Ligação de Dados

Redes de Computadores

Bruno Carvalho
up201606517

João Malheiro
up201605926

Carlos Daniel Gomes
up201603404

November 12, 2018

Contents

Summary

This project was developed for to the curricular course RCOM (Computer Networks). It was finished successfully by implementing all the required functionalities, passing all the tests, and building a fully working application, with more options, functionalities and high robustness to errors.

The main conclusion to take from this project is the convenience of implementing a communication protocol with a layered architecture, which allows for technical implementation details, hardware configuration and transmission errors to be abstracted from practical applications.

Introduction

This program's goal is to transfer files from one computer to another through a serial port connection, using a *Step and Wait* protocol made with two layers: the *Link Layer*, immediately above the physical layer, whose function is to detect and correct transmission errors and regulate data flow; and the *Application Layer*, handling packets transmission through the link layer. The protocol to implement has a specification described in detail, which we will not reiterate here.

To build the program simply call `make` in the main directory. Afterwards, call `./ll --help`. Some use cases can be found in subsection 4.2.

Quick rundown

Architecture	Architectural aspects, bottom up
Use cases and control flow	Top down function call tree, example program invocations
Link Layer Protocol	Link layer requirements, subunits and sample code
Application Layer Protocol	App layer requirements and sample code
Validation	Description of tests performed
Protocol Efficiency	Program practical vs theoretical efficiency analysis

Concepts

We'll use the following labels a few times throughout the report:

LL	The Link Layer
AL	The Application Layer
R	The Receiver (in a program call)
T	The Transmitter (in a program call)
TLV	A tuple (`Type`,`Length`,`Value`) used in control packets of the app layer to store information, namely filesize and filename

Architecture

In this section we will traverse *bottom-up* the program's architectural aspects, including the internal and auxiliary functions of each layer, their interfaces and the primary data structures used. At the end we list some of the auxiliary features such as terminal setup, program options, signal handlers and execution timing.

String

Given that the byte arrays used throughout the program are *not* `NULL`-terminated strings — but rather of variable length and dynamically allocated — they must be *accompanied* by their size anytime they are passed between functions. The *`string`* structure is a simple wrapper around a *`char*`* and a *`size_t`*.

Link Layer

The link layer contains one fundamental, yet simple data structure: *`frame`*. It holds information both for the frame's header (A and C fields) and the frame's data field, which is a *`string`*.

Functions

The core has the following functionalities and respective functions:

- Byte stuffing and destuffing: `stuffData`, `destuffData`, `destuffText`
- Convert a *`frame`* to a *`string`* and write it to the device: `buildText`, `writeFrame`
- Read text from the device and convert it to a *`frame`*: `readText`, `readFrame`
- Introduce flip bits in read frames with a certain probability: `introduceErrors`

On top of these, we have simple utility functions used by the interface:

- Inquire frames: `is*frame` — `isIframe`, `isSETframe`, ...
- Write frames: `write*frame` — `writeIframe`, `writeSETframe`, ...

And the interface follows the specification:

- Establish a connection through an open device: `llopen`
- Terminate a connection through an open device: `llclose`
- Write a message (frame): `llwrite`
- Read a message (frame): `llread`

It should be noted that `llopen` and `llclose` *do not* open or close the device, nor do they modify any terminal settings.

App Layer

The app-layer makes public three data structures: *`tlv`*, *`control_packet`* and *`data_packet`*. *`control_packet`* holds an array of *`tlv`*s, which are the TLV tuples described in the introduction. The *`data_packet`* structure holds the data of both the packet's header and data field, which is a *`string`*.

