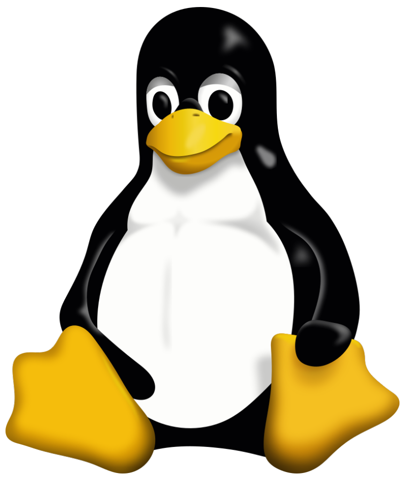
Faculty of Engineering of the University of Porto

1st Project Report

Computer Networks



Patrícia de Sousa, Tomás Sucena Lopes

Bachelor in Informatics and Computing Engineering

Teacher: Prof. João Soares

December 8, 2024

Abstract

This report was written for the Computer Networks course, which is part of the Bachelor in Informatics and Computing Engineering at the Faculty of Engineering of the University of Porto (FEUP). It details the development of the course's first project: a file-transfer application that relies on two communication protocols for transmitting data.

Both protocols were aptly implemented whilst adhering to the proposed specification. In addition, the application meets all functional requirements. Therefore, the project was accomplished without issues.

Contents

Chapter 1

Introduction

The objective of this project was the development of a command-line application for transferring files between two computers connected by an RS-232 serial port. For reliably transmitting data, the application would implement a *Stop&Wait*[1] logical link protocol and an application protocol.

This report details the implementation of the application as well as its underlying protocols. It is divided into the following sections:

- **Architecture:** Details the application's functional blocks and interfaces by exploring the different layers that constitute it, as well as their implementation in code.
- **Protocols:** Summarizes the main functional aspects of the data transmission protocols the application relies on.
- **Validation:** Describes the tests that were performed to verify the correctness of the application and statistically characterize its efficiency.
- **Conclusion:** Synthesizes the information presented in the previous sections, reflecting on the learning objectives achieved.

Disclaimer

We are aware that we exceeded the maximum number of pages that was stipulated for this report. However, we believe that the added explanations in some sections in conjunction with the use of spacing for a more pleasant reading experience justify the extra pages. We humbly ask for your understanding regarding this decision.

[1]https://www.geeksforgeeks.org/stop-and-wait-arq/

Chapter 2

Architecture

The application adopts a layered architecture[1] that mirrors a condensed version of the TCP/IP and OSI reference models (Figure 2.1).

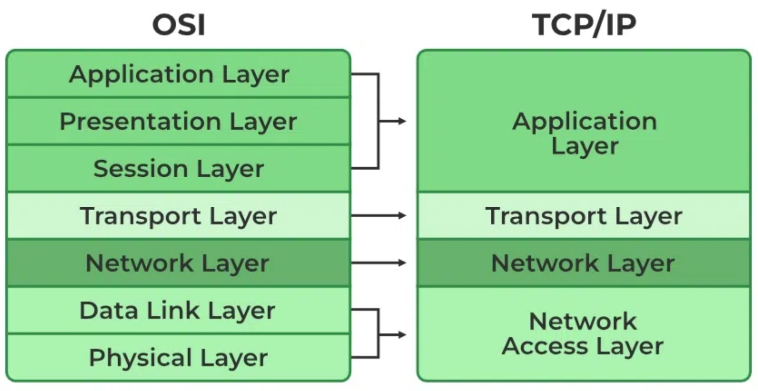


Figure 2.1: The OSI and TCP/IP reference models

Instead of the usual seven layers, only three are used: the Application, Data-Link, and Physical Layers. Each layer exposes its services by providing a consistent API to the layer immediately above it. Similarly, it utilizes the services of the layer immediately below it by calling its API.

This architecture respects the **layer independence principle**, which states that no layer should depend on the implementation of any other. Each layer only knows how to access the services of the layer directly beneath it, without being aware of its inner workings. This design pattern ensures that, if any of the layers need to be modified but their API remains the same, no other layer needs to be altered, thus streamlining the development process.

2.1 Coding standard

This project was developed in the C programming language. Despite it being a procedural language, we opted to follow an object-oriented approach to software development and use objects to model the program's state.

