U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

RCOM - 1st Lab Work

Data Link Protocol

3LEIC01
(November 10th, 2024)

Filipe Gaio, 202204985
Henrique Fernandes, 202204988

Summary

This report presents the development and implementation of a data transmission protocol based on the Stop & Wait method, designed to ensure reliable data transmission in a controlled laboratory environment. Conducted as part of the RCOM course, this project provides hands-on experience with core data-link layer mechanisms, including packet framing, error-checking, and acknowledgment handling. The protocol's design focuses on achieving reliable communication through the structured use of control packets, byte-stuffing for data integrity, and Stop & Wait to manage data flow.

The results demonstrate that the protocol effectively handles packet framing and error detection, maintaining data integrity throughout the transmission process. The implementation validates the reliability of the protocol and shows the fundamental strengths and limitations of this approach in real-world applications.

1 - Introduction

This report outlines the design, structure, and functionality of a data link layer protocol based on the Stop & Wait approach. The protocol is designed to ensure reliable data transfer over potentially unreliable communication channels by employing control mechanisms, alarm handling, and byte-stuffing techniques for error-checking and data integrity.

The implementation is divided into two primary modules: `application_layer.c` and `link_layer.c`. The `application_layer.c` module focuses on high-level packet management, handling file metadata and data transmission at the application level. In contrast, `link_layer.c` manages the core link-layer functionalities, including connection establishment, byte-stuffing for data integrity, frame transmission, and acknowledgment handling.

2 - Architecture

The protocol consists of two main layers:

2.1. Application Layer

This layer is responsible for fragmenting files into data packets and generating control packets to manage the start and end of transmission. Data packets include a control field, sequence number, and payload, enabling structured and reliable file transmission.

2.2. Link Layer

Encapsulates application layer packets with headers and flags, adding Byte Count Check (BCC) for error detection. It handles the stuffing and destuffing of flags within frames to ensure accurate transmission.

3 - Code Structure

3.1 Application Layer (application_layer.c)

3.1.1 Initialization

The applicationLayer function initializes the protocol's settings and handles connection management for file transfer. During initialization, it sets up the LinkLayer configuration by populating the structure with parameters like serialPort, baudRate, nTries (for retransmissions), and timeout. It also assigns the appropriate role (LlTx for transmitter or LlRx for receiver) based on the provided role parameter.

Once the LinkLayer is configured, the function establishes a serial connection using llopen, which initiates the communication channel. If the connection fails, it attempts to close the link successfully. The initialization process ensures that all protocol parameters are correctly set before beginning data transmission or reception, providing a stable foundation for subsequent data handling.

3.1.2 Transmitter

- **`sendControlPacket`**:
  Constructs and sends control packets containing the file name and size. Typically, it uses control values to mark the start and end of the transmission session, which are crucial for both initiating and concluding the data exchange.
- **`sendDataPacket`**:
  This function constructs data packets based on the given data and sequence number. It encapsulates the data in a packet format and sends it using the link layer's `llwrite` function. This ensures each packet follows a standard layout, facilitating orderly and reliable data transfer.

3.1.3 Receiver

- **`receiveStartControlPacket`**:
  This function parses the start control packet to extract metadata, specifically the file size and file name. It expects a predefined packet format and saves the extracted metadata in a structure provided for later use.
- **`receiveEndControlPacket`**:
  Similar to the start control packet function, this function parses the end control packet to extract metadata. However, it also validates that the extracted metadata, such as file size and file name, matches the information in the start control packet.
- **`receiveDataControlPacket`**:
  This function validates the received data packet by checking for a correct header, verifying the sequence number, and extracting the packet size. If the packet passes all checks, it is considered valid for further processing.

3.2 Link Layer (link_layer.c)

- **`alarmHandler`** and **`disableAlarm`**:
  The `alarmHandler` is a signal handler that manages alarm signals, increments the alarm count, and enables a timeout flag when required. The `disableAlarm` function stops the alarm and resets the alarm counter, which is essential for managing retransmissions if a frame is lost or delayed.
- **`stuffPacket`** and **`destuffPacket`**:
  These functions implement byte-stuffing and destuffing mechanisms, critical for avoiding misinterpretation of control characters within the data. `stuffPacket` replaces flags (`0x7E)` and escape (`0x7D`) with escape

sequences, and `destuffPacket` reverses this process to recover the original data.

- **`sendControlFrame`** and **`receiveControlFrame`**:
  `sendControlFrame` constructs and sends control frames, including address and control fields and a Block Check Character (BCC) for error detection. The `receiveControlFrame` function validates incoming frames using a state machine to ensure the expected structure and values.
- **`sendControlAndAwaitAck`**:
  This function sends control frames and waits for acknowledgment using a state machine to monitor incoming responses. If no acknowledgment is received within a timeout period, the frame is retransmitted.
- **`llopen`**, **`llwrite`**, **`llread`**, **`llclose`**:
  These core functions manage the connection lifecycle. `llopen` establishes the connection, `llwrite` and `llread` manage data transmission and reception, adding headers and verifying responses. `llclose` terminates the connection, providing a clean disconnection process.

4 - Main Use Cases

The main use case is when there is a transmitter sending a file to a receiver. Depending on the role, they have different function calls.

Firstly, they open the connection by having the transmitter send a signal to the receiver, which will then respond with another signal. If the connection opens successfully, the transmitter opens the file and sends a control packet to the receiver with the file metadata. While that happens, the receiver is waiting for a packet, so it will receive the metadata. After that, the transmitter will split the file into small packages and add some headers that will be used by the receiver to check for errors. After the receiver receives each packet, it will validate it and send a signal to the transmitter with the result, either approving or rejecting the packet. If this response takes too long, the transmitter will try again up to a certain limit. After that limit is reached, the connection will be forced to close. If the packet is accepted, the transmitter will then proceed to the next one, until the whole file is sent. After that, it will send another packet with the file metadata to validate the original data. Then, they will attempt to close the connection by doing a 3-way handshake: the transmitter will send a signal to the receiver, which then responds with an acknowledgment. After receiving the acknowledgment, the transmitter sends a final disconnect signal to complete the process. This ensures that both sides agree to close the connection, avoiding potential data loss or errors. Once the handshake is complete, the connection is closed, and the communication session ends.

