Trabalho 1 - Camada Física

Denilson Figueiredo de Sá

2010-04-27

1 Introdução

Redes de computadores são estudadas e implementadas em um modelo de camadas, de modo que a camada de baixo presta serviços para a camada de cima, e cada camada tem responsabilidades distintas. Neste primeiro trabalho, o objetivo é emular a camada mais baixa do modelo — a camada física — e preparar uma base para os próximos trabalhos.

2 Visão geral

A camada física é aquela em contato direto com o meio físico e é responsável por converter bits em algum tipo de sinal a ser propagado por um meio de transmissão, assim como converter o sinal de volta para bits. Estes são alguns dos tipos mais comuns de sinais e meios de transmissão:

- variação de tensão ou corrente elétrica em um ou mais fios condutores;
- onda eletromagnética propagada pelo ar;
- feixes de luz dentro de uma fibra óptica.

A forma como os dados (os bits) são codificados é tarefa exclusiva da camada física. Isto inclui questões como modulação, sincronia, multiplexação, colisões, interferências, ruído e velocidade de transmissão.

Estudar todas as tarefas e dificuldades da camada física está muito além do escopo deste trabalho e da disciplina, e por isso objetivo do trabalho é emular a camada física sobre algum canal de comunicação já existente. Por motivos de praticidade e flexibilidade, foi decidido que a emulação seria feita por cima do protocolo UDP.

3 Projeto da camada física

O projeto de emulação da camada física possui 5 funções básicas:

```
int P_Activate_Request(int, char*);
```

Inicializa a emulação da camada física. Recebe dois parâmetros: a porta UDP e o endereço IP do host de destino. Retorna 0 em caso de falha.

```
void P_Deactivate_Request(void);
```

Finaliza a emulação da camada física.

```
void P_Data_Request(char);
```

Envia um caractere.

```
int P_Data_Indication(void);
```

Retorna 1 caso exista uma byte disponível para leitura.

```
char P_Data_Receive(void);
```

Lê (consome) o byte disponível para leitura.

4 Implementação (emulação) da camada física

A implementação está dividida em 4 arquivos:

fisica.c contém toda a API da camada física, encapsulando o uso do UDP.

fisica-teste.c contém a função main() e utiliza as funções dos outros 3 arquivos.

fisica-teste-args.c contém código para reconhecer os parâmetros da linha de comando.

nbiocore.c abreviação de ***Non-Blocking I/O Core***, contém todo o código responsável pela leitura não-bloqueante do socket e do stdin, encapsulando o uso do `select()`.

Apenas o arquivo principal **fisica-teste.c** conhece a existência dos outros arquivos (na verdade, conhece apenas a API disponibilizada através dos ***.h**). Cada um dos arquivos restantes pode ser considerado um módulo completamente encapsulado, alheio à existência e ao funcionamento dos outros. Cabe ao **fisica-teste.c** possuir o código "cola" para integrar todos os módulos.

5 Dificuldades encontradas

5.1 Extensões da API de emulação da camada física

Todas as funções originais da API são precedidas por P_. Todas as funções extras, de extensão da API original, são precedidas por Pex_.

```
void Pex_Set_Local_Port(int);
```

A descrição original da API de emulação da camada física possuía a limitação de apenas receber uma porta na função de inicialização. Por esse motivo, era impossível escutar numa porta e enviar dados para outra porta, e portanto era impossível realizar a comunicação entre múltiplas instâncias do programa na mesma máquina.

Esta função foi escrita para solucionar essa limitação. Esta função deve ser chamada antes da P_Activate_Request(), indicando em qual a porta local o programa deve escutar por pacotes UDP.

Se esta função não for chamada, ou se for chamada com parâmetro igual zero, então o módulo da camada física vai assumir que a porta local e a porta remota serão iguais, exatamente como previsto na API original.

```
int Pex_Get_Socket_Fd(void);
void Pex_Receive_Callback(int);
```

Estas duas funções existem para permitir a integração do módulo de emulação da camada física com o módulo de entrada e saída não-bloqueante. A primeira destas funções retorna o *file descriptor* do socket UDP, necessário para o uso do select(). A segunda função é simplesmente um callback a ser executado quando houver dados disponíveis no socket.

