Computer Networks

Serial Port Project Report

João Baltazar[1] and Nuno Costa[1]

[1]Faculty of Engineering of the University of Porto

December 11, 2021

Abstract

The goal of the project was to send information between two machines through a serial port, building a communication protocol with two well-established and independent layers - application layer and data link layer.

This goal was achieved, resulting in an application capable of sending information between two machines in a consistent manner, with error checking and handling according to the required specifications. The development of this application made the importance of proper layer isolation clear, since it made the structure of the project easier to design and code, more robust, more modular and less prone to errors.

1 Introduction

The project had two main objectives - to implement a data linking protocol, according to the given specification, and test it with a simple file transfer application, also obeying given specifications. This protocol should provide a reliable communication service between the two systems connected by the serial port, ensuring under the hood that it has proper syncing and framing, connection establishment and termination, frame numeration, positive confirmation and adequate error handling and flow control.

This report seeks to explain the thought process on the development of the project. Its structure is as follows:

- Architecture
- Code Structure
- Main Use Cases
- Data Link Protocol
- Application Protocol
- Validation
- Data Linking Protocol Efficiency
- Main Conclusions

2 Architecture

2.1 Layers

Two distinct layers were created for the information transfer.

2.1.1 Data Link Layer

This layer is the *Low Level* layer. It is responsible for the direct communication with the serial port, and assuring that any arbitrary information it was asked to send is passed on correctly.

2.1.2 Application Layer

This layer is the *High Level* layer. It is responsible for unpacking the information to send, send it to the **Data Link Layer** and, on the other side, retrieve it from the **Data Link Layer** and assemble it back together.

2.2 Program Execution

The program was conceived to execute as follows:

```
app serialport role [filepath]
```

where:

- *serialport* is the path to the port
- *role* defines if it is receiving information (*receiver*) or sending information (*sender*)
- *filepath* is the path of the file to send (when *role* = *sender*)

3 Code Structure

3.1 Application Layer

As previously mentioned, the application shares its main executable for both the sender and receiver. However, the flow of the program depends on the *role* parameter. This flow is defined in *main.c*, using the *app.c* functions. As previously mentioned, the application layer functions use the data link layer functions.

3.1.1 app.c

```
typedef struct {
    int fileDescriptor;
    Source status;
    int sequenceNumber;
} applicationLayer;
```

- Needed information for running the application. The *fileDescriptor* field refers to the serial port file descriptor. The *status* field refers to the type of execution (sender or receiver). The *sequenceNumber* refers to the current sequence number of the received data packets - useful for error checking on *receive_ data*.

```
typedef struct {
    size_t fileSize;
    u_int8_t fileNameSz;
    char* fileName;
} fileInfo;
```

- Metadata of the file to send/receive. The *fileSize* field refers to the size of the file. The *fileNameSz* field refers to size of the *fileName* field, which itself refers to the relative path of the file to send/receive

```
int app_start(int port, Source role)
```

- Starts the application, defining the port to run in and the role (sender/receiver) of the current execution, and opening the serial port connection (*llopen*)

```
int send_file(const char *filepath)
```

- Sends the file, through the serial port, given by *filepath*. This function opens the file, sends important metadata and calls *send_ data* to send the file data

```
int send_data(FILE *file, size_t size)
```

- Sends the data of the file pointed by *file*, of size *size* to the serial port (using *llwrite*)

```
int receive_file()
```

- Receives a file from the serial port. Firstly receives the file metadata, then calls *receive_ data* to receive the file data

```
int receive_data(FILE* file)
```

- Receives the file data (using *llread*) and writes it to *file*

```
int app_end()
```

- Ends the application, closing the connection (*llclose*) and cleaning memory

3.2 Data Link Layer

The Data Link Layer is centered in the *ll.c* file, which controls the flow of this layer. The lower level link layer functions are located in the *comms.c* file. These are the ones which interact directly with the serial port (with the read/write calls).

