Emulador de Camada de Enlace (DCCNET)
Trabalho Prático 3 - Redes de Computadores

Bruno Esteves Campoi {2020054250}[1]
Mirna Mendonça e Silva {2021421940}[1]
Vinicius Silva Gomes {2021421869}[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{bec, mirnamendonca, vinicius.gomes}@dcc.ufmg.br

1. Introdução

Nesse TP foi implementado um emulador de camada de enlace para uma rede fictícia chamada DCCNET. Esse emulador será responsável por lidar com o enquadramento, o sequenciamento, a detecção de erros e a retransmissão de quadros/dados.

A ideia é que sejam desenvolvidas funções/classes que implementem as especificações definidas para o enlace e para o tipo de comunicação e que, em seguida, esse emulador seja testado em duas aplicações diferentes: uma aplicação de transferência de arquivos entre um cliente e um servidor e uma aplicação que recebe mensagens de um servidor e devolve essas mensagens codificadas usando o algoritmo de codificação MD5.

A seção 2 discute como o problema foi modelado computacionalmente; na seção 3 os detalhes da implementação realizada são discutidos; na seção 4 os passos para executar o código são detalhados; e a seção 5 encerra o relatório.

2. Modelagem computacional

Para implementar as especificações definidas para o enlace, a classe `DCCNET` foi criada com uma série de funções que as duas aplicações propostas na especificação iriam utilizar. A ideia por trás foi definir os comportamentos comuns do enlace de forma separada, modularizada, e, com isso, dar mais liberdade para codificação das aplicações, que poderiam usar essas funcionalidades como bem entendessem.

Nessa classe, alguns métodos importantes foram especificados, como `resolve_connection()`, `checksum()`, `is_ack_frame()`, `is_reset_frame()`, `is_end_frame()`, `encode()`, `encode_ack()`, `send_frame()`, `receive_frame()`, `reconstruct_frame()`, `checksum_match()` e `is_acceptable_frame()`.

2.1. `resolve_connection()`

Método usado no código do cliente das aplicações para identificar, a partir do *host* informado, o IPv4 ou IPv6 do servidor (preferenciando o IPv6) para que a conexão aconteça e a troca de mensagens possa começar.

2.2. `checksum()`

Método que computa o *checksum* usando o mesmo algoritmo de *checksum* usado na Internet. O número de bytes no cabeçalho do quadro é par, portanto, caso o número de bytes dos dados seja ímpar, o algoritmo insere um byte nulo nos dados para que o *checksum* possa ser calculado adequadamente.

2.3. `is_ack_frame()`, `is_end_frame()` e `is_reset_frame()`

Métodos que recebem o campo *flag* do quadro recebido e comparam se ele contém a *flag* do respectivo tipo de quadro preenchida: `0x80` para *acknowledgments* (ACKs), `0x40` para *end frame* e `0x20` para *reset frame*.

2.4. `encode()`

Método responsável por montar e retornar um pacote a partir das informações recebidas como parâmetro. Para tanto, o método recebe os dados a serem enviados, o ID do pacote e a *flag* do pacote, calcula o *checksum*, computa e retorna um conjunto de bytes através da função `struct.pack()`. que representam o pacote a ser enviado pela rede usando o *socket*.

2.5. `encode_ack()`

Utiliza o método `encode()` informando como parâmetro de dados um byte vazio, o ID recebido como parâmetro e a *flag* pré-definida para quadros de ACK (`0x80`).

2.6. `send_frame()`

Recebe uma instância de um `socket` e um quadro e encaminha esse quadro pelo `socket` informado.

2.7. `receive_frame()`

Método responsável por escutar o canal procurando por transmissões de quadros que sigam o formato especificado. Para tanto, o método começa escutando o canal e procurando pelos bytes de sincronização (os detalhes da escuta de dos bytes de sincronização e recuperação dos erros serão melhor destrinchados na subseção 3.1).

Quando eles são encontrados, o restante do cabeçalho do quadro é lido e, a partir do tamanho indicado no campo *length*, os dados contidos naquele quadro são lidos e retornados na estrutura de um dicionário pelo método.

2.8. `reconstruct_frame()`, `checksum_match()` e `is_acceptable_frame()`

Essas funções são responsáveis por analisar os quadros recebidos e verificar se não há inconsistência no cálculo do *checksum*. Para tanto, `is_acceptable_frame()` recebe o frame recebido como parâmetro, reconstrói ele a partir dos dados e compara o *checksum* calculado com o *checksum* recebido. Se eles não forem iguais, o quadro é descartado.

