DCCNET

João Francisco B. S. Martins, Victor B. R. Jorge

14 de Maio de 2017

1 Introdução

Em Redes de Computadores, a camada de enlace representa uma das sete camadas do modelo OSI. Ela é responsável pela detecção de erros - e possíveis correções - na camada física. Além disso, é responsável pela delimitação de quadros tanto na recepção quanto no envio e também pelo controle do fluxo de rede.

Sendo assim, cabe a essa camada garantir a implementação de alguns preceitos básicos da comunicação entre computadores, como: confiabilidade de transmissão, controle do fluxo de envio e recebimento, além de tornar possível o múltiplo acesso ao meio de transmissão através da comutação de pacotes. Portanto, essa camada é de extrema importância na padronização e eficiência da comunicação entre computadores.

Neste trabalho, será implementado um emulador de camada de enlace para uma rede fictícia chamada DCCNET. Esse emulador implementará todas as funções de uma camada de enlace real: sequenciamento e enquadramento, detecção de erros com retransmissão de dados e controle de fluxo.

2 Implementação

O quadro DCCNET é composto de duas sequências para sincronização da forma 0xDCC023C2 (32 bits, cada), um *checksum* (16 bits), um campo *lenght* que informa o tamanho do *payload* (16 bits), um campo ID de para sequenciamento (8 bits) e um campo *flag* para informar se aquele pacote é um ACK ou se é o último pacote de dados (8 bits).

A DCCNET implementa um protocolo do tipo *stop-and-wait* e cada nó da rede pode funcionar como receptor e transmissor simultaneamente. Portanto, nossa camada de enlace deve ser capaz de lidar com essa especificação.

O emulador da camada de enlace pode ser aberto em dois modos. O modo servidor, em que é realizada uma abertura passiva do *socket* TCP, e o modo cliente, quando é realizada uma abertura ativa em um servidor de endereço conhecido. Porém, tanto cliente quanto servidor devem ser capazes de enviar e receber dados. Cada um dos dois pares possui como parâmetro de entrada dois arquivos: um que deve ser lido e outro que deve ser escrito no *socket*. Como não há nenhum tipo de *handshake* os dois lados têm de estar sempre preparados para receber/enviar os arquivos assim que a comunicação for estabelecida.

2.1 Envio

A lógica do envio de dados pode ser representada pelo fluxograma abaixo:

