ELEC3227 Embedded Networked Systems Coursework 2019-2020

Rohan Bungre

29466423

rsb1g17@soton.ac.uk

06/01/20

Team B2

Designing and implementing a full, working, network architecture capable of supporting a smart lighting application.

Contents

1 Introduction

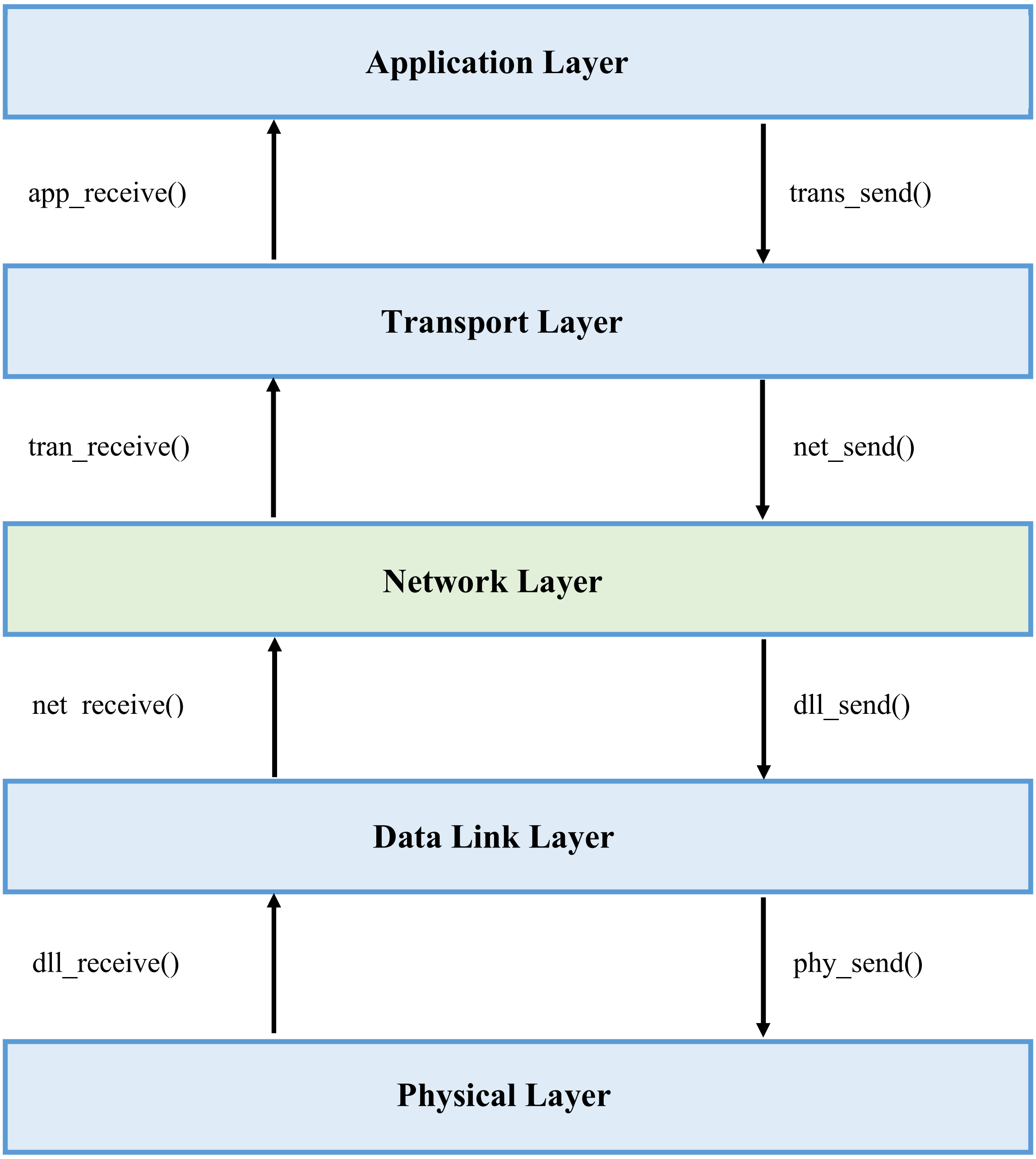

Figure 1 - Design to implement a protocol stack

Figure 1 shows the protocol stack that Team B2 would develop. Each layer would call primitives provided by other neighbouring layers allowing for data abstraction. This report will cover the design and implementation of the Network Layer in a working full stack as demonstrated in the labs.