5 - Protocol Operation

5.1 Logical Link Protocol

The **logical link protocol** is responsible for establishing a reliable connection between the transmitter and receiver, implementing error-checking mechanisms, and managing retransmissions to ensure successful data transfer.

5.1.1 Functional Aspects

- **Frame Validation**: Ensures each frame meets expected structure requirements, including address, control fields, and BCC for integrity.
- **Byte-stuffing and Destuffing**: Prevents data misinterpretation by escaping specific control bytes within the data, essential for accurate packet reassembly.
- **Stop-and-Wait Protocol**: Ensures that each frame is acknowledged before proceeding to the next. If an acknowledgment is not received, the frame is retransmitted using an alarm-based timeout.

5.1.2 Implementation Strategy

The `sendControlFrame` and `receiveControlFrame` functions utilize a state machine to process each incoming byte, enabling control over acknowledgment handling and retransmissions. Here's a code excerpt for `sendControlFrame`, which illustrates frame construction and BCC calculation:

C/C++
```
int sendControlFrame(unsigned char A, unsigned char C) {
  unsigned char frame[5] = {0x7E, A, C, 0, 0x7E};
  frame[3] = frame[1] ^ frame[2];

  if (write(fd, frame, 5) < 0) {
       perror("Error sending control frame!\n");
       return -1;
  }
  return 0;
}
```

Here's a code excerpt for `receiveControlFrame`, which illustrates the receive process through the state machine:

C/C++
```c
      int  receiveControlFrame(unsigned  char  expectedA,  unsigned  char
expectedC) {
        enum states currentState = START;

        while (currentState != STOP) {
             unsigned char byte = 0;
             int bytes;
             if ((bytes = read(fd, &byte, sizeof(byte))) < 0) {
                    perror("Error reading byte from control frame!\n");
                    return -1;
             }
             if (bytes > 0) {
                    switch (currentState) {
                    case START:
                           if (byte == 0x7E)
                                  currentState = FLAG_RCV;
                           break;
                    case FLAG_RCV:
                           if (byte == 0x7E)
                                  continue;
                           if (byte == expectedA)
                                  currentState = A_RCV;
                           break;
                    case A_RCV:
                           if (byte == 0x7E)
                                  currentState = FLAG_RCV;
                           else if (byte == expectedC)
                                  currentState = C_RCV;
                           else
                                  currentState = START;
                           break;
                    case C_RCV:
                           if (byte == 0x7E)
                                  currentState = FLAG_RCV;
                           else if (byte == (expectedC ^ expectedA))
                                  currentState = BCC_OK;
                           else
                                  currentState = START;
                           break;
                    case BCC_OK:
                           if (byte == 0x7E)
                                  currentState = STOP;
                           else
                                  currentState = START;
                           break;
                    default:
                           currentState = START;
                    }
             }
        }
        return 0;
      }
```

The stop-and-wait protocol relies on the `alarmHandler` function to manage retransmissions when an acknowledgment is not received within the timeout. The function increments the alarm count and triggers a retransmission if necessary, ensuring data reliability over possible errors and malfunctions.

5.2 Application Protocol

The **application protocol layer** provides higher-level functionality, managing the structure and segmentation of data to be sent, handling metadata for session initiation, and ensuring data sequence integrity.

5.2.1 Functional Aspects

- **Session Management**: Utilizes control packets to mark the start and end of a session.
- **Packet Segmentation**: Splits the file into manageable data packets for transmission.
- **Sequence Number Management**: Ensures data integrity by maintaining an ordered sequence of packets.

5.2.2 Implementation Strategy

On the transmitter side, the `sendDataPacket` function creates a data packet and adds the necessary information, like a sequence number for ordering. Control packets created in `sendControlPacket` contain metadata to facilitate session management.

This method ensures that each control packet is formatted correctly, containing the necessary metadata for the receiver to understand the transmitted file's size and name.

On the receiver side, the packets are then reassembled into a file, which is equal to the file that was sent.

6 - Statistics and Validation

To validate the protocol's functionality, tests were performed by transmitting the file **penguin.gif**, which has a size of 10968 bytes. Analysis of the results led to the following conclusions:

- At the default baud rate of 9600 and a Bit Error Ratio (BER) of 0, sending the file took, on average, 11.77 seconds, which means a speed of around 7456 bits/s. By comparing the speed with the baud rate, the efficiency is 77.67%.
- Keeping the baud rate at 9600 and increasing the BER to 10^{-4}, the process took 23 seconds, which resulted in an efficiency of 41%. On average, 38.3%

of the frames were rejected (each frame contains 1000 bytes of file data). This is an extreme case since common industry BER standards range from 10^{-6} to 10^{-12}. Further increasing the BER will cause the file to never get sent, because there will be at least one error on every frame.

- Increasing the propagation delay does not cause issues in the protocol, it only reduces the speed, thus its efficiency. At a BER of 0 and baud rate of 9600, the process took, on average, 49.7 seconds, which is a 422% increase in time (38 seconds, which makes sense, since 32 frames are exchanged, it results in around 1.1 seconds increase per frame) when compared with a propagation delay of 0.
- Increasing the baud rate to the maximum (115200), while keeping the propagation delay and BER at 0 results in an efficiency almost equal to the default baud rate. The whole process took 0.98 seconds, which translates into a speed of 89604 bit/s and an efficiency of 77.78%. This proves that the protocol achieves similar efficiency regardless of the baud rate.
- Keeping the baud rate at 115200 and increasing the BER to 10^{-4}, we get similar results to the baud rate at 9600. Around 40% of the frames were rejected, and the efficiency decreased to 39%. Despite this, a high baud rate can cause the system to struggle to maintain synchronization between the sender and receiver, which can result in a crash, but due to the error verification process, the protocol detects the issue and aborts the process.
- With a baud rate of 115200, which is considered very high, increasing the propagation delay has a huge impact on the speed: previously, what took 0.98 seconds, now takes 28.9 seconds with a propagation delay of 1 second, which is an increase in time of 2956%! Furthermore, increasing the BER to 10^{-4} drops the efficiency to as low as 1.1%.

