Cairo University

Network Project

Semester [CMP405_A6]

Team: Bit Suffering

Name	Email	Sec	B.N.
Evram yousef	evramyousef@gmail.com	1	8
Omar Ahmed	omar.ahmed983@eng-st.cu.edu.eg	1	36
Muhammad Sayed	muhammad.mahmoud98@eng-st.cu.edu.eg	2	14
Kareem Osama	kareemosamasobeih@gmail.com	2	5

Table of Contents

Implemented Functionalities

Mesh Architecture

Implementing “Mesh” architecture, we designed an “Orchestrator” simple node module, that handles the generation and distribution of messages among all other nodes.

Related Functions

1- *orchestrate_msgs(int line_index)*: line_index is the index of the line at the randomly generated msgs.txt file -using python-, this function allows the orchestrator to order a message to a certain node with randomly generated variables.

2- *buffer_msg (cMessage *msg)*: when a node[i] receives a message from Orchestrator, it immediately calls this function to schedule a self-message so that it could be buffered at the appropriate time of sending, and to be sent if the window size allows it.

3- *initialize() and schedule_self_msg(int line_index)*: These functions are responsible for the initialization of the orchestrator, sending the very first message, and scheduling self messages to the orchestrator to work in a non-stopping manner.

Orchestrator Order Message

In order to make the “Orchestrator” model the existence of Network layer, it sends “Raw” string messages to node[i], appending information like: receiver id, and time to send this message to it (ie. the time this message will be buffered to be sent). For this type of messages -that we called Orchestrator_order_base- we inherited the cMessage module and appended the appropriate information to it.

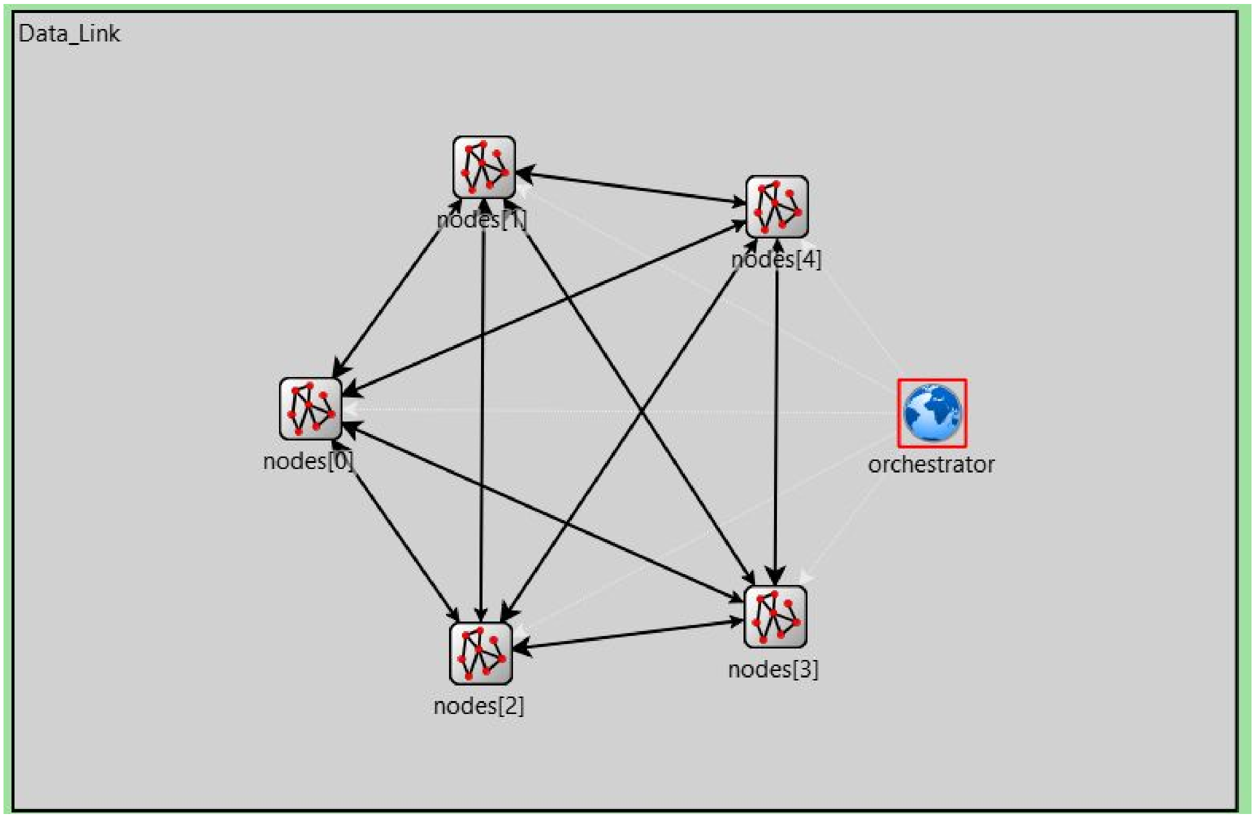
```
// TODO generated message class
//
cplusplus {{
//  Any includes goes here
    #include <string>
    typedef std::string message_str;
}}
class noncobject message_str;
packet Orchestrator_order {
    @customize(true);  // see the generated C++ header for more info
    int sender_id;
    int recv_id;
    double interval;
    message_str message_body;
}
```