2 Standards Document

2.1 Primitives

Send(transport_data, destination, length) – A function to send the network packet between nodes via the optimal route. Requires the transport segment, the address of the destination node and the length of the transport segment as arguments.

Receive(network_data, length) – A function to receive the network packet and decide what happens to it. Requires the network packet and its length as arguments.

Flood(transport_data, length) – A function to flood the network packet to every node in the system. Requires the transport segment and the length of the segment as arguments.

2.2 Network Packet Structure

Control [1]	Control [1]	SRC Address [1]	DEST Address [1]	Length [1]	Data [8]	Checksum [1]	Checksum [1]
0,1,3	0	0-2	0-2	8	Transport Segment	(0-4)	(5-12)

Table 1: The structure that all network packets follow and the values that the field can contain. Square brackets indicate the length of a field in bytes.

The valid values of the "Control" field correspond to a "normal" (0) packet, a "flood" (1) packet, and a "weight" (3) packet respectively.

In the case of the checksum fields, the "values" are the bytes of the packet that are used to form the checksum.

Control [1]	Control [1]	SRC Address [1]	DEST Address [1]	Length [1]	Data [8]	Checksum [1]	Checksum [1]
3	0	0-2	0-2	8	00 01 02 10 11 12 20 21	(0-4)	(5-12)

Table 2: The structure of a network weights packet.

To be a network weights packet, the control field must contain the value 3 - 1 from the flag to indicate that the packet is a flood, and 2 to indicate that the packet contains weights.

The other fields are the same as a normal packet with the length of the data is always being 8 bytes and the data that is transmitted is the network weights matrix where 01 is the weight from node 0 to node 1.

3 Design

As each layer could use multiple varying protocols to implements its features, the layers must be abstracted from each other and assume no knowledge of the packet structure it is receiving. This means that it was important to consider a design where the primitives required important data such as the length of data segments and next nodes to be explicitly given as input arguments, by the neighbouring layers.

3.1 Packet Structure

To abide by the packet structure shown in table 1 and 2, typedef structs were used. As the project required 8-bit bytes, each element of the packet was assigned a uint8_t variable. These structs were very useful as they were global, so each function could access them. Three structs were created, the first being to hold each individual byte of the network packet: uint8_t control[2], uint8_t src_address, uint8_t dest_address, uint8_t length, uint8_t tran_segment[8], uint8_t checksum[2]. The second struct was created to hold the full network packet in a uint8_t net_packet[15] array. This was created by the create_net_packet() function within the code. The third struct was a network weights struct that held the network traffic weights between each node. This was used when calculating the optimal route.

3.2 Setup

A net_setup() function was created to allow for initial values to be assigned when the system started up, or was reset. It assigns the number of nodes in the system and sets the initial weights for the routing algorithm.

3.3 Routing Algorithm

To route data across a network of nodes there are many algorithms to calculate the shortest path based on distance and traffic on the network. As each node in the network is connected to the other, Dijkstra's Shortest Path is the best option as no forwarding needs to be done. It has a time complexity of n^2 so isn't any less efficient than distance-vector routing for example. It is based on greedy technique.