5.2 Non-Blocking I/O Core

Este módulo foi desenvolvido para permitir o uso não-bloqueante da entrada padrão e do socket. Basicamente, este módulo encapsula a função select() numa API de fácil uso.

```
void nbio_register(int, nbio_callback_t);
```

Registra um novo *file descriptor* e associa uma função a ser executada sempre que esse *file descriptor* estiver disponível para leitura.

```
void nbio_unregister(int);
```

Esta função é o contrário da função anterior: faz um *file descriptor* não ser mais monitorado pelo nbiocore.

```
void nbio_register_idle(int, nbio_callback_t);
```

Define um limite de tempo que o nbiocore vai esperar até ter uma leitura disponível. Caso nesse tempo nenhuma leitura fique disponível, a função aqui registrada será chamada. Por padrão, ele vai esperar um tempo infinito.

```
void nbio_loop();
```

Inicia o loop principal do nbiocore.

```
void nbio_stop_loop();
```

Sinaliza a parada do loop principal. Esta função deve ser chamada a partir de alguma função de callback. Se esta função nunca for chamada, o loop principal nunca vai terminar.

5.3 Unbuffered standard input

Como o objetivo deste trabalho é transmitir bytes individualmente, não faz muito sentido ter um buffer de "linha" na entrada padrão. Portanto, foram usadas funções da termios.h para mudar colocar terminal em modo *non-canonical*.

6 Execução do programa fisica-teste

6.1 Compilação

A compilação é bastante simples, pois um Makefile está disponível.

```
make
```

Também é possível limpar os arquivos compilados usando:

```
make clean
```

6.2 Parâmetros de execução

O programa fisica-teste aceita os seguintes parâmetros:

```
./fisica-teste [local_port] [remote_host]:remote_port
```

local_port é a porta local na qual o programa vai escutar por pacotes UDP. Este parâmetro é opcional, caso omitido, é assumido que a porta local será igual à porta remota.

remote_host é o endereço IP do host com o qual o programa tentará se comunicar. Caso omitido, é assumido o endereço *loopback* (127.0.0.1).

remote_port é a porta usada para comunicação com o host remoto. Também será a porta local, caso esta tenha sido omitida na linha de comando.

6.3 Exemplo de execução

Digite cada um dos comandos a seguir em terminais diferentes na mesma máquina.

```
./fisica-teste 1235 :1234
./fisica-teste 1234 :1235
```

A partir de então, cada byte digitado em um programa será enviado através de pacotes UDP para o outro programa.

A combinação *Ctrl+D* envia um caractere que sinaliza o fim da transmissão, e encerra o programa.

7 Conclusões

O trabalho foi implementado usando variáveis globais para armazenar o estado dos diversos módulos (em especial, do módulo de emulação da camada física). Embora seja uma solução simples, não é *thread-safe* e não permite o uso de múltiplas interfaces físicas simultaneamente. Por outro lado, não há outra forma de implementar esse módulo e manter a API original.

Uma possível melhoria futura é alterar a API para inserir conceitos de orientação a objetos. Não é necessário usar uma linguagem OO para isso, basta a inicialização retornar uma **struct**, e todas as outras funções passarem a operar em cima dessa **struct**, de maneira similar à forma como funciona a API de leitura e escrita de arquivos (stdio.h) e a API de sockets.

Uma vez que a API seja modificada para suportar múltiplas interfaces da camada física, podemos criar um programa que funcionará como *hub* virtual, replicando um pacote recebido para todas as outras "portas". Embora não seja possível simular uma colisão na nossa camada física emulada, essa ferramenta irá permitir a simulação e o estudo de topologias de rede *broadcast*, ou de meio físico compartilhado.

8 Referências Bibliográficas

Todos as referências a respeito do código estão incluídas dentro do próprio código-fonte, na forma de comentários. Além disso, as manpages do Linux foram amplamente consultadas.