3. Detalhes da implementação

3.1. Recuperação após erros ocorrerem

Diversos tipos de erro podem acontecer ao longo da comunicação entre as pontas. A primeira checagem é quanto ao *checksum*. Após receber um quadro da outra ponta, o receptor calcula o *checksum* do quadro recebido e compara com o *checksum* enviado junto do quadro. Caso esses *checksums* não sejam iguais, o quadro é descartado e o receptor prossegue tentando ouvir mais quadros ou tentando enviar novas mensagens.

Outro erro que pode acontecer é a corrupção do campo *length* do quadro. Isso pode fazer com que mais ou menos bytes sejam recebidos, prejudicando a recepção do quadro corrompido e, possivelmente, de quadros futuros. Para resolver esse problema, a recepção dos quadros executa uma rotina que, inicialmente, recebe 8 bytes da outra ponta. Caso esses 8 bytes correspondam aos bytes de sincronização que devem iniciar o cabeçalho de um quadro, o restante do quadro é recebido.

No entanto, caso não, uma janela com esses 8 bytes é criada e a recepção passa a acontecer byte a byte. Assim, o receptor recebe um novo byte, insere no final da janela e remove o primeiro byte da janela. Esse processo é repetido até que a janela contenha os 8 bytes de sincronização esperados. Quando isso acontecer, o restante dos dados do quadro pode ser lido.

Como a rotina que desloca a janela continua executando até que os bytes de sincronização sejam observados, após o *timeout* a outra ponta irá retransmitir o pacote e, eventualmente, uma transmissão sem erros será escutada no nó receptor, o que faz com que a comunicação prossiga.

3.2. Transmissão e recepção em paralelo

Assim como definido na especificação, os dois nós na comunicação são capazes de enviar e receber dados (comunicação *full-duplex*). Sempre que um novo quadro de dados for recebido, a ponta receptora deve enviar um quadro de *acknowledgment* para a ponta transmissora. Não existe ordem atrelada ao envio dos quadros de dados (um envia e depois o outro, cada um envia apenas um quadro por vez, etc), apenas que, ao receber um quadro de dados, a ponta receptora deve confirmá-lo para que o transmissor prossiga.

Para permitir que isso aconteça, as aplicações funcionam de tal forma que, após um nó enviar um quadro de dados, ele continua retransmitindo esse quadro enquanto não receber um ACK da outra ponta mas, caso receba um quadro de dados (enquanto espera um ACK), ele processa esse quadro e envia a confirmação para a outra ponta. Após fazer isso, essa ponta volta a escutar o enlace esperando pelo ACK da ponta receptora.

Isso foi feito através de um `while` que mantém o código retransmitindo o pacote de dados enquanto o ACK não for recebido e, caso um quadro com dados seja recebido, ele é processado e seu ACK enviado. Dessa forma, o nó que espera um ACK ainda é capaz de receber e processar pacotes de dados, mas não prossegue no envio das suas informações até que o ACK seja recebido. Quando ele é recebido, o nó transmissor pode continuar suas rotinas normais de recepção e/ou envio de dados.

Dessa forma, é possível que uma ponta envie todos os dados de uma vez e depois receba, envie um *stream* de uma vez e depois passe um tempo recebendo dados, etc. Não foram empregadas diferentes *threads* ou estruturas que permitem concorrência propriamente, mas o comportamento esperado no cenário onde um nó espera um ACK mas recebe um quadro de dados e, ainda assim, é capaz de processá-lo e continuar esperando pelo ACK do quadro enviado previamente, pode ser replicado assim.

3.3. Interface com o DCCNET

A interface das aplicações com a implementação do DCCNET é realizada principalmente através dos métodos fornecidos pela classe `DCCNET`. Esta classe abstrai as complexidades da manipulação direta de *sockets* e quadros de dados, permitindo que as aplicações se concentrem na lógica específica do que fazer com cada dado recebido e como operar corretamente.

No contexto das aplicações desenvolvidas, a integração com o DCCNET ocorre assim: as aplicações, ao serem executadas, utilizam o método `resolve_connection()` para determinar o endereço IP e a família de endereços adequados para estabelecer a conexão com o servidor. Isso garante que a comunicação seja possível independentemente do tipo de endereço (IPv4, IPv6 ou tradução de *hostname* para um desses).

