Data Link Protocol

Redes de Computadores 2024-2025

1st Lab Work

3LEIC09

Gabriel da Quinta Braga (up202207784)
Guilherme Silveira Rego (up202207041)

November 10, 2024

Summary:

This report presents the implementation of a data link protocol over RS-232 communication for file transfer between two computers.

The project applies knowledge gained from both theoretical and practical coursework, including framing, connection establishment and termination, frame numbering, acknowledgment, Stop-and-Wait error control, and flow control, implemented in C.

Introduction:

The objective of this work is to establish reliable communication between two systems connected via RS-232 to facilitate file transfer according to the provided specifications. This report is organized into eight sections:

- **Architecture**: Overview of the program's functional blocks and interfaces.
- **Code Structure**: Description of APIs, primary data structures, and main functions.
- **Main Use Cases**: Identification of primary use cases and associated function call sequences.
- **Logical Link Protocol**: Key functional aspects and the implementation strategy.
- **Application Protocol**: Principal functional aspects and the corresponding implementation strategy.
- **Validation**: Summary of the tests performed to verify functionality.
- **Data Link Protocol Efficiency**: Statistical analysis of the protocol's efficiency.
- **Conclusions**: Summary and reflections on the project outcomes.

Architecture:

Functional blocks

The project architecture is organized into two primary functional blocks: the Application Layer and the Link Layer.

The **Application Layer** facilitates user interaction with the program, allowing users to configure parameters such as the serial port, baud rate, and execution role (either receiver or transmitter) as well as the filename. Here, the receiver specifies the filename for the incoming data, while the transmitter selects the file to be transferred. The Application Layer components are defined in application_layer.h and implemented in application_layer.c.

The **Link Layer** implements the core file transfer protocol, managing essential operations such as connection setup and termination, framing, and Stop-and-Wait mechanisms. This layer is defined in link_layer.h and its functionality is implemented in link_layer.c.

Interfaces

In two separate terminals, the program is executed with one terminal assigned the transmitter role and the other assigned the receiver role, like the following command on the code folder:

```
$ <PROGRAM> <SERIAL PORT> <BAUDRATE> <rx | tx> >FILENAME>
```

Code Structure

Application Layer

The layer with the main function and responsible for the interaction between the user and link layer:

```
// Auxiliary function to update display of the progress on the written
content from the transmitter to the receiver
void updateProgressBar(int bytesWritten, int fileSize);

// Auxiliary function used by the transmitter to send control packets
(start, data or end) to the receiver
int sendControlPacket(int type, const char *filename, int fileSize);

// Auxiliary function used by receiver to read the control packets sent
by the transmitter
int readControlPacket(int type, unsigned char *buffer, size_t *fileSize,
char *filename);

// Auxiliary function used by the transmitter to send the data packet
(invokes llwrite)
int sendDataPacket(unsigned char *buffer, int contentSize);

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

Link Layer

The layer responsible for the actual data writing/reading, where we have been given two auxiliary data structures:

```
// Data structure to identify which role the current program is in
typedef enum
{
    LlTx,
    LlRx,
} LinkLayerRole;
```

```c
// Data structure with crucial information to the data transfer
typedef struct
{
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

And the implemented functions:

```c
// Alarm function handler
void alarmHandler(int signal);

// Function that prints the statistics of the transfer
void printStats(role)

// Function to establish connection between the two computers
int llopen(LinkLayer connectionParameters);

// Function to write data to the serial port
int llwrite(const unsigned char *buf, int bufSize);

// Function to read data from the serial port
int llread(unsigned char *packet);

// Function to close connection between the two computers
int llclose(int showStatistics);
```

Main Use Cases

The program's execution varies significantly depending on the computer's role as either a transmitter or receiver, with specific functions called based on this role. However, two functions are common to both roles:

- **llopen()**: Establishes a connection between the two computers by invoking openSerialPort() to open and configure the serial port.
- **llclose()**: Closes the connection by calling closeSerialPort() to restore the original port settings and close the serial port.

Transmitter:

- **sendControlPacket()**: Initializes and constructs a control packet based on the specified type, then calls llwrite() to transmit it over the serial port.
- **sendDataPacket()**: Initializes and constructs a data packet with buffer contents, then calls llwrite() to send it to the serial port.

- **llwrite()**: Constructs a frame following the specified protocol structure, including the header, BCC2 calculation, and byte stuffing. The function then writes the frame to the serial port and waits for the receiver's response (either RR or REJ) using a state machine.
- **updateProgressBar()**: Creates a visualization of the file transmission progress based on the bytes already written.

Receiver:

- **readControlPacket()**: Calls llread() to receive the control packet sent by the transmitter, verifies its type, and parses it if it matches the expected packet.
- **llread()**: Reads data from the serial port as sent by the transmitter, processes it through a state machine, and verifies it with BCC2. If the check succeeds, the function sends an RR acknowledgment; otherwise, it sends a REJ to request retransmission.