From these results, it is possible to conclude that both a high baud rate, a low propagation delay, and a low BER are required to ensure a high file transfer speed.

To achieve optimal efficiency, the only parameter that can be changed is the frame size, since the other parameters do not depend on the protocol. Depending on the conditions, different frame sizes can make a big difference in the efficiency. If the propagation delay is high and the BER is low, then a big frame will be recommended, while a low propagation delay with a high BER will benefit from smaller frames.

7. Conclusion

This implementation successfully demonstrated a reliable data link layer protocol that performs robustly under varying conditions. The project achieved a reliable transfer with minimal data loss, making it suitable for applications requiring consistent data integrity. Future improvements could focus on optimizing the frame size to different conditions and incorporating dynamic timeout adjustments to further enhance transmission efficiency.

Appendix I - Source Code

1. link_layer.c

```
C/C++
       // Link layer protocol implementation

       #include "../include/link_layer.h"
       #include "../include/serial_port.h"
       #include <bits/time.h>
       #include <bits/types/struct_timeval.h>
       #include <fcntl.h>
       #include <signal.h>
       #include <stdio.h>
       #include <stdlib.h>
       #include <string.h>
       #include <sys/stat.h>
       #include <sys/types.h>
       #include <termios.h>
       #include <time.h>
       #include <unistd.h>

       // MISC
       #define _POSIX_SOURCE 1 // POSIX compliant source
       #define FLAG_BUFFER_SIZE 5

       #define RR0 0xAA
       #define RR1 0xAB
       #define REJ0 0x54
       #define REJ1 0x55

       enum states {
         START,
         FLAG_RCV,
         A_RCV,
         C_RCV,
         BCC_OK,
         STOP,
         DATA,
       };

       typedef struct {
         int nBytes;                      // Totla bytes sent / received.
         int rejectedBytes;               // Bytes that were rejected.
         int nFrames;                     // Total number of frames sent /
received.
         int rejectedFrames;              // Number of rejected frames.
         size_t filesize;                 // Size of the file, hardcoded.
         struct timespec globalStart;     // Registered when `llopen()` is
called.
          struct timespec connectionStart; // Registered when `llopen()
finishes.`
} Statistics;

Statistics statistics = {0, 0, 0, 0, 10968, 0, 0};
int alarmEnabled;
int alarmCount = 0;
enum states current_state = START;
LinkLayer parameters;
int fd;
unsigned char informationFrameNumber =
      0; // Used to generate the information frame.

/**
 * @brief Signal handler for alarm signals.
 *
 * This function is called when an alarm signal is received. It
increments the
 * alarm count, enables the alarm flag, and prints the current
alarm count.
 *
 * @param signal The signal number.
 */
void alarmHandler(int signal) {
  alarmCount++;
  alarmEnabled = TRUE;
  printf("Alarm! Count: %d\n", alarmCount);
}

/**
 * @brief Disables the alarm.
 *
 * This function disables the alarm by setting the global
variable
 * `alarmEnabled` to FALSE, stopping any active alarm by calling
`alarm(0)`, and
 * resetting the `alarmCount` to 0.
 */
void disableAlarm() {
  alarmEnabled = FALSE;
  alarm(0);
  alarmCount = 0;
}

/**
 * @brief Stuffs a packet by replacing FLAG and ESC bytes with
escape sequences.
 *
 * This function processes the input packet and replaces any
occurrence of the
 * FLAG byte (0x7E) with the sequence 0x7D5E and any occurrence
of the ESC byte
 * (0x7D) with the sequence 0x7D5D. The resulting packet is
stored in the
 * provided newPacket buffer.
 *
 * @param packet The input packet to be stuffed.
```

```
        * @param packetSize The size of the input packet.
        * @param newPacket The buffer to store the stuffed packet. It
should have at
        * least double the size of the input packet to accommodate the
worst-case
        * scenario.
        * @param newPacketSize A pointer to a variable where the size of
the stuffed
        * packet will be stored.
        * @return 0 on success, 1 if any of the input pointers are NULL.
        */
       int stuffPacket(const unsigned char *packet, size_t packetSize,
                    unsigned char *newPacket, size_t *newPacketSize) {
          // Assumes newPacket has at least double the size of the
packet.

         if (packet == NULL || newPacket == NULL || newPacketSize ==
NULL) {
             return 1;
         }

         size_t newPacketIndex = 0;

         for (size_t packetIndex = 0; packetIndex < packetSize;
             packetIndex++, newPacketIndex++) {
             // Replace FLAG (0x7E) with 0x7D5E.
             if (packet[packetIndex] == 0x7E) {
             newPacket[newPacketIndex++] = 0x7D;
             newPacket[newPacketIndex] = 0x5E;
             }
             // Replace ESC (0x7D) with 0x7D5E.
             else if (packet[packetIndex] == 0x7D) {
             newPacket[newPacketIndex++] = 0x7D;
             newPacket[newPacketIndex] = 0x5D;
             } else {
             newPacket[newPacketIndex] = packet[packetIndex];
             }
         }
         *newPacketSize = newPacketIndex;

         return 0;
       }

       /**
        * @brief Destuffs a packet by removing escape sequences.
        *
        * This function processes an input packet and removes escape
sequences,
        * producing a new packet with the escape sequences removed.
        *
        * @param packet The input packet to be destuffed.
        * @param packetSize The size of the input packet.
        * @param newPacket The output buffer where the destuffed packet
will be stored.
```

```
        * @param newPacketSize A pointer to a variable where the size of
the destuffed
        * packet will be stored.
        * @return 0 on success, 1 if any of the input pointers are NULL.
        */
       int destuffPacket(const unsigned char *packet, size_t packetSize,
                         unsigned     char     *newPacket,     size_t
*newPacketSize) {
          if  (packet  ==  NULL  ||  newPacket  ==  NULL  ||  newPacketSize  ==
NULL) {
              return 1;
         }

         size_t newPacketIndex = 0;

         for (size_t packetIndex = 0; packetIndex < packetSize;
              packetIndex++, newPacketIndex++) {
              if (packet[packetIndex] != 0x7D) {
              newPacket[newPacketIndex] = packet[packetIndex];
              } else {
              if (packet[packetIndex + 1] == 0x5E) {
              newPacket[newPacketIndex] = 0x7E;
              } else if (packet[packetIndex + 1] == 0x5D) {
              newPacket[newPacketIndex] = 0x7D;
              }
              packetIndex++;
              }
         }

         *newPacketSize = newPacketIndex;

         return 0;
       }

       /**
        * @brief Sends a control frame.
        *