The steps of the implemented Dijkstra's Algorithm [1]:

1. Create a 2D adjacency matrix that represents the nodes on the network and the weights on each vertex between each node.
2. Create a cost matrix by setting any zero elements to a very large "infinity" value.
3. Create a visited array, initialised at 0 i.e. visited[i]=0.
4. Calculate a distance and a predecessor array from the cost matrix, indexed at the starting node i.e. distance[i]=cost[startnode][i], pred[i]=startnode.
5. Check the condition distance[i]<mindistance&&!visited[i], to find the node at a smallest distance away that isn't itself and then check the condition mindistance+cost[nextnode][i]<distance[i], to see if a shorter path exists through other nodes.
6. To plot the next node in the shortest path from a startnode to the endnode, the condition pred[currnode] != startnode must be met and currnode = pred[currnode] will store the next node to take.

This algorithm was implemented by the calculate_next_node() function. It requires an input argument of the number of nodes in the network, the start node and the end node. It returns the next node to send the data via the shortest path.

3.4 Update Weights

To keep the weights across all nodes equal and updated, so that the shortest paths were chosen a net_update_weights() function was created. Using the packet structure from table 2, the data segment is replaced by the weights used by the adjacency matrix in Dijkstra's. This is flooded to all nodes, and an averaged value for each vertex weight is calculated. After a set time period the weights are all reset back to the default value of 1. This is done using a timer and the reset_weights() function. The control bit was also set as 3, to distinguish it as a weight packet.

3.5 Send

Following the standards document, the net_send() function would take in the transport layer segment, the destination address and its length. The initial control bit is set to 0 and then the network layers packet is filled in with the correct data. The source address is found from the address of the sending node, the destination address is from the input argument. The calculate_next_node() and create_net_packet() is called and then the function calls the dll_send() function. The checksum is calculated and stored in the checksum bits. The network packet data and its length along with the next node is passed in as input arguments for the dll layer, to allow for layer abstraction.

3.6 Checksum

Using an even parity checksum, the network packet is split into the first 5 bytes, then the last 8 bytes. Each of these are converted into binary and the even parity is stored as its decimal representation into checksum[0] and checksum[1] respectively. This is used when creating and checking the checksum to see if the data had been corrupted.

3.7 Resend

If the data hasn't reached the final node, the next node is calculated by the Dijkstra's function and the data is sent using the same method as explained by 3.5.

3.8 Receive

If the data has reached the final node, the first thing done it to compare the checksum to the generated values. If the checksum the control bytes are correct or wrong the process continues, however a message is printed out indicating success or failure. If the packet has reached the final node and control[0] is 0, the trans_receive() function is called. The function takes the transport segment and the length of the data. If the control[0] is 3 the weight struct is updated.

3.9 Flood

The flood function acts very simply, it calls the dll_flood() function, passing it just the network packet data and its length.

4 Testing, Results and Analysis

4.1 Network Routing

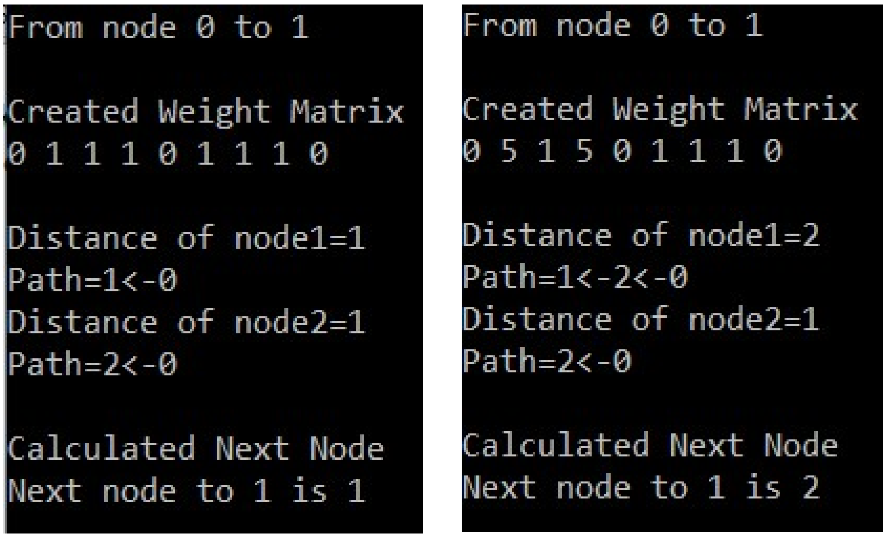

Figure 2 - Routing Algorithm

Figure 2 shows the network routing algorithm in action. When the weighting is equal between all nodes, the data is routed via the most direct path. When the weighting is increased the shortest route is via the other node. In this test case the starting address was 0 and the destination address was 1. In the first case the next node returned was just 1. In the more complex case where weights are not equal the next node returned was 2 as it was the shorter route.