Functions

The app-layer core has the following internal functions:

- Convert integers and strings to *`tlv`*s: `build_tlv_str`, `build_tlv_uint`
- Convert a generic array of *`tlv`*s to a *`control_packet`*: `build_control_packet`
- Convert a *`string`* to a *`data_packet`*: `build_data_packet`
- Extract any *`tlv`* value from a *`control_packet`*: `get_tlv`
- Inquire packets: `isDATApacket`, `isSTARTpacket`, `isENDpacket`

The interface provided to the application user includes:

- Send packets using `llwrite`: `send_data_packet`, `send_start_packet`, `send_end_packet`
- Receive (any) packet using `llread`: `receive_packet`
- Extract filename and filesize from a *`control_packet`*: `get_tlv_filesize`, `get_tlv_filename`

Parallel features

- Open chosen device and set new terminal settings: `setup_link_layer`
- Close chosen device and set old terminal settings: `reset_link_layer`
- Alarm utilities for write timeouts: `set_alarm`, `unset_alarm`, `was_alarmed`
- Execution timing: `begin_timing`, `end_timing`, `await_timeout`
- Compute and print statistics about error probabilities, speed and efficiency: `print_stats`
- File I/O entry functions: `receive_file`, `send_file`, `receive_files`, `send_files`

Use cases and control flow

Top level

Setup in function `main` includes parsing and validating program options with `parse_args`, setting up signal handlers with `set_signal_handlers`, testing the system's alarms with `test_alarm`, and adjusting the terminal configuration with `setup_link_layer`. This is identical for R and for T.

In function `send_file`, called by T, the selected file is opened, its size is calculated and it is then read into a single *buffer* and closed. The *buffer* is then split into multiple *packets* with length *`packetsize`* — each stored in a *`string`* — and then freed. These packets are then sent to R in the communications phase, each in its own data packet — see Figure 1a.

Function `receive_file`, called by R, enters the communications phase immediately — see Figure 1b. Once the transmission is completed, it has received a filename and several packet strings. The output file is created with that filename, the packets are written successively, and then it is closed.

Neither of these functions deal with any LL or internal AL errors. The only errors that may surface are timeout and write retries being capped due to too many errors (see options `--time`, `--answer` and `--timeout`). At the end of `main`, the terminal settings are reset with `reset_link_layer`.

Calling the program

- Transmitter:
  `./ll -t [option...]  files...`
- Receiver:
  `./ll -r [option...]  number_of_files`

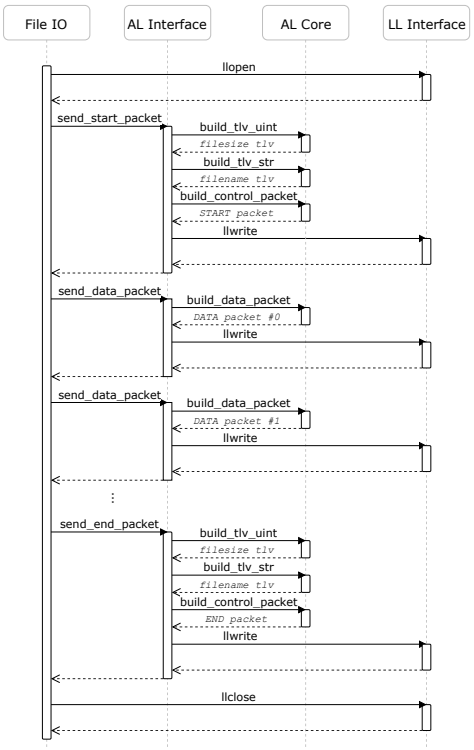
The program supports the usual `--help`, `--usage` and `--version` options. The first two will present all the other possible options and exit.

The program is usually called with just one file, so the only positional argument for T is the name of the file — which will be used as output filename for R. The only position argument for R is the number of files to be received, usually 1. Multiple files are sent in succession, one after another.