(Figure 1: orchestrator_order.msg content)

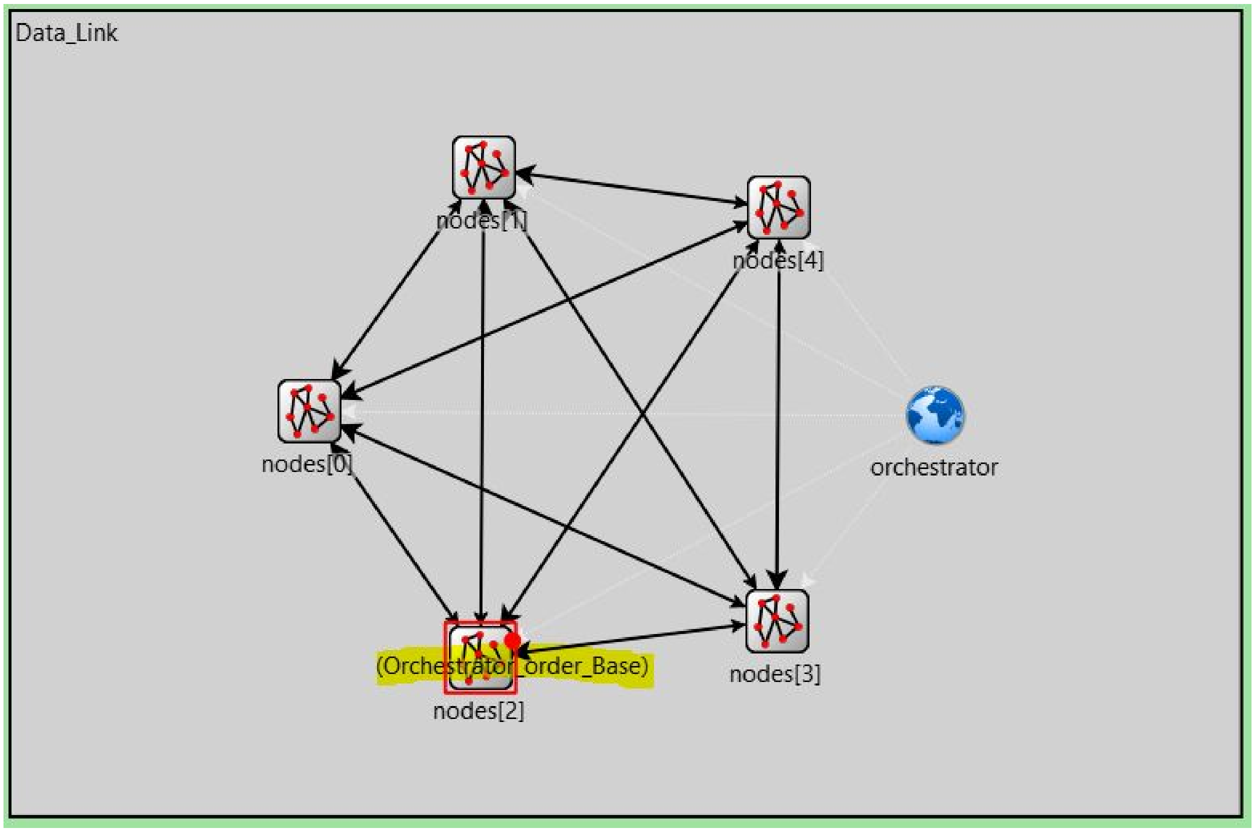
Orchestrator

The orchestrator node has a simple yet important role, it works infinitely -can be moduled- providing messages to random nodes that contains [sender id, receiver id, interval, message string].