In object-oriented programming languages, objects are created using **constructors**, which are special functions that configure the object's properties. Upon being created, objects can then be used to store

[1] https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

data and to call its **methods**, that is, functions that are associated with an object and run within its context (Sheldon 2023). Finally, when the object's lifecycle ends, another special function - the **destructor** - is invoked, usually automatically by the compiler, that clears the memory allocated for the object. Listing 2.1 depicts a Java code snippet that highlights typical object-oriented code.

```java
LinkLayer ll = new LinkLayer(); // constructor
ll.open(); // method
// the destructor is called automatically here
```

Listing 2.1: Example of object-oriented code in Java

In C, however, these features are not built into the language, so we had to mimic them. Instead of objects, we used `structs` to group properties. In addition, **methods**, including constructors and destructors, were implemented as functions that take the object they belong to as their first parameter. Listing 2.2 illustrates the coding standard we adopted.

```c
LinkLayer *ll = llInit(); // constructor
llOpen(ll); // method
llFree(ll); // destructor
```

Listing 2.2: Example of object-oriented code in C

2.2 Layers

2.2.1 Physical Layer

The Physical Layer is the bottommost layer and is responsible for physically transferring bits via the serial port. It is the only layer that directly interacts with hardware.

Files

- **`serial_port.h`** - Defines the Physical Layer's data structures and functions.
- **`serial_port.c`** - Implements the Physical Layer's functions.

Data structures

- **`SerialPort`** - A structure that encapsulates important variables for managing the Physical Layer.

```c
/**
 * @brief A struct that represents the serial port.
 */
typedef struct {
    int fd;                         /** the file descriptor of the serial port */
    int baudRate;                   /** the baud rate */
    struct termios oldSettings; /** the original serial port settings (to be restored) */
} SerialPort;
```

Listing 2.3: Code definition of the `SerialPort` data structure

API

- **spInit()** - Initializes and configures the serial port. Returns a reference to the created object on success, otherwise, a null pointer.
- **spFree()** - Restores the original serial port settings and frees the memory allocated by the object.
- **spWrite()** - Writes a stream of bytes to the serial port using `write()`[2] from the C standard library.
- **spRead()** - Reads a byte from the serial port using `read()`[3] from the C standard library.

```c
/* API */
SerialPort *spInit(const char *filename, int baudRate);
int spFree(SerialPort *sp);

int spWrite(SerialPort *sp, const unsigned char *bytes, int numBytes);
int spRead(SerialPort *sp, unsigned char *byte);
```

Listing 2.4: Code definition of the serial port API

2.2.2 Data-Link Layer

The Data-Link Layer sits in between the Application and Physical Layers and provides an interface for transmitting data between computers. It is responsible for establishing the connection between the sender and receiver, preparing and processing the frames that will encapsulate the transmitted data, and, finally, terminating the connection.

Files

- **link_layer.h** - Defines the data structures and functions of the Data-Link Layer.
- **link_layer.c** - Implements the functions of the Data-Link Layer.

Data structures

- **LinkLayer** - A structure that encapsulates important variables for managing the Data-Link Layer and providing a statistical analysis of the logical link protocol.

```c
typedef struct {
    SerialPort *sp;          /** the serial port */
    _Bool isSender;          /** indicates whether the program is the sender or receiver */
    int maxRetransmissions; /** the maximum number of times an individual frame can be
    ↪   retransmitted */
    int timeout;             /** the number of seconds before a timeout occurs */
    int iFrames;             /** the number of I-frames sent/received */
    FILE *logbook;           /** the file to which logs will be output */

    // for statistical purposes
    struct timeval startTime;      /** the time when the program starts */
    int numFramesTransmitted;      /** the number of frames transmitted */
```

[2] https://man7.org/linux/man-pages/man2/write.2.html
[3] https://man7.org/linux/man-pages/man2/read.2.html

```
    int numFramesRetransmitted;   /** the number of frames retransmitted */
    int numFramesReceived;        /** the number of frames received */
    int numFramesRejected;        /** the number of frames rejected */
    int numTimeouts;              /** the number of timeouts that occurred */
    long numDataBytesTransferred; /** the total number of data bytes sent/received */
} LinkLayer;
```

Listing 2.5: Code definition of the `LinkLayer` structure

API

- **`llInit()`** - Initializes and configures a `LinkLayer` instance. Returns a reference to the created object on success, otherwise, a null pointer.
- **`llFree()`** - Frees the memory allocated by the `LinkLayer` object.
- **`llOpen()`** - Establishes the connection with the other computer.
- **`llWrite()`** - Encapsulates data in a frame and sends it via the serial port.
- **`llRead()`** - Receives a frame from the serial port and retrieves its data.
- **`llClose()`** - Terminates the connection with the other computer and, optionally, prints a brief statistical analysis of the file-transferring session.

```
/* API */
LinkLayer *llInit(const char *serialPort, const char *role, int baudRate, int
↪  maxRetransmissions, int timeout);
int llFree(LinkLayer *ll);

int llOpen(LinkLayer *ll);
int llWrite(LinkLayer *ll, const unsigned char *packet, int packetSize);
int llRead(LinkLayer *ll, unsigned char *packet);
int llClose(LinkLayer *ll, _Bool showStatistics);
```

Listing 2.6: Code definition of the Data-Link Layer API

2.2.3 Application Layer

The Application Layer is the topmost layer and is the only one to directly interface with the end-user. It handles the file-transferring process: if it is the sender, it fragments the file into data packets and oversees their transmission, otherwise, it receives said packets and ensures the original file can be properly reconstructed.

