DEPARTMENT OF MATHEMATICS
AND COMPUTER SCIENCE

DM548 Conputer Architecture Project

Gabriel Howard Jadderson : gajad16@student.sdu.dk

DM557
November 2, 2017

List of Figures

1 Introduction

We are required to extend the capabilities of an implemented RDT (Reliable Data Transfer) model on the link layer to multiple neighbours.

The given implementation functions for a setup of two stations communicating to each other illustrated in figure 1.

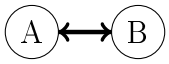

Figure 1: Original setup

The extended version of the implementation should work for a fixed number of neighbours, an example of which is given in figure 2, where three stations each have the other two stations as neighbours, which they both send to and receive from.

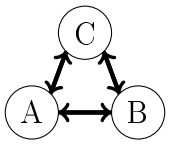

Figure 2: Three station setup

2 Design

The original design of the system was designed for two hosts to communicate reliably with each other. To expand on this system, the information associated with a neighbour needs to exist in one instance per neighbour. The neighbours are stored in an array of structs for ease and in order encapsulate the information associated with a neighbour.
Furthermore, a central component in multiway communication is a means of distinguishing which neighbour is which. As the stations on the subnet already have unique IDs, these IDs can be used to uniquely identify the neighbours in a linear time lookup, which has a simple implementation for arrays and won't take significant time for the array sizes being used. These station IDs are only used by the functions for which they are relevant though, as the general means of access for neighbour data are the neighbour IDs (array indices).

2.1 Stations and Neighbours

Neighbours are determined by indices in a neighbour array, called the neighbour IDs, and as such are transparently identified regardless of which actual station that neighbour represents. The ID of a station is stored in a neighbour, but is only meant to be utilized for sending to and receiving from the physical layer, and should be encapsulated as closely to these events as possible. When receiving, a neighbour ID for the given station should be returned.
With this method, the actual station represented by a neighbour is unimportant for how the system works at large.
It is not allowed for two neighbours to correspond to the same station.

2.2 Distinguishing Neighbours

When a station is sending to a neighbour, it knows the id of the neighbour it is sending. The change to be made from a sender's perspective is to add a parameter to the functions related in that process, which takes the id of the neighbour that is being sent to, rather than using station IDs, which can be looked up using the neighbour ID if necessary, such as for the to_physical_layer function, which sends the frames to the subnet based on the station ID. Furthermore, the communication from the fake network layer to the link layer, at the time the fake network layer prepares the package to send, needs to contain information about which neighbour the package should be sent to. This was decided on to be done by setting the msg parameter of the network_layer_ready signal to the neighbour ID of the neighbour to send the packet to, which the selective repeat function can extract from the signal directly.
When receiving, the station needs to be able to determine which neighbour sent a packet to it. Since the use of station IDs should be encapsulated as closely to the communication with the subnet as possible, it was decided that the function receiving frames from the subnet should perform the translation from station ID to neighbour ID in order to avoid having the selective repeat function interact with station IDs at all.

To make the expansion to multiple neighbours as simple as possible, a restraint was decided upon; only one neighbour's communication is processed at a given time. This means that the selective repeat function must alternate between the neighbours. Fortunately, this is possible to do for all the signals the selective repeat function processes with few alterations.

- *network_layer_ready*: The fake network layer identifies the neighbour to send to in the msg field of the signal.
- *frame_arrival*: The station ID of the sender is determined when the frame is received and is translated to the corresponding neighbour ID by the from_physical_layer function.
- *timeout*: The functions for timer control already require the neighbour ID to determine which neighbour it is starting a timer for. Add this information to the timer msg and extract the information when handling the signal in selective repeat.

A single variable can be used to hold this value and for the array indexing.

This approach avoids the need to run multiple selective repeat processes, which would cause excessive resource usage and would possibly be slower due to thread scheduling than the chosen solution.