These messages are called orders, because they give information to a node (i) and order it to send it to node (j) (where: i != j) and (j does not receive two consecutive messages).



(Figure 2: Architecture Before Orchestrator sends an Order)



(Figure 3: Orchestrator Orders nodes[2] with a message)

As seen in fig.1, the construction of an order, now nodes[2] will handle this order through *Node::buffer_msg(cMessage *msg)* function. Which schedule a self-message with its content to be received at the buffering time, to be appended at the buffer, and to be sent to its destination if the window size allows it.

Framing

- We implemented byte stuffing
- It can be found in Node.cc
- It consists of 2 functions: *byte_stuff* and *byte_destuff*

Byte stuffing

- It takes in a message string, creates a frame and adds a flag byte at the start and end of the frame.
- It also adds the ESC byte when it encounters a flag or ESC byte in the message.
- It then converts the message of bytes into a vector of booleans, then sets the payload of the frame to that vector and returns the frame.

```
Frame_Base* Node::byte_stuff (const std::string& msg)
{
   std::cout <<"Add byte stuffing.\n";
   // add byte stuffing
   std::vector<char>bytes;
   bytes.push_back(FLAG);
   for (int i = 0; i < (int)msg.size(); i++)
   {
       char byte = msg[i];
       if (byte == FLAG || byte == ESC)
           bytes.push_back(ESC);
       bytes.push_back(byte);
   }
   bytes.push_back(FLAG);
   // frame and transform to bits
   std::vector<bool> payload;
   for (int i = 0; i < (int)bytes.size(); i++)
   {
       std::bitset<8> bits = std::bitset<8>(bytes[i]);
       for (int j = 7; j >= 0; j--)
           payload.push_back(bits[j]);
   }
   Frame_Base* frm = new Frame_Base();
   frm->setPayload(payload);
   return frm;
}
```

Byte destuffing

- It does the opposite of byte stuffing, ignoring the start and end byte of the payload.
- It Converts the bits into bytes.
- It removes the extra ESC bytes.
- It returns the final reconstructed message string.

```
std::string Node::byte_destuff (Frame_Base* frame)
{
    std::cout <<"Remove byte stuffing.\n";

    std::vector<bool> payload = frame->getPayload();
    std::vector<char> bytes;
    for (int i = 8; i < (int)payload.size()-8;)
    {
        char byte = 0;
        for (int j = 7; j >= 0; j--, i++)
            if (payload[i])
                byte |= (1<<j);
        bytes.push_back(byte);
    }

    std::string msg;
    for (int i = 0; i < (int)bytes.size(); i++)
    {
        char byte = bytes[i];
        if (byte == ESC)
        {
            i++;
            if (i >= (int)bytes.size())
            {
                std::cout<<"Error: There was an ESC without character
at the end of the frame";
                break;
```

```
            }
            byte = bytes[i];
        }
        msg.push_back(byte);
    }
    return msg;
}
```

Error detection and correction

- We implemented hamming code
- It can be found in Node.cc
- It consists of 2 functions: *add_hamming* and *error_detect_correct*

Add hamming

It takes in a frame and modifies its payload vector, adding the parity bits of hamming code.

```
void Node::add_haming (Frame_Base* frame)
{
    std::cout<<"Add hamming\n"<<endl;
    std::vector<bool> payload = frame->getPayload();
    int m = payload.size();
    int r = 0;
    while ((1<<r) < r+m+1)
        r++;
    std::vector<bool> hamming(r+m+1);
    for (int i = 1, j = 0; i <= r+m; i++)
    {
        // if i not a power of 2
        // (number of ones in binary representation of i not equal
one)
        if(__builtin_popcount(i) != 1)
            hamming[i] = payload[j++];
    }
```

```
    for (int i = 0; i < r; i++)
    {
        int num = (1<<i);
        for (int j = num+1; j <= r+m; j++)
        {
            if (j&num)
                hamming[num] = hamming[num] ^ hamming[j];
        }
    }
    frame->setPayload(hamming);
}
```

