CMPUT 313 - Lab 3

- Answers to Part 1 Questions

- Report for Part 2 CNET code

Submitted by: Neel Kumar
CCID: neel1

Part 1: Exploring the Supplied Geographical Routing Algorithm

1.

The first instance where we see georouting randomize transmissions is shown in the following line, where a defined variable called TX_NEXT is given a random value between 5000000 and 10000000:

```
#define TX_NEXT          (5000000 + CNET_rand()%5000000)
```

TX_NEXT is used in a CNET timer event, that calls EV_TIMER1:

```
CNET_start_timer(EV_TIMER1, TX_NEXT, 0);
```

And an EV_TIMER1 event is handled by the "transmit" function:

```
CNET_set_handler(EV_TIMER1, transmit, 0);
```

Thus,

```
static EVENT_HANDLER(transmit)
```

is called to transmit frames randomly every 5 to 10 seconds.

2.

When a mobile node receives a packet destined to another mobile, it does not 'randomize' packet retransmission. Retransmission occurs if the current node is closer to the intended destination than the previous node.
This is indicated in the 'else' statement, between lines 93-106 in georouting.c

---

3.

```
// POPULATE A NEW FRAME
do {
      frame.header.dest    = CNET_rand() % NNODES;
} while (frame.header.dest == nodeinfo.nodenumber);
```

Like the comment indicates, the above line is one of the steps in populating a new frame for transmission. The loop chooses a random node within the range of number-of-nodes, and if by chance happens to choose itself, it will keep trying until a node (other than itself) is chosen.

4.

The payload of each frame is a char[2304] array. In this char-array is contained a message saying "hello from _____", where the blank is filled with the sending node's number.

As discussed in Q1, the mobile generates a frame between 5 to 10 seconds, as shown in:

```
#define TX_NEXT          (5000000 + CNET_rand()%5000000)
```

5.

a. CNET_shmem2(const char *name, size_t length) creates a pointer to a block of memory with the 'length' number of bytes. This block of memory is accessible by all nodes.
In the georouting program, it is used to maintain global statistics.

```
positions    = CNET_shmem2("p", NNODES*sizeof(CnetPosition));
```

The above line in specific, effectively creates an array the size of the number of nodes. The value at each index is the position of the node.
For example, positions[1], would indicate the position of node 1 (in the form of a CnetPosition type).

b. A mobile node will call this function twice, it is declared once for use in georouting.c, and another time in mobility.c
Both these C files need access to these shared memory variables to function properly.

c.

Two shared memory regions are being allocated by each mobile node, shown in the following:

```
stats    = CNET_shmem2("s", 2*sizeof(int));
positions    = CNET_shmem2("p", NNODES*sizeof(CnetPosition));
```

To clarify: <u>all</u> nodes have access to these two shared memory variables, (stats and position).

Since <u>each node</u> is allocating memory for these two shared memory variables, then the total number of shared memory regions in the entire simulation is NNODES*2 regions.

6.

We have a 400m x 400m square map. Meaning the maximum possible distance is from one corner of the map to the other. When we take into account the margins from the random_point() function, our map area is 380m in each direction
The maximum possible distance is from one corner of the map to the other:

$$\sqrt{380^2 + 380^2} = 537.4011537$$

We have a WALKING_SPEED of 3.0

In EVENT_HANDLER(mobility), assuming we are starting a new walk, we'd be inside the 'if' statement of line 45.

The following variables will be set:

```
double dx    = 380
double dy    = 380
double metres    = 537.4011537
double nsteps    = 179.13337179
walk->dx     = 2.121320344
walk->dy     = 2.121320344
```

We will keep traversing towards our destination in increments of walk->dx and walk->dy.

We will deem our destination as reached when the following condition is met:

```
if((dx*dx + dy*dy) <= CLOSE_ENOUGH);
```

CLOSE_ENOUGH is set to 100. Since our dx and dy are the same in this case, we can reduce the equation to:

2(dx*dx) <= CLOSE_ENOUGH

which results in a dx or dy of 7.071067812.

When we are within 7.071067812m of our destination (in either axis), we can consider it reached.

This means that we have to travel a distance of at least 380 - 7.071067812 = 377.8786797m in both the x and y axis to reach our dest.

We are walking in increments of walk->dx = 2.121320344, so it will take a total of $\lceil 377.8786797/2.121320344 \rceil$ steps to reach our destination, which is 178.1337179=179 steps.

Since we pause USEC_PER_STEP = 1000000 (1 second) between every step, it will take a total of **179 seconds** to reach our destination. This is the maximum amount of time we can be travelling until we trigger another pause.

7.

```
new->x = CNET_nextrand(walk->mt) % (long)(walk->maparea.x-2*MARGIN) + MARGIN;
new->y = CNET_nextrand(walk->mt) % (long)(walk->maparea.y-2*MARGIN) + MARGIN;
```

These lines choose a random x and y value that correspond to a node's new destination coordinate. They take into account a defined MARGIN size.

