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`

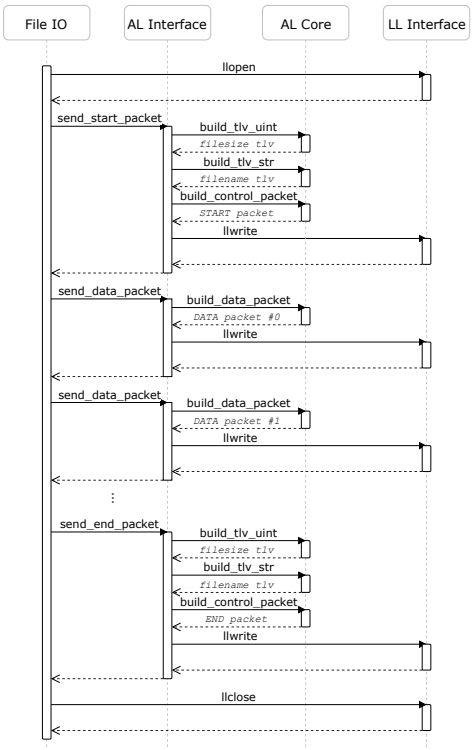
The program supports the usual `--help`, `--usage` and `--version` options. The first two will present all the other possible options and exit.

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.

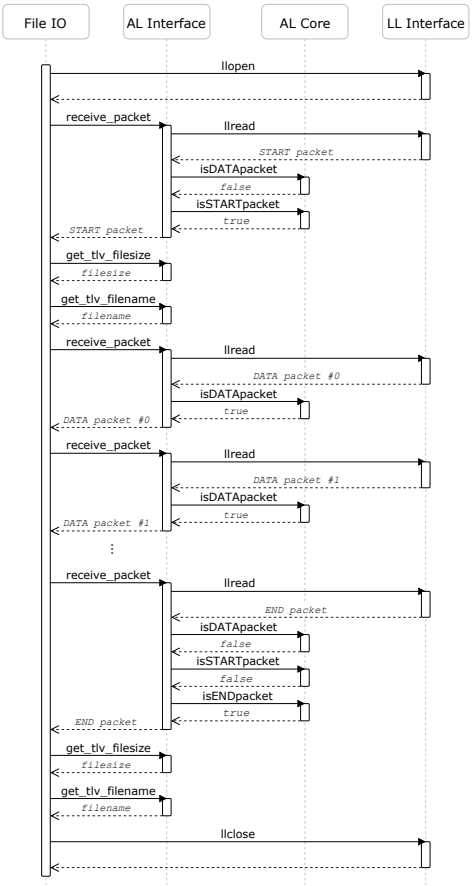
Some example calls:

- 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`

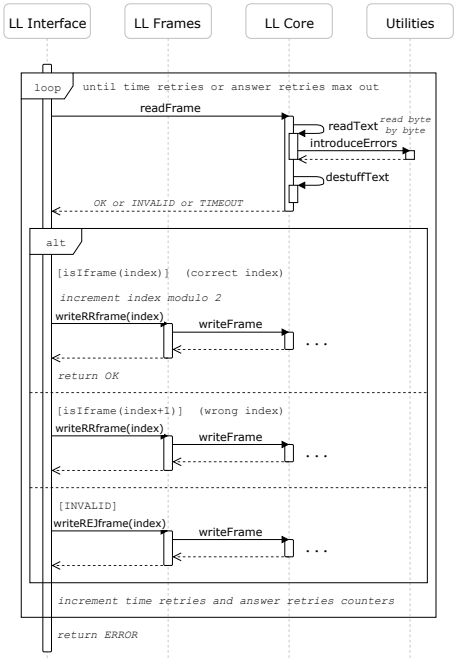
Figure 1: Sequence diagrams of some important functions



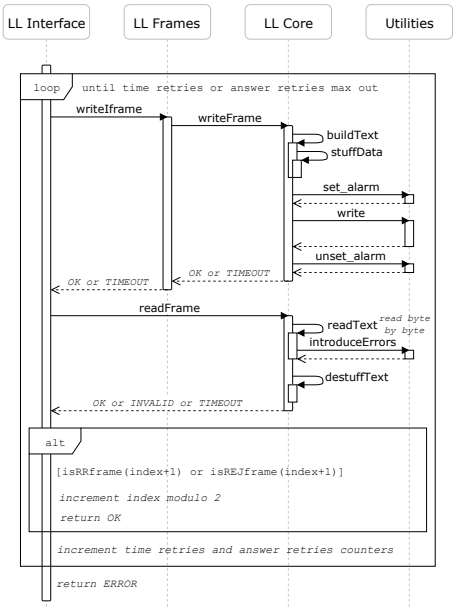
(a) File communications as seen through the app layer (part of function `send_file`)



(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.

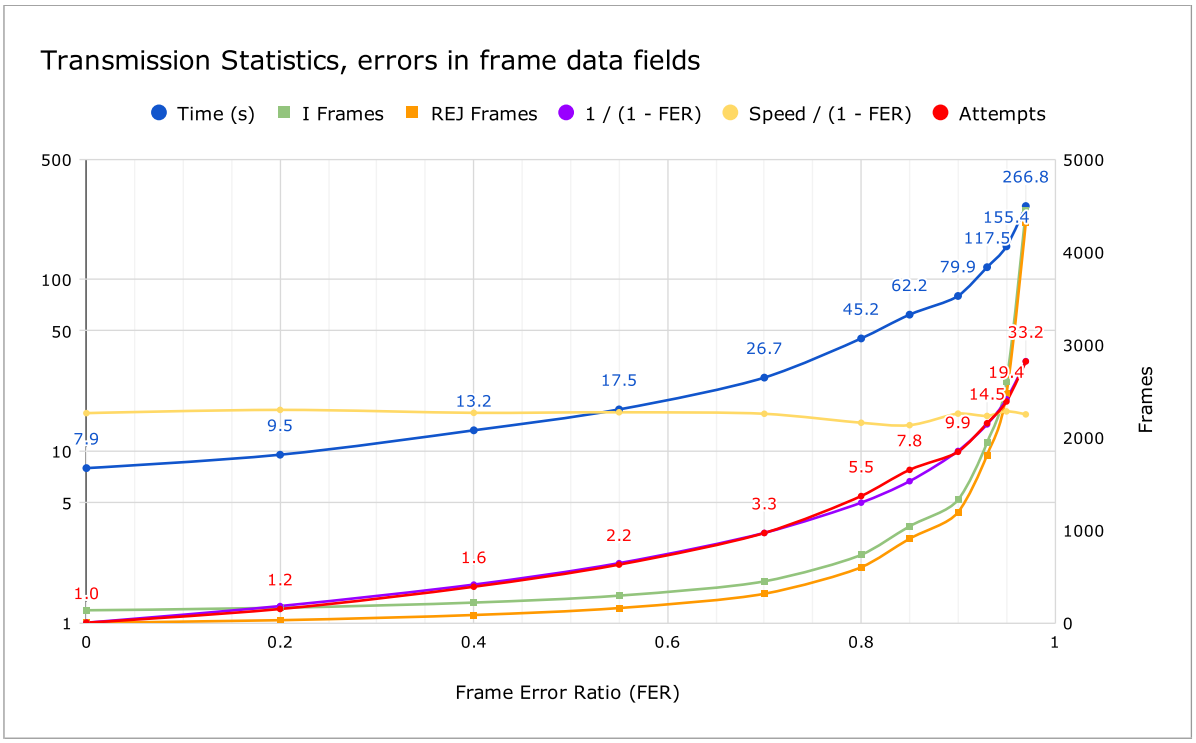


Figure 2: Transmission statistics of a 268.849B file, using packetsize 2048 and baudrate 38400 bytes/s. Error probability samples (`-f`): `FER` $= p \in \{0, 0.2, 0.4, 0.55, 0.7, 0.8, 0.85, 0.9, 0.93, 0.95, 0.97\}$.

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.

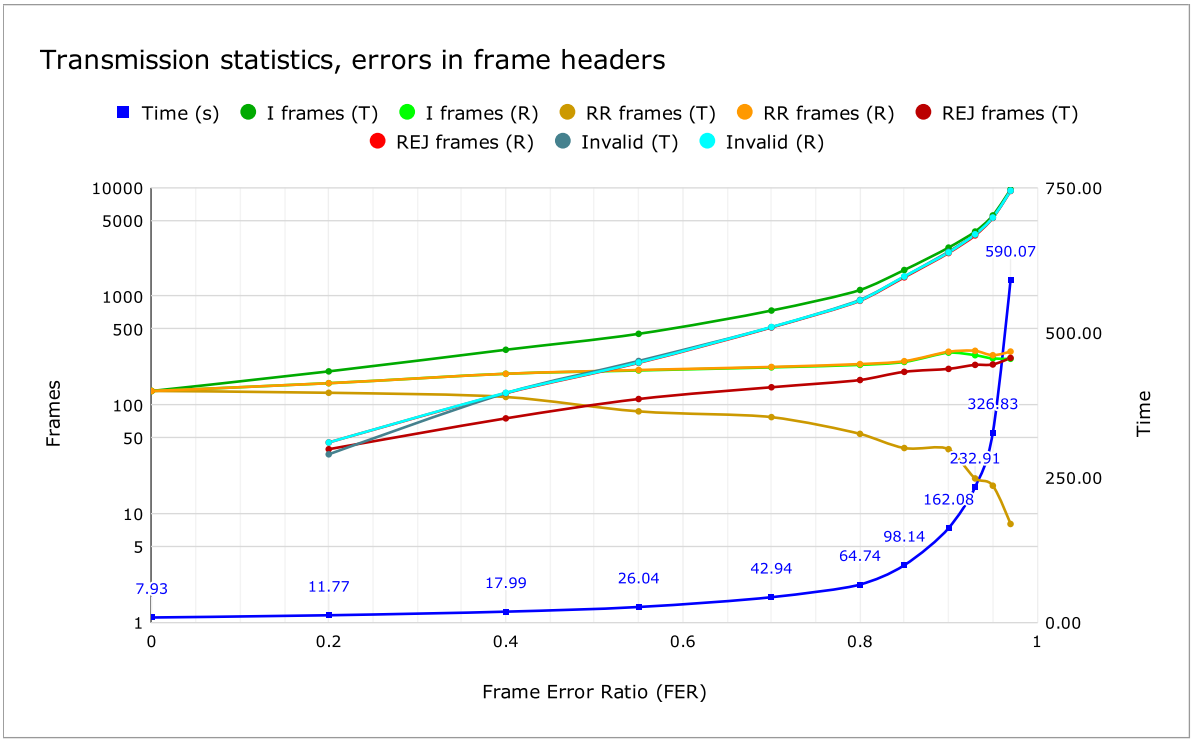


Figure 3: Transmission statistics of a 268.849B file, using packetsize 2048 and baudrate 38400 bytes/s. Error probability samples (`-h`): `FER` $= p \in \{0, 0.2, 0.4, 0.55, 0.7, 0.8, 0.85, 0.9, 0.93, 0.95, 0.97\}$.

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`).