Error detect and correct

- It detects the one bit error -if any- and corrects it using hamming code.
- It removes the extra parity bits of hamming code.
- It returns true if there was a detected and corrected error, and false otherwise.

```
bool Node::error_detect_correct (Frame_Base* frame)
{
    std::vector<bool> payload = frame->getPayload();
    int z = payload.size();
    int r = ceil(log2(z));
    int m = z-r-1;
    int errorBit = 0;
    for (int i = 0; i < r; i++)
    {
        int num = (1<<i);
        bool parity = payload[num];
        for (int j = num+1; j <= r+m; j++)
        {
            if (j&num)
                parity ^= payload[j];
        }
        if (parity)
```

```
                errorBit |= num;
        }
        if (errorBit)
        {
            payload[errorBit] = payload[errorBit] ^ 1;
            std::cout<<"Error detected at bit "<<errorBit<<" and
corrected\n";
        }
        else
            std::cout<<"There was no error detected in the frame\n";

        std::vector<bool> finalPayload(m);
        for (int i = 3, j = 0; i <= r+m; i++)
        {
            if (__builtin_popcount(i) != 1)
                finalPayload[j++] = payload[i];
        }

        frame->setPayload(finalPayload);
        if (errorBit)
            return true;
        return false;
    }
```

Sliding Window Protocol:

Go Back N:

- we implemented go back N with maximum window size 7 (m = 3)

Noises

- Implementing (modification - delay - loss - duplicate) noises
- Found in Node.cc

- 4 functions: *modify_msg* , *delay_msg* , *loss_msg* and *dup_msg*

Modify_msg

- It modifies the message based on a uniform distribution since they are all equally probable.
- To choose if the msg will be modified or not, it follows bernoulli trials.

```
/**
 * Corrupting msg follows bernoulli trails to check if corruption will be performed or not
 * since the events are all equally probable so:
 * The generated random number follows uniform distribution
 * The corruption itself follows uniform distribution
 * The Corrupted index(bit) in the payload follows uniform distribution
 */
void Node::modify_msg (Frame_Base* frame)
{
    double rand_corrupt = uniform(0,1);

    // double p_corrupt = 0.7;
    double p_corrupt = par("p_corrupt").doubleValue();

    if(rand_corrupt < p_corrupt )
    {
        int payloadSize = frame->getPayload().size();
        if(payloadSize)
        {
            int rand_corrupt_index = uniform(0,1)*payloadSize;
            std::cout<<"index = "<<rand_corrupt_index<<endl;
            message_vec modified_msg = frame->getPayload();
            modified_msg[rand_corrupt_index] = !modified_msg[rand_corrupt_index];
            frame->setPayload(modified_msg);
        }
    }

    return;
}
```

-

delay_msg

- To choose if the msg will be delayed or not, it follows bernoulli trials.
- It returns boolean to delay or not and the delayed time.
- Other logic is handled in the main.

```
/*
 *  Delaying msg using bernoulli distribution
 */
bool Node::delay_msg (double& delayedTime)
{
    double rand_delay = uniform(0,1);
    double p_delay = par("p_delay").doubleValue();
    // double p_delay = 0.6;

    if(rand_delay < p_delay )
    {
        double rand_delay = uniform(0,1)*par("delay_range").doubleValue();
        delayedTime = rand_delay;
        return true;
    }
    delayedTime = 0;
    return false;
}
```

-

loss_msg

- To choose if the msg will be lost or not, it follows bernoulli trials.
- It returns boolean to delay or not.
- Other logic is handled in the main.

```
/*
 *   losing msg using bernoulli distribution
 */
bool Node::loss_msg ()
{
    double rand_loss = uniform(0,1);
    double p_loss = par("p_loss").doubleValue();
    // double p_loss = 0.6;

    if(rand_loss < p_loss )
        return true;
    return false;
}
```

-

dup_msg

- To choose if the msg will be duplicated or not, it follows bernoulli trials.
- It returns boolean to duplicate or not.
- Other logic is handled in the main.

```
/*
* duplicating msg using bernoulli distribution
*/
bool Node::dup_msg ()
{
    double rand_dup = uniform(0,1);
    double p_dup = par("p_dup").doubleValue();
    // double p_dup = 0.65;

    if(rand_dup < p_dup )
        return true;
    return false;
}
```

-

Work Distribution

Team Member	Activities
Omar Ahmed	• Transmission channel noise modeling
Kareem Osama	• Go back N protocol
Muhammad Sayed	• Framing • Error detection and correction
Evram Yousef	• Network architecture (Distributed) • Orchestrator module and messages generation