4.2 Primitives

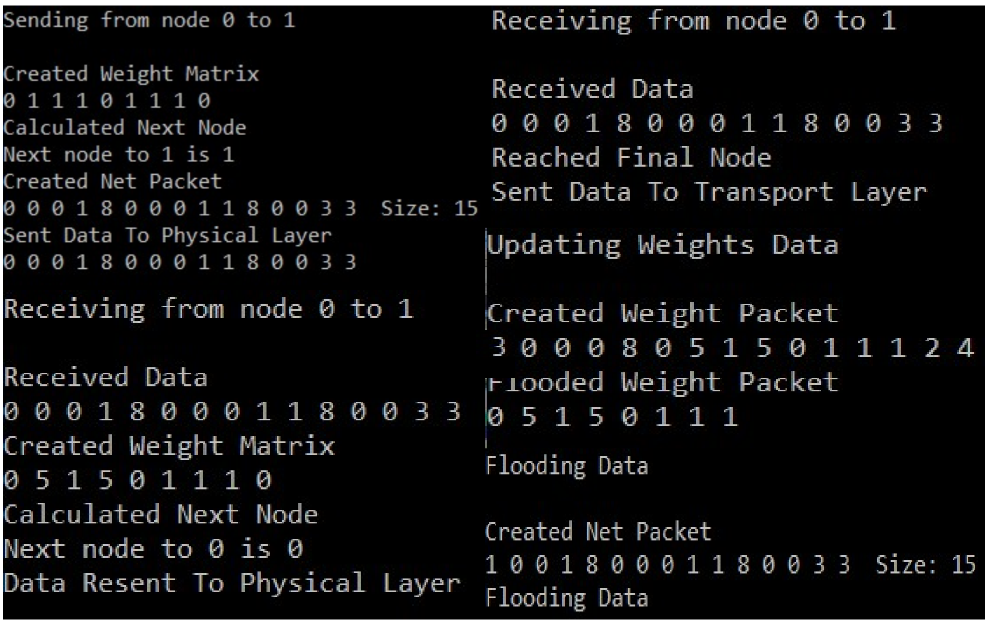

Figure 3 - Primitives in action

Figure 3 shows the primitive working as expected. When sending data, the network packet is successfully created showing that the packet structure worked as intended. This was then passed onto the physical layer to send via the radios. When receiving data, if the final node has been reached the transport layer receive function was correctly called. If the final node had not been reached the next node would be calculated and resent to the physical layer. The updating of weights worked as intended with the first control bit being set as 3 and the correct weights being stored in the data bytes. Flooding also worked by calling the dll_flood() function.

4.3 Checksum

```
Checksum Test

Sent Data: 0 0 0 1 8 0 0 0 1 1 8 0 0 3 3

Received Data
0 0 0 1 8 0 0 0 1 1 8 0 0 3 3
Checksum 1 Passed
Checksum 2 Passed
Reached Final Node
Sent Data To Transport Layer
```

```
Checksum Test

Sent Data: 0 0 0 1 8 0 0 0 1 1 8 0 0 5 7

Received Data
0 0 0 1 8 0 0 0 1 1 8 0 0 3 3
Checksum 1 Failed
Checksum 2 Failed
Reached Final Node
Sent Data To Transport Layer
```

Figure 4 - Demonstrating the Checksum

Figure 4 shows the checksum operating in a pass and a fail situation. When the send and received checksum, both match the checksum passes. If the checksums do not match, then then it fails.