Appendix: Code

app-layer.c

```c
#include "app-layer.h"
#include "ll-interface.h"
#include "debug.h"

#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

static int out_packet_index = 0; // only supports one fd.
static int in_packet_index = 0; // only supports one fd.

void free_control_packet(control_packet packet) {
    for (size_t i = 0; i < packet.n; ++i) {
        free(packet.tlvs[i].value.s);
    }
    free(packet.tlvs);
}

void free_data_packet(data_packet packet) {
    free(packet.data.s);
}

static bool isDATApacket(string packet_str, data_packet* outp) {
    char c = packet_str.s[0];

    if (packet_str.len < 5 || packet_str.s == NULL || c != PCONTROL_DATA) {
        if (TRACE_APP) {
            printf("[APP] isDATApacket() ? 0\n");
        }
        return false;
    }

    int index = (unsigned char)packet_str.s[1];
    unsigned char l2 = packet_str.s[2];
    unsigned char l1 = packet_str.s[3];
    size_t len = (size_t)l1 + 256 * (size_t)l2;

    bool b = len == (packet_str.len - 4);

    if (TRACE_APP) {
        printf("[APP] isDATApacket() ? %d [index=%d len=%lu]\n", (int)b, b ? index % 256 : 0, b ? len : 0);
    }

    if (b) {
        string data;

        data.len = len;
        data.s = malloc((len + 1) * sizeof(char));
        memcpy(data.s, packet_str.s + 4, len + 1);

        data_packet out = {index, data};

        *outp = out;

        if (index != in_packet_index % 256) {
            printf("[APP] Error: Expected DATA packet #%d, got #%d\n", in_packet_index % 256, index);
        }
    }
    return b;
}

static bool isCONTROLpacket(char c, string packet_str, control_packet* outp) {
    char c_char = packet_str.s[0];
    if (c_char != c) return false;

    if (packet_str.len == 0 || packet_str.s == NULL || c_char != c) return false;

    control_packet out;
    out.c = c;
    out.n = 0;

    out.tlvs = malloc(2 * sizeof(tlv));
    size_t reserved = 2;

    size_t j = 1;

    while (j + 2 < packet_str.len) {
        char type = packet_str.s[j++];
        size_t l = (unsigned char)packet_str.s[j++];

        if (j + l > packet_str.len) break;

        if (out.n == reserved) {
            out.tlvs = realloc(out.tlvs, 2 * reserved * sizeof(tlv));
            reserved *= 2;
        }
```

```
        string value;

        value.len = l;
        value.s = malloc((l + 1) * sizeof(char));
        memcpy(value.s, packet_str.s + j, l);
        value.s[l] = '\0';

        out.tlvs[out.n++] = (tlv){type, value};

        j += l;
    }

    bool b = j == packet_str.len;

    if (!b) {
        free_control_packet(out);
    } else {
        *outp = out;
    }

    return b;
}

static bool isSTARTpacket(string packet_str, control_packet* outp) {
    bool b = isCONTROLpacket(PCONTROL_START, packet_str, outp);

    if (TRACE_APP) {
        printf("[APP] isSTARTpacket() ? %d\n", (int)b);
    }
    return b;
}

static bool isENDpacket(string packet_str, control_packet* outp) {
    bool b = isCONTROLpacket(PCONTROL_END, packet_str, outp);

    if (TRACE_APP) {
        printf("[APP] isENDpacket() ? %d\n", (int)b);
    }
    return b;
}


bool get_tlv(control_packet control, char type, string* outp) {
    for (size_t i = 0; i < control.n; ++i) {
        if (control.tlvs[i].type == type) {
            string value;
            value.len = control.tlvs[i].value.len;
            value.s = strdup(control.tlvs[i].value.s);
            *outp = value;
            return true;
        }
    }
    return false;
}

bool get_tlv_filename(control_packet control, char** outp) {
    string value;
    bool b = get_tlv(control, PCONTROL_TYPE_FILENAME, &value);
    if (!b) {
        if (TRACE_APP_INTERNALS) {
            printf("[APPCORE] Get TLV filename: FAILED\n");
        }
        return false;
    }

    *outp = value.s;

    if (TRACE_APP_INTERNALS) {
        printf("[APPCORE] Get TLV filename: OK [filename=%s]\n", value.s);
    }
    return true;
}

bool get_tlv_filesize(control_packet control, size_t* outp) {
    string value;
    bool b = get_tlv(control, PCONTROL_TYPE_FILESIZE, &value);
    if (!b) {
        if (TRACE_APP_INTERNALS) {
            printf("[APPCORE] Get TLV filesize: FAILED\n");
        }
        return false;
    }

    long parse = strtol(value.s, NULL, 10);
    free(value.s);
    if (parse <= 0) {
        if (TRACE_APP_INTERNALS) {
            printf("[APPCORE] Get TLV filesize: BAD PARSE [long=%ld]\n", parse);
        }
        return false;
    }

    *outp = (size_t)parse;

    if (TRACE_APP_INTERNALS) {
        printf("[APPCORE] Get TLV filesize: OK [filesize=%lu]\n", (size_t)parse);
```

```
    }
    return true;
}

static int build_data_packet(string fragment, char index, string* outp) {
    static const size_t mod = 256;
    static const size_t max_len = 0x0ffff;

    if (fragment.len > max_len) return 1;

    string data_packet;

    data_packet.len = fragment.len + 4;
    data_packet.s = malloc((data_packet.len + 1) * sizeof(char));

    data_packet.s[0] = PCONTROL_DATA;
    data_packet.s[1] = index;
    data_packet.s[2] = fragment.len / mod;
    data_packet.s[3] = fragment.len % mod;
    memcpy(data_packet.s + 4, fragment.s, fragment.len + 1);

    if (TRACE_APP_INTERNALS) {
        printf("[APPCORE] Built DP [c=0x%02x index=0x%02x l2=0x%02x l1=0x%02x flen=%lu]\n",
            (unsigned char)data_packet.s[0], (unsigned char)data_packet.s[1],
            (unsigned char)data_packet.s[2], (unsigned char)data_packet.s[3],
            fragment.len);
        if (TEXT_DEBUG) print_stringn(data_packet);
    }

    *outp = data_packet;
    return 0;
}

static int build_tlv_str(char type, string value, string* outp) {
    static const size_t max_len = 0x0ff;

    if (value.len > max_len) return 1;

    string tlv;

    tlv.len = value.len + 2;
    tlv.s = malloc((tlv.len + 3) * sizeof(char));

    tlv.s[0] = type;
    tlv.s[1] = value.len;
    memcpy(tlv.s + 2, value.s, value.len + 1);

    if (TRACE_APP_INTERNALS) {
        printf("[APPCORE] Built TLV [t=0x%02x l=%lu]", type, value.len);
        print_stringn(tlv);
    }

    *outp = tlv;
    return 0;
}

static int build_tlv_uint(char type, long unsigned value, string* outp) {
    char buf[10];
    string tmp;
    tmp.s = buf;
    sprintf(tmp.s, "%lu", value);
    tmp.len = strlen(tmp.s);
    return build_tlv_str(type, tmp, outp);
}

static int build_control_packet(char control, string* tlvp, size_t n, string* outp) {
    string control_packet;
    control_packet.len = 1;

    for (size_t i = 0; i < n; ++i) {
        control_packet.len += tlvp[i].len;
    }

    control_packet.s = malloc((control_packet.len + 1) * sizeof(char));
    char* tmp = control_packet.s + 1;

    control_packet.s[0] = control;
    control_packet.s[1] = '\0';

    for (size_t i = 0; i < n; ++i) {
        memcpy(tmp, tlvp[i].s, tlvp[i].len);
        tmp += tlvp[i].len;
    }

    if (TRACE_APP_INTERNALS) {
        printf("[APPCORE] Built CP [c=0x%02x n=%lu tlen=%lu]\n",
        control, n, control_packet.len);
        if (TEXT_DEBUG) print_stringn(control_packet);
    }

    *outp = control_packet;
    return 0;
}

int send_data_packet(int fd, string packet) {
    int s;
```

```
    string data_packet;
    s = build_data_packet(packet, out_packet_index % 256lu, &data_packet);
    if (s != 0) return s;

    if (TRACE_APP) {
        printf("[APP] Sending DATA packet #%d [plen=%lu]\n",
        out_packet_index % 256, packet.len);
    }

    ++out_packet_index;
    s = llwrite(fd, data_packet);
    free(data_packet.s);
    return s;
}

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);

    if (TRACE_APP) {
        printf("[APP] Sending START packet [filesize=%lu,filename=%s,plen=%lu]\n", filesize, filename, start_packet.len);
    }

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

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

    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 end_packet;
    s = build_control_packet(PCONTROL_END, tlvs, 2, &end_packet);
    if (s != 0) return s;

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

    if (TRACE_APP) {
        printf("[APP] Sending END packet [filesize=%lu,filename=%s,plen=%lu]\n", filesize, filename, end_packet.len);
    }

    s = llwrite(fd, end_packet);
    free(end_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;
}
```

app-layer.h

```
#ifndef APP_LAYER_H___
#define APP_LAYER_H___

#include "strings.h"

#include <stdbool.h>

#define PCONTROL_DATA          0x41
#define PCONTROL_START         0x42
#define PCONTROL_END           0x43
#define PCONTROL_BAD_PACKET    0x40

#define PCONTROL_TYPE_FILESIZE 0x00
#define PCONTROL_TYPE_FILENAME 0x01

#define FILESIZE_TLV_N         0
#define FILENAME_TLV_N         1

#define PRECEIVE_DATA          0x51
#define PRECEIVE_START         0x52
#define PRECEIVE_END           0x53
#define PRECEIVE_BAD_PACKET    0x54

#define MAXIMUM_PACKET_SIZE    0x0fffflu

typedef struct {
    char type;
    string value;
} tlv;

typedef struct {
    char c;
    tlv* tlvs;
    size_t n;
} control_packet;

typedef struct {
    int index;
    string data;
} data_packet;


void free_control_packet(control_packet packet);

void free_data_packet(data_packet packet);


bool get_tlv(control_packet controlp, char type, string* outp);

bool get_tlv_filename(control_packet controlp, char** outp);

bool get_tlv_filesize(control_packet controlp, size_t* outp);


int send_data_packet(int fd, string packet);

int send_start_packet(int fd, size_t filesize, char* filename);

int send_end_packet(int fd, size_t filesize, char* filename);

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

#endif // APP_LAYER_H___
```

debug.c

```
#include "debug.h"

communication_count_t counter;

void reset_counter() {
    communication_count_t dummy = {0};
    counter = dummy;
}
```

debug.h

```c
#ifndef DEBUG_H___
#define DEBUG_H___

// Call asserts
//#define NDEBUG

#include <assert.h>
#include <stddef.h>

// Trace LL interface behaviour (ll-interface)
#define TRACE_LL 0

// Trace LL is functions (ll-frames)
#define TRACE_LL_IS 0

// Trace LL write functions (ll-frames)
#define TRACE_LL_WRITE 0

// Trace LL read errors (ll-core)
#define TRACE_LLERR_READ 0

// Trace LL write errors (ll-core)
#define TRACE_LLERR_WRITE 0

// Trace chars read in LL (ll-core)
#define DEEP_DEBUG 0

// Trace LL frame corruption errors (ll-core)
#define TRACE_LL_ERRORS 0

// Trace IntroduceError behaviour (ll-errors)
#define TRACE_CORRUPTION 0

// Trace APP behaviour (app-layer)
#define TRACE_APP 0

// Trace APP internals (app-layer)
#define TRACE_APP_INTERNALS 0

// Trace FILE behaviour (fileio)
#define TRACE_FILE 0

// Print strings to stdout
#define TEXT_DEBUG 0

// Trace Setup (signals, options, ll-setup)
#define TRACE_SETUP 1

// Trace Signals (signals)
#define TRACE_SIG 1

// Trace Timing (timing)
#define TRACE_TIME 0

// Dump Options (options)
#define DUMP_OPTIONS 0

// Exit receive_file if a BAD packet is received
#define EXIT_ON_BAD_PACKET 0

typedef struct {
    size_t len, bcc1, bcc2;
} read_count_t;

typedef struct {
    size_t I[2], RR[2], REJ[2], SET, DISC, UA;
} frame_count_t;

typedef struct {
    frame_count_t in;
    frame_count_t out;
    read_count_t read;
    size_t invalid;
    size_t timeout;
    size_t bcc_errors;
} communication_count_t;

extern communication_count_t counter;

void reset_counter();

#endif // DEBUG_H___
```

