Cairo University - Faculty of Engineering

Computer Engineering Department

Computer Networks Project

Team One

Mohamed Shawky Zaky
SEC:2, BN:15

Remonda Talaat Eskarous
SEC:1, BN:19

Mohamed Ahmed Mohamed Ahmed
SEC:2, BN:10

Mohamed Ramzy Helmy
SEC:2, BN:13

Contents

List of Figures

1 Overall System

Our system is a **Centralized Network**, consisting of N nodes connected to a hub, the number of nodes N are given from external file. This topology is, also, referred to as *star network*. Since, we use *full-duplex* communication, then each **node** acts as a sender and a receiver at the same time, sending data and acknowledges *(piggybacked)*.

The **hub** has the following functionalities:

- Allocate sessions between nodes.
- Navigate packets and acknowledges between peers.
- It's considered the control device of access medium, so it's responsible for the different types of channel noise.
- Gather the required statistics for each node.

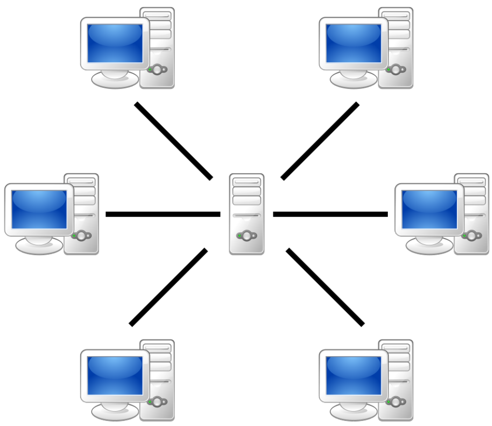

Figure 1: Centralized Star Network.

2 Implemented Functionalities

2.1 Hamming Code (Error Detection/Correction)

Hamming code is implemented to handle $1-bit$ error detection and correction. Since the message modification can only be of 1 bit, the usage of hamming code ensures that **no corrupted frames** are received. The algorithm implementation follows exactly the hamming code algorithm. *Hamming code* is applied on the message payload and the result is padded with zeros to be sent as a **string of characters**.

The implementation of the *hamming code* algorithm can be found in `src/Node.cc` under two functions :

- `string Node::computeHamming(string s, int &to_pad);`

```
std::string Node::computeHamming(std::string s, int &to_pad)
{
int str_length = s.length();
int m = 8 * str_length;
int r = 1;
while(m + r + 1 > pow(2 , r))
{
r++;
}
...
}
```

- `string Node::decodeHamming(string s, int padding);`

```
std::string Node::decodeHamming(std::string s, int padding)
{
if (!s.size())
{
return "";
}
int str_length = s.length();
int mr = 8 * str_length - padding;
int r = 1;
while(mr + 1 > (pow(2,r)))
{
r++;
```

```
}
...
}
```

2.2 Character Count (Framing)

Character Count is implemented as a **framing method** of the transmitted message. It is applied to the **message payload** to count characters of the string and **prepend** the count as one byte to the beginning of the **message payload**. The *decoding* is done in the same manner. *Character count* framing is applied **before** hamming code.

The implementation of the *character count* algorithm can be found in `src/Node.cc` under two functions :

- `string Node::addCharCount(string msg);`

```
std::string Node::addCharCount(std::string msg)
{
uint8_t msg_size = (uint8_t) msg.size() + 1;
std::string framed_msg = "";
framed_msg += msg_size;
framed_msg += msg;
return framed_msg;
}
```

- `bool Node::checkCharCount(string &msg);`

```
bool Node::checkCharCount(std::string &msg)
{
if (!msg.size())
{
return false;
}
int msg_size = (int)msg[0] - 1;
msg.erase(msg.begin());
if (msg_size == msg.size())
{
return true;
}
return false;
}
```

2.3 Go Back N (Sliding Window Protocol)

The used **data link protocol** is *Go Back N*, where the sender re-transmits the whole current window upon **acknowledge timeout**. The node can receive a message *(of kind 4)* from the hub, so that it can start talking to another node. Also, the node receives a message *(of kind 5)* to mark the end of the session and it responds with a message *(of kind 4)* to indicate its last acknowledged frame. It can, also, send and receive messages *(of kind 3)* to and from the other node, which contains both data and acknowledges *(piggybacked)*. Finally, the node has two kinds of self-messages. *One* is used for marking **acknowledge timeout**, so that the **window pointer** is reset. *Two* is used for sending **piggybacked messages** at certain intervals. Moreover, the node can have 3 states :

1. **Active state :** where the node is currently participating in a session.
2. **Inactive state :** where the node is not in a session.
3. **Dead state :** where the node has no more messages to send.

