Trabalho 2 - Camada de Enlace

Denilson Figueiredo de Sá

2010-06-07

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 segundo trabalho, o objetivo é emular a camada de enlace utilizando os serviços oferecidos pela camada física, já implementados no trabalho anterior.

2 Visão geral da camada de enlace

A camada de enlace é responsável pelo envio de pacotes entre dois hosts adjacentes. Para que isso seja possível, esses dois hosts precisam estar conectados diretamente através da camada física, conexão essa que pode ser ponto-a-ponto ou broadcast, e pode ser por um meio guiado ou não-guiado.

3 Projeto da camada de enlace

A parte mais importante no projeto e implementação desta camada foi definir o protocolo de comunicação, as limitações e os serviços oferecidos.

3.1 Características gerais

- Endereço de enlace (MAC) possuindo apenas 1 byte.
- Endereço de broadcast é o caractere asterisco.
- Detecção de erros tanto no cabeçalho quanto no payload.
- Detecção de erros usando XOR.
- Tamanho máximo de 255 bytes para o payload.
- Possui um marcador de início-de-quadro e um marcador de final-de-quadro (! e #, respectivamente).
- Não há retransmissão.
- Não há controle de acesso ao meio.
- Não há controle de fluxo.

Esta implementação da camada de enlace funciona sobre uma emulação de camada física de maneira similar às redes Ethernet: cada enlace físico é ponto-a-ponto (liga um host a outro host, ou um host a um comutador), mas o meio lógico pode ser considerado broadcast. Além disso, cada host tem seu próprio endereço de enlace.

Por outro lado, devido às características do meio físico emulado, certas funcionalidades do Ethernet (como retransmissões e controle de fluxo) foram abandonadas em favor de uma implementação mais simples, mais próxima ao protocolo PPP.

Por motivos de simplificação do problema e simplificação da implementação, o endereço de enlace possui apenas 1 byte (256 valores possíveis). Para o escopo desta série de trabalhos, essa quantidade de endereços é suficiente.

Novamente por motivos de simplificação, o tamanho do payload foi limitado em 255 bytes e a função XOR foi escolhida para o cálculo do checksum. Aumentar o tamanho do payload e escolher uma função melhor para o checksum seriam decisões bastante importantes para aumentar a eficiência e a confiabilidade do protocolo, porém não acrescentam nenhum valor didático significativo a este projeto.

A escolha do caractere asterisco como endereço broadcast tem um simples motivo prático: é muito mais fácil digitar caracteres presentes no teclado do que digitar algum código que possa representar o caractere 255. Como a escolha de um endereço broadcast é completamente arbitrária, foi decidido simplificar a implementação e a interação com o usuário usando um endereço fácil de digitar.

Pensando em detectar prematuramente quadros inválidos, foi decidido usar um checksum apenas para o payload e outro checksum apenas para o cabeçalho. Desta forma, é possível identificar "lixo" após aproximadamente 8 bytes, bastando calcular o checksum do cabeçalho, sem esperar o recebimento do payload.

Por fim, não é feito nenhum tratamento para o caso dos marcadores de início-de-quadro e de fim-de-quadro aparecerem dentro do payload. Nesta implementação, o marcador de início-de-quadro apenas sinaliza um possível início de um quadro, cabendo ao par de checksums a verificação dessa possibilidade. O marcador de fim-de-quadro é redundante, pois, uma vez que o checksum do cabeçalho seja verificado, o tamanho do payload é conhecido.

3.2 API (quase) orientada a objetos

Para este trabalho, foi necessário aplicar uma mudança na API da camada física, devido limitações existentes na API anterior. Similarmente, a API proposta para a camada de enlace também sofreu o mesmo tipo de mudança.

Todas as funções de camada física recebem um ponteiro para um objeto physical_state_t, o qual contém todo o estado da camada física emulada. Analogamente, as funções da camada de enlace recebem um ponteiro para um objeto link_state_t. Essa foi a mais importante mudança em relação à API originalmente proposta, e permitiu que fossem implementados programas com múltiplas interfaces emuladas, permitindo, portanto, a fácil implementação de um hub e de um switch.