Logical Link Protocol

Here we will talk about the logic behind our code in order to make it fulfill what it is intended to do, following the protocol. As already stated before, the link layer is the layer responsible for interacting directly with the serial port, making it responsible as well for the entire actual communication between the transmitter and the receiver. Through the entire process of sending and receiving data on both computers we used the *Stop-and-Wait* technique we learned in the theoretical classes to ensure the preservation of data without losses.

Upon the execution of the program on two different computers, both of them will call the **llopen** that, as said before, establishes the connection through the serial port between them:

1. First, we start by calling the given function **openSerialPort** defined on serial_port.h and implemented on serial_port.c that opens and configures the serial port with the parameters given on the application layer.
2. With the serial port open and configured, the transmitter sends a SET supervision frame that informs the initiation of the connection, and waits for the receiver's response.
3. When the receiver gets the frame, it answers with a UA supervision frame, to confirm the reception of a valid supervision frame.
4. If the transmitter receives the UA frame, the connection has successfully been established.

After this, the transmitter can now start to send the information of the file he wants to transmit, and it will do it through the **llwrite** function, that receives a data packet or a control packet and does the following:

1. Constructs the information frame that it will write to the serial port, with the correct specifications given by the protocol (*framing*).
2. Uses the byte stuffing technique to ensure that there are no errors (data bytes equal to the flag or bit errors).
3. Writes it to the serial port and waits for a receiver's response.
4. If the response is a REJ (reject, a supervision frame rejecting the frame), the transmitter will rewrite the frame to the serial port. We will repeat this process until the transmitter gets a RR (receiver ready, a supervision frame confirming the frame information has been received without errors) or we exceed the pre-established number of retransmissions or if there's a timeout.

The reading of information frames is done by the receiver using the **llread** function, that:

1. Reads the information written by the transmitter on the serial port
2. Verifies it by making the *destuffing* and validating the BCC1 and BCC2.
3. If everything went on without any errors, the receiver writes to the serial port RR, otherwise, REJ.

Finally, the connection's termination is done by the **llclose** function that works similarly to the **llopen** function:

1. The transmitter sends a DISC supervision frame to inform the receiver of the termination of the connection, and waits for the receiver's response.
2. When the receiver gets the DISC frame, it sends back a DISC frame too.

3. Upon getting the receiver's response, the function calls an auxiliary function that prints the statistics of the file transfer, including the total runtime, frames sent, frames received, and data transfer time. These statistics will change depending on the computer's role.

Application Protocol

The application layer is the layer responsible for the interaction between the file being transferred, the user, and the link layer. It is also in the layer that receives the connection parameters, that define the connection's baudrate, the maximum time to wait for packet reception until it times out (timeout), the maximum number of retransmissions, and, most importantly, the file being transferred, its name, and its size. The transmission of every byte from the file is handled by the link layer, sent in packets inside information frames.

The first function called in the application layer is **llopen**. This function, as already explained before, receives the connection parameters and sends out from the transmitter's side an initial frame, which is met by a response frame from the receiver's side, signaling if the connection was successfully established. After this initial 'handshake', the transmitter reads the file's size and name, and proceeds to send an initial control packet calling out the function **sendControlPacket**, which sends a control packet in the TLV format, meaning Type, Length and Value. In TLV1 we store the file's size and in TLV2 its name. This packet is sent out recurring to the **llwrite** function responsible for writing a frame through the serial port, and received by the **llread** function in the receiver's side, both functions implemented in the Link Layer.

The receiver's side reads the control packet using the **readControlPacket** function, and proceeds to open a file for writing the received content in. The transmitter then enters a loop, where it reads the file, and writes its contents into a buffer, which is then sent with the **sendDataPacket** function, that assembles a packet, identifies it as being a data packet in the control field, indicates the length of its contents, and sends it out with **llwrite** until there is no more to read on the file.

Every packet sent by the transmitter, including the initial and final control packets, are met by a response sent by the receiver, confirming the integrity of the sent packets. If the response doesn't match the sent content, the packet is then rejected, and the transmitter immediately resends the same content block, ensuring that the received file exactly matches the file in the transmitter's side.

When the transmitter has sent every data packet with success, it sends a last control packet via **sendControlPacket** that informs this to the receiver. The receiver that is still in the loop to read packets, if it detects that the packet is an end control packet, it leaves this loop.

