U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Computer Networks – 1st Project Report

Luís Duarte - 202108734
Madalena Ye - 202108795

November 9, 2023

Abstract

This project was undertaken as part of the Computer Networks course and aims to develop a data connection protocol for file transmission via the RS-232 port. It provided us with an opportunity to practically implement and reinforce the theoretical knowledge acquired in class in a technical context. Furthermore, the protocol development process highlighted the crucial role of proper layer isolation. This not only simplified the project's structure and coding, but also enhanced its robustness, modularity, and reduced its error susceptibility.

1 Introduction

In this project, our main goal was to transmit a data file from the storage disk of one computer to another, utilizing a serial port. Our approach is based on two key elements: firstly, the development of a data link protocol in accordance with the given specifications, and secondly, the evaluation of this protocol via an application protocol, which should also conform to the specified guidelines. The end goal should be an application that can efficiently transfer files between two systems, with proper syncing and framing. It should also facilitate the establishment and termination of connections, in addition to providing adequate error checking and handling. This report seeks to explain the thought process on the development of this project. Its structure is as follows:

- Architecture – layers and interfaces.
- Code structure – APIs, main data structures, and functions.
- Main Use Cases – description of the project's operation and main sequences of function calls.
- Data Link Protocol – operation of the data link and implementation strategies/methods.
- Application Protocol – operation of the application layer and implementation strategies/methods.
- Validation – tests performed to evaluate the implementation.
- Data Link Protocol Efficiency – analysis of protocol efficiency on best/worst case scenarios.
- Conclusions – summary of the information presented and a reflection on the achieved learning objectives.

2 Architecture

2.1 Layers

Two distinct layers were created for the information transfer: Data Link Layer and Application Layer.

2.1.1 Data Link Layer

This is the lower level layer that directly interacts with the serial port driver. It's tasked with initiating and terminating connections, as well as guaranteeing error-free data transmission and reception.

2.1.2 Application Layer

This is the high level layer that communicates with both the Data Link Layer and the file system. It's responsible for reading a file with the information to send, unpack it, send it to the Data Link Layer and, on the other side, retrieve it from the Data Link Layer and reassemble it back into a complete file.

2.2 Program Execution

The program is run on two separate terminals, each on a different computer. One terminal operates the binary in transmitter mode, while the other functions in receiver mode. The program can be executed by calling **make_role**, where role is one of rx or tx, for receiving or transmitting a file, respectively;.

3 Code Structure

3.1 Data Link Layer

The Data Link Layer primarily revolves around link_layer.c. This file controls the flow of the Data Link Layer and contains the protocol that can be used by upper layers. On the other hand, the protocol.c file is composed of auxiliary functions for link_layer.c, in addition to being responsible for creating and manipulating frames.

3.1.1 link_layer.c

- ```
  typedef enum {
      LlTx,
      LlRx,
  } LinkLayerRole;
  ```

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

- ```
  typedef struct {
      char serialPort[50];
      LinkLayerRole role;
      int baudRate;
      int nRetransmissions;
      int timeout;
  } LinkLayer;
  ```

  Defines the link layer parameters.

- ```
  int llopen(LinkLayer
      connectionParameters)
  ```

  Opens the connection to the serial port.

- ```
  int llwrite(const unsigned char
      *buf, int bufSize)
  ```

  Sends data in a buf with bufSize.

- ```
  int llread(unsigned char *packet)
  ```

  Receive data in packet.

- ```
  int llclose(int showStatistics)
  ```

  Close previously opened connection. If showStatistics is true, link layer should print statistics in the console on close.