fileio.c

```c
#include "fileio.h"
#include "app-layer.h"
#include "ll-interface.h"
#include "options.h"
#include "timing.h"
#include "signals.h"
#include "debug.h"
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

static void free_packets(string* packets, size_t number_packets) {
    for (size_t i = 0; i < number_packets; ++i) {
        free(packets[i].s);
    }
    free(packets);
}

int send_file(int fd, char* filename) {
    int s = 0;

    int filefd = open(filename, O_RDONLY);
    if (filefd == -1) {
        printf("[FILE] Error: Failed to open file %s [%s]\n", filename, strerror(errno));
        return 1;
    }

    FILE* file = fdopen(filefd, "r");
    if (file == NULL) {
        printf("[FILE] Error: Failed to open file stream %s [%s]\n", filename, strerror(errno));
        close(filefd);
        return 1;
    }

    // Seek to the end of the file to extract its size.
    s = fseek(file, 0, SEEK_END);
    if (s != 0) {
        printf("[FILE] Error: Failed to seek file %s [%s]\n", filename, strerror(errno));
        fclose(file);
        return 1;
    }

    long lfs = ftell(file);
    size_t filesize = lfs;
    rewind(file);

    if (lfs <= 0) {
        printf("[FILE] Error: Invalid filesize %ld (probably 0) %s\n", filesize, filename);
        fclose(file);
        return 1;
    }

    // Read the entire file and close it.
    char* buffer = malloc((filesize + 1) * sizeof(char));
    fread(buffer, filesize, 1, file);
    buffer[filesize] = '\0';

    fclose(file);

    if (TRACE_FILE) {
        printf("[FILE] File read and closed [filesize=%lu,filename=%s]\n", filesize, filename);
    }

    // Split the buffer into various strings.
    // The last one will have a smaller size.
    size_t number_packets = number_of_packets(filesize);
    string* packets = malloc(number_packets * sizeof(string));

    for (size_t i = 0; i < number_packets - 1; ++i) {
        size_t offset = i * packetsize;

        packets[i].s = malloc((packetsize + 1) * sizeof(char));
        packets[i].len = packetsize;

        memcpy(packets[i].s, buffer + offset, packetsize);
        packets[i].s[packetsize] = '\0';
    }

    // Last packet
    {
        size_t offset = (number_packets - 1) * packetsize;
        size_t size = filesize - offset;

        packets[number_packets - 1].s = malloc((size + 1) * sizeof(char));
        packets[number_packets - 1].len = size;

        memcpy(packets[number_packets - 1].s, buffer + offset, size);
        packets[number_packets - 1].s[size] = '\0';
    }

    free(buffer);

    // Start communications.
    begin_timing(0);
    s = llopen(fd);
    if (s != LL_OK) goto error;

    if (TRACE_FILE) printf("[FILE] BEGIN Packets %s\n", filename);
```

```
    begin_timing(1);
    s = send_start_packet(fd, filesize, filename);
    if (s != LL_OK) goto error;

    // Send data packets.
    for (size_t i = 0; i < number_packets; ++i) {
        s = send_data_packet(fd, packets[i]);
        if (s != LL_OK) goto error;
    }

    // End communications.
    s = send_end_packet(fd, filesize, filename);
    if (s != LL_OK) goto error;
    end_timing(1);

    if (TRACE_FILE) printf("[FILE] END Packets %s\n", filename);

    s = llclose(fd);
    end_timing(0);

    if (show_statistics) print_stats(1, filesize);

    free_packets(packets, number_packets);
    return s ? 1 : 0;

    error:
    free_packets(packets, number_packets);
    return 1;
}

int receive_file(int fd) {
    int s = 0;

    // File variables.
    size_t filesize = 0;
    char* filename = NULL;

    // Packet variables.
    int type;
    control_packet cp;
    data_packet dp;

    // Start communications.
    begin_timing(0);
    s = llopen(fd);
    if (s != LL_OK) return 1;

    if (TRACE_FILE) printf("[FILE] BEGIN Packets\n");
    begin_timing(1);

    type = receive_packet(fd, &dp, &cp);

    switch (type) {
    case PRECEIVE_START:
        get_tlv_filesize(cp, &filesize);
        get_tlv_filename(cp, &filename);
        if (TRACE_FILE) {
            printf("[FILE] Received START packet [filesize=%lu,filename=%s]\n",
            filesize, filename);
        }
        free_control_packet(cp);
        break;
    case PRECEIVE_DATA:
        printf("[FILE] Error: Expected START packet, received DATA packet. Exiting\n");
        free_data_packet(dp);
        return 1;
    case PRECEIVE_END:
        printf("[FILE] Error: Expected START packet, received END packet. Exiting\n");
        free_control_packet(cp);
        return 1;
    case PRECEIVE_BAD_PACKET: default:
        printf("[FILE] Error: Expected START packet, received BAD packet. Exiting\n");
        return 1;
    }

    string* packets = malloc(10 * sizeof(string));
    size_t reserved = 10, number_packets = 0;
    bool done = false, reached_end = false;

    while (!done) {
        type = receive_packet(fd, &dp, &cp);

        if (number_packets == reserved) {
            packets = realloc(packets, 2 * reserved * sizeof(string));
            reserved *= 2;
        }

        switch (type) {
        case PRECEIVE_START:
            printf("[FILE] Error: Expected DATA/END packet, received START packet. Continuing\n");
            free_control_packet(cp);
            break;
        case PRECEIVE_DATA:
            packets[number_packets++] = dp.data;
            break;
```

```
        case PRECEIVE_END:
            done = true;
            reached_end = true;

            size_t end_filesize = 0;
            char* end_filename = NULL;
            get_tlv_filesize(cp, &end_filesize);
            get_tlv_filename(cp, &end_filename);

            if (TRACE_FILE) {
                printf("[FILE] Received END packet [ndata=%lu,filesize=%lu,filename=%s]\n", number_packets, filesize,
    filename);

                if (end_filesize == filesize) {
                    printf("[FILE] END packet: filesize OK\n");
                } else {
                    printf("[FILE] END packet: filesize NOT OK [start=%lu]", filesize);
                }

                if (strcmp(filename, end_filename) == 0) {
                    printf("[FILE] END packet: filename OK\n");
                } else {
                    printf("[FILE] END packet: filename NOT OK [start=%s]", filename);
                }
            }

            free(end_filename);
            free_control_packet(cp);
            break;
        case PRECEIVE_BAD_PACKET:
            printf("[FILE] Error: Expected DATA/END packet, received BAD packet.\n");
            if (EXIT_ON_BAD_PACKET) done = true;
            break;
        default:
            done = true;
            break;
        }
    }

    if (!reached_end) goto error;

    end_timing(1);
    if (TRACE_FILE) printf("[FILE] END Packets %s\n", filename);

    s = llclose(fd);
    if (s != LL_OK) {
        printf("[FILE] llclose failed. Writing to file %s anyway\n", filename);
    }
    end_timing(0);

    if (show_statistics) print_stats(1, filesize);

    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        perror("[FILE] Failed to open output file");
        goto error;
    }

    if (TRACE_FILE) printf("[FILE] Writing to file %s...\n", filename);

    for (size_t i = 0; i < number_packets; ++i) {
        fwrite(packets[i].s, packets[i].len, 1, file);
    }

    if (TRACE_FILE) printf("[FILE] Finished writing to file %s\n", filename);

    fclose(file);

    free_packets(packets, number_packets);
    free(filename);
    return s ? 1 : 0;

    error:
    free_packets(packets, number_packets);
    free(filename);
    return 1;
}

int send_files(int fd) {
    for (size_t i = 0; i < number_of_files; ++i) {
        int s = send_file(fd, files[i]);
        await_timeout();
        reset_counter();
        if (s != 0) return 1;
    }
    return 0;
}

int receive_files(int fd) {
    for (size_t i = 0; i < number_of_files; ++i) {
        int s = receive_file(fd);
        await_timeout();
        reset_counter();
        if (s != 0) return 1;
    }
    return 0;
```

```
}
```

fileio.h

```
#ifndef FILEIO_H___
#define FILEIO_H___

#include "app-layer.h"

size_t number_of_packets(size_t filesize);

int send_file(int fd, char* filename);

int receive_file(int fd);

int send_files(int fd);

int receive_files(int fd);

#endif // FILEIO_H___
```

ll-core.c

```
#include "ll-core.h"
#include "ll-errors.h"
#include "options.h"
#include "signals.h"
#include "debug.h"

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdbool.h>
#include <errno.h>
#include <assert.h>

//
// Performs stuffing on the string in, writing the result in the string out.
// The bcc2 char is computed and appended to the out string, and also returned
// through the out argument bcc2.
//
// This function does not fail.
//
// @param  in    String to be stuffed
// @param  outp  [out] Stuffed string, appended with computed bcc2
// @param  bcc2p [out] Computed bcc2
// @return 0
//
static int stuffData(string in, string* outp, char* bcc2p) {
    size_t stuff_count = 0;
    char parity = 0;

    for (size_t i = 0; i < in.len; ++i) {
        if (FRAME_MUST_ESCAPE(in.s[i])) ++stuff_count; parity ^= in.s[i];
    }

    bool stuff_bcc2 = FRAME_MUST_ESCAPE(parity);
    if (stuff_bcc2) ++stuff_count;

    string stuffed_data;
    stuffed_data.len = in.len + stuff_count + 1;
    stuffed_data.s = malloc((stuffed_data.len + 1) * sizeof(char));

    size_t j = 0;
    for (size_t i = 0; i < in.len; ++i) {
        switch (in.s[i]) {
        case FRAME_FLAG:
            stuffed_data.s[j++] = FRAME_ESC;
            stuffed_data.s[j++] = FRAME_FLAG_STUFFING;
            break;
        case FRAME_ESC:
            stuffed_data.s[j++] = FRAME_ESC;
            stuffed_data.s[j++] = FRAME_ESC_STUFFING;
            break;
        default:
            stuffed_data.s[j++] = in.s[i];
        }
    }

    switch (parity) {
    case FRAME_FLAG:
        stuffed_data.s[j++] = FRAME_ESC;
        stuffed_data.s[j++] = FRAME_FLAG_STUFFING;
        break;
    case FRAME_ESC:
        stuffed_data.s[j++] = FRAME_ESC;
        stuffed_data.s[j++] = FRAME_ESC_STUFFING;
        break;
    default:
        stuffed_data.s[j++] = parity;
```

```
    }

    stuffed_data.s[stuffed_data.len] = '\0';
    assert(j == stuffed_data.len);

    *outp = stuffed_data;
    *bcc2p = parity;
    return 0;
}

//
// Performs destuffing on the string in, writing the result in the string out.
// A bcc2 is presumed to be found at the end of in, possibly escaped.
// This character is removed, and not written to string out.
// The data's bcc2.is computed returned through the out argument bcc2.
//
// @param  in    String to be destuffed
// @param  outp  [out] Destuffed string, without appended bcc2
// @param  bcc2p [out] Computed bcc2
// @return 0 if successful
//         FRAME_READ_BAD_ESCAPE if there is a badly escaped character
//         FRAME_READ_BAD_BCC2 if bcc2 check does not pass
//
static int destuffData(string in, string* outp, char* bcc2p) {
    size_t count = 0;

    for (size_t i = 0; i < in.len; ++i) {
        if (in.s[i] == FRAME_ESC) ++count;
    }

    string destuffed_data;
    destuffed_data.len = in.len - count - 1;
    destuffed_data.s = malloc((destuffed_data.len + 1) * sizeof(char));

    char parity = 0;

    size_t j = 0;
    for (size_t i = 0; i < in.len; ++i, ++j) {
        if (in.s[i] == FRAME_ESC) {
            switch (in.s[++i]) {
            case FRAME_FLAG_STUFFING:
                destuffed_data.s[j] = FRAME_FLAG;
                break;
            case FRAME_ESC_STUFFING:
                destuffed_data.s[j] = FRAME_ESC;
                break;
            default:
                if (TRACE_LL_ERRORS) {
                    printf("[LLERR] Bad Escape [c=%c,0x%02x,i=%lu]\n", in.s[i], (unsigned char)in.s[i], i);
                }
                free(destuffed_data.s);
                return FRAME_READ_BAD_ESCAPE;
            }
        } else {
            destuffed_data.s[j] = in.s[i];
        }

        parity ^= destuffed_data.s[j];
    }

    assert(j == destuffed_data.len + 1);

    char bcc2 = destuffed_data.s[destuffed_data.len];
    parity ^= bcc2;

    if (bcc2 != parity) {
        if (TRACE_LL_ERRORS) {
            printf("[LLERR] Bad BCC2 [calc=0x%02x,read=0x%02x] [len=%lu]\n", (unsigned char)parity, (unsigned char)bcc2,
    destuffed_data.len);
        }
        free(destuffed_data.s);
        return FRAME_READ_BAD_BCC2;
    }

    destuffed_data.s[destuffed_data.len] = '\0'; // clear bcc2

    *outp = destuffed_data;
    *bcc2p = parity;
    return 0;
}

//
// A wrapper function for destuffData, which first extracts
// the data fragment a string from the frame string text.
//
// @param  text Full frame text
// @param  outp [out] Destuffed data string
// @param  bcc2 [out] Computed bcc2
// @return 0 if successful
//
static int destuffText(string text, string* outp, char* bcc2) {
    string data;

    data.len = text.len - 5;
    data.s = malloc((data.len + 1) * sizeof(char));
```

```
    memcpy(data.s, text.s + 4, data.len);
    data.s[data.len] = '\0';

    int s = destuffData(data, outp, bcc2);

    free(data.s);
    return s;
}

//
// Constructs the frame string text from a frame struct.
//
// This function does not fail.
//
// @param  f     Frame to be stringified
// @param  textp [out] Stringified frame
// @return 0
//
static int buildText(frame f, string* textp) {
    if (f.data.s == NULL) {
        // S or U frame (control frame)
        string text;

        text.len = 5;
        text.s = malloc(6 * sizeof(char));

        text.s[0] = FRAME_FLAG;
        text.s[1] = f.a;
        text.s[2] = f.c;
        text.s[3] = f.a ^ f.c;
        text.s[4] = FRAME_FLAG;
        text.s[5] = '\0';

        *textp = text;
        return 0;
    } else {
        // I frame (data frame)
        string stuffed_data;
        char bcc2;
        stuffData(f.data, &stuffed_data, &bcc2);

        string text;

        text.len = stuffed_data.len + 5;
        text.s = malloc((text.len + 1) * sizeof(char));

        text.s[0] = FRAME_FLAG;
        text.s[1] = f.a;
        text.s[2] = f.c;
        text.s[3] = f.a ^ f.c;
        memcpy(text.s + 4, stuffed_data.s, stuffed_data.len);
        text.s[text.len - 1] = FRAME_FLAG;
        text.s[text.len] = '\0';

        free(stuffed_data.s);
        *textp = text;
        return 0;
    }
}

//
// State machine for readText function
//
typedef enum {
    READ_PRE_FRAME, READ_START_FLAG, READ_WITHIN_FRAME, READ_END_FLAG
} FrameReadState;

//
// Primary Link Layer reading function.
//
// State is controlled by the FrameReadState enum. It starts reading
// the frame, including one initial and one terminal flag characters,
// once it encounters a non-flag character after a flag. This non-flag
// character must be a valid A character, as tested by FRAME_VALID_A,
// otherwise it assumes it is reading noise.
//
// Enable DEEP_DEBUG in debug.h to echo characters read in the terminal.
//
// @param  fd    Communications file descriptor
// @param  textp [out] Frame text read
// @return 0 if successful
//         FRAME_READ_TIMEOUT if a timeout occurred
//         FRAME_READ_INVALID if some other unknown error occurred
//
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
        if (DEEP_DEBUG) {
            printf("[LLREAD] s:%01d  c:%02x  state:%01d  |  %c\n",
            (int)s, (unsigned char)c, state, c);
        }

        if (s == 0) {
            if (++timed == timeout) {
                if (TRACE_LLERR_READ) {
                    printf("[LLREAD] Timeout [len=%lu]\n", text.len);
                }
                free(text.s);
                return FRAME_READ_TIMEOUT;
            } else {
                continue;
            }
        }

        if (s == -1) {
            if (errno == EINTR) {
                if (TRACE_LLERR_READ) {
                    printf("[LLREAD] Error EINTR [len=%lu]\n", text.len);
                }
                continue;
            } else if (errno == EIO) {
                if (TRACE_LLERR_READ) {
                    printf("[LLREAD] Error EIO [len=%lu]\n", text.len);
                }
                continue;
            } else if (errno == EAGAIN) {
                if (TRACE_LLERR_READ) {
                    printf("[LLREAD] Error EAGAIN [len=%lu]\n", text.len);
                }
                continue;
            } else {
                if (TRACE_LLERR_READ) {
                    printf("[LLREAD] Error %s [len=%lu]\n", strerror(errno), text.len);
                }
                free(text.s);
                return FRAME_READ_INVALID;
            }
        }

        if (text.len + 1 == reserved) {
            text.s = realloc(text.s, 2 * reserved * sizeof(char));
            reserved *= 2;
        }

        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 {
                    if (TRACE_LLERR_READ) {
                        printf("[LLREAD] Bad A 0x%02x, back to pre-frame\n", c);
                    }
                    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;
}

//
// Writes a frame to communication device.
// @param  fd Communications file descriptor
```

```
// @param  f  Frame to be written
// @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) {
        if (TRACE_LLERR_WRITE) {
            printf("[LLWRITE] Timeout [alarm=%d s=%d errno=%d] [%s]\n",
            (int)b, (int)s, err, strerror(err));
        }
        return FRAME_WRITE_TIMEOUT;
    } else {
        return FRAME_WRITE_OK;
    }
}

//
// Reads a frame from communication device.
//
// @param  fd Communications file descriptor
// @param  fp [out] Frame read
// @return FRAME_READ_OK if successful
//         FRAME_READ_TIMEOUT if a timeout occurred while reading
//         FRAME_READ_INVALID if the frame read is invalid
//
int readFrame(int fd, frame* fp) {
    string text;
    frame dummy = {0, 0, {NULL, 0}};
    *fp = dummy;

    int s = readText(fd, &text);

    if (s != 0) return s;

    if (text.len < 5 || text.len == 6) {
        if (TRACE_LL_ERRORS) {
            printf("[LLERR] Bad Length [len=%lu]\n",
            text.len);
        }
        free(text.s);
        ++counter.read.len;
        return FRAME_READ_INVALID;
    }

    frame f = {
        .a = text.s[1],
        .c = text.s[2],
        .data = {NULL, 0}
    };

    char bcc1 = text.s[3];

    if (bcc1 != (f.a ^ f.c)) {
        if (TRACE_LL_ERRORS) {
            printf("[LLERR] Bad BCC1 [a=0x%02x,c=0x%02x,bcc1=0x%02x]\n", f.a, f.c, bcc1);
        }
        free(text.s);
        ++counter.read.bcc1;
        return FRAME_READ_INVALID;
    }

    if (text.len > 6) {
        string data;
        char bcc2;
        int s = destuffText(text, &data, &bcc2);

        if (s != 0) {
            free(text.s);
            ++counter.read.bcc2;
            return FRAME_READ_INVALID;
        }

        f.data = data;
    }

    *fp = f;

    free(text.s);
    return FRAME_READ_OK;
}
```

