

**1º TRABALHO LABORATORIAL**

**Autores**:

Pedro

Guilherme

Paula Arias Fernández

**Turma**: 3LEIC06

**UC**: RCOM

The main goal of this project is to implement a data link protocol, which provides reliable communication between two systems connected by a serial cable. It includes both transmitter and receiver. Data is organized into a set of frames, whose start and end are delimited by flags. Every frame has a header which includes several information, such as the data size. It involves positive acknowledgement transmitted by the destination station to signal the successful frame reception. In terms of error control, stop and wait architecture is used. The transmission is organized into three types of frames, Information(I), Supervision (S) and Unnumbered (U).

This project has helped us to apply the contents studied in the theoretical classes, even though it was sometimes hard. We found some problems in terms of programming different parts, and a lot of errors appeared when debugging the code, but finally we achieved the main purpose demanded, although it may not be the most efficient code, and some functions may not be as good as they could.

1. **Introduction**

All the code is divided into the link layer and the application layer. The use of each one of them will be extensively described in the following sections. In the first one, we explain the auxiliar functions we created, and their specific uses. Then, we explain how functions are called one to another, so that the application layer manages all the process. Since we are explaining in detail all the functions implemented, sections which refer to implementation details are included in the rest.

1. **Architecture and code structure**
   1. **Application Layer**

The only enumeration we use here is the CONTROL\_TYPE, which can be START, END or DATA. The function *createControlPacket* receives the type, the fileSize, and a dynamic array where the information of the control packet will be saved. It writes in the control packet the type, and the bytes needed for that specific file size. It is organized into packets of 256.

The function *readControlPacket* receives the type, fileSize and controlPacket by reference rebuild the information extracted in the last function, in the opposite direction.

The function *createDataPacket* receives the *sequenceNumber*, the *fileSize*, the *dataPacket* and the *inputData*. It saves the control field (DATA), the sequence number and the number of octets. Then, it saves the input data after the rest of the attributes.

The function *readDataPacket* interprets the saved information as the previous function, in the opposite way.

The function *findSize* helps calculating the size of a file. It does it by setting the file position of the stream to the end position (with fseek) and then counting the position in which it is (with ftell). It returns the result obtained.

The function *setupTransmitter* establishes the transmission by calling *llopen* (in the link layer), then creating the control packet and sending the first control packet with *llwrite*.

The function *setupReceiver* works similarly, opening the transmission with *llopen*, reading the control packet received and indicating it with a message.

* 1. **Link Layer**

Firstly, we define the enumerations. We have an ENUM for the two possible types of machines: transmitter and receiver. Also, we have one for the type of header that frames could have: INFO, SET, DISC, UA, RR, REJ or INVALID. Finally, we have one for flag and escape in byte stuffing (FLAG, ESC\_FLAG, …), addresses (A\_TRANSMITTER\_COMMAND, A\_RECEIVER\_COMMAND), control commands (CNTRL\_INFO\_0, CNTRL\_INFO\_1, CNTRL\_UA, …) and control responses, with the respective hexadecimal codes for each.

On the other hand, the functions in the link layer are called by the application layer and describe more extensively how things are working.

The function ***getBCC*** receives a dynamic array with the content, and an integer with the size. It calculates the result, returned by the function, with an XOR of the result and the content of the array passed, from one to the size given.

The function ***getHeaderType*** receives the header and the *responseParity*. It checks the BCC given by the previous function, returns the type of header found and fills the corresponding parity with the address and control. For example, if the transmitter is receiving, and we have A\_TRANSMITTER\_COMMAND and CNTRL\_REJ\_0; the *responseParity* is set to 0 and REJ is returned. In case of error, it returns INVALID.

The function ***createHeader*** defines the array mentioned before, filling the command and the response depending on the case. For example, if type is SET, we have the A\_TRANSMITTER\_COMMAND (as it is not UA) and CNTRL\_SET.