So, we can summarize our *Go Back N* implementation as follows :

- The node receives a message *(of kind 4)* from the hub, so that it can start transmission.
- The node sends its first **piggybacked message** with acknowledge of -1 and sends a self-message *(of kind 2)* to schedule **next message** to be sent and a self-message *(of kind 1)* to set **acknowledge timeout**.
- When the node receives a message from the other node *(of kind 3)*, it decodes the **incoming frame** to advance its **excepted frame count**. Also, it decodes the **incoming acknowledge** to advance **its window**.
- When the node receives self-message *(of kind 1)*, it checks whether the window is advanced. Accordingly, it can reset **window pointer**.
- When the node receives self-message *(of kind 2)*, it sends the next message to the hub.
- When the session times out, the node receives an *"end session"* message *(of kind 5)* from the hub. The node turns into **inactive state** and sends its last acknowledged frame to the hub.
- If the last message of the node is acknowledged, the node turns into **dead state**.

The implementation of the *Go Back N* protocol can be found in `src/Node.cc` under functions :

- `void Node::handleMessage(cMessage *msg);`

```
void Node::handleMessage(cMessage *msg)
{
if (!this->is_dead || msg->getKind() == 5)
{
if (msg->isSelfMessage() && !this->is_inactive)
{
switch(msg->getKind())
{
case 1:
{
...
}
case 2:
{
...
}
}
}
else
{
if (msg->getKind() == 3 && !this->is_inactive)
{
...
}
else if (msg->getKind() == 4)
{
...
}
else if (msg->getKind() == 5 && !this->is_inactive)
{
...
}
}
}
}
```

- `void Node::sendMsg();`

```
void Node::sendMsg()
{
if (this->S <= this->Sl)
{
std::string framed_msg = this->addCharCount(this->msgs[this->S]);
const char* msg_payload = framed_msg.c_str();
int padding = 0;
...
}
cMessage * send_next_self_msg = new cMessage("");
send_next_self_msg->setKind(2);
scheduleAt(simTime() + 1 , send_next_self_msg);
}
```

- void Node::post_receive_ack(cMessage *msg);

```
void Node::post_receive_ack(cMessage *msg)
{
if (((Imessage_Base *)msg)->getAcknowledge() == -1)
{
return;
}
while(this->Sf <= (((Imessage_Base *)msg)->getAcknowledge()))
{
this->Sf ++;
}
...
}
```

- void Node::post_receive_frame(cMessage *msg);

```
void Node::post_receive_frame(cMessage *msg)
{
int received_frame_seq_num =
((Imessage_Base *)msg)->getSequence_number();
if (received_frame_seq_num == this->R)
{
this->ack = R;
...
bool check = this->checkCharCount(payload);
if (check)
{
```

```
EV << " message char count right !" <<endl;
}
else
{
EV << " message char count wrong !" << endl;
}
}
}
```

2.4 Transmission Channel Noise Modelling

As mentioned, the hub acts as a **medium controller**, that is why the hub is responsible for adding **noise** and **delay** to the transmitted messages. The implemented noise types are *1-bit* **modification** *(on message payload)*, **delay**, **drop** and **duplication**. The noise is added with **random probability** and **multiple types** might be applied on a single message at once.

The implementation of the *noise modelling* can be found in `src/Hub.cc` under function :

- `int Hub::applyNoise(Imessage_Base *msg);`

```
int Hub::applyNoise(Imessage_Base * msg)
{
// generate random action from inverted actions
// ( losing , delaying , none )
int choice =uniform(0,3);
if (choice == 0)
{
EV << " message lost !" << endl;
return 0;
}
...
int prob_modify  = uniform(0,1)*10;
...
if (choice == 2)
{
return 2;
}
else
{
return 3;
```

```
}
}
if (choice == 2)
{
return 1;
}
// if none return 4
return 4;
}
```

2.5 Centralized Network Architecture

Since we are implementing a **centralized network**, we use a **hub** to communicate between nodes. The hub is, mainly, responsible for allocating **sessions** in the following way :

- Generate a **table of pairs** at the beginning of the simulation that includes all the node pairs to communicate.
- At *each session time*, the hub starts a new session through sending the two nodes a message *(of kind 4)* to start transmission. The message contains the expected frame to be sent for the opposite nodes.
- The hub schedules a self-message *(of kind 1)*, as well, in order to mark the **beginning** of a new session.
- The hub continues to *re-direct* the messages, until the session *times out*.
- Once the session times out, the hub sends to the two active nodes an *"end session"* message and receives the last acknowledged frame from each of them.
- The hub starts a *new session* between two new nodes and **ignores** any other messages.

