data[3];

```

```

//get the battery of the last hop
battery = data[4];
battery = battery << 8;
battery = battery | data[5];

//get rssi of last hop
rssi = packetbuf_attr(PACKETBUF_ATTR_RSSI);

//get broadcast id of last hop
brdcstIDLH = data[6];

if (data[0] == COMMANDROUTREQUEST || data[0] == COMMANDROUTREQUESTS) {
    for (i = 0; i < TABLELENGTH; i++) {
        if (rvsTable[i].destination == destination) {
            found = 1;
            break;
        }
    }
    printf("*****\n\r");
    printf("recieve broadcast from %02x, \n\r brdcstID is %d\n\r", nextHop,
    printf("*****\n\r");
    //the destination is in the routing table
    if (found) {
        if (rvsTable[i].hopCounts < brdcstCounter) {
            //if no shorter path found, do nothing
        }
        else if (rvsTable[i].hopCounts == brdcstCounter) {
            //if length of path is the same, compare battey and rssi
            if (rvsTable[i].battery < battery) {
                rvsTable[i].nextHop = nextHop;
                rvsTable[i].battery = battery;
                rvsTable[i].rssi = rssi;
                rvsTable[i].hopCounts = brdcstCounter + 1;
            }
            else {
                if (rvsTable[i].rssi < rssi) {
                    rvsTable[i].nextHop = nextHop;
                    rvsTable[i].battery = battery;
                    rvsTable[i].rssi = rssi;
                    rvsTable[i].hopCounts = brdcstCounter + 1;
                }
            }
        }
        else {
            rvsTable[i].nextHop = nextHop;
            rvsTable[i].battery = battery;
            rvsTable[i].rssi = rssi;

```

```

        rvsTable[i].hopCounts = brdcstCounter + 1;
    }
}
else {
    for (i = 0; i < TABLELENGTH; i++) {
        if (rvsTable[i].destination == 0x000) {
            rvsTable[i].destination = destination;
            rvsTable[i].nextHop = nextHop;
            rvsTable[i].rssi = rssi;
            rvsTable[i].battery = battery;
            rvsTable[i].hopCounts = brdcstCounter + 1;
        }
    }
}

//if it is not the same broadcast, rebroadcast
if (brdcstID != brdcstIDLH) {
    dataBuffer[0] = COMMANDROUTEREQUEST;
    dataBuffer[1] = destination >> 8;
    dataBuffer[2] = destination;
    dataBuffer[3] = brdcstCounter + 1;

    //calculate the battery
    sensor = sensors_find(ADC_SENSOR);
    rv = sensor->value(ADC_SENSOR_TYPE_VDD);
    sane = rv * 3.75 / 2047;
    battery = sane * 1000;

    dataBuffer[4] = battery >> 8;
    dataBuffer[5] = battery;
    dataBuffer[6] = brdcstIDLH;

    packetbuf_copyfrom(dataBuffer, 7);
    broadcast_send(&bc);
    brdcstID = brdcstIDLH;
    printf("*****\n");
    printf("send broadcast, brdcstID is %d\n\r", brdcstIDLH);
    printf("*****\n");
}

}

packetbuf_clear();
}

static void recv_uc(struct unicast_conn *c, const rimeaddr_t *from) {
    uint8_t *data;
    data = packetbuf_dataptr();

```

```

found = 0;
//get the address of next node
nextHop = from->u8[1];
nextHop = nextHop << 8;
nextHop = nextHop | from->u8[0];

//get the destination of packet
destination = data[1];
destination = destination << 8;
destination = destination | data[2];

if (data[0] == COMMAND.ROUTERRESPONSE) {
    printf("*****\n\r");
    printf("recv router response from %02x\n\r", nextHop);
    for (i = 0; i < TABLELENGTH; i++) {
        if (fwdTable[i].destination == destination) {
            found = 1;
            break;
        }
    }

    if (found) {
        fwdTable[i].nextHop = nextHop;

        for (i = 0; i < TABLELENGTH; i++) {
            if (rvsTable[i].destination == destination) {
                address.u8[0] = rvsTable[i].nextHop;
                address.u8[1] = rvsTable[i].nextHop >> 8;
                break;
            }
        }
        packetbuf_copyfrom(data, 3);
        unicast_send(&uc, &address);
        printf("send router response to %02x\n\r", rvsTable[i].nextHop);
        printf("*****");
    } else {
        for (i = 0; i < TABLELENGTH; i++) {
            if (fwdTable[i].destination == 0x000) {
                fwdTable[i].destination = destination;
                fwdTable[i].nextHop = nextHop;
                break;
            }
        }
        for (i = 0; i < TABLELENGTH; i++) {
            if (rvsTable[i].destination == destination) {
                address.u8[0] = rvsTable[i].nextHop;

```

