U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Protocolo de Ligação de Dados

Rede de Computadores 2023-2024

1º Trabalho Laboratorial

grupo 8

Guilherme Monteiro - up202108668
Sofia Sá - up202108676

Sumário

Este trabalho foi desenvolvido no âmbito da Unidade Curricular **Redes de Computadores**, da Licenciatura em Engenharia Informática e Computação, num ambiente de Linux, a linguagem C e Portas Série RS-232, e cujo objetivo era implementar um protocolo de comunicação de dados com a porta série para a transferência de ficheiros entre 2 computadores.

Através deste projeto, foi possível colocar em prática os conceitos lecionados nas aulas desta Unidade Curricular relacionadas com o protocolo ***Stop-and-Wait,*** na transferência de um ficheiro através da Porta Série. Todos os objetivos propostos para a realização deste trabalho foram concluídos com sucesso.

Introdução

O objetivo deste trabalho laboratorial baseou-se na implementação de um protocolo do nível de ligação de dados, pretendendo-se fornecer um serviço de comunicação fiável entre dois sistemas ligados por um canal de transmissão, o cabo série. Para tal, este protocolo foi separado entre duas camadas distintas e independentes: a Camada da Aplicação e a Camada de Ligação de Dados. O presente relatório destina-se à descrição do protocolo implementado e à análise da eficiência do mesmo e está dividido em oito secções:

- **Arquitetura**: Descrição dos blocos funcionais e interfaces.
- **Estrutura do código**: APIs, principais estruturas de dados, principais funções e a sua relação com a arquitetura.
- **Casos de uso principais**: Identificação e sequências de chamada de funções.
- **Protocolo de ligação lógica**: Aspetos funcionais e estratégias de implementação.
- **Protocolo de aplicação**: Aspetos funcionais e estratégias de implementação.
- **Validação**: Testes efetuados e respetivos resultados.
- **Eficiência do protocolo de ligação de dados**: Caracterização estatística da eficiência e comparação com a caracterização teórica de um protocolo *Stop&Wait.*
- **Conclusões**: Sumário da informação apresentada e reflexão acerca dos objetivos de aprendizagem alcançados.

Arquitetura

Blocos Funcionais

A arquitetura deste projeto está dividida em duas camadas: a Camada de Aplicação e a Camada da Ligação de Dados. A **Camada da Aplicação** cria os pacotes de dados com o conteúdo do ficheiro a ser transferido e utiliza as funções principais da Camada da Ligação de Dados para os enviar/receber. A **Camada da Ligação de Dados** é responsável por estabelecer e terminar a ligação entre os dois computadores, enviar e receber as tramas que contêm os pacotes mencionados acima e implementar um mecanismo de validação destas tramas e de retransmissão na de ocorrência de possíveis erros.

Interface

Para executar este programa é necessário abrir dois terminais, um para cada computador. Após compilar o programa, corremos o comando "./bin//main <localização da porta série> <papel(rx, recetor; tx, transmissor)> <nome do ficheiro a transferir/receber>". Este processo é facilitado com o uso da Makefile fornecida, que nos permite executar o programa com os comandos "make run_tx" e "make run_rx", dependendo do papel da máquina.

Estrutura de código

Camada da Aplicação

Para implementar a Camada da Aplicação, apenas foi necessária a função já presente, por defeito, no ficheiro **application_layer.c,** e criar uma instância de uma estrutura de dados auxiliar para o registo de informação relevante ao estabelecimento da conexão entre transmissor e recetor.

```
ESTRUTURAS
/* Estrutura auxiliar para registo dos parâmetros necessários à conexão estabelecida. É
enviada à Camada de Ligação de Dados quando é chamada a função llopen(). */
 LinkLayer connectionParameters = {
      .baudRate = baudRate,
      .timeout = timeout,
      .nRetransmissions = nTries,
  };
strcpy(connectionParameters.serialPort, serialPort);
  connectionParameters.role = strcmp(role, "tx") == 0 ? LlTx : LlRx;
```

```
FUNÇÕES
/* Função responsável pela construção da interface de comunicação entre o transmissor e o
recetor, assim como dos pacotes de controlo e de dados a serem enviados à Camada de Ligação
de Dados. */
void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries,
int timeout, const char *filename)
```

Camada de Ligação de Dados

Na Camada de Ligação de Dados, foram utilizadas as funções previamente definidas em **link_layer.h / link_layer.c,** uma função extra para lidar com o alarme e uma função para a construção e envio de tramas**.** Adicionalmente, foram implementadas três estruturas de dados para auxiliar a troca de informação entre transmissor e recetor.

```
ESTRUTURAS

/* Estrutura responsável por representar os vários estados de uma máquina de estados
que efetua a leitura e processamento das tramas de supervisão, não-numeradas e de informação . */
typedef enum {
  START, FLAG_RCV, A_RCV, C_RCV, BCC1_OK, READING, ESC_FOUND, STOP_State,
} State;

/* Estrutura que define o papel de transmissor ou recetor na conexão. */
typedef enum {
```

```
    LlTx, LlRx,
} LinkLayerRole;

/* Estrutura onde são caracterizados os parâmetros associados à transferência de dados. */
typedef struct {
    char serialPort[50];
    LinkLayerRole role;
    int baudRate;
    int nRetransmissions;
    int timeout;
} LinkLayer;
```

```
FUNÇÕES

/* Função que desativa o alarme e que lida com a contagem de alarmes acionados. */
void alarmHandler(int signal);

/* Função para a construção e envio das tramas de supervisão e não-numeradas. */
int sendFrame(int fd, unsigned char A, unsigned char C);

/* Função que estabelece a conexão entre transmissor e recetor. */
int llopen(LinkLayer connectionParameters);

/* Função responsável pelo envio de tramas.*/
int llwrite(const unsigned char *buf, int bufSize);

/* Função responsável pela leitura de tramas. */
int llread(unsigned char *packet);

/* Função que fecha a conexão entre transmissor e recetor. */
int llclose();
```

Casos de uso principais

O programa permite tanto enviar um ficheiro, no lado do **transmissor**, como receber um ficheiro, no lado do **recetor**. Para cada um dos casos, existem diferentes funções a serem utilizadas e, portanto, uma diferente sequência de chamadas.

