U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 1 Report

Computer Networks
L.EIC025

Class L.EIC05

Bruno Miguel Lopes da Rocha Fernandes up202108871
Vasco Moutinho de Oliveira up202108881

Summary

This project, which was created for the Computer Networks Curricular Unit, implements an RS-232 serial port data transfer protocol for file transmission.
We were able to use the material we had learned in class and solidify it through this assignment.

Introduction

The goals of this project are to implement a simple version of a data link layer protocol according to the specification provided in the guide. Also, a simple transmitter and receiver data transfer application should be developed to test the implemented protocol.

The report is divided into eight sections:

- **Introduction:** an overview of the report's logic and a summary of the information that is provided in each of the following sections, in tandem with an explanation of the goals of the work and the report.

- **Architecture:** description of the functional blocks and interfaces.

- **Main Use Cases:** identification of the main use cases of the application and the function call sequence.

- **Link Layer Protocol** and **Application Layer Protocol:** pinpointing of the main features of the protocol and description of their implementation strategy with code snippets and how it is structured.

- **Validation:** catalog of the tests performed showing quantitative results.

- **Link Layer Protocol Efficiency:** statistical characterization of the efficiency of the protocol.

- **Conclusions:** recap of the information from the previous sections and list of the main learning goals achieved.

Architecture

The architecture of the application is a layered architecture. The core concept of layered architectures is the idea of layer independence. Each layer is independent of the other layers, and each layer only depends on the layer immediately below it.

Functional Blocks

The project is divided into two separate layers: the Link Layer and the Application Layer.

The Link Layer is the implementation of the protocol referred earlier, and its implementation is in the *link_layer.h* and *link_layer.c* files. This layer is responsible for establishing and breaking connections, creating and delivering data frames via the serial port, and providing error messages in the event that something goes wrong.

The Application Layer serves as the implementation of another previously mentioned protocol. It is realized through the application_layer.h and application_layer.c files, and it leverages the Link Layer API to facilitate the transmission and reception of data packets belonging to a file. This layer operates in close proximity to the end-user, defining crucial parameters such as information frame size, *baudrate*, and the maximum number of retransmissions.

Interfaces

The program is executed by opening two terminals, one in each computer. One of the computers runs the executable in the transmitter mode and the other in the receiver mode.

Here is the usage of the program:

```
$ <PROGRAM> <SERIAL_PORT> <ROLE> <FILE>
- Program: binary file to be executed
- SERIAL_PORT: path to the serial port
- ROLE: 'rx' for the receiver, 'tx' for the transmitter
- FILE: name of the file to create/transfer (transmitter/receiver)
```

Main Use Cases

As was already specified earlier, the program can be executed in the transmitter or receiver modes. The function call sequence for each of these modes is different.

Transmitter

1. ***llopen()*****:** establishes connection between the transmitter and the receiver by exchanging set and acknowledgement messages.
2. the program then fetches all the data from the file to transmit.
3. ***createControlPacket()*****:** generates the start control packet.

4. ***llwrite()*****:** generates and transmits an information frame, with the start control packet inside it. After sending, it waits for a supervision frame from the receiver, which indicates if the packet was well received or not.
5. ***createDataPacket()*****:** generates the data packet.
6. **llwrite():** generates and transmits an information frame, with the data packet inside it. After sending, it waits for a supervision frame from the receiver, which indicates if the packet was well received or not.
7. ***createControlPacket()*****:** generates the end control packet.
8. ***llwrite()*****:** generates and transmits an information frame, with the end control packet inside it. After sending, it waits for a supervision frame from the receiver, which indicates if the packet was well received or not.
9. ***llclose()*****:** terminates the connection between the transmitter and the receiver by exchanging set and acknowledgement messages.

