Protocolo de Ligação de Dados

Trabalho 1

Turma 15

Manuel Ramos Leite Carvalho Neto up202108744

Matilde Isabel da Silva Simões up202108782

Licenciatura em Engenharia Informática e Computação

Redes de Computadores

2023/2024

Sumário

Este projeto – desenvolvido no âmbito da Unidade Curricular de Redes de Computadores – enquadra-se no estudo dos protocolos de ligação de dados, tendo em vista o desenvolvimento e a caracterização de um protocolo do tipo *Stop & Wait*. Este protocolo deve seguir as diretrizes fornecidas para a especificação do mesmo e possibilitar a transferência de dados através da Porta Série RS-232.

Assim, o trabalho prático permitiu-nos aplicar na prática os conceitos aprendidos nas aulas, consolidando o nosso conhecimento sobre os protocolos de ligação de dados *Stop-and-Wait* num contexto real.

Introdução

O propósito do trabalho é a implementação de um protocolo de comunicação para a transmissão de ficheiros através da Porta Série RS-232, bem como de uma aplicação que permita testar este protocolo, transferindo um ficheiro entre dois dispositivos. A estratégia utlizada, *Stop & Wait,* pressupõe o envio de um único pacote de dados e a espera por uma confirmação de receção positiva antes de enviar o próximo pacote, assegurando, assim, que todos os dados são recebidos corretamente e em ordem. A finalidade do relatório é documentar o trabalho efetuado, caracterizando o protocolo desenvolvido e medindo a sua eficiência.

Este relatório encontra-se estruturado em mais sete secções, que contêm as informações seguintes:

1. **Arquitetura**: descrição das duas camadas implementadas e das interfaces que fornecem
2. **Estrutura do Código**: apresentação das APIs, estruturas de dados e principais funções utilizadas
3. **Casos de Uso Principais**: identificação das funcionalidades essenciais do projeto e explicação das sequências de chamada de funções
4. **Protocolo de Ligação Lógica**: caracterização do funcionamento do protocolo de ligação lógica e das estratégias subjacentes à sua implementação
5. **Protocolo de Aplicação**: caracterização do funcionamento da camada de aplicação e das estratégias de implementação recorridas
6. **Validação**: referência aos testes realizados para avaliar a robustez do trabalho desenvolvido
7. **Eficiência do Protocolo de Ligação de Dados**: comparação e análise dos resultados sobre a eficiência do protocolo implementado na camada de ligação de dados

Arquitetura

A arquitetura desenvolvida tem como base a implementação de duas camadas principais.

Por um lado, a **camada de ligação de dados** (*Link Layer*) – presente no ficheiro *link_layer.c* – é responsável pela implementação do protocolo de comunicação. Esta camada tem quatro funções principais: estabelecer a ligação entre transmissor e recetor, terminá-la, enviar tramas de dados através da porta série e validar as tramas recebidas. Se alguma destas etapas falhar, a camada de ligação de dados deve enviar uma mensagem de erro. A receção de dados, analisando-os e verificando sua a integridade, é também uma função crucial desta camada. Deste modo, são implementadas estratégias de deteção de erros e de retransmissão de tramas em caso de necessidade. A par disto, a *Link Layer* recorre a um *timeout* para decidir se e quando deve tentar efetuar retransmissões. Por fim, esta camada implementa também um mecanismo de *stuffing* e *destuffing* dos dados para permitir a inclusão de determinados caracteres na informação a transmitir. Através do recurso a um protocolo *Stop & Wait*, a camada de ligação de dados garante que a transmissão é feita de forma eficiente e segura, minimizando possíveis perdas ou corrupções dos dados.

Por outro lado, a **camada de aplicação** (*Application Layer)* – presente no ficheiro *application_layer.c* – recorre à camada de ligação de dados para transferir e receber pacotes. Esta camada é responsável por criar pacotes de controlo – sinalizam o início e o fim do conteúdo a transmitir – e pacotes de dados – contêm a informação do ficheiro. Nesta camada são definidos parâmetros relevantes para a comunicação (como o tamanho máximo das tramas de informação), configurados de acordo com as necessidades da aplicação e as capacidades do sistema de comunicação.

O programa é executado usando **dois terminais**, um em cada computador interveniente na ligação, ligados por um cabo série. Um terminal executa o programa em modo de transmissão e o outro em modo de receção. Isto permite que um dispositivo envie dados (transmissor) e o outro receba esses dados (recetor).

Estrutura do Código

O código é constituído por várias estruturas e funções, que se destacam nas duas subsecções abaixo.

Link Layer

State: enumeração que representa os estados possíveis da máquina de estados, correspondendo cada um à leitura bem-sucedida de um campo da trama

LinkLayer**:** estrutura de dados que armazena os parâmetros com os quais configurar a porta série

alarmHandler(int signal)*:* acionada quando um sinal de alarme é recebido, incrementa o valor da variável *alarmCount*

processByte(unsigned char a, unsigned char c1, unsigned char c2, unsigned char *aCheck, unsigned char *cCheck, State *state)*:* lê e processa cada byte recebido, utilizando um *switch-case* para gerir a máquina de estados

llopen(LinkLayer connectionParameters): inicia/estabelece a ligação, configurando a porta série para iniciar a comunicação entre o transmissor e o recetor

llwrite(const unsigned char *buf, int bufSize)*:* escreve dados na porta série (recorrendo a *byte stuffing* para evitar conflitos com os *bytes* reservados para a implementação do protocolo) e envia-os para o recetor

llread(unsigned char *packet): lê os dados da porta série, validando-os e respondendo de forma adequada

llclose(int showStatistics): fecha a ligação anteriormente estabelecida, imprimindo estatísticas se solicitado

Application Layer

buildControlPacket(unsigned char controlField, long int fileSize, const char *fileName, int *packetSize): constrói um pacote de controlo com o tamanho e o nome do ficheiro a transmitir

buildDataPacket(int dataSize, unsigned char *data, int *packetSize)*:* constrói um pacote de dados a partir de um determinado conteúdo e do seu tamanho

sendDataPacket(int size, unsigned char *fileContent)*:* envia um pacote de dados com um determinado número de dados do conteúdo do ficheiro

parseControlPacket(unsigned char *packet, int *fileSize)*:* lê e interpreta um pacote de controlo recebido

applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries, int timeout, const char *filename): coordena a comunicação entre o emissor e o recetor, realizando todo o processo desde o estabelecimento até ao término da ligação, passando pela transferência (transmissão e receção) dos dados

Casos de Uso Principais

Na Tabela 1 apresentam-se as funções utilizadas e a sequência de chamadas para o transmissor e o recetor.

Transmissor	Recetor
1. Inicializa as configurações de ligação e estabelece-a, invocando ***llopen***	1. Inicializa as configurações de ligação e estabelece-a, invocando ***llopen***
2. Abre o ficheiro para a leitura dos dados e obtém o seu tamanho	2. Abre um novo ficheiro para escrita
3. Constrói e envia um pacote de controlo *START*, invocando ***buildControlPacket*** e ***llwrite***	3. Entra num ciclo que aguarda a chegada de pacotes, invocando ***llread***
4. Constrói e envia pacotes de dados a partir do conteúdo lido do ficheiro, invocando ***buildDataPacket*** e ***sendDataPacket*** (que, por sua vez, invoca ***llwrite***)	4. Quando um pacote de controlo *START* é recebido, obtém o nome e o tamanho do ficheiro, invocando ***parseControlPacket***
5. Constrói e envia um pacote de controlo *END,* invocando ***buildControlPacket*** e ***llwrite***	5. Quando um pacto de dados é recebido, escreve-os no ficheiro aberto
6. Fecha o ficheiro aberto para leitura	6. Quando um pacote de controlo *END* é recebido, obtém o nome e o tamanho do ficheiro, invocando ***parseControlPacket*** e terminando o ciclo
7. Conclui a ligação, invocando ***llclose***	7. Fecha o ficheiro aberto para escrita
	8. Conclui a ligação, invocando ***llclose***

Tabela 1 - Funções Utilizadas e Sequência de Chamadas

Protocolo de Ligação Lógica

A camada de ligação de dados desempenha um papel crucial na comunicação entre transmissor e recetor através da porta série. É responsável por **estabelecer e encerrar a ligação**, além de **gerir o envio e receção das tramas**. Para alcançar esse objetivo, é utilizado o protocolo *Stop & Wait*.