Ambos começam por chamar a função **llopen()**, de modo a estabelecer uma ligação entre as duas partes. De seguida, o transmissor lê e reparte o ficheiro em várias partes com tamanho até um dado valor máximo, passando esses fragmentos em pacotes para a Camada de Ligação de Dados, através da função **llwrite().** Por sua vez, esta cria as tramas de informação e envia-as, ficando a aguardar uma resposta do recetor (**RR** ou **REJ**), que através da função **llread()**, se prepara para receber as tramas enviadas pelo transmissor, valida-as e envia uma resposta para o transmissor. O recetor, desta forma, vai reconstruindo o ficheiro com os vários fragmentos recebidos ao longo das chamadas à função. Por fim, ambos terminam a ligação, a partir da função **llclose()**. É de mencionar que, sempre que é necessário o envio de uma trama do tipo **S** (supervisão) ou **U** (não-numerada), é chamada a função **sendFrame()**, que constrói e envia as tramas referidas.

Protocolo de Ligação de Dados

O protocolo de Ligação de Dados interage diretamente com a Porta Série, e tem como objetivo garantir a transmissão de informação entre o transmissor e recetor de uma forma fiável. É responsável pela configuração da Porta Série, pelo estabelecimento e término da

conexão e pela transmissão de tramas de vários tipos. Esta comunicação segue o protocolo *Stop-and-Wait*.

Nesta camada, todas as tramas são validadas com o auxílio de uma *state machine* que lê a trama *byte* a *byte*, de forma a garantir a integridade da informação. Para além disso, no caso do transmissor, foi implementado um mecanismo de retransmissão de tramas, que permite reenviar uma trama caso este não receba uma resposta do recetor dentro do tempo esperado (*Timeout*), ou caso receba uma resposta errada. Após um certo número de retransmissões, o transmissor desiste de enviar a trama e a função retorna -1.

A função **llopen()** está encarregue da configuração e estabelecimento da conexão. Aqui é feito um *handshake* entre as duas máquinas, onde o transmissor envia uma trama de supervisão (tipo S), **SET**, e fica a aguardar uma resposta por parte do recetor, uma trama não numerada (tipo U), **UA**. O recetor, após receber e validar a trama **SET**, envia uma trama **UA**. Se o transmissor receber uma trama **UA**, a conexão é estabelecida com sucesso e a função retorna o *file descriptor* da Porta Série.

A função **llwrite()** é utilizada exclusivamente pelo transmissor e é responsável pelo envio dos dados. Esta função recebe como argumento pacotes de dados ou de controlo. Estes pacotes e o respetivo **BCC2** são submetidos a um processo de **byte stuffing**, onde os *bytes* de *flag* (**0x7e**) e de *escape* (**0x7d**) são substituídos pelas sequências **0x7d 0x5e**, e **0x7d 0x5d**, respetivamente. É então criada uma **trama de informação (Tipo I)**, com um *frame header* (F, A, C, BCC1), os *bytes* de dados, e um *frame trailer* (BCC2, F). Após envio desta trama, o transmissor fica a aguardar resposta por parte do recetor, na forma de uma trama de supervisão **REJ** ou **RR**, e reage de acordo com esta resposta (reenvia a trama ou prossegue no envio da próxima trama).

```
/*BCC incorrect */ (...)
else {  if (expectedN == 0) {
          if (sendFrame(fd, A_TR, C_REJ0) < 0) {
                   printf("Error: problem sending REJ0 frame\n"); return -1; }
            printf("REJ0 sent\n"); }
          else { (...) } /*for expected N=1 */
```

O recetor utiliza a função **llread()** para ler as tramas de informação. Verifica se o número da trama recebida é o esperado e, caso isso não aconteça, responde com ***RR<nº da trama esperada>***. Após fazer **destuffing** ao campo de dados, verifica o BCC2, e envia uma trama **REJ**, se este estiver errado, ou uma trama **RR**, no caso de não haver problemas.

```
   if (byte == 0x5e) packet[i++] = FLAG;    /* destuffing */
   else if (byte == 0x5d) packet[i++] = ESC;
   else { packet[i++] = ESC;  packet[i++] = byte;}
```

A ligação é terminada na função **llclose()** e o emissor invocará esta função quando o número de tentativas falhadas for excedido ou quando a transferência dos pacotes de dados der finalmente por concluída. De seguida, o emissor irá enviar uma trama de supervisão **DISC** e esperar pela resposta do recetor, que deverá ser também uma trama de supervisão **DISC**. Por fim, o emissor responde com **UA** e a ligação será interrompida.

Protocolo da Aplicação

É na Camada da Aplicação que ocorre a interação com o utilizador e o ficheiro que se pretende transferir. Para tal, define-se os aspetos que dizem respeito a essa transferência, nomeadamente qual é o ficheiro, a Porta Série, o número de bytes de dados do ficheiro a

serem enviados em cada pacote e a velocidade da transferência. Simultaneamente, também são passadas as informações de retransmissão, tal como o número máximo de retransmissões e o tempo máximo que o transmissor esperará pela resposta do recetor, até ser necessária uma retransmissão. Alguns destes parâmetros são passados para a camada de Ligação de Dados quando se chama a função **llopen()**, para iniciar a conexão entre o transmissor e recetor.

Se o primeiro passo for bem sucedido, o transmissor lê o ficheiro indicado, e começa por enviar um pacote de controlo para simbolizar o início da transmissão de dados (SOT - *Start Of Transmission*). O conteúdo do pacote de controlo é composto por 2 blocos do formato TLV (*Type, Length, Value*), um para representar o tamanho do ficheiro a ser transferido (Tipo 0) e o outro com o nome do ficheiro (Tipo 1).

```
    long int auxFileSize = size;
    unsigned char countBytes = 0;

    while (auxFileSize != 0) {
      auxFileSize = auxFileSize >> 8;
      countBytes++; }

    unsigned char c = 2;
    unsigned char t1 = 0, l1 = countBytes;
    unsigned char *v1 = malloc(countBytes);

    long int tempSize = size;
    for (int i = countBytes - 1; i >= 0; i--){
      v1[i] = (0xFF & tempSize);
      tempSize = tempSize >> 8;}
    unsigned char t2 = 1, l2 = strlen(filename);
    unsigned char *v2 = malloc(l2);

    for (int i = 0; i < l2; i++) v2[i] = filename[i];
    unsigned char *controlPacket = malloc(5 + l1 + l2);
    controlPacket[0] = c;
    controlPacket[1] = t1;
    controlPacket[2] = l1;

    for (int i = 0; i < l1; i++) controlPacket[3 + i] = v1[i];
    controlPacket[3 + l1] = t2;
    controlPacket[4 + l1] = l2;

    for (int i = 0; i < l2; i++) controlPacket[5 + l1 + i] = v2[i];
    if (llwrite(controlPacket, 5 + l1 + l2) < 0)
```