Files

- **`application_layer.h`** - Defines the data structures and functions of the Application Layer.
- **`application_layer.c`** - Implements the functions of the Application Layer.

Data structures

- **`ApplicationLayer` -** A structure that encapsulates important variables for managing the Application Layer.

```
typedef struct {
    LinkLayer *ll;  /** the data-link layer */
    FILE *file;     /** pointer to the file to be sent/received */
    char *filename; /** the name of the file to be sent/received */
    long fileSize;  /** the size of the file to be sent/received */
    int dataSize;   /** the maximum number of data bytes of a data packet */
} ApplicationLayer;
```

Listing 2.7: Code definition of the `ApplicationLayer` structure

API

- **`appInit()` -** Initializes and configures the Application Layer. It also triggers the initialization of all layers beneath it by invoking `llInit()`. Returns a reference to an `ApplicationLayer` instance on success, otherwise, a null pointer.
- **`appFree()` -** Frees the memory allocated to the Application Layer. It also clears the memory occupied by all layers beneath it by invoking `llFree()`.
- **`appRun()` -** Executes the file-transfer application: if the program is the sender, it sends the file, or else receives it.

```
/* API */
ApplicationLayer *appInit(const char *serialPort, const char *role, int baudRate, int
↪  maxRetransmissions, int timeout, const char *filepath, int dataSize);
int appFree(ApplicationLayer *app);

// Application layer main function
int appRun(ApplicationLayer *app);
```

Listing 2.8: Code definition of the Application Layer API

2.3 Use cases

Considering our application is a command-line interface (CLI), it is executed by typing the name of its executable followed by its arguments, as Listing 2.9 demonstrates.

Note: Arguments wrapped around square brackets ([]) are optional.

```
$ bin/main /dev/ttySxx <baud rate> tx <filepath> [data size] # sender
$ bin/main /dev/ttySxx <baud rate> rx [filepath] # receiver
```

Listing 2.9: Basic application usage

Figure 2.2 depicts typical error-free program execution from within the Data-Link Layer.

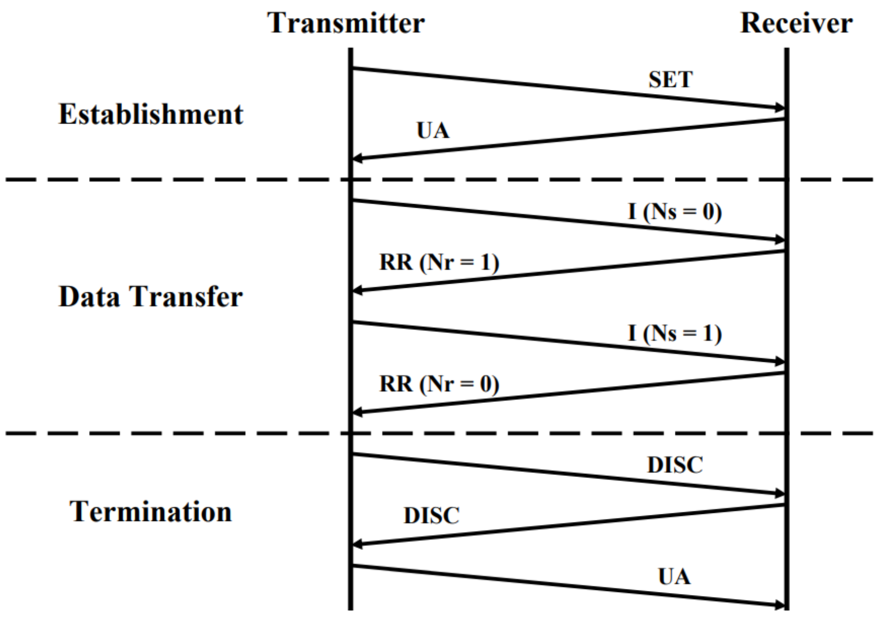


Figure 2.2: Program execution in the Data-Link Layer, without errors

The following section enumerates the program execution steps from the perspectives of the sender and receiver.

2.3.1 Sender

1. Initializes the application by calling `appInit()`.
2. Invokes `appRun()` to begin the file-transferring process.
3. Calls `llOpen()`, where it sends a SET frame to the receiver and awaits a UA frame as its acknowledgment. If the SET and UA frames are correctly transmitted and received, respectively, the connection has been established.
4. Sends a control packet with `llWrite()`. Said packet contains the **name** and **size** of the file to be transferred, as well as the maximum number of **data bytes** that will be sent in each data packet.
5. Enters the file-transferring loop. Until the file has not been completely transmitted:
   (a) Reads a stream of bytes no larger than the agreed-upon maximum number of data bytes from the file. This is achieved with `fread()`[4] from the C standard library
   (b) Inserts the bytes into a data packet.
   (c) Uses `llWrite()` to send the packet to the receiver and confirm its reception/rejection. In the case of a rejection, the packet is retransmitted until the maximum number of retransmissions is reached.
6. Sends another control packet using `llWrite()`, this time signaling the end of the file transfer.
7. Calls `llClose()` to terminate the connection with the sender. This function receives the sender's DISC frame, sends a DISC frame of its own, and, finally, awaits the sender's acknowledgment.
8. Invokes `appFree()` to clear the memory used by the program.