A x and y value are chosen that is within the map area size, and within the desired margin.

Report for Part 2 - Routing in MANETs with Anchor Nodes

Neel Kumar, CCID: neel1

Objectives

The objective of this project was to modify an existing georouting MANET algorithm in CNET to remove some impractical assumptions it made, and also add additional functionality that improved the old algorithm.

Design Overview – Important Ideas and Features

The following modifications were made to the original georouting algorithm:

- Remove the assumption that nodes have access to the locations of other nodes
- Implement anchor nodes that notify mobiles of the anchor location
- Implement anchor nodes that relay mobile transmissions
- Use of node addresses (rather than node numbers) for transmissions
- Clearly output each node's and anchor's trace of transmitted and received packets

To achieved the above goals, the following additions were made to the original code (loosely in this order):

1. The topology file added hosts to be used as anchors (in my case, I used two anchors). Furthermore, addresses were explicitly assigned for mobiles and anchors, rather than using the default CNET values. Anchor nodes and mobile nodes are assigned addresses in the range [1-99] and [100-199] respectively. The mobiles and anchors both use WLAN.
2. The topology file has two strings, containing all the mobile and anchor addresses.
3. The lab3.c code parsed these strings in reboot_node, and added them to global integer arrays for further use throughout the program.
4. Wherever node number was previously used, node address was used instead. This was mainly in the source and destination fields of the frame.
5. The anchor nodes were programmed to frequently broadcast beacons, to let mobile nodes know their location.
6. Mobiles transmissions have (mostly) the same behaviour, with some small changes. However, mobiles have a significantly different receive behaviour:
    a. If a mobile receives a transmission from an anchor, it checks if it has that anchor's address stored in its 'history'. If not, it adds it to a list of known anchors. Furthermore, the mobile will request the anchor for data that belongs to its own address.
    b. If a mobile receives a transmission from another mobile, it checks to see if it was the intended recipient. If so, it prints that a packet was received. If not, (if the mobile received a transmission indented for another address), the mobile will check if there is a nearby anchor from its list of known anchors. If there is an anchor nearby, it will forward the message to the anchor.
7. Anchor's have the following receive behaviour:
    a. If the anchor receives a transmission from a mobile for forwarding to another mobile, it will store that frame in a buffer (in this case, the buffer is an array containing FRAME structs). The anchor checks if that particular frame is already in its buffer before adding it (as two mobiles could potentially forward the same frame to an anchor for forwarding).
    b. If the frame receives a request from a mobile for data, the frame will search its buffer for any data belonging to that mobile. If it finds any, it will transmit the frame, and remove it from the buffer.

Many other details had to be implemented, please see the code for further details, as it is commented throughout. However, the following are some major code additions:

- The frame has a field "retransmitted" to indicate a retransmitted frame
- The frame has a field "anchor_request", meant only to indicate that a mobile is requesting data from an anchor.
- Arrays such as mobile_addresses, anchor_addresses, anchor_buffer, anchor_locations were added to keep track of data and make this whole thing possible.
- An additional timer was added, that only allows mobiles to ask anchors for data once every five to ten seconds.
- When a frame is added to an anchor's buffer, the buffer capacity is printed, with zeros indicating an empty slot, and ones indicating an occupied slot.

Project Status and Challenges Faced

The simulation runs well. All the objectives were met, and the code does not crash. The following challenges were faced during development:

- A mobile should ask an anchor for data when it is close and can hear an anchor's beacons. When this functionality was initially added, when being in close proximity to an anchor the mobile would go crazy and continually ask the anchor for data, flooding it with requests until it was out of bounds. To combat this, a 'cooldown' period of five to ten seconds is put in place before a mobile can ask one again ask an anchor for data.
- Since everything is done over link 1, there are a lot of requests floating around in the air. A mobile will only print 'pkt received' if it receives a packet intended for itself. Without this, stdout would be flooded with irrelevant receive messages.
- As discussed in the section above, not all messages are merely transmissions. Now, we have explicit retransmissions, requests, replies etc. Additional fields were added to FRAME to help with this.
- CNET says the WLAN range is ~173m. For testing purposes, an anchor was deemed "within range" when it is 100m or less from a mobile.

Testing and Results

To test the program, a lot of simulations were run. Furthermore, two anchors were used to confirm they are behaving correctly. The results (the messages printed to stdout) were closely examined to ensure the code was running properly.

Examples:

- If a mobile retransmitted a message or requested data from an anchor, I would check the map to see if the mobile was close to an anchor.
- If an anchor relayed a message to a mobile, I would check if it was within range on the map.
- Checking stdout to ensure mobiles are sending frames within the appropriate time frames.
- Check stdout to ensure that mobiles aren't flooding anchors with requests.
- Check the anchor's buffers to ensure they are properly loading and removing frames
- Much more…