Depois disso, o reparte o conteúdo do ficheiro em fragmentos e cria pacotes de dados de tamanho *MAX_PAYLOAD_SIZE* e enviado através da função **llwrite()** da API da Camada de Ligação de Dados. De acordo com o valor retornado por esta função, a Camada de Aplicação sabe se deve enviar o próximo fragmento do ficheiro, reenviar o último fragmento a ser enviado, ou terminar a ligação através do **llclose()**, caso o número máximo de retransmissões tenha sido atingido.

```
while (sizeRemaining >= 0){
      int dataSize = sizeRemaining > (MAX_PAYLOAD_SIZE - 3) ? (MAX_PAYLOAD_SIZE - 3) :
sizeRemaining;
      int datapacketSize = dataSize + 3;
      unsigned char *dataPacket = malloc(datapacketSize);
      dataPacket[0] = 1;
      dataPacket[1] = dataSize >> 8;
      dataPacket[2] = dataSize & 0xFF;

      memcpy(dataPacket + 3, content + (size - sizeRemaining), dataSize);
      int res = 0;
```

```
      if ((res = llwrite(dataPacket, dataSize + 3)) < 0)
      {
        printf("Error: writing data packet to serial port\n");
        if (llclose(showStats) < 0){ printf("Error: closing serial port\n"); exit(-1); }
      exit(-1); }
      if (res == 0) continue;
      sizeRemaining -= MAX_PAYLOAD_SIZE - 3;
    }
```

Por fim, envia outro pacote de controlo para simbolizar o fim da transmissão (EOT), cujo conteúdo é semelhante ao pacote de controlo SOT.

Já o recetor, recebe todos os pacotes enviados através da função **llread()** da API. Após consumir o primeiro pacote de controlo, cria um ficheiro com o tamanho especificado e reconstrói-o com o conteúdo dos pacotes de dados que são recebidos corretamente. Após todos estes pacotes terem sido recebidos, consome outro pacote de controlo que simboliza o fim da transmissão.

```
long int sizeRemaining = fileSize;
    while (sizeRemaining >= 0) {
 int packetSize = sizeRemaining + 3 > MAX_PAYLOAD_SIZE ? MAX_PAYLOAD_SIZE : sizeRemaining + 3;
      unsigned char dataPacket[packetSize];
      int res;
      if ((res = llread(dataPacket)) < 0) {... /*error handling*/ }
      if(res == 0) continue;

      if (dataPacket[0] != 1){
        printf("Error: data packet is not correct\n");
        exit(-1); }

      int dataSize = 256 * dataPacket[1] + dataPacket[2];
      printf("Received Data Size: %d\n", dataSize);

      int bytesWritten = fwrite(dataPacket + 3, 1, dataSize, newFile);
      printf("Bytes written to file: %d\n", bytesWritten);
      fseek(newFile, 0L, SEEK_END);

      sizeRemaining -= MAX_PAYLOAD_SIZE - 3; }
```

Se todo este processo for executado corretamente, ambas as máquinas chamam a função **llclose()**, para dar término à sua conexão**.**

Validação

De modo a validar os resultados obtidos pelo nosso programa e garantir o bom funcionamento do mesmo em diversas situações, realizamos os seguintes testes:

- Envio de ficheiros de tamanhos diferentes (11KB, 150KB, 1.5 MB, 3.5MB)
- Envio do ficheiro com diferentes valores de *baudrate*
- Envio do pacotes de dados com diferentes tamanhos (variação do *MAX_PAYLOAD_SIZE*)
- Interrupção na ligação por diferentes períodos de tempo, de modo a testar o mecanismo de retransmissão com os alarmes e timeouts, e nas diversas fases da transmissão (estabelecimento da ligação → llopen, transmissão de dados → llwrite e llread, encerramento da ligação → llclose)
- Criação de ruído na porta série com um curto-circuito durante o envio do ficheiro
- Envio de tramas duplicadas
- Variação da FER através da geração aleatória de erros nas tramas de Informação
- Variação dos tempos de propagação das tramas

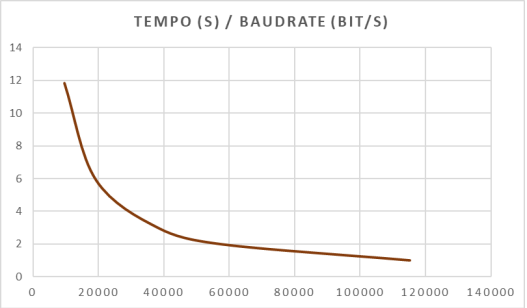
Todos estes parâmetros foram verificados devidamente, não só durante a nossa fase de testes, de uma forma mais exaustiva, mas também na presença do professor durante a apresentação do trabalho.

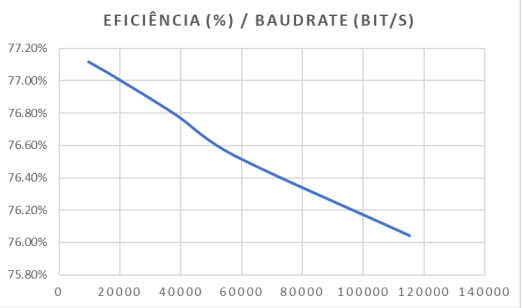
Eficiência do protocolo de Ligação de Dados

Variação do *Baudrate*

Para um ficheiro de 10968 bytes e um tamanho máximo de pacotes de 1000 bytes, obteve-se os seguintes resultados com a variação do *baudrate*:

Baudrate (bits/s)	Tempo (s)	Flow (bits/s)	Eficiência (%)
9600	11.8524	7403.0745	77.12%
19200	5.93424	14786.0601	77.01%
38400	2.97575	29486.3876	76.79%
57600	1.99032	44085.3068	76.54%
115200	1.00162	87601.7348	76.04%