ll-core.h

```
#ifndef LL_CORE_H___
```

```
#define LL_CORE_H___

#include "strings.h"

#define FRAME_ESC               0x7d
#define FRAME_FLAG_STUFFING     0x5e
#define FRAME_ESC_STUFFING      0x5d
#define FRAME_FLAG              0x7e
#define FRAME_MUST_ESCAPE(c)    ((c == FRAME_ESC) || (c == FRAME_FLAG))

#define FRAME_READ_OK           0x00
#define FRAME_READ_INVALID      0x10
#define FRAME_READ_TIMEOUT      0x20

#define FRAME_READ_BAD_LENGTH   0x11
#define FRAME_READ_BAD_BCC1     0x12
#define FRAME_READ_BAD_BCC2     0x13
#define FRAME_READ_BAD_ESCAPE   0x14

#define FRAME_WRITE_OK          0x00
#define FRAME_WRITE_TIMEOUT     0x30

#define FRAME_A_COMMAND         (char)0x03
#define FRAME_A_RESPONSE        (char)0x01
#define FRAME_VALID_A(c)        ((c == FRAME_A_COMMAND) || (c == FRAME_A_RESPONSE))

#define FRAME_C_I(n)            (char)((n % 2) ? 0x01 : 0x00)
#define FRAME_C_SET             (char)0x03
#define FRAME_C_DISC            (char)0x0b
#define FRAME_C_UA              (char)0x07
#define FRAME_C_RR(n)           (char)((n % 2) ? 0x85 : 0x05)
#define FRAME_C_REJ(n)          (char)((n % 2) ? 0x81 : 0x01)

//
// Frames I, SET, DISC: Command
//
// Frames UA, RR, REJ: Response
//

typedef struct {
    char a, c;
    string data;
} frame;

int writeFrame(int fd, frame f);

int readFrame(int fd, frame* fp);

#endif // LL_CORE_H___
```

ll-errors.c

```
#include "ll-errors.h"
#include "debug.h"
#include "options.h"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

static bool srand_seeded = false;

static void seed_srand() {
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    srand(t.tv_nsec);
    srand_seeded = true;
}

static inline char corruptByte(char byte) {
    return byte ^ (1 << (rand() % 8));
}

static int introduceErrorsByte(string text) {
    int header_p = RAND_MAX * h_error_prob;
    int frame_p = RAND_MAX * f_error_prob;

    for (size_t i = 1; i < 4; ++i) {
        int header_r = rand();

        if (header_r < header_p) {
            char c = corruptByte(text.s[i]);

            if (TRACE_CORRUPTION) {
                printf("[CORR] [header i=%lu] Corrupted 0x%02x to 0x%02x\n", i, (unsigned char)text.s[i], (unsigned char)c);
            }

            text.s[i] = c;
        }
    }

    for (size_t i = 4; i < text.len - 1; ++i) {
        int frame_r = rand();

        if (frame_r < frame_p) {
```

```
            char c = corruptByte(text.s[i]);

            if (TRACE_CORRUPTION) {
                printf("[CORR] [frame i=%lu] Corrupted 0x%02x to 0x%02x\n", i, (unsigned char)text.s[i], (unsigned char)c);
            }

            text.s[i] = c;
        }
    }

    return 0;
}

static int introduceErrorsFrame(string text) {
    int header_p = RAND_MAX * h_error_prob;
    int frame_p = RAND_MAX * f_error_prob;

    int header_r = rand();
    size_t header_b = 1 + (rand() % 3);

    if (header_r < header_p) {
        char c = corruptByte(text.s[header_b]);

        if (TRACE_CORRUPTION) {
            printf("[CORR] [frame i=%lu] Corrupted 0x%02x to 0x%02x\n",
                header_b, (unsigned char)text.s[header_b], (unsigned char)c);
        }

        text.s[header_b] = c;
    }

    if (text.len > 5) {
        int frame_r = rand();
        size_t frame_b = 4 + (rand() % (text.len - 5));

        if (frame_r < frame_p) {
            char c = corruptByte(text.s[frame_b]);

            if (TRACE_CORRUPTION) {
                printf("[CORR] [frame i=%lu] Corrupted 0x%02x to 0x%02x\n",
                    frame_b, (unsigned char)text.s[frame_b], (unsigned char)c);
            }

            text.s[frame_b] = c;
        }
    }

    return 0;
}

int introduceErrors(string text) {
    if (!srand_seeded) seed_srand();

    if (error_type == ETYPE_BYTE) {
        return introduceErrorsByte(text);
    } else if (error_type == ETYPE_FRAME) {
        return introduceErrorsFrame(text);
    }

    return 0;
}
```

ll-errors.h

```
#ifndef LL_ERRORS_H___
#define LL_ERRORS_H___

#include "strings.h"

void reset_counter();

int introduceErrors(string text);

#endif // LL_ERRORS_H___
```

ll-frames.c

```
#include "ll-frames.h"
#include "debug.h"

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

bool isIframe(frame f, int parity) {
    bool b = f.a == FRAME_A_COMMAND &&
    f.c == FRAME_C_I(parity) &&
    f.data.s != NULL && f.data.len > 0;

    if (b) ++counter.in.I[parity % 2];

    if (TRACE_LL_IS) printf("[LL] isIframe(%d) ? %d\n", parity % 2, (int)b);
    return b;
}
```

```
bool isSETframe(frame f) {
    bool b = f.a == FRAME_A_COMMAND &&
    f.c == FRAME_C_SET &&
    f.data.s == NULL;

    if (b) ++counter.in.SET;

    if (TRACE_LL_IS) printf("[LL] isSETframe() ? %d\n", (int)b);
    return b;
}

bool isDISCframe(frame f) {
    bool b = f.a == FRAME_A_COMMAND &&
    f.c == FRAME_C_DISC &&
    f.data.s == NULL;

    if (b) ++counter.in.DISC;

    if (TRACE_LL_IS) printf("[LL] isDISCframe() ? %d\n", (int)b);
    return b;
}

bool isUAframe(frame f) {
    bool b = f.a == FRAME_A_RESPONSE &&
    f.c == FRAME_C_UA &&
    f.data.s == NULL;

    if (b) ++counter.in.UA;

    if (TRACE_LL_IS) printf("[LL] isUAframe() ? %d\n", (int)b);
    return b;
}

bool isRRframe(frame f, int parity) {
    bool b = f.a == FRAME_A_RESPONSE &&
    f.c == FRAME_C_RR(parity) &&
    f.data.s == NULL;

    if (b) ++counter.in.RR[parity % 2];

    if (TRACE_LL_IS) printf("[LL] isRRframe(%d) ? %d\n", parity % 2, (int)b);
    return b;
}

bool isREJframe(frame f, int parity) {
    bool b = f.a == FRAME_A_RESPONSE &&
    f.c == FRAME_C_REJ(parity) &&
    f.data.s == NULL;

    if (b) ++counter.in.REJ[parity % 2];

    if (TRACE_LL_IS) printf("[LL] isREJframe(%d) ? %d\n", parity % 2, (int)b);
    return b;
}



int answerBADframe(int fd, frame f) {
    static int rrpar = 0;
    int s = 0;

    if (isIframe(f, 0) || isIframe(f, 1)) {
        if (isIframe(f, 0)) {
            s = writeRRframe(fd, 1);
        } else {
            s = writeRRframe(fd, 0);
        }

        free(f.data.s);
    } else {
        s = writeRRframe(fd, rrpar++);
    }

    return s;
}



int writeIframe(int fd, string message, int parity) {
    frame f = {
        .a = FRAME_A_COMMAND,
        .c = FRAME_C_I(parity),
        .data = message
    };

    ++counter.out.I[parity % 2];

    if (TRACE_LL_WRITE) {
        printf("[LL] writeIframe(%d) [flen=%lu]\n", parity % 2, f.data.len);
        if (TEXT_DEBUG) print_stringn(message);
    }
    return writeFrame(fd, f);
}

int writeSETframe(int fd) {
    frame f = {
```

```
        .a = FRAME_A_COMMAND,
        .c = FRAME_C_SET,
        .data = {NULL, 0}
    };

    ++counter.out.SET;

    if (TRACE_LL_WRITE) printf("[LL] writeSETframe()\n");
    return writeFrame(fd, f);
}

int writeDISCframe(int fd) {
    frame f = {
        .a = FRAME_A_COMMAND,
        .c = FRAME_C_DISC,
        .data = {NULL, 0}
    };

    ++counter.out.DISC;

    if (TRACE_LL_WRITE) printf("[LL] writeDISCframe()\n");
    return writeFrame(fd, f);
}

int writeUAframe(int fd) {
    frame f = {
        .a = FRAME_A_RESPONSE,
        .c = FRAME_C_UA,
        .data = {NULL, 0}
    };

    ++counter.out.UA;

    if (TRACE_LL_WRITE) printf("[LL] writeUAframe()\n");
    return writeFrame(fd, f);
}

int writeRRframe(int fd, int parity) {
    frame f = {
        .a = FRAME_A_RESPONSE,
        .c = FRAME_C_RR(parity),
        .data = {NULL, 0}
    };

    ++counter.out.RR[parity % 2];

    if (TRACE_LL_WRITE) printf("[LL] writeRRframe(%d)\n", parity % 2);
    return writeFrame(fd, f);
}

int writeREJframe(int fd, int parity) {
    frame f = {
        .a = FRAME_A_RESPONSE,
        .c = FRAME_C_REJ(parity),
        .data = {NULL, 0}
    };

    ++counter.out.REJ[parity % 2];

    if (TRACE_LL_WRITE) printf("[LL] writeREJframe(%d)\n", parity % 2);
    return writeFrame(fd, f);
}
```

