Computer Networks – 1st Project Report

João Pereira, Nuno Pereira

October 29, 2022

1 Introduction

The objective of this project was to send files between two computers by using the serial port. To do this a specified protocol was implemented, separated into two distinct and independent layers, the data link layer and the application layer.

This objective was reached and resulted in an application capable of sending files between two computers with proper syncing and framing, connection establishment and termination, and error checking and handling as per the specifications.

2 Architecture

2.1 Layers

The code was divided into two distinct layers. This had the advantage of making the code more modular, reusable and versatile, as either layer could be swapped for another with a similar interface with minimal effort.

2.1.1 Data Link Layer

This is the lower level layer, it interfaces with the serial port driver directly. It is responsible for opening and closing a connection, and ensuring the data gets sent and received, without errors.

2.1.2 Application Layer

This is the higher level layer, it interfaces with the data link layer and with the file system. It is responsible for reading a file, breaking it up into chunks, sending these chunks through the data link layer, receiving these chunks, and assembling them back into a file.

2.2 Program Execution

The program can be executed by calling `app serial_port role file_name`, where:

- `app` is the program executable;
- `serial_port` is the serial port file path;
- `role` is one of `rx` or `tx`, for receiving or transmitting a file, respectively;
- `file_name` is the path of the file to be sent (ignored when receiving a file).

3 Code Structure

3.1 Data Link Layer

The Data Link Layer is centered on the `link_layer.c` file and the `link_layer` folder. The `link_layer.c` file contains the protocol interface that can be used by upper layers, the `timer.c` file contains code related to interfacing with POSIX timers, and the `frame.c` file contains code to create and manipulate frames.

3.1.1 link_layer.c

- ```
  typedef enum { LL_TX, LL_RX }
      LLRole;
  ```

  Defines the role of a link layer connection, either transmitter or receiver.

- ```
  typedef struct {
      LLRole role;
      struct termios old_termios;
      int fd;
      bool closed;
      uint8_t tx_sequence_nr;
      uint8_t rx_sequence_nr;
      int n_retransmissions_sent;
      timer_t timer;
      Frame *last_command_frame;
  } LLConnection;
  ```

  Defines the state of a link layer connection.

- ```
  LLConnection *llopen(
      const char *serial_port,
      LLRole role);
  ```

  Opens a new link layer connection and returns it.
  ```

- ```
  ssize_t llwrite(
      LLConnection *connection,
      const uint8_t *buf,
      size_t buf_len);
  ```

  Sends data through a connection, ensuring it gets received.

- ```
  ssize_t llread(
      LLConnection *connection,
      uint8_t *buf);
  ```

  Reads data from a connection.

- ```
  int llclose(
      LLConnection *connection);
  ```

  Closes a connection.

3.1.2 link_layer/timer.c

- ```
  void timer_setup(
      LLConnection *connection);
  ```

  Sets up a connection's POSIX timer.

- ```
  void timer_destroy(
      LLConnection *connection);
  ```

  Deallocates a connection's POSIX timer.

- ```
  void timer_arm(
      LLConnection *connection);
  ```

  Arms a connection's POSIX timer.

- ```
  void timer_disarm(
      LLConnection *connection);
  ```

  Disarms a connection's POSIX timer.

- ```
  void timer_force(
      LLConnection *connection);
  ```

  Forcibly calls a connection's timer handler.

- ```
  void timer_handler(
      union sigval val);
  ```

  The connection's timer handler. Resends the last command sent by the connection.

- ```
  void signal_handler();
  ```

  The connection's SIGCONT handler. Does nothing, is only needed so blocking syscalls can be stopped.

3.1.3 link_layer/frame.c

- ```
  typedef struct {
      uint8_t address;
      uint8_t command;
      ByteVector *information;
  } Frame;
  ```

  Defines all the data needed to represent a frame.

- ```
  Frame *create_frame(
      LLConnection *connection,
      uint8_t cmd);
  ```

  Creates a new frame.

- ```
  Frame *read_frame(
      LLConnection *connection);
  ```

  Reads a frame from the connection's serial port.