        *  This  function  constructs  a  control  frame  with  the  given
address (A) and
        *  control  (C)  fields,  calculates  the  Block  Check  Character
(BCC), and sends the
        * frame through a file descriptor.
        *
        * @param A The address field of the control frame.
        * @param C The control field of the control frame.
        * @return int Returns 0 on success, or -1 on failure.
        */
       int sendControlFrame(unsigned char A, unsigned char C) {
         unsigned char frame[5] = {0x7E, A, C, 0, 0x7E};
         frame[3] = frame[1] ^ frame[2];

         if (write(fd, frame, 5) < 0) {
              perror("Error sending control frame!\n");
              return -1;
         }
         return 0;
       }

       /**
        *  @brief  Receives  a  control  frame  and  validates  it  against
   expected values.
        *
        * This function reads bytes from a file descriptor and processes
   them to
        * validate a control frame. The frame is validated based on the
   expected
        * address (A) and control (C) values provided as arguments.
        *
        *  @param  expectedA  The  expected  address  byte  of  the  control
   frame.
        *  @param  expectedC  The  expected  control  byte  of  the  control
   frame.
        * @return int Returns 0 on successful reception and validation
   of the control
        * frame, or -1 if an error occurs during reading.
        */
       int  receiveControlFrame(unsigned  char  expectedA,  unsigned  char
   expectedC) {
         enum states currentState = START;

         while (currentState != STOP) {
             unsigned char byte = 0;
             int bytes;
             if ((bytes = read(fd, &byte, sizeof(byte))) < 0) {
             perror("Error reading byte from control frame!\n");
             return -1;
             }
             if (bytes > 0) {
             switch (currentState) {
             case START:
             if (byte == 0x7E)
             currentState = FLAG_RCV;
             break;
             case FLAG_RCV:
             if (byte == 0x7E)
             continue;
             if (byte == expectedA)
             currentState = A_RCV;
             break;
             case A_RCV:
             if (byte == 0x7E)
             currentState = FLAG_RCV;
             else if (byte == expectedC)
             currentState = C_RCV;
             else
             currentState = START;
             break;
             case C_RCV:
             if (byte == 0x7E)
             currentState = FLAG_RCV;
```

```
                else if (byte == (expectedC ^ expectedA))
                currentState = BCC_OK;
                else
                currentState = START;
                break;

                case BCC_OK:
                if (byte == 0x7E)
                currentState = STOP;
                else
                currentState = START;
                break;
                default:
                currentState = START;
                }
                }
            }
            return 0;
        }

        /**
         * @brief Sends a control frame and waits for an acknowledgment.
         *
         * This function sends a control frame with the specified address
(A) and
         * control (C) fields, and waits for an acknowledgment frame with
the expected
         * address (expectedA) and control (expectedC) fields. It uses a
state machine
         * to process the received bytes and verify the acknowledgment.
If the
         * acknowledgment is not received within the specified timeout,
the control
         * frame is retransmitted.
         *
         * @param A The address field of the control frame to be sent.
         * @param C The control field of the control frame to be sent.
          * @param expectedA The expected address field of the
acknowledgment frame.
          * @param expectedC The expected control field of the
acknowledgment frame.
          * @return int Returns 0 if the acknowledgment is received
successfully, -1
         * otherwise.
         */
        int sendControlAndAwaitAck(unsigned char A, unsigned char C,
                                   unsigned char expectedA, unsigned char
expectedC) {
            enum states currentState = START;
            (void)signal(SIGALRM, alarmHandler);
            if (sendControlFrame(A, C))
                return -1;
            alarm(parameters.timeout);
```

```
              while    (currentState    !=    STOP    &&    alarmCount    <=
parameters.nRetransmissions) {
              unsigned char byte = 0;
              int bytes;

              if ((bytes = read(fd, &byte, sizeof(byte))) < 0) {
              perror("Error reading byte from control frame!\n");
              return -1;
              }
              if (bytes > 0) {
              switch (currentState) {
              case START:
              if (byte == 0x7E)
              currentState = FLAG_RCV;
              break;
              case FLAG_RCV:
              if (byte == 0x7E)
              continue;
              if (byte == expectedA)
              currentState = A_RCV;
              break;
              case A_RCV:
              if (byte == 0x7E)
              currentState = FLAG_RCV;
              else if (byte == expectedC)
              currentState = C_RCV;
              else
              currentState = START;
              break;
              case C_RCV:
              if (byte == 0x7E)
              currentState = FLAG_RCV;
              else if (byte == (expectedC ^ expectedA))
              currentState = BCC_OK;
              else
              currentState = START;
              break;

              case BCC_OK:
              if (byte == 0x7E)
              currentState = STOP;
              else
              currentState = START;
              break;
              default:
              currentState = START;
              }
              }
              if (currentState == STOP) {
              disableAlarm();
              return 0;
              }

              if (alarmEnabled) {
              alarmEnabled = FALSE;
              if (alarmCount <= parameters.nRetransmissions) {
              printf("Retransmitting...\n");
              if (sendControlFrame(A, C))
              return -1;
              alarm(parameters.timeout);
              }
              currentState = START;
              }
          }
          disableAlarm();
          return -1;
        }

        void   sendFilesize(size_t   filesize)   {   statistics.filesize   =
filesize; }

        void printStatistics() {
          struct timespec end;
          clock_gettime(CLOCK_MONOTONIC, &end);
                   float      globalDuration      =      (end.tv_sec      -
statistics.globalStart.tv_sec) +
                                   (end.tv_nsec                           -
statistics.globalStart.tv_nsec) / 1e9;
                     float       duration       =       (end.tv_sec       -
statistics.connectionStart.tv_sec) +