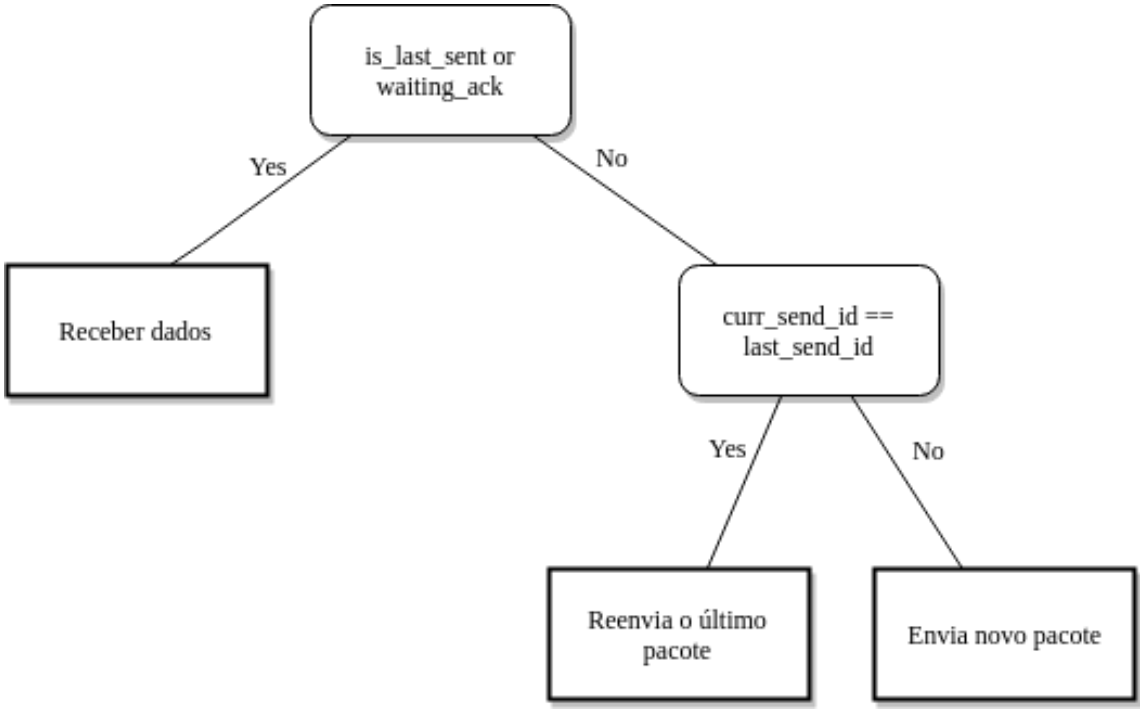

Figure 1: Fluxograma para o envio de dados

Se o último pacote que possuíamos pra enviar já foi enviado ou se estamos esperando por um ACK, ignoramos a etapa de envio e partimos pra etapa de recebimento dos dados. Caso não tenhamos enviado todos os pacotes do arquivo ou não estejamos esperando por um ACK - ou seja, já recebemos o ACK para o último pacote enviado - podemos enviar um novo pacote. Se o id do próximo pacote a ser enviado (cur_send_id) for diferente do id do último pacote enviado (last_send_id), enviamos um novo pacote. Caso sejam iguais, isso quer dizer que não recebemos o ACK e aquele pacote sofreu um *timeout* e portanto devemos reenviá-lo.

2.1.1 Montagem do pacote

Assim que é decido qual pacote deve ser enviado, ele é montado de acordo com as especificações da camada. Se aquele pacote é o último, o valor correspondente a END é setado no campo *flag*, buscando evitar o *overhead* de se enviar um pacote vazio avisando o fim do envio. Posteriormente, o *lenght* é calculado e adicionado ao cabeçalho, e os dados são adicionados ao final do pacote. O *checksum* é então calculado e adicionado ao seu respectivo slot em network byte order. Montado o pacote, ele é enviado pela função *socker.sendall()*.

2.2 Recebimento

Após enviar quaisquer dados pendentes, a camada espera pelo recebimento de um ACK ou de um pacote com dados. Ao chegar qualquer dado no *socket*, é checado se o pacote possui o formato correto. A primeira checagem realizada consiste em verificar se os oito primeiros *bytes* do pacote correspondem à sequência SYNC duas vezes. Isso pode ser realizado através do seguinte DFA:

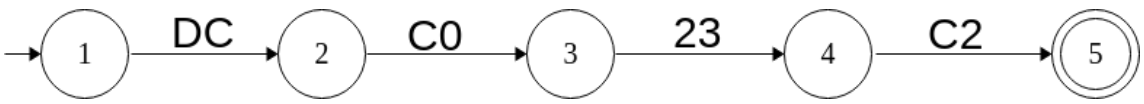

Figure 2: DFA para a sequência SYNC

Reconhecidas as duas sequências SYNC pelo autômato, o pacote é recebido por inteiro e seu *checksum* é calculado logo após convertermos o campo *chksum* para sua sequência de bytes original. Caso esteja correto, o pacote é então recebido com sucesso. A função que recebe o pacote também

retorna a informação do campo *flag* (is_ack, is_last), além do id do pacote para o programa. Retornada a função de recebimento, o funcionamento do recebimento pode ser representado pelo seguinte fluxograma:

