Data Link Protocol

- First Lab Report
- FEUP - Computer Networks
- Alexandre Ramos - up202208028
- Francisco Afonso - up202208115

Summary

This report documents the work developed for the first laboratory of the computer networks course (course?). The objective of the project is to transfer data between 2 machines by a layered application approach and the Stop & Wait protocol.

We achieved our goals, as the application successfully sends the file while having the robustness to withstand noise and temporary termination of the connection betwwen the two machines.

Introduction

In this report, we aim to provide insight into the structure, implementation and testing of the project, as well as performance and efficiency statistics. The objective of the project is to develop an application using the C programming language that transfers files between 2 linux machines connected through a RS-232 serial cable. This is achieved by a layered application approach and the Stop & Wait protocol.

The report is stuctured as follows:

1. Architecture
    - Functional blocks and interfaces.
2. Code Structure
    - APIs.
    - Main data structures.
    - Main functions and their relationship with the architecture.
3. Main Use Cases
    - Identification.
    - Function call sequences.
4. Logical Link Protocol
    - Identification of the main functional aspects.
    - Description of the implementation strategy of these aspects with code excerpts.
5. Application Protocol
    - Identification of the main functional aspects.
    - Description of the implementation strategy of these aspects with code excerpts.
6. Validation
    - Description of the tests performed with quantified presentation of the results.
7. Data Link Protocol Efficiency
    - Statistical characterization of the protocol's efficiency.

    - Theoretical characterization of a Stop & Wait protocol.
8. Conclusions
    - Synthesis of the information presented in the previous sections.
    - Reflection on the learning objectives achieved.

Architecture

The application is divided in the Application Layer, Link Layer and the Serial Port Layer. We were tasked with implementing the first two.

Application Layer

The application layer is the highest level of abstraction, performing the tasks in the most superficial way. It is responsible for connecting and disconnecting the two machines, as well as opening, sending, receiving and closing the file to be transfered.

Link Layer

The Link Layer receives orders from the application layer and executes them. It is responsible for applying the Stop & Wait protocol and assuring data is correcly trasnfered between the 2 machines.

Code Structure

The code we implemented is structured as four files and their respective headers.

application_layer.c

Function

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

Global Variables

```
int framesSent = 0; // total number of frames sent, used for statistics
int alarmTotalCount = 0; // total number of alarms set off, used for
statistics
int rejCount = 0; // total number of rejection commands sent, used for
statistics
extern long fileSize; // total size of the file in bytes, used for
statistics
extern double delta; // total time passed between the connection and
disconnection phases, used for statistics
extern int nRej; // number of consecutive rejection commands sent, used for
terminating the program if necessary
```

link_layer.c

Functions

```c
// Open a connection using the "port" parameters defined in struct
linkLayer, return "1" on success or "-1" on error.
int llopen(LinkLayer connectionParameters);

// Send data in buf with size bufSize, return number of chars written, or
"-1" on error.
int llwrite(const unsigned char *buf, int bufSize);

// Receive data in packet, return number of chars read, or "-1" on error.
int llread(unsigned char *packet);

// Close previously opened connection, return "1" on success or "-1" on
error.
int llclose(int showStatistics);
```

Global Variables

```c
extern int alarmEnabled, alarmCount, alarmTotalCount, rejCount; // alarm
control flags and counters for monitoring
extern int iFrame; // identifier for frame numbering in link layer protocol
extern int framesSent; // count of successfully sent frames
extern long fileSize; // total size of the transmitted file
extern double delta; // elapsed time during connection session
int timeout, nTries; // parameters for retransmission timeout and retry
count
LinkLayerRole role; // role of the connection, either transmitter or
receiver
```

Macros

```c
// SIZE of maximum acceptable payload.
// Maximum number of bytes that application layer should send to link layer
#define MAX_PAYLOAD_SIZE 1000

// MISC
#define FALSE 0
#define TRUE 1
```

Structures

```
typedef enum
{
    LlTx, // transmitter role for link layer
    LlRx, // receiver role for link layer
} LinkLayerRole;

typedef struct
{
    char serialPort[50]; // name of the serial port for connection (e.g.,
/dev/ttyS0)
    LinkLayerRole role; // link layer role for this connection (LlTx or
LlRx)
    int baudRate; // communication speed of the connection
    int nRetransmissions; // max retry attempts for retransmission on
failure
    int timeout; // timeout for receiving acknowledgment before retrying
} LinkLayer;
```

state_machines.c

Functions

```
// State machine for opening a connection, transitioning through each
state, returns the current state of the open process
int openStateMachine(State *state, unsigned char *buf, LinkLayerRole role);

// State machine for handling write operations, returns the final state
int writeStateMachine();

// State machine for handling read operations, returns the final state
int readStateMachine(unsigned char *packet);

// State machine for handling disconnect (DISC) control, returns final
control byte
unsigned char discStateMachine();

// State machine for handling acknowledgment (UA) control, returns final
control byte
unsigned char uaStateMachine();
```