3.2.1 ll.c

```
typedef struct {
    char port[MAX_PORT_SIZE];
    int baudRate;
    unsigned sequenceNumber;
    unsigned timeout;
    unsigned numTransmissions;
} linkLayer;
```

- Defines the link layer parameters

```
typedef enum {SENDER, RECEIVER} Source;
```

- Defines the execution type and message source

```
int llopen(int port, Source newRole)
```

- Opens the connection to the serial port

```
int llwrite(int fd, char *buffer,
            int length)
```

- Writes the frame in *buffer* of length *lengt* to the port given by *fd*

```
int llread(int fd, char *buffer)
```

- Reads a frame to *buffer* from the port given by *fd*

```
int llclose(int fd)
```

- Closes the connection to the serial port

3.2.2 comms.c

```
int send_s_u_frame(int fd,
                   Source src,
                   int ctrl)
```

- Sends a control frame with *ctrl* parameter, given its *src* (sender/receiver), to the port given by *fd*

```
u_int8_t receive_s_u_frame(int fd,
                           Source src)
```

- Receives a control frame, given its *src* (sender/receiver), from the port given by *fd*

```
int send_i_frame(int fd,
                 char *buffer,
                 int length,
                 bool seqNum)
```

- Sends an information frame from *buffer* of length *length*, with a sequence number of *seqNum* into the port given by *fd*

```
int receive_i_frame(int fd,
                    char *buffer,
                    bool seqNum)
```

- Receives an information frame into *buffer* with sequence number *seqNum* from the port given by *fd*

4 Main Use Cases

There are two main use cases: either one sends a file or one receives a file. As such, both of these use cases's function call sequence is described below.

4.1 Sender

1. Begins the application (*app_start*)
2. *app_start* calls *llopen*, which opens the connection to the serial port
3. After the connection is open, *send_file* is called with the path of the file to send as a parameter
4. In *send_file*, the file to send is opened, and the file metadata is sent to the receiver
5. After successfully sending the metadata, *send_data* is called with the file to send and its size as parameters
6. In *send_data*, the information is split in packets of a user defined size and are sent to the receiver using *llwrite*, one by one - which adds a header, BCCs for errors and start and end flags to the packet, as well as stuffing it
7. After successfully sending the information, the file is closed
8. The application is closed using *app_close* - which calls *llclose*

4.2 Receiver

1. Begins the application (*app_start*)
2. *app_start* calls *llopen*, which opens the connection to the serial port
3. After the connection is open, *receive_file* is called to receive a file
4. In *receive_file*, the metadata of the file is received and saved in the *fileInfo* struct
5. After successfully receiving the metadata, *receive_data* is called with the file to receive as a parameter
6. In *receive_data*, the information packets are received - using *llread*, which destuffs and checks BCCs for errors. After also being stripped of their header, BCCs and start and end flags, the packets are written directly to the file, one by one
7. After successfully receiving all the information, the file is closed
8. The application is closed using *app_close* - which calls *llclose*

5 Data Link Protocol

As mentioned before, the Data Link Layer's main flow is as follows: open the connection using *llopen*; write/read frames of information using *llwrite*/*llread*; close the connection using *llclose*. The implementation strategy of each of these steps is described below.

5.1 llopen

```
int llopen(int port, Source newRole)
```

This function is used to open and establish the connection with the serial port. Given that it should only be possible to consider the connection **open** when, in fact, a *receiver* is waiting to read the information - from a sender's standpoint -, a syncing protocol must be followed. To do that, the sender sends a *SET* control packet to the receiver, to which the receiver must respond with a *UA* control packet - meaning that message was received and acknowledged.

From a sender's standpoint, the following code is executed:

```
while(timeout_no < ll.numTransmissions)
{
    send_s_u_frame(fd, SENDER, SET);

    if(receive_s_u_frame(fd, SENDER) == UA)
        break;
    else timeout_no++;
    }

    if(timeout_no == ll.numTransmissions)
        return -2;
}
```

If the syncing fails, a timeout mechanism is in place, to ensure that the system does not block. In fact, there are two timeout mechanisms. Firstly, a 3 second (user defined) timeout is set for all read/open calls. Secondly, the system can fail up to 3 times (*ll.numTransmissions*, also user defined), accounting for 9 seconds for a complete timeout. When this limit is reached, this function terminates.

The code from a receiver's standpoint is similar.