```

        address.u8[1] = rvsTable[i].nextHop >> 8;
        break;
    }
}
printf("send router repsonse to a new router %02x", rvsTable[i].
printf("*****\n\r");
packetbuf_copyfrom(data, 3);
unicast_send(&uc, &address);
}

} else if (data[0] == COMMANDDTATTX) {
    printf("*****\n\r");
    printf("recieve data from %02x\n\r", nextHop);

    for (i = 0; i < TABLELENGTH; i++) {
        if (fwdTable[i].destination == destination) {
            packetbuf_copyfrom(data, 5);
            address.u8[0] = fwdTable[i].nextHop;
            address.u8[1] = fwdTable[i].nextHop >> 8;
            unicast_send(&uc, &address);

            printf("send data to %02x\n\r", fwdTable[i].nextHop);
            printf("*****\n\r");
            break;
        }
    }
}
packetbuf_clear();
}

/*-----*/
PROCESS_THREAD(routenode_process, ev, data) {
    static struct etimer et;

    PROCESS_EXITHANDLER(unicast_close(&uc);)
    PROCESS_BEGIN();
    printf("*****\n\r");
    printf("Router Start, address is CC:CC\n\r");
    printf("*****\n\r");

    for (i = 0; i < TABLELENGTH; i++)
    {
        rvsTable[i].destination = 0x0000;
        rvsTable[i].nextHop = 0xffff;
        rvsTable[i].battery = 0;
        rvsTable[i].rssi = 0;
    }
}

```

```

    for (i = 0; i<TABLELENGTH; i++)
    {
        fwdTable[i].destination = 0x0000;
        fwdTable[i].nextHop = 0xffff;
    }

    broadcast_open(&bc, 128, &broadcast_callbacks);
    unicast_open(&uc, 129, &unicast_callbacks);

    etimer_set(&et, CLOCK_SECOND * 2);

    while (1)
    {
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

        etimer_reset(&et);
    }

    PROCESS_END();
}

```

### 6.3 Code for receiver

```

#include "contiki.h"
#include "net/rime.h"
#include <string.h>
#include "dev/button-sensor.h"
#include "dev/sensinode-sensors.h"
#include "dev/leds.h"

#include <stdio.h>
#define TABLELENGTH      10

#define COMMANDROUTEREQUEST  0x20    //Command for requesting route
#define COMMANDROUTERESPONSE 0x21    //Command for route response
#define COMMANDDTATIX        0x22    //Command for sending data through unicast
#define COMMANDROUTEREQUESTS 0x23    //Command for requesting route

#define BATTERY_AVERGELVL    3000

struct
{
    uint16_t destination;                //The destation address

```

```

        uint16_t nextHop;                //The next hop which point to the destination a
        uint16_t battery;
        uint16_t rssi;
        uint8_t hopCounts;

} rvsTable[TABLELENGTH];

static rimeaddr_t address;
static uint16_t destination;
static struct unicast_conn uc;
static struct broadcast_conn bc;
static uint8_t dataBuffer[50];
static uint8_t found = 0;

static const struct broadcast_callbacks broadcast_callbacks = { recv_bc };
static const struct unicast_callbacks unicast_callbacks = { recv_uc };

static int rv;
static uint16_t nextHop = 0;
static uint16_t rssi = 0;
static float sane = 0;

static uint16_t battery = 0;

/*static uint8_t temperature1 = 0;
static uint8_t temperature2 = 0;*/

static uint8_t brdcstCounter = 0;
static uint8_t brdcstLimit = 4;
static uint8_t brdcstID = 0;
static uint8_t brdcstIDLH = 0;
static uint8_t i = 0;
/*-----*/
PROCESS(routenode_process, "Example unicast");
AUTOSTART_PROCESSES(&routenode_process);
/*-----*/

static void recv_bc(struct broadcast_conn *c, rimeaddr_t *from) {
    uint8_t * data;
    data = packetbuf_dataptr();

    found = 0;

```

```

//get the address of last hop
nextHop = from->u8[1];
nextHop = nextHop << 8;
nextHop = nextHop | from->u8[0];