O **estabelecimento de ligação** é realizado pela função ***llopen***. Nesta função, depois de abrir e configurar a porta série com os valores pretendidos para a ligação, o transmissor envia uma trama de supervisão *SET* e aguarda a resposta do recetor na forma de uma trama *UA*. O recetor, ao receber o *SET*, responde com *UA*, indicando que a ligação foi estabelecida com sucesso. A partir desse momento, a transmissão/receção de dados está pronta para ser iniciada.

A **transmissão de dados** é tratada pela função ***llwrite***. Esta função recebe os pacotes a enviar, computa o *BCC2* como o *XOR* de todos os *bytes* do pacote e aplica *byte stuffing* para evitar conflitos com *bytes* que coincidam com a *flag* ou com o caracter de *escape*. Posteriormente, o conteúdo é encapsulado numa trama de informação e enviado ao recetor, ficando o transmissor a aguardar por uma resposta. Se a resposta for *RR*, a trama foi recebida com sucesso e a função termina. Se ocorrer *timeout* ou se a resposta for *REJ*, a trama não foi recebida com sucesso e o envio é repetido até ser aceite ou até atingir o número máximo de tentativas de retransmissão.

A **receção de dados** é efetuada pela função ***llread***. Esta função lê os dados recebidos pela porta série e, num primeiro momento, verifica se a trama recebida é a esperada, através dos campos *BCC1* e de controlo. Depois, realiza o processo de *destuffing* do campo de dados e valida o *BCC2* para identificar possíveis erros durante a transmissão, respondendo de forma apropriada: com *RR* ou *REJ*, conforme explicado anteriormente.

O **fecho da ligação** é feito pela função ***llclose***. Nesta função, o transmissor envia uma trama de supervisão *DISC* e aguarda pela resposta do recetor, que também envia uma trama *DISC* e encerra imediatamente a ligação. Após a receção do *DISC*, o transmissor responde com uma trama *UA* e finaliza a ligação.

Protocolo de Aplicação

O protocolo de aplicação foi projetado para **construir e interpretar pacotes de controlo e de dados**, bem como **enviar e receber** esses pacotes. A camada de aplicação interage com o utilizador, permitindo a especificação do ficheiro a ser transferido, da porta série, da capacidade da ligação, do número máximo de *bytes* de dados a transferir de cada vez, do número máximo de retransmissões e do tempo máximo de espera pela resposta do recetor.

O processo de transferência do ficheiro começa com a **construção de um pacote de controlo do tipo *START***, através da função ***buildControlPacket***. Este pacote contém o tamanho e o nome do ficheiro a transferir.

De seguida, o **conteúdo do ficheiro é lido e dividido** em pacotes de dados, com recurso à função ***buildDataPacket***. Cada pacote codifica o tamanho dos dados que contém, num valor não superior a *MAX_DATA_SIZE.* Estes pacotes são enviados pela função ***sendDataPacket***.

Terminada a transferência do ficheiro, um **pacote de controlo do tipo *END* é construído** pela função ***buildControlPacket***. Este pacote contém as mesmas informações do que o pacote *START*.

No lado do transmissor, todos os pacotes são enviados com invocações a *llwrite* (*Link Layer*).

No lado do recetor, todos os pacotes são lidos/recebidos por chamadas sucessivas à função *llread* (*Link Layer*). Para **interpretar os pacotes de controlo, extraindo o tamanho do ficheiro e o nome**, recorre-se à função ***parseControlPacket***. O ficheiro é escrito em sequência, correspondendo cada parte a um pacote de dados recebido.

Validação

Foram realizados múltiplos testes para validar a correta implementação do protocolo, destacando-se:

1. Transferência de ficheiros diferentes (em tipo e em tamanho)
2. Transferência do ficheiro com diferentes capacidades da ligação (*baudrate*)
3. Transferência de pacotes de dados com diferentes tamanhos máximos
4. Transferência de pacotes de dados criados especificamente com caracteres que necessitam de *stuffing*
5. Transferência do ficheiro com interrupção total da ligação
6. Transferência do ficheiro com desfasamento entre o início do transmissor e o início do recetor
7. Transferência do ficheiro com a introdução de ruído na ligação
8. Transferência do ficheiro com a introdução de erros nas tramas (usando uma versão modificada do *cable.c*)

Eficiência do Protocolo de Ligação de Dados

Nesta secção, pretende-se caracterizar estatisticamente a eficiência do protocolo de ligação de dados e compará-lo com a definição teórica de um protocolo do tipo *Stop & Wait*. Nesse sentido, foram variados quatro parâmetros e efetuadas medições dos tempos (de propagação e de *frame*) de maneira a calcular o valor da eficiência. Em cada uma das quatro subsecções seguintes apresentam-se os dados obtidos experimentalmente e os valores calculados a partir deles, bem como um gráfico que compara a eficiência determinada a partir dos tempos medidos – $S(FER, a)$ ou $S(a)$ – com a eficiência em termos de aproveitamento da capacidade da ligação – $S(R, C)$.

As fórmulas utilizadas foram as seguintes (R representa o débito recebido):

$$a = \frac{T_{prop}}{T_{frame}} \qquad S(FER, a) = \frac{1 - FER}{1 + 2a} \qquad S(a) = \frac{1}{1 + 2a} \qquad S(R, C) = \frac{R}{C}$$

Variação do *Frame Error Ratio* (FER)

Fixando-se a capacidade de ligação em $C = 9600\ bit/s$ e o tamanho máximo das Tramas de Informação em $L = 256\ bytes$, variou-se o *Frame Error Ratio* entre 0 e 0,7 através da geração aleatória de erros em Tramas de Informação, tendo-se obtido os valores expostos na Tabela 2 e no gráfico da Figura 1.

FER	$T_{prop}\ [ms]$	$T_{frame}\ [ms]$	a	$S(FER, a)$	$R\ [bit/s]$	$S(R, C)$
0	8	254	0,031	**94** %	7733	**81** %
0,1	7	256	0,027	**85** %	7559	**79** %
0,2	5	255	0,021	**77** %	6792	**71** %
0,3	6	254	0,025	**67** %	5034	**52** %
0,4	10	254	0,040	**56** %	4334	**45** %
0,5	11	255	0,042	**46** %	3960	**41** %
0,6	10	254	0,041	**37** %	3448	**36** %
0,7	8	255	0,031	**28** %	2613	**27** %