Receiver

1. ***llopen()*****:** establishes connection between the transmitter and the receiver by exchanging set and acknowledgement messages.
2. ***llread()*****:** state machine that validates all types of received frames and sends a supervision frame with either a reject or acceptance control field.
3. ***parseControlPacket()*****:** checks if it is a start control packet and returns the size of the file to be received.
4. ***llread()*****:** state machine that validates all types of received frames and sends a supervision frame with either a reject or acceptance control field.
5. ***parseDataPacket()*****:** validates and returns the data from the data packet.
6. the program checks if the control field of the received data packet is an end signal (0x03) and, if so, skips to **8**.
7. the program writes the received data to the output file.
8. ***llclose()*****:** terminates the connection between the transmitter and the receiver by exchanging set and acknowledgement messages.

Link Layer Protocol

The Link Layer is the layer that directly interacts with the serial port and that is responsible for the communication between both nodes. To achieve this, we implemented the *Stop-and-Wait* protocol to establish and terminate the connection, as well as, send supervision and information frames.

In the implementation of this layer, four auxiliary data structures were used:

- ***LinkLayer*****:** stores the parameters associated with the data transfer.
- ***LinkLayerRole*****:** stores the mode of the node.
- ***State*****:** stores the state of the transmission or reception of supervision and information frames.
- ***Statistics*****:** stores the statistics related to the protocol.

```c
typedef struct {                       typedef enum {
                                           START,
    char serialPort[50];                   FLAG_RCV,
    LinkLayerRole role;                    A_RCV,
    int baudRate;                          C_RCV,
    int nRetransmissions;                  BCC_OK,
    int timeout;                           STOP
} LinkLayer;                            } State;

typedef struct {                       typedef enum {
    double open_time;                      LLTX,
    double data_time;                      LLRX
    double debit;                      } LinkLayerRole;
} Statistics;
```

The implemented functions were:

```c
// Open a connection using the "port" parameters defined in struct linkLayer.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize.
int llwrite(int fd, LinkLayer connectionParameters, const unsigned char *buf,
int bufSize);

// Receive data in packet.
int llread(int fd, LinkLayer connectionParameters, unsigned char *packet);

// Close previously opened connection.
int llclose(int fd, LinkLayer connectionParameters, int showStatistics,
Statistics stats);

// Handles byte stuffing on the data
unsigned char* byteStuffing(const unsigned char* buf, int bufSize, int*
stuffedBufSize);

// Handles byte destuffing on the data
unsigned char* byteDestuffing(const unsigned char* stuffedBuf, int
stuffedBufSize, int* destuffedBufSize);
```

The connection is established in the ***llopen*** function. After opening and configuring the serial port, the transmitter sends a SET supervision frame and waits for a UA supervision frame. When the receiver gets the SET frame, it sends the UA frame. If the transmitter receives the UA frame then the connection is successful, if not, the transmitter sends another SET frame. The maximum number of retransmissions and the time for a timeout to occur are defined in the *main.c* file. After establishing the connection, the transmitter starts sending information frames for the receiver to read.

The ***llwrite*** function is responsible for sending the information frames. This function receives a packet that can be a control or data packet, and applies the *byte stuffing* strategy to it. This has to be done, because packets can have bytes that are either 0x7E or 0x7D, which are FLAG and ESC, respectively. The problem with these bytes is FLAG marks the end of a frame, so if the ***llread*** function receives a information frame with a packet that has a FLAG byte, it will think the frame ends in this byte. So, we put an ESC byte before every FLAG or ESC byte and change that byte to the result of Byte XOR 0x20. The transmitter then waits for the response of the receiver. If the frame is rejected then it is transmitted again until it gets accepted or the maximum number of retransmissions is exceeded. Each attempt has a time limit after which a timeout occurs.

The ***llread*** function handles the reading of the information that is sent by the transmitter. It reads an information frame received through the serial port and validates it. To accomplish this, it starts by *destuffing* the received packet and BCC2 and validates both BCC1 and BCC2. This checks if any errors ensue during the transmission.

The ***llclose*** function terminates the connection between the two nodes. The transmitter calls it when it finishes sending all the data or when the number of retransmissions is reached. The receiver calls it after receiving the end control packet that is sent by the transmitter before it calls ***llclose***. A DISC supervision frame is sent by the transmitter, which then waits for the receiver to respond with a DISC frame as well. After the receiver sends this DISC, the program terminates. The transmitter after receiving the DISC from the receiver, sends a UA supervision frame and also shuts down the connection.

