Relatório do Trabalho Prático 02

Redes de Computadores

Thiago Martin Poppe

04/08/2021

1 Uma breve descrição

1.1 Problema proposto

Desenvolvimento de um emulador de camada de enlace para uma rede fictícia chamada DCCNET, permitindo tanto a transmissão quanto a recepção de dados por ambas pontas do enlace.

Foi definido um protocolo a ser seguido pelos quadros a serem transmitidos tanto pela ponta passiva (servidor) quanto pela ponta ativa (cliente). O quadro terá 112 bits de cabeçalho, possuindo a seguinte configuração:

- Os primeiros 32 bits serão reservados para uma sequência de sincronização. Foi adotado a sequência de valor hexadecimal `0xdcc023c2`, como indicado pelos exemplos disponibilizados.
- Os próximos 32 bits serão uma repetição da mesma sequência de sincronização definida anteriormente.
- Os próximos 16 bits serão reservados para o campo de tamanho, em bytes, dos dados a serem enviados pelo quadro.
- Os próximos 16 bits serão reservados para o campo de checksum. Para o cálculo do checksum foi utilizado a mesma lógica presente no cálculo do checksum de pacotes do IPv4, tendo um link para o seu estudo disponibilizado no enunciado do trabalho prático.
- Os próximos 8 bits serão reservados para um campo de ID do quadro, podendo assumir valor 0 ou 1.
- Os próximos 8 bits serão reservados para um campo de flag do quadro, indicando por exemplo se o quadro é um quadro de dados, confirmação ou de término da comunicação.
- Os demais bits serão reservados para os dados a serem transmitidos pelo quadro.

1.2 Observações

- Todo o sistema foi desenvolvido utilizando a linguagem de programação Python, especificamente a versão 3.8.5. Além disso, foi utilizado a biblioteca `socket`, nativa da linguagem, para a programação com sockets e toda a comunicação das pontas, que foi feita utilizando TCP com protocolo IPv4.
- Para a codificação e decodificação dos dados em base16 foi utilizado as funções `b16encode` e `b16decode` do módulo `base64`, nativa da linguagem utilizada.
- O sistema desenvolvido é 1:1, isto é, o servidor apenas consegue tratar um único cliente e, após encerrada a comunicação, ele irá encerrar. Caso uma das pontas desconecte por algum erro da rede, a outra ponta irá encerrar também.

2 Criação dos quadros

Sempre antes de iniciarmos o processo de comunicação propriamente dito, ambas pontas irão previamente criar uma lista de quadros a serem enviados, facilitando assim a manipulação e a comunicação de uma forma geral. Essa lista irá conter apenas quadros com dados, sendo o último o quadro que indica o término do envio de mensagens, isto é com a flag `ACK` ativada.

A criação do quadro sempre irá visar encaixar o maior número de dados por quadro, evitando assim muitas fragmentações, tornando o processo de comunicação mais eficiente. Em outras palavras, caso o dado possua menos do que o valor máximo do campo `length`, que será igual à $2^{16} - 1$ bytes, iremos conseguir enviar todo esse dado em um quadro apenas; caso contrário teremos que realizar sucessivas fragmentações em múltiplos quadros até conseguirmos enquadrar todo o dado de entrada.

Para criar os quadros propriamente ditos, utilizei tanto a função `pack` quanto a função `unpack` presente no módulo `struct`. Esse módulo visa estruturar um certo dado em bytes para, por exemplo, transmitirmos ele pela rede. Como os próprios nomes já sugerem, a função `pack` realiza o processo de "empacotamento"dos dados em bytes; já a função `unpack` realiza o processo de "desempacotamento", retornando assim os valores presentes em tais bytes dado um formato específico. Para realizarmos a estruturação do cabeçalho do quadro, decidi modelar os bytes de sincronização como `unsigned int`, os bytes de length e checksum como `unsigned short` e os bytes de ID e flag como `unsigned char`.