Validation

To ensure the protocol operates correctly, we tested the program under varying conditions by adjusting the following parameters:

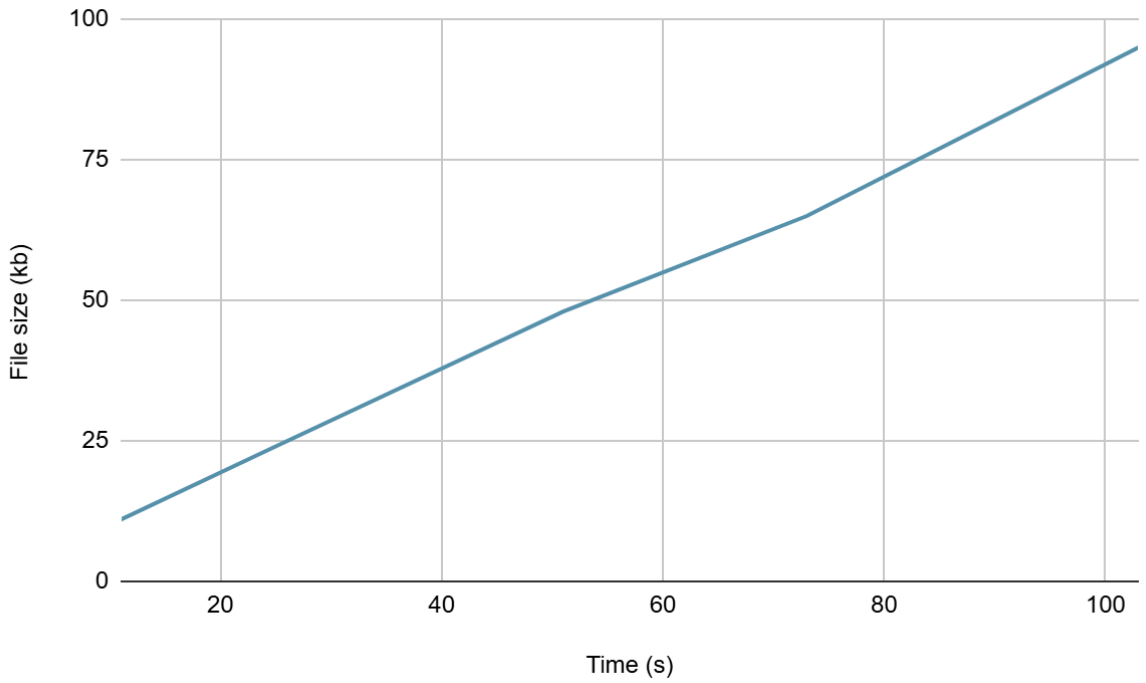
- file size;
- file name;
- baudrate;
- noise;
- interruption of the serial port.

In each scenario, the file transfer was completed successfully, confirming the robustness of the protocol.

Data Link Protocol Efficiency

- File size

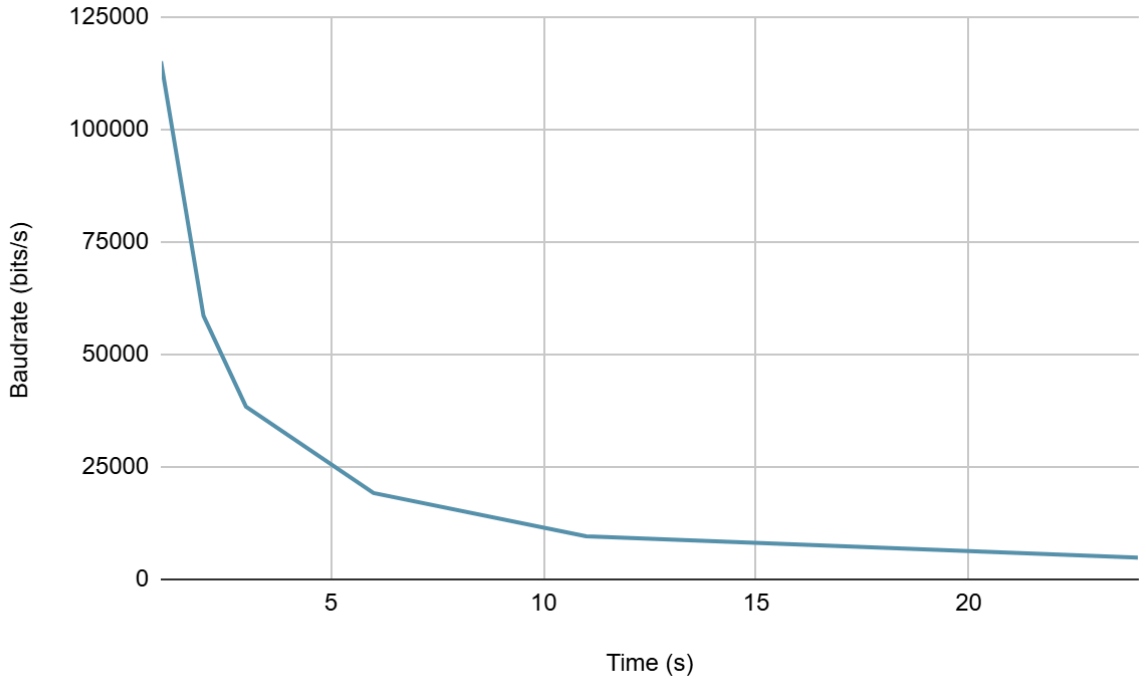
We evaluated the impact of file size on transfer time by varying the file size while keeping other parameters at their default values:



Results indicate an almost directly proportional relationship between file size and transfer time, with a slight increase in the rate of time growth as file size increases. For instance, transferring an 11 KB file took approximately 11 seconds, while a 65 KB file required about 73 seconds.