- ```
  ssize_t write_frame(
      LLConnection *connection,
      Frame *frame);
  ```

  Writes a frame to the connection's serial port.

- ```
  void frame_destroy(Frame *this);
  ```

  Deallocates all memory allocated by a frame.

- ```
  ssize_t send_frame(
      LLConnection *connection,
      Frame *frame);
  ```

  Sends a frame to the connection's serial port and sets up retransmission if the frame is a command.

- ```
  Frame *expect_frame(
      LLConnection *connection,
      uint8_t command);
  ```

  Continuously receives frames from a connection until a certain command is received. Calls `handle_frame` for all received frames.

- ```
  Frame *handle_frame(
      LLConnection *connection,
      uint8_t command);
  ```

  Does something in response to a frame, depending on it's type.

  For *SET* commands and *DISC* commands from the receiver, sends a *UA* frame in response.

For *DISC* commands from the transmitter, sends a *DISC* frame in response and expects a *UA* response.

For *I* commands, sends an *RR* frame in response, or a *REJ* frame if there was an error in the frame body.

For *UA* or *RR* responses, it calls `timer_disarm`.

For *REJ* responses, it calls `timer_force`.

3.2 Application Layer

The Application Layer is centered around the `application_layer.c` file and the `application_layer` folder. The `application_layer.c` file contains the client facing interface of the application which is responsible for all the higher level setup and communication between transmitter and receiver. The `packet.c` file contains code that is responsible for correctly creating control and data packets.

3.2.1 application_layer.c

- ```
  LLConnection *connect(
      const char *serial_port,
      LLRole role);
  ```

  Opens a connection to one of the computer's serial ports.

- ```
  int init_transmission(
      LLConnection *connection,
      const char *filename);
  ```

  Starts the communication process between transmitter and receiver.

- ```
  ssize_t receiver(
      LLConnection *connection);
  ```

  Executes the receiver's operational flow.

- ```
  ssize_t transmitter(
      LLConnection *connection,
      const char *filename);
  ```

  Executes the transmitter's operational flow.

- ```
  void application_layer(
      const char *serial_port,
      const char *role,
      const char *filename);
  ```

  Entrypoint for the application layer.

3.2.2 application_layer/packet.c

- ```
  ByteVector *create_start_packet(
      size_t file_size,
      const char *file_name);
  ```

  Creates a *START* packet containing the transmitted file's size and name.

- ```
  ByteVector *create_data_packet(
      const uint8_t *buf
      uint16_t size);
  ```

  Creates a *DATA* packet containing a chunk of the transmitted file.

- ```
  ByteVector *create_end_packet();
  ```

  Creates an *END* packet.

- ```
  ssize_t send_packet(
      LLConnection *connection,
      ByteVector *packet);
  ```

  Sends a packet using `llwrite`.

4 Main Use Cases

There are two main use cases: either sending a file or receiving a file. These use cases are described below.

4.1 Common behavior

For both cases described bellow, a `LLConnection` object is created by calling the `connect` function on the Application Layer. This function just calls `llopen` with the correct role and catches any errors.

In the end, both *receiver* and *transmitter* call `llclose` to end the communication process and print the execution time to the console.

4.2 Sending a file

When sending a file, the application layer first tries to open a descriptor to the specified file and initiates the transmission process by calling the `init_transmission`: this function sends a control packet to the receiver in order to broadcast that it's ready to send data through the underlying Data Link Layer.

Then, chunks of `PACKET_DATA_SIZE` bytes are read from the previously opened file descriptor and the actual number of bytes is returned. If it is:
  ```

- -1: An error has occurred while reading the file, break out of the transmitter loop and close both the file descriptor and serial connection.

- 0: The end of the file has been reached, so the application layer sends an *END* control packet to the receiver to signal the correct end of transmission.

- another value: The read operation was successful and the data is stored in a buffer, using it to send a *DATA* packet.