4.4 Full Team Demonstration

```
Resend Frame 0
Checksum: 168
DATA STORED IN BUFFER: 0
102211235168

Transmit enable is false
TRANSMITTED DATA:
102211235168

Created Weight Matrix
0 1 1 1 0 1 1 1 0
Calculated Next Node
Next node to 1 is 1
Increased Weights
Created Net Packet
0 0 0 1 8 1 0 2 2 1 1 235 168 3 6

Send Buffers:
Frame: 0, Checksum: 4D8A
AACC0000010F0000000108010002020101EBA803060000000000000004D8AAA
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000

Resend Frame 0
```

Figure 5 - Full Stack Communication

In previous sections the tests were performed on a PC and on a singular Il-Matto, simulating communication between layers. *Figure 5* shows the network layer interacting with the other layers on 3 Il-Mattos.

The layer was able to route data between nodes, calculate weights by monitoring network traffic and was able to work with my teammate's layers, producing a working stack as shown in the demonstration.

5 Critical Reflection and Evaluation

Over the project I was able to produce a working network layer that was able to route data over a network of nodes. Despite producing a working layer, there are some things that could be improved and developed further.

If I was to the project differently, I would start by agreeing a much more complex brief at an earlier stage. The original brief was not sufficient enough for me and my peer teammate to produce the exact same standard. It took some last-minute changes to both of our work to agree with the standard. I would also start developing code on the Il-Matto much earlier as most of the code written on my PC didn't work at first on the Il-Matto due to memory and other issues. I would have saved a lot of time developing code rather than problem solving. I would also have tested out all the outputs of my Il-Matto before starting the project as I found that some pins were broken, so I assumed my code was wrong. It was only when I tried my code on a different Il-Matto that it worked.

Although my design worked, it isn't very scalable. It meets the standards developed by the team and peer team; however, it was decided that we were going to use a fixed application data size. This meant I produced a network layer that could only work for one transport segment data length of 8. Ideally this would be a variable size to allow for data abstraction. The network layer also only worked with 3 nodes, when it should be able to work with any amount.

When working as a team there were a few challenges. The main challenge was waiting for teammates code to be working before I could fully test my own. As my network layer had to communicate with two peoples work, that made it difficult. Working with my peer teammate was easy initially as we both agreed on an idea before developing it. The only issue came at the end when the other team changed their mind about variable data size so wanted to include it on the standard. As it wasn't initially agreed, I was able to convince them not to add it on.

References

[1]https://www.thecrazyprogrammer.com/2014/03/dijkstra-algorithm-for-finding-shortest-path-of-a-graph.html