5.2 llwrite

```
int llwrite(int fd, char *buffer,
            int length)
```

This function is used to write an information frame to the serial port. As stated previously, it writes the information of *buffer*, of size *length*, to the port given by *fd*. Firstly, this function tries to send an information frame to the buffer using *send_i_frame*. Given the receiver's answer and *s* being the current sequence number, the following action may vary. If the response is *REJ(s)*, then it tries to send the same packet again and resets the timeouts. Otherwise, if the response is different than zero and different than *RR(inv(s))*, llwrite tries to send the same frame again. If the response is 0, then no value was read by the receiver or no response was sent, which adds one to the timeout count and, as *llopen*'s behaviour, when the *ll.numTransmissions* limit is reached, the function returns unsuccessfully. Finally, if the response is equal to *RR(inv(s))*, it means that the receiver is 'Receiver Ready for the inverse of i', which means it's ready for the next frame, and so the function returns successfully - the number of bytes written is returned.

To send information, the *send_i_frame* function is used, and is explained below.

```
int send_i_frame(int fd,
            char *buffer,
            int length,
            bool seqNum)
```

To ensure that the information is passed on correctly, a frame with a determined structure is built, and *send_i_frame* follows that structure. The frame is firstly composed by a header body composed by the address of the information and a control byte which determines the sequence number, and a BCC - calculated through an exclusive or of the previous two bytes - to ensure header integrity and error control. Secondly, all the data bytes follow, ending in another BCC - calculated through an exclusive or of all data bytes - to ensure data integrity and error control. Finally, two delimiting flags must be added, so that the receiver knows where the frame begins and ends. However, that same flag may very well appear in the middle of the frame. In that case, a stuffing mechanism is applied to the frame, which changes any delimiting flag byte with an escape byte followed by a different byte, representing that same flag. If the escape byte is also in the frame, then it is adds another byte following it, indicating that it is, in fact, an escape byte. With all of this done, this function

adds the delimiting flags to the beginning and end of the frame, and sends it to the user through a write call.

5.3 llread

```
int llread(int fd, char *buffer)
```

This function is used to receive an information frame from the serial port. As stated previously, it receives the information from the port given by *fd* and writes it to *buffer*. Firstly, this function tries to receive an information frame and write it to the buffer using *receive_i_frame*. Given the return value of this function and *s* being the current sequence number, the following action may vary. If the response is -1, it means that there was a data error. As such, the message must be rejected (it can be because the integrity of the header of the message is assured) before trying to receive it again - a *REJ(s)* is sent to the sender. If the response is 0, it means that the packet was valid, but the sequence number is wrong. As such, the sender sends a *RR(s)* asking for the correct frame again. If the result is less than -1, it means that nothing can be said about the message received, because it is not in the expected format. As such, *llread* ignores it and tries to read it again. Finally, whenever the result is greater than 0, it means that *buffer* has now *result* number of bytes. As such, the receiver responds to the sender with *RR(inv(s))* and the function can return successfully.

To receive information, the *receive_i_frame* function is used, and is explained below.

```
int receive_i_frame(int fd,
                    char *buffer,
                    bool seqNum)
```

Given that a message from *send_i_frame* is expected to be delimited by two flags, *receive_i_frame* reads all the information from the serial port, starting with a flag and ending with a flag. After having all those received bytes stored, it iterates through them via state machine, ensuring that the message structure is correct. As such, it checks that the first two bytes are the expected address and a valid control byte, and that the following byte is the correct BCC for those two bytes. If not, it leaves the function with the *HEADER_ERR* code. If the control byte is valid and the corresponding BCC is also correct, it means that the received byte has the wrong sequence number. This probably means that this byte is repeated and, as such, the function returns *SEQNUM_ERR*. After evaluating the header, it proceeds to iterate through all data bytes. While iterating them, it calculates a BCC, which will be compared with the last byte. If they are not equal, then the function returns with a *DATA_ERR* code, since the integrity of the data was not maintained. If everything goes well, the function returns the number of received data bytes.

5.4 llclose

```
int llclose(int fd)
```

This function is used to close the connection with the serial port. Following the same logic used in *llopen*, both the sender and receiver must close at roughly the same time. To do so, a syncing routine is applied. In this function, the receiver is waiting for the sender to send a *DISC* - meaning that the program should end. Upon receiving this message, the receiver answers with *DISC*, and expects the sender's acknowledgement.

If the syncing fails, a timeout system is, yet again, in place, to ensure that the system does not block and that it closes gracefully.

The routine from a sender's standpoint is similar.