Global Variables

```
extern int iFrame; // frame identifier for link layer
extern int alarmEnabled; // flag indicating if alarm is enabled
```

tools.c

Functions

```
// Insert an element at a specific position in an array
void arrayInsert(unsigned char arr[], int *n, int value, int pos);

// Alarm signal handler for managing retry mechanism
void alarmHandler(int signal);

// Write a positive or negative acknowledgment (RR or REJ) based on input,
returns success status
int writeResponse(int rr, int iFrame);

// Write a control packet (start or end) with file info and returns
generated packet
unsigned char* writeControl(long fileSize, const char* fileName, int*
packetSize, int type);

// Write a data packet with sequential data, returns generated packet
unsigned char* writeData(unsigned char* data, int dataSize, int seqNum);

// Calculate power of an integer x raised to y
long power(int x, int y);
```

Global Variables

```
extern int alarmEnabled; // alarm status flag
extern int alarmCount; // counter for active alarms
extern int iFrame; // current frame index for transmissions
extern long fileSize; // total size of the file in bytes
extern double delta; // elapsed time since start of connection session
```

Macros

```
#define FLAG 0x7E // frame boundary flag byte

#define A_RX 0x01 // address byte for receiver
#define A_TX 0x03 // address byte for transmitter

#define C_SET 0x03 // SET control byte for initiating connection
#define C_UA 0x07 // UA control byte for acknowledgment
#define C_RR0 0xAA // RR0 control byte for acknowledgment (frame 0)
#define C_RR1 0xAB // RR1 control byte for acknowledgment (frame 1)
#define C_REJ0 0x54 // REJ0 control byte for rejection (frame 0)
#define C_REJ1 0x55 // REJ1 control byte for rejection (frame 1)
#define C_DISC 0x0B // DISC control byte for disconnect request

#define C_I0 0x00 // information control byte for frame 0
#define C_I1 0x80 // information control byte for frame 1
```

```
#define ESC 0x7D // escape byte for special characters in frame data
#define FLAG_SEQ 0x5E // replacement sequence for FLAG within data
#define ESC_SEQ 0x5D // replacement sequence for ESC within data

#define T_SIZE 0x00 // file size type for control packet
#define T_NAME 0x01 // file name type for control packet

#define P_START 0x01 // packet type for start of transmission
#define P_DATA 0x02 // packet type for data transmission
#define P_END 0x03 // packet type for end of transmission
```

Structures

```
typedef enum {
    START_S = 0, // initial state for state machine
    FLAG_RCV_S, // state after receiving FLAG byte
    A_RCV_S, // state after receiving address byte
    C_RCV_S, // state after receiving control byte
    BCC_OK_S, // state after receiving BCC byte without error
    STOP_S // final state indicating successful end of reception
} State;
```

```
// DAS STRUCTURES DA LINK LAYER PARA A FRENTE FOI TUDO COMENTADO PELO CHAT
GPT
```

Main use Cases

The application is capable of **transferring the file** and composing it in the receiving machine, after **receiving** and **validating** it.

Sending the file

The application is capable of transmitting the file to the other machine by following these steps

Establishing a connection

The transmitter establishes a connection to the other machine by calling `llopen()`. This function runs `openSerialPort()` to open the connection to the other machine, it later sends a `SET` frame with `writeBytesSerialPort()` to request a connection. After it receives the `UA` frame with `readByteSerialPort()` and confirming it with `openStateMachine()` the connection has been established successfully.

Preparing the packet

Before sending the packet throught the serial cable, the application divides the file in various packets. The first type of packet is known as a control packet, it is whether a start packet or an end packet. In this type of packet we signal the receiver various aspects of the file, such as it's name and size. The second type is known as a data packet, and is mainly composed by the bytes present in the original file.

The control packet is prepared by the `writeControl()` function with the `P_START` or `P_END` depending on it's purpose.

The writeData() function prepares the data packet by appending the file's original bytes along with essential information for the entire file transfer. This includes details like the number of bytes in the packet and the packet's sequence number.

Sending the packet

When sending frames between two machines we delimit them with the flag `0x7E`, a problem arises when one of the bytes of the original file is `0x7E`. To solve this inconvenience we apply the byte stuffing technique, we replace every `0x7E` byte with the sequence `0x7D 0x5E` and every `0x7D` with the sequence `0x7D 0x5D`.