3.1.2 protocol.c

- ```
  typedef enum {
      PACKET_BEGIN = 0,
      ADDR,
      CONTROL,
      BCC,
      DATA,
      BCC2,
      PACKET_END,
      SUCCESS
  } packet_status_t;
  ```

  Possible states of a frame during its processing.

- ```
  typedef struct packet_t {
      unsigned char addr;
      unsigned char control;
      unsigned char bcc;
      unsigned char * data;
      unsigned char bcc2;
      int data_size;
      int alloc_size;
      packet_status_t status;
  } packet_t;
  ```

  Contents of a frame.

- ```
  int write_command(int fd, bool
      tx, char command)
  ```

  Responsible for sending supervision and unnumbered frames

- ```
  int write_data(int fd, unsigned
      char * buf, int size)
  ```

  Responsible for sending information frames

- ```
  void read_packet(int fd,
      packet_t * packet, bool tx)
  ```
  ```

Reads a frame from the serial port and stores the frame data in a packet_t struct, handling each byte of the frame according to its current state.

- `unsigned char * stuff_packet(unsigned char * buf, int * size)`

  Byte stuffing technique.

3.2 Application Layer

The Application Layer is centered around the application_layer.c file. This file encompasses the client-facing interface of the application, handling all the higher-level setup and communication between the transmitter and receiver. Supporting the application_layer.c, the application_protocol.c file contains code that assists in the functioning of the application_layer.c.

3.2.1 application_layer.c

- `void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename);`

  Entry point for the application layer.

- `void mainTx(const char * filename)`

  Executes the transmitter's operational flow.

- `void mainRx(const char * filename)`

  Executes the receiver's operational flow.

3.2.2 application_protocol.c

- ```
  typedef struct control_parameter_t{
  unsigned char type;
  unsigned char parameter_size;
  unsigned char parameter[256];

  } control_parameter_t;
  ```

  Defines the control packet parameters.

- ```
  typedef struct control_packet_t{
      unsigned char packet_type;
      control_parameter_t * parameters;
      int length;

  } control_packet_t;
  ```

  Defines the data needed to represent a control packet.

- ```
  typedef struct data_packet_t{
      unsigned short data_size;
      unsigned char * data;

  } data_packet_t;
  ```

  Defines the elements needed to represent a data packet.

4 Main Use Cases

There are two primary use cases in this scenario: receiving a file and sending a file. The function call sequences for both of these use cases are described below.

4.1 Sender

1. When sending a file, the application layer first tries to open a descriptor to the specified file and initiates the transmission process through llopen().
2. llopen(): This is used for the handshake between the transmitter and receiver, achieved through the exchange of control frames (the transmitter sends a SET and waits for the receiver to respond/acknowledge).
3. Next, control packets containing the file size and filename information are created through the mainTx() function.
4. The transmitter continuously reads chunks of data from the file and sends data packets using the llwrite() function until the entire file is transmitted. Throughout this process, the transmitter adds a header, BCCs for error detection, start and end flags, and performs packet stuffing to ensure data integrity during transmission.
5. After successfully sending the information, an end packet is sent, and llclose() is called to end the connection.

4.2 Receiver

1. In the case of the receiver, when the llopen() function is called, the receiver enters a
  ```

waiting state until it receives a SET command from the transmitter. Once the SET command is received, the receiver sends a UA (Unnumbered Acknowledgement) in response.

2. When receiving the file (llread()), the application layer reads a packet and processes the data packets by writing them to a file after destuffing them. The first step in the processing is identifying the packet type:

   - (1) Data Packet: the application layer retrieves the fragment size to write the received data fragment to the file.
   - (2) Start Packet: the packet is deconstructed to extract the transmitted file's size and name.
   - (3) End Packet: the application layer exits the receiver loop to close connections and perform cleanup.

3. this function receives data and control packets using llread(), processes the data packets by writing them to a file, and handles control packets to extract file size and filename information, and finally closes the file.

5 Link Layer Protocol

The main flow of the Data Link Layer involves opening the connection using llopen, writing and reading frames of information using llwrite and llread, and closing the connection using llclose. In this process, we utilize the Stop-and-Wait protocol for establishing and terminating the connection, as well as for sending supervision and information frames.

5.1 llopen

The establishment of the connection is performed by the llopen function. Initially, after opening and configuring the serial port, the transmitter sends a SET supervision frame and waits for the receiver to respond with a UA supervision frame. When the receiver receives the SET frame, it responds with UA. If the transmitter receives the UA frame, the connection has been successfully established. After establishing the connection, the transmitter starts sending information that will be read by the receiver. If a response is not received, the command is retransmitted a configurable number of times, with a configurable amount of time between each transmission.