4.3 Receiving a file

When receiving a file, the application layer reads a packet and stores the number of bytes read. Then the packet is processed, starting with the packet type. If it is:

- *START_PACKET*: the packet is further deconstructed in order to collect the transmitted file's size and name. Then, a file descriptor is opened to the provided file.

- *DATA_PACKET*: the packet is deconstructed, first retrieving the packet's sequence number: if it does not match the expected sequence number then the program aborts since a critical failure has happened at the Data Link Layer.

  If the sequence number matches the expected one then the fragment size is retrieved in order to write to the file the data fragment received, which corresponds to the rest of the packet.

- *END_PACKET*: the application layer breaks out of the receiver loop in order to close connections and perform cleanup.

5 Link Layer Protocol

The link layer protocol's flow is as follows: open a connection using `llopen`, send arbitrary data using `llwrite`, receive arbitrary data using `llread`, and close the connection using `llclose`. Each step's implementation is described below.

This implementation uses an `LLConnection` struct to save the state of a connection and in theory allows multiple serial port connections to be open in a single process. In practice this was not required or tested.

5.1 llopen

```
LLConnection *llopen(
    LLConnectionParams params);
```

This function opens a connection to the serial port and creates a new Data Link Layer connection object.

Firstly, this functions calls `setup_serial` to setup the serial port connection, then it calls `timer_setup` to setup a POSIX timer to handle timeouts when sending commands. Afterwards, the implementation differs depending on the role of a connection.

From a transmitter's standpoint, a connection can only be considered **open** when a receiver is ready to receive data. As so, this function implements a syncing procedure and sends a *SET* command, which must get a *UA* response in return. If a response is not received, the command is resent a configurable amount of times, with an also configurable amount of time between them.

On the other hand, a receiver's connection can be considered open immediately, which means this syncing procedure is not handled by this function.

5.2 llwrite

```
ssize_t llwrite(
    LLConnection *connection,
    const uint8_t *buf,
    size_t buf_len);
```

This function sends some arbitrary data to the serial port.

To do this, first the function creates a new `Frame` with an *I* command, and fills its information vector with the data. Afterwards it sends this frame by using the `send_frame` function.

This function in turn calls `write_frame`, which assembles the frame in a `ByteVector` into the correct structure and byte order, after performing byte stuffing on its information. It then writes the bytes in this vector to the serial port.

After sending the frame, a response of type *RR* with the next sequence bit is expected by calling the `expect_frame` function. If this function does not get a response within the set timeout, a POSIX timer that was armed in the `send_frame` function is triggered and sends the information frame again. This repeats until the response arrives, after which the timer is disarmed and the amount of bytes sent is returned, or until a set maximum amount of retries is reached, after which the timer is also disarmed and -1 is returned, signaling an error.

5.3 llread

```
ssize_t llread(
    LLConnection *connection,
    uint8_t *buf);
```

This function reads some arbitrary data from the serial port.

To do this, first the function expects to receive a command of type *I* with the next sequence bit by calling `expect_frame`, this function, which is explained in more detail above, automatically sends the correct *RR* response, disassembles the received frame and destuffs its information.

Afterwards the received data is copied to the provided buffer and its length is returned. If some part of this process is unsuccessful -1 is returned, signaling an error.

5.4 llclose

```
int llclose(
    LLConnection *connection,
    bool show_stats);
```

This function closes a connection to the serial port and destroys all of its state.

To do this, first a syncing protocol is employed, so both layers close at roughly the same time. In this protocol the transmitter sends a *DISC* command, then receives a *DISC* command, and then sends a *UA* response. The receiver does the same but reversed, receives a *DISC* command, then sends a *DISC* command, and then receives a *UA* response. This is implemented using a combination of the functions `send_frame` and `expect_frame`, which are explained in more detail above.

Afterwards `connection_destroy` is called, deallocating all resources related to the connection and resetting the serial port settings to their original state.

6 Application Layer Protocol

The application layer protocol's flow is as follows: open a connection to the other system, transmit the specified file and close the connection. It does so using the interfaces provided by the underlying Data Link Layer.