Appendix

```
#include "net.h"

uint64_t time;

int num_nodes;
int num_vert;
int next_node;
int sent_count;

typedef struct
{
   uint8_t control[2];
   uint8_t src_address;
   uint8_t dest_address;
   uint8_t length;
   uint8_t tran_segment[8];
   uint8_t checksum[2];

} network;
network packet;

typedef struct
{
   uint8_t net_packet[15];

} comb;
comb net;

typedef struct
{
   uint8_t w_0_1;
   uint8_t w_0_2;
   uint8_t w_1_2;

   uint8_t G[9];
} weight;
weight w;



void net_setup()
{//fuction to set up initial values needed in the code
   num_nodes = 3;
   sent_count = 0;
   w.w_0_1 = 1;
   w.w_0_2 = 1;
   w.w_1_2 = 1;
```

```
    w.G[0] = 0;
    w.G[1] = w.w_0_1;
    w.G[2] = w.w_0_2;
    w.G[3] = w.w_0_1;
    w.G[4] = 0;
    w.G[5] = w.w_1_2;
    w.G[6] = w.w_0_2;
    w.G[7] = w.w_1_2;
    w.G[8] = 0;
}

void net_send(uint8_t* tran_data, uint8_t dest, uint8_t length)
{//fucntion to send data received from the transport layer to the dll layer
    sent_count = sent_count+1;
    if (sent_count == 20)
    {
        reset_weights();
    }

    packet.control[0] = 0;
    packet.control[1] = 0;
    packet.src_address = system_address;
    packet.dest_address = dest;
    packet.length = length;
    int i = 0;
    for(i; i<length; i++)
        {
        packet.tran_segment[i] = tran_data[i];
        }

    num_vert = num_nodes*((num_nodes-1)/2);
    next_node = calculate_next_node(num_vert,packet.src_address,packet.dest_address);
            increase_weights();
    create_net_packet();

    //phil's dll_send(net_data,next_node,data_size)
    //put_str("got to this point\n\r");
    dll_send(net.net_packet,next_node,15);
    put_str("Sent Data To Physical Layer\n\r");

    /*i = 0;
    char text[4];
    for(i; i<15; i++)
    {
        sprintf(text, "%d ",net.net_packet[i]);
        put_str(text);
    }
    put_str("\n\r");*/
}
```

```
void increase_weights()
{
   if(packet.src_address == 0)
   {
      if(packet.dest_address == 1)
      {
         w.w_0_1 = w.w_0_1 + 1;
      }
   }

   if(packet.src_address == 1)
   {
      if(packet.dest_address == 0)
      {
         w.w_0_1 = w.w_0_1 + 1;
      }
   }

   if(packet.src_address == 0)
   {
      if(packet.dest_address == 2)
      {
         w.w_0_2++;
      }
   }

   if(packet.src_address == 2)
   {
      if(packet.dest_address == 0)
      {
         w.w_0_2++;
      }
   }

   if(packet.src_address == 1)
   {
      if(packet.dest_address == 2)
      {
         w.w_1_2++;
      }
   }

   if(packet.src_address == 2)
   {
      if(packet.dest_address == 1)
      {
         w.w_1_2++;
      }
   }
```

```
        w.G[0] = 0;
    w.G[1] = w.w_0_1;
    w.G[2] = w.w_0_2;
    w.G[3] = w.w_0_1;
    w.G[4] = 0;
    w.G[5] = w.w_1_2;
    w.G[6] = w.w_0_2;
    w.G[7] = w.w_1_2;
    w.G[8] = 0;
        put_str("Increased Weights\n\r");
}

void net_resend(uint8_t* net_data, uint8_t length)
{//function to send data from an intermediate node to the final node
    int i=0;
    for(i; i<length; i++)
    {
        net.net_packet[i] = net_data[i];
    }
    num_vert = num_nodes*((num_nodes-1)/2);
    next_node = calculate_next_node(num_vert,system_address,net.net_packet[2]);

    //phil's dll_send(net_data,next_node,data_size)
    dll_send(net.net_packet,next_node,15);

    put_str("Data Resent To Physical Layer\n\r");
}

void net_receive(uint8_t* net_data, uint8_t length)
{//function to send data from the dll layer to the transport layer
    int i=0;
    for(i; i<length; i++)
    {
        net.net_packet[i] = net_data[i];
    }

    put_str("Received Data\n\r");
    i = 0;
    char text[4];
    for(i;i<length;i++){

        sprintf(text, "%d ",net.net_packet[i]);
        put_str(text);