Tabela 2 – Variação do Frame Error Ratio *(FER)*

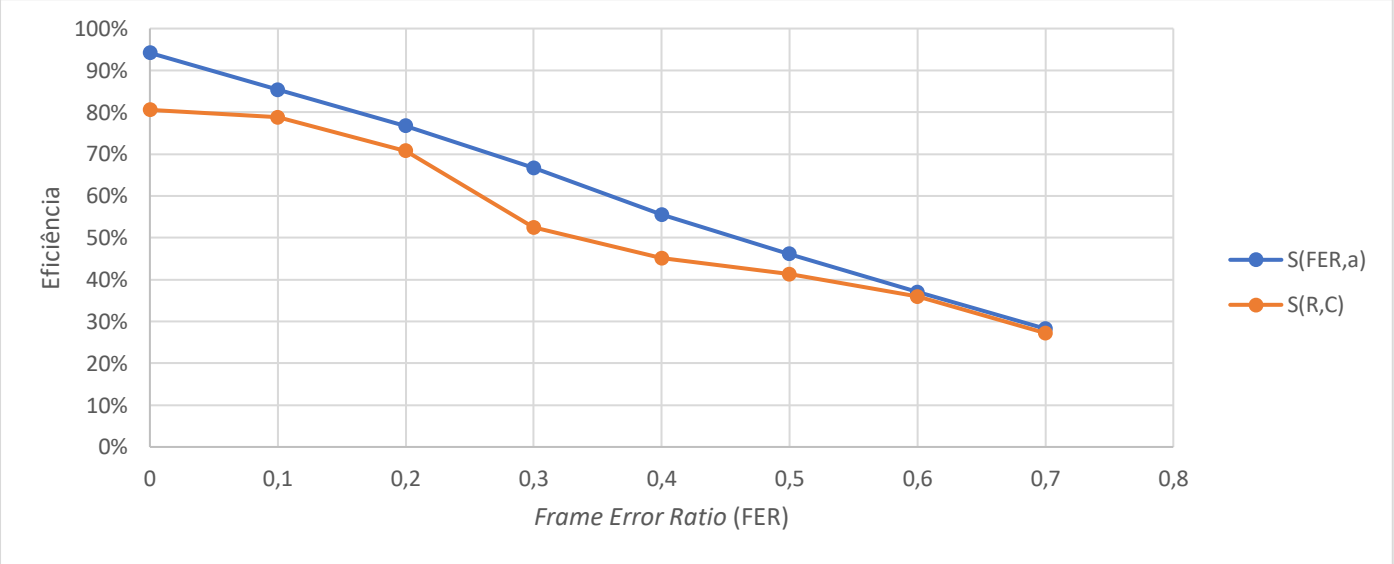

Figura 1 – Variação do Frame Error Ratio *(FER)*

Conforme comprovam os valores calculados para a eficiência, esta decresce de forma aproximadamente linear em relação ao aumento do *Frame Error Ratio*. Este comportamento vai ao encontro do esperado, pois uma maior probabilidade de ocorrência de erros por Trama de Informação obriga a um maior número de retransmissões, o que se reflete num aumento do tempo necessário para transferir os dados corretos e num menor aproveitamento da capacidade da ligação.

Variação do Tempo de Propagação

Fixando-se a capacidade de ligação em $C = 9600\ bit/s$ e o tamanho máximo das Tramas de Informação em $L = 256\ bytes$, variou-se o tempo de propagação através da introdução de um atraso (entre 0 e 1,4 segundos) no processamento de cada trama recebida, tendo-se obtido os valores expostos na Tabela 3 e no gráfico da Figura 2.

Note-se que, por se alterar o valor do tempo de propagação apenas no processamento das tramas recebidas, não se apresenta o valor do tempo de propagação, mas sim do tempo de ida e volta ($T_{roundtrip} = 2T_{prop} + T_{frame}$), tendo a eficiência $S(a)$ sido calculada com a fórmula equivalente à anteriormente mencionada: $S(a) = \frac{T_{frame}}{T_{roundtrip}}$.

$Atraso\ [s]$	$T_{rountrip}\ [ms]$	$T_{frame}\ [ms]$	$S(a)$	$R\ [bit/s]$	$S(R,C)$
0	270	255	**94** %	7735	**81** %
0,2	278	254	**91** %	7511	**78** %
0,4	410	255	**62** %	5095	**53** %
0,6	610	254	**42** %	3424	**36** %
0,8	810	254	**31** %	2579	**27** %
1	1010	255	**25** %	2068	**22** %
1,2	1210	256	**21** %	1726	**18** %
1,4	1410	254	**18** %	1481	**15** %

Tabela 3 – Variação do Tempo de Propagação

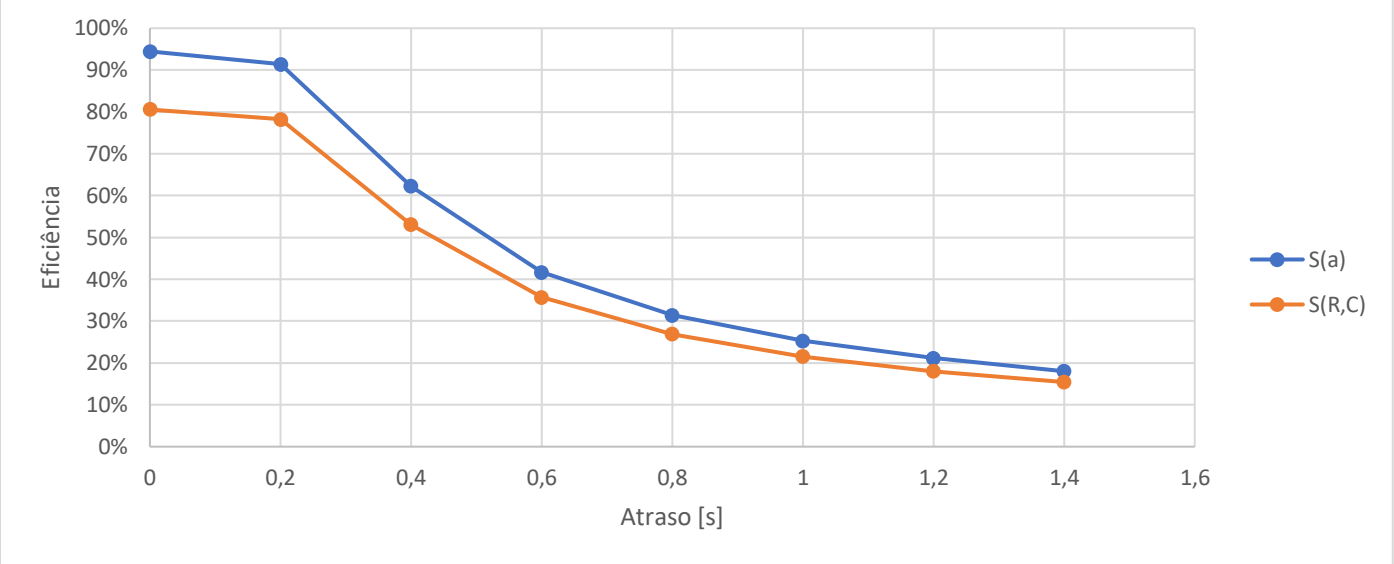

Figura 2 – Variação do Tempo de Propagação

Conforme comprovam os valores calculados para a eficiência, esta decresce de forma aproximadamente quadrática em relação ao aumento do tempo de propagação. Este comportamento vai ao encontro do esperado, pois um atraso no tempo de propagação implica forçosamente um desaproveitamento da capacidade máxima da ligação.

Variação da Capacidade da Ligação

Fixando-se o tamanho máximo das Tramas de Informação em $L = 256\ bytes$, variou-se a capacidade da ligação entre $1200\ bit/s$ e $155200\ bit/s$, tendo-se obtido os valores expostos na Tabela 4 e no gráfico da Figura 3.