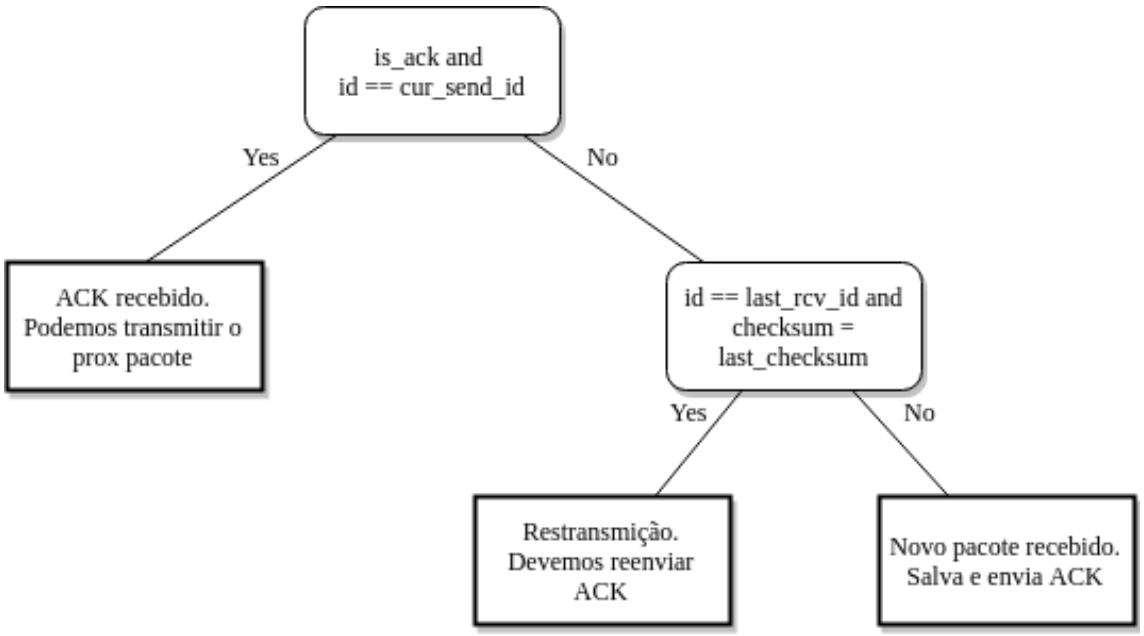

Figure 3: Fluxograma para o recebimento de dados

2.3 Timeout

O *timeout* dos pacotes é um dos aspectos mais delicados deste trabalho e requer uma atenção especial. Há alguns casos especiais a serem tratados após o envio de um pacote de dados. Na nossa implementação, são necessários dois *timers*: um do próprio *socket* que é implementado pela função *socket.settimeout()* e outro implementado por nós. Cada caso é tratado por um dos timers. É possível visualizar melhor a dinâmica dos *timers* pelo fluxograma abaixo.

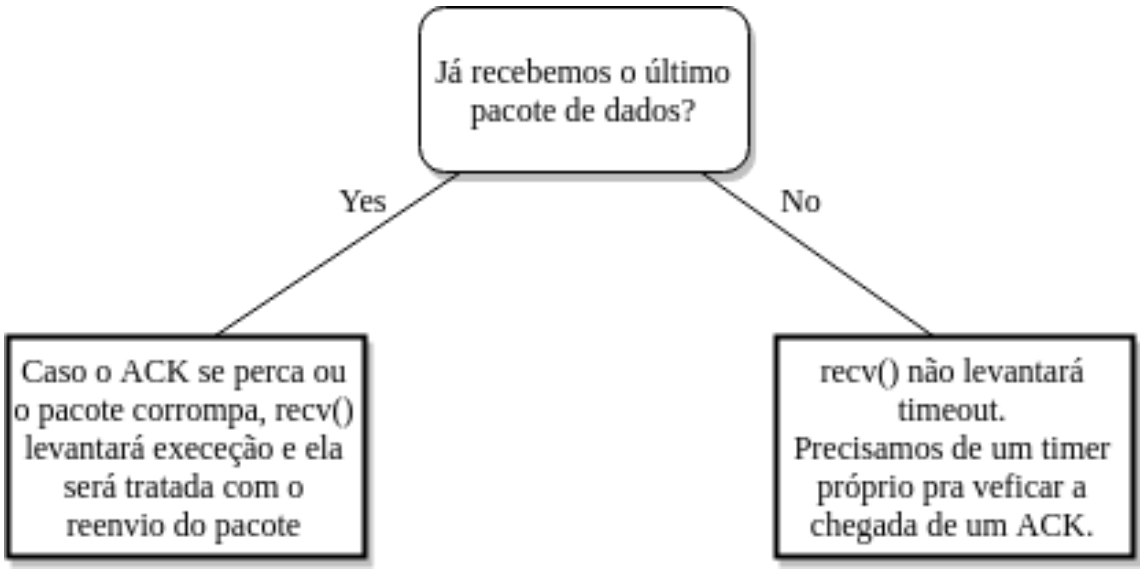

Figure 4: Fluxograma para o recebimento de dados

Caso já tenhamos recebidos todos os pacotes que nos seriam enviados (pode ser verificado pela *flag* END), a função *socket.recv()* receberá somente ACK's e portanto, caso ela atinja o *timeout* definido por *socket.settimeout()*, devemos tratar essa exceção com o reenvio do pacote. Entretanto,

caso ainda estejamos recebendo pacotes de dados, a probabilidade da função *socket.recv()* levantar uma exceção de *timeout* é muito baixa, mesmo passando-se o tempo de um ACK. Portanto, nesse caso, devemos usar nosso *timer* que é implementado da seguinte maneira: sempre que enviamos um pacote, o tempo é armazenado. Sempre antes de conferir se temos de enviar outro pacote o tempo transcorrido é calculado. Caso ele seja maior que o *timeout* estipulado de um segundo, o pacote é reenviado.

3 Conclusão

Concluímos que a camada de enlace é um dos componentes mais importantes das redes de computadores atuais. Através dela é que podemos separar a informação em pacotes e permitir o múltiplo acesso aos meios através da comutação dos pacotes. Além disso, ela que controla a confiabilidade e robustez da rede.

Ademais, o trabalho foi desafiador, principalmente na parte de planejamento e organização da ordem de envio e recebimento dos dados, já que se trata de uma comunicação *full-duplex*. A maior dificuldade encontrada pelo grupo foi na implementação dos *timeouts* e como conciliar o *timeout* do próprio *socket* com nosso *timeout* de pacote.