6 Application Protocol

The application protocol promises to open the connection on the desired port, send files from the sender to the receiver through the opened port, and also close said connection when the user finds that appropriate.

6.1 app_start

This function accepts a port number to hook to */dev/ttyS[port_num]*, and a role in the communication, and tries to initiate communication using *llopen* on the given port.

6.2 send_file

This function takes in a filepath to the file that is to be sent and performs various tasks in the following order:

1. open the file

2. read its metadata - filename and filesize
3. send the metadata through a control packet, announcing the start of the transmission
4. send the data packets with the contents of the file
5. announce the end of the transmission through a control packet
6. close the file

Regarding points 2. and 3., it's worth noting that the metadata is sent in the TLV (Type, Length, Value) form, which is fully modular in the code - the start control packet is, effectively, a TLV array, so more information could be added with very few tweaks to the code:

```
int send_control_packet(u_int8_t ctrl,
TLV* tlv_arr, unsigned n_tlv)
{
    size_t size = 1;

    for (int i = 0; i < n_tlv; i++)
    {
        size += tlv_arr[i].L + 2;
    }

    char *packet = malloc(size);

    int curr_idx = 0;

    packet[curr_idx++] = ctrl;

    for (int i = 0; i < n_tlv; i++)
    {
      packet[curr_idx++] = tlv_arr[i].T;
      int len = tlv_arr[i].L;
      packet[curr_idx++] = len;
      memcpy(packet + curr_idx,
                tlv_arr[i].V, len);
      curr_idx += len;
    }

    ...

}
```

On 4., the size of each data packet sent to the receiver is limited by MAX_PACK_SIZE, and each packet is a separate *llwrite* call, managed independently.

6.3 receive_file

This function reads the start control packet, parses the metadata, opens the new file, reads the data packets into this new file and then closes it once the end control packet is received. For each data packet, llread is called, and the packet's sequence number and size are checked before writing. This operation is performed until an end packet is received, at which time the function returns.

6.4 app_end

This function simply cleans up the malloc'd memory to store the file name and closes the link layer.

7 Validation

The following tests were performed:

- File transfer with different sizes, extensions and names: hehehe.txt (22 B); magnum_a_fazer_fixe.jpg (59,1 KB); pinguim.gif (10,7 KB); rolando.gif (3,62MB)
- File transfer with interference
- File transfer with temporary disconnect
- File transfer with different Baudrates: $B115200, B57600, B38400, B19200, B9600$
- File transfer with different Packet Sizes: $16, 32, 64, 128, 256, 512, 1024, 2048$
- File transfer with simulated time delay (emulate real-life transfer with bigger distance connections, based on the benchmark that the delay is equal to $5\mu s/km$): 0, 1km, 10km, 100km, 1000km, 10000km

All tests were sucessful. The integrity of the transfered file was verified using *diff*.

8 Data Linking Protocol Efficiency

The previously mentioned tests mostly focused on varying 4 main customizable fields of the connection: the Baudrate, the Frame Error Ratio (emulated), the connection distance/propagation time (emulated) and the Information Frame Size. All the tests are present in Appendix II.

8.1 Varying Baudrate

The variance in the Baudrate did not result in a significant efficiency change (slight downwards slope function approximation). Theoretically that makes sense, since the efficiency is a result of dividing the real flux of information by the Baudrate. Since the real flux is directly proportional to the Baudrate, the efficiency should stay constant.

8.2 Varying Frame Error Ratio (FER)

Note: For this, a random error was emulated in the sent frames, according to the defined FER.

The variance in the Frame Error Ratio resulted in a significant efficiency change (symmetric sigmoidal growth function approximation). First of all, the penalties of the applied data link protocol must be clarified. An error on BCC1 in a control or information frame results in a penalty of 3 seconds (timeout). An error solely on BCC2 results in a re-transmission request, where the penalty is substantially smaller than the previous. 3 consecutive timeouts result in the protocol giving up the connection. Moving on to the test properties, these follow a binomial probability model. As such, an error in a random byte every $1/FER$ frames leads to the following assertions:

$$P(error_{header}|error_i) = \frac{len_{header}}{len_{frame}}$$

The probability of having a header error in an information frame is equal to the length of the header divided by the length of the frame.

$$P(error_{data}|error_i) = 1 - P(error_{header}|error_i)$$

The probability of having a data error in an information frame is equal to 1 minus the probability of having a header error in an information frame.

$$P(error_{header}|error_{control}) = P(error_{control})$$

The probability of having a header error in a control frame is equal to the probability of having an error in a control frame (because a control frame only has header).