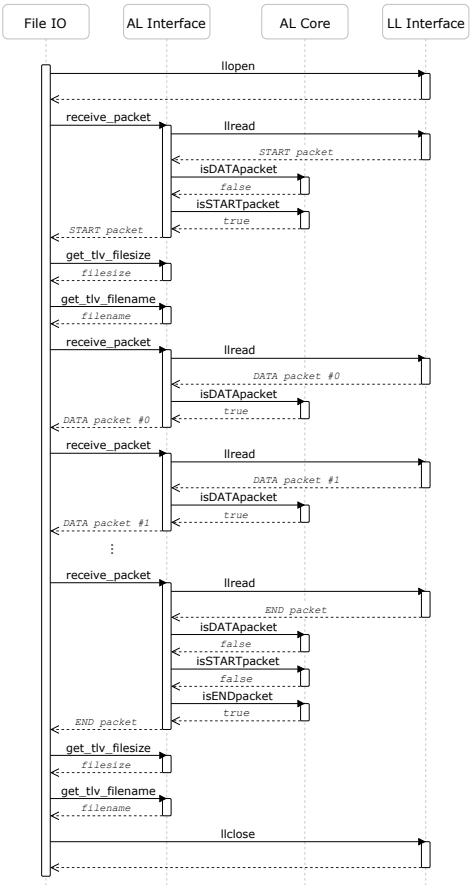
Some example calls:

- Send `penguin.png`. In R, introduce errors both in frames' headers and data fields:
  `./ll -t --stats penguin.png`
  `./ll -r --stats -h 0.4 -f 0.6 1`
- Calls made for the statistics in Figure 2, with `ERRORP` $\in \{0, 0.2, \ldots, 0.97\}$:
  `./ll -t -s 2048 -b 38400 --compact --timeout=5 --answer=3000 ~/img/landscape.jpg`
  `./ll -r -s 2048 -b 38400 --compact --timeout=5 --answer=3000 1 -f ERRORP`

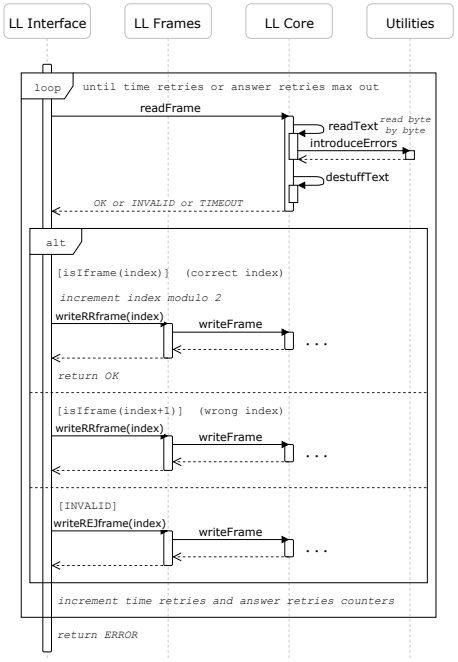
Figure 1: Sequence diagrams of some important functions



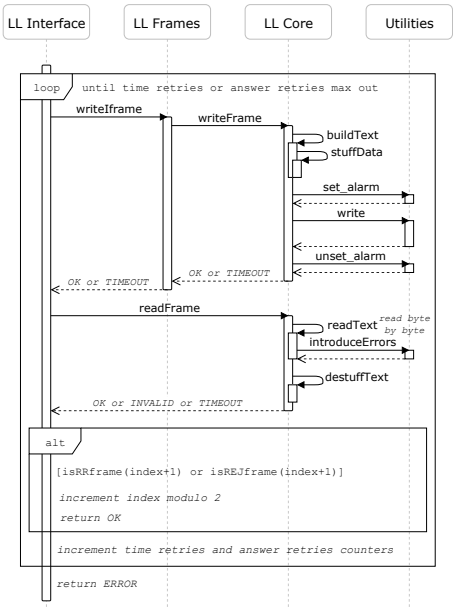
(a) File communications as seen through the app layer (part of function `send_file`)



(b) File communications as seen through the app layer (part of function `receive_file`)



(c) Function `llread`



(d) Function `llwrite`

Link Layer Protocol

In the *LL* protocol we recognize these requirements:

(a) Canonical, state machine based reading loop of variable length byte arrays;
(b) Timeout in long reads and writes;
(c) Conversions between byte arrays and a generic frame data structure;
(d) Stuffing and destuffing byte arrays, representing valid or invalid generic frames;
(e) Error detection and reporting in read frames;
(f) Probability based error introduction in read frames;
(g) Identification and writing of protocol-defined frames.