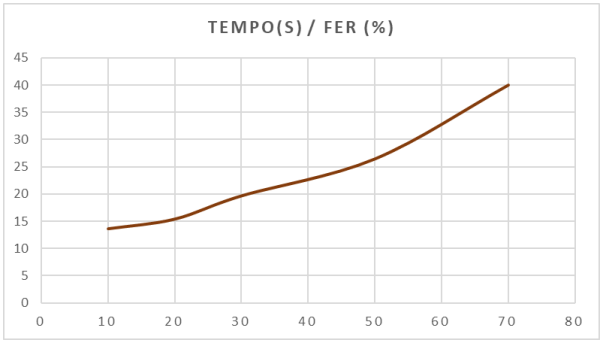


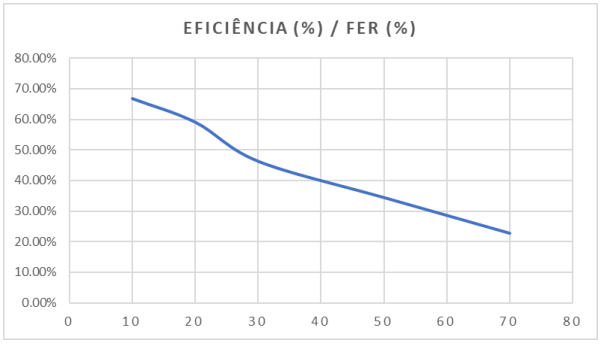
A partir dos gráficos obtidos, conclui-se que o tempo total da transferência do ficheiro é inversamente proporcional ao *baudrate*, apresentando uma descida mais acentuada para valores de *baudrate* menores, e uma estabilização progressiva. Uma vez que, consoante o aumento do *baudrate*, o tempo de transmissão da informação também aumenta, a eficiência diminui de forma aproximadamente linear.

Variação da taxa de erros (FER)

Para um ficheiro de 10968 bytes, um tamanho máximo de pacotes de 1000 bytes e um baudrate fixo de 9600 bits/s, obteve-se os seguintes resultados com a variação da FER (Frame Error Ratio):

FER (%)	Tempo(s)	Flow (bits/s)	Eficiência (%)
10	13.66562633	6420.781445	66.88%
20	15.44624	5680.605766	59.17%
30	19.70127767	4453.721301	46.39%
50	26.47217067	3314.575186	34.53%
70	39.994691	2193.891184	22.85%



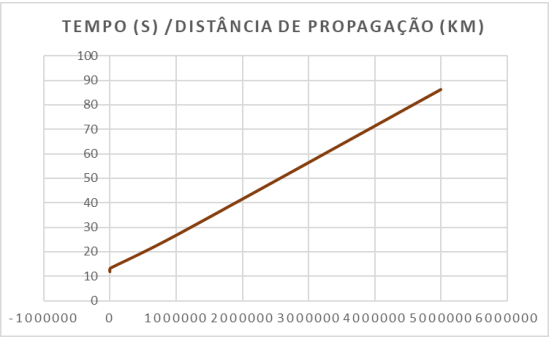


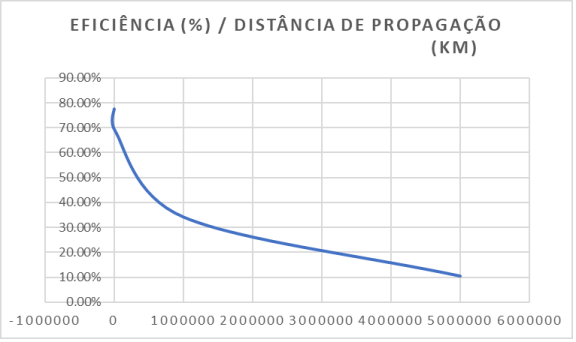
O tempo total da transferência do ficheiro é diretamente proporcional à percentagem de erros nas tramas (FER) como observado no gráfico. Tendo em conta uma base teórica, no protocolo *Stop-and-wait,* quando ocorre um erro na trama transferida, é necessário que ocorra uma retransmissão da mesma. Assim, logicamente, o tempo de transferência médio de cada trama correta será consequentemente maior, o que leva a um aumento do tempo global gasto na transferência. O segundo gráfico comprova, deste modo, que a eficiência diminui com o aumento da taxa de erros.

Variação da distância de propagação

Para simular a distância de propagação, utilizamos a *benchmark* de 5us/km. Para simular este atraso no tempo de propagação, utilizamos a função **usleep(),** que funciona em microssegundos, facilitando a implementação. Para um ficheiro de 10968 bytes, um tamanho máximo de pacotes de 1000 bytes e um *baudrate* fixo de 9600 bits/s, obteve-se os seguintes resultados com a variação da distância de propagação:

Distância (km)	Tempo (s)	Flow (bits/s	Eficiência (%)
1	11.7965	7438.110503	77.48%
100	11.8086	7430.511013	77.40%
10000	13.286	6604.222376	68.79%
1000000	26.7231	3283.445555	34.20%
5000000	86.4275	1015.232424	10.58%





Analisamos, pelos gráficos, que o tempo de transferência é diretamente proporcional à distância de propagação e que a eficiência, por outro lado, é inversamente proporcional à mesma. Os resultados relativos à eficiência estão de acordo com a expressão teórica:

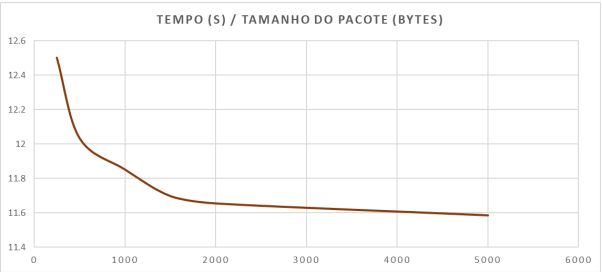
$$S = \frac{T_f}{T_{prop} + T_f + T_{prop}} = \frac{1}{1+2a}$$

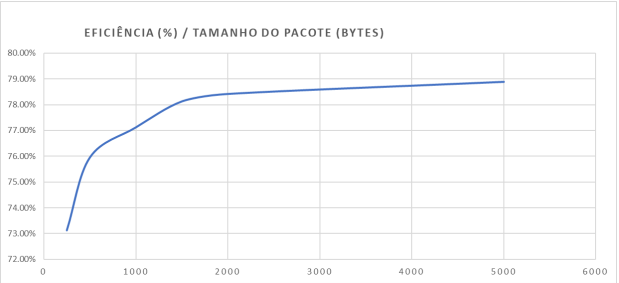
Uma maior distância traduz-se num tempo de propagação maior. Isto leva a um denominador maior nesta fração e, assim, a um "*S*", correspondente à eficiência, menor.