    }

    put_str("\n\r");

    net_checksum();

    if(packet.checksum[0] != net_data[13])
```

```
    {
      put_str("Checksum 1 Passed\n\r");
    }

    if(packet.checksum[1] != net_data[14])
    {
      put_str("Checksum 2 Passed\n\r");
    }

    if(net.net_packet[0] == 0)
    {
      if(net.net_packet[3] == system_address)
      {
        uint8_t tran_data[8];
        put_str("Reached Final Node\n\r");
        int i=0;
        for(i; i<8; i++)
        {
          tran_data[i] = net.net_packet[i+5];
        }

      //ket's tran_recieve(tran_data,len)
        put_str("Sent Data To Transport Layer\n\r");
        trans_netw_receive(tran_data, net.net_packet[2]);
      }
      else
      {
       net_resend(net.net_packet,15);
      }
    }

    if(net.net_packet[0] == 1)
    {
     for(i; i<8; i++)
      {
        w.G[i] = (w.G[i] + net.net_packet[i+5])/2;
      }
    }
}

void net_update_weigts()
{//fucntion to update the weights across all nodes
    w.G[0] = 0;
    w.G[1] = w.w_0_1;
    w.G[2] = w.w_0_2;
    w.G[3] = w.w_0_1;
    w.G[4] = 0;
    w.G[5] = w.w_1_2;
    w.G[6] = w.w_0_2;
    w.G[7] = w.w_1_2;
```

```
    create_weight_packet();
    put_str("Flooded Weight Packet\n\r");
    int i;
    i = 0;
    char text[4];
    for(i; i<8; i++)
        {
        sprintf(text, "%d ",w.G[i]);
        put_str(text);
        }
    put_str("\n\r");
    //phil's flood function
    dll_flood(net.net_packet,15);
}

void create_weight_packet()
{//function to create the weight packet to be sent
    net.net_packet[0] = 1;
    net.net_packet[1] = 0;
    net.net_packet[2] = 0;
    net.net_packet[3] = 0;
    net.net_packet[4] = 8;
    int i = 0;
    for(i; i<8; i++)
        {
        net.net_packet[i+5] = w.G[i];
        }
    net_checksum();
    net.net_packet[8+5] = packet.checksum[0];
    net.net_packet[8+6] = packet.checksum[1];

    put_str("Created Weight Packet\n\r");

    i = 0;
    char text[4];
    for(i; i<15; i++)
        {
        sprintf(text, "%d ",net.net_packet[i]);
        put_str(text);
        }
    put_str("\n\r");
}

void reset_weights()
{//function to reset the weights to the default value
    w.w_0_1 = 1;
    w.w_0_2 = 1;
    w.w_1_2 = 1;
```

```
        w.G[0] = 0;
    w.G[1] = w.w_0_1;
    w.G[2] = w.w_0_2;
    w.G[3] = w.w_0_1;
    w.G[4] = 0;
    w.G[5] = w.w_1_2;
    w.G[6] = w.w_0_2;
    w.G[7] = w.w_1_2;
    w.G[8] = 0;

    put_str("Resetting Weights\n\r");
}

void net_flood(uint8_t* tran_data, uint8_t length)
{//function to flood the network
    packet.control[0] = 1;
    packet.src_address =  tran_data[2];
    packet.dest_address = tran_data[3];
    packet.length = length;
    int i = 0;
    for(i; i<length; i++)
        {
        packet.tran_segment[i] = tran_data[i];
        }
    create_net_packet();
    //phil's flood()
    //dll_flood(net.net_packet,15);
}

void create_net_packet()
{//function to create the network packet to be sent
    net.net_packet[0] = packet.control[0];
    net.net_packet[1] = packet.control[1];
    net.net_packet[2] = packet.src_address;
    net.net_packet[3] = packet.dest_address;
    net.net_packet[4] = packet.length;
    int i = 0;
    for(i; i<packet.length; i++)
        {
        net.net_packet[i+5] = packet.tran_segment[i];
        }
    net_checksum();
    net.net_packet[packet.length+5] = packet.checksum[0];
    net.net_packet[packet.length+6] = packet.checksum[1];

    put_str("Created Net Packet\n\r");
    i = 0;
    char text[4];
    for(i; i<15; i++)
        {
```

```
        sprintf(text, "%d ",net.net_packet[i]);
        put_str(text);
        }
           put_str("\n\r");

}
int calculate_next_node(int n,int startnode,int endnode)
{//function to calculate the next node