2.3 variable locality

The local variables of the selective repeat function have been retained local, and as such a second struct for neighbours were devised to contain the neighbour data for the selective repeat function. This is to retain the variable locality that was present in the original implementation.

3 Implementation

3.1 Information restructuring

```
typedef struct {
    int stationID;
    int ack_timer_id;
    int timer_ids[NR_BUFS];
    boolean no_nak;
} neighbour;
```

Each neighbour will have it's own associated stationID, this way we can differentiate between each neighbour/station. Furtheremore each neighbour has it's own `ack_timer_id` that way we ensure that there's only one instance per neighbour Each neighbour will start it's own ack timer when they recieve a pakcet, this way we can ensure the piggy-backing remains functional.

The following struct is only used for the selective-repeat function, encapsulating our variables in a struct proved to be much more manageable.

```
typedef struct { //The data only visible to selective repeat.
  seq_nr ack_expected;              // lower edge of sender's window
  seq_nr next_frame_to_send;        // upper edge of sender's window + 1
  seq_nr frame_expected;            // lower edge of receiver's window
  seq_nr too_far;                   // upper edge of receiver's window + 1
  packet out_buf[NR_BUFS];          // buffers for the outbound stream
  packet in_buf[NR_BUFS];           // buffers for the inbound stream
  boolean arrived[NR_BUFS];         // inbound bit map
  seq_nr nbuffered;                 // how many output buffers currently used
} neighbour_SR_Data;
```

We have further implemented `packetTimerMessage` a struct that is sent to the library whenever we start a timer. This way we know which sequence the timer was started for and the respective neighbour id.

```
typedef struct {
  seq_nr k;
  neighbourid neighbour;
} packetTimerMessage;
```

3.2 Actual Code

3.2.1 Timers

We've changed the following functions to take the neighbours id's as parameters. Inside the function we've used the parameter to retrieve the neighbour specified from a global neighbours array. This allows us to stop and start the timers for each respective neighbour.

```
void start_timer(neighbourid neighbour, seq_nr k)
void stop_timer(neighbourid neighbour, seq_nr k)
void start_ack_timer(neighbourid neighbour)
void stop_ack_timer(neighbourid neighbour)
```

Our implementation of `start_timer` is as follows. We first allocate memory for our packet-TimerMessage The sequence number is updated accordingly to our parameter, on the `start_ack_timer` function however, we set the sequence number to -1 (not used) since it redundant in that case. Afterwards the neighbour id is updated accordingly and a timer is initiated from the SetTimer function, a function which starts a timer and returns the id of that timer, which we then store in our specified neighbour. we use modulo to wrap it around the index range since the sequence numbers increase linearly.

```
1 void start_timer(neighbourid neighbour, seq_nr k) {
2         packetTimerMessage *msg = malloc(sizeof(packetTimerMessage));
3         msg->k = k;
4         msg->neighbour = neighbour;
5
6   neighbours[neighbour].timer_ids[k % NR_BUFS] = SetTimer( frame_timer_timeout_millis, (
      void *)msg );
7   logLine(trace, "start_timer for seq_nr=%d timer_ids=[%d, %d, %d, %d] %s\n", k,
      neighbours[neighbour].timer_ids[0], neighbours[neighbour].timer_ids[1], neighbours[
      neighbour].timer_ids[2], neighbours[neighbour].timer_ids[3], msg);
8 }
```

To stop the timers we simply invoke the subnet function: `StopTimer` with the id of the timer and in return we get the associated **msg. (note: this is why we store the id of thet timer in the first place)

3.2.2 Neighbours

The neighbour switching is handled in the selective repeat using a variable called *currentNeighbour*. This variable is set at as early as possible when handling each of the three types of signals handled by selective repeat.
The snippets here are trimmed extensively. To get a clear overview, please read the event handling cases in the source file rdt.c. Comments have also been removed.