Variação do tamanho dos pacotes

Mantendo o baudrate de 9600 bits/s e o tamanho do ficheiro de 10968 bytes, a variação do tamanho máximo dos pacotes, em bytes, gerou os seguintes resultados:

Tamanho do pacote (bytes)	Tempo (s)	Flow (bits/s)	Eficiência (%)
250	12.5006486	7019.1558	73.12%
500	12.03434848	7291.1301	75.95%
1000	11.852373	7403.0745	77.12%
1500	11.69828519	7500.5865	78.13%
2000	11.65564309	7528.0274	78.42%
5000	11.5862849	7573.0919	78.89%





Através da análise dos gráficos, observa-se que o tempo total da transferência do ficheiro é inversamente proporcional ao tamanho do pacote, pois quanto menor for este tamanho, mais transmissões serão necessárias. Quanto à eficiência do protocolo, observamos um aumento acentuado quando o tamanho do pacote é menor, e uma certa estabilização consoante o tamanho também aumenta. Esta espode ser explicada pela fórmula menciona em cima, onde o tempo de propagação fica com uma preponderância cada vez menor à medida que se aumenta o tamanhos dos pacotes enviados.

Conclusões

O objetivo do projeto foi atingido com sucesso e vários desafios foram ultrapassados com determinação. Neste projeto, foi possível aplicar e consolidar os conceitos lecionados nas aulas práticas e teóricas, tal como o *byte stuffing* e *framing* e tornou-se evidente o funcionamento do protocolo ***Stop-and-Wait***. O projeto foi desenvolvido de acordo com a especificação fornecida, tendo em conta uma boa estruturação e robustez.