After preparing the packets, the transmitter begins to send the packets throught the serial cable. This task is initiated by the `llwrite()` function, where we annex address, control and independent protection fields to the packet. It is also in this function where we perform the byte stuffing technique, with help of the `arrayInsert()` function. The application sends the prepared packet with `writeBytesSerialPort()` and awaits a response from the receiver with the `writeStateMachine()` function.

Receiving the file

The application is capable of receiving the file from the other machine by following these steps

Establishing a connection

The receiver initiates the connection by calling `llopen()`, where it initializes the serial port, and then enters a loop to await a `SET` frame from the transmitter. Once a `SET` frame is received and verified, the receiver responds with a `UA` frame, signaling that the connection has been established. This step concludes with `llopen()` returning the file descriptor for the established connection.

Receiving the packet

When a packet arrives, the receiver uses `llread()` to process it, which calls `readStateMachine()` to read and validate incoming frames. If the packet is received correctly, `llread()` performs a byte-check to confirm integrity. If no data is read, `llread()` calls `writeResponse(TRUE, ...)` to request retransmission. If data is received but with errors, `llread()` invokes `writeResponse(FALSE, ...)` to signal a reject and requests retransmission of the corrupted frame.

The received data is then unpacked from the frames and stored in packet, ready to be reassembled into the final file.

Writing to the file

Once the receiver confirms the integrity of the packet, it writes the data from packet to the output file using `fwrite()`. It continues to read and write each packet sequentially until all packets are received. If a control packet with `P_END` is detected, the receiver knows the transmission is complete and can finalize the file write operation.

Validating the file

After receiving all packets, the application performs several validation checks to ensure data integrity and completeness.

Receiving the frame

Each frame is verified during transmission using `llread()`, which ensures that every packet has a valid frame structure. `llread()` identifies frame boundaries and checks for byte-stuffed sequences. Byte-stuffed sequences, like `0x7D 0x5E` for `0x7E`, are destuffed as needed to reconstruct the original data accurately.

Calculating BCC1 and BCC2

The receiver calculates the Block Check Characters, `BCC1` and `BCC2`, as part of data integrity verification. `BCC1` is derived from applying the `xor` operation tothe control and address fields of each frame, ensuring the frame structure is correct. `BCC2` is calculated over the data portion of the packet to confirm its integrity.

In `llread()`, `BCC2` is calculated by applying the `xor` operation to each byte of the packet data and comparing it with the transmitted `BCC2` byte. If they match, the packet is accepted; otherwise, it's rejected with a `REJ` response, triggering retransmission.

Confirming the values

The final step in validation involves confirming that the calculated `BCC2` and `BCC1` matches their received counterparts, as seen in `llread()`. If they match, the packet is accepted, and `writeResponse(TRUE, ...)` sends an acknowledgment. If they don't match, `writeResponse(FALSE, ...)` triggers a rejection.

Throughout the execution of the program, the application subtracts the size of the file with the size of each packet. After receiving the end packet, it expects the final result of the subtraction to be 0. It throws an error indicating bytes were lost during transmission otherwise.

Logical Link Protocol

The Link Layer interacts directly with the serial port. It is responsible for sending and receiving frames, as well as opening and closing the connection.

llopen()

```
int llopen(LinkLayer connectionParameters);
```

`llopen()` receives the `connectionParameters` to instruct the connection process, it makes use of the following attributes:

- `timeout` for the time the alarm wait before setting off
- `nRetransmissions` for the number it tries to send the same packet before ending the operation
- `role` to execute the appropriate algorithm according to it's objective
- `serialPort` and `baudRate` to configure the `openSerialPort()` function

It begins by setting up the signal according to the `alarmHandler()` function. Afterwards, it uses the function `openSerialPort()` which was provided by the teachers.

If it is called by the transmitter, it builds the `SET` frame, writes it to the serial port and activates the alarm. In case the function `readByteSerialPort()` continues to return 0 bytes read for the remainder of the timeout value provided in the function call, it will trigger the alarm and start the process from the `SET` frame stage. Otherwise, it verifies the `UA` frame sent by the receiver with the function `openStateMachine()`.

If it is called by the receiver, after reading the `SET` frame sent by the transmitter, it verifies it with the `openStateMachine()` function. Given the verification was successful, it writes the `UA` frame to the serial port.

llwrite()

```
int llwrite(const unsigned char *buf, int bufSize);
```

`llwrite()` is called by the transmitter and receives the array to write to the serial port in the form of `buf` and it's respective size in `bufSize`.