Our *LL* implementation consists of four units: *core*, *errors*, *frames* and *interface*.

Core The lowest unit of the entire program, it handles the first five requirements. Internally it uses mostly ***`string`*** instead of ***`frame`***, both for reading and writing. It exposes only `writeFrame` and `readFrame`, which have ***`frame`*** arguments. Function `writeFrame` calls `buildText` to transform the given ***`frame`*** into a string, which is *stuffed* by `stuffData` before being written. Function `readFrame` calls `readText`, the canonical read loop, and destuffs the string read with `destuffText`, while validating it and reporting on any detected errors.

This unit supports the entire *LL* by providing a generic read/write facility for frames of any kind — supporting all valid frame headers and all frame lengths — by handling only byte stuffing, error detection, and reading/writing timeouts.

Errors A simple unit whose purpose is to intentionally introduce errors (bit flips), with a certain probability, in both the header and data fields of frames. Entered at the end of function `readText`.

Frames For each frame in the specification – ***`I`***, ***`SET`***, ***`DISC`***, ***`UA`***, ***`RR`***, and ***`REJ`*** – this unit exposes a function which identifies it, `is*frame`, and another which writes it to a given file, `write*frame`.

Interface Includes specified functions `llopen`, `llwrite`, `llread` and `llclose`. These functions use only the facilities provided by the *frames* unit. `llopen` is used to establish a connection between *R* and *T* by ensuring both ends are in sync; `llclose` is used to end it. These functions have different *versions* for *R* and *T*. While the connection is active, `llread` and `llwrite` are used to read and write from the connection respectively.

```c
static int readText(int fd, string* textp) {
    string text;

    text.len = 0;
    text.s = malloc(8 * sizeof(char));

    size_t reserved = 8;
    FrameReadState state = READ_PRE_FRAME;
    int timed = 0;

    while (state != READ_END_FLAG) {
        char readbuf[2];
        ssize_t s = read(fd, readbuf, 1);
        char c = readbuf[0];

        // [...] Errors and text.s realloc

        switch (state) {
        case READ_PRE_FRAME:
            if (c == FRAME_FLAG) {
                state = READ_START_FLAG;
                text.s[text.len++] = FRAME_FLAG;
            }
            break;
        case READ_START_FLAG:
            if (c != FRAME_FLAG) {
                if (FRAME_VALID_A(c)) {
                    state = READ_WITHIN_FRAME;
                    text.s[text.len++] = c;
                } else {
                    state = READ_PRE_FRAME;
                    text.len = 0;
                }
            }
            break;
        case READ_WITHIN_FRAME:
            if (c == FRAME_FLAG) {
                state = READ_END_FLAG;
                text.s[text.len++] = FRAME_FLAG;
            } else {
                text.s[text.len++] = c;
            }
            break;
        default:
            break;
        }
    }

    text.s[text.len] = '\0';

    introduceErrors(text);

    *textp = text;
    return 0;
}


int writeFrame(int fd, frame f) {
    string text;
    buildText(f, &text);

    set_alarm();
    errno = 0;
    ssize_t s = write(fd, text.s, text.len);

    int err = errno;
    bool b = was_alarmed();
    unset_alarm();

    free(text.s);

    if (b || err == EINTR) {
        // [...] Report error
        return FRAME_WRITE_TIMEOUT;
    } else {
        return FRAME_WRITE_OK;
    }
}
```

Application Layer Protocol

In the *AL* protocol we recognize these requirements:

(a) Representation of generic control packets and data packets;
(b) Construction of a control packet from a list of values;
(c) Construction of a data packet from a string;
(d) Identification, parsin g and writing of protocol-defined packets;
(e) Extraction of *tlv* values from control packets, namely filesize and filename;
(f) Error detection and reporting of mis-indexed *DATA* packets or bad packets.

Our *AL* implementation, unlike the *LL* implementation, is not further subdivided. Each of these requirements is satisfied by a set of specialized functions, and the interface is essentially `send_data_packet`, `send_start_packet`, `send_end_packet` and `receive_packet`.