Anexo I

link_layer.c

```c
// Link layer protocol implementation

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

// MISC
#define _POSIX_SOURCE 1 // POSIX compliant source
#define BAUDRATE 38400
#define BUF_SIZE 256

#define FLAG 0x7E
#define ESC 0x7D
#define A_TR 0x03   // commands by Transmitter and replies by Receiver
#define A_RT 0x01   // commands by Receiver and replies by Transmitter
#define C_SET 0x03  // sent by the transmitter to initiate a connection
#define C_UA 0x07   // confirmation to the reception of a valid supervision frame
#define C_RR0 0x05  // sent by the Receiver that it is ready to receive an information frame
number 0
#define C_RR1 0x85  // sent by the Receiver that it is ready to receive an information frame
number 1
#define C_REJ0 0x01 // sent by the Receiver that it rejects an information frame number 0
(detected an error)
#define C_REJ1 0x81 // sent by the Receiver that it rejects an information frame number 1
(detected an error)
#define C_DISC 0x0B // indicates the termination of a connection
#define C_N0 0x00   // info frame number 0
#define C_N1 0x40   // info frame number 1

#define TRUE 1
#define FALSE 0

struct timeval start_time, end_time; // start and end time of the transfer

int ns = 0, nr = 1; // info frame sequence number: ns -> sender number, nr -> receiver number
int expectedN = 0;  // expected info frame number
int fd;             // file descriptor for serial port
int alarmEnabled = FALSE;
int alarmCount = 0;
```

```c
int timeout;
int nRetransmissions;
LinkLayerRole role;
struct termios oldtio, newtio;

typedef enum
{
  START,
  FLAG_RCV,
  A_RCV,
  C_RCV,
  BCC1_OK,
  READING,
  ESC_FOUND,
  STOP_State,
} State;

void alarmHandler(int signal)
{
  alarmEnabled = FALSE;
  alarmCount++;

  printf("Alarm #%d\n", alarmCount);
}

int sendFrame(int fd, unsigned char A, unsigned char C)
{
  unsigned char frame[5];
  frame[0] = FLAG;
  frame[1] = A;
  frame[2] = C;
  frame[3] = A ^ C;
  frame[4] = FLAG;

  int res = write(fd, frame, 5);
  if (res < 0)
  {
    printf("Error: problem writing to serial port\n");
    return -1;
  }
  return 0;
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters)
{

  gettimeofday(&start_time, NULL);
```

```
printf("serialPort - %s\n", connectionParameters.serialPort);

fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);

printf("fd - %d\n", fd);

if (fd < 0)
{
  perror(connectionParameters.serialPort);
  exit(-1);
}

if (tcgetattr(fd, &oldtio) == -1)
{
  perror("tcgetattr");
  exit(-1);
}

memset(&newtio, 0, sizeof(newtio));

newtio.c_cflag = connectionParameters.baudRate | CS8 | CLOCAL | CREAD;
newtio.c_iflag = IGNPAR;
newtio.c_oflag = 0;
newtio.c_lflag = 0;
newtio.c_cc[VTIME] = 1;
newtio.c_cc[VMIN] = 0;

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1)
{
  perror("tcsetattr");
  return -1;
}

State state = START;

unsigned char byte;
timeout = connectionParameters.timeout;
nRetransmissions = connectionParameters.nRetransmissions;
role = connectionParameters.role;
switch (connectionParameters.role)
{
case LlTx:
{
  (void)signal(SIGALRM, alarmHandler);

  while (state != STOP_State && alarmCount < nRetransmissions)
  {
    if (sendFrame(fd, (unsigned char)A_TR, (unsigned char)C_SET) < 0)
    {
```

```
    printf("Error: problem sending SET frame\n");
    return -1;
  }
  printf("SET sent\n");

  alarm(connectionParameters.timeout);
  alarmEnabled = TRUE;

  while (state != STOP_State && alarmEnabled)
  {
    if (read(fd, &byte, 1) < 0)
    {
      printf("Error: problem reading from serial port\n");
      return -1;
    }
    switch (state)
    {
    case START:
      if (byte == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (byte == A_TR)
        state = A_RCV;
      else if (byte != FLAG)
        state = START;
      break;
    case A_RCV:
      if (byte == C_UA)
        state = C_RCV;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (byte == (A_TR ^ C_UA))
        state = BCC1_OK;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case BCC1_OK:
      if (byte == FLAG)
      {
        state = STOP_State;
        alarm(0);
        printf("UA received\n");
      }
      else
```

```
          state = START;
        break;
      default:
        break;
      }
    }
  }
  if (alarmCount == nRetransmissions)
  {
    printf("Error: maximum number of retransmissions reached\n");
    return -1;
  }

  break;
}
case LlRx:
{

  // state machine
  while (state != STOP_State)
  {
    if (read(fd, &byte, 1) < 0)
    {
      printf("Error: problem reading from serial port\n");
      return -1;
    }

    switch (state)
    {
    case START:
      if (byte == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (byte == A_TR)
        state = A_RCV;
      else if (byte != FLAG)
        state = START;
      break;
    case A_RCV:
      if (byte == C_SET)
        state = C_RCV;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (byte == (A_TR ^ C_SET))
        state = BCC1_OK;
      else if (byte == FLAG)
```

```
          state = FLAG_RCV;
        else
          state = START;
        break;
      case BCC1_OK:
        if (byte == FLAG)
        {
          state = STOP_State;
          printf("SET received\n");
        }
        else
          state = START;
        break;
      default:
        break;
      }
    }
    if (sendFrame(fd, A_TR, C_UA) < 0)
    {
      printf("Error: problem sending UA frame\n");
      return -1;
    }
    printf("UA sent\n");
    break;
  }

  default:
    return -1;
    break;
  }

  return fd;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize)
{
  alarmCount = 0;

  int frameSize = 6 + bufSize;
  unsigned char *frame = (unsigned char *)malloc(frameSize);

  frame[0] = FLAG;
  frame[1] = A_TR;
  frame[2] = ns == 0 ? C_N0 : C_N1;
  frame[3] = frame[1] ^ frame[2];

  int j = 4; // data index in frame
  for (unsigned int i = 0; i < bufSize; i++)
```

```
{
  if (buf[i] == FLAG || buf[i] == ESC)
  {
    frame = realloc(frame, ++frameSize);
    frame[j++] = ESC;
    frame[j++] = buf[i] ^ 0x20;
  }
  else
  {
    frame[j++] = buf[i];
  }
}

unsigned char BCC2 = buf[0];

for (unsigned int i = 1; i < bufSize; i++)
  BCC2 ^= buf[i];

if (BCC2 == FLAG || BCC2 == ESC)
{
  frame = realloc(frame, ++frameSize);
  frame[j++] = ESC;
  frame[j++] = BCC2 ^ 0x20;
}
else
{
  frame[j++] = BCC2;
}
frame[j++] = FLAG;

State state = START;
unsigned char byte;
unsigned char c_byte;
int rejected = 0, accepted = 0;

while (alarmCount < nRetransmissions && state != STOP_State)
{
  printf("I frame #%d sent\n", ns);
  write(fd, frame, j);
  alarm(timeout);
  alarmEnabled = TRUE;

  // state machine
  while (state != STOP_State && alarmEnabled)
  {
    if (read(fd, &byte, 1) < 0)
    {
      printf("Error: problem reading from serial port\n");
      return -1;
    }
```

```
    switch (state)
    {
    case START:
      if (byte == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (byte == A_TR)
        state = A_RCV;
      else if (byte != FLAG)
        state = START;
      break;
    case A_RCV:
      if (byte == C_RR0 || byte == C_RR1)
      {
        c_byte = byte;
        state = C_RCV;
      }
      else if (byte == C_REJ0 || byte == C_REJ1)
      {
        c_byte = byte;
        state = C_RCV;
      }
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (byte == (A_TR ^ c_byte))
        state = BCC1_OK;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case BCC1_OK:
      if (byte == FLAG)
      {
        state = STOP_State;
        if (c_byte == C_RR0 || c_byte == C_RR1)
        {
          printf("RR%d received\n", c_byte == C_RR0 ? 0 : 1);
          ns = (ns + 1) % 2;
          rejected = 0;
          accepted = 1;
        }
        else if (c_byte == C_REJ0)
        {
          printf("REJ0 received\n");
          ns = 0;
```

```
            rejected = 1;
          }
          else if (c_byte == C_REJ1)
          {
            printf("REJ1 received\n");
            ns = 1;
            rejected = 1;
          }
          alarm(0);
        }

        else
          state = START;
        break;
      default:
        break;
      }
    }
  }

  free(frame);

  if (alarmCount == nRetransmissions)
  {