network_layer_ready:
The id of the neighbour the packet is intended to be sent to is stored in the msg variable of the event.
The fake network layer written for testing gives this value when signaling:

```
1 Signal(network\_layer\_ready, i);
```

And the selective repeat function reads this value when handling the event:

```
1 currentNeighbour = event.msg;
```

frame_arrival:
The id of the neighbour that sent the packet is returned by the function $from\_physical\_layer$.

```
1 currentNeighbour = from\_physical\_layer(&r);
```

timeout:
The timer message contains the id of the neighbour it enables.

```
1 currentNeighbour = ((packetTimerMessage*)event.msg)->neighbour;
```

3.2.3 Main Function

In main we've set up our neighbours/stations accordingly, then activate the networklayer and selective-repeat for each consequtive neighbour. this provides some limitations in terms of flexibility, essentially in the future we want to improve this setup to be more dynamic instead of it's current hard-coded state.

```
switch (ThisStation) {
case 1:
  neighbours[0].stationID = 2;
  neighbours[1].stationID = 3;
  neighbours[2].stationID = -1;
  neighbours[3].stationID = -1;
  break;
case 2:
  neighbours[0].stationID = 1;
  neighbours[1].stationID = 3;
  neighbours[2].stationID = -1;
  neighbours[3].stationID = -1;
  break;
case 3:
  neighbours[0].stationID = 1;
  neighbours[1].stationID = 2;
  neighbours[2].stationID = -1;
  neighbours[3].stationID = -1;
  break;
}

ACTIVATE(1, FakeNetworkLayer_Test1);
ACTIVATE(1, selective_repeat);

ACTIVATE(2, FakeNetworkLayer_Test1);
ACTIVATE(2, selective_repeat);

ACTIVATE(3, FakeNetworkLayer_Test1);
ACTIVATE(3, selective_repeat);
```

3.2.4 Signals

The following is a description of what the three events in our code do:

`network_layer_allowed_to_send`
: Signals the network layer that it can send a piece of data. Here the network layer should make sure the `from_network_layer_queue` contains at least one element, after which it can signal `network_layer_ready`, which means that the network layer has prepared at least one element in the queue.

`network_layer_ready`
: This signal signals to the link layer that the network layer has prepared an element to be sent in the `from_network_layer_queue` queue and that it should be sent now.

`data_for_network_layer`
: This signal signals to the network layer that the `for_network_layer_queue` contains a data

element for it to take care of. (a data element has been received)

4 Testing

The scenario used for testing is the one shown in figure 2.
Each station sends a message that looks like this:

$$\begin{aligned} Packet &= StoR \\ S &= ThisStation \\ R &= neighbours[receiver].stationID \end{aligned}$$

Where Packet is the packet that is sent to the receiver, S is the station ID of the sending station and R is the station ID of the receiving station.
For example, for station 1 sending to station 3: 1 *to* 3

The test was made with the three stations and an error frequency of 90% (command: ./network -pmynetwork -e900 -n3). This test was run multiple times to reduce the possibility of lucky streaks.