The first function, `send_data_packet`, takes a *string*, prepends it with a packet header using `build_data_packet`, and writes it using `llwrite`. The packet index is kept in an internal counter *out_packet_index*. The other functions `send_start_packet` and `send_end_packet` first build two *tlv* for the filesize and filename using `build_tlv_*`, then build the control packet string using `build_control_packet`, and finally write it using `llwrite`.

The `receive_packet` function reads an arbitrary packet using `llread`, and then uses `isDATApacket`, `isSTARTpacket` or `isENDpacket` to identify *and* parse said packet. The packet index is also kept in an internal counter *in_packet_index*.

```
int send_start_packet(int fd, size_t filesize,
    char* filename) {
    int s;
    string tlvs[2];

    out_packet_index = 0;

    s = build_tlv_uint(PCONTROL_TYPE_FILESIZE,
     filesize, tlvs + FILESIZE_TLV_N);
    if (s != 0) return s;

    s = build_tlv_str(PCONTROL_TYPE_FILENAME,
     string_from(filename), tlvs + FILENAME_TLV_N);
    if (s != 0) return s;

    string start_packet;
    s = build_control_packet(PCONTROL_START,
     tlvs, 2, &start_packet);
    if (s != 0) return s;

    free(tlvs[0].s);
    free(tlvs[1].s);

    // [...] Report app write

    s = llwrite(fd, start_packet);
    free(start_packet.s);
    return s;
}
```

```
int receive_packet(int fd, data_packet* datap,
      control_packet* controlp) {
    int s;

    string packet;
    s = llread(fd, &packet);
    if (s != 0) return s;

    data_packet data;
    control_packet control;

    if (isDATApacket(packet, &data)) {
        ++in_packet_index;
        *datap = data;

        free(packet.s);
        return PRECEIVE_DATA;
    }

    if (isSTARTpacket(packet, &control)) {
        in_packet_index = 0;
        *controlp = control;

        free(packet.s);
        return PRECEIVE_START;
    }

    if (isENDpacket(packet, &control)) {
        in_packet_index = 0;
        *controlp = control;

        free(packet.s);
        return PRECEIVE_END;
    }

    printf("[APP] Error: Received BAD packet\n");
    free(packet.s);
    return PRECEIVE_BAD_PACKET;
}
```

Validation

Multiple successive tests were made in order to determine the robustness of the program:

(a) Send a file without errors;
(b) Introduce errors in the serial port with pins (RCOM lab);
(c) Introduce errors (flip bits) internally, in frames' headers and/or data fields;
(d) Turn off and on the connection in the serial port repeatedly, halting communications;
(e) All the previous attempt of failures together.

The program is robust to these errors, and only (b) can potently corrupt a frame in an undetectable manner — due to the weaknesses in the protocol — resulting in a corrupted output file. For point (c), errors were introduced in control frames's headers as well, corrupting frames *SET*, *DISC* and *UA* used in functions `llopen` and `llclose`. These succeed regardless, although possibly only after several timeouts. See options `-f`, `-h`, `--time`, `--timeout`.

Protocol Efficiency

Errors in frame data field: Noisy transmission

To approach this topic, analyze the following chart, Figure 2.