5.2 llwrite

The transmission of information is performed by the llwrite function. This function takes a control or data packet and applies byte stuffing to it, in order to avoid conflicts with data bytes that are the same as the frame flags or the bcc. Afterwards, the packet is transformed into an information frame (framing), sent to the receiver, and the sender waits for a response. If the frame is rejected, the transmission is repeated until it is accepted or the maximum number of attempts is exceeded. Each transmission attempt has a time limit, after which a timeout occurs. If the receiver's response is equal to RR(inv(s)), where s is the current sequence number, it indicates that the receiver is ready now to receive the inverse of s. Conversely, if the response is REJ(s), it means that the frame was rejected, and the sender attempts to resend the same packet while resetting the timeouts.

- **Byte stuffing**

```
for(int i = 0; i < old_size;
    i++, nbuffer_pos++){
     if(buf[i] == FLAG || buf[i]
        == ESCAPE_CHAR){
         new_size++;
         nbuffer =
             realloc(nbuffer,
             new_size);
         nbuffer[nbuffer_pos] =
             ESCAPE_CHAR;
         nbuffer_pos++;
         nbuffer[nbuffer_pos] =
             buf[i] == FLAG ?
             ESCAPE_FLAG :
             ESCAPE_ESCAPE;
      } else
         {nbuffer[nbuffer_pos] =
         buf[i];}
}
```

5.3 llread

The reading of information is performed by the llread function. This function reads the information received from the serial port and verifies its validity. Initially, it performs destuffing of the data field of the frame and validates the BCC1 and BCC2, which check for any errors that may have occurred during transmission. The next step is to copy the received data to the provided

buffer, and the length of the data is returned by the function. If any part of this process is unsuccessful, -1 is returned to signal an error.

5.4 llclose

The termination of the connection is performed by the llclose function. This function is invoked by the sender when the number of failed attempts is exceeded or when the transfer of data packets is completed. The sender sends a DISC supervision frame and waits for the receiver to respond with the same frame, signaling the end of its operation. When the sender receives DISC again, it responds with UA and terminates the connection.

6 Application Layer Protocol

The application layer interacts with the file and user, allowing for configuration of transfer parameters. The flow of the application layer protocol is as follows: establish a connection with the other computer, transmit the specified file, and then close the connection. This is achieved by utilizing the interfaces provided by the underlying Data Link Layer.

6.1 mainTx

This function handles the transmitter logic of the application layer. Firstly, after a connection is established, the metadata (filename and filesize) is read and sent through a control packet, announcing the start of the transmission. After that, 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.

6.2 mainRx

This function is responsible for managing the receiver's logic at the application layer. It continuously receives packets using the llread function from the Data Link Layer and handles them based on their type. When a start packet is received, it's parsed to extract the metadata of the file. This information allows us to open a file descriptor to which the data will be written. For data packets, they are parsed, and their sequence number is checked. If the packets are not received in the expected order, it's assumed that a major error has occurred. The data contained in these packets is then written to the previously opened file descriptor. When an END packet is received, the loop is terminated, and the file descriptor is closed.

7 Validation

The application was tested in several circumstances:

- Using penguin.gif (10 968 B) and a 100 kB text file;
- 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, 256 B, 512 B, 1000 B, 2000 B and 4000 B;
- Using different cable lengths (simulated in software using the benchmark of 5 µs/km): 1km, 10km, 100km, 1000km, 5000km, 7500km, 10000km, and of course, the diameter of the Earth.

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

8 Link Layer Efficiency

We can define link layer efficiency, S, as:

$$S = \frac{Baudrate_r}{Baudrate_t}$$

In our case, we are measuring the number of bytes sent by the application layer on *llwrite* calls so, we can expand this formula further to give us the direct comparison between the data transfer speed and the theoretical baud rate:

$$S = \frac{Speed_{bytes} * \frac{Bits_{data} + Bits_{stop}}{Bits_{data}} * 8}{Baudrate_t}$$

In this case, we always use 8N1[1] for the serial connection.

[1] 8 data bits and 1 stop bit.

8.1 Methodology

In order to measure the data transfer speed of bytes sent, we do the following steps:

1. Take a timestamp when the connection is established between the transmitter and receiver;
2. Count the bytes sent by the application layer, excluding possible re-transmissions;
3. Take a second timestamp when connection is closed.