                            (end.tv_nsec                                  -
statistics.connectionStart.tv_nsec) / 1e9;
          printf("\nSTATISTICS\n");
          printf("\tRole: %s\n", parameters.role == LlTx ? "Transmitter"
: "Receiver");
          if (parameters.role == LlTx) {
              printf("\tGlobal duration: %fs\n", globalDuration);
              printf("\tTransmission duration: %fs\n", duration);
              printf("\tFrames sent: %d\n", statistics.nFrames);
              printf("\tAccepted frames: %d\n",
                     statistics.nFrames - statistics.rejectedFrames);
              printf("\tRejected              frames:              %d\n",
statistics.rejectedFrames);
              printf("\tBytes sent: %d\n", statistics.nBytes);
              printf("\tAccepted bytes: %d\n",
                     statistics.nBytes - statistics.rejectedBytes);
              printf("\tRejected              bytes:               %d\n",
statistics.rejectedBytes);
              printf("\tAverage frame size: %f bytes\n",
                     (float)statistics.nBytes / statistics.nFrames);
              printf("\tSpeed: %f bit/s (filesize divided by time)\n",
                     statistics.filesize * 8.0 / duration);
              printf("\tEfficiency: %f%% (speed / baudrate)\n",
                     (statistics.filesize    *    8.0    /    duration)    /
(parameters.baudRate) *
                     100);
          } else if (parameters.role == LlRx) {
              printf("\tGlobal duration: %fs\n", globalDuration);
              printf("\tTransmission duration: %fs\n", duration);
              printf("\tFrames received: %d\n",
                     statistics.nFrames + statistics.rejectedFrames);
              printf("\tAccepted frames: %d\n", statistics.nFrames);
              printf("\tRejected              frames:              %d\n",
statistics.rejectedFrames);
              printf("\tBytes received: %d\n", statistics.nBytes);
              printf("\tAccepted bytes: %d\n",
                     statistics.nBytes - statistics.rejectedBytes);
              printf("\tRejected              bytes:               %d\n",
statistics.rejectedBytes);
              printf("\tAverage frame size: %f bytes\n",
                     (float)statistics.nBytes /
                     (statistics.nFrames + statistics.rejectedFrames));
              printf("\tSpeed: %f bit/s (filesize divided by time)\n",
                     statistics.filesize * 8.0 / duration);
              printf("\tEfficiency: %f%% (speed / baudrate)\n",
                     (statistics.filesize   *   8.0   /   duration)   /
(parameters.baudRate) *
                     100);
          }
        }

        ////////////////////////////////////////////////
        // LLOPEN
        ////////////////////////////////////////////////
        int llopen(LinkLayer connectionParameters) {
          parameters = connectionParameters;
          (void)signal(SIGALRM, alarmHandler);
          clock_gettime(CLOCK_MONOTONIC, &statistics.globalStart);

          fd = openSerialPort(connectionParameters.serialPort,
                            connectionParameters.baudRate);
          if (fd < 0) {
              return -1;
          }

          if (connectionParameters.role == LlTx) { // Transmitter
              //  Sends  control  frame  with  A=0x03  and  C=0x03  and  waits
for a response with
              // A=0x03 and C=0x07.
              if (sendControlAndAwaitAck(0x03, 0x03, 0x03, 0x07))
              return -1;
              statistics.nFrames++;
              statistics.nBytes += 5;
              printf("Control   frame   sent   and   received   correctly,
connection established "
                     "successfully.\n");
          } else {
              // Waits for a control frame with A=0x03 and C=0x03.
              if (receiveControlFrame(0x03, 0x03))
              return -1;
              // Responds with a control frame with A=0x03 and C=0x07.
              if (sendControlFrame(0x03, 0x07))
              return -1;
              statistics.nFrames++;
              statistics.nBytes += 5;
```

```
            printf("Control   frame   received   and   sent   correctly,
connection established "
                   "successfully.\n");
        }

        clock_gettime(CLOCK_MONOTONIC, &statistics.connectionStart);
        return 0;
      }

      ////////////////////////////////////////////////
      // LLWRITE
      ////////////////////////////////////////////////
      int llwrite(const unsigned char *buf, int bufSize) {
        if (buf == NULL) {
            return -1;
        }

        // Add header and footer to packet (A, C, BCC1, ..., BCC2). The
flags will be
        // added later.
        unsigned char newPacket[bufSize + 1];
        memcpy(&newPacket, buf, bufSize);

        unsigned char bcc2 = 0;
        for (int i = 0; i < bufSize; i++) {
            bcc2 ^= buf[i];
        }
        newPacket[bufSize] = bcc2;

        unsigned char stuffedPacket[(bufSize + 1) * 2 + 5];
        size_t stuffedPacketSize;

        if (stuffPacket(newPacket, bufSize + 1, stuffedPacket + 4,
                        &stuffedPacketSize)) {
            perror("Error stuffing packet!\n");
            return -1;
        }
        stuffedPacketSize += 5;

        // Insert the flags.
        stuffedPacket[0] = 0x7E;
        stuffedPacket[1] = 0x03;
        stuffedPacket[2] = informationFrameNumber;
        stuffedPacket[3] = 0x03 ^ informationFrameNumber;
        stuffedPacket[stuffedPacketSize - 1] = 0x7E;
        informationFrameNumber ^=
            0x80;  //  Toggle  informationFrameNumber  between  0x00  and
0x80.

        // Send the packet.
        if (write(fd, stuffedPacket, stuffedPacketSize) < 0) {
            perror("Error writing stuffed packet.\n");
            return -1;
        }
        statistics.nFrames++;
        statistics.nBytes += stuffedPacketSize;
        printf("Packet sent!\n");

        // Verify response
        enum states currentState = START;
        (void)signal(SIGALRM, alarmHandler);
        alarm(parameters.timeout);

        unsigned char receivedA = 0;
        unsigned char receivedC = 0;

             while    (currentState    !=    STOP    &&    alarmCount    <=
parameters.nRetransmissions) {
             unsigned char byte = 0;
             int bytes;

             if ((bytes = read(fd, &byte, sizeof(byte))) < 0) {
             perror("Error reading byte from control frame!\n");
             return -1;
             }
             if (bytes > 0) {
             switch (currentState) {
             case START:
             receivedA = 0;
             receivedC = 0;
             if (byte == 0x7E)
             currentState = FLAG_RCV;
             break;
             case FLAG_RCV:
             if (byte == 0x7E)
             continue;
             if (byte == 0x03) {
             receivedA = byte;
             currentState = A_RCV;
             } else {
             currentState = START;
             }