   int G[MAX][MAX];
   int ii = 0;
   int jj = 0;
   int k = 0;
   for(jj=0; jj<3; jj++)
   {
      for(ii=0; ii<3; ii++)
      {
         G[ii][jj] = w.G[k];
         k = k+1;
      }
   }

   put_str("Created Weight Matrix\n\r");

   ii = 0;
   jj = 0;
   for(jj=0; jj<3; jj++)
   {
      char text[4];
      for(ii=0; ii<3; ii++)
      {
         sprintf(text, "%d ",G[ii][jj]);
         put_str(text);
      }
   }
   put_str("\n\r");


         int cost[MAX][MAX],distance[MAX],pred[MAX];
         int visited[MAX],count,mindistance,nextnode,i,j;

         //pred[] stores the predecessor of each node
         //count gives the number of nodes seen so far
         //create the cost matrix
         for(i=0;i<n;i++)
                  for(j=0;j<n;j++)
                           if(G[i][j]==0)
                                    cost[i][j]=MINI_INFINITY;
                           else
                                    cost[i][j]=G[i][j];
```

```
	//initialize pred[],distance[] and visited[]
	for(i=0;i<n;i++)
	{
		distance[i]=cost[startnode][i];
		pred[i]=startnode;
		visited[i]=0;
	}

	distance[startnode]=0;
	visited[startnode]=1;
	count=1;

	while(count<n-1)
	{
		mindistance=MINI_INFINITY;

		//nextnode gives the node at minimum distance
		for(i=0;i<n;i++)
			if(distance[i]<mindistance&&!visited[i])
			{
				mindistance=distance[i];
				nextnode=i;
			}

			//check if a better path exists through nextnode
			visited[nextnode]=1;
			for(i=0;i<n;i++)
				if(!visited[i])
					if(mindistance+cost[nextnode][i]<distance[i])
					{
						distance[i]=mindistance+cost[nextnode][i];
						pred[i]=nextnode;
					}
		count++;
	}

	/*for(i=0;i< ((int)(sizeof(pred) / sizeof(pred[0])));i++)
{
   put_str("%d\n",pred[i]);
	}*/

int currnode = endnode; //(node you want to reach)
while (pred[currnode] != startnode){
  currnode = pred[currnode];
}
put_str("Calculated Next Node\n\r");
char text[20];
	sprintf(text, "Next node to %d is %d\n\r",endnode,currnode);
put_str(text);
return currnode;
```

```
        //print the path and distance of each node
        /*for(i=0;i<n;i++)
                if(i!=startnode)uint8_t create_packet()
                {uint8_t create_packet()
                        printf("\nDistance of node%d=%d",i,distance[i]);
                        printf("\nPath=%d",i);

                        j=i;uint8_t net_packet[128];
                        do
                        {
                                j=pred[j];
                                printf("<-%d",j);
                        }while(j!=startnode);
        }*/
}

void net_update()
{
   if (time < system_time) {
      time = system_time + 15000;
      net_update_weigts();
   }

}
uint8_t countSetBits(uint8_t n)
{//function that counts the number of 1s in a byte
   uint8_t count = 0;
   while (n) {
      count += n & 1;
      n >>= 1;
   }
   //printf("Count is %d",count);
   return count;
}

void net_checksum()
{//function that calculates the checksum
   uint8_t arr_1[5];
   uint8_t arr_2[8];
   uint8_t byte = 0;
   uint8_t no_1 = 0;
   uint8_t total_1 = 0;
   uint8_t total_2 = 0;

   int i=0;
   for(i;i<5;i++)
   {
    byte = net.net_packet[i];
    no_1 = countSetBits(byte);
    //put_str("%d",no_1);
```

```
    if(no_1%2 == 0)
    {
      arr_1[i] = 0;
    }

    else
    {
      arr_1[i] = 1;
    }
    total_1 = total_1 + pow(arr_1[i],i);

  }

  i= 5;
  for(i;i<13;i++)
  {
    byte = net.net_packet[i];
    no_1 = countSetBits(byte);

    if(no_1%2 == 0)
    {
      arr_2[i] = 0;
    }

    else
    {
      arr_2[i] = 1;
    }

    total_2 = total_2 + pow(arr_2[i],i);
  }

  packet.checksum[0] = total_1;
  packet.checksum[1] = total_2;
}
```