It begins by verifying if the buffer provided in the function call has a valid size, ending the execution otherwise. It continues by building the frame, calculating the `BCC2` and byte-stuffing the frame with help of the `arrayInsert()` function. After appending the final `FLAG` byte, it writes every byte to the serial port adn enables the alarm in case of the need for a retransmission. It awaits for acknowledgement from the receiver and return the size of the frame in case of success.

llread()

```
int llread(unsigned char *packet);
```

`llread()` is called by the receiver and receives the array to read in the form of `packet`.

It reads the entire packet, checking if the control bytes were transmitted correctly and de-stuffing it in the process. After checking for possible errors in the reading process, it calculates the `BCC2` and compares it to the one calculated by the transmitter.

In case the calculated `BCC2` matches the one sent by the transmitter, the receiver sends back a frame signaling it is ready to receive the next frame in line. Otherwise, it sends a frame signaling some error occured and requests a retransmission to assure data correctness throughout the process.

llclose()

```
int llclose(int showStatistics);
```

`llclose()` is called by both parties and receives a boolean `showStatistics` that determines whether or not statistics about the process will be displayed to the user.

In case of the transmitter, it begins by building the `DISC`, writing it to the serial port and enabling the alarm in case of a timeout. It proceeds to wait for a frame of the same kind coming from the receiver. After receiving the frame, it sends back a `UA` frame and prints the statistics if requested to do so.

In case of the receiver, it waits and receives the `DISC` frame from the transmitter. To confirm the intention to disconnect from the other machine, it responds with a `DISC` frame of it's own. After receiving the `UA` frame it can end the program and print the statistics to the console if necessary.

Application Protocol

The Application Layer is responsible for managing the file transfer process, interacting with the Logical Link Layer to control data flow, and handling the file's reading or writing based on the transmitter or receiver roles.

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

The `applicationLayer()` is the main function for managing file transmission or reception, and it takes several parameters to guide the process:

- `serialPort` specifies the serial port used (e.g., /dev/ttyS0).
- `role` defines the application role, either tx for transmitting or rx for receiving.
- `baudRate`, `nTries`, and `timeout` configure the connection and retransmission limits.
- `filename` specifies the file to be sent by the transmitter or received by the receiver.

Transmitter role

In the transmitter role, `applicationLayer()` begins by opening the file specified by filename for reading. It then determines the file size and prepares a start control packet using `writeControl()` with `P_START`. The `llwrite()` function is then used to transmit this start packet.

The transmitter then reads the file in chunks, creating and sending data packets with `writeData()` and `llwrite()`, each with a sequence number that increments per packet. Once all data packets are sent, the transmitter prepares and sends an end control packet using `writeControl()` with `P_END`, signaling the receiver that transmission is complete.

Receiver role

In the receiver role, `applicationLayer()` waits for the initial control packet using `llread()` and validates it. Once the file size is confirmed, the receiver opens the specified filename for writing and begins

reading incoming data packets.

The receiver reads each packet sequentially, verifies its integrity with `llread()`, and writes the data to the file until the end control packet is received. If any transmission errors occur, the receiver requests retransmissions until the data is correctly received or the maximum retry limit is reached.

Completion and statistics

After all packets have been sent or received, `applicationLayer()` calculates the total time taken and calls `llclose()` to close the connection. If `llclose()` is called with `TRUE`, transmission statistics such as bytes sent, errors encountered, and retransmissions are displayed.

Validation

In order to validate the correctness of our code, we performed various tests, here are the results:

Files with different sizes

Baudrate: 115200 Packet Size: 700

File Size (byte)	Time (s)	T Frame (bit/s)	Efficiency
1650	0,153	86274	74,89%
10968	0,984	89170	77,40%
20832	1,868	89216	77,44%
75076	6,713	89469	77,66%
502549	44,831	89678	77,85%

The results show that as the file size increases, the efficiency of the data link protocol stabilizes around 77-78%. This suggests that larger files experience less variation in efficiency, likely due to reduced overhead per byte transmitted in larger datasets.

Transmission of different sized packets

Baudrate: 19200

Frame Size (byte)	Time (s)	T Frame (bit/s)	Efficiency
128	6,452	13599	70,83%
256	6,114	14351	74,74%
512	5,953	14739	76,77%
700	5,903	14864	77,42%

The test demonstrates that increasing packet size improves efficiency, reaching a peak efficiency of around 77.42% at 700 bytes. Larger packets reduce the number of headers needed, which minimizes overhead and

maximizes data transmission within each frame.