    printf("Error: maximum number of retransmissions reached\n");
    return -1;
  }

  if (accepted && !rejected)
    return frameSize;
  else if (rejected && !accepted)
    return 0;
  return -1;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet)
{

  unsigned char byte;
  int i = 0;
  State state = START;
  char c;

  // state machine
  while (state != STOP_State)
  {
    if (read(fd, &byte, 1) < 0)
    {
```

```
    printf("Error: problem reading from serial port\n");
    return -1;
  }

  switch (state)
  {
  case START:
  {
    i = 0;
    if (byte == FLAG)
      state = FLAG_RCV;
    break;
  }
  case FLAG_RCV:
  {
    if (byte == A_TR)
      state = A_RCV;
    else if (byte != FLAG)
      state = START;
    else
      return 0;
    break;
  }
  case A_RCV:
  {
    if (byte == C_N0 || byte == C_N1)
    {
      c = byte;
      state = C_RCV;
    }
    else if (byte == C_DISC)
    {
      printf("DISC received\n");
      return -1;
    }
    else if (byte == FLAG)
      state = FLAG_RCV;
    else
      state = START;
    break;
  }
  case C_RCV:
  {
    if (byte == (A_TR ^ c))
    {
      if (c == C_N0)
      {
        if (expectedN == 0)
          state = READING;
        else if (expectedN == 1)
        {
```

```
            printf("Error: Wrong frame received. Expected %d\n", expectedN);
            if (sendFrame(fd, A_TR, C_RR1) < 0)
            {
              printf("Error: problem sending RR1 frame\n");
              return -1;
            }
            printf("RR1 sent\n");

            return 0;
          }
        }
        else if (c == C_N1)
        {
          if (expectedN == 1)
            state = READING;
          else if (expectedN == 0)
          {
            printf("Error: Wrong frame received. Expected %d\n", expectedN);
            if (sendFrame(fd, A_TR, C_RR0) < 0)
            {
              printf("Error: problem sending RR0 frame\n");
              return -1;
            }
            printf("RR0 sent\n");

            return 0;
          }
        }
      }
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    }
    case READING:
    {
      if (byte == ESC)
        state = ESC_FOUND;
      else if (byte == FLAG)
      {

        unsigned char bcc2 = packet[i - 1];
        i--;
        packet[i] = '\0';
        unsigned char verif = packet[0];

        for (int j = 1; j < i; j++)
        {
          verif = verif ^ packet[j];
        }
```

```
    if (bcc2 == verif)
    {
      state = STOP_State;
      printf("I frame #%d received\n", expectedN);

      if (expectedN == 0)
      {
        expectedN = 1;
        if (sendFrame(fd, A_TR, C_RR1) < 0)
        {
          printf("Error: problem sending RR1 frame\n");
          return -1;
        }
        printf("RR1 sent\n");
      }
      else
      {
        expectedN = 0;
        if (sendFrame(fd, A_TR, C_RR0) < 0)
        {
          printf("Error: problem sending RR0 frame\n");
          return -1;
        }
        printf("RR0 sent\n");
      }
      return i;
    }
    else
    {
      printf("BCC2 incorrect\n");
      if (expectedN == 0)
      {
        if (sendFrame(fd, A_TR, C_REJ0) < 0)
        {
          printf("Error: problem sending REJ0 frame\n");
          return -1;
        }
        printf("REJ0 sent\n");
      }
      else
      {
        if (sendFrame(fd, A_TR, C_REJ1) < 0)
        {
          printf("Error: problem sending REJ1 frame\n");
          return -1;
        }
        printf("REJ1 sent\n");
      }

      return 0;
```

```
        }
      }
      else
      {
        packet[i] = byte;
        i++;
      }
      break;
    }
    case ESC_FOUND:
    {
      state = READING;

      if (byte == 0x5e) // destuffing
      {
        packet[i] = FLAG;
        i++;
      }
      else if (byte == 0x5d)
      {
        packet[i] = ESC;
        i++;
      }
      else
      {
        packet[i] = ESC;
        i++;
        packet[i] = byte;
        i++;
      }
      break;
    }
    default:
      break;
    }
  }

  return 0;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics)
{
  alarmCount = 0;

  State state = START;
  unsigned char byte;
  printf("Preparing to close the connection...\n");
  switch (role)
```

```
{
case LlTx:
{
  (void)signal(SIGALRM, alarmHandler);

  while (alarmCount < nRetransmissions && state != STOP_State)
  {
    if (sendFrame(fd, A_TR, C_DISC) < 0)
    {
      printf("Error: problem sending DISC frame\n");
      return -1;
    }
    printf("DISC sent\n");
    alarm(timeout);
    alarmEnabled = TRUE;

    // state machine
    while (state != STOP_State && alarmEnabled)
    {
      if (read(fd, &byte, 1) < 0)
      {
        printf("Error: problem reading from serial port\n");
        return -1;
      }
      switch (state)
      {
      case START:
        if (byte == FLAG)
          state = FLAG_RCV;
        break;
      case FLAG_RCV:
        if (byte == A_TR)
          state = A_RCV;
        else if (byte != FLAG)
          state = START;
        break;
      case A_RCV:
        if (byte == C_DISC)
          state = C_RCV;
        else if (byte == FLAG)
          state = FLAG_RCV;
        else
          state = START;
        break;
      case C_RCV:
        if (byte == (A_TR ^ C_DISC))
          state = BCC1_OK;
        else if (byte == FLAG)
          state = FLAG_RCV;
        else
          state = START;
```

```
        break;
      case BCC1_OK:
        if (byte == FLAG)
        {
          state = STOP_State;
          alarm(0);
          printf("DISC received\n");
        }
        else
          state = START;
        break;
      default:
        break;
      }
    }
  }

  if (alarmCount == nRetransmissions)
  {
    printf("Error: maximum number of retransmissions reached\n");
    return -1;
  }

  if (sendFrame(fd, A_TR, C_UA) < 0)
  {
    printf("Error: problem sending UA frame\n");
    return -1;
  }
  printf("UA sent\n");
  break;
}
case LlRx:
{
  while (state != STOP_State)
  {
    if (read(fd, &byte, 1) < 0)
    {
      printf("Error: problem reading from serial port\n");
      return -1;
    }
    switch (state)
    {
    case START:
      if (byte == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (byte == A_TR)
        state = A_RCV;
      else if (byte != FLAG)
        state = START;
```

```
      break;
    case A_RCV:
      if (byte == C_DISC)
        state = C_RCV;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (byte == (A_TR ^ C_DISC))
        state = BCC1_OK;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case BCC1_OK:
      if (byte == FLAG)
      {
        state = STOP_State;
        printf("DISC received\n");
      }
      else
        state = START;
      break;
    default:
      break;
    }
  }

  if (sendFrame(fd, (unsigned char)A_TR, (unsigned char)C_DISC) < 0)
  {
    printf("Error: problem sending DISC frame\n");
    return -1;
  }
  printf("DISC sent\n");

  state = START;

  // state machine
  while (state != STOP_State)
  {
    if (read(fd, &byte, 1) < 0)
    {
      printf("Error: problem reading from serial port\n");
      return -1;
    }
    switch (state)
    {
    case START:
```

```
      if (byte == FLAG)
        state = FLAG_RCV;
      break;
    case FLAG_RCV:
      if (byte == A_TR)
        state = A_RCV;
      else if (byte != FLAG)
        state = START;
      break;
    case A_RCV:
      if (byte == C_UA)
        state = C_RCV;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case C_RCV:
      if (byte == (A_TR ^ C_UA))
        state = BCC1_OK;
      else if (byte == FLAG)
        state = FLAG_RCV;
      else
        state = START;
      break;
    case BCC1_OK:
      if (byte == FLAG)
      {
        state = STOP_State;
        printf("UA received\n");
      }
      else
        state = START;
      break;
    default:
      break;
    }
  }

  break;
}
}

if (tcsetattr(fd, TCSANOW, &oldtio) == -1)
{
  perror("tcsetattr");
  exit(-1);
}

printf("Closing...\n");
close(fd);
```

```
  if (showStatistics)
  {
    gettimeofday(&end_time, NULL);
    double elapsed_time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_usec -
start_time.tv_usec) / 1000000.0;
    printf("Total time: %f seconds\n", elapsed_time);
  }

  return 1;
}
```