With these values we can successfully measure the average data transfer speed of the connection.

In order to reduce error in our measurements we took them 5-10 times and average them out. This is not a perfect method because we also should measure in different computers and different cables to further reduce possible measurement error.

8.2 Relationship between efficiency and Baud rate

We do this by manually varying the serial baud rate on the setup of the serial connection by changing the appropriate flag with the tcsetattr function.

As shown in **Fig. 1**, we can see somewhat a downwards trend in efficiency when baudrate increases, with the exception of one outlier on the 9600 baud. This downward trend is excepted and is due to the increase in error when sending a byte, causing re-transmissions on the physical layer.

8.3 Relationship between efficiency and FER

We do this by introducing a chance on the receiver side (only on information frames), that will reject the frame after it's been completely received and validated. Since we are using a RNG[2]*, we need a larger file to ensure that we can eliminate any statistical variance on the FER*[3]*. All measurements have been done at 115200 baud.*

As shown in **Fig. 2**, we can see an expected linear downwards trend in efficiency when *FER* increases. It's remarkable to see that even at 50% error rate, we still get a somewhat usable connection, and we are able to send data reliably at cost of time and data transfer speed.

[2]Random Number Generator

[3]Frame Error Rate

8.4 Relationship between efficiency and Packet Size

To measure this relationship, we change the number of bytes read each time from the source file by the application layer and sent by the Data Link Layer connection. All measurements have been done at 19200.

As we can see in **Fig. 3**, we note that the plot represents a logarithmic function that stabilizes at around 256 bytes (note that in **Fig. 3** the plot is in logarithmic form on the x axis, to improve readability). This measurement is particularly useful because, we want to go as low as possible on the packet size, without losing performance, so that, the validation of the data is done more times and help to avoid *BCC* collisions and therefore further help with the reliability of the implemented protocol.

8.5 Relation between efficiency and propagation time

We do this by adding a delay of 5 µs *per* km *after each packet is successfully read, even if it's rejected afterwards. All measurements have been done at 19200.* Observing **Fig. 4**, it's evident that the efficiency remains steady until approximately 10^4, at which point there's a sudden decrease to 74%. This is anticipated given the use of a Stop-and-Wait protocol, which introduces a delay of about 10 ms per packet (in this order of magnitude and assuming no FER), which is a delay that can be easily measured by our methods.

9 Conclusions

It has been shown that all the project's goals have been achieved. The protocol was developed in compliance with the given specifications and is working as intended. Furthermore, the test results corroborate and enhance the knowledge gained from the lectures. Through this project, we are able to internalize the concepts of byte stuffing, framing, and the operation of Stop-and-Wait protocol, along with its error detection and handling mechanisms. Thereupon, the development of this project as a whole has proved to be a hands-on, practical learning experience.

A Appendix

A.1 Code

In folder `code/`.

A.2 Figures

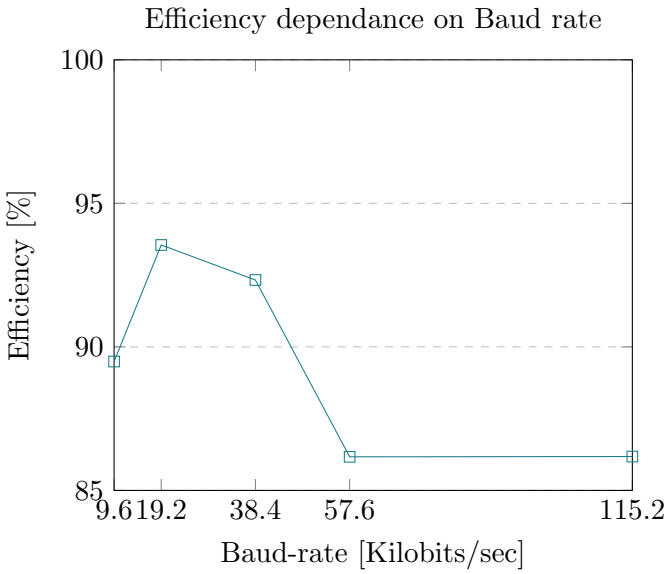

Fig. 1. Plot that shows the relationship between efficiency of Link Layer and Baud rate

