First Project
RCOM 23/24

Miguel Marinho – 202108822

Rúben Fonseca – 202108830

Summary

This work was carried out within the scope of the course RCOM, where the objective was to implement a data link layer protocol, according to the specification provided in the project description file. We managed to achieve a successful transfer of the penguin file even though the error handling didn't work fully as intended.

Introduction

The project involves creating a data link layer protocol that facilitates the transfer of a computer file between two connected computers using an RS-232 serial cable. This protocol should have both transmitter and receiver functionality. Additionally, a simple data transfer application was to be developed to test this protocol. In summary, the project aims to build a protocol for file transfer and a test application that utilizes this protocol, enabling communication between computers via an RS-232 serial cable.

This report is divided into these sections:

- **Architecture** - "Functional blocks and interfaces"
- **Code structure** - "APIs, main data structures, main functions, and their relationship with the architecture."
- **Main use cases -** "Identification; function call sequences."
- **Logic connection protocol -** "Identification of key functional aspects; description of the implementation strategy for these aspects with the presentation of code excerpts."
- **Application protocol -** "Identification of key functional aspects; description of the implementation strategy for these aspects with the presentation of code excerpts."
- **Validation - "**Description of the tests performed with a quantified presentation of the results, if possible.**"**
- **Efficiency - "**Statistical characterization of the protocol's efficiency, performed using measurements on the developed code.**"**
- **Conclusions - "**Conclusion"

Architecture

Functional blocks

In this project there are 2 different layers, the Data Link layer and the Apllication layer.

The files LinkLayer.c and LinkLayer.h represent the Data Link layer, while the ApplicationLayer.c and ApplicationLayer.h represent the ApplicationLayer.

The **Data Link Layer** is tasked with both establishing and terminating the connection between the transmitter and the receiver. Within this layer, we also handle the packaging of data into packets and process messages from either the transmitter or the receiver. Additionally, error handling is a key function of this layer.

The Application Layer plays a crucial role in sending data from a file and writing data to a file. It's responsible for segmenting the file into smaller units and forwarding this data to the Link Layer for transmission to the receiver. On the receiver's side, the Link Layer retrieves the data from its counterpart and assembles it into a new file based on the received information.

Interface

In the command line, it is possible for the user to run either the transmitter or the receiver using the same binary. For this, they must specify if they want to be the receiver, or the transmitter and which serial port will be used. They also need to specify either the name of the file that they want to send, or the name of the file that will be created. There are also other optional values like Baudrate and the number of tries to send a packet.

Code Structure

LinkLayer

This layer has a data structure that stores the used serial port, the baudrate, the number of retransmissions and the time it has to wait for a packet.

```
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

There is also another structure used in the state_machine file, which is responsible for handling the state machine.

```
struct state_machine
{
    enum mode mode;
    enum state state;
    enum type type;
    unsigned char a;
    unsigned char c;
    unsigned char bcc;
    unsigned char data[DATA_SIZE];
    int curIndx;
    unsigned char bcc2;
};
```

The main functions for this layer are:

```
int llopen(LinkLayer connectionParameters);
int llwrite(const unsigned char *buf, int bufSize);
int llread(unsigned char *packet);
int llclose(int showStatistics);
int writeInfo(const unsigned char *buffer,int bufSize);
void setStateMachine(enum mode mode, enum type type);
void stateMachine(unsigned char byte);
```

Application Layer

This layer only has one function worth mentioning and no data structures. The other functions are auxiliary functions that build the packets.