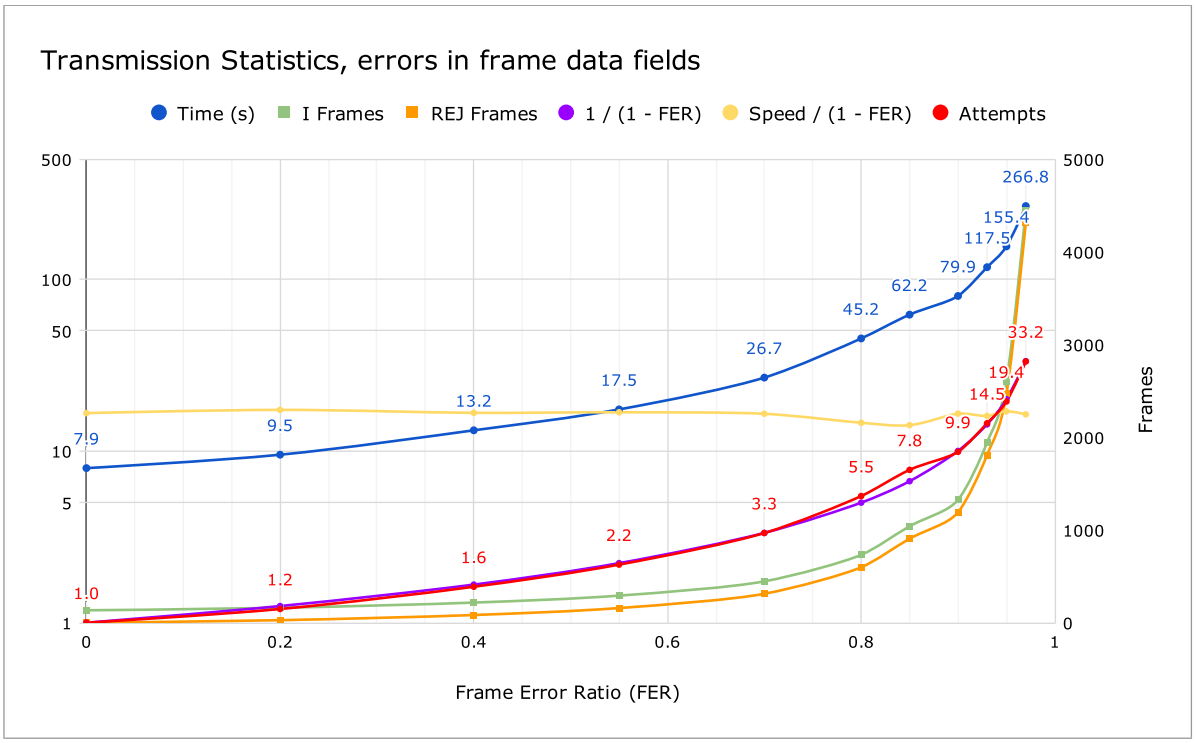


Figure 2: Transmission statistics of a 268.849B file, using packetsize 2048 and baudrate 38400 bytes/s. Error probability samples (`-f`): `FER` $= p \in \{0, 0.2, 0.4, 0.55, 0.7, 0.8, 0.85, 0.9, 0.93, 0.95, 0.97\}$.

First of all, with the specified parameters the file is split into 132 packets, with average packet size 2037 and *I* frame size 2047 bytes. Since there are no frame header errors, a total of 134 packets is transmitted, and T receives exactly 134 *RR* acknowledgments.

The green and orange graphs count the number of transmitted *I* and *REJ* frames respectively (right vertical axis). At $p = 0.93$, these numbers round 2000. Since the total number of packets is constant, the orange curve is just a translation of the first.

Attempts is the graph of $\frac{\#I}{\#RR} = \frac{\#I}{\#I - \#REJ}$. This is the average number of times T has to write a frame *I* to the device before it is acknowledged by R and received without errors. We can see it follows the graph of $1/(1 - \texttt{FER})$, a consequence of the relation

$$\mathbb{E}[Attempts] = \sum_k k \cdot \mathbb{P}[Attempts = k] = \sum_k k \cdot p^{k-1}(1-p) = \frac{1}{1-p}.$$

Furthermore, *Speed* is the data packet rate, i.e. the number of packets transmitted per second. Its formula is $\frac{filesize}{2037 \cdot s}$ — or in this case simply $\frac{132}{Time}$. The yellow graph traces $Speed/(1 - \texttt{FER})$, which is approximately constant — around 16.2. Indeed, *Speed* is just a scaled measure of *Efficiency*, which in our case is theoretically

$$S = \frac{T_f}{\mathbb{E}[A] \cdot (T_f + 2T_{prop})} = \frac{1}{\mathbb{E}[A] \cdot (1 + 2a)} = \frac{1-p}{1+2a} \overset{a=0}{\simeq} 1 - p.$$

Finally, with this analysis we can conclude in multiple ways — the simplest of which just reading the chart — that the total transmission time is inversely proportional to $\frac{1}{1-p}$, i.e. linear in $1-p$.

Errors in frame header

For completion, we leave here a chart similar to the previous one, with the same parameters expect replacing `-f` with `-h`, i.e. introducing the errors in the frames' headers instead of data fields.

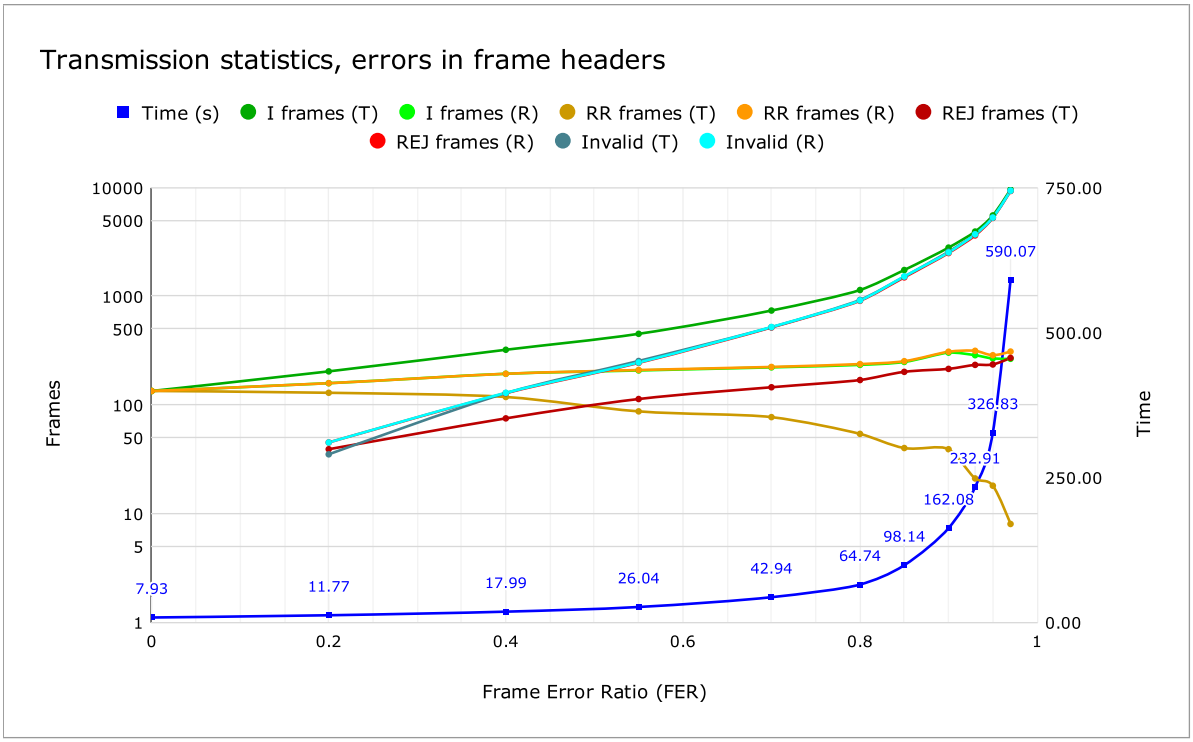


Figure 3: Transmission statistics of a 268.849B file, using packetsize 2048 and baudrate 38400 bytes/s. Error probability samples (`-h`): `FER` $= p \in \{0, 0.2, 0.4, 0.55, 0.7, 0.8, 0.85, 0.9, 0.93, 0.95, 0.97\}$.

Conclusion

We implemented all of the specified requirements, and built an implementation of the protocol with multiple configurations, performance analysis and robustness.

The most challenging part in the development phase was implementing `llopen` and `llclose` in such a way they would succeed even with high error probabilities in frame headers (`-h`).