[4]https://en.cppreference.com/w/c/io/fread

2.3.2 Receiver

1. Initializes the application by calling `appInit()`.
2. Invokes `appRun()` to begin the file-transferring process.
3. Calls `llOpen()`, where it expects a SET frame from the sender and sends a UA frame as its acknowledgment. If the SET and UA frames are correctly received and transmitted, respectively, the connection has been established.
4. Receives a control packet with `llRead()` that details the **name** and **size** of the file to be transferred and the maximum number of **data bytes** the sender will insert in a data packet. The latter is of extreme relevance, as it enables the allocation of a buffer big enough to store the incoming data packets, thus avoiding potential memory issues.
5. Enters the file-transferring loop. Until the file has not been completely received:
   (a) Receives a packet using `llRead()`.
   (b) Parses the **control byte** of the packet to determine whether it is a control or data packet.
      - If the packet is a data packet, its data is appended to the file using `fwrite()`[5] from the C standard library.
      - If the packet is a control packet, that means the file has been fully transferred. As such, the receiver quits the file-transferring loop.
6. Calls `llClose()` to terminate the connection with the sender. This function receives a DISC frame from the sender, sends a DISC frame of its own, and, finally, awaits a UA frame from the sender that serves as its acknowledgment.
7. Invokes `appFree()` to clear the memory used by the program.

[5] `https://en.cppreference.com/w/c/io/fwrite`

Chapter 3

Protocols

Developing the file-transfer application required implementing two data transmission protocols: the logical link protocol in the Data-Link Layer and the application protocol in the Application Layer.

3.1 Logical link protocol

The logical link protocol defines standards for reliable data transmission over the serial port. It guarantees the control and synchronization of the information sent and received.

The main functionalities of this protocol are briefly described below.

Framing

All data is inserted in **frames** before being transmitted via the serial port. Frames are byte streams constituted by a header and a trailer. The header contains address and control fields, so the frame receiver can identify its purpose. In turn, the trailer holds error detection fields for determining if the frame was correctly transmitted. Moreover, all frames are delimited by flags - the byte `0x7E` - so the receiver can accurately recognize when a frame begins and ends.

There are three types of frames:

- **Information (I-frames) -** Contain data pertaining to the Application Layer. In the case of this file-transfer program, they carry the bytes from the file to be transferred.
- **Supervision (S-frames) -** Hold commands to be sent to the other computer, namely requests to connect and disconnect.
- **Unnumbered (U-frames) -** Sent as responses to the other types of frames. Mainly utilized by the receiver to acknowledge any requests from the sender.

Error detection is accomplished by appending Block Check Characters (BCCs) to a frame. These are special bytes whose value is the exclusive disjunction (XOR) of all other bytes in a frame, excluding the flags. This way, when parsing a frame, the program can verify if the XOR of the bytes received equals the BCC field, so most discrepancies that arise from a faulty transmission are discovered. As mentioned above, all frames contain a BCC in their header, however, I-frames are doubly protected, as their data field also contains a BCC for protecting the data bytes.

Considering the similarity between all types of frames, we opted to create a single function for receiving and parsing frames. Listing 3.10 features said function, which relies on a state machine that is updated with each byte read.

```c
/**
 * @brief Receives and parses a frame.
 * @param ll the data-link layer
 * @param address pointer which will store the address of the frame
 * @param control pointer which will store the control byte of the frame
 * @param data buffer which will store the data bytes of the frame, in case it is an I-frame
 * @return the number of data bytes read (for I-frames) or 1 on success, a negative value
 ↪  otherwise
 */
static int receiveFrame(LinkLayer *ll, unsigned char *address, unsigned char *control, unsigned
↪   char *data) {
    State state = STATE_START;
    unsigned char BCC;

    // activate the alarm
    setAlarm(ll->timeout);

    while (alarmIsEnabled)  {
        unsigned char byte;

        // ensure a byte was read
        if (spRead(ll->sp, &byte) <= 0) {
            continue;
        }

        switch (state) {
            case STATE_START:
                // verify if the byte is the flag
                if (byte == FLAG) {
                    state = STATE_FLAG_RCV;
                }

                break;

            case STATE_FLAG_RCV:
                BCC = (*address = byte); // assign the address and update the BCC
                state = STATE_ADDRESS_RCV;

                break;

            case STATE_ADDRESS_RCV:
                BCC ^= (*control = byte); // assign the control byte and update the BCC
                state = STATE_CONTROL_RCV;

                break;

            case STATE_CONTROL_RCV:
                // verify if the current byte is the expected BCC,
                // that is, if it is the XOR of the previous two bytes
                if (byte == BCC) {
                    state = STATE_BCC_OK;
                }
                else if (byte == FLAG) {
                    state = STATE_FLAG_RCV;
                }
                else {
                    state = STATE_START;
                }

                break;

            case STATE_BCC_OK:
                // verify if the current byte is the flag, that is,
                // if we have reached the end of the frame
                if (byte == FLAG) {
                    ++ll->numFramesReceived;
                    return STATUS_SUCCESS;
                }
                // if the current byte is NOT the flag, that means
                // the frame contains data, so receive it
                else if (data) {
                    return receiveData(ll, data, byte);
                }

                state = STATE_START;
                break;
        }
    }

    ++ll->numTimeouts;
    return STATUS_TIMEOUT; // a timeout occurred
}
```