Essa mudança segue a mesma convenção usada pelas funções de leitura/escrita de arquivos da stdio.h e pode ser considerada como um tipo primitivo de orientação a objetos.

3.3 Estruturas de dados

`physical_state_t`

É uma estrutura que armazena todo o estado (contexto) de uma interface física virtual. Sempre que possível, a variável que aponta para essa estrutura chama-se PS.

`link_address_t`

Tipo de dado para o endereço de enlace. É um **typedef** para um caractere.

LINK_ADDRESS_BROADCAST

É uma constante para o endereço broadcast de enlace. Atualmente definido como '*'.

`link_state_t`

É uma estrutura que armazena todo o estado (contexto) da camada de enlace associada a uma interface física. Sempre que possível, a variável que aponta para essa estrutura chama-se LS. Dentro dessa estrutura há um ponteiro PS para a physical_state_t associada a este enlace.

3.4 Funções da camada de enlace

```
link_state_t* L_Activate_Request(link_state_t*,
    link_address_t, physical_state_t*);
```

Inicializa a camada de enlace. Pode receber um ponteiro para uma estrutura link_state_t previamente alocada, ou pode receber NULL para alocar uma nova estrutura. Recebe também o endereço de enlace e um ponteiro para a physical_state_t associada a este enlace.

Retorna um ponteiro para link_state_t em caso de sucesso, ou NULL em caso de falha.

A API originalmente proposta para esta função recebia todos os parâmetros da camada física. Isto seria uma redundância, pois já existe uma função dedicada a realizar inicializações da camada física, e portanto a API foi modificada para receber uma estrutura já inicializada pela camada física.

```
void L_Deactivate_Request(link_state_t*);
```

Desativa o enlace, mas não desativa a camada física. Atualmente esta função não faz nada.

```
void L_Data_Request(link_state_t*,
    link_address_t, const char*, int);
```

Requisita o envio de uma certa quantidade de bytes para um determinado endereço. O envio é síncrono/bloqueante.

```
int L_Data_Indication(link_state_t*);
```

Retorna 1 caso haja pelo menos um quadro disponível no buffer de leitura, retorna 0 caso contrário.

```
int L_Data_Receive(link_state_t*,
    link_address_t*, link_address_t*, char*, int);
```

Retira o quadro do buffer de leitura e armazena o payload no buffer passado como parâmetro. Opcionalmente também armazena os endereços fonte e destino.

A API originalmente proposta não previa o retorno do endereço de destino (afinal, na maior parte das vezes o destino será o próprio host). Foi decidido retornar também o endereço de destino para que fosse possível a implementação do comutador da camada de enlace. Como bônus, esta mudança também tornou possível identificar quadros broadcast.

```
void L_Set_Loss_Probability(link_state_t*, float);
```

Define a probabilidade (entre 0.0 e 1.0) de um quadro recebido com sucesso ser descartado, com o objetivo de simular falhas na transmissão.

```
void L_Set_Promiscuous(link_state_t*, int);
```

Define se o enlace será promíscuo, ou seja, se todos os quadros serão disponibilizados para a aplicação ou se somente os quadros cujo destinatário é o próprio host (ou broadcast).

```
void L_Receive_Callback(link_state_t*);
```

Função de callback que deve ser executada sempre que houver um byte disponível para leitura. Esta função realiza as seguintes etapas:

- chama a função de callback da camada física (P_Receive_Callback());
- copia para o buffer de enlace quaisquer bytes disponíveis na camada física;
- chama a função L_Analyze_Buffer().

```
void L_Analyze_Buffer(link_state_t*);
```

Esta função é apenas para uso interno, e não está exposta para a aplicação.

Esta função implementa a parte de leitura do protocolo de enlace. Ela detecta e descarta lixo (bytes soltos, fora de quadros), cabeçalhos corrompidos e quadros corrompidos. Caso encontre um quadro inteiro, disponibiliza esse quadro para o usuário (através das funções citadas acima).