Transmission with different baudrates

Packet Size: 700

Baudrate (bit/s)	Time (s)	T Frame (bit/s)	Efficiency
2400	47,225	1857	77,38%
4800	23,613	3715	77,40%
9600	11,806	7432	77,42%
19200	5,903	14864	77,42%
38400	2,952	29723	77,40%
57600	1,968	44585	77,40%
115200	0,984	89170	77,40%

The efficiency remains consistently around 77.4% across all baudrates, indicating that the baudrate does not significantly impact efficiency for a fixed packet size. However, higher baudrates reduce transmission time, increasing data throughput.

Artificially manipulated byte error rates

Baudrate: 19200 Packet Size: 700

Byte Error Rate	Time (s)	T Frame (bit/s)	Efficiency
0,00001	6,091	14405	75,03%
0,00005	7,713	11376	59,25%
0,0001	11,674	7516	39,15%

As the byte error rate increases, efficiency drops sharply from 75.03% to 39.15%. This reflects the protocol's need to retransmit corrupted frames, which leads to increased transmission time and a significant reduction in overall efficiency with higher error rates.

Simulated propagation delay

Baudrate: 19200 Packet Size: 700

Propagation Delay (usec)	Time (s)	T Frame (bit/s)	Efficiency
0	5,903	14864	77,42%
1041	5,941	14769	76,92%
9895	6,26	14016	73,00%
99999	9,503	9233	48,09%

Propagation Delay (usec)	Time (s)	T Frame (bit/s)	Efficiency
999999	41,903	2093	10,90%

Increasing propagation delay has a marked effect on efficiency, which decreases from 77.42% to 10.9% as delay increases. This demonstrates that longer propagation delays lead to greater transmission times and reduced efficiency, as the protocol's Stop & Wait mechanism requires additional time for acknowledgments.

Data Link Protocol Efficiency

Stop & Wait Protocol Overview

The Stop & Wait protocol allows the sender to transmit one frame at a time, waiting for an acknowledgment from the receiver before sending the next frame. If an error is detected, the receiver sends a negative acknowledgment, prompting retransmission. Frames and acknowledgments are numbered to distinguish between new transmissions and repeats, and a timeout mechanism ensures that the sender retransmits if no response is received within a set period.

Key Factors Affecting Efficiency

1. **Frame Error Rate**
    - Errors detected in BCC1, which handle integrity checks, reduce efficiency dramatically as they trigger timeouts that halt transmission until acknowledgment. In contrast, errors in BCC2, which merely prompt retransmissions, cause efficiency to decline more gradually.
2. **Propagation Delay**
    - Extended propagation delays lead to longer response times and thus reduce overall efficiency, as each frame exchange consumes more time, amplifying the impact of each additional delay.
3. **Frame Size**
    - Larger frame sizes enhance efficiency by reducing the number of required frames, which lowers the frequency of header processing and minimizes protocol overhead. Efficiency gains level off when frame size exceeds the total data size, as further increases provide diminishing returns.
4. **Baud Rate**
    - While higher baud rates increase transmission speed, they do not proportionally increase efficiency due to the Stop & Wait mechanism, which inherently limits performance gains. Thus, efficiency remains largely unaffected by baud rate changes.

Results Summary

- BCC1 Errors lead to the most significant efficiency loss due to timeouts, contrasting with BCC2 Errors, which result in a more linear efficiency decline.
- Propagation Delays greatly impact efficiency by extending transmission times for each frame exchange.
- Larger Frame Sizes effectively improve efficiency by reducing overhead, with gains tapering off when frame size exceeds the data size.
- Higher Baud Rates increase transmission speed but have a minimal effect on efficiency in Stop & Wait, as protocol constraints limit performance scalability.

Conclusions

This project successfully implemented a data link protocol for communication over a serial port, achieving the primary goal of ensuring reliable data transfer between two systems. A key focus was maintaining strict layer independence, where each layer fulfills its specific responsibilities without influencing the other. The Data Link layer managed communication protocols, error handling, framing, and byte stuffing, while the Application layer accessed these services without knowledge of their underlying processes.

Through this approach, the project reinforced core concepts such as error detection, Stop & Wait, and byte stuffing. The separation of concerns between the Application and Data Link layers not only streamlined the protocol's design but also demonstrated the benefits of modular architecture, where one layer’s implementation details do not impact the other.

The successful completion of this project provided valuable practical and theoretical experience in protocol design, deepening our understanding of data link protocols and layered communication systems. Despite challenges posed by remote collaboration, the project met all outlined objectives, contributing to great communication protocols, layer independence, and serial port interactions.