$C\ [bit/s]$	$T_{prop}\ [ms]$	$T_{frame}\ [ms]$	a	$S(a)$	$R\ [bit/s]$	$S(R,C)$
1200	40	2079	0,019	**96** %	967	**81** %
2400	36	1041	0,035	**94** %	1877	**78** %
4800	18	520	0,035	**93** %	3753	**78** %
9600	10	259	0,037	**93** %	7504	**78** %
19200	5	130	0,037	**93** %	15000	**78** %
38400	2	65	0,038	**93** %	29973	**78** %
57600	2	43	0,040	**93** %	44905	**78** %
155200	1	21	0,060	**89** %	89645	**78** %

Tabela 4 – Variação da Capacidade da Ligação

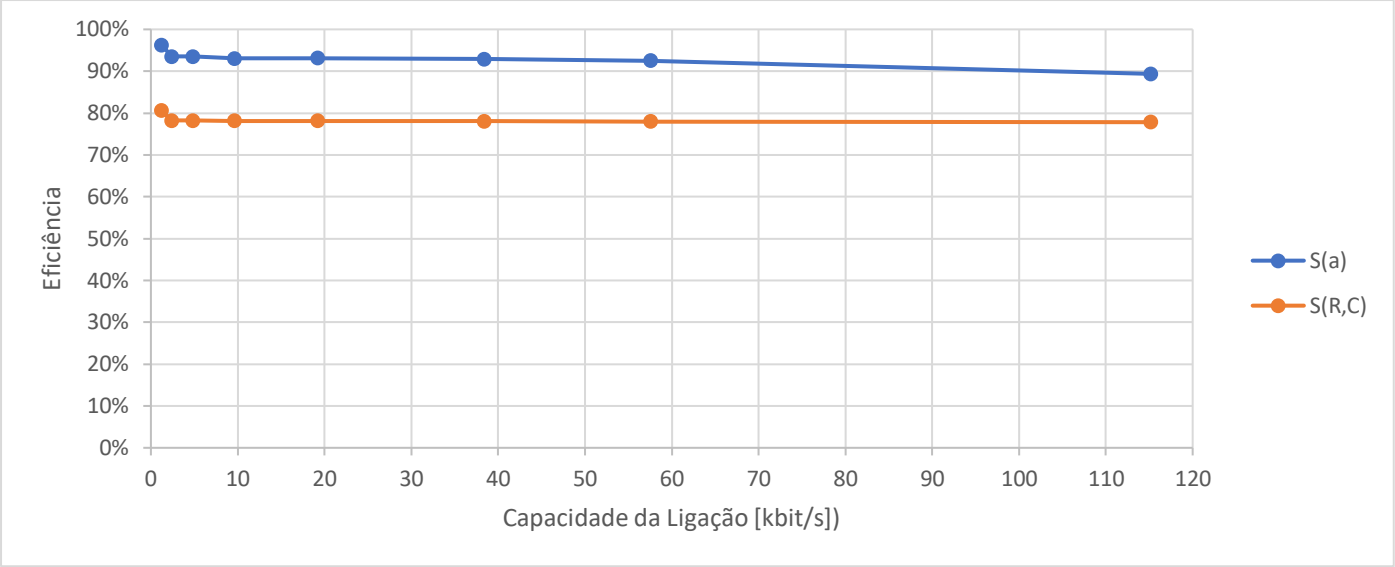

Figura 3 – Variação da Capacidade da Ligação

Conforme comprovam os valores calculados para a eficiência, esta decresce ligeiramente com o aumento da capacidade da ligação. Este comportamento deverá ser, em princípio, consequência das características físicas do meio que se revela com maior dificuldade para suportar uma maior velocidade/exigência no processo de transmissão.

Variação do Tamanho das Tramas de Informação

Fixando-se a capacidade da ligação em $C = 9600\ bit/s$, variou-se o tamanho máximo das Tramas de Informação entre $64\ bits$ e $896\ bits$, tendo-se obtido os valores expostos na Tabela 5 e no gráfico da Figura 4.

Note-se que, na verdade, estes valores-limite correspondem ao tamanho máximo para o campo de dados das Tramas de Informação antes de *stuffing* e depois de *destuffing*, pelo que os cálculos foram ajustados de modo a refletir isso mesmo e a garantir uma aproximação à realidade quanto ao número de bits transmitidos aquando do envio de cada trama.

$L\ [bit]$	$T_{prop}\ [ms]$	$T_{frame}\ [ms]$	a	$S(a)$	$R\ [bit/s]$	$S(R,C)$
64	5	70	0,078	**86** %	6842	**71** %
128	6	133	0,044	**92** %	7328	**76** %
256	8	255	0,030	**94** %	7733	**81** %
384	9	368	0,023	**96** %	8077	**84** %
512	11	471	0,023	**96** %	8384	**87** %
640	11	567	0,019	**96** %	8758	**91** %
768	14	661	0,022	**96** %	8958	**93** %
896	15	752	0,019	**96** %	9234	**96** %

Tabela 5 - Variação do Tamanho das Tramas de Informação

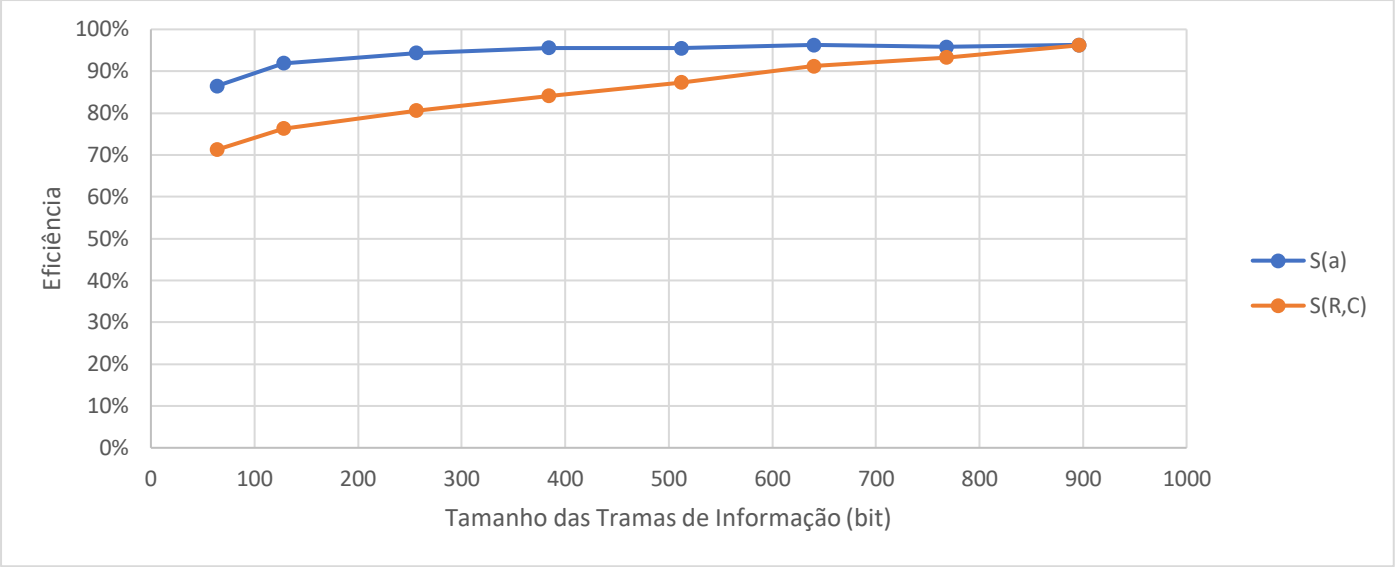

Figura 4 - Variação do Tamanho das Tramas de Informação

Conforme comprovam os valores calculados para a eficiência, esta aumenta com o aumento da capacidade da ligação (até um valor máximo aproximadamente constante). Este comportamento deve-se ao facto de, enquanto o número de bits a transmitir em cada trama for inferior àquele que o meio físico consegue suportar, este valor poder sempre ser aumentado sem que isso revele nenhuma desvantagem, resultando, portanto, num aumento da eficácia da transmissão.

Em suma, as quatro análises efetuadas revelam que a caracterização teórica dos protocolos *Stop & Wait* coincide com os resultados para a eficiência obtidos na prática. De facto, foi-nos possível calcular e comparar os valores para a eficiência, quer em termos dos tempos de propagação e de *frame*, quer em termos do débito recebido e da capacidade da ligação. As diferenças entre os valores obtidos e os esperados podem ser justificadas por erros de medição, nomeadamente no que concerne à medição dos tempos por parte do computador (flutuações entre diferentes execuções do programa e baixa precisão nas medições).

Note-se que, para conseguir variar os parâmetros em causa nas quatro subsecções acima e para medir os tempos necessários aos cálculos, tiveram de ser acrescentados alguns elementos ao código, como, por exemplo, uma estrutura de controlo do tipo *switch-case* que converte a capacidade de ligação de *int* para *speed_t*.

Conclusões

Este projeto pretendeu que se desenvolvesse e verificasse um protocolo de ligação de dados para transferir ficheiros através da Porta Série RS-232. O protocolo *Stop & Wait* foi implementado para garantir a transferência correta dos dados e a minimização dos erros. O sistema possui duas camadas principais: a *Link Layer* – que gere a comunicação, incluindo o envio e receção de tramas – e a *Application Layer* – que lida com a criação e análise de pacotes de controlo e dados, além da transmissão e receção dos mesmos. A sequência de operações envolve a configuração/estabelecimento da ligação, a transferência e receção de dados, seguidos do fecho da ligação. Finalmente, foi medida também a eficiência do protocolo desenvolvido e comparados os valores com os esperados, de acordo com a caracterização teórica dos protocolos deste tipo.

Este projeto permitiu-nos uma compreensão prática aprofundada sobre o funcionamento e a implementação dos protocolos de comunicação, bem como sobre a sua eficiência: como calculá-la e quais os parâmetros que a afetam.

Anexo I – Código Fonte

link_layer.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/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 0x7E
#define A 0x03
#define A_CLOSE 0x01
#define C_SET 0x03
#define C_UA 0x07
#define C_RR(r) (((r) << 7) | 0x05)
#define C_REJ(r) (((r) << 7) | 0x01)
#define C_DISC 0x0B

#define N(s) ((s) << 6)

#define ESC 0x7D
#define FLAG_ESCAPED 0x5E
#define ESC_ESCAPED 0x5D

typedef enum {
    START_STATE,
    FLAG_RCV_STATE,
    A_RCV_STATE,
    C_RCV_STATE,
    BCC_OK_STATE,
    STOP_STATE
} State;

int fd;
int alarmEnabled = FALSE;
int alarmCount = 0;
int nRetransmissions;
```

```c
int timeout;
LinkLayerRole role;

// Estatísticas
clock_t start;
int totalTramas = 0;
int totalTramasI = 0;
int totalTramasSU = 0;
int totalSET = 0;
int totalUA = 0;
int totalRR = 0;
int totalREJ = 0;
int totalDISC = 0;
int totalBytes = 0;
int totalRetransmissions = 0;
int totalBCC1 = 0;
int totalBCC2 = 0;
int totalDuplicados = 0;
int totalOpen = 0;
int totalWrite = 0;
int totalRead = 0;
int totalClose = 0;
int totalStuffed = 0;
int totalFlagStuffed = 0;
int totalEscStuffed = 0;

// Imprime "Link Layer" seguido do título e do conteúdo
void printLL(char *title, unsigned char *content, int contentSize) {
    // DEBUG
    printf("\nLink Layer\n");
    for (int i = 0; title[i] != '\0'; i++) printf("%c", title[i]);
    printf("\n");
    for (int i = 0; i < contentSize; i++) printf("0x%x ", content[i]);
    printf("\n");
}

// Converte int em speed_t
speed_t get_baudrate(int baudrate) {
    switch (baudrate) {
        case 1200:
            return B1200;
        case 2400:
            return B2400;
        case 4800:
            return B4800;
        case 9600:
            return B9600;
        case 19200:
            return B19200;
        case 38400:
            return B38400;
        case 57600:
```

```
            return B57600;
        case 115200:
            return B115200;
        default:
            return B0;
    }
}

// Lida com uma interrupção do alarme: desativa-o, incrementa um contador e imprime
"ALARM"
void alarmHandler(int signal) {
    alarmEnabled = FALSE;
    alarmCount++;
    printf("\nALARM\n");
}

/**
 * Máquina de estados que processa cada byte lido da porta série
 * @param a valor esperado no campo A
 * @param c1 um dos possíveis valores esperados no campo C
 * @param c2 outro dos possíveis valores esperados no campo C
 * @param aCheck valor lido do campo A
 * @param cCheck valor lido do campo C
 * @param state estado atual
 *
 * @details
 * A existência dos parâmetros c1 e c2 permite aproveitar a mesma máquina de estados para
llopen, llwrite, llread e llclose
 * Em llopen, só existe um valor esperado para o campo C (C_SET ou C_UA), pelo que c1 = c2
 * Em llread, existem dois valores esperados para o campo C (C_RR e C_REJ), pelo que c1 !=
c2
 * Em llwrite, existem dois valores esperados para o campo C (N(0) e N(1)), pelo que c1 !=
c2
 * Em llclose, só existe um valor esperado para o campo C (C_DISC), pelo que c1 = c2
 */
void processByte(unsigned char a, unsigned char c1, unsigned char c2, unsigned char
*aCheck, unsigned char *cCheck, State *state) {
    unsigned char byteRead;

    if (read(fd, &byteRead, sizeof(byteRead)) == sizeof(byteRead)) {
        totalBytes++;
        printLL("Byte Lido", &byteRead, sizeof(byteRead));  // DEBUG
        switch (*state) {
            case START_STATE:
                if (byteRead == FLAG)
                    *state = FLAG_RCV_STATE;
                else
                    *state = START_STATE;
                break;
            case FLAG_RCV_STATE:
                if (byteRead == FLAG)
                    *state = FLAG_RCV_STATE;
```

```
                else if (byteRead == a) {
                    *aCheck = byteRead;
                    *state = A_RCV_STATE;
                } else
                    *state = START_STATE;
                break;
            case A_RCV_STATE:
                if (byteRead == FLAG)
                    *state = FLAG_RCV_STATE;
                else if (byteRead == c1 || byteRead == c2) {
                    *cCheck = byteRead;
                    *state = C_RCV_STATE;
                } else
                    *state = START_STATE;
                break;
            case C_RCV_STATE:
                if (byteRead == FLAG)
                    *state = FLAG_RCV_STATE;
                else if (byteRead == (*aCheck ^ *cCheck))
                    *state = BCC_OK_STATE;
                else {
                    totalBCC1++;
                    *state = START_STATE;
                }
                break;
            case BCC_OK_STATE:
                if (byteRead == FLAG)
                    *state = STOP_STATE;
                else
                    *state = START_STATE;
                break;
            default:
                break;
        }
    }
}

////////////////////////////////////////////////
// LLOPEN
////////////////////////////////////////////////
int llopen(LinkLayer connectionParameters) {
    totalOpen++;
    start = clock();
    fd = open(connectionParameters.serialPort, O_RDWR | O_NOCTTY);
    if (fd < 0) {
        printf("Erro a abrir a porta série %s\n", connectionParameters.serialPort);
        return -1;
    }

    struct termios oldtio;
    struct termios newtio;
```

```
    if (tcgetattr(fd, &oldtio) == -1) {
        printf("Erro a usar tcgetattr\n");
        return -1;
    }

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

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

    tcflush(fd, TCIOFLUSH);

    if (tcsetattr(fd, TCSANOW, &newtio) == -1) {
        printf("Erro a usar tcsetattr\n");
        return -1;
    }

    nRetransmissions = connectionParameters.nRetransmissions;
    timeout = connectionParameters.timeout;
    role = connectionParameters.role;

    State state = START_STATE;
    unsigned char aCheck;
    unsigned char cCheck;

    if (connectionParameters.role == LlTx) {
        (void)signal(SIGALRM, alarmHandler);
        int tries = nRetransmissions;
        unsigned char set[5] = {FLAG, A, C_SET, A ^ C_SET, FLAG};

        do {
            printLL("LLOPEN - enviado SET", set, sizeof(set));  // DEBUG
            totalTramas++;
            totalTramasSU++;
            totalSET++;
            write(fd, set, sizeof(set));
            alarm(timeout);
            alarmEnabled = TRUE;
            while (alarmEnabled == TRUE && state != STOP_STATE) {
                // Enquanto o alarme não tiver disparado e estado não for o final,
processa os bytes da porta série (um de cada vez)
                processByte(A, C_UA, C_UA, &aCheck, &cCheck, &state);  // espera um UA
            }
            if (state == STOP_STATE) {
                // O estado final foi alcançado, pelo que o alarme pode ser desativado
                alarm(0);
                alarmEnabled = FALSE;
            } else {
```

```
                // O alarme tocou, pelo que ocorreu timeout e deve haver retransmissão (se
ainda não tiver sido excedido o número máximo de tentativas)
                tries--;
                totalRetransmissions++;
            }
        } while (tries >= 0 && state != STOP_STATE);

        if (state != STOP_STATE) {
            // Foi excedido o número máximo de tentativas de retransmissão
            totalRetransmissions--;
            printf("LLOPEN - UA não foi recebido\n");
            return -1;
        }
    } else if (connectionParameters.role == LlRx) {
        while (state != STOP_STATE) {
            // Processa os bytes da porta série (um de cada vez)
            processByte(A, C_SET, C_SET, &aCheck, &cCheck, &state);  // espera um SET
        }
        unsigned char ua[5] = {FLAG, A, C_UA, A ^ C_UA, FLAG};
        printLL("LLOPEN - enviado UA", ua, sizeof(ua));  // DEBUG
        totalTramas++;
        totalTramasSU++;
        totalUA++;
        write(fd, ua, sizeof(ua));  // quando receber o SET, responde com UA
    } else {
        printf("Erro em connectionParameters.role\n");
        return -1;
    }

    return 1;
}

////////////////////////////////////////////////
// LLWRITE
////////////////////////////////////////////////
int llwrite(const unsigned char *buf, int bufSize) {
    totalWrite++;
    unsigned char bcc2 = buf[0];
    for (int i = 1; i < bufSize; i++) {
        // Cálculo do BCC2
        bcc2 ^= buf[i];
    }

    unsigned char *dataBcc2 = (unsigned char *)malloc(2 * bufSize + 2);  // aloca memória
dinâmica para o pior caso: ter de fazer stuffing de todos os bytes de dados e do BCC2
    int index = 0;
    for (int i = 0; i < bufSize; i++) {
        // Stuffing dos dados
        if (buf[i] == FLAG) {
            totalStuffed++;
            totalFlagStuffed++;
            dataBcc2[index++] = ESC;
```

```
            dataBcc2[index++] = FLAG_ESCAPED;
        } else if (buf[i] == ESC) {
            totalStuffed++;
            totalEscStuffed++;
            dataBcc2[index++] = ESC;
            dataBcc2[index++] = ESC_ESCAPED;
        } else {
            dataBcc2[index++] = buf[i];
        }
    }

    // Stuffing do BCC2
    if (bcc2 == FLAG) {
        totalStuffed++;
        totalFlagStuffed++;
        dataBcc2[index++] = ESC;
        dataBcc2[index++] = FLAG_ESCAPED;
    } else if (bcc2 == ESC) {
        totalStuffed++;
        totalEscStuffed++;
        dataBcc2[index++] = ESC;
        dataBcc2[index++] = ESC_ESCAPED;
    } else {
        dataBcc2[index++] = bcc2;
    }

    static unsigned char tramaI = 0;
    unsigned char n = N(tramaI);
    unsigned char bcc1 = A ^ n;

    unsigned char next = (tramaI + 1) % 2;

    // Construção do frame a transmitir
    unsigned char *frame = malloc(index + 5);  // 5 -> F A C BCC1 F;
    frame[0] = FLAG;
    frame[1] = A;
    frame[2] = n;
    frame[3] = bcc1;
    for (int i = 4; i < (index + 4); i++) {
        frame[i] = dataBcc2[i - 4];
    }
    frame[index + 4] = FLAG;

    int size = index + 5;  // 5 -> F A C BCC1 F

    State state = START_STATE;
    unsigned char aCheck;
    unsigned char cCheck;
    unsigned char accepetedCheck;
    unsigned char rejectedCheck;

    int tries = nRetransmissions;
```

```
    do {
        printLL("LL WRITE - frame enviado", frame, size);  // DEBUG
        totalTramas++;
        totalTramasI++;
        write(fd, frame, size);
        alarm(timeout);
        alarmEnabled = TRUE;
        accepetedCheck = FALSE;
        rejectedCheck = FALSE;
        while (alarmEnabled == TRUE && rejectedCheck == FALSE && accepetedCheck == FALSE)
{
            state = START_STATE;
            while (state != STOP_STATE && alarmEnabled == TRUE) {
                // Enquanto o alarme não tiver disparado e estado não for o final,
processa os bytes da porta série (um de cada vez)
                processByte(A, C_RR(next), C_REJ(tramaI), &aCheck, &cCheck, &state);  //
espera um RR ou REJ
            }

            if (state == STOP_STATE) {
                // O estado final foi alcançado, pelo que o alarme pode ser desativado
                alarm(0);
                alarmEnabled = FALSE;
            } else {
                // O alarme tocou, pelo que ocorreu timeout e deve haver retransmissão (se
ainda não tiver sido excedido o número máximo de tentativas)
                tries--;
                totalRetransmissions++;
                continue;
            }

            // Interpretação da Resposta
            if (cCheck == C_RR(next)) {
                // O frame enviado foi recebido e aceite - o recetor está pronto para
receber o próximo frame
                accepetedCheck = TRUE;
                tramaI = next;
            } else if (cCheck == C_REJ(tramaI)) {
                // O frame enviado foi rejeitado - deve ser retransmitido (se ainda não
tiver sido excedido o número máximo de tentativas)
                rejectedCheck = TRUE;
            }
        }
    } while (tries >= 0 && accepetedCheck == FALSE);

    if (state != STOP_STATE) {
        // Foi excedido o número máximo de tentativas de retransmissão
        totalRetransmissions--;
        printf("LLWRITE - não foi recebida resposta\n");
        return -1;
    }
```

```
    free(dataBcc2);
    free(frame);
    return size;
}

////////////////////////////////////////////////
// LLREAD
////////////////////////////////////////////////
int llread(unsigned char *packet) {
    totalRead++;
    static unsigned char tramaI = 0;

    State state = START_STATE;
    unsigned char byteRead;
    unsigned char aCheck;
    unsigned char cCheck;
    unsigned char escFound = FALSE;

    int index = 0;
    int size;

    while (state != BCC_OK_STATE) {
        // Enquanto o estado não for o BCC OK, processa os bytes da porta série (um de
cada vez)
        processByte(A, N(0), N(1), &aCheck, &cCheck, &state);
    }

    if (cCheck != N(tramaI)) {
        // Recebeu uma trama de que não estava à espera (duplicada)
        totalDuplicados++;
        while (byteRead != FLAG) {
            // Lê os bytes da porta série (um de cada vez), mas ignora-os - apenas para
limpar
            totalBytes++;
            read(fd, &byteRead, sizeof(byteRead));
        }
        // Responde com a indicação de qual é o índice da trama que está pronto para
receber
        unsigned char n = C_RR(tramaI);
        unsigned char rr[5] = {FLAG, A, n, A ^ n, FLAG};
        printLL("LL WRITE - RR enviado", rr, sizeof(rr));  // DEBUG
        totalTramas++;
        totalTramasSU++;
        totalRR++;
        write(fd, rr, sizeof(rr));
        return -1;
    }

    while (state != STOP_STATE) {
        // Enquanto o estado não for o final, processa os bytes da porta série (um de cada
vez)
```

```
        if (state == BCC_OK_STATE && read(fd, &byteRead, sizeof(byteRead)) ==
sizeof(byteRead)) {
            totalBytes++;
            // Destuffing dos dados e do BCC2
            if (escFound) {
                if (byteRead == FLAG_ESCAPED) {
                    totalStuffed++;
                    totalFlagStuffed++;
                    packet[index++] = FLAG;
                } else if (byteRead == ESC_ESCAPED) {
                    totalStuffed++;
                    totalEscStuffed++;
                    packet[index++] = ESC;
                }
                escFound = FALSE;
            } else if (byteRead == ESC) {
                escFound = TRUE;
            } else if (byteRead == FLAG) {
                size = index + 5;  // 5 -> F A C BCC1 F
                unsigned char bcc2 = packet[index - 1];
                index--;
                packet[index] = '\0';                                   // retira o BCC2 do
pacote de dados
                printLL("LLWRITE - pacote recebido", packet, index);  // DEBUG
                unsigned char bcc2Acc = packet[0];
                for (int i = 1; i < index; i++) {
                    // Cálculo do BCC2
                    bcc2Acc ^= packet[i];
                }
                if (bcc2 == bcc2Acc) {
                    // O valor de BCC2 está correto, pelo que a trama foi recebida com
sucesso e o recetor está pronto para a próxima
                    tramaI = (tramaI + 1) % 2;
                    unsigned char n = C_RR(tramaI);
                    unsigned char rr[5] = {FLAG, A, n, A ^ n, FLAG};
                    printLL("LLWRITE - RR enviado", rr, sizeof(rr));  // DEBUG
                    totalTramas++;
                    totalTramasSU++;
                    totalRR++;
                    write(fd, rr, sizeof(rr));
                    state = STOP_STATE;
                } else {
                    // O valor de BCC está incorreto, pelo que a trama deve ser
retransmitida
                    totalBCC2++;
                    unsigned char n = C_REJ(tramaI);
                    unsigned char rej[5] = {FLAG, A, n, A ^ n, FLAG};
                    printLL("LLWRITE - REJ enviado", rej, sizeof(rej));  // DEBUG
                    totalTramas++;
                    totalTramasSU++;
                    totalREJ++;
                    write(fd, rej, sizeof(rej));
```

```
                    state = STOP_STATE;
                    return -1;
                }
            } else {
                packet[index++] = byteRead;
            }
        }
    }
    return size;
}

////////////////////////////////////////////////
// LLCLOSE
////////////////////////////////////////////////
int llclose(int showStatistics) {
    totalClose++;
    State state = START_STATE;
    unsigned char aCheck;
    unsigned char cCheck;

    int tries = nRetransmissions;

    if (role == LlTx) {
        unsigned char disc[5] = {FLAG, A, C_DISC, A ^ C_DISC, FLAG};
        do {
            printLL("LLCLOSE - enviado DISC", disc, sizeof(disc));  // DEBUG
            totalTramas++;
            totalTramasSU++;
            totalDISC++;
            write(fd, disc, sizeof(disc));
            alarm(timeout);
            alarmEnabled = TRUE;
            while (alarmEnabled == TRUE && state != STOP_STATE) {
                // Enquanto o alarme não tiver disparado e estado não for o final,
processa os bytes da porta série (um de cada vez)
                processByte(A_CLOSE, C_DISC, C_DISC, &aCheck, &cCheck, &state);  // espera
um DISC
            }
            if (state == STOP_STATE) {
                // O estado final foi alcançado, pelo que o alarme pode ser desativado
                alarm(0);
                alarmEnabled = FALSE;
            } else {
                // O alarme tocou, pelo que ocorreu timeout e deve haver retransmissão (se
ainda não tiver sido excedido o número máximo de tentativas)
                tries--;
                totalRetransmissions++;
            }
        } while (tries >= 0 && state != STOP_STATE);

        if (state != STOP_STATE) {
            // Foi excedido o número máximo de tentativas de retransmissão
```

```
            totalRetransmissions--;
            printf("LLCLOSE - DISC não foi recebido\n");
            return -1;
        }

        unsigned char ua[5] = {FLAG, A_CLOSE, C_UA, A_CLOSE ^ C_UA, FLAG};
        printLL("LLCLOSE - enviado UA", ua, sizeof(ua));  // DEBUG
        totalTramas++;
        totalTramasSU++;
        totalUA++;
        write(fd, ua, sizeof(ua));  // quando receber o DISC, rsponde com UA
    } else if (role == LlRx) {
        while (state != STOP_STATE) {
            // Enquanto o estado não for o final, processa os bytes da porta série (um de
cada vez)
            processByte(A, C_DISC, C_DISC, &aCheck, &cCheck, &state);  // espera um DISC
        }
        unsigned char disc[5] = {FLAG, A_CLOSE, C_DISC, A_CLOSE ^ C_DISC, FLAG};
        printLL("LLCLOSE - enviado DISC", disc, sizeof(disc));  // DEBUG
        totalTramas++;
        totalTramasSU++;
        totalDISC++;
        write(fd, disc, sizeof(disc));  // quando receber o DISC, responde com DISC
    } else {
        printf("Erro em connectionParameters.role\n");
        return -1;
    }

    close(fd);

    if (showStatistics) {
        clock_t end = clock();
        float seconds = (float)(end - start) / CLOCKS_PER_SEC;
        printf("\n---------- Estatísticas ----------\n");
        printf("\nTempo de Execução: %f segundos\n", seconds);
        printf("\nInvocações a llopen: %d\n", totalOpen);
        printf("Invocações a llwrite: %d\n", totalWrite);
        printf("Invocações a llread: %d\n", totalRead);
        printf("Invocações a llclose: %d\n", totalClose);
        printf("\nTramas Enviadas: %d\n", totalTramas);
        printf("\nTramas de Informação: %d\n", totalTramasI);
        printf("Tramas de Supervisão/Não Numeradas: %d\n", totalTramasSU);
        printf("\nTramas SET: %d\n", totalSET);
        printf("Tramas UA: %d\n", totalUA);
        printf("Tramas RR: %d\n", totalRR);
        printf("Tramas REJ: %d\n", totalREJ);
        printf("Tramas DISC: %d\n", totalDISC);
        printf("\nBytes Recebidos: %d\n", totalBytes);
        printf("\nBytes Stuffed/Destuffed: %d\n", totalStuffed);
        printf("FLAG Stuffed/Destuffed: %d\n", totalFlagStuffed);
        printf("ESC Stuffed/Destuffed: %d\n", totalEscStuffed);
        printf("\nAlarmes: %d\n", alarmCount);
```

```
        printf("Retransmissões: %d\n", totalRetransmissions);
        printf("Erros no BCC1: %d\n", totalBCC1);
        printf("Erros no BCC2: %d\n", totalBCC2);
        printf("Tramas Duplicadas: %d\n", totalDuplicados);
    }

    return 1;
}
```

application_layer.c

```
// Application layer protocol implementation

#include "application_layer.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "link_layer.h"

#define DATA_PACKET 1
#define CONTROL_PACKET_START 2
#define CONTROL_PACKET_END 3

#define CONTROL_PACKET_FILE_SIZE 0
#define CONTROL_PACKET_FILE_NAME 1

#define MAX_DATA_SIZE 256

// Imprime "Application Layer" seguido do título e do conteúdo
void printAL(char *title, unsigned char *content, int contentSize) {
    // DEBUG
    printf("\nApplication Layer\n");
    for (int i = 0; title[i] != '\0'; i++) printf("%c", title[i]);
    printf("\n");
    for (int i = 0; i < contentSize; i++) printf("0x%x ", content[i]);
    printf("\n");
}

// Calcula o logaritmo de base 2 de n
char logaritmo2(int n) {
    char res = -1;
    while (n > 0) {
        n /= 2;
        res++;
    }
    return res;
```

```
}

// Constrói e retorna um pacote de controlo de tipo (START/END) dado por 'controlField',
com o tamanho do ficheiro e o nome do ficheiro
unsigned char *buildControlPacket(unsigned char controlField, long int fileSize, const
char *fileName, int *packetSize) {
    unsigned char fileSizeLength = 1 + (logaritmo2(fileSize) / 8);  // número de bits
necessários para representar o tamanho do ficheiro
    unsigned char fileNameLength = strlen(fileName);               // comprimento do nome
do ficheiro

    *packetSize = 5 + fileSizeLength + fileNameLength;  // 5 -> C + T1 + L1 + T2 + L2
    unsigned char *controlPacket = (unsigned char *)malloc(*packetSize);

    controlPacket[0] = controlField;              // C
    controlPacket[1] = CONTROL_PACKET_FILE_SIZE;  // T1
    controlPacket[2] = fileSizeLength;            // L1

    int index;
    for (index = 3; index < (fileSizeLength + 3); index++) {
        // V1 - tamanho do ficheiro
        unsigned leftmost = fileSize & 0xFF << (fileSizeLength - 1) * 8;
        leftmost >>= (fileSizeLength - 1) * 8;
        fileSize <<= 8;
        controlPacket[index] = leftmost;
    }

    controlPacket[index++] = CONTROL_PACKET_FILE_NAME;        // T2
    controlPacket[index++] = fileNameLength;                  // L2
    memcpy(controlPacket + index, fileName, fileNameLength);  // V2 - nome do ficheiro

    printAL("Pacote de Controlo Construído", controlPacket, *packetSize);  // DEBUG

    return controlPacket;
}

// Constrói e retorna um pacote de dados
unsigned char *buildDataPacket(int dataSize, unsigned char *data, int *packetSize) {
    *packetSize = dataSize + 3;  // 3 -> C + L2 + L1
    unsigned char *dataPacket = (unsigned char *)malloc(*packetSize);

    dataPacket[0] = DATA_PACKET;     // C
    dataPacket[1] = dataSize / 256;  // L1
    dataPacket[2] = dataSize % 256;  // L2
    // dataSize = 256 * L2 + L1

    memcpy(dataPacket + 3, data, dataSize);  // dados

    printAL("Pacote de Dados Construído", dataPacket, *packetSize);  // DEBUG

    return dataPacket;
}
```

```c
// Envia um pacote de dados com 'size' dados do conteúdo do ficheiro
void sendDataPacket(int size, unsigned char *fileContent) {
    unsigned char *data;
    data = (unsigned char *)malloc(size);
    memcpy(data, fileContent, size);  // dados

    int dataPacketSize;
    unsigned char *dataPacket = buildDataPacket(size, data, &dataPacketSize);

    if (llwrite(dataPacket, dataPacketSize) < 0) {
        printf("Erro a enviar um pacote de dados com %d bytes\n", dataPacketSize);
        exit(-1);
    }

    free(data);
}

// Lê e interpreta um pacote de controlo, retirando o tamanho do ficheiro e retornando o
nome do ficheiro
char *parseControlPacket(unsigned char *packet, int *fileSize) {
    unsigned char fileSizeLength = packet[2];
    *fileSize = 0;
    for (unsigned char i = 3; i < (fileSizeLength + 3); i++) {
        *fileSize |= packet[i];
        *fileSize <<= 8;
    }
    *fileSize >>= 8;
    unsigned char fileNameLength = packet[fileSizeLength + 4];
    char *fileName = (char *)malloc(fileNameLength + 1);
    memcpy(fileName, packet + fileSizeLength + 5, fileNameLength);

    printAL("Pacote de Controlo Recebido", packet, fileSizeLength + fileNameLength +
5);  // DEBUG

    return fileName;
}

void applicationLayer(const char *serialPort, const char *role, int baudRate, int nTries,
int timeout, const char *filename) {
    LinkLayer connectionParameters;
    strcpy(connectionParameters.serialPort, serialPort);
    if (strcmp(role, "tx") == 0)  // role == "tx"
        connectionParameters.role = LlTx;
    else if (strcmp(role, "rx") == 0)  // role == "rx"
        connectionParameters.role = LlRx;
    else {
        printf("role = %s (deve ser tx ou rx)\n", role);
        exit(-1);
    }

    connectionParameters.baudRate = baudRate;
```

```
    connectionParameters.nRetransmissions = nTries;
    connectionParameters.timeout = timeout;

    if (llopen(connectionParameters) < 0) {
        printf("Erro a estabelecer a ligação\n");
        exit(-1);
    }

    if (connectionParameters.role == LlTx) {
        FILE *file = fopen(filename, "rb");
        if (file == NULL) {
            printf("Erro a abrir o ficheiro %s para ler\n", filename);
            exit(-1);
        }

        long int fileSize;
        struct stat st;
        if (stat(filename, &st) == 0) {
            fileSize = st.st_size;
            printf("O tamanho do ficheiro é %ld bytes\n", fileSize);  // DEBUG
        } else {
            printf("Erro a obter o tamanho do ficheiro\n");
            exit(-1);
        }

        // Construir e enviar pacote de controlo 'start'
        int startControlPacketSize;
        unsigned char *startControlPacket = buildControlPacket(CONTROL_PACKET_START,
fileSize, filename, &startControlPacketSize);
        if (llwrite(startControlPacket, startControlPacketSize) < 0) {
            printf("Erro a enviar pacote de controlo 'start'\n");
            exit(-1);
        }

        unsigned char *fileContent = (unsigned char *)malloc(fileSize * sizeof(unsigned
char));
        fread(fileContent, sizeof(unsigned char), fileSize, file);

        int completePackets = fileSize / MAX_DATA_SIZE;
        int incompletePacketSize = fileSize % MAX_DATA_SIZE;

        // Enviar pacotes de dados 'completos'
        for (int i = 0; i < completePackets; i++) {
            sendDataPacket(MAX_DATA_SIZE, fileContent);
            fileContent += MAX_DATA_SIZE;
        }

        // Enviar pacote de dados 'incompleto' (caso exista)
        if (incompletePacketSize != 0) {
            sendDataPacket(incompletePacketSize, fileContent);
        }
```

```
        // Construir e enviar pacote de controlo 'end'
        int endControlPacketSize;
        unsigned char *endControlPacket = buildControlPacket(CONTROL_PACKET_END, fileSize,
filename, &endControlPacketSize);
        if (llwrite(endControlPacket, endControlPacketSize) < 0) {
            printf("Erro a enviar pacote de controlo 'end'\n");
            exit(-1);
        }

        fclose(file);
    } else if (connectionParameters.role == LlRx) {
        FILE *newFile = fopen(filename, "wb");
        if (newFile == NULL) {
            printf("Erro a abrir o ficheiro %s para escrever\n", filename);
            exit(-1);
        }

        unsigned char *packet = (unsigned char *)malloc(MAX_DATA_SIZE);
        while (TRUE) {
            if (llread(packet) > 0) {
                if (packet[0] == CONTROL_PACKET_START) {
                    int fileSize;
                    char *newFileName = parseControlPacket(packet, &fileSize);
                    printf("Início da receção do ficheiro %s (%d bytes)\n", newFileName,
fileSize);
                } else if (packet[0] == DATA_PACKET) {
                    int dataSize = packet[1] * 256 + packet[2];
                    fwrite(packet + 3, sizeof(unsigned char), dataSize, newFile);

                    printAL("Pacote de Dados Recebido", packet, dataSize + 3);  // DEBUG
                } else if (packet[0] == CONTROL_PACKET_END) {
                    int fileSize;
                    char *newFileName = parseControlPacket(packet, &fileSize);
                    printf("Fim da receção do ficheiro %s (%d bytes)\n", newFileName,
fileSize);
                    break;
                }
            }
        }

        fclose(newFile);
        free(packet);
    }

    if (llclose(TRUE) < 0) {
        printf("Erro a concluir a ligação\n");
        exit(-1);
    }
}
```