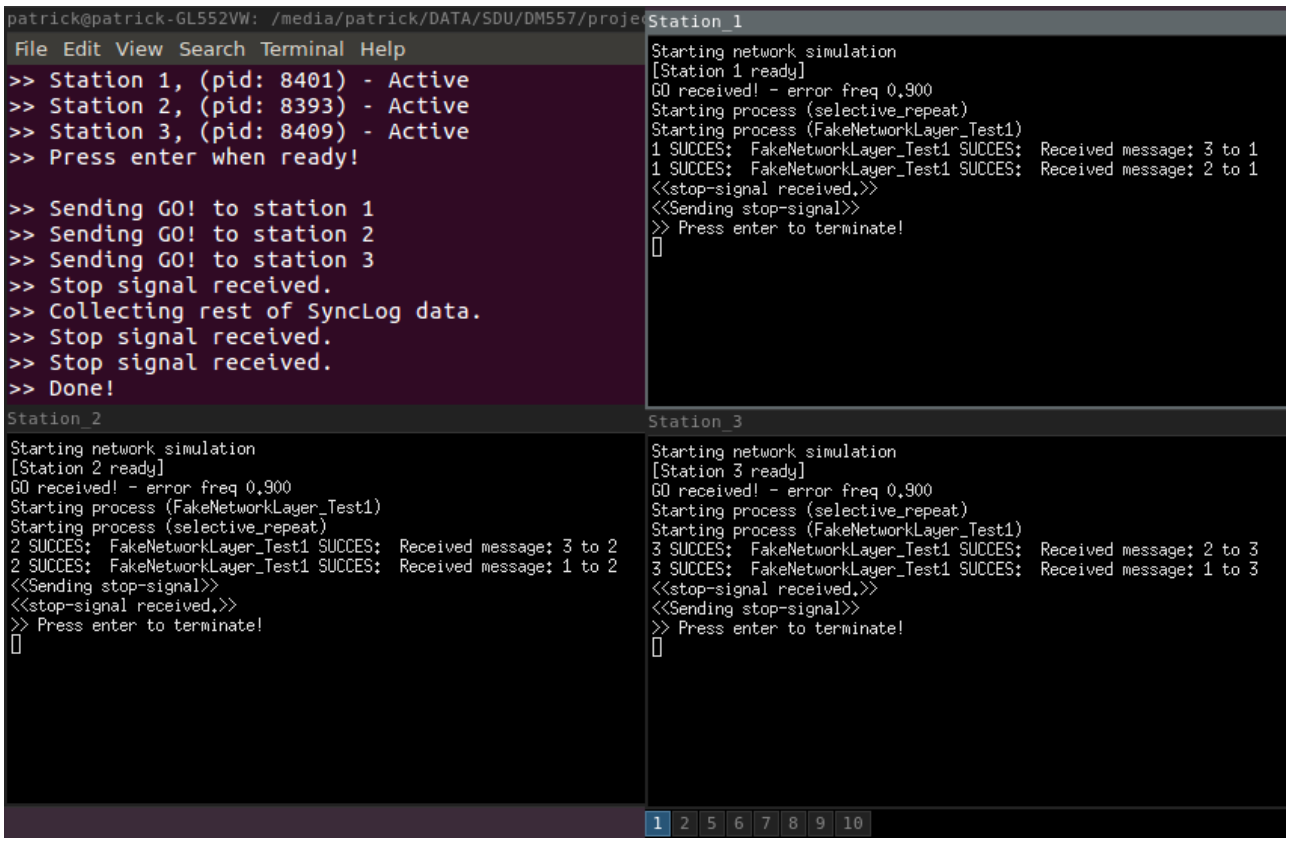

Figure 3: Test of three stations, each sending a message to the other two

As seen in figure 3, the three stations each received two messages, one from each of the other two stations.

5 Conclusion

A major initial challenge was to fully grasp how to implement the theoretical knowledge into the RDT implementation, as it was difficult for us to understand the interactions at play, especially with regard to the subnet library. This resulted in having to read the project description several times. Our initial approach was to expand on the data structures used in place, which proved to be a greater challenge than to organize the data properly with structs, which also provided a clearer perspective of the actual workings at play in the program dynamics.
From that point on, and after further investigation of the underlying systems, it was an almost trivial task to expand the system to support multiple neighbours.

Our testing has then verified that the hosts can communicate reliably with each other, which was done by running the simulation multiple times with a 90% error rate. (-e 900)

Based on the perceived reliability of the solution, based on the tests, it is concluded that RDT between multiple hosts on the link level has been correctly implemented.

A possible improvement of the system would be to support dynamic establishment of new connections. The current connection requires that the connections are preconfigured, which is only useful when the connections are known beforehand. Dynamically establishing connections would also require dynamic expansion of the number of neighbours and some means of tearing down connections as well.