In fact, the last assertion is problematic. In an environment with a high frame error ratio, the probability of having control frames with errors is higher. As explained previously, failing to send a valid BCC1 results in a 3 second penalty. Furthermore, even if the error is in BCC2, the data link protocol to handle these kinds of errors is to send another control packet (either REJ or RR). However, as seen above, more control packets represents more possible penalties, and that leads to more time without any information being transferred. It could also be asserted that, with sufficiently high FER, the probability of a connection not being complete also increases, upon giving up after 3 timeouts, which affects directly how the efficiency is calculated (this measure should be disabled for these values of FER in order to test if the system's efficiency reliably). With all of this, it is clear that the obtained results make sense in this context.

8.3 Varying the Connection Distance (t_{prop})

Note: For this, a sleep function was used before each write() in the program. To define the relationship between distance and time of travel, a $5\mu s/km$ benchmark was used.

The variance in the Connection Distance resulted in a significant efficiency change (inverse proportional decrease function approximation). Theoretically that makes sense, since the propagation time makes the efficiency decrease following an inverse proportional function ($\frac{1}{1+2a}$, where $a = \frac{t_{prop}}{t_f}$, t_{prop} being the time of propagation, t_f being the time of transfer). It must be noted that this parameter starts to decrease the efficiency substantially when the time of propagation becomes a bottleneck, which happens when $t_{prop} > t_f$. Whenever this happens, the time spent on hold is greater than the time spent actually transferring the files, diminishing the due time substantially.

8.4 Varying the Information Frame Size

The variance in the Information Frame Size resulted in a significant efficiency change (symmetric sigmoidal growth function approximation). When increasing the information frame size, this results in less calls to the data link protocol - this results

in less header management overhead, that being either header constructing/deconstructing as well as frame consistency checking. However, the rate at which this decrease in data link protocol calls also decreases - the decrease in link layer calls is smaller, mathematically. As such, the theory would indicate to a inverse proportional function, which is what was obtained experimentally. It should also be noted that, since FER was 0 in these tests, there are no downsides to increasing the information frame size - if it was not, the overhead associated with the re-transmission of a bigger frame, which has a greater probability of having errors, would decrease efficiency. One can only assume that a sweet spot would be found between the FER and information frame size, where the penalty of having a greater frame leading to more re-transmission requests would balance out with the penalty of having a smaller frame and leading with the header associated overhead. This thesis - which is only an assumption, by now - must be tested and sustained by data.

9 Conclusions

All of the project's goals were achieved - the delivery is complete, in accordance with the specifications, well designed and working as intended; the analysis of the test results matches and reinforces what was discussed in class on a theoretical level; and the development of the project as a whole was a solid practical learning experience.

Appendix I

Annex code - folder *code/*.