Listing 3.10: Code snippet of the function that receives and parses frames

Byte stuffing

As became apparent, flags are indispensable for recognizing frames. In this protocol, flags are represented by the `0x7E` byte. However, this byte can also occur in the data to be transmitted - after all, the Application Layer is unaware of the framing mechanism. For this reason, it becomes imperative to prevent the false recognition of a flag in a frame.

The solution we adopted was **byte stuffing**, which consists of replacing special bytes in the data with a sequence of two bytes. Said sequence always starts with the same byte - the escape byte, `0x7D` - followed by the exclusive disjunction (XOR) of the special byte with `0x20`. For instance, byte stuffing the flag would result in the sequence `0x7D 0x5E`. This way, the flag never appears in the body of a frame, yet its value is still transmitted.

Naturally, this scheme requires the escape byte itself to be stuffed, since its appearance on a frame would cause the false identification of an escape sequence. Thankfully, byte stuffing can be used with any value, meaning the same principle that applied to the flag applies to the escape byte: replace every occurrence of `0x7D` with `0x7D 0x5D`.

It is important to note that byte stuffing also affects the data field BCC. Firstly, it should be computed using the _original data bytes, before any stuffing takes place. Secondly, it also needs to be stuffed in the unlikely event its value equals `0x7E` or `0x7D`.

Implementing byte stuffing in code is extremely simple, as Listing 3.11 demonstrates.

```
// append the data to the frame
int index = 4;
unsigned char BCC = 0;
```

```c
for (int i = 0; i < dataSize; ++i) {
    unsigned char byte = data[i];
    BCC ^= byte;

    switch (byte) {
        case FLAG:
        case ESCAPE:
            // escape the byte
            frame[index++] = ESCAPE;
            byte ^= XOR_BYTE;

        default:
            frame[index++] = byte;
            break;
    }
}
```

Listing 3.11: Code snippet for byte stuffing data

As for byte destuffing, that is, reconstructing the original payload from the stuffed data, it is similarly straightforward, as can be seen in Listing 3.12.

```c
// receive the remaining data bytes
while (alarmIsEnabled) {
    // ensure a byte was read
    if (spRead(ll->sp, &byte) < 0) {
        continue;
    }

    switch (byte) {
        case FLAG:
            /* ... */

        case ESCAPE:
            escape = TRUE;
            break;

        default:
            // verify if the byte must be escaped
            if (escape) {
                byte ^= XOR_BYTE;
                escape = FALSE;
            }

            // append the byte to the data
            data[index++] = byte;
            BCC ^= byte;

            break;
    }
}
```

Listing 3.12: Code snippet for receiving and byte destuffing data

Sequence number

The logical link protocol needs to be robust in order to not block when errors occur. To ensure this, it imposes a frame numbering system that allows the receiver to identify anomalies and appropriately notify the sender.

In this system, I-frames have a binary **sequence number** (either 0 or 1) that is imbued on their control byte. For the sender, this value represents the number of the next frame it will send. For the receiver, it represents the number it expects the next frame to be numbered with. Initially, the sender transmits a frame numbered 0 and the receiver expects to receive a frame with the same number. As soon as both confirm the transmission succeeded, they toggle the sequence number and repeat the process for the next frame. This continues until no more I-frames are sent.

This system is very effective when it comes to error handling. On one hand, if the receiver detects an I-frame has been corrupted, it responds by indicating the number of the frame it rejects, thus enabling the sender to **retransmit** it. On the other hand, repeated frames are easily identifiable because they share sequence numbers, hence the receiver can simply discard them.