Application Layer Protocol

The application layer is the layer that interacts with the user and with the files being read and written. This is where the LinkLayer struct is defined. The file transfer is done through the Link Layer API, but the data and control packets being sent are generated and parsed by the application layer.
Firstly, it calls the ***llopen*** function to establish connections. Then the data transfer starts.

On the transmitter side, after the start control packet is successfully sent, a chunk of data with the max payload size is read from the file and a data packet is created. This is done repeatedly until all data from the file is sent. Then the end control packet is generated and sent. All of this is sent using ***llwrite***.

On the receiver side, it first waits for a start control packet, and then it receives multiple data packets and writes that data into the file, until it gets the end control packet. All of this is read using ***llread***.

After that, the ***llclose*** function is called. If an error occurs in any of the calls to the Link Layer API's functions, the llclose is called and the program is terminated.

```
// Application layer main function.
void applicationLayer(const char *serialPort, const char *role, int baudRate,
                      int nTries, int timeout, const char *filename);

unsigned char* createControlPacket(unsigned char controlField, unsigned long
long* packetSize);

unsigned char* createDataPacket(unsigned char* data, unsigned int* packetSize);

int parseControlPacket(unsigned char* packet, unsigned int packetSize, unsigned
long long* fileSize);

int parseDataPacket(unsigned char* packet, unsigned int packetSize, unsigned
char* data);
```

Validation

To ensure the correctness of the implementation of the protocols developed, the following tests were made:

- transferring files using different *baudrates*.
- transferring files with different names.
- transferring files with different sizes.
- transferring data packets with different sizes.
- total or partial interruption of the serial port connection.
- noise interference in the serial port.

The *Stop-and-Wait* protocol that was implemented insured the consistency of the transferred file. The tests were conducted in the presence of the professor in the project presentation that took place in the lab class.

Link Layer Protocol Efficiency

Varying the *baudrate*

Using a file of 10968 bytes and a fixed maximum frame size of 1000 bytes, the results of varying the baudrate were the following:

***Baudrate* (bits/s)**	**Time (s)**	**Debit (bits/s)**	**Efficiency (%)**
9600	16.99144	5515.937	57.46
19200	16.99000	5516.067	28.73
38400	15.81151	6067.134	15.80
76800	15.80960	6067.719	7.90
115200	15.81028	6067.169	5.27

Looking at the data in this table, we can observe that the efficiency and the time is inversely proportional to the baudrate. Also, as we increase the baudrate, the program's debit and total time taken don't fluctuate much.

Varying the maximum frame size

Using a file of 10968 bytes and a fixed baudrate of 38400 bits per second, the results of varying the maximum frame size:

Frame size (bytes)	Time (s)	Debit (bits/s)	Efficiency (%)
1000	15.81151	6067.134	15.80
1500	15.71425	6116.139	15.93
2000	15.71582	6115.542	15.93
2500	15.71472	6115.797	15.93

Looking at the data in this table, we can observe that, although there is some improvement in increasing the size from 1000 to 1500 bytes, the efficiency stays pretty much the same.

Varying the Frame Error Ratio (FER)

Using a file of 10968 bytes, a fixed baudrate of 38400 bits per second and a fixed maximum frame size of 1000 bytes, the results of varying the FER were the following:

FER (%)	Time (s)	Debit (bits/s)	Efficiency (%)
10	15.74548	6115.856	15.93
20	17.86669	5574.248	14.52
30	18.92740	5484.984	14.28

Looking at the data in this table, we can observe that the link layer protocol's performance gets worse as the FER increases. This means that the efficiency and the debit are inversely proportional to the FER and that the time taken is directly proportional to the FER.

Conclusions

Finally, the creation of this data link protocol allowed us to solidify the concepts we had studied in class. While working on this project, we gained additional knowledge about how a layered architecture works, what is byte stuffing, and the operation of the Stop-and-Wait protocol, as well as how it detects errors.