ll-frames.h

```
#ifndef LL_FRAMES_H___
#define LL_FRAMES_H___

#include "ll-core.h"
#include "strings.h"

#include <stdbool.h>

bool isIframe(frame f, int parity);

bool isSETframe(frame f);

bool isDISCframe(frame f);

bool isUAframe(frame f);

bool isRRframe(frame f, int parity);

bool isREJframe(frame f, int parity);


int answerBADframe(int fd, frame f);


int writeIframe(int fd, string message, int index);

int writeSETframe(int fd);

int writeDISCframe(int fd);
```

```c
int writeUAframe(int fd);

int writeRRframe(int fd, int parity);

int writeREJframe(int fd, int parity);

#endif // LL_FRAMES_H___
```

ll-interface.c

```c
#include "ll-interface.h"
#include "ll-frames.h"
#include "options.h"
#include "debug.h"

#include <stdlib.h>
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

//
// llopen for T
//
// @param  fd Link layer's file descriptor
// @return LL_OK if llopen succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
static int llopen_transmitter(int fd) {
    int time_count = 0, answer_count = 0;

    tcflush(fd, TCIFLUSH);

    while (time_count < time_retries && answer_count < answer_retries) {
        int s = writeSETframe(fd);
        if (s != FRAME_WRITE_OK) {
            ++time_count, ++counter.timeout;
            continue;
        }

        frame f;
        s = readFrame(fd, &f);

        switch (s) {
        case FRAME_READ_OK:
            if (isUAframe(f)) {
                if (TRACE_LL || TRACE_FILE) {
                    printf("[LL] llopen (T) OK\n");
                }
                return LL_OK;
            }
            // FALLTHROUGH
        case FRAME_READ_INVALID:
            if (TRACE_LL || TRACE_FILE) {
                printf("[LL] llopen (T) ASSUME UA OK\n");
            }
            ++counter.invalid;
            return LL_OK;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llopen (T) FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llopen (T) FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

//
// llopen for R
//
// @param  fd Link layer's file descriptor
// @return LL_OK if llopen succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
static int llopen_receiver(int fd) {
    int time_count = 0, answer_count = 0;

    tcflush(fd, TCOFLUSH);

    while (time_count < time_retries && answer_count < answer_retries) {
        frame f;
        int s = readFrame(fd, &f);

        switch (s) {
        case FRAME_READ_OK:
            if (isSETframe(f)) {
                writeUAframe(fd);
                if (TRACE_LL || TRACE_FILE) {
```

```
                    printf("[LL] llopen (R) OK\n");
                }
                return LL_OK;
            }
            // FALLTHROUGH
        case FRAME_READ_INVALID:
            ++answer_count, ++counter.invalid;
            break;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llopen (R) FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llopen (R) FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

//
// llclose for T
//
// @param  fd Link layer's file descriptor
// @return LL_OK if llclose succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
static int llclose_transmitter(int fd) {
    int time_count = 0, answer_count = 0;

    while (time_count < time_retries && answer_count < answer_retries) {
        int s = writeDISCframe(fd);
        if (s != FRAME_WRITE_OK) {
            ++time_count, ++counter.timeout;
            continue;
        }

        frame f;
        s = readFrame(fd, &f);

        switch (s) {
        case FRAME_READ_OK:
            if (isDISCframe(f)) {
                writeUAframe(fd);
                if (TRACE_LL || TRACE_FILE) {
                    printf("[LL] llclose (T) OK\n");
                }
                return LL_OK;
            }
            // FALLTHROUGH
        case FRAME_READ_INVALID:
            ++answer_count, ++counter.invalid;
            break;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llclose (T) FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llclose (T) FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

// So here we have an issue. The protocol goes:
//    (1) Wait for a DISC.
//        If this is difficult keep trying/waiting to read a DISC,
//        and increment the counts accordingly.
//    (2) Finally we read a good DISC.
//    (3) Send DISC.
//    (4) Wait for an answer.
//    (5.1) Bad answer: ???
//    (5.2) Good answer:
//        If the answer is UA all is good and we leave.
//        If the answer is DISC we got back to (2).
//
// Consider a situation where the line has a lot of noise.
// After we answer a good DISC with a DISC,
// the answer we receive at (4) is all mangled up.
// We can't assume UA in (5.1) similarly to how we did in llopen
// because if the answer is DISC we'd let T hung up,
// which isn't nice enough. So we test T. Set a testing flag to change
// the usual behaviour (test_status) and do as follows:
//
//    (5.1) Bad answer: Respond with DISC.
//    (5.1.1) If the write times out, the answer was a UA and T is gone.
//    (5.1.2) If the read times out, no answer, same thing.
//    (5.1.3) If neither times out, there is an answer, so we assume
```

```
//        we had gotten a DISC in (4) and goto (2).

//
// llclose for R
//
// @param  fd Link layer's file descriptor
// @return LL_OK if llclose succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
static int llclose_receiver(int fd) {
    int time_count = 0, answer_count = 0;

    bool answered_disc = false;

    while (time_count < time_retries && answer_count < answer_retries) {
        frame f;
        int s = readFrame(fd, &f); // 1

        switch (s) {
        case FRAME_READ_OK:
            if (isDISCframe(f)) { // 2
                answer:
                s = writeDISCframe(fd); // 3
                if (s != FRAME_WRITE_OK) {
                    if (answered_disc) {
                        if (TRACE_LL || TRACE_FILE) {
                            printf("[LL] llclose (R) WRITE TIMEOUT ASSUME OK\n");
                        }
                        return LL_OK;
                    } else {
                        ++time_count, ++counter.timeout;
                        continue;
                    }
                }

                answered_disc = true;

                s = readFrame(fd, &f); // 4

                switch (s) {
                case FRAME_READ_OK: // 5.2
                    if (isUAframe(f)) {
                        if (TRACE_LL || TRACE_FILE) {
                            printf("[LL] llclose (R) OK\n");
                        }
                        return LL_OK;
                    }
                    // FALLTHROUGH
                case FRAME_READ_INVALID: // 5.1
                    answered_disc = false;
                    ++answer_count, ++counter.invalid;
                    goto answer;
                case FRAME_READ_TIMEOUT:
                    if (answered_disc) {
                        if (TRACE_LL || TRACE_FILE) {
                            printf("[LL] llclose (R) READ TIMEOUT ASSUME OK\n");
                        }
                        ++counter.timeout;
                        return LL_OK;
                    }
                    ++time_count, ++counter.timeout;
                    break;
                }

                break;
            }
            // FALLTHROUGH
        case FRAME_READ_INVALID:
            answerBADframe(fd, f);
            ++answer_count, ++counter.invalid;
            break;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llclose (R) FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llclose (R) FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

//
// @param  fd Link layer's file descriptor
// @return LL_OK if llopen succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
int llopen(int fd) {
    if (my_role == TRANSMITTER) {
        return llopen_transmitter(fd);
```

```
    } else {
        return llopen_receiver(fd);
    }
}

//
// @param fd      Link layer's file descriptor
// @param message String to be sent over LL
// @return LL_OK if llwrite succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
int llwrite(int fd, string message) {
    static int index = 0; // only supports one fd.

    int time_count = 0, answer_count = 0;

    while (time_count < time_retries && answer_count < answer_retries) {
        int s = writeIframe(fd, message, index);
        if (s != FRAME_WRITE_OK) {
            ++time_count, ++counter.timeout;
            continue;
        }

        frame f;
        s = readFrame(fd, &f);

        switch (s) {
        case FRAME_READ_OK:
            if (isRRframe(f, index + 1) || isREJframe(f, index + 1)) {
                ++index;
                if (TRACE_LL) {
                    printf("[LL] llwrite OK [index=%d]\n", index);
                }
                return LL_OK;
            } else if (isRRframe(f, index) || isREJframe(f, index)) {
                ++answer_count;
            } else {
                if (TRACE_LL) {
                    printf("[LL] llwrite: invalid response (not RR or REJ)\n");
                }
                ++answer_count, ++counter.invalid;
            }
            break;
        case FRAME_READ_INVALID:
            ++answer_count, ++counter.invalid;
            break;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llwrite FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llwrite FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

//
// @param fd       Link layer's file descriptor
// @param messagep Where to store the read message
// @return LL_OK if llread succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
int llread(int fd, string* messagep) {
    static int index = 0; // only supports one fd.

    int time_count = 0, answer_count = 0;

    while (time_count < time_retries && answer_count < answer_retries) {
        frame f;
        int s = readFrame(fd, &f);

        switch (s) {
        case FRAME_READ_OK:
            if (isIframe(f, index)) {
                writeRRframe(fd, ++index);
                *messagep = f.data;
                if (TRACE_LL) {
                    printf("[LL] llread OK [index=%d]\n", index);
                }
                return LL_OK;
            } else if (isIframe(f, index + 1)) {
                writeRRframe(fd, index);
                ++answer_count;
                if (TRACE_LL) {
                    printf("[LL] llread: Expected frame %d, got frame %d\n",
                    index % 2, (index + 1) % 2);
                }
                free(f.data.s);
            }
```

```
            break;
        case FRAME_READ_INVALID:
            writeREJframe(fd, index);
            ++answer_count, ++counter.invalid;
            break;
        case FRAME_READ_TIMEOUT:
            ++time_count, ++counter.timeout;
            break;
        }
    }

    if (time_count == time_retries) {
        printf("[LL] llwrite FAILED: %d time retries ran out\n", time_retries);
        return LL_NO_TIME_RETRIES;
    } else {
        printf("[LL] llwrite FAILED: %d answer retries ran out\n", answer_retries);
        return LL_NO_ANSWER_RETRIES;
    }
}

//
// @param  fd Link layer's file descriptor
// @return LL_OK if llclose succeeded,
//         LL_NO_TIME_RETRIES if timeouts maxed out,
//         LL_NO_ANSWER_RETRIES if answer errors maxed out.
//
int llclose(int fd) {
    if (my_role == TRANSMITTER) {
        return llclose_transmitter(fd);
    } else {
        return llclose_receiver(fd);
    }
}
```

ll-interface.h

```
#ifndef LL_INTERFACE_H___
#define LL_INTERFACE_H___

#include "strings.h"

#define LL_OK                   0x00
#define LL_NO_TIME_RETRIES      0x20
#define LL_NO_ANSWER_RETRIES    0x21

int llopen(int fd);

int llclose(int fd);

int llwrite(int fd, string message);

int llread(int fd, string* messagep);

#endif // LL_INTERFACE_H___
```