To implement frame numbering, we stored an I-frame counter on the `LinkLayer` data structure (see Listing 2.5) that was incremented on every successful I-frame transmission. Then, we utilized a few macros to compute the sequence number from this counter and write the appropriate control byte, as Listing 3.13 demonstrates.

```
#define CONTROL_I0      0x00
#define CONTROL_I1      0x80
#define CONTROL_I(n)    ((n & 1) << 7)
#define CONTROL_RR0     0xAA
#define CONTROL_RR1     0xAB
#define CONTROL_RR(n)   (CONTROL_RR0 | ((n) & 1))
#define CONTROL_REJ0    0x54
#define CONTROL_REJ1    0x55
#define CONTROL_REJ(n)  (CONTROL_REJ0 | ((n) & 1))
```

Listing 3.13: Code macros for utilizing the sequence number

3.2 Application protocol

The application protocol defines procedures for transferring a file between two computers in a shared medium. It is responsible for fragmenting the file on the sender's end and reconstructing it on the receiver's. In addition, it relies on the logical link protocol to transfer data over the access medium.

The main functionality of this protocol is its packet-based communication system, which is succinctly described below:

Packaging

Similarly to the logical link protocol, all information concerning the Application Layer is encapsulated in byte streams - **packets** - before being transmitted. Each packet contains a control byte for identifying its type followed by one or more Type-Length-Value sequences.

Note: Type-Length-Value (TLV) is an encoding scheme for compactly representing data in a byte stream. Type is a byte that uniquely identifies the purpose of the data, Length is a byte that represents the number of data bytes that follow, and Data is a field of size Length that contains the data itself.

There are two types of packets:

- **Control -** Hold information regarding the file transferring process, such as the name and size of the file to be transferred and the maximum number of data bytes the sender will insert in a data packet.
- **Data -** Contain bytes from the file to be transferred, as well as a sequence number for preserving the file's integrity.

Unlike the logical link protocol, only the sender can transmit packets. As such, for simplicity, we opted against writing reusable code that would work for all packets and instead devised four functions, each with a concrete purpose: `sendControlPacket()`, `sendDataPackets()`, `receiveControlPacket()`, and `receiveDataPackets()`. The first two (Listing 3.14) are used by the sender, while the last two (Listing 3.15) are invoked by the receiver. Hence, implementing the Application Layer was only a matter of appropriately calling these functions.

```c
/**
 * @brief Creates a control packet and sends it via the serial port.
 * @param app the application
 * @param control the control byte of the control packet
 * @return 1 on success, a negative value otherwise
 */
static int sendControlPacket(ApplicationLayer *app, unsigned char control) { /* ... */ }

/**
 * @brief Creates the data packets and sends them via the serial port.
 * @param app the application layer
 * @return 1 on success, a negative value otherwise
 */
static int sendDataPackets(ApplicationLayer *app) { /* ... */ }
```

Listing 3.14: Code definition of the Application Layer functions called by the sender

```c
/**
 * @brief Receives and parses a control packet sent via the serial port.
 * @param app the application layer
 * @return 1 on success, a negative value otherwise
 */
static int receiveControlPacket(ApplicationLayer *app) { /* ... */ }

/**
 * @brief Receives and parses data packets sent via the serial port.
 * @param app the application layer
 * @return 1 on success, a negative value otherwise
 */
static int receiveDataPackets(ApplicationLayer *app) { /* ... */ }
```

Listing 3.15: Code definition of the Application Layer functions called by the receiver

Chapter 4

Validation

4.1 Testing

To guarantee the protocols had been correctly implemented, we tested our application numerous times in distinct scenarios, namely:

- Transferring different files.
- Transferring in a noisy channel.
- Varying the Frame Error Rate (FER).
- Varying the frame propagation delay (T_{prop}).
- Varying the baud rate.
- Varying the size of the data packets.

In all of the above, the application worked as expected, thus corroborating its correctness.

4.2 Efficiency

From the testing scenarios presented in Section 4.1, the last four were also used to experimentally measure the execution time of the application. Then, we computed the efficiency of our logical link protocol with the formula presented in Figure 4.1.

$$S = \frac{Throughput}{Channel\ bandwidth} = \frac{Bits/s\ transmitted}{Baud\ rate}$$

Figure 4.1: Formula for calculating the efficiency of the logical link protocol

For each scenario, we chose at least five distinct values for the parameter to be varied. In turn, for each value, we measured the execution time at least four times, excluding any outliers in the process, and then calculated the average. To obtain the most accurate results possible, we did not round the measurements. However, we rounded the intermediate values, namely the average execution time and the throughput, and the result to four decimal places.

Note: Tables displaying the measurements for all testing scenarios can be found on Section I.

Unless explicitly stated otherwise, the default parameters for each test are as follows:

- **File size:** 10968 B
- **Frame Error Rate:** 0%
- **Propagation delay:** 0 s
- **Baud rate:** 9600 bit/s
- **Data packet size:** 200 B

Frame Error Rate

Frame Error Rate (FER) is, as the name suggests, the rate of frames that contain errors. It is obtained by dividing the number of erroneous frames by the total amount of frames. To vary this parameter, we toggled a bit in a uniformly distributed selection of transmitted frames. Figure 4.2 shows the efficiency curve for distinct Frame Error Rates.