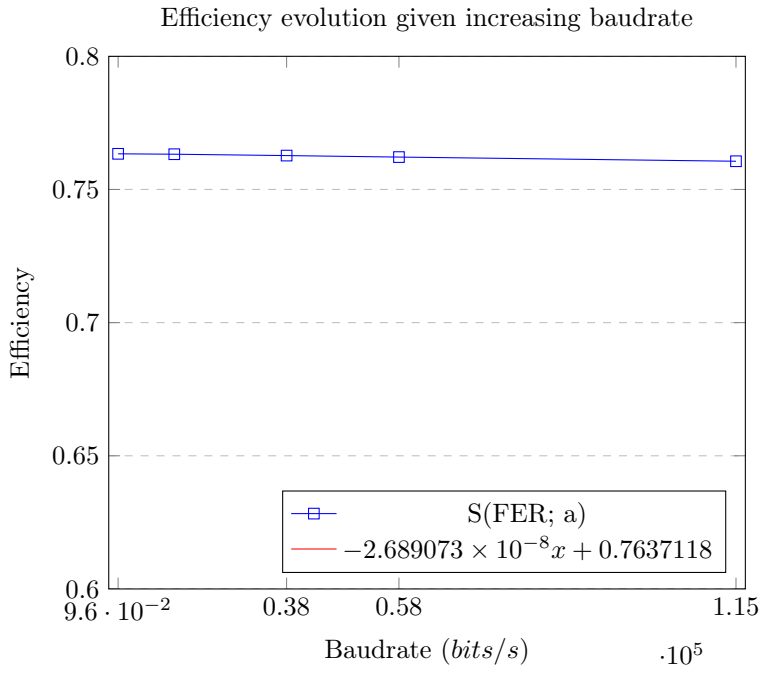
Appendix II

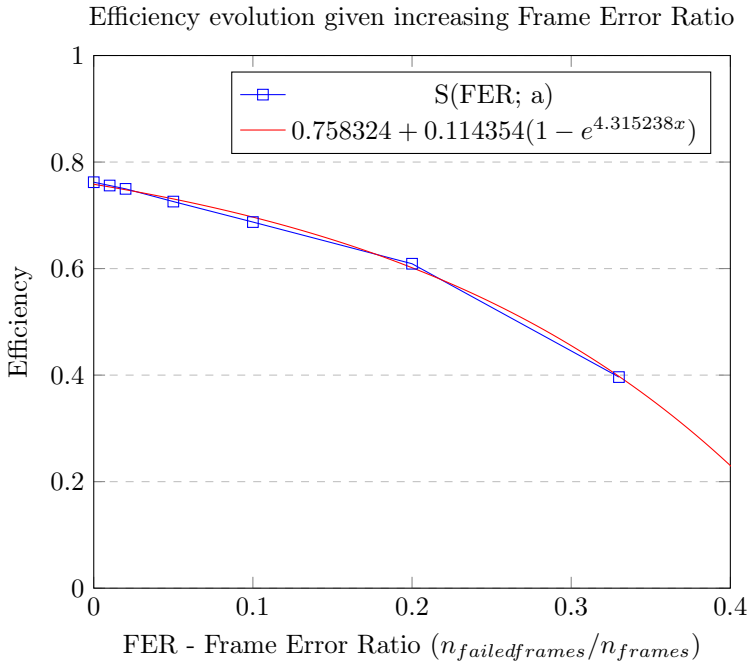
Baud Rate $bits/s$	File Size $bytes$	Frame Size $bytes$
57600	484632	512

Table 1: Default values.

Baudrate $C = bits/s$	t1 s	t2 s	t $mean(t1, t2)$	Flow $R = bits/s$	Efficiency $S = \frac{R}{C}$
115200	5,53047	5,53153	5,531	87621,04502	0,7605993491
57600	11,040068	11,038594	11,039331	43900,48636	0,7621612215
38400	16,54689	16,546753	16,5468215	29288,52529	0,7627220128
19200	33,071012	33,071745	33,0713785	14654,12154	0,7632354968
9600	66,140327	66,118949	66,129638	7328,514334	0,7633869098

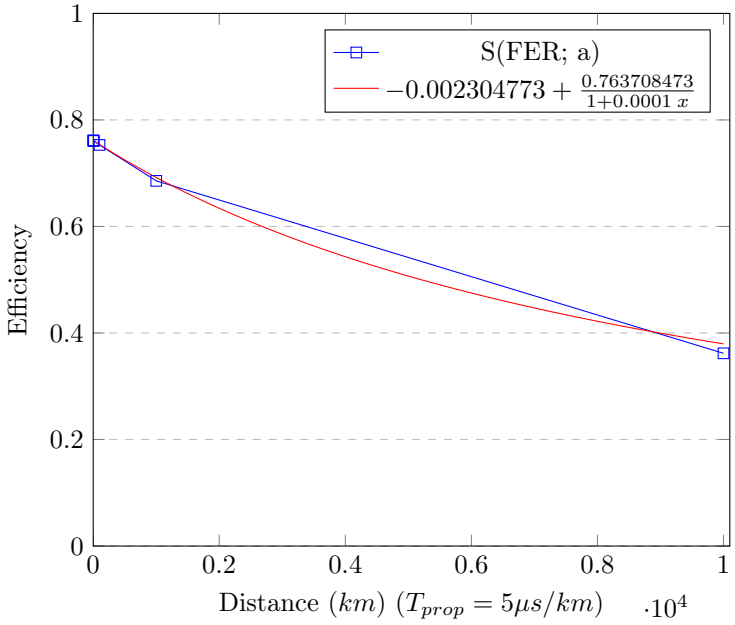
Table 2: Calculated Values for a varying baudrate





