

Redes de Computadores

*1º Trabalho Laboratorial*

*Licenciatura em Engenharia Informática e Computação*

*2023*

João Alves **up202108670**@fe.up.pt

Eduardo Sousa **up202103342**@fe.up.pt

Índice

|  |  |
| --- | --- |
| **Sumário** | 2 |
| **Introdução** | 2 |
| **Arquitetura** | 3 |
| **Estrutura do código** | 3 |
| **Casos de uso principais** | 5 |
| **Protocolo de ligação lógica** | 5 |
| **Protocolo de aplicação** | 7 |
| **Validação** | 9 |
| **Eficiência do protocolo de ligação de dados** | 9 |
| **Conclusões** | 11 |
| **Anexo** | 12 |
|  |  |

Sumário

Este projeto foi desenvolvido como parte da unidade curricular de Redes de Computadores e tem como foco a implementação de um protocolo de transferência de dados para a comunicação de arquivos entre computadores por meio da porta série.

Por meio deste trabalho prático, pudemos fortalecer a nossa compreensão dos conceitos abordados nas aulas práticas, permitindo-nos implementar o protocolo necessário e aprimorar as nossas habilidades em relação às estratégias empregadas por ele.

Introdução

O objetivo primordial deste trabalho prático consistiu na implementação e avaliação de um protocolo de transmissão de dados, conforme as diretrizes estabelecidas no roteiro das aulas práticas. O presente relatório tem por finalidade esclarecer a estrutura adotada neste projeto, abrangendo os seguintes tópicos:

1. **Arquitetura:** Descrição das interfaces e blocos funcionais envolvidos.
2. **Estrutura do Código:** Apresentação das principais funções, APIs e estruturas de dados empregadas.
3. **Principais Casos de Uso:** Identificação dos cenários de aplicação e demonstração das sequências de chamadas de funções associadas.
4. **Protocolo de Ligação Lógica:** Explicação sobre o funcionamento da camada de ligação lógica e a estratégia de implementação utilizada.
5. **Protocolo de Aplicação:** Detalhes do funcionamento da camada de aplicação e a estratégia de implementação correspondente.
6. **Validação:** Testes realizados para avaliar a eficácia da implementação do protocolo.
7. **Eficiência do Protocolo de Ligação de Dados:** Apresentação de estatísticas relativas à eficiência do protocolo implementado na camada de ligação de dados.
8. **Conclusão:** Síntese das informações fornecidas nas seções anteriores e reflexão sobre os objetivos de aprendizagem atingidos.

Arquitetura

Este projeto foi desenvolvido com base no código inicialmente fornecido pelos docentes e disponibilizado no Moodle. Esse código base segue uma estrutura dividida em duas camadas principais para a implementação do protocolo: a ApplicationLayer e a LinkLayer.

A ApplicationLayer, representada nos ficheiros application\_layer.h e application\_layer.c, desempenha o papel de camada de aplicação. Ela utiliza a API do LinkLayer para transmitir e receber pacotes de dados a partir de um arquivo. Esta camada é a mais visível para o utilizador e permite a configuração de parâmetros como o número máximo de tentativas de retransmissão e a velocidade de transferência.

A LinkLayer, encontrada nos ficheiros link\_layer.h e link\_layer.c, corresponde à camada de ligação de dados. Ela é encarregue do estabelecimento e término da conexão, bem como da criação e envio de tramas, além de validar as tramas recebidas.

O projeto é organizado em dois blocos funcionais, concentrando-se nas funções do emissor e do recetor. Cada um desses blocos engloba as camadas de ligação de dados e de aplicação de forma independente. Essa divisão foi adotada com o propósito de isolar e abordar distintos cenários de uso.

Estrutura de código

**Camada de Aplicação (ApplicationLayer):**

Para a Camada de Aplicação, foram implementadas as seguintes funções:

1. **void applicationLayer(const char \*serialPort, const char \*role, int baudRate, int nTries, int timeout, const char \*filename):** Esta é a função principal deste bloco e é responsável por orquestrar todo o processo.
2. **int transmitData(const char \*filename):** Esta função é responsável por preparar os pacotes de dados para a transmissão e solicitar à Camada de Ligação (LinkLayer) que lide com a transmissão real.
3. **int receiveData(const char \*filename):** Esta função lida com a receção de pacotes de dados enviados através da Camada de Ligação e reconstrói o ficheiro transmitido.

**Camada de Ligação (LinkLayer):**

Na Camada de Ligação, as estruturas de dados implementadas incluem LinkLayerRole, utilizada para identificar se o computador é um emissor (transmitter) ou recetor (receiver), e LinkLayer, que define os parâmetros associados à transferência de dados:

typedef enum {

LlTx,

LlRx

} LinkLayerRole;

typedef struct {

char serialPort[50];

LinkLayerRole role;

int baudRate;

int nRetransmissions;

int timeout;

} LinkLayer;

As funções implementadas no bloco da Camada de Ligação (LinkLayer) são as seguintes

1. **int state\_machine(unsigned char a, unsigned char c, int isData, int RR\_REJ):** Esta função gere a máquina de estados e altera o estado atual do programa com base nas informações fornecidas.
2. **int send\_buffer(unsigned char a, unsigned char c):** Responsável pelo envio de tramas de supervisão.
3. **int llopen(LinkLayer connectionParameters):** Estabelece a ligação com a porta série.
4. **int llwrite(const unsigned char \*buf, int bufSize):** Lida com o envio de tramas.
5. **int llread(unsigned char \*buffer):** Gerencia a leitura de tramas.
6. **int llclose(int showStatistics):** Trata do encerramento da ligação.
7. **void alarmHandler(int signal):** Cria e gere alarmes para o programa.
8. **int destuff(unsigned char \*destuffed, const unsigned char \*trama, int len):** Aplica o mecanismo de destuffing às tramas.
9. **int stuff(unsigned char \*stuffed, const unsigned char \*trama, int len):** Aplica o mecanismo de stuffing às tramas.

Casos de uso principais

Como mencionado anteriormente, o programa pode ser executado em dois modos: transmissor e recetor. Dependendo do modo de execução, a sequência de chamadas de funções é significativamente diferente devido à utilização de códigos distintos.

**Modo Transmissor:**

1. **llopen:** Esta função é responsável pelo handshake inicial entre o transmissor e o recetor, envolvendo a troca de pacotes de controlo.
2. **transmitData:** Nesta etapa, o programa abre o ficheiro a ser enviado e divide-o em pacotes para posterior transmissão.
3. **llwrite:** Aqui, são criadas e enviadas tramas de informação pela porta série, com base no pacote fornecido como argumento. Além disso, o estado atual do programa é modificado através da máquina de estados.
4. **llclose:** Utilizada para encerrar a ligação entre o recetor e o transmissor por meio da troca de pacotes de controlo.

**Modo Recetor:**

1. **receiveData:** Nesta fase, o programa é responsável por criar o ficheiro de destino, que servirá como cópia do ficheiro enviado. Esse processo envolve a construção gradual do ficheiro à medida que os pacotes chegam.
2. **llread:** A função llread é encarregue de receber as tramas enviadas, verificar se foram recebidas corretamente e alterar o estado da máquina de estados.
3. **send\_buffer:** Essa função é usada para escrever o resultado obtido na leitura das tramas no ficheiro de destino.

Protocolo de ligação lógica

A camada de ligação de dados desempenha um papel crucial na comunicação direta com a porta série, sendo encarregue de gerir a interação entre o emissor e o recetor. Para essa finalidade, implementamos o protocolo Stop and Wait, que é empregue para estabelecer e encerrar a conexão, bem como para transmitir tramas de supervisão e informações.

A criação da conexão é executada por meio da função llopen. Inicialmente, após a abertura e configuração da porta série, o emissor envia uma trama de supervisão de "SET" e aguarda a resposta do recetor na forma de uma trama de supervisão "UA." Se o recetor responder com a trama "UA," isso indica que a conexão foi estabelecida com sucesso. A partir desse momento, o emissor inicia a transmissão de informações destinadas a serem lidas pelo recetor.

A função llwrite é responsável pelo envio de informações. Ela processa um pacote de controle ou dados, aplicando a estratégia de stuffing para evitar conflitos com bytes de dados que possam ser interpretados erradamente como flags da trama. Em seguida, transforma esse pacote em uma trama de informação, através do mecanismo de framing, que é enviada ao recetor. A função aguarda uma resposta e, caso a trama seja rejeitada, repete o processo de envio até que a trama seja aceita ou o número máximo de tentativas seja excedido. Cada tentativa de envio possui um limite de tempo, após o qual ocorre um timeout.

A função llread é responsável pela leitura de informações recebidas pela porta série, validando a integridade dos dados. Ela executa inicialmente o processo de destuffing do campo de dados da trama e verifica os valore do BCC2 para identificar erros que possam ter ocorrido durante a transmissão.

A finalização da conexão é realizada por meio da função llclose. Essa função é invocada pelo emissor quando ocorre um número excessivo de tentativas mal sucedidas ou quando a transferência dos pacotes de dados é concluída. O emissor envia uma trama de supervisão "DISC" e aguarda que o recetor responda com a mesma trama, marcando o encerramento da comunicação. Quando o emissor recebe novamente a trama "DISC," ele responde com "UA" e encerra a conexão.

Protocolo de aplicação

A camada de aplicação desempenha um papel fundamental na interação direta com o ficheiro a ser transferido, bem como com o utilizador. Nessa camada, é possível especificar vários parâmetros, como o ficheiro a ser transferido, a porta série a ser utilizada, a taxa de transferência, o tamanho dos pacotes em bytes que compõem o ficheiro, o número máximo de tentativas de retransmissão e o tempo máximo de espera por uma resposta do recetor. A transferência do ficheiro ocorre por meio da utilização da API da LinkLayer, que converte os pacotes de dados em tramas de informação.

Após a conclusão do processo de handshake entre o transmissor e o recetor, o conteúdo integral do ficheiro é copiado para um buffer local usando a função transmitData e é fragmentado pela camada de aplicação, de acordo com o tamanho de bytes especificado no argumento. O primeiro pacote enviado pelo transmissor contém dados formatados como TLV (Tipo, Tamanho, Valor), sendo criado pela função transmitData. Esse pacote inclui informações sobre o tamanho do ficheiro em bytes e o seu nome. No lado do recetor, esse pacote é descompactado pela função receiveData, alocando assim o espaço necessário para receber o ficheiro.

Cada fragmento do ficheiro a ser transferido é encapsulado num pacote de dados através da função transmitData e é enviado pela porta série usando a função llwrite da API da LinkLayer. Após cada envio, o recetor responde, indicando se aceitou ou rejeitou o pacote enviado. No primeiro caso, o transmissor prossegue com o envio do fragmento seguinte; no segundo caso, o mesmo fragmento é reenviado. Cada pacote é avaliado individualmente pelo recetor, usando a função llread e a função receiveData, que extrai o fragmento original do ficheiro quando este é recebido com sucesso.

A ligação entre as duas máquinas é encerrada quando a função llclose da API é invocada, seja após a conclusão da transferência de pacotes de dados ou se o número máximo de tentativas sem sucesso for atingido.

Validação

Para avaliar o desempenho da aplicação desenvolvida, realizamos os seguintes testes:

* Envio de arquivos de diversos tamanhos e formatos.
* Simulação de curto-circuito durante a transmissão de um arquivo.
* Interrupção temporária da conexão durante a transmissão de um arquivo.
* Envio de um arquivo com variação no tamanho dos pacotes.
* Envio de um arquivo com variação nas capacidades da conexão (baudrate).

Quase todos os testes foram concluídos com sucesso, excluindo o teste do curto-circuito e em algumas situações o teste de interrupção da conexão.

Eficiência do protocolo de ligação de dados

Os dados apresentados em seguida foram obtidos transferindo o penguin.gif, que tem 10968 bytes.

**Variação Baudrate**

Com um tamanho de trama de 256 bytes, obtivemos os seguintes dados variando o *baudrate*:

**Tabela**

**Graficos**

O tempo total da transferência do ficheiro e a *baudrate* devem ser inversamente proporcionais, tal como se observou. Assim, a eficiência do protocolo também diminui quando a *baudrate* aumenta.

**Variação Tamanho da Trama**

Com um *baudrate* de 9600, obtivemos os seguintes dados variando o tamanho da trama:

**Tabela**

**Graficos**

Quanto maior o tamanho da trama, menos trasmissões são efetuadas, de maneira que o tempo total do protocolo diminui. A eficiência do protocolo é maior quanto maior a trama.

**Variação de Taxa de Erros**

Com um tamanho de trama de 256 bytes e uma *baudrate* de 9600, obtivemos os seguintes dados variando a taxa de erros:

**Tabela**

**Graficos**

Como seria de esperar, à medida que a taxa de erros aumenta, o tempo necessário para a transferência também aumenta, e a eficiência do protocolo diminui.

Conclusões

Este trabalho aborda o protocolo de ligação de dados, que garante uma comunicação confiável entre dois sistemas conectados por uma porta série. Além disso, enfatiza a independência entre camadas, com os blocos funcionais ApplicationLayer e LinkLayer demonstrando essa independência. A camada de ligação de dados não processa cabeçalhos de pacotes, enquanto a camada de aplicação não requer conhecimento detalhado do protocolo de ligação. O trabalho atingiu quase todos os objetivos com sucesso, contribuindo para uma compreensão mais aprofundada do assunto, tanto teoricamente quanto na prática.

Anexos

1. **link\_layer.h**

#ifndef \_LINK\_LAYER\_H\_

#define \_LINK\_LAYER\_H\_

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

#include <time.h>

typedef enum{ LlTx, LlRx } LinkLayerRole;

typedef struct

{

char serialPort[50];

LinkLayerRole role;

int baudRate;

int nRetransmissions;

int timeout;

} LinkLayer;

#define \_POSIX\_SOURCE 1

#define MAX\_SIZE 256

#define FALSE 0

#define TRUE 1

#define FLAG 0x7E

#define A\_TRANSMITER 0x03

#define A\_RECEIVER 0x01

#define C\_SET 0x03

#define C\_UA 0x07

#define BCC(a, c) (a^c)

#define DISC 0x0B

#define ESCAPE 0x7D

#define RR(n) (0x05 | (n << 7))

#define REJ(n) (0x01 | (n << 7))

#define C\_INF(n) (0x00 | (n << 6))

#define RR\_R 0

#define START 1

#define FLAG\_RCV 2

#define A\_RCV 3

#define C\_RCV 4

#define BCC\_OK 5

#define WAITING 6

#define RR0 0

#define RR1 1

#define REJ0 2

#define REJ1 3

#define OTHER 4

#define MIDDLE\_PACKET 1

#define STARTING\_PACKET 2

#define ENDING\_PACKET 3

int state\_machine(unsigned char a, unsigned char c, int isData, int RR\_REJ);

int send\_buffer(unsigned char a, unsigned char c);

int llopen(LinkLayer connectionParameters);

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

int llread(unsigned char \* buffer);

int llclose(int showStatistics);

#endif // \_LINK\_LAYER\_H\_

1. **link\_layer.c**

#include "link\_layer.h"

#define \_POSIX\_SOURCE 1

struct termios oldtio;

struct termios newtio;

int alarmEnabled = FALSE;

int alarmCount = 0;

volatile int STOP = FALSE;

int current\_state = START;

int fd, bytes = 0;

unsigned char readbyte;

int answer = -1;

LinkLayer link\_layer;

int state\_machine(unsigned char a, unsigned char c, int isData, int RR\_REJ) {

if (!RR\_REJ){ answer = -1; }

unsigned char byte = 0;

int bytes = 0;

bytes = read(fd, &byte, 1);

if (bytes > 0) {

if (current\_state == START) {

if (byte == FLAG)

current\_state = FLAG\_RCV;

}

else if (current\_state == FLAG\_RCV) {

if (byte == a)

current\_state = A\_RCV;

else if (byte != FLAG)

current\_state = START;

} else if (current\_state == A\_RCV) {

if (RR\_REJ) {

if (byte == (RR(0))) {

answer = RR0;

current\_state = C\_RCV;

} else if (byte == (RR(1))) {

answer = RR1;

current\_state = C\_RCV;

} else if (byte == (REJ(0))) {

answer = REJ0;

current\_state = C\_RCV;

} else if (byte == (REJ(1))) {

answer = REJ1;

current\_state = C\_RCV;

} else {

current\_state = START;

}

}

else {

if (byte == c)

current\_state = C\_RCV;

else if (byte == FLAG)

current\_state = FLAG\_RCV;

else

current\_state = START;

}

}

else if (current\_state == C\_RCV) {

if (answer == RR0)

c = RR(0);

else if (answer == RR1)

c = RR(1);

else if (answer == REJ0)

c = REJ(0);

else if (answer == REJ1)

c = REJ(1);

if (byte == (a ^ c)) {

if (isData)

current\_state = WAITING;

else

current\_state = BCC\_OK;

}

else if (byte == FLAG)

current\_state = FLAG\_RCV;

else

current\_state = START;

}

else if (current\_state == WAITING) {

if (byte == FLAG) {

STOP = TRUE;

}

}

else if (current\_state == BCC\_OK) {

if (byte == FLAG) {

STOP = TRUE;

current\_state = START;

if (c == C\_UA)

alarm(0);

}

else {

current\_state = START;

}

}

readbyte = byte;

return TRUE;

}

return FALSE;

}

int send\_buffer(unsigned char a, unsigned char c) {

unsigned char buffer[5];

buffer[0] = FLAG;

buffer[1] = a;

buffer[2] = c;

buffer[3] = buffer[1] ^ buffer[2];

buffer[4] = FLAG;

return write(fd, buffer, sizeof(buffer));

}

void alarmHandler(int signal) {

alarmEnabled = FALSE;

alarmCount++;

}

////////////////////////////////////////////////

// LLOPEN

////////////////////////////////////////////////

int llopen(LinkLayer connectionParameters) {

link\_layer = connectionParameters;

(void)signal(SIGALRM, alarmHandler);

fd = open(connectionParameters.serialPort, O\_RDWR | O\_NOCTTY);

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] = 0;

newtio.c\_cc[VMIN] = 0;

tcflush(fd, TCIOFLUSH);

if (tcsetattr(fd, TCSANOW, &newtio) == -1){

perror("tcsetattr");

exit(-1);

}

sleep(1);

STOP = FALSE;

if (link\_layer.role == LlTx) {

while (STOP == FALSE && alarmCount < link\_layer.nRetransmissions) {

if (alarmEnabled == FALSE) {

bytes = send\_buffer(A\_TRANSMITER, C\_SET);

alarm(link\_layer.timeout);

alarmEnabled = TRUE;

current\_state = START;

}

state\_machine(A\_TRANSMITER, C\_UA, 0, 0);

}

alarm(0);

if (alarmCount >= link\_layer.nRetransmissions) {

printf("ERROR: TIMEOUT\n");

return -1;

}

printf("Connection established successfully\n");

}

else if (connectionParameters.role == LlRx) {

while (STOP == FALSE) {

state\_machine(A\_TRANSMITER, C\_SET, 0, 0);

}

bytes = send\_buffer(A\_TRANSMITER, C\_UA);

}

return 0;

}

////////////////////////////////////////////////

// LLWRITE

////////////////////////////////////////////////

int stuff(unsigned char \*stuffed, const unsigned char \*trama, int len) {

int size = 0;

stuffed[size++] = trama[0];

for (int i = 1; i < len; i++) {

if (trama[i] == FLAG || trama[i] == ESCAPE) {

stuffed[size++] = ESCAPE;

stuffed[size++] = trama[i]^0x20;

}

else

stuffed[size++] = trama[i];

}

return size;

}

int destuff(unsigned char \*destuffed, const unsigned char \*trama, int len) {

int size = 0;

destuffed[size++] = trama[0];

for (int i = 1; i < len; i++) {

if (trama[i] == ESCAPE) {

destuffed[size++] = trama[i+1]^0x20;

i++;

}

else

destuffed[size++] = trama[i];

}

return size;

}

int llwrite(const unsigned char \*buf, int bufSize) {

int size = bufSize + 5;

unsigned char trama[size];

static int pacote = 0;

int tries = 1;

int rej = FALSE;

// apply header

trama[0] = FLAG;

trama[1] = A\_TRANSMITER;

trama[2] = C\_INF(pacote);

trama[3] = BCC(A\_TRANSMITER, C\_INF(pacote));

// content

unsigned char bcc2 = buf[0];

for (int i = 0; i < bufSize; i++) {

trama[i + 4] = buf[i];

if (i > 0){ bcc2 ^= buf[i]; }

}

// apply tail

trama[bufSize + 4] = bcc2;

unsigned char stuffed[size\*2];

size = stuff(stuffed, trama, size);

stuffed[size] = FLAG;

size++;

STOP = FALSE;

alarmEnabled = FALSE;

alarmCount = 0;

current\_state = START;

while (!STOP && alarmCount < link\_layer.nRetransmissions) {

if (!alarmEnabled) {

bytes = write(fd, stuffed, size);

printf("> %d Written bytes at %d try\n", bytes, tries);

alarm(link\_layer.timeout);

alarmEnabled = TRUE;

current\_state = START;

tries++;

}

state\_machine(A\_TRANSMITER, 0, 0, 1);

if ((pacote == 0 && answer == REJ1) || (pacote == 1 && answer == REJ0)) { rej = TRUE; }

if (rej == TRUE) { alarm(0); alarmEnabled = FALSE; }

}

pacote = (pacote + 1) % 2;

alarm(0);

printf("> Packet arrived nicely\n\n");

return 0;

}

////////////////////////////////////////////////

// LLREAD

////////////////////////////////////////////////

int llread(unsigned char \*buffer) {

int readed = 0, size;

static int packet = 0;

STOP = FALSE;

current\_state = START;

unsigned char stuffed[(MAX\_SIZE \* 2) + 7];

unsigned char destuffed[MAX\_SIZE + 7];

while (STOP == FALSE){

if (state\_machine(A\_TRANSMITER, C\_INF(packet), 1, 0)) {

stuffed[readed] = readbyte;

readed++;

}

}

size = destuff(destuffed, stuffed, readed);

unsigned char original\_bcc2 = destuffed[size - 2];

unsigned char cmp\_bcc2 = 0x00;

for (unsigned int i = 4; i < (size-2); i++){ cmp\_bcc2 ^= destuffed[i]; }

if (destuffed[2] == C\_INF(packet) && original\_bcc2 == cmp\_bcc2) {

packet = (packet + 1) % 2;

send\_buffer(A\_TRANSMITER, RR(packet));

memcpy(buffer, &destuffed[4], size - 5);

printf("> Correct packet: \n > bcc2 -> %x \n > message -> %x\n", cmp\_bcc2, destuffed[2]);

size -= 5;

return size;

}

else if (cmp\_bcc2 == original\_bcc2) {

send\_buffer(A\_TRANSMITER, RR(packet));

printf("> Received a duplicated packet, rejection ongoing\n");

}

else {

send\_buffer(A\_TRANSMITER, REJ(packet));

current\_state = START;

printf("> Bcc2 didnt match, rejection ongoing\n");

}

return -1;

}

////////////////////////////////////////////////

// LLCLOSE

////////////////////////////////////////////////

int llclose(int showStatistics) {

STOP = FALSE;

alarmEnabled = FALSE;

alarmCount = 0;

current\_state = START;

answer = OTHER;

if (link\_layer.role == LlTx) {

while (STOP == FALSE && alarmCount < link\_layer.nRetransmissions) {

if (alarmEnabled == FALSE) {

bytes = send\_buffer(A\_TRANSMITER, DISC);

alarm(link\_layer.timeout);

alarmEnabled = TRUE;

}

state\_machine(A\_RECEIVER, DISC, 0, 0);

}

if (alarmCount == link\_layer.nRetransmissions) {

return -1;

}

bytes = send\_buffer(A\_RECEIVER, C\_UA);

} else if (link\_layer.role == LlRx) {

while (STOP == FALSE) {

state\_machine(A\_TRANSMITER, DISC, 0, 0);

}

bytes = send\_buffer(A\_RECEIVER, DISC);

STOP = FALSE;

current\_state = START;

while (STOP == FALSE) {

state\_machine(A\_RECEIVER, C\_UA, 0, 0);

}

}

if (tcsetattr(fd, TCSANOW, &oldtio) == -1) {

perror("tcsetattr");

exit(-1);

}

printf("> Everything went well and the file was transfered!\n");

close(fd);

return 0;

}

1. **application\_layer.h**

#ifndef \_LINK\_LAYER\_H\_

#define \_LINK\_LAYER\_H\_

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <termios.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

#include <time.h>

typedef enum{ LlTx, LlRx } LinkLayerRole;

typedef struct

{

char serialPort[50];

LinkLayerRole role;

int baudRate;

int nRetransmissions;

int timeout;

} LinkLayer;

#define \_POSIX\_SOURCE 1

#define MAX\_SIZE 256

#define FALSE 0

#define TRUE 1

#define FLAG 0x7E

#define A\_TRANSMITER 0x03

#define A\_RECEIVER 0x01

#define C\_SET 0x03

#define C\_UA 0x07

#define BCC(a, c) (a^c)

#define DISC 0x0B

#define ESCAPE 0x7D

#define RR(n) (0x05 | (n << 7))

#define REJ(n) (0x01 | (n << 7))

#define C\_INF(n) (0x00 | (n << 6))

#define RR\_R 0

#define START 1

#define FLAG\_RCV 2

#define A\_RCV 3

#define C\_RCV 4

#define BCC\_OK 5

#define WAITING 6

#define RR0 0

#define RR1 1

#define REJ0 2

#define REJ1 3

#define OTHER 4

#define MIDDLE\_PACKET 1

#define STARTING\_PACKET 2

#define ENDING\_PACKET 3

int state\_machine(unsigned char a, unsigned char c, int isData, int RR\_REJ);

int send\_buffer(unsigned char a, unsigned char c);

int llopen(LinkLayer connectionParameters);

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

int llread(unsigned char \* buffer);

int llclose(int showStatistics);

#endif // \_LINK\_LAYER\_H\_

1. **application\_layer.c**

// Application layer protocol implementation

#include "application\_layer.h"

#include <signal.h>

#include <stdlib.h>

struct stat file\_stat;

clock\_t start\_t, end\_t;

double total\_t;

int transmitData(const char \*filename) {

start\_t = clock();

if (stat(filename, &file\_stat) < 0) {

perror("Error getting file information.");

return -1;

}

FILE \*filePointer = fopen(filename, "rb");

unsigned fileSizeLength = sizeof(file\_stat.st\_size);

unsigned filenameLength = strlen(filename);

unsigned packetSize = 5 + fileSizeLength + filenameLength;

unsigned char packet[packetSize];

packet[0] = STARTING\_PACKET;

packet[1] = 0;

packet[2] = fileSizeLength;

memcpy(&packet[3], &file\_stat.st\_size, fileSizeLength);

packet[3 + fileSizeLength] = 1;

packet[4 + fileSizeLength] = filenameLength;

memcpy(&packet[5 + fileSizeLength], filename, filenameLength);

printf("> Sending start packet\n");

if (llwrite(packet, packetSize) < 0)

return -1;

unsigned char buf[MAX\_SIZE];

unsigned bytes\_to\_send;

unsigned sequenceNumber = 0;

while ((bytes\_to\_send = fread(buf, sizeof(unsigned char), MAX\_SIZE - 4, filePointer)) > 0) {

printf("> Sending middle packet\n");

unsigned char dataPacket[MAX\_SIZE];

dataPacket[0] = MIDDLE\_PACKET;

dataPacket[1] = sequenceNumber % 255;

dataPacket[2] = (bytes\_to\_send / 256);

dataPacket[3] = (bytes\_to\_send % 256);

memcpy(&dataPacket[4], buf, bytes\_to\_send);

if (llwrite(dataPacket, ((bytes\_to\_send + 4) < MAX\_SIZE) ? (bytes\_to\_send + 4) : MAX\_SIZE) == -1) {

break;

}

printf("> Another packet sent\n");

sequenceNumber++;

}

fileSizeLength = sizeof(file\_stat.st\_size);

filenameLength = strlen(filename);

packetSize = 5 + fileSizeLength + filenameLength;

packet[0] = ENDING\_PACKET;

packet[1] = 0;

packet[2] = fileSizeLength;

memcpy(&packet[3], &file\_stat.st\_size, fileSizeLength);

packet[3 + fileSizeLength] = 1;

packet[4 + fileSizeLength] = filenameLength;

memcpy(&packet[5 + fileSizeLength], filename, filenameLength);

printf("> Sending end packet\n");

if (llwrite(packet, packetSize) < 0)

return -1;

fclose(filePointer);

end\_t = clock();

total\_t = (double)(end\_t - start\_t) / CLOCKS\_PER\_SEC;

printf("\nTotal time taken: %f seconds\n", total\_t);

printf("Size transferred: %d bytes\n", (int)file\_stat.st\_size);

printf("Transfer Speed: %f B/s\n\n", file\_stat.st\_size / total\_t);

return 0;

}

int receiveData(const char \*filename) {

int size;

static FILE \*dest;

int STOP = FALSE;

while (!STOP) {

unsigned char buf[MAX\_SIZE];

if ((size = llread(buf)) < 0){

continue;

}

if(buf[0] == 2) {

printf("\n> Receiving start packet\n");

dest = fopen(filename, "wb");

}

else if(buf[0] == 1) {

printf("\n> Receiving middle packet\n");

static unsigned int buf\_position = 0;

if (buf[1] != buf\_position){ return -1; }

buf\_position = (buf\_position + 1) % 255;

unsigned int data\_size = buf[2] \* 256 + buf[3];

fwrite(&buf[4], sizeof(unsigned char), data\_size \* sizeof(unsigned char), dest);

}

else if (buf[0] == 3) {

printf("\n> Receiving end packet\n");

fclose(dest);

STOP = TRUE;

}

}

return 0;

}

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

LinkLayer link\_layer;

link\_layer.baudRate = baudRate;

link\_layer.nRetransmissions = nTries;

link\_layer.timeout = timeout;

if (strcmp(role, "tx") == 0)

link\_layer.role = LlTx;

else if (strcmp(role, "rx") == 0)

link\_layer.role = LlRx;

else {

fprintf(stderr, "Unknown role: %s\n", role);

return;

}

sprintf(link\_layer.serialPort, "%s", serialPort);

if (llopen(link\_layer) < 0) {

exit(-1);

return;

}

if (link\_layer.role == LlTx) {

transmitData(filename);

}

else if (link\_layer.role == LlRx) {

receiveData(filename);

}

else {

fprintf(stderr, "Unknown role: %s\n", role);

return;

}

llclose(FALSE);

}