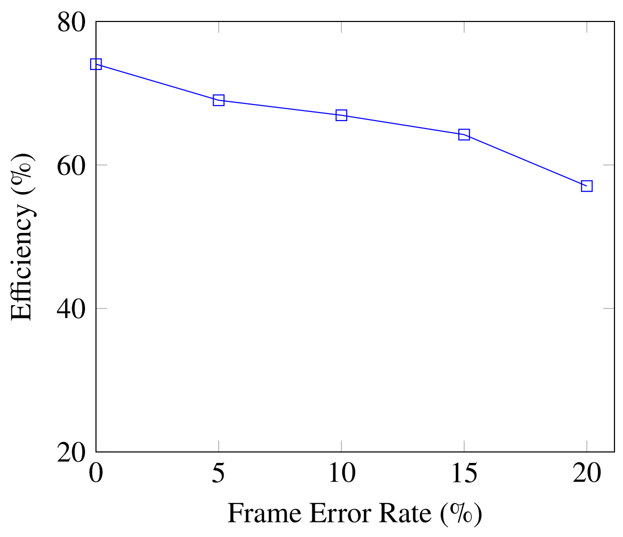


Figure 4.2: Efficiency curve for different Frame Error Rates

Unsurprisingly, efficiency is inversely proportional to the Frame Error Rate. This can be explained by the fact that, unlike other logical link protocols, no frame recovery mechanisms were implemented, meaning that each erroneous frame needs to be retransmitted. Since each retransmission delays the file transfer completion, it naturally decreases the efficiency.

Propagation delay

Propagation delay (T_{prop}) refers to the time it takes a signal to reach its destination. In this project, it represents the delay between the emission of a bit via the serial port and its reception by the other computer. Figure 4.3 depicts the efficiency curve for a few propagation delay values.

As can be deduced, propagation delay and efficiency are inversely proportional because more propagation delay leads to more time transmitting each frame, which naturally increases the total transfer time.

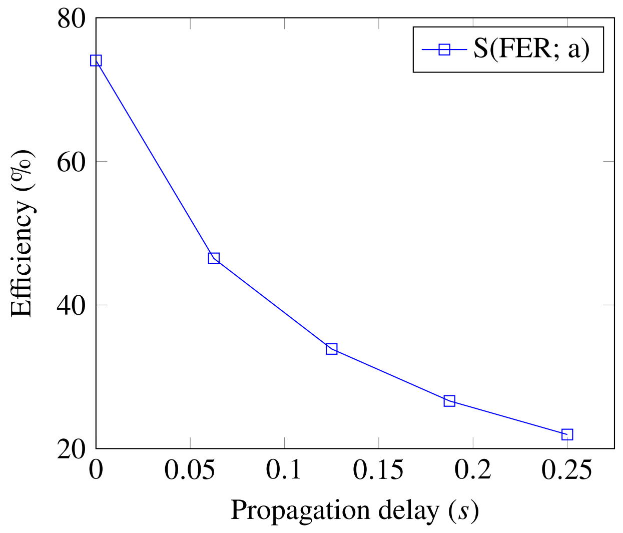


Figure 4.3: Efficiency curve for different propagation delay values

Baud rate

Baud rate is the theoretical capacity of the access medium, that is, the number of symbols that are transmitted per second. Since the serial port only sends one bit at a time, it represents the number of bits that are transmitted per second.

As Figure 4.4 portrays, the efficiency curve is mostly linear, though there is a slight downward slope along the higher values. This can seem quite odd, given that the transfer time decreases drastically (see Table 3). However, it is important to remember that efficiency is computed by dividing the actual bitrate by the baud rate, and, since the two are directly proportional, the result remains mostly consistent.

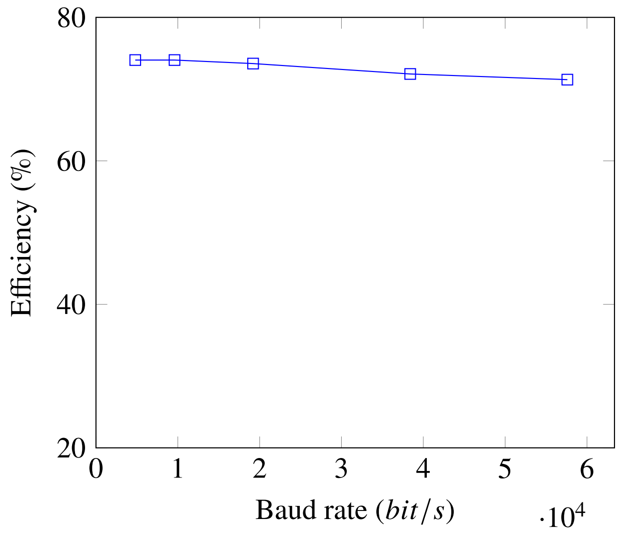


Figure 4.4: Efficiency curve for different baud rates

Data packet size

Note: The size of the data packets also directly affects the size of the I-frames, as the latter transport the data packets in the Data-Link layer.