//get the destination of the packet
destination = data[1];
destination = destination << 8;
destination = destination | data[2];

//get hop count of the packet
brdcstCounter = data[3];

//get the battery of the last hop
battery = data[4];
battery = battery << 8;
battery = battery | data[5];

//get rssi of last hop
rssi = packetbuf_attr(PACKETBUF_ATTR_RSSI);

//get broadcast id of last hop
brdcstIDLH = data[6];

if (data[0] == COMMANDROUTEREQUEST) {
    printf("*****\n\r");
    printf("recieve broadcast from %02x\n\r", brdcstID);
    printf("the battery is %d, the rssi is %d\n\r", battery, rssi);
    printf("*****\n\r");
    for (i = 0; i < TABLELENGTH; i++) {
        if (rvsTable[i].destination == destination) {
            found = 1;
            break;
        }
    }

    //the destination is in the routing table
    if (found) {
        if (rvsTable[i].hopCounts < brdcstCounter) {
            //if no shorter path found, do nothing
        }
        else if (rvsTable[i].hopCounts == brdcstCounter) {
            //if length of path is the same, compare battey and rssi
            if (rvsTable[i].battery < battery) {
                rvsTable[i].nextHop = nextHop;
                rvsTable[i].battery = battery;
                rvsTable[i].rssi = rssi;
            }
        }
    }
}

```



```

        rvsTable[i].hopCounts = brdcstCounter + 1;
    }
    else {
        if (rvsTable[i].rssi < rssi) {
            rvsTable[i].nextHop = nextHop;
            rvsTable[i].battery = battery;
            rvsTable[i].rssi = rssi;
            rvsTable[i].hopCounts = brdcstCounter + 1;
        }
    }
}
else {
    rvsTable[i].nextHop = nextHop;
    rvsTable[i].battery = battery;
    rvsTable[i].rssi = rssi;
    rvsTable[i].hopCounts = brdcstCounter + 1;
}
}
else {
    for (i = 0; i < TABLELENGTH; i++) {
        if (rvsTable[i].destination == 0x000) {
            //create new table
            rvsTable[i].destination = destination;
            rvsTable[i].nextHop = nextHop;
            rvsTable[i].rssi = rssi;
            rvsTable[i].battery = battery;
            rvsTable[i].hopCounts = brdcstCounter + 1;
            break;
        }
    }
}

if (brdcstID != brdcstIDLH) {
    dataBuffer[0] = COMMANDROUTERRESPONSE;
    dataBuffer[1] = destination >> 8;
    dataBuffer[2] = destination;

    address.u8[0] = nextHop;
    address.u8[1] = nextHop >> 8;

    packetbuf_copyfrom(dataBuffer, 3);
    unicast_send(&uc, &address);

    printf("*****\n");
    printf("send router response to %02x\n\r", nextHop);
    printf("*****\n");
    brdcstID = brdcstIDLH;
}

```

```

    }
}
packetbuf_clear();
}

static void recv_uc(struct unicast_conn *c, const rimeaddr_t *from) {
    uint8_t * data;
    data = packetbuf_dataptr();

    //get the address of next node
    nextHop = from->u8[1];
    nextHop = nextHop << 8;
    nextHop = nextHop | from->u8[0];

    //get the destination of packet
    destination = data[1];
    destination = destination << 8;
    destination = destination | data[2];
    printf("*****\n\r");
    printf("recv the data from: %02x\n\r", nextHop);
    if (data[0] == COMMANDDTATX) {

        printf("temperature1 is: %d\n\r", data[3]);
        printf("temperature2 is: %d\n\r", data[4]);
        printf("*****\n\r");
    }
    packetbuf_clear();
}

/*-----*/
PROCESS_THREAD(routenode_process, ev, data)
{
    static struct etimer et;

    PROCESS_EXITHANDLER(unicast_close(&uc);)
    PROCESS_BEGIN();

    printf("*****\n\r");
    printf("Receiver Start, address is DD:DD\n\r");
    printf("*****\n\r");

    for (i = 0; i < TABLELENGTH; i++)
    {
        rvsTable[i].destination = 0x0000;
        rvsTable[i].nextHop = 0xffff;
        rvsTable[i].battery = 0;
        rvsTable[i].rssi = 0;
    }
}

```

```

    }

    broadcast_open(&bc, 128, &broadcast_callbacks);
    unicast_open(&uc, 129, &unicast_callbacks);

    etimer_set(&et, CLOCK_SECOND * 2);

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

    PROCESS_END();
}

```