6.1 connect

```
LLConnection *connect(
    const char *serial_port,
    LLRole role)
```

This function is basically a wrapper around `llopen` that catches errors in the establishment of the connection.

6.2 init_transmission

```
int init_transmission(
    LLConnection *connection,
    const char *filename)
```

This functions is responsible for signaling the beginning of the file transmission process to the receiver. This also catches any errors that happened in the transmission of the *START* packet. First we `stat` the file to be transmitted in order to get its size. Then a *START* packet is constructed and sent to the receiver system. Finally we return 1 on success or -1 on failure.

6.3 receiver

```
ssize_t receiver(
    LLConnection *connection)
```

This function handles the *receiver* logic of the application layer.

It continuously receives packets using `llread`, which are handled depending on their type.

START packets are parsed to set the relevant file properties, namely the name and size, which let us open a file descriptor to which the data will be written.

DATA packets are parsed and their sequence number is verified, if they are not received in order, a major error is assumed to have happened. The data in these packets is then written to the opened file descriptor.

END packets simply end the loop and lead to the opened file descriptor being closed.

6.4 transmitter

```
ssize_t transmitter(
    LLConnection *connection,
    const char *filename)
```

This function handles the *transmitter* logic of the application layer.

Firstly, a connection is established using `init_transmission`, then the file is read in chunks, which are sent in *DATA* packets until the end of the file is reached, at which point an *END* packet is sent to signal the end of the transmission. The opened file is then closed.

7 Validation

The application was tested in several circumstances:

- Using different files: penguin.gif (10 968 B), neuron.jpg (31 802 B), test (9 594 624 B);
- With temporary disconnects;
- With interference;
- Using different baud rates: 9600 baud, 19 200 baud, 38 400 baud, 57 600 baud and 115 200 baud;
- Using different packet sizes: 16 B, 32 B, 64 B, 128 B, 256 B, 512 B, 1024 B, 2048 B and 4096 B;
- Using different cable lengths (simulated in software using the benchmark of 5 µs/km): 0 km, 1 km, 5 km, 10 km, 50 km, 100 km, 500 km, 1000 km, 5000 km and 10 000 km

All tests were successful. The received file's integrity was verified visually and by using `diff`.

8 Link Layer Efficiency

The data link layer efficiency was tested mainly by varying a few parameters, namely the connection capacity, the frame error ratio (simulated), the cable length (simulated), and the packet size. All the test results are in appendix **A.2**.

8.1 Varying the Connection Capacity

The connection capacity was varied by setting the relevant baud rate flag when setting up the serial port connection.

Varying the connection capacity resulted in an almost constant efficiency function (a slightly downwards slope linear function).

This makes sense from a theoretical standpoint, as the efficiency is calculated by dividing the real flow by the connection capacity, and the real flow is directly proportional to the connection capacity.

8.2 Varying the Frame Error Ratio

The frame error ratio was varied by simulating either an error in the BCC or BCC2 fields when receiving a frame, according to the desired ratio. As stated below, this methodology proved to be problematic.

Varying the frame error ratio resulted in a steep decline in efficiency as the FER increases. This is due to the fact that in our error simulation, an error in the frame header is equally as likely as an error in the frame body, but an error in the header leads to a much worse penalty. This lead to a large amount of retransmissions, each taking 4 s with the timeout parameters we used, significantly reducing the efficiency.

Better looking results could have been achieved by lowering the timeout parameter significantly, or by implementing a constant bit (or byte) error ratio instead, or by simulating errors in the body only, as these are much more likely for the packet size we used.

As so, these results cannot be considered valid, but they do confirm how bad errors in frame headers are for the link layer efficiency when compared to errors in the frame body. If the methodology had been better we would have expected to see an efficiency function similar to $S = 1 - FER$, instead of the erratic function we got.

8.3 Varying the Cable Length

The cable length was varied by simulating a propagation time associated to this cable length. A benchmark of 5 µs/km *was used for this approximation, and the propagation time associated with the physical cable was considered negligible. This propagation time was implemented by the use of a sleep function after each frame was received.*

Varying the cable length resulted in a small efficiency decrease over short distances, but a significant decrease over large distances (an inversely proportional function).