As Figure 4.5 denotes, the size of a data packet is directly proportional to efficiency. To understand this correlation, it is important to recall that each data packet implies double processing: the Application layer processes the data packet and the Data-Link layer processes the I-frame that will carry it. Moreover, the larger the data packet, the more data it holds, thus decreasing the amount of these that is sent. As such, an increase in the size of data packets leads to a decrease in the time the program spends processing them.

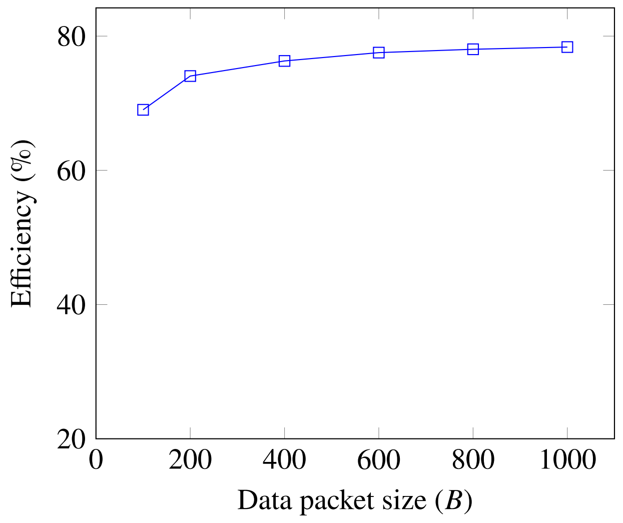


Figure 4.5: Efficiency curve for different data packet sizes

Chapter 5

Conclusion

Overall, the project was completed successfully. On one hand, both the logical link and application protocols were duly implemented and are according to the specification. On the other hand, the file-transfer application works as expected, even under a faulty connection.

From an educational perspective, this project was incredibly enriching. On a technical level, it enabled us to test our low-level programming skills, ranging from memory management to I/O manipulation. Furthermore, it proved to be a valuable opportunity for applying our theoretical knowledge of data transmission protocols in a meaningful context.

Bibliography

Sheldon, R. (2023, February). method (in object-oriented programming). https://www.techtarget.com/whatis/definition/method. [Accessed: November 7, 2024].

Appendix

I Efficiency measurements

Frame Error Rate

FER (%)	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_5 (s)	t_{avg} (s)	Capacity (bit/s)	Efficiency (%)
0	12.442	12.443	12.443	12.443	12.442	12.4426	7108.4822	74.05
5	14.235	12.667	13.793	12.929	13.121	13.349	6625.8147	69.02
10	13.79	13.883	13.12	13.571	14.46	13.7648	6425.6655	66.93
15	14.17	13.579	13.83	14.738	15.409	14.3452	6165.6861	64.23
20	15.377	15.637	18.42	16.157	15.138	16.1458	5478.081	57.06

Table 1: Efficiency measurements for different Frame Error Rates

Propagation delay

T_{prop} (s)	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_{avg} (s)	Capacity (bit/s)	Efficiency (%)
0	12.442	12.443	12.443	12.443	12.4428	7108.3679	74.05
0.0625	19.818	19.817	19.818	19.817	19.8175	4463.126	46.49
0.125	27.197	27.204	27.197	27.193	27.1978	3252.0277	33.88
0.1875	34.574	34.573	34.611	34.574	34.583	2557.5572	26.64
0.25	41.943	41.973	41.947	41.964	41.9568	2108.0731	21.96

Table 2: Efficiency measurements for different propagation delay values

Baud rate

Baud rate (bit/s)	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_{avg} (s)	Capacity (bit/s)	Efficiency (%)
4800	24.884	24.884	24.885	24.885	24.8845	3554.3411	74.05
9600	12.442	12.443	12.443	12.443	12.4428	7108.3679	74.05
19200	6.244	6.259	6.263	6.284	6.2625	14123.4331	73.56
38400	3.178	3.216	3.184	3.198	3.194	27691.9224	72.11
57600	2.127	2.188	2.148	2.149	2.153	41081.2819	71.32

Table 3: Efficiency measurements for different baud rates

Data packet size

Data packet size (B)	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_{avg} (s)	Capacity (bit/s)	Efficiency (%)
100	13.465	13.302	13.328	13.302	13.3493	6625.6658	69.02
200	12.442	12.443	12.443	12.443	12.4428	7108.3679	74.05
400	12.206	12.022	12.051	12.023	12.0755	7324.5828	76.3
600	11.882	11.881	11.882	11.881	11.8815	7444.1779	77.54
800	11.805	11.804	11.805	11.807	11.8053	7492.2281	78.04
1000	11.757	11.757	11.756	11.756	11.7565	7523.3275	78.37

Table 4: Efficiency measurements for different data packet sizes