             break;
             case A_RCV:
             if (byte == RR0 || byte == RR1 || byte == REJ0 || byte ==
REJ1) {
             receivedC = byte;
             currentState = C_RCV;
             } else if (byte == 0x7E)
             currentState = FLAG_RCV;
             else
             currentState = START;
             break;
             case C_RCV:
             if (byte == (receivedA ^ receivedC)) {
             currentState = BCC_OK;
             } else if (byte == 0x7E)
             currentState = FLAG_RCV;
             else
             currentState = START;
```

```
            break;
            case BCC_OK:
            if (byte == 0x7E) {
            currentState = STOP;
            } else
            currentState = START;
            break;
            default:
            currentState = START;
            }
            }
            if (currentState == STOP) {
            printf("Received response.\n");
            // if the information number is 0, it means the packet was
sent with
            // information number 0x80, thus it is expecting a RR0.
            if ((informationFrameNumber == 0 && receivedC == RR0) ||
            (informationFrameNumber == 0x80 && receivedC == RR1)) {
            printf("Packet  accepted  by  receiver,  proceding  to  the
next.\n");
            disableAlarm();
            return stuffedPacketSize;
            }
            alarmEnabled = TRUE;
            alarmCount = 0;
            statistics.rejectedFrames++;
            statistics.rejectedBytes += stuffedPacketSize;
            printf("Packet rejected by receiver, trying again...\n");
            }
            // Verify if the alarm fired
            if (alarmEnabled) {
            alarmEnabled = FALSE;
            if (alarmCount <= parameters.nRetransmissions) {
            printf("Retransmitting packet...\n");
            if (write(fd, stuffedPacket, stuffedPacketSize) < 0) {
            perror("Error writing stuffed packet.\n");
            return -1;
            }
            statistics.nFrames++;
            statistics.nBytes += stuffedPacketSize;
            alarm(parameters.timeout);
            }
            currentState = START;
            }
        }
        disableAlarm();
        return -1;
      }

      ////////////////////////////////////////////////
      // LLREAD
      ////////////////////////////////////////////////
      int llread(unsigned char *packet) {
        enum states currentState = START;
        size_t packetIndex = 0;
        unsigned char receivedA = 0;
        unsigned char receivedC = 0;

        while (currentState != STOP) {
            unsigned char byte = 0;
            int bytes;

            if ((bytes = read(fd, &byte, sizeof(byte))) < 0) {
            perror("Error reading byte from control frame!\n");
            return -1;
            }
            if (bytes > 0) {
            statistics.nBytes++;
            switch (currentState) {
            case START:
            receivedC = 0;
            receivedA = 0;
            if (byte == 0x7E)
            currentState = FLAG_RCV;
            break;
            case FLAG_RCV:
            if (byte == 0x7E)
            continue;
            if (byte == 0x03) {
            receivedA = byte;
            currentState = A_RCV;
            } else
            currentState = START;
            break;
            case A_RCV:
            if (byte == 0x00 || byte == 0x80) {
            receivedC = byte;
            currentState = C_RCV;
            } else if (byte == 0x7E)
            currentState = FLAG_RCV;
            else
            currentState = START;
            break;
            case C_RCV:
            if (byte == (receivedC ^ receivedA)) {
            currentState = DATA;
            } else if (byte == 0x7E)
            currentState = FLAG_RCV;
            else
            currentState = START;
            break;
            case DATA:
            if (byte == 0x7E) {
            unsigned char destuffedPacket[packetIndex];
            size_t destuffedPacketSize;

            if (destuffPacket(packet, packetIndex, destuffedPacket,
                                    &destuffedPacketSize)) {
                   return -1;
```

```
              }
              unsigned            char           receivedBCC2           =
destuffedPacket[destuffedPacketSize - 1];
              unsigned char bcc2 = 0;
              for (size_t i = 0; i < destuffedPacketSize - 1; i++) {
                     bcc2 ^= destuffedPacket[i];
              }
              unsigned char responseC;
              if (bcc2 == receivedBCC2) {
                     // if the current frame is 0, ready to receive 1.
                     responseC = (receivedC == 0x00) ? RR1 : RR0;
                     printf("BCC2  matches,  approving  with  0x%02x\n",
responseC);
              } else {
                     responseC = (receivedC == 0x00) ? REJ0 : REJ1;
                     printf("BCC2   doesn't   match,   rejecting   with
0x%02x\n", responseC);
                     statistics.rejectedFrames++;
                     statistics.rejectedBytes += packetIndex + 5;
                     if (sendControlFrame(0x03, responseC))
                     return -1;
                     return 0;
              }
              if (sendControlFrame(0x03, responseC)) {
                     return -1;
              }
              memcpy(packet, destuffedPacket, destuffedPacketSize - 1);
              statistics.nFrames++;
              return destuffedPacketSize - 1; // -1 to remove BCC2
              } else {
              packet[packetIndex++] = byte;
              }
              break;
              default:
              currentState = START;
              }
              }
         }

         return -1;
       }

       ///////////////////////////////////////////////////
       // LLCLOSE
       ///////////////////////////////////////////////////
       int llclose(int showStatistics) {
         printf("Attempting to close connection...\n");
          // Transmitter sends disc, receiver sends disc and waits for
response.
         if (parameters.role == LlTx) {
              //  Sends  A=0x03  and  C=0x0B,  waits  for  response  A=0x01,
C=0x0B (disconnect
              // frames).
              if (sendControlAndAwaitAck(0x03, 0x0B, 0x01, 0x0B) < 0)
              return closeSerialPort();
```

```
        if (sendControlFrame(0x01, 0x07) < 0)
        return closeSerialPort();
        statistics.nFrames += 2;
        statistics.nBytes += 10;
        printf("Disconnected!\n");

   } else if (parameters.role == LlRx) {
        if (receiveControlFrame(0x03, 0x0B) < 0)
        return closeSerialPort();
        if (sendControlAndAwaitAck(0x01, 0x0B, 0x01, 0x07) < 0)
        return closeSerialPort();
        statistics.nFrames += 2;
        statistics.nBytes += 10;
        printf("Disconnected!\n");
   }

   if (showStatistics) {
        printStatistics();
   }
   int clstat = closeSerialPort();
   return clstat;
}
```