The byte stuffing is done and undone with ***addStuffing*** and ***removeStuffing*** functions. What *addStuffing* does, firstly, is to add one to the size if it finds a FLAG or an ESCAPE, so that the respective ESC\_FLAG or ESC\_ESCAPE can be added in a subsequent loop. It finally copies the info to an array which is passed by reference. The way removeStuffing works is basically the opposite. It reduces the size in one every time it finds a FLAG or an ESCAPE.

The function ***receivePacket*** implements the state machine with the states explained in the following paragraph. It starts with WAIT\_FOR\_FLAG, where it reads the bytes received until it finds a FLAG. Then, it passes to BUILDING\_HEADER, where it fills the first three bytes with the ones received, and then uses the *getHeaderType* function to reset if it is INVALID, to switch into FILLING\_INFO if it is an information packet, or into WAIT\_FOR\_LAST\_FLAG in other case. In this last state, it waits for a flag, in which case it would switch to OVER and finish the loop, and in other case, it would continue waiting for the flag. If it is FILLING\_INFO, it adds the bytes received to the data array, until it finds a flag, when it switches to INFO\_FILLED. The purpose of this state is to remove the stuffing made in *sendPacket* (as explained before), and, in the case the BCC goes as expected, it switches to WAIT\_FOR\_LAST\_FLAG. Otherways, the state is over, which exits the loop.

The function ***sendPacket*** is implemented with a loop. In the first place, it creates the header, and checks that it is correct. It saves all the data, BCC and FLAG in the corresponding variables, and waits for three seconds to receive the answer. If it receives UA, RR or REJ, the acknowledge is 1, so it would not have a next step in the loop. It calls *receivePacket* to read the response. If type is set and it receives an UA, acknowledge is 1. If the type is DISC, it waits for another DISC if it is the transmitter, or an UA if it is the receiver. If type is INFO and response is REJ, it passes to the next loop step, and if it is RR and *parityReceived* is different from *messageParity*, the acknowledge is 1, so the loop finishes and the parity changes.

The function *llopen* receives the connection parameters and sets the machine and parity corresponding to them. It opens the serial port, and if it is the transmitter starts the transmission by sending a SET to the receiver. If the receiver is performing, it calls *receivePacket* and sends the corresponding UA response, so that the confirmation is received.

The function *llwrite* only sends the information in case that the machine is the transmitter, in which case it sends the INFO packet with all the data to send.

The function *llread* only receives the information in case that the machine is the receiver. It calls *receivePacket*, and if the type is INFO and the parity received is different from the message parity, it changes the message parity and returns the size of the message. In case the type is DISC, it sends the packet with another DISC, and returns 0.

Finally, the function llclose (if the machine is the transmitter) sends a packet with the DISC signal, and receives the DISC response, sending another UA response to terminate the connection.

1. **Main use case**

When it comes to passing the file penguin.gif, the main file starts establishing the connection, and calling the application layer. Firstly, we initialize the parameters given, such as the serial port, the role, … Then, in the case of the transmitter, we call *setUpTransmitter* (explained before), and we continue reading and creating the data packets until the file ends. We call *llwrite*, so that the frames are correctly sent, and we write the corresponding error messages if things don’t go as expected. Once we have created the data packet, we create the control packet, and send it with *llwrite* to the receiver. Finally, we end up closing the transmission with *llclose*.

On the other hand, if we handle the receiver, we open the penguin, and setup the parameters with the predefined function *setupReceiver*. We do a loop in which we call llread to read the data that is being sent. We read all the packets and print the messages to inform the user how the process is going. We repeat this until the last frame, in which closes the file and finishes the retransmission.

1. **Validation**
2. **Data link protocol efficiency**
3. **Conclusions**

In conclusion, we could accomplish the main functions required in the project. We can send the penguin and receive it correctly, including all the coding stuff it implies. However, some of them weren’t in our final prototype, as it gave lots of problems when making the tests. For example, our code does not manage the noise, as we did not implement REJ.