```
void applicationLayer(const char *serialPort, const char *role, int
baudRate, int nTries, int timeout, const char *filename);
```

Main use cases

There are two main use cases. You either run as a receiver or a transmitter. Like we said before, you can run both of these cases using the same binary, so the user has to specify what case they want to use.

In the case the user selects the **transmitter**, the execution flow would look like this:

1- ApplicationLayer(). In this function, it is determined whether the role is of a transmitter or receiver, and then acts accordingly. The entire flow will basically run inside this function with some calls to functions in the link layer.
2- Llopen(). This function establishes the connection to the serial port.
3- Llwrite(). This function will write the data received from the file fragments to the serial port, so they can be read by the receiver and then interprets the response. It also calls the writeInfo() function wich assembles the packet and then sends it. After that, it then calls the state machine to check the validity of the response.
4- Llclose(). After the program finishes reading the file, it calls llclose() to close the connection to the serial port.

In the case the user selects the **receiver**, the execution flow would look like this:

1. ApplicationLayer(). In this function, it is determined whether the role is of a transmitter or receiver, and then acts accordingly. The entire flow will basically run inside this function with some calls to functions in the link layer.
2. Llopen(). This function establishes the connection to the serial port.
3. Llread(). This function will read and interpret the data received from the transmitter. It calls the state machine to interpret the packet and then builds a response based on the resuts given by the state machine.
4. Llclose(). After receiving a DISC command or getting an END packet, the program leaves the loop where it was building the file and calls llclose() to close the connection.

Logic Connection Protocol

This layer is the one that interacts the most with the serial port.

To establish the connection, we use the llopen() function. It initially opens and configures the serial port according to the parameters we selected, and the ones given by the user when starting the program. After that, the transmitter sends a SET command and awaits a UA command sent by the receiver.

To write data, we use the llwrite() function. Here we call the writeInfo() function to assemble the packet, stuff the bytes in order to avoid conflicts and then send it. After sending the packet, it waits for a response given by the receiver, so the program knows if it has to send the packet again, close the connection or advance to the next packet.

To read data, llread() is used. We use the state machine to detect any errors in the packet. If it detects any error in the BCC2, it sends a REJ to the transmitter, if the packet is the same as the last one, it notifies the transmitter that the packet was repeated. If it has no errors, the receiver sends a command telling the transmitter to send a new packet.

To close the connection, we use llclose(). If no errors occurred, the transmitter would send a DISC and will wait for another DISC sent by the receiver. The transmitter then sends an UA and closes the connection. After getting the UA, the reiver also closes the connection. When an error occurs, the transmitter will leave the loop where it reads the file and enter llclose(), where it will do same thing as if an error never occurred. The difference lies in the receiver, where it will get the DISC inside llread() and will then leave with a forced var set to True. It then skips the waiting for a DISC and writes a DISC to the receiver. After that, it works identically to the no errors case.

Application Protocol

This layer is the one that interacts directly with the user and the file. In this layer, it is possible to define the configuration of the serial port, which serial port to open and the name of the file to be copied or created.

After the program calls lllopen(), the transmitter will first open the file and get its size. It will then create a control packet with the name of the file and its size by calling the function getControlPacket. It then sends the packet to the receiver via the linklayer API. After this, the program will start reading the bytes from the specified file, calling the buildDataPacket() function to build data packets and will send them to the receiver. Once it finishes reading the file, it calls getControlPacket() and creates an end packet to tell the receiver that there is no more information to receive.

The receiver gets inside a loop receiving the packets from the transmitter. If it gets a data packet, it will parse the data and write it to a new file. If it gets an end packet, it will quit the loop.

The connection between the receiver and the transmitter is disabled when both call the llclose() from the link layer API.

Validation

In order to guarantee the correct implementation of our protocol, these tests were done:

- Regular test with the penguin file.
- Changed the names of the files.
- Tested with partial or total interruption of the serial port.
- Tested with different files.
- Tested with noise.

In our internal testing, all of the cases mentioned above were successful but during the presentation, the noise test failed.

Efficiency

Baud rate variation

With a file of size 10968 bytes and a packet of size 1000 bytes, we got these results

Baud Rate	Time (s)	T frame (bits/s)	Efficiency
9600	14.2	6179	64%
19200	13.75	6381	33%
38400	13.23	6632	17%
76800	12,88	6812	8,87%

These values were not the ones we were predicting. We expected the time to decrease significantly as the Baud Rate goes up, but as you can see from our numbers, it barely changed.

Packet size variation

With a file of size 10968 bytes and a baud rate of 9600, we got these results

Frame size	Time (s)	T frame (bits/s)	Efficiency
300	14.6	6010	62%
600	13.18	6657	69%
1000	14.12	6214	65%
1500	13.84	6340	66%

These values were also not the ones we were predicting. We expected the time to decrease significantly as the frame size goes up, but as you can see from our numbers, the values are all over the place.

Error rate variation

We didn't do this test, since our error handling wasn't working as expected. However, we predict that the efficiency would go down as the error rate increased, since the program would spend more time resending packets.

Conclusions

Overall, this project presented several challenges, particularly in the debugging phase. While we successfully achieved file transfers under certain conditions, we encountered difficulties in establishing a robust error handling mechanism for the noise tests. Nevertheless, we believe that this project was valuable in terms of solidifying our understanding of the concepts covered in our coursework.