The implementation of the *centralized network* can be found in `src/Hub.cc` under function :

- `void handleMessage(cMessage *msg);`

```
void Hub::handleMessage(cMessage *msg)
{
if (msg->isSelfMessage())
```

```
{
if (msg->getKind() == 1)
{
...
}
else if (msg->getKind() == 2)
{
...
}
else if (msg->getKind() == 6)
{
...
}
}
else
{
if (msg->getKind() == 3)
{
...
}
else if (msg->getKind() == 4)
{
...
}
}
}
```

- `void generatePairs();`

```
void Hub::generatePairs()
{
int node1, node2;
for(int i =0; i < this->n*this->n; ++i)
{
...
}
}
```

- `void startSession();`

```
void Hub::startSession()
{
```

```
this->indexer ++;
if(this->indexer == this->senders.size())
{
this->indexer = 0;
}
this->sender = this->senders[this->indexer];
this->receiver = this->receivers[this->indexer];
...
}
```

- void parseMessage(Imessage_Base *msg);

```
void Hub::parseMessage(Imessage_Base * msg)
{
int msg_sender = msg->getSenderModule()->getIndex();
EV << " message sender id = "<< msg_sender << endl;
EV << " hub sender id = " << sender << endl;
EV << " hub receiver id = " << receiver << endl;
if (msg_sender != this->sender && msg_sender != this->receiver)
{
cancelAndDelete(msg);
return;
}
int msg_receiver;
if (msg_sender == this->sender)
{
msg_receiver = this->receiver;
}
else
{
msg_receiver = this->sender;
}
int noise = applyNoise(msg);
int delay = exponential( par("mean_delay").doubleValue());
EV << " delay time = " << delay <<endl;
std::string msg_payload = msg->getMessage_payload();
switch(noise)
{
...
}
this->last_sent_frames[msg_sender] =
```

```
msg->getSequence_number();
}
```

2.6 Statistics Gathering

The **statistics gathering** function is implemented inside the hub, since the hub is the one controlling the **whole transmission process**. Also, the **transmission noise** is created in the hub. So, the hub keeps track of the **generated**, **lost**, **re-transmitted** and **duplicated** message for every node. It sets up a self-message *(of kind 2)* to schedule the **statistics print**. **Statistics** can be printed for each node *separately* or *collective* for all nodes. So, the following statistics are printed :

1. The total number of **generated** frames.
2. The total number of **dropped** frames.
3. The total number of **re-transmitted** frames.
4. The **percentage** of useful transmitted data *(Efficiency of the system)*.

The implementation of the *statistics gathering* can be found in `src/Hub.cc` under function :

- `void Hub::initializeStats();`
- `void Hub::updateStats(int node_idx, int seq_num, int frame_size, bool is_dropped, bool is_duplicated);`

```
void Hub::updateStats(int node_idx, int seq_num, int frame_size,
bool is_dropped, bool is_duplicated)
{
...
if(it != this->generated_frames[node_idx].end())
{
int index = it - this->generated_frames[node_idx].begin();
// increase retransmission times
this->retransmitted_frames[node_idx][index] ++;
// increase drop times (if dropped)
if (is_dropped)
{
this->is_dropped[node_idx][index] ++;
}
```

```
// increase duplication times (if duplicated)
if (is_duplicated)
{
this->is_duplicated[node_idx][index] ++;
}
}
else
{
...
}
}
```

- void Hub::printStats(bool collective);

```
void Hub::printStats(bool collective)
{
if (collective)
{
// count all generated frames
int total_generated = 0;
for (int i = 0; i < this->n; i++)
{
total_generated += this->generated_frames[i].size();
}
// count all dropped frames
int total_dropped = 0;
...
// count all retransmitted frames
int total_retransmitted = 0;
for (int i = 0; i < this->n; i++)
{
total_retransmitted += std::accumulate(this->
retransmitted_frames[i].begin(),
this->retransmitted_frames[i].end(), 0);
}
float useful_data = 0;
float all_data = 0;
for (int i = 0; i < this->n; i++)
{
for (int j = 0; j < this->generated_frames[i].size(); j++)
{
```

```
useful_data += this->frame_sizes[i][j];
all_data += (this->retransmitted_frames[i][j] +
this->is_duplicated[i][j] + 1) *
(4*3 + this->frame_sizes[i][j]);
}
}
float efficieny = useful_data/all_data;.
...
}
else
{
...
}
}
}
```

3 Workload Division

Name	Workload
Remonda Talaat Eskarous	- Centralized Network *(Hub)*. - Transmission Channel Noise Modelling. - Code Integration and Refinement.
Mohamed Shawky Zaky	- Character Count. - Statistics Gathering. - Code Integration and Refinement. - Final Report.
Mohamed Ahmed Mohamed Ahmed	- Go Back N.
Mohamed Ramzy Helmy	- Hamming Code.