- Baudrate

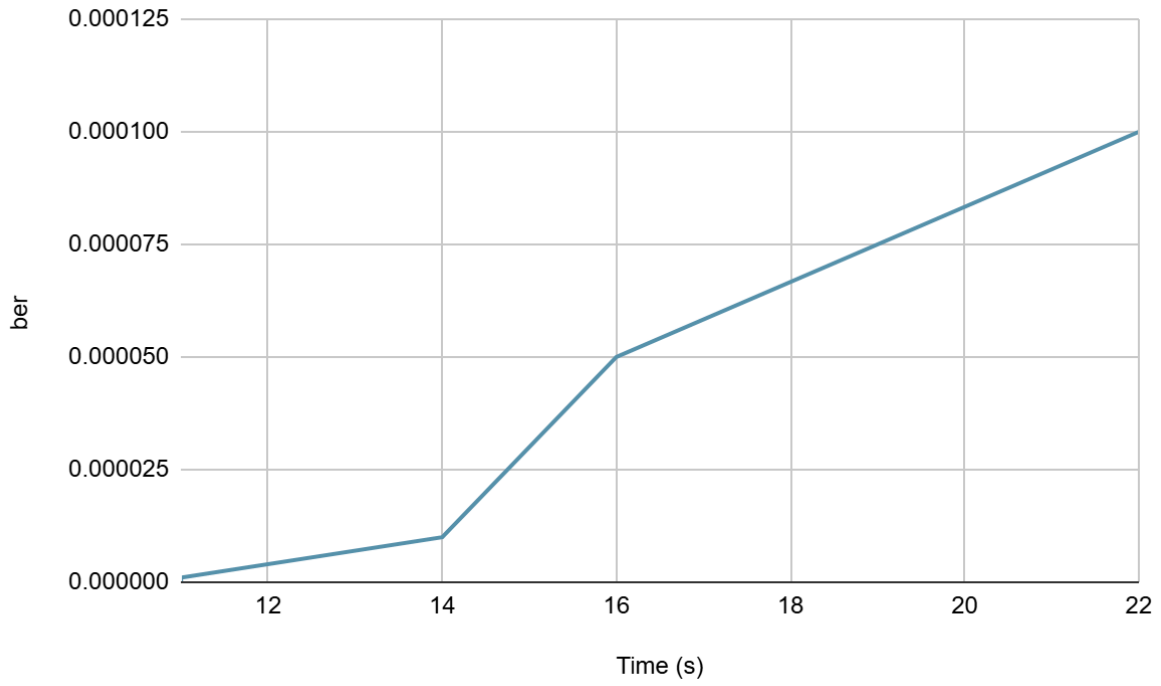
To assess the effect of baud rate on transfer time, we measured the time required to transmit an 11 KB file at different baud rates:



The resulting graph shows an inverse relationship between transfer time (in seconds) and baud rate (in bits per second), revealing that as baud rate increases, the average transfer time decreases exponentially.

- Noise (ber)

Increasing noise on the cable makes it susceptible to more errors on the communication between the transmitter and receiver, increasing therefore the time it takes to transfer a file. We will use the same 11kb file to test with different ber values and see the relation between these values, their impact on their time and quantity of errors:



The data shows a positive correlation between time (in seconds) and the bit error rate (BER), suggesting that as the BER increases, time tends to rise. The gradual increase in time mostly results from the cumulative delays introduced by the Stop-and-Wait protocol. As more errors occur, the protocol will require more retransmissions, effectively prolonging the communication session.

Conclusions

In summary, this report has presented the implementation of a data link protocol over RS-232 communication, applying the Stop-and-Wait protocol to manage error control and flow regulation.

The analysis has shown how baud rate and bit error rate (BER) directly influence transmission efficiency: higher baud rates decrease transmission time, whereas increased BER leads to additional delays due to retransmissions. These results highlight the inherent trade-offs in maintaining reliable data transfer within environments prone to noise, even though the Stop-and-Wait protocol is reliable on data transfer, it takes more time to do it due to its retransmission process.

This work has met the learning objectives by applying the content lectured on the theoretical classes with the implementation in C of concepts like Stop-and-Wait and byte stuffing.