4 Implementação da camada de enlace

O código do projeto ficou dividido em três diretórios:

lib/	contém o código-fonte compartilhado entre os programas.
src/	contém o código-fonte dos programas.
run/	contém scripts para execução rápida dos programas.

Dentro do diretório *lib/* podemos encontrar os seguintes arquivos fonte (e seus cabeçalhos):

fisica.c	contém toda a API da camada física, encapsulando o uso do UDP.
enlace.c	contém toda a API da camada de enlace, utilizando os serviços da camada física.

nbiocore.c módulo *Non-Blocking I/O Core*, encapsulando o uso do select() para a leitura não-bloqueante (apesar do nome I/O, no momento este módulo só trata a entrada).

terminal.c funções para alterar as flags do terminal para o modo não-canônico.

util.c pequenas funções comuns a todos os outros arquivos.

Dentro do diretório *src/* podemos encontrar os seguintes programas:

fisica-teste.c permite enviar e receber bytes diretamente pela camada física.

fisica-hub.c implementação de um hub, simplesmente replica os bytes recebidos para todas as outras interfaces.

enlace-teste.c permite enviar e receber quadros pela camada de enlace.

enlace-switch.c implementação de um switch, repassa os quadros apenas para a interface à qual o endereço de destino está conectado. O tempo de expiração do cache é definido pela constante CACHE_EXPIRE.

5 Dificuldades encontradas

5.1 Limitações da API proposta

Como citado no trabalho anterior, e como já mencionado em bastantes detalhes neste relatório, a API proposta originalmente possuía diversas limitações. A fim de contornar tais limitações e tornar possível a execução deste trabalho, a proposta original foi modificada, e as mudanças estão descritas neste relatório.

5.2 Execução automatizada dos programas

Rodar cada programa manualmente era possível (embora um pouco inconveniente) quando tínhamos apenas duas instâncias simultâneas. Porém, com o crescimento da complexidade, com a implementação de hub e switch, e com o aumento da quantidade de instâncias rodadas simultaneamente, a execução manual torna-se inviável.

Para solucionar esse problema, foram criados alguns scripts dentro do diretório *run/*. Esses scripts executam diversas instâncias dos programas, cada uma com seus devidos parâmetros de execução, em diferentes terminais. Isso tornou possível rodar testes com um número arbitrário de instâncias simultâneas. O máximo de trabalho manual é organizar as várias janelas abertas de modo que seja possível visualizar todas simultaneamente.

5.3 Mainloop

A proposta original da API de enlace previa uma função **void** L_MainLoop(**void**). Devido ao uso do módulo *nbiocore*, essa função tornou-se completamente desnecessária, e nem sequer foi implementada (as atribuições dessa função foram implementadas na função de callback).

6 Compilação e execução

6.1 Compilação

A compilação é bastante simples:

```
make
```

Também é possível limpar os arquivos compilados usando:

```
make clean
```

6.2 Execução

Cada programa será compilado no diretório raiz deste projeto (no mesmo diretório do *Makefile*). Cada programa possui um pequeno texto de ajuda embutido, descrevendo de maneira resumida quais parâmetros ele espera receber.

Para todos os programas, a ideia básica é a mesma: a porta local de uma instância deve estar associada à porta remota de outra. Quando isto acontece, então essas instâncias estão virtualmente conectadas e podem se comunicar.

Para simplificar a execução, existem vários scripts dentro do diretório *run/*, e em geral é mais fácil rodar esses scripts do que executar o programa manualmente. Também é bastante fácil criar novos scripts para novos casos de teste.

7 Conclusões

Com as mudanças na API, a implementação das camadas física e de enlace tornou-se bastante robusta e fácil de usar. O módulo de I/O não-bloqueante também mostrou-se extremamente valioso, mantendo o código simples de escrever e o uso de CPU mínimo. Juntando todos os módulos desenvolvidos durante o projeto, implementar um hub de camada física e um switch de camada de enlace foi uma tarefa bastante simples, rápida e agradável.

Uma possível melhora para trabalhos futuros é manter um único idioma em todo o código. A API proposta possui nomes em inglês, porém os arquivos precisam ter nomes em português.

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.