2. application_layer.c

```
C/C++
// Application layer protocol implementation

#include "../include/application_layer.h"
#include "../include/link_layer.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAXFILENAMESIZE 256

typedef struct {
  size_t fileSize;
  char filename[MAXFILENAMESIZE];
} FileMetadata;

FileMetadata fileMetadata = {0, ""};

/**
 * @brief Generates a hexdump of the specified file.
 *
 * This function constructs a command to generate a hexdump of
the file
```

```
        * specified by the filename parameter and executes it using the
system
        * function. The hexdump command used is "hexdump -C <filename>".
        * Useful for debugging purposes.
        *
        * @param filename The path to the file to be hexdumped.
        * @return Returns 0 on success, or 1 if an error occurs while
executing the
        * hexdump command.
        */
       int hexdump(const char *filename) {
         char command[256];
         snprintf(command, sizeof(command), "hexdump -C %s", filename);
         int result = system(command);

         if (result == -1) {
              perror("Error executing hexdump");
              return 1;
         }
         return 0;
       }

       /**
        * @brief Sends a control packet containing the file size and
filename.
        *
        * This function constructs a control packet with the specified
control value,
        * file size, and filename, and sends it using the llwrite
function.
        *
        * @param filename The name of the file to be included in the
control packet.
        * @param controlValue The control value indicating the type of
control packet.
        *                   Typically, 1 for start and 3 for end.
        * @param fileSize The size of the file to be included in the
control packet.
        * @return int Returns 0 on success, or -1 if the filename is
NULL.
        */
       int sendControlPacket(const char *filename, unsigned char
controlValue,
                            size_t fileSize) {
         if (filename == NULL) {
              return -1;
         }

         unsigned char L1 = sizeof(fileSize);
         unsigned char L2 = strlen(filename);

         unsigned char packet[5 + L1 + L2];
         packet[0] = controlValue; // 1 for start, 3 for end
         packet[1] = 0;            // 0 for file size
         packet[2] = L1;           // size of file size
         // extract individual bytes from the file size, little-endian
(least
         // significant byte is stored first)
         for (int i = 0; i < L1; i++) {
              packet[3 + i] = (fileSize >> (8 * i)) & 0xFF;
         }

         packet[3 + L1] = 1;                    // 1 for file name
         packet[4 + L1] = L2;                   // size of file name
         memcpy(&packet[5 + L1], filename, L2); // filename
         return llwrite(packet, 5 + L1 + L2);
       }

       /**
        * @brief Sends a data packet.
        *
        * This function constructs a data packet with the given data and
sequence
        * number, and sends it using the `llwrite` function. The packet
format is as
        * follows:
        * - Byte 0: Control field (2 for data)
        * - Byte 1: Sequence number
        * - Byte 2: Data size (most significant byte)
        * - Byte 3: Data size (least significant byte)
        * - Bytes 4 to (dataSize + 3): Data
        *
        * @param data Pointer to the data to be sent.
        * @param dataSize Size of the data to be sent.
        * @param sequenceNumber Sequence number of the packet.
        * @return 0 on success, -1 if the data pointer is NULL, or the
return value of
        * `llwrite`.
        */
       int sendDataPacket(unsigned char *data, size_t dataSize, int
sequenceNumber) {
         if (data == NULL) {
              return -1;
         }

         unsigned char packet[dataSize + 4];

         packet[0] = 2; // Control field 2 (data)
         packet[1] = sequenceNumber;
         packet[2] = (dataSize >> 8) & 0xFF;
         packet[3] = dataSize & 0xFF;
         memcpy(packet + 4, data, dataSize);

       #ifdef DEBUG
         printf("Data packet with size %zu:\n", dataSize + 4);
         for (size_t i = 0; i < dataSize + 4; i++) {
              printf("%02x ", packet[i]);
         }
         printf("\n");
```

```
#endif

  return llwrite(packet, dataSize + 4);
}

/**
 * @brief Receives and parses a start control packet to extract
file metadata.
 *
 * This function processes a start control packet to extract the
file size and
 * file name, storing them in the provided metadata structure.
The packet is
 * expected to follow a specific format:
 * - The first byte should be 1, indicating a start control
packet.
 * - The second byte should be 0, indicating the file size
parameter.
 * - The third byte indicates the size (in octets) of the file
size value.
 * - The next bytes contain the file size value.
 * - The byte following the file size should be 1, indicating the
file name
 * parameter.
 * - The next byte indicates the size of the file name.
 * - The remaining bytes contain the file name.
 *
 * @param packet Pointer to the start control packet.
 * @param metadata Pointer to the FileMetadata structure where
the extracted
 *                  file size and file name will be stored.
 * @return int Returns 0 on success, or 1 on error (e.g., invalid
packet format,
 *             filename too big).
 */
int receiveStartControlPacket(const unsigned char *packet,
                              FileMetadata *metadata) {
  if (packet == NULL || packet[0] != 1) {
      return 1;
  }

  // If the first parameter isn't the file size, return error.
  if (packet[1] != 0) {
      return 1;
  }

  unsigned char filesizeSize =
      packet[2]; // size in octects of the file size value.

  int fileSize = 0;
  for (int i = 0; i < filesizeSize; i++) {
      fileSize |= (packet[i + 3]) << (i * 8);
  }
  metadata->fileSize = fileSize;
```

```
            // If the second parameter isn't the file name, return error.
            if (packet[filesizeSize + 3] != 1) {
                return 1;
            }

            unsigned char filenameSize = packet[filesizeSize + 4];
            if (filenameSize + 1 > MAXFILENAMESIZE) {
                perror("Filename too big.\n");
                return 1;
            }
            for (int i = 0; i < filenameSize; i++) {
                metadata->filename[i] = packet[filesizeSize + 5 + i];
            }
            metadata->filename[filenameSize] = '\0';

            return 0;
        }

        /**
         * @brief Receives and validates the end control packet.
         *
         * This function checks the integrity of the end control packet
by verifying
         * the packet type, file size, and file name against the provided
metadata.
         *
         * @param packet Pointer to the end control packet.
          * @param metadata Pointer to the FileMetadata structure
containing the expected
         * file size and file name.
          * @return int Returns 0 if the packet is valid, otherwise
returns 1.
         */
        int receiveEndControlPacket(const unsigned char *packet,
                                    const FileMetadata *metadata) {
            if (packet == NULL || packet[0] != 3) {
                return 1;
            }

            if (packet[1] != 0) {