ll-setup.c

```
#include "ll-setup.h"
#include "options.h"
#include "debug.h"

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>

#define BAUDRATE B9600

static struct termios oldtios;

static int baudrates_list[] = {50, 75, 110, 134, 150, 200, 300, 600, 1200,
    1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800,
    500000, 576000};

static int baudrates_macros[] = {B50, B75, B110, B134, B150, B200, B300, B600,
    B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200,
    B230400, B460800, B500000, B576000};

static size_t baudrates_length = sizeof(baudrates_list) / sizeof(int);

static int select_baudrate() {
    for (size_t i = 0; i < baudrates_length; ++i) {
        if (baudrate == baudrates_list[i]) {
            return baudrates_macros[i];
        }
    }
    printf("[SETUP] Bad select_baudrate\n");
    exit(EXIT_FAILURE);
}
```

```
bool is_valid_baudrate(int baudrate) {
    for (size_t i = 0; i < baudrates_length; ++i) {
        if (baudrate == baudrates_list[i]) return true;
    }
    printf("[SETUP] Baudrate %d is invalid.\n", baudrate);
    return false;
}

//
// Opens the terminal with given file name, changes its configuration
// according to the specs, and returns the file descriptor fd.
//
// Assumption: name should be /dev/ttyS0 or /dev/ttyS1.
//
// @param  name The terminal's name
// @return The terminal's file descriptor, exit if unsuccessful
//
int setup_link_layer(const char* name) {
    // Open serial port device for reading and writing. Open as NOt Controlling TTY
    // (O_NOCTTY) because we don't want to get killed if linenoise sends CTRL-C.

    int fd = open(name, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        perror("[SETUP] Failed to open terminal");
        exit(EXIT_FAILURE);
    }

    if (TRACE_SETUP) printf("[SETUP] Opened device %s\n", name);

    // Save current terminal settings in oldtios.
    if (tcgetattr(fd, &oldtios) == -1) {
        perror("[SETUP] Failed to read old terminal settings (tcgetattr)");
        exit(EXIT_FAILURE);
    }

    // Setup new termios
    struct termios newtio;
    memset(&newtio, 0, sizeof(struct termios));

    int baud = select_baudrate();

    // c_iflag   Error handling...
    // IGNPAR :- Ignore framing errors and parity errors
    // ...
    newtio.c_iflag = IGNPAR;

    // c_oflag   Output delay...
    // ...
    newtio.c_oflag = 0;

    // c_cflag   Input manipulation, baud rates...
    // CS*    :- Character size mask (CS5, CS6, CS7, CS8)
    // CLOCAL :- Ignore modem control lines
    // CREAD  :- Enable receiver
    // CSTOPB :- Set two stop bits, rather than one
    // ...
    newtio.c_cflag = baud | CS8 | CLOCAL | CREAD;

    // c_lflag   Canonical or non canonical mode...
    // ICANON :- Enable canonical mode
    // ECHO   :- Echo input characters
    // ...
    newtio.c_lflag = 0;

    // c_cc   Special characters
    // VTIME  :- Timeout in deciseconds for noncanonical read (TIME).
    // VMIN   :- Minimum number of characters for noncanonical read (MIN).
    // ...
    newtio.c_cc[VTIME] = 1;
    newtio.c_cc[VMIN] = 0;

    // VTIME e VMIN devem ser alterados de forma a proteger com um temporizador a
    // leitura do(s) proximo(s) caracter(es)

    tcflush(fd, TCIOFLUSH);

    //cfsetispeed(&newtio, B38400);
    //cfsetospeed(&newtio, B38400);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        perror("[SETUP] Failed to set new terminal settings (tcsetattr)");
        exit(EXIT_FAILURE);
    }

    if (TRACE_SETUP) printf("[SETUP] Setup link layer on %s\n", name);
    return fd;
}

//
// Resets the terminal's settings to the old ones.
//
// @param  fd The terminal's open file descriptor
// @return 0 if successful, 1 otherwise.
//
int reset_link_layer(int fd) {
    if (tcsetattr(fd, TCSANOW, &oldtios) == -1) {
```

```
        perror("[RESET] Failed to set old terminal settings (tcsetattr)");
        close(fd);
        return 1;
    } else {
        if (TRACE_SETUP) {
            printf("[RESET] Reset device\n");
        }
        close(fd);
        return 0;
    }
}
```

ll-setup.h

```
#ifndef LL_SETUP_H___
#define LL_SETUP_H___

#include <stdbool.h>

bool is_valid_baudrate(int baudrate);

int setup_link_layer(const char* name);

int reset_link_layer(int fd);

#endif // LL_SETUP_H___
```

main.c

```
#include "options.h"
#include "signals.h"
#include "fileio.h"
#include "ll-setup.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

static void adjust_args() {
    if (packetsize > MAXIMUM_PACKET_SIZE) {
        printf("[MAIN] packetsize lowered from %lu to maximum size %lu\n",
        packetsize, MAXIMUM_PACKET_SIZE);
        packetsize = MAXIMUM_PACKET_SIZE;
    }
}

int main(int argc, char** argv) {
    parse_args(argc, argv);
    adjust_args();

    set_signal_handlers();
    test_alarm();

    int fd = setup_link_layer(device);

    if (my_role == TRANSMITTER) {
        send_files(fd);
    } else {
        receive_files(fd);
    }

    sleep(1);
    reset_link_layer(fd);
    return 0;
}
```

options.c

```
#include "options.h"
#include "ll-setup.h"
#include "debug.h"

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <getopt.h>
#include <errno.h>
#include <locale.h>
#include <wchar.h>
#include <limits.h>

// <!--- OPTIONS
static int show_help = false; // h, help
static int show_usage = false; // usage
static int show_version = false; // V, version
static int dump = false; // dump

int time_retries = TIME_RETRIES_DEFAULT; // time-retries
int answer_retries = ANSWER_RETRIES_DEFAULT; // answer-retries
int timeout = TIMEOUT_DEFAULT; // timeout
int baudrate = BAUDRATE_DEFAULT;
char* device = DEVICE_DEFAULT; // d, device
```

```
size_t packetsize = PACKETSIZE_DEFAULT; // p, packetsize
int send_filesize = PACKET_FILESIZE_DEFAULT; // filesize, no-filesize
int send_filename = PACKET_FILENAME_DEFAULT; // filename, no-filename
int my_role = DEFAULT_ROLE; // t, transmitter, r, receiver
const char* role_string = NULL;
double h_error_prob = H_ERROR_PROB_DEFAULT; // header-p
double f_error_prob = F_ERROR_PROB_DEFAULT; // frame-p
int error_type = ETYPE_DEFAULT; // error-byte, error-frame
int show_statistics = STATS_DEFAULT;

// Positional
char** files = NULL;
size_t number_of_files = 0;
// ----> END OF OPTIONS



static const struct option long_options[] = {
    // general options
    {HELP_LFLAG,                    no_argument, &show_help,               true},
    {USAGE_LFLAG,                   no_argument, &show_usage,              true},
    {VERSION_LFLAG,                 no_argument, &show_version,            true},
    {DUMP_LFLAG,                    no_argument, &dump,                    true},

    {TIME_RETRIES_LFLAG,      required_argument, NULL,          TIME_RETRIES_FLAG},
    {ANSWER_RETRIES_LFLAG,    required_argument, NULL,       ANSWER_RETRIES_FLAG},
    {TIMEOUT_LFLAG,           required_argument, NULL,              TIMEOUT_FLAG},
    {BAUDRATE_LFLAG,          required_argument, NULL,             BAUDRATE_FLAG},
    {DEVICE_LFLAG,            required_argument, NULL,               DEVICE_FLAG},
    {PACKETSIZE_LFLAG,        required_argument, NULL,           PACKETSIZE_FLAG},
    //{PACKET_FILESIZE_LFLAG,         no_argument, &send_filesize,           true},
    //{PACKET_NOFILESIZE_LFLAG,       no_argument, &send_filesize,          false},
    //{PACKET_FILENAME_LFLAG,         no_argument, &send_filename,           true},
    //{PACKET_NOFILENAME_LFLAG,       no_argument, &send_filename,          false},
    {TRANSMITTER_LFLAG,             no_argument, NULL,          TRANSMITTER_FLAG},
    {RECEIVER_LFLAG,                no_argument, NULL,             RECEIVER_FLAG},
    {HEADER_ERROR_P_LFLAG,    required_argument, NULL,       HEADER_ERROR_P_FLAG},
    {FRAME_ERROR_P_LFLAG,     required_argument, NULL,        FRAME_ERROR_P_FLAG},
    {ETYPE_BYTE_LFLAG,              no_argument, &error_type,         ETYPE_BYTE},
    {ETYPE_FRAME_LFLAG,             no_argument, &error_type,        ETYPE_FRAME},
    {NOSTATS_LFLAG,                 no_argument, &show_statistics,    STATS_NONE},
    {STATS_LFLAG,                   no_argument, &show_statistics,    STATS_LONG},
    {COMPACT_LFLAG,                 no_argument, &show_statistics, STATS_COMPACT},
    // end of options
    {0, 0, 0, 0}
    // format: {const char* lflag, int has_arg, int* flag, int val}
    //   {lflag, [no|required|optional]_argument, &var, true|false}
    // or
    //   {lflag, [no|required|optional]_argument, NULL, flag}
};

// Enforce POSIX with leading +
static const char* short_options = "Vtra:d:s:b:i:h:f:";
// x for no_argument, x: for required_argument,
// x:: for optional_argument (GNU extension),
// x; to transform  -x foo  into  --foo

static const wchar_t* version = L"FEUP RCOM 2018-2019\n"
"RCOM Comunicacao Serie v1.0\n"
"  Bruno Carvalho        up201606517\n"
"  Joao Malheiro         up201605926\n"
"  Carlos Daniel Gomes   up201603404\n"
"\n";

static const wchar_t* usage = L"usage:\n"
"                                                                  \n"
"When TRANSMITTER:                                                 \n"
"    ./ll -t [option...] files...                                  \n"
"                                                                  \n"
"When RECEIVER:                                                    \n"
"    ./ll -r [option...] number_of_files                           \n"
"                                                                  \n"
"Send one or more files through a device using a layered protocol. \n"
"                                                                  \n"
"General:                                                          \n"
"      --help,                                                     \n"
"      --usage                 Show this message and exit          \n"
"  -V, --version               Show 'version' message and exit     \n"
"      --dump                  Dump options and exit               \n"
"                                                                  \n"
"Options:                                                          \n"
"      --time=N                Write stop&wait attempts for the    \n"
"                              link-layer when timeout occurs.     \n"
"                                [Default is 5]                    \n"
"  -a, --answer=N              Write stop&wait attempts for the    \n"
"                              link-layer when an answer is invalid. \n"
"                                [Default is 100]                  \n"
"      --timeout=N             Timeout for the link-layer, in ds.  \n"
"                                [Default is 10]                   \n"
"  -b, --baudrate=N            Set the connection's baudrate.      \n"
"                              Should be equal for T and R.        \n"
"                                [Default is 115200]               \n"
"  -d, --device=S              Set the device.                     \n"
"                                [Default is /dev/ttyS0]           \n"
"  -s, --packetsize=N          Set the packets' size, in bytes.    \n"
"                              * Relevant only for the Transmitter.  \n"
```

```
"                                [Default is 1024 bytes]             \n"
"  -t, --transmitter,                                                \n"
"  -r, --receiver                Set the program's role.             \n"
"                                  [Default is Receiver]             \n"
"  -h, --header-p=P              Probability of inputing an error in \n"
"                                a read frame's header.              \n"
"                                  [Default is 0]                    \n"
"  -f, --frame-p=P               Probability of inputing an error in \n"
"                                a read frame's data.                \n"
"                                  [Default is 0]                    \n"
"                                * Relevant only for the Receiver.   \n"
"      --error-byte,                                                 \n"
"      --error-frame             Introduce the errors per-byte       \n"
"                                or per-frame.                       \n"
"                                  [Default is per-frame]            \n"
"                                Introducing errors per-byte may cause \n"
"                                corrupted messages to pass undetected,\n"
"                                corrupting the output file(s).      \n"
"      --no-stats,                                                   \n"
"      --compact,                                                    \n"
"      --stats                   Show performance statistics.        \n"
"\n";

//
// Free all resources allocated to contain options by parse_args.
//
static void clear_options() {
    free(files);
}

static void dump_options() {
    static const char* dump_string = " === Options ===\n"
    " show_help: %d             \n"
    " show_usage: %d            \n"
    " show_version: %d          \n"
    " time_retries: %d          \n"
    " answer_retries: %d        \n"
    " timeout: %d               \n"
    " baudrate: %d              \n"
    " device: %s                \n"
    " packetsize: %lu           \n"
    " my_role: %d (T=%d, R=%d) \n"
    " number_of_files: %d       \n"
    " files: 0x%08x             \n"
    " header-p: %lf             \n"
    " frame-p: %lf              \n"
    " show_statistics: %d       \n"
    "\n";

    printf(dump_string, show_help, show_usage, show_version, time_retries,
        answer_retries, timeout, baudrate, device, packetsize, my_role,
        TRANSMITTER, RECEIVER, number_of_files, files, h_error_prob,
        f_error_prob, show_statistics);

    if (files != NULL) {
        for (size_t i = 0; i < number_of_files; ++i) {
            printf(" > file#%lu: %s\n", i, files[i]);
        }
    }
}

static void exit_usage() {
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("%ls", usage);
    exit(EXIT_SUCCESS);
}

static void exit_version() {
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("%ls", version);
    exit(EXIT_SUCCESS);
}

static void exit_nofiles() {
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("[ARGS] Error: Expected 1 or more positionals (filenames), but got none.\n");
    printf("%ls", usage);
    exit(EXIT_SUCCESS);
}

static void exit_nonumber(int n) {
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("[ARGS] Error: Expected 1 positionals (number of files), but got %d.\n", n);
    printf("%ls", usage);
    exit(EXIT_SUCCESS);
}

static void exit_badpos(int n, const char* pos) {
```

```
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("[ARGS] Error: Bad positional #%d: %s.\n%ls", n, pos, usage);
    exit(EXIT_SUCCESS);
}

static void exit_badarg(const char* option) {
    if (DUMP_OPTIONS || dump) dump_options();

    setlocale(LC_ALL, "");
    printf("[ARGS] Error: Bad argument for option %s.\n%ls", option, usage);
    exit(EXIT_SUCCESS);
}

static int parse_int(const char* str, int* outp) {
    char* endp;
    long result = strtol(str, &endp, 10);

    if (endp == str || errno == ERANGE || result >= INT_MAX || result <= INT_MIN) {
        return 1;
    }

    *outp = (int)result;
    return 0;
}

static int parse_double(const char* str, double* outp) {
    char* endp;
    double result = strtod(str, &endp);

    if (endp == str || errno == ERANGE) {
        return 1;
    }

    *outp = result;
    return 0;
}

static int parse_ulong(const char* str, size_t* outp) {
    char* endp;
    long result = strtol(str, &endp, 10);

    if (endp == str || errno == ERANGE || result < 0) {
        return 1;
    }

    *outp = (size_t)result;
    return 0;
}

//
// Standard unix main's argument parsing function. Allocates resources
// that are automatically freed at exit.
//
void parse_args(int argc, char** argv) {
    // Uncomment to disable auto-generated error messages for options:
    // opterr = 0;

    atexit(clear_options);

    // Standard getopt_long Options Loop
    while (true) {
        int c, lindex = 0;

        c = getopt_long(argc, argv, short_options,
        long_options, &lindex);

        if (c == -1) break; // No more options

        switch (c) {
        case 0:
            // If this option set a flag, do nothing else now.
            if (long_options[lindex].flag == NULL) {
                // ... Long option with non-null var ...
                // getopt_long already set the flag.
                // Inside this is normally a nested switch
                // *** Access option using long_options[index].*
                // (or struct option opt = long_options[index])
                // optarg - contains value of argument
                // optarg == NULL if no argument was provided
                // to a field with optional_argument.
            }
            break;
        case HELP_FLAG:
            show_help = true;
            break;
        case VERSION_FLAG:
            show_version = true;
            break;
        case TIME_RETRIES_FLAG:
            if (parse_int(optarg, &time_retries) != 0 || time_retries <= 0) {
                exit_badarg(TIME_RETRIES_LFLAG);
            }
            break;
        case ANSWER_RETRIES_FLAG:
```

```
            if (parse_int(optarg, &answer_retries) != 0 || answer_retries <= 0) {
                exit_badarg(ANSWER_RETRIES_LFLAG);
            }
            break;
        case TIMEOUT_FLAG:
            if (parse_int(optarg, &timeout) != 0 || timeout <= 0) {
                exit_badarg(TIMEOUT_LFLAG);
            }
            break;
        case BAUDRATE_FLAG:
            if (parse_int(optarg, &baudrate) != 0 || !is_valid_baudrate(baudrate)) {
                exit_badarg(BAUDRATE_LFLAG);
            }
            break;
        case DEVICE_FLAG:
            device = optarg;
            break;
        case PACKETSIZE_FLAG:
            if (parse_ulong(optarg, &packetsize) != 0 || packetsize == 0) {
                exit_badarg(PACKETSIZE_LFLAG);
            }
            break;
        case TRANSMITTER_FLAG:
            my_role = TRANSMITTER;
            break;
        case RECEIVER_FLAG:
            my_role = RECEIVER;
            break;
        case HEADER_ERROR_P_FLAG:
            if (parse_double(optarg,&h_error_prob) != 0
            || h_error_prob < 0 || h_error_prob >= 1) {
                exit_badarg(HEADER_ERROR_P_LFLAG);
            }
            break;
        case FRAME_ERROR_P_FLAG:
            if (parse_double(optarg,&f_error_prob) != 0
            || f_error_prob < 0 || f_error_prob >= 1) {
                exit_badarg(FRAME_ERROR_P_LFLAG);
            }
            break;
        case '?':
        default:
            // getopt_long already printed an error message.
            exit_usage();
        }
    } // End Options loop

    if (show_help || show_usage) {
        exit_usage();
    }

    if (show_version) {
        exit_version();
    }

    role_string = my_role == TRANSMITTER ? "Transmitter" : "Receiver";

    // Positional arguments processing
    switch (my_role) {
    case TRANSMITTER:
        if (optind == argc) {
            exit_nofiles();
        }

        number_of_files = argc - optind;
        files = malloc(number_of_files * sizeof(char*));

        for (size_t i = 0; i < number_of_files; ++i, ++optind) {
            files[i] = argv[optind];
        }

        break;
    case RECEIVER:
        if (optind + 1 != argc) {
            exit_nonumber(argc - optind);
        }

        if (parse_ulong(argv[optind++], &number_of_files) != 0 || number_of_files == 0) {
            exit_badpos(1, argv[argc - 1]);
        }

        break;
    }

    if (DUMP_OPTIONS || dump) dump_options();
    if (dump) exit(EXIT_SUCCESS);
}
```

options.h

```
#ifndef OPTIONS_H___
#define OPTIONS_H___

#include <stddef.h>
#include <stdbool.h>
```

```
// NOTE: All externs are resolved in options.c
// Too add/edit an option, do:
//      1. Add a block entry here
//      2. Resolve the extern in options.c
//      3. Update short_options and long_options in options.c
//      4. Update the usage string in options.c
//      5. Update parse_args() in options.c
//      6. Update dump_options() in options.c
// then go on to add the option's functionality...


// <!--- GENERAL OPTIONS
// Show help/usage message and exit
#define HELP_FLAG '0'
#define HELP_LFLAG "help"

// Show help/usage message and exit
#define USAGE_FLAG // NONE
#define USAGE_LFLAG "usage"

// Show version message and exit
#define VERSION_FLAG 'V'
#define VERSION_LFLAG "version"

// Dump options and exit
#define DUMP_FLAG // none
#define DUMP_LFLAG "dump"
// ----> END OF GENERAL OPTIONS


// <!--- OPTIONS
// Set number of send retries for link-layer communications, when the
// receiver does not acknowledge the frame.
#define TIME_RETRIES_FLAG '1'
#define TIME_RETRIES_LFLAG "time"
#define TIME_RETRIES_DEFAULT 5
extern int time_retries;

#define ANSWER_RETRIES_FLAG 'a'
#define ANSWER_RETRIES_LFLAG "answer"
#define ANSWER_RETRIES_DEFAULT 100
extern int answer_retries;

// Set timeout in deciseconds for link-layer communications.
#define TIMEOUT_FLAG '2'
#define TIMEOUT_LFLAG "timeout"
#define TIMEOUT_DEFAULT 10
extern int timeout;

// Set device (presumably serial port) to use.
#define DEVICE_FLAG 'd'
#define DEVICE_LFLAG "device"
#define DEVICE_DEFAULT "/dev/ttyS0"
extern char* device;

// Set packet size, in bytes
#define PACKETSIZE_FLAG 's'
#define PACKETSIZE_LFLAG "packetsize"
#define PACKETSIZE_DEFAULT 1024
extern size_t packetsize;

// Set baudrate
#define BAUDRATE_FLAG 'b'
#define BAUDRATE_LFLAG "baudrate"
#define BAUDRATE_DEFAULT 115200
extern int baudrate;

// Send or do not send filesize in START packet
#define PACKET_FILESIZE_FLAG // none
#define PACKET_FILESIZE_LFLAG "filesize"
#define PACKET_NOFILESIZE_LFLAG "no-filesize"
#define PACKET_FILESIZE_DEFAULT true
extern int send_filesize; // NOT IMPLEMENTED

// Send or do not send filename in START packet
#define PACKET_FILENAME_FLAG // none
#define PACKET_FILENAME_LFLAG "filename"
#define PACKET_NOFILENAME_LFLAG "no-filename"
#define PACKET_FILENAME_DEFAULT true
extern int send_filename; // NOT IMPLEMENTED

#define TRANSMITTER_FLAG 't'
#define TRANSMITTER_LFLAG "transmitter"
#define TRANSMITTER 1
#define RECEIVER_FLAG 'r'
#define RECEIVER_LFLAG "receiver"
#define RECEIVER 2
#define DEFAULT_ROLE RECEIVER
extern int my_role;
extern const char* role_string;

#define HEADER_ERROR_P_FLAG 'h'
#define FRAME_ERROR_P_FLAG 'f'
```

```
#define HEADER_ERROR_P_LFLAG "header-p"
#define FRAME_ERROR_P_LFLAG "frame-p"
#define H_ERROR_PROB_DEFAULT 0.0
#define F_ERROR_PROB_DEFAULT 0.0
extern double h_error_prob;
extern double f_error_prob;

#define ETYPE_FLAG // none
#define ETYPE_BYTE_LFLAG "error-byte"
#define ETYPE_FRAME_LFLAG "error-frame"
#define ETYPE_BYTE 0x71
#define ETYPE_FRAME 0x72
#define ETYPE_DEFAULT ETYPE_FRAME
extern int error_type;

#define STATS_FLAG '3'
#define NOSTATS_LFLAG "no-stats"
#define STATS_LFLAG "stats"
#define COMPACT_LFLAG "compact"
#define STATS_NONE 0
#define STATS_LONG 1
#define STATS_COMPACT 2
#define STATS_DEFAULT STATS_NONE
extern int show_statistics;
// ----> END OF OPTIONS

// <!--- POSITIONAL
extern char** files;

extern size_t number_of_files;
// ----> END POSITIONAL

void parse_args(int argc, char** argv);

#endif // OPTIONS_H___
```

signals.c

```
#include "signals.h"
#include "options.h"
#include "debug.h"

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdbool.h>
#include <sys/time.h>

#define ALARM_TEST_SET           2500
#define ALARM_TEST_SHORT_SLEEP   500
#define ALARM_TEST_LONG_SLEEP    8500

static const char str_kill[]  = "[SIG] -- Terminating...\n";
static const char str_abort[] = "[SIG] -- Aborting...\n";
static const char str_alarm[] = "[SIG] -- Alarmed...\n";

static volatile sig_atomic_t alarmed = 0;

// SIGHUP, SIGQUIT, SIGTERM, SIGINT
static void sighandler_kill(int signum) {
    if (TRACE_SIG) write(STDOUT_FILENO, str_kill, strlen(str_kill));
    exit(EXIT_FAILURE);
}

// SIGABRT
static void sighandler_abort(int signum) {
    if (TRACE_SIG) write(STDOUT_FILENO, str_abort, strlen(str_abort));
    abort();
}

// SIGALRM
static void sighandler_alarm(int signum) {
    if (TRACE_SIG) write(STDOUT_FILENO, str_alarm, strlen(str_alarm));
    alarmed = 1;
}

//
// Set the process's signal handlers and overall dispositions
//
int set_signal_handlers() {
    struct sigaction action;
    sigset_t sigmask, current;

    // Get current sigmask
    int s = sigprocmask(SIG_SETMASK, NULL, &current);
    if (s != 0) {
        printf("[SIG] Error getting process signal mask: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    // Set kill handler
    sigmask = current;
```

```
    action.sa_handler = sighandler_kill;
    action.sa_mask = sigmask;
    action.sa_flags = SA_RESETHAND;
    s = sigaction(SIGHUP, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGHUP: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    s = sigaction(SIGQUIT, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGQUIT: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    s = sigaction(SIGTERM, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGTERM: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    s = sigaction(SIGINT, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGINT: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    // Set abort handler
    sigmask = current;
    action.sa_handler = sighandler_abort;
    action.sa_mask = sigmask;
    action.sa_flags = SA_RESETHAND;
    s = sigaction(SIGABRT, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGABRT: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    // Set alarm handler
    sigmask = current;
    action.sa_handler = sighandler_alarm;
    action.sa_mask = sigmask;
    action.sa_flags = 0;
    s = sigaction(SIGALRM, &action, NULL);
    if (s != 0) {
        printf("[SIG] Error setting handler for SIGALRM: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    if (TRACE_SETUP) printf("[SIG] Set all signal handlers\n");
    return 0;
}

static void set_alarm_general(unsigned long us) {
    static const unsigned long mill = 1000000lu;
    const struct itimerval new_value = {
        .it_interval = {0, 0},
        .it_value = {us / mill, us % mill}
    };

    setitimer(ITIMER_REAL, &new_value, NULL);
    alarmed = 0;
}

void set_alarm() {
    set_alarm_general((unsigned long)timeout * 100000);
}

void unset_alarm() {
    set_alarm_general(0lu);
}

bool was_alarmed() {
    bool r = alarmed ? true : false;
    alarmed = 0;
    return r;
}

void test_alarm() {
    int s;
    errno = 0;

    // Short sleep test
    set_alarm_general(ALARM_TEST_SET);

    s = usleep(ALARM_TEST_SHORT_SLEEP);
    if (s != 0) {
        printf("[ALARM] Failed test_alarm() -- short sleep interrupted\n");
        exit(EXIT_FAILURE);
    }

    set_alarm_general(ALARM_TEST_SET);

    s = usleep(ALARM_TEST_LONG_SLEEP);
    if (s == 0 || errno != EINTR) {
        printf("[ALARM] Failed test_alarm() -- long sleep not interrupted\n");
        exit(EXIT_FAILURE);
    }
```

```
    unset_alarm();
    errno = 0;

    if (TRACE_SETUP) printf("[SETUP] Passed test_alarm()\n");
}

void await_timeout() {
    if (my_role == TRANSMITTER) {
        usleep(160000 * timeout);
    } else {
        usleep(60000 * timeout);
    }
}
```

signals.h

```
#ifndef SIGNALS_H___
#define SIGNALS_H___

#include <stdbool.h>

int set_signal_handlers();

void set_alarm();

void unset_alarm();

bool was_alarmed();

void test_alarm();

void await_timeout();

#endif // SIGNALS_H___
```

strings.c

```
#include "strings.h"

#include <string.h>
#include <stdio.h>

string string_from(char* str) {
    string s = {str, strlen(str)};
    return s;
}

void print_string(string str) {
    printf("  [len=%lu]: ", str.len);
    fwrite(str.s, str.len, 1, stdout);
}

void print_stringn(string str) {
    printf("  [len=%lu]: ", str.len);
    fwrite(str.s, str.len, 1, stdout);
    printf("\n");
}
```

strings.h

```
#ifndef STRINGS_H___
#define STRINGS_H___

#include <stddef.h>

typedef struct {
    char* s;
    size_t len;
} string;

string string_from(char* str);

void print_string(string str);

void print_stringn(string str);

#endif // STRINGS_H___
```

timing.c

```
#include "timing.h"
#include "ll-errors.h"
#include "debug.h"
#include "options.h"

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>

static struct timespec timestamp[3];
static double times[3];
```

```
size_t number_of_packets(size_t filesize) {
    return (filesize + packetsize - 1) / packetsize;
}

double average_packetsize(size_t filesize) {
    return (double)filesize / number_of_packets(filesize);
}

static void print_stats_compact(size_t i, size_t filesize) {
    static const char* stats_string = "[STATS %s]\n"
    "==STATS==  %.5lf seconds                        \n"
    "==STATS==  %9.2f Bits/s                         \n"
    "==STATS==  %9.2f Bytes/s                        \n"
    "==STATS==  %9.2f Packs/s                        \n"
    "==STATS==   %6d Timeouts                        \n"
    "==STATS==   %6d I | %d RR | %d REJ              \n"
    "==STATS==   %6d Invalid | %d BCC1 | %d BCC2     \n"
    "==STATS==\n";

    double ms = times[i];
    double s = ms / 1000.0;

    // Observed
    double obs_bits = 8.0 * filesize / s;
    double obs_bytes = filesize / s;
    double obs_packs = obs_bytes / average_packetsize(filesize);

    if (my_role == TRANSMITTER) {
        printf(stats_string, role_string,
        s,
        obs_bits,
        obs_bytes,
        obs_packs,
        counter.timeout,
        counter.out.I[0] + counter.out.I[1],
        counter.in.RR[0] + counter.in.RR[1],
        counter.in.REJ[0] + counter.in.REJ[1],
        counter.invalid,
        counter.read.bcc1,
        counter.read.bcc2);
    } else {
        printf(stats_string, role_string,
            s,
            obs_bits,
            obs_bytes,
            obs_packs,
            counter.timeout,
            counter.in.I[0] + counter.in.I[1],
            counter.out.RR[0] + counter.out.RR[1],
            counter.out.REJ[0] + counter.out.REJ[1],
            counter.invalid,
            counter.read.bcc1,
            counter.read.bcc2);
    }
}

static void print_stats_receiver(size_t i, size_t filesize) {
    static const char* stats_string = "[STATISTICS RECEIVER]\n"
    "==STATS==  Total Time:  %.5lf seconds                \n"
    "==STATS==  Error probabilities:                     \n"
    "==STATS==    header-p      %7.5f                    \n"
    "==STATS==    frame-p       %7.5f                    \n"
    "==STATS==    FER I frames  %7.5f                    \n"
    "==STATS==  Communication:                           \n"
    "==STATS==    Baudrate     %7d Symbols/s             \n"
    "==STATS==    Packet Size  %7d bytes (average)       \n"
    "==STATS==    I frame Size %7d bytes (average)       \n"
    "==STATS==    %6d Data packets (+2 total)            \n"
    "==STATS==  Efficiency:                              \n"
    "==STATS==     Observed |  Max (b)  |  Max (B)       \n"
    "==STATS==    %9.2f | %9.2f | %9.2f Bits/s          \n"
    "==STATS==    %9.2f | %9.2f | %9.2f Bytes/s         \n"
    "==STATS==    %9.3f | %9.3f | %9.3f Packs/s         \n"
    "==STATS==    %6d Timeouts                           \n"
    "==STATS==  Frames Received:                         \n"
    "==STATS==    %6d I    | %6d I0    | %6d I1         \n"
    "==STATS==    %6d Invalid or unexpected              \n"
    "==STATS==  Frames Transmitted:                      \n"
    "==STATS==    %6d RR   | %6d RR0   | %6d RR1        \n"
    "==STATS==    %6d REJ  | %6d REJ0  | %6d REJ1       \n"
    "==STATS==  Reading Errors:                          \n"
    "==STATS==    %6d Bad frame length                   \n"
    "==STATS==    %6d Bad BCC1                           \n"
    "==STATS==    %6d Bad BCC2                           \n"
    "==STATS==\n";

    double ms = times[i];
    double s = ms / 1000.0;
    double h = h_error_prob, f = f_error_prob;

    double ferI = h + f - (h * f);
    int numpackets = number_of_packets(filesize);
    int average = average_packetsize(filesize);
    int frameIsize = 10 + average;
```

```
    // Observed
    double obs_bits = 8.0 * filesize / s;
    double obs_bytes = filesize / s;
    double obs_packs = obs_bytes / average_packetsize(filesize);

    // Maximum
    double max_bits = baudrate;
    double max_bytes = baudrate / 8.0;
    double max_packs = max_bytes / average_packetsize(filesize);

    printf(stats_string,
        s, h, f, ferI,
        baudrate,
        average,
        frameIsize,
        numpackets,
        obs_bits, max_bits, max_bits * 8.0,
        obs_bytes, max_bytes, max_bytes * 8.0,
        obs_packs, max_packs, max_packs * 8.0,
        counter.timeout,
        counter.in.I[0] + counter.in.I[1],
        counter.in.I[0], counter.in.I[1],
        counter.invalid,
        counter.out.RR[0] + counter.out.RR[1],
        counter.out.RR[0], counter.out.RR[1],
        counter.out.REJ[0] + counter.out.REJ[1],
        counter.out.REJ[0], counter.out.REJ[1],
        counter.read.len,
        counter.read.bcc1,
        counter.read.bcc2);
}

static void print_stats_transmitter(size_t i, size_t filesize) {
    static const char* stats_string = "[STATISTICS TRANSMITTER]\n"
    "==STATS==  Total Time:  %.5lf seconds                \n"
    "==STATS==  Error probabilities:                      \n"
    "==STATS==    header-p      %7.5f                     \n"
    "==STATS==  Communication:                            \n"
    "==STATS==    Baudrate     %7d Symbols/s              \n"
    "==STATS==    Packet Size  %7d bytes (average)        \n"
    "==STATS==    I frame Size %7d bytes (average)        \n"
    "==STATS==    %6d Data packets (+2 total)             \n"
    "==STATS==  Efficiency:                               \n"
    "==STATS==     Observed |  Max (b)  |  Max (B)        \n"
    "==STATS==    %9.2f | %9.2f | %9.2f Bits/s           \n"
    "==STATS==    %9.2f | %9.2f | %9.2f Bytes/s          \n"
    "==STATS==    %9.3f | %9.3f | %9.3f Packs/s          \n"
    "==STATS==    %6d Timeouts                            \n"
    "==STATS==  Frames Transmitted:                       \n"
    "==STATS==    %6d I    | %6d I0    | %6d I1          \n"
    "==STATS==  Frames Received:                          \n"
    "==STATS==    %6d RR   | %6d RR0   | %6d RR1         \n"
    "==STATS==    %6d REJ  | %6d REJ0  | %6d REJ1        \n"
    "==STATS==    %6d Invalid or unexpected               \n"
    "==STATS==  Reading Errors:                           \n"
    "==STATS==    %6d Bad frame length                    \n"
    "==STATS==    %6d Bad BCC1                            \n"
    "==STATS==    %6d Bad BCC2                            \n"
    "==STATS==\n";

    double ms = times[i];
    double s = ms / 1000.0;
    double h = h_error_prob;

    int numpackets = number_of_packets(filesize);
    int average = average_packetsize(filesize);
    int frameIsize = 10 + average;

    // Observed
    double obs_bits = 8.0 * filesize / s;
    double obs_bytes = filesize / s;
    double obs_packs = obs_bytes / average_packetsize(filesize);

    // Maximum
    double max_bits = baudrate;
    double max_bytes = baudrate / 8.0;
    double max_packs = max_bytes / average_packetsize(filesize);

    printf(stats_string,
        s, h,
        baudrate,
        average,
        frameIsize,
        numpackets,
        obs_bits, max_bits, max_bits * 8.0,
        obs_bytes, max_bytes, max_bytes * 8.0,
        obs_packs, max_packs, max_packs * 8.0,
        counter.timeout,
        counter.out.I[0] + counter.out.I[1],
        counter.out.I[0], counter.out.I[1],
        counter.in.RR[0] + counter.in.RR[1],
        counter.in.RR[0], counter.in.RR[1],
        counter.in.REJ[0] + counter.in.REJ[1],
        counter.in.REJ[0], counter.in.REJ[1],
        counter.invalid,
        counter.read.len,
```

```
        counter.read.bcc1,
        counter.read.bcc2);
}

void print_stats(size_t i, size_t filesize) {
    if (show_statistics == STATS_COMPACT) {
        print_stats_compact(i, filesize);
    } else if (show_statistics == STATS_LONG) {
        if (my_role == RECEIVER) {
            print_stats_receiver(i, filesize);
        } else {
            print_stats_transmitter(i, filesize);
        }
    }
}

void begin_timing(size_t i) {
    if (TRACE_TIME) {
        printf("[TIME] START Timing [%lu]\n", i);
    }

    times[i] = 0.0;
    clock_gettime(CLOCK_MONOTONIC, &timestamp[i]);
}

void end_timing(size_t i) {
    struct timespec end;
    clock_gettime(CLOCK_MONOTONIC, &end);

    if (TRACE_TIME) {
        printf("[TIME] END timing [%lu]\n", i);
    }

    double ts = (end.tv_sec - timestamp[i].tv_sec) * 1e3;
    double tns = (end.tv_nsec - timestamp[i].tv_nsec) / 1e6;

    times[i] = ts + tns;
    timestamp[i].tv_sec = 0; timestamp[i].tv_nsec = 0;

    if (TRACE_TIME) {
        printf("[STATS] Time [%lu] [ms=%.1lf]\n", i, times[i]);
    }
}
```

timing.h

```
#ifndef TIMING_H___
#define TIMING_H___

#include <stddef.h>

size_t number_of_packets(size_t filesize);

double average_packetsize(size_t filesize);

void print_stats(size_t i, size_t filesize);

void begin_timing(size_t i);

void end_timing(size_t i);

#endif // TIMING_H___
```

Makefile

```
.PHONY: all clean createbin debug

CC := gcc

SRC_DIR := src
OBJ_DIR := bin
OUT_DIR := .

SRC_FIL := $(wildcard $(SRC_DIR)/*.c)
SRC_OBJ := $(patsubst $(SRC_DIR)/%.c,$(OBJ_DIR)/%.o,$(SRC_FIL))

OBJECTS := $(SRC_OBJ)

OUT := $(OUT_DIR)/ll

CFLAGS := -std=gnu11 -Wall -Wextra -march=native -g
CFLAGS += -Wno-switch -Wno-unused-result -Wno-unused-parameter -Wno-unused-function
```

```
LIBS :=
INCLUDE := -I $(SRC_DIR)



all: clean createbin $(OBJECTS)
    $(CC) $(CFLAGS) $(INCLUDE) -o $(OUT) $(OBJECTS) $(LIBS)

createbin:
    @mkdir -p bin

$(SRC_OBJ): $(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@ $(INCLUDE)

$(GVW_OBJ): $(OBJ_DIR)/%.o: $(GVW_DIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@ $(INCLUDE)

clean:
    @rm -f $(OBJECTS) $(OUT)
```