Uma vez estabelecida a conexão, as aplicações utilizam os métodos de codificação (`encode()`, `encode_ack()`) e de envio (`send_frame()`) para transmitir dados estruturados conforme o protocolo. Os dados são encapsulados nos quadros e os métodos retornam informações sobre cada campo para facilitar a manipulação interna por parte de cada aplicação. Os dados são continuamente retransmitidos enquanto o ACK não for recebido, ACKs duplicados são descartados e quadros duplicados não são processados novamente, o ACK para eles é reenviado e a comunicação prossegue.

Para receber dados, as aplicações invocam o método `receive_frame()`, que é responsável por escutar o canal de comunicação, identificar e validar os quadros recebidos, e extrair os dados de acordo com o formato esperado. Este método lida automaticamente com a detecção e correção de erros, permitindo que as aplicações lidem com os quadros mais facilmente. Com o quadro recebido, a `DCCNET` fornece *helpers* que auxiliam as aplicações a identificarem o tipo do quadro recebido e, daí em diante, determinar o que fazer com o quadro: aceitar um quadro de dados, processá-lo e enviar sua confirmação; aceitar um ACK; descartar o quadro; etc.

Assim, as aplicações interagem com a implementação do DCCNET de maneira modular, delegando as operações de baixo nível para a classe `DCCNET` e focando-se na lógica a ser implementada e em quais dados serão enviados.

3.4. Dificuldades encontradas

De modo geral, o trabalho foi bem desafiador. Dentre os problemas maiores encontrados pelo grupo, destaca-se, primeiramente, a implementação da janela que identifica erros de sincronização. Apesar de ser um código pequeno, sua implementação e seu funcionamento junto da outra ponta foi relativamente complicado de ser feito.

Além disso, outro grande desafio foi a implementação da comunicação *full-duplex*. Inicialmente, para começar o trabalho de passo em passo, uma implementação *simplex* que depois evoluiu para uma *half-duplex* foi feita. No entanto, o grupo enfrentou muitos problemas para sair dessa implementação e chegar em uma comunicação *full-duplex*. O comportamento de continuar recebendo e confirmando pacotes da outra ponta enquanto espera um ACK foi bastante complicado de ser pensado e, em seguida, implementado.

Por fim, a escrita de outros testes de comportamento também foi um desafio para o grupo. Alguns comportamentos esperados, como *checksums* inválidos e problemas nos bytes de sincronização puderam ser testados isoladamente. Além disso, alguns testes da capacidade *full-duplex* do enlace, como o envio de vários pacotes por parte de um ponta até que a outra envie seus primeiros pacotes, entre outros; também foram feitos.

4. Como executar o código

O trabalho foi desenvolvido utilizando a linguagem de programação Python e alguns de seus módulos *built-in* (*socket*, *struct*, *hashlib*, etc). Além disso, a versão da linguagem que foi utilizada durante o desenvolvimento do trabalho foi a 3.10.12. Uma vez que o ambiente esteja configurado, o comando

```
python3 dccnet-md5.py <IP>:<PORT> <GAS>
```

deve ser utilizado para executar a aplicação de *hashs* MD5 usando o DCCNET e os comandos abaixo devem ser utilizados para executar o servidor e o cliente para a aplicação de transferência de arquivos, respectivamente.

```
python3 dccnet-xfer.py -s <PORT> <INPUT> <OUTPUT>
python3 dccnet-xfer.py -c <IP>:<PORT> <INPUT> <OUTPUT>
```

Para a execução da aplicação de MD5, o GAS utilizado foi:

```
# GAS
2021421869 :44:87407f792f59b7dde2bf51a0ae7216cf8c246a7169b52ac336bbf166938d91a1
    +2020054250 :44:50527
    ec32fc4c6fd5493533c67ce42f5fcad7bb59723976ff54acc6ae84385b8+2021421940 :44:
    a70a80b0528f580bb6c0a94ae37e3d8efdfb7adb9f939f3af675e9ea69694db4+
    f16d50fda86436470ba832a3f63525650dbd1fe021e867069f35ef4073d1b637
```

Entre os números de matrícula e o `:44` de cada *token* individual existem dois espaços. O formatador de código do LaTeX infelizmente não permite colocar esses dois espaços explicitamente.

5. Conclusão

A execução desse TP possibilitou o exercício de diversos conceitos vistos em sala de aula e aprofundou o conhecimento do grupo sobre os detalhes e desafios por trás das atividades que ocorrem no enlace de uma rede.

Em enlaces do mundo real, a detecção de erros normalmente é feita com estratégias mais robustas, como o Cyclic Redundancy Check (CRC), e a transmissão é feita usando algoritmos de janelas deslizantes. No entanto, mesmo a implementação de protocolos mais simples se mostrou bastante desafiadora.