                return 1;
            }

            unsigned char filesizeSize =
                packet[2]; // size in octects of the file size value.

            size_t fileSize = 0;
            for (size_t i = 0; i < filesizeSize; i++) {
                fileSize |= (packet[i + 3]) << (i * 8);
            }
            if (metadata->fileSize != fileSize) {
                perror("Error: start packet filesize doesn't match end
packet filesize.\n");
                return 1;
            }
            // If the second parameter isn't the file name, return error.
            if (packet[filesizeSize + 3] != 1) {
                return 1;
            }

            unsigned char filenameSize = packet[filesizeSize + 4];
            char filename[filenameSize + 1];
            for (int i = 0; i < filenameSize; i++) {
                filename[i] = packet[filesizeSize + 5 + i];
            }
            filename[filenameSize] = '\0';
            if (strcmp(metadata->filename, filename) != 0) {
                perror("Error: start packet filename doesn't match end
packet filename.\n");
                return 1;
            }
            return 0;
        }

        /**
         * @brief Receives a data packet and validates its contents.
         *
         * This function checks if the provided packet is valid, verifies
the sequence
         * number, and extracts the packet size. If the packet is valid,
the size of the
         * packet is stored in the provided packetSize pointer.
         *
         * @param packet Pointer to the data packet.
         * @param packetSize Pointer to an integer where the size of the
packet will be
         * stored.
         * @param expectedSequenceNumber The expected sequence number of
the packet.
         * @return 0 if the packet is valid and the size is successfully
extracted, 1
         * otherwise.
         */
        int receiveDataPacket(unsigned char *packet, int *packetSize,
                              int expectedSequenceNumber) {

            if (packet == NULL || packetSize == NULL || packet[0] != 2) {
                return 1;
            }

            if (packet[1] != expectedSequenceNumber) {
                printf(
                "The data packet received contains an unexpected sequence
number.\n");
                return 1;
            }

            *packetSize = packet[2] * 256 + packet[3];
            packet += 4;

        #ifdef DEBUG
            printf("Data packet with size %d\n", *packetSize);
            for (int i = 0; i < *packetSize; i++) {
                printf("%02x ", packet[i]);
            }
            printf("\n");
        #endif

            return 0;
        }

        void applicationLayer(const char *serialPort, const char *role,
int baudRate,
                           int   nTries,   int   timeout,   const   char
*filename) {
            // Initialize link layer.
            LinkLayer linkLayer;
            strcpy(linkLayer.serialPort, serialPort);
            linkLayer.baudRate = baudRate;
            linkLayer.nRetransmissions = nTries;
            linkLayer.timeout = timeout;
            linkLayer.role = (!strcmp(role, "tx")) ? LlTx : LlRx;

            // Open serial connection
            if (llopen(linkLayer)) {
                perror("Error opening link layer.\n");
                if (llclose(FALSE)) {
                perror("Error closing link layer.\n");
                };
                return;
            };

            // Transmitter
            if (linkLayer.role == LlTx) {

                // Open the file
                FILE *fptr = fopen(filename, "r");
                if (fptr == NULL) {
                perror("File not found.\n");
                fclose(fptr);
                return;
                }

                unsigned char buffer[1000];
                size_t bytesRead;

                fseek(fptr, 0, SEEK_END);
                size_t fileSize = ftell(fptr);
                rewind(fptr);

                if (sendControlPacket(filename, 1, fileSize) < 0) {
                perror("Error sending the start control packet.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }

                int sequenceNumber = 0;
                while ((bytesRead = fread(buffer, 1, 1000, fptr)) > 0) {
                if (sendDataPacket(buffer, bytesRead, sequenceNumber++) <
    0) {
                perror("Error sending data packet");
                fclose(fptr);
                llclose(FALSE);
                return;
                }
                }

                if (sendControlPacket(filename, 3, fileSize) < 0) {
                perror("Error sending the end control packet.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }

                fclose(fptr);
                llclose(TRUE);
                return;
            }

            if (linkLayer.role == LlRx) {

                unsigned char packet[1020]; // the data packet should be
    at most 1000, but
                                            // lets add 20 to be sure.

                FILE *fptr = fopen(filename, "wb");

                if (fptr == NULL) {
                perror("Error opening file.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }

                int receiving = 1;
                int sequenceNumber = 0;

                while (receiving) {
                int bytesRead = llread(packet);
                if (bytesRead == 0) {
                continue;
                }
                if (bytesRead < 0) {
                perror("Failed to read from packet from link layer.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
```

```
                }

                if (packet[0] == 1) {
                // Start control packet
                if (receiveStartControlPacket(packet, &fileMetadata)) {
                perror("Error reading start control packet.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }

                printf("Metadata  received:\n\tfilename:  %s\n\tsize:  %zu
bytes\n",
                       fileMetadata.filename, fileMetadata.fileSize);
                } else if (packet[0] == 3) {
                // End control packet
                if (receiveEndControlPacket(packet, &fileMetadata)) {
                perror("Error reading end control packet.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }
                printf("End   control   packet   received,   file   metadata
matches.\n");
                receiving = 0;

                } else if (packet[0] == 2) {
                // Data packet
                if          (receiveDataPacket(packet,          &bytesRead,
sequenceNumber++)) {
                perror("Error reading data packet.\n");
                fclose(fptr);
                llclose(FALSE);
                return;
                }

        #ifdef DEBUG
                printf("Data packet with size %d\n", bytesRead);
                for (int i = 0; i < bytesRead + 4; i++) {
                printf("%02x ", packet[i]);
                }
                printf("\n");
        #endif
                fwrite(packet + 4, 1, bytesRead, fptr);
                }
                }
                fclose(fptr);
           }

           if (llclose(TRUE) == -1) {
                perror("Error closing connection.\n");
                return;
           }
        }
```