This makes sense from a theoretical standpoint, as the efficiency follows the function $S = \frac{1}{1-2a}$ where $a = \frac{t_{prop}}{t_f}$ and t_f is constant, while t_{prop} increases linearly with the cable length.

8.4 Varying the Packet Size

The packet size was varied directly.

Varying the packet size resulted in a significant increase in efficiency up to a certain size, when diminishing returns were reached. This makes sense, as the time spent assembling/verifying headers decreases as the overall frame size increases and the amount of frames decreases.

It is worth noting that the error ratio in these tests was zero, so there was no downside in increasing the frame size. In the real world there would be a higher amount of errors in larger frames, as-

suming a constant bit error ratio, so there would be a "sweet spot" where the error ratio cancels out the efficiency gains from increasing the frame size.

9 Conclusions

To sum up, the goal of the project was achieved without much trouble. In addition, it helped us better understand the contents lectured in the theoretical lessons.

The protocol was developed in compliance with the given specification and the code is well structured, making it more robust and resistent to changes in the future.

A Appendix

A.1 Code

In folder `code/`.

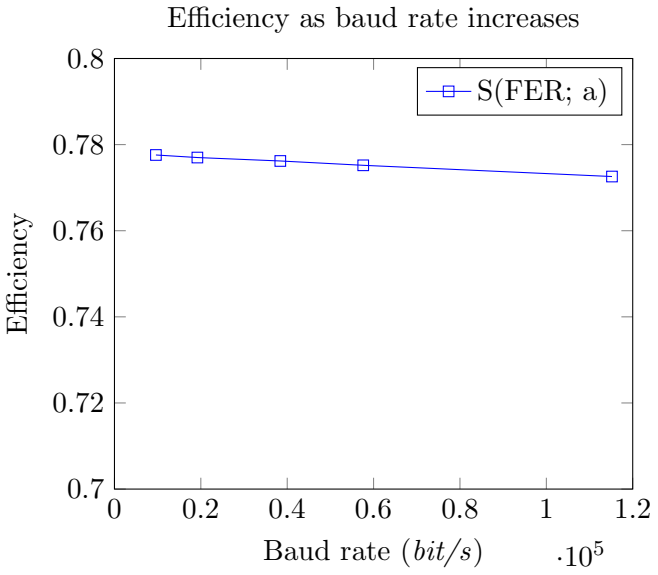
A.2 Figures

Baud rate *bit/s*	File size *bit*	Packet size *byte*
57,600	254,416	1,024

Table 1: Default values

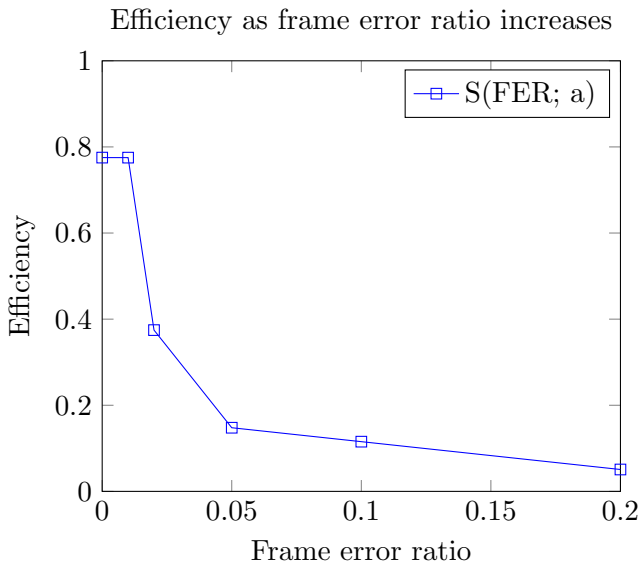
Baud rate *bit/s*	t_1 *s*	t_2 *s*	t_{mean} *s*	Flow *bit/s*	Efficiency
9,600	34.0828	34.0806	34.0817	7,464.885	0.7776
19,200	17.0532	17.0545	17.0539	14,918.3851	0.777
38,400	8.5352	8.5362	8.5357	29,806.0922	0.7762
57,600	5.6977	5.698	5.6978	44,651.2809	0.7752
115,200	2.8578	2.8593	2.8585	89,002.1708	0.7726