3 Fluxo de comunicação das pontas

Foi definido que ambas pontas devem ser capazes de tanto receber como transmitir dados, não podendo, por exemplo, apenas receber e posteriormente apenas enviar. Ao invés de utilizar um sistema de `threads` para tornar os processos paralelos, adotei um fluxo mais simples de envio e recepção alternada, permitindo assim com que a ponta consiga enviar dados e também receber dados da outra ponta. Defini um timeout de 1 segundo para a recepção dos dados, ou seja, se, após 1 segundo, a ponta não receber nenhum dado, iremos disparar uma "exceção"de timeout e iremos retomar o fluxo enviando dados.

Foi criado uma função chamada de `search_frame` que irá procurar pelo padrão de sincronização e, uma vez encontrado, irá retornar um quadro válido (já decodificado), caso não haja nenhum erro no quadro ou `None` caso contrário. Para detectar possíveis erros no quadro, antes de realizar a verificação com o checksum, decidi fazer algumas verificações "triviais", como observar se o valor do campo ID está definido como 0 ou 1 ou se a flag passada está consistente com o campo length, isto é, se tivermos um quadro ACK o campo length deverá ser 0, por exemplo. Além desses retornos, essa função poderá disparar tanto uma exceção por conta de um timeout (como mencionado anteriormente) como uma exceção devido à um erro na comunicação do enlace caso uma das pontas desconecte de forma inesperada.

Para o envio dos dados, foi definido uma função chamada de `send_data_frame` que irá enviar para a rede o quadro de dados atual já codificado em base16. Iremos apenas enviar o próximo dado apenas se recebermos um quadro ACK referente ao quadro de dados enviado. Além dessa verificação, iremos guardar o ID e checksum do último quadro a fim de verificar possíveis duplicações e inconsistências no sistema de comunicação do enlace.

Uma vez que uma ponta termina de enviar os seus dados, ela entrará em um loop apenas para receber possíveis dados presentes no buffer de leitura ao chamarmos um `recv`. A ponta irá encerrar a comunicação total somente se sofrer um timeout durante esse período, indicando assim, que o buffer de leitura está vazio e que não temos mais dados para serem lidos, ou seja, que a outra ponta terminou de enviar os seus dados.

4 Observações gerais

4.1 Organização do código

Para o desenvolvimento desse trabalho prático, criei um sistema de classes onde teremos uma classe "mãe"chamada de `BaseNode` possuíndo as funcionalidades em comum entre as pontas, como por exemplo a recepção e envio de dados bem como a criação de quadros. Além dessa classe teremos as classes `Client` e `Server` que irão inicilizar de forma adequada a ponta ativa e passiva, respectivamente, bem como definir o fluxo de execução

delas. Mesmo que a ideia do fluxo seja a mesma, optei por manter em arquivos distintos para conseguir realizar testes e depurações com mais facilidade. Além disso, criei um arquivo `constants.py` para guardar as constantes definidas para esse trabalho prático, como por exemplo o valor da sequência de sincronização, tamanho máximo de dados em um pacote, entre outras constantes.

O código "main" do nosso enlace ficará no arquivo `dcc023c2.py`, inicializando e executando a ponta ativa ou passiva mediante os parâmetros passados via linha de comando. Para executar esse código, basta seguir digitar `python dcc023c2.py` na linha de comando e fornecer em seguida os argumentos apropriados para a inicialização do servidor e do cliente.

4.2 Testes e verificações

Tanto o código do servidor quanto o do cliente foram testados em uma máquina local Linux versão Ubuntu 20.04 LTS. Foi desenvolvido testes "aleatórios" onde dada uma probabilidade um evento ocorre durante a transmissão, como por exemplo o primeiro campo SYNC da mensagem acaba sendo enviado com um erro. Esses testes não foram enviados juntamente com os arquivos do trabalho, uma vez que modificