Redes de Computadores

Protocolo de Ligação de Dados

Ângela Cardoso e Bruno Madeira

4 de Novembro de 2015

Sumário

TODO: Parágrafo sobre contexto.
TODO: Parágrafo sobre conclusões.

Conteúdo

Capítulo 1

Introdução

TODO: Indicação dos objectivos do trabalho e do relatório; descrição da lógica do relatório com indicações sobre o tipo de informação que poderá ser encontrada em cada uma secções seguintes.

Relatório relativo ao primeiro trabalho prático de Redes de Computadores que consiste na implementação de uma aplicação que transfere imagens entre dois computadores fazendo uso da porta-série. A aplicação deve usar um protocolo de ligação de dados *Stop N Wait ARQ* híbrido que deve assegurar a fiabilidade da transmissão mesmo em caso de desconexão. Deve também usar um protocolo de aplicação que é responsável pelo envio da imagem. O código desenvolvido deve ser estruturado em camadas, respeitando o princípio de encapsulamento, de modo a assegurar cada protocolo funciona de forma independente.

O trabalho foi utilizando a linguagem de programação C num ambiente com um sistema operativo baseado em Linux. Para testes foi usada uma porta-seire XPTO???.

Este relatório visa reportar qual o estado final da aplicação desenvolvia, clarificar detalhes do processo de implementação/código e a opinião dos estudantes face ao projecto realizado.

Do capítulo 2 ao 4 são expostas as estruturas e os mecanismos implementados na concepção da aplicação sem incidir em detalhes específicos destes.

Os capítulos 5 e 6 incidem sobre as particularidades da implementação dos protocolos usados.

O capítulo 7 apresenta os testes efectuados sobre a aplicação.

O

Capítulo 2

Arquitetura

TODO:blocos funcionais e interfaces

Capítulo 3

Estrutura do código

TODO: APIs, principais estruturas de dados, principais funções e sua relação com a arquitetura

Capítulo 4

Casos de uso principais

TODO: identificação; sequências de chamada de funções

Capítulo 5

Protocolo de ligação lógica

Para implementar o protocolo de ligação lógica, seguimos as indicações do enunciado do projeto. Sendo assim, usamos a variante *Stop and Wait*, o que significa que o Emissor, após cada mensagem, aguarda uma resposta do Recetor antes de enviar a mensagem seguinte. Isto significa, entre outras coisas, que podemos utilizar uma numeração módulo 2 para as mensagens, dado que nunca temos mais do que 2 mensagens em jogo (aquela que foi enviada e a que se pretende enviar de seguida).

O interface deste protocolo tem 5 funções:

- `int llopen(app_status_type status)` - abre a porta de série para comunicação, o parâmetro indica se é o Emissor ou o Receptor quem chama a função, retorna o número da ligação de dados, ou -1 em caso de erro;
- `int llclose(int fd, int hard)` - termina a ligação no final da comunicação, o parâmetro fd é o número da ligação, o parâmetro hard indica que estamos numa situação de terminar à força após um erro, retorna 0 ou -1 consoante seja bem sucedida ou ocorra erro;
- `int llwrite(int fd, char *buffer, int length)` - dado um número de ligação e um array de dados com determinado comprimento, constrói a trama de acordo com as regras do protocolo e envia essa trama, retorna 0 ou -1 consoante seja bem sucedida ou ocorra erro;
- `int llread(int fd, char **buffer)` - dado um número de ligação e um apontador para um array de dados, lê os dados da porta e coloca-os no array, retorna 0 ou -1 consoante seja bem sucedida ou ocorra erro;
- `void set_basic_definitions(int nTries, char* port, int baudrate, int packetSize)` - consoante as preferências do utilizador, determina o número de tentativas em caso de timeout, a porta, o baudrate e o tamanho dos dados nos pacotes de dados, guardando estas informações na estrutura `datalink`;

Estas funções utilizam a função `update_state_machine` para determinar a cada instante o estado em que está. Ao chamar esta função são definidos os estados finais, e só depois disso, quando um estado final é atingido, é feito o reset à máquina de estados.

Para obter a trama que pretende enviar é utilizada uma função `getMessage` que tem como parâmetros o Emissor original, o tipo e o número da trama e o array de caracteres onde será colocada a flag e o cabeçalho da trama. Posteriormente a estes bytes, em tramas de Informação, serão adicionados os dados e o respetivo BCC2. Nas restantes tramas, é apenas reutilizada a flag que está na primeira posição do array. A sua especificação pode ser encontrada no Anexo B.

TODO:

ligação de dados assíncrona vmin=0 e ???min=0 -> read retorna de imediato -> e usado um ciclo -> blablabla

paridade usada foi XPTO, e possivel alterala no codigo

alarm

Capítulo 6

Protocolo de aplicação

TODO: identificação dos principais aspectos funcionais; descrição da estratégia de implementação destes aspectos com apresentação de extractos de código

foi implementada de acordo com o enunciado.

a aplicação guarda o tamanho de imagem e o numero de pacotes ja enviados/recebidos

envia Start recebe Start

envia info pode enviar pacotes de tamanho maximo definido pelo utilizador recebe info

envia end recebe end

Capítulo 7

Validação

TODO: descrição dos testes efectuados com apresentação quantificada dos resultados, se possível
foram realizados testes nas aulas práticas e fora destas usando uma porta-série.
foram realizados testes nas máquinas virtuais usadas para desenvolver a aplicação.
??? +

Capítulo 8

Elementos de valorização

A camada de ligação de dados regista o número de tramas do tipo I e REJs recebidas/enviadas e o número de ocorrencias de timeouts de uma transmissão. A informação fica registada numa variável do tipo *occurrences_Log* e pode ser acedida pela componente de interface através do método *get_occurrences_log*.

O utilizador pode definir qual o *baudrate* a usar, o *packetSize* (numero de bytes máximos que envia da imagem em cada trama I antes de se realizar o stuffing) e o número máximo de tentativas consecutivas de reconexão. Uma vez escolhidas as opções o intervalo de duração de um timeout é calculado em função do *baudrate* e do *packetSize* escolhidos pelo utilizador como se pode verificar no anexo A.1. Na nossa implementação o receptor pode e deve configurar o *baudrate* e *packetSize* iguais aos do transmissor, pois, apesar de estes não serem usados na prática pelo receptor, são necessários para calcular um intervalo de timeout que seja fiável face às definições do transmissor.

São gerados erros aleatoriamente nas tramas enviadas pelo transmissor. São gerados erros no cabeçalho e de nos dados. Estes ocorrem de forma independente e são gerados respectivamente pelas funções *randomErrorCabe* e *randomErrorData*. A frequência destes erros é defina a partir das constantes *CABE_PROB* e *DATA_PROB* que representam quantas tramas são enviadas por cada um que apresenta um erro do tipo correspondente. Pode ser verificada a função *randomErrorData* no anexo A.2. A *randomErrorCabe* não foi incluida em anexo uma vez que é bastante semelhante à *randomErrorData*.

Na AppProtocol.c, quando a aplicação está enviar/receber uma imagem, trata imprimir na consola a quantidade de bytes enviados/recebidos e quantos faltam para receber/enviar a imagem na totalidade. Este display (impressão na consola) é actualizado cada vez que um pacote é enviado/recebido na totalidade se já tiver passado um tempo mínimo definido na constante *UPDATE_DISPLAY_MIN_TIME_INTERVAL* desde a ultima actualização de display de modo a evitar que ocorra com elevada frequência o que poderia tornar difícil a sua leitura ou afectar a leitura/recepção de tramas.

Capítulo 9

Conclusões

TODO: síntese da informação apresentada nas secções anteriores; reflexão sobre os objectivos de aprendizagem alcançados

Capítulo 10

Bibliografia

Anexos

Anexo A

Código Fonte

```
int llopen(app_status_type status)
{
    int fd;

    if (open_tio(&fd, 0, 0) != OK) {
        printf("\nERROR: Could not open terminal\n");
        return -1;
    }
    occ_log.num_of_Is = 0;
    occ_log.total_num_of_timeouts= 0;
    occ_log.num_of_REJs          = 0;
    if (status == APP_STATUS_TRANSMITTER)
    {
        app_status = APP_STATUS_TRANSMITTER;
        if( llopen_transmitter(fd) != OK) {
            close_tio(fd);
            return -1;
        };
    }
    else if (status == APP_STATUS_RECEIVER)
    {
        app_status = APP_STATUS_RECEIVER;
        if( llopen_receiver(fd) != OK) {
            close_tio(fd);
            return -1;
        };
    }
    else
    {
        printf("\nWARNING(llo):invalid app_status found on llopen().\n"
           );
        close_tio(fd);
        return 1;
    }
    return fd;
}
```

A.1 Função set_basic_definitions em DataLinkProtocol.c

```
void set_basic_definitions(int number_of_tries_when_failing, char* port
    , int baudrate, int packetSize)
{
    DEBUG_SECTION(DEBUG_PRINT_SECTION_NUM,
        printf("\n-section1-");
    sleep(1);
    );

    signal(SIGALRM, timeout_alarm_handler);

    int realbauds[20] = {0, 50, 75, 110, 134, 150, 200, 300, 600, 1200,
        1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400,
       460800};
```

```
    int realBaudrate;
    if (baudrate <= 15) realBaudrate = realbauds[baudrate];
    else realBaudrate = realbauds[baudrate - 4081];

    int timeout_in_seconds = ((packetSize + 4 + 1) * 2 + 5) / (
       realBaudrate / 8) + 1;
    printf("size: %d\n", packetSize);
    printf("baudrate: %d\n", realBaudrate);
    printf("timeout: %d\n", timeout_in_seconds);
    link_layer_data.timeout = timeout_in_seconds;
    link_layer_data.numTransmissions = number_of_tries_when_failing;
    if (port_name_was_set == NO) { strcpy(link_layer_data.port, port);
       port_name_was_set = YES; }
    link_layer_data.baudRate = baudrate;
}
```

A.2 Função randomErrorData em DataLinkProtocol.c

```
void randomErrorData(char* trama, int data_size) {
    int trama_byte;
    if (DATA_PROB > 0) {
        int data_error = rand() % DATA_PROB;
        if (data_error == 0) {
            trama_byte = rand() % data_size;    // escolher o byte de
               erro na zona dos dados da trama
            trama[trama_byte]++;            // o erro e incrementar o
               byte escolhido
        }
    }
}
```

Anexo B

Tipos de Tramas Usadas

As tramas que utilizamos podem ser de três tipos:

- Informação (I) - transportam dados;
- Supervisão (S) - são usadas para iniciar e terminar a transmissão, assim como para responder a tramas do tipo I;
- Não Numeradas (U) - são usadas para responder a tramas de início e fim de transmissão.

Todas as tramas são delimitadas pelas *Flags* F - 01111110. Além disso, independentemente do tipo da trama, o cabeçalho é sempre o mesmo conjunto de 3 bytes:

1. A - Campo de Endereço - 00000011 em comandos enviados pelo Emissor e respostas enviadas pelo Receptor, 00000001 na situação inversa;
2. C - Campo de Controlo:
   - tramas I - 00S00000, onde S é o bit que identifica a trama;
   - tramas SET (*set up*) - 00000111;
   - tramas DISC (*disconnect*)- 00001011;
   - tramas UA (*unnumbered acknowledgment*) - 00000011;
   - tramas RR (*positive acknowledgment*) - 00R00001, onde R identifica a trama;
   - tramas REJ (*negative acknowledgment*) - 00R00101, onde R identifica a trama;
3. BCC1 (*Block Check Character*) - Campo de Proteção do Cabeçalho - é obtido realizando a disjunção exclusiva bit a bit de A e C.

Se se tratar de uma trama I, após o cabeçalho temos:

- $D_1, D_2, \ldots, D_N$ - bytes de dados;
- BCC2 - Campo de Proteção dos Dados - calculado de forma que exista um número par de 1s em cada bit dos dados, incluindo o BCC2.

As restantes tramas apenas têm o respetivo cabeçalho delimitado por flags.