Table 2: Efficiency for a varied baud rate



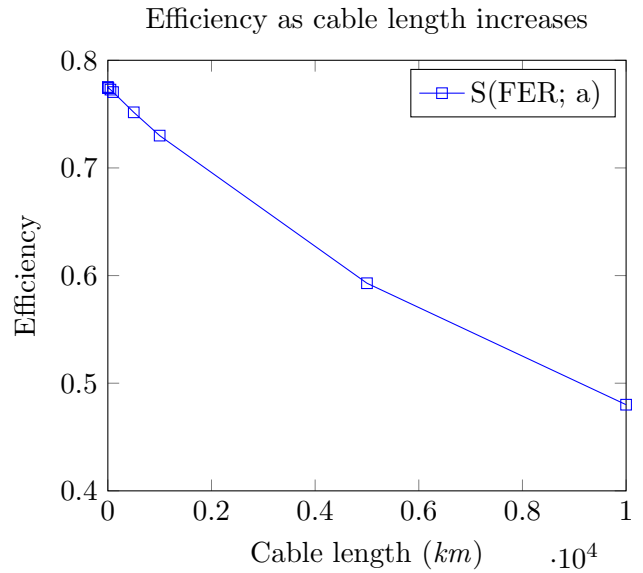
Frame error ratio	t_1 *s*	t_2 *s*	t_{mean} *s*	Flow *bit/s*	Efficiency
0	5.6977	5.698	5.6978	44,651.2809	0.7752
0.01	5.6995	5.6971	5.6983	44,647.4744	0.7751
0.02	9.8828	13.7015	11.7922	21,575.0158	0.3746
0.05	38.0662	21.6976	29.8819	8,514.0539	0.1478
0.1	22.0806	54.4303	38.2555	6,650.4469	0.1155
0.2	98.9961	74.8337	86.9149	2,927.184	0.0508

Table 3: Efficiency for a varied frame error ratio



Cable length km	t_1 s	t_2 s	t_{mean} s	Flow bit/s	Efficiency
0	5.6977	5.698	5.6978	44,651.2809	0.7752
1	5.7053	5.7009	5.7031	44,609.8146	0.7745
5	5.706	5.706	5.706	44,587.5421	0.7741
10	5.7045	5.7025	5.7035	44,606.9228	0.7744
50	5.7156	5.716	5.7158	44,511.0586	0.7728
100	5.7334	5.732	5.7327	44,379.7014	0.7705
500	5.8774	5.875	5.8762	43,295.8646	0.7517
1,000	6.052	6.0489	6.0504	42,049.1418	0.73
5,000	7.4483	7.4513	7.4498	34,150.7687	0.5929
10,000	9.2014	9.2009	9.2011	27,650.492	0.48

Table 4: Efficiency for a varied cable length



Packet size *byte*	t_1 *s*	t_2 *s*	t_{mean} *s*	Flow *bit/s*	Efficiency
16	14.043	14.0353	14.0391	18,121.9212	0.3146
32	9.8028	9.8026	9.8027	25,953.7463	0.4506
64	7.6958	7.6865	7.6911	33,079.2027	0.5743
128	6.6335	6.6286	6.6311	38,367.2109	0.6661
256	6.1016	6.0983	6.0999	41,707.9874	0.7241
512	5.8331	5.8349	5.834	43,608.9731	0.7571
1,024	5.6977	5.698	5.6978	44,651.2809	0.7752
2,048	5.6333	5.6303	5.6318	45,174.763	0.7843
4,096	5.5965	5.5988	5.5976	45,450.618	0.7891

Table 5: Efficiency for a varied packet size