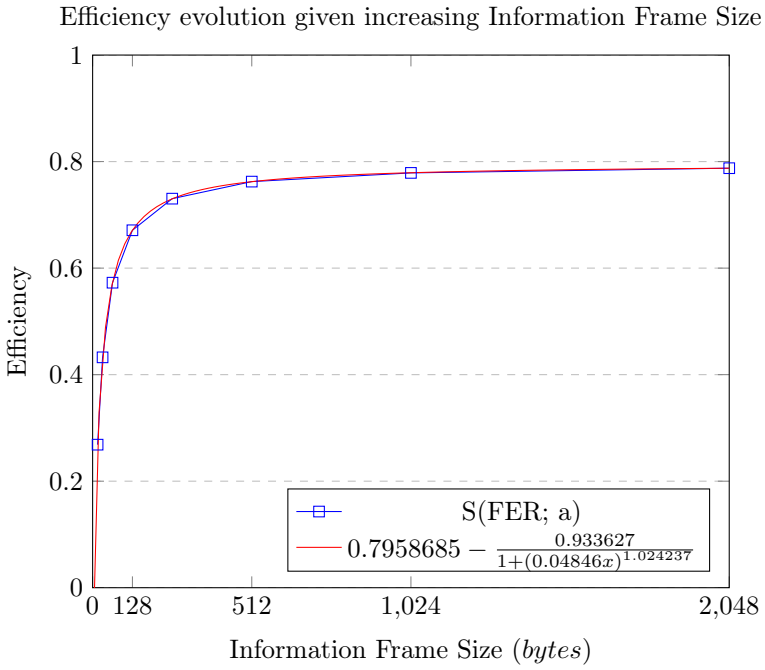
Frame Error Ratio $FER = \frac{n_{failedframes}}{n_{frames}}$	t1 s	t2 s	t $mean(t1, t2)$	Flow $R = bits/s$	Efficiency $S = \frac{R}{C}$
0	11,040068	11,038594	11,039331	43900,48636	0,7621612215
0,01	11,130485	11,129787	11,130136	43542,32509	0,7559431439
0,02	11,22316	11,223345	11,2232525	43181,06538	0,749671274
0,05	11,592117	11,592556	11,5923365	41806,23984	0,725802775
0,1	12,240672	12,24139	12,241031	39590,78284	0,6873399798
0,2	13,813091	13,814321	13,813706	35083,41643	0,6090870907
0,33	22,754256	19,706312	21,230284	22827,39129	0,3963088765

Table 3: Calculated Values for a varying frame error ratio



Distance km	t1 s	t2 s	t $mean(t1, t2)$	Flow $R = bits/s$	Efficiency $S = \frac{R}{C}$
0	11,050612	11,049718	11,050165	43857,44466	0,7614139698
1	11,052096	11,051472	11,051784	43851,01989	0,7613024286
10	11,062725	11,062326	11,0625255	43808,44139	0,7605632186
100	11,172951	11,172936	11,1729435	43375,49903	0,7530468582
1000	12,2716688	12,2721	12,2718844	39491,24553	0,6856119016
10000	23,253007	23,253229	23,253118	20841,59208	0,3618331959

Table 4: Calculated Values for a varying distance



Information Frame Size $bytes$	t1 s	t2 s	t $mean(t1, t2)$	Flow $R = bits/s$	Efficiency $S = \frac{R}{C}$
16	31,332732	31,328677	31,3307045	15468,2765	0,2685464669
32	19,451333	19,455602	19,4534675	24912,37102	0,4325064413
64	14,693694	14,692548	14,693121	32983,59824	0,5726319139
128	12,539763	12,535943	12,537853	38653,5079	0,6710678455
256	11,51947	11,519144	11,519307	42071,28085	0,7304041814
512	11,035489	11,03581	11,0356495	43915,13159	0,762415479
1024	10,80353	10,804727	10,8041285	44856,18623	0,7787532331
2048	10,682791	10,682879	10,682835	45365,48585	0,7875952404

Table 5: Calculated Values for a varying information frame size