application_layer.c

```
// Aplication layer protocol implementation

#include "application_layer.h"
#include "link_layer.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int showStats = TRUE;

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

  LinkLayer connectionParameters = {
      .baudRate = baudRate,
      .timeout = timeout,
      .nRetransmissions = nTries,
  };
  strcpy(connectionParameters.serialPort, serialPort);
  connectionParameters.role = strcmp(role, "tx") == 0 ? LlTx : LlRx;

  printf("Opening connection...\n");
  int fd = llopen(connectionParameters);
  if (fd < 0)
  {
    printf("Error: opening connection\n");

    if (llclose(showStats) < 0)
      {
```

```
      printf("Error: closing serial port\n");
      exit(-1);
    }
}
printf("Connection opened sucessfully\n");
printf("--------------------\n");

switch (connectionParameters.role)
{
case LlTx:
{

  FILE *file = fopen(filename, "r");
  if (file == NULL)
  {
    printf("Error: opening file\n");
    exit(-1);
  }

  long int pos1 = ftell(file);
  fseek(file, 0L, SEEK_END);
  long int size = ftell(file) - pos1;
  printf("File size: %ld\n", size);
  fseek(file, 0L, SEEK_SET);

  long int auxFileSize = size;
  unsigned char countBytes = 0;

  while (auxFileSize != 0)
  {
    auxFileSize = auxFileSize >> 8;
    countBytes++;
  }

  unsigned char c = 2;
  unsigned char t1 = 0, l1 = countBytes;
  unsigned char *v1 = malloc(countBytes);

  long int tempSize = size;
  for (int i = countBytes - 1; i >= 0; i--)
  {
    v1[i] = (0xFF & tempSize);
    tempSize = tempSize >> 8;
```

```
}

unsigned char t2 = 1, l2 = strlen(filename);
unsigned char *v2 = malloc(l2);

for (int i = 0; i < l2; i++)
{
  v2[i] = filename[i];
}


unsigned char *controlPacket = malloc(5 + l1 + l2);
controlPacket[0] = c;
controlPacket[1] = t1;
controlPacket[2] = l1;

for (int i = 0; i < l1; i++)
{
  controlPacket[3 + i] = v1[i];
}

controlPacket[3 + l1] = t2;
controlPacket[4 + l1] = l2;

for (int i = 0; i < l2; i++)
{
  controlPacket[5 + l1 + i] = v2[i];
}

if (llwrite(controlPacket, 5 + l1 + l2) < 0)
{
  printf("Error: writing SOT Control Packet to serial port\n");
  if (llclose(showStats) < 0)
  {
    printf("Error: closing serial port\n");
    exit(-1);
  }
  exit(-1);
}
printf("SOT \n");

unsigned char *content = malloc(size);
fread(content, 1, size, file);
```

```
    long int sizeRemaining = size;
    printf("---------------------\n");
    while (sizeRemaining >= 0)
    {
      printf("Size Remaining: %ld\n", sizeRemaining);
      int dataSize = sizeRemaining > (MAX_PAYLOAD_SIZE - 3) ? (MAX_PAYLOAD_SIZE
- 3) : sizeRemaining;

      int datapacketSize = dataSize + 3;

      unsigned char *dataPacket = malloc(datapacketSize);
      dataPacket[0] = 1;

      dataPacket[1] = dataSize >> 8;
      dataPacket[2] = dataSize & 0xFF;


      memcpy(dataPacket + 3, content + (size - sizeRemaining), dataSize);


      int res = 0;
      if ((res = llwrite(dataPacket, dataSize + 3)) < 0)
      {
        printf("Error: writing data packet to serial port\n");
        if (llclose(showStats) < 0)
      {
        printf("Error: closing serial port\n");
        exit(-1);
      }
        exit(-1);
      }
      printf("Bytes written: %d\n", res);
      if (res == 0) continue;


      sizeRemaining -= MAX_PAYLOAD_SIZE - 3;
      printf("---------------------\n");
    }

    c = 3;
    controlPacket[0] = c;

    if (llwrite(controlPacket, 5 + l1 + l2) < 0)
```

```
    {
      printf("Error: writing EOT control packet to serial port\n");
      if (llclose(showStats) < 0)
      {
        printf("Error: closing serial port\n");
        exit(-1);
      }
      exit(-1);
    }
    printf("End of Transmission\n");
    printf("--------------------\n");

    if (llclose(showStats) < 0)
    {

      printf("Error: closing serial port\n");
      exit(-1);
    }

    break;
  }

  case LlRx:
  {

    unsigned char *controlPacket = malloc(5);

    if (llread(controlPacket) < 0)
    {
      printf("Error: reading SOT control packet from serial port\n");
      exit(-1);
    }

    if (controlPacket[0] != 2)
    {
      printf("Error: control packet is not SOT\n");
      exit(-1);
    }

    printf("Received SOT Control Packet\n");

    long int fileSize = 0;
    for (int i = 0; i < controlPacket[2]; i++)
```

```
    {
      fileSize = fileSize << 8;
      fileSize += controlPacket[3 + i];
    }

    FILE *newFile = fopen(filename, "wb");
    if (newFile == NULL)
    {
      printf("Error: opening file\n");
      exit(-1);
    }
    printf("---------------------\n");
    long int sizeRemaining = fileSize;
    while (sizeRemaining >= 0)
    {
      int packetSize = sizeRemaining + 3 > MAX_PAYLOAD_SIZE ? MAX_PAYLOAD_SIZE :
sizeRemaining + 3;

      unsigned char dataPacket[packetSize];
      int res;
      if ((res = llread(dataPacket)) < 0)
      {
        printf("Error: reading data packet from serial port\n");

        if (llclose(showStats) < 0)
        {
          printf("Error: closing serial port\n");
          exit(-1);
        }
        return;
      }
      if(res == 0){
        continue;
      }

      if (dataPacket[0] != 1)
      {
        printf("dataPacket[0]=%0x", dataPacket[0]);
        printf("Error: data packet is not correct\n");
        exit(-1);
      }
```

```
      int dataSize = 256 * dataPacket[1] + dataPacket[2];
      printf("Received Data Size: %d\n", dataSize);

      int bytesWritten = fwrite(dataPacket + 3, 1, dataSize, newFile);
      printf("Bytes written to file: %d\n", bytesWritten);
      fseek(newFile, 0L, SEEK_END);

      sizeRemaining -= MAX_PAYLOAD_SIZE - 3;
      printf("---------------------\n");
    }

    fclose(newFile);

    if (llread(controlPacket) < 0)
    {
      printf("Error: reading EOT control packet from serial port\n");
      exit(-1);
    }

    if (controlPacket[0] != 3)
    {
      printf("Error: control packet is not EOT\n");
      exit(-1);
    }

    printf("Received EOT Control Packet\n");
    printf("---------------------\n");

    if (llclose(showStats) < 0)
    {
      printf("Error: closing serial port\n");
      exit(-1);
    }

    break;
  }

  default:
    exit(-1);
    break;
  }
}
```