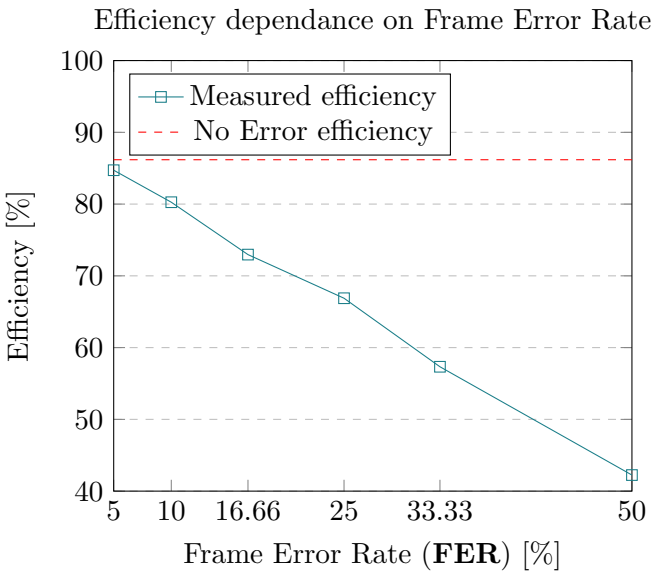

Fig. 2. Plot that shows the relationship between efficiency of Link Layer and FER

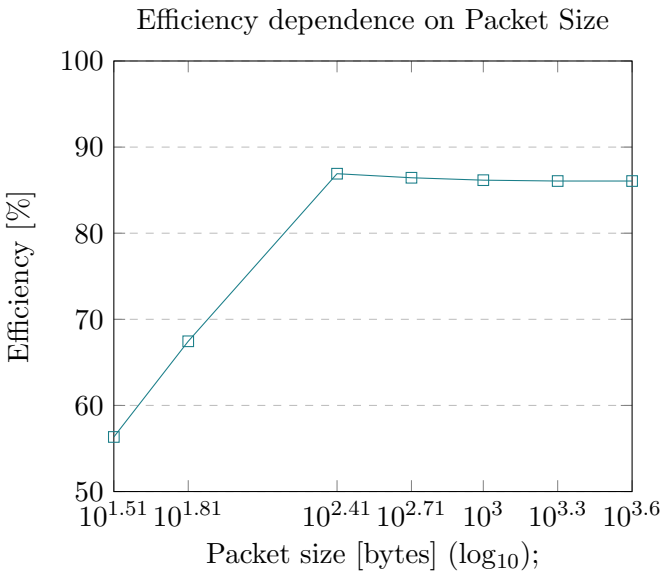

Fig. 3. Plot that shows the relationship between efficiency of Link Layer and Packet Size

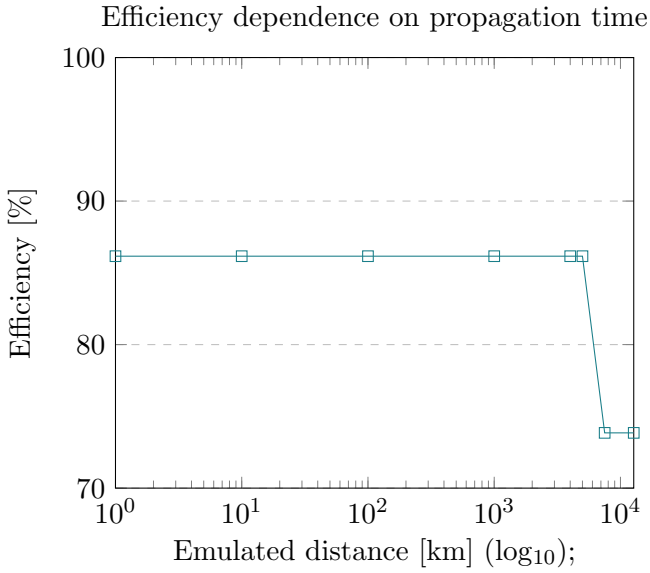

Fig. 4. Plot that shows the relationship between efficiency of Link Layer and propagation time