U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Ligação de Dados

Redes de Computadores

Bruno Carvalho
up201606517

João Malheiro
up201605926

Carlos Daniel Gomes
up201603404

November 7, 2018

## Contents

## Architecture

We will now 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. We will also discuss some parallel features, such as terminal setup, program options, signal handlers, execution timing and forced error introduction.

### String

Given that the byte arrays used throughout the program are *not* `NULL`-terminated strings — but rather 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's core 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.

#### Functions

The core has the following 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`

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`*. The control packet holds a sequence of *`tlv`*, which are type/value pairs as described in the introduction. The data packet holds information for both the packet's header field and data field.

### 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 control and data packets using llwrite and llread: `send_data_packet`, `send_start_packet`, `send_end_packet`
- Receive (any) packet using llwrite and 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`
- Compute and print statistics about error probabilities, communication speed and efficiency: `print_stats`
- Introduce flip bits in read frames: `introduceErrors`

## Use cases and control flow

### Top level

Setup in function `main` includes parsing and validating program options — `parse_args` —, setting up signal handlers — `set_signal_handlers` — testing the system's alarms — `test_alarm` — and adjusting the terminal configuration (namely noncanonical mode) — `setup_link_layer`. This is the same for $R$ and for $T$.

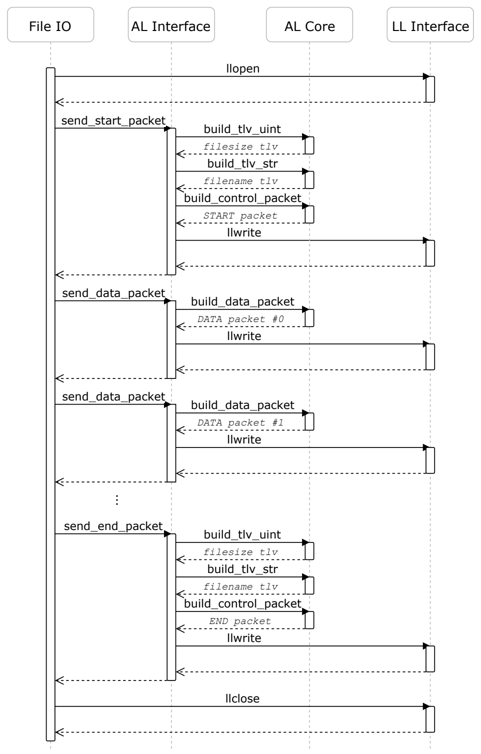
In function `send_file`, called by $T$, the selected file is opened, its size is calculated and it is then read into a single *buffer* and closed. The *buffer* is then split into multiple *packets* with length *`packetsize`* — each stored in a *`string`* — and then freed. These packets are then sent 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.

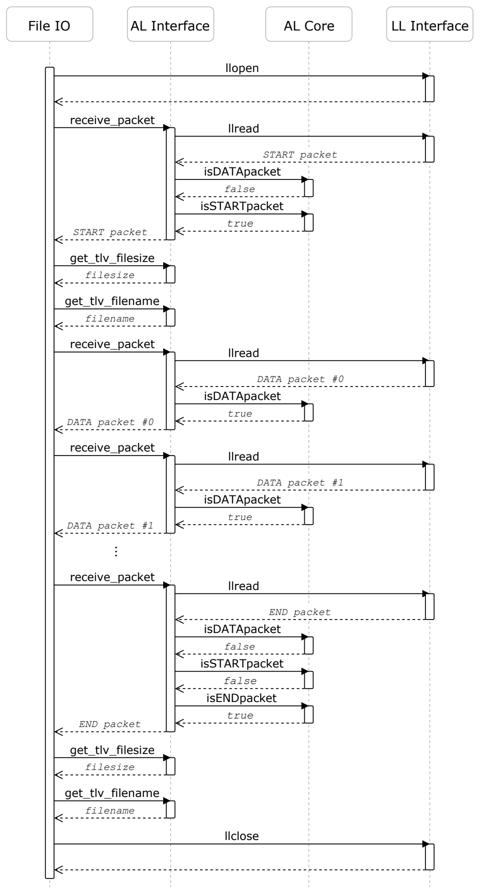
Neither of these functions deal with any $AL$ errors. At the end of `main`, the terminal settings are reset with `reset_link_layer`.

Simplified sequence diagrams for functions `llread` and `llwrite` can be found in Figure 1c and Figure 1d respectively.

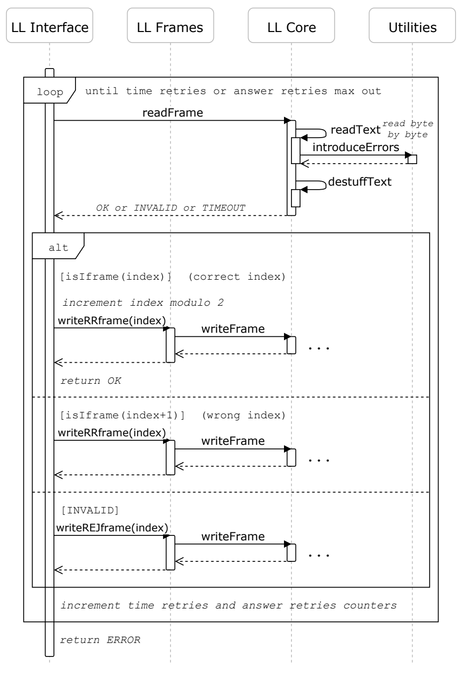
Figure 1: Sequence diagrams for some important functions



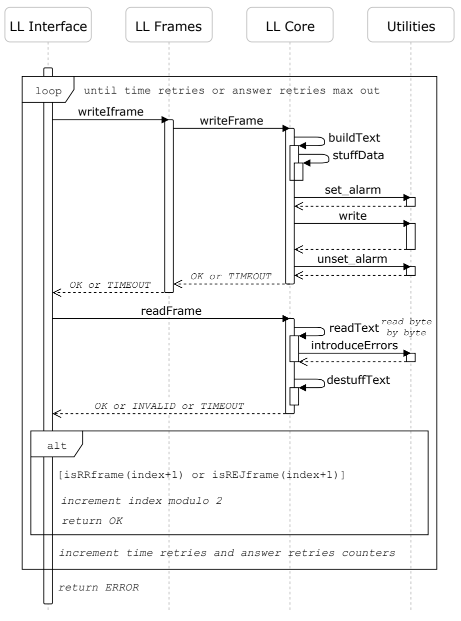
(a) File communications as seen through the app layer (part of function `send_file`)



(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, both in 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 read 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;
}
```

```c
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, parsing 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 divided. 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`*.

```c
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) {
        // [...] Report result
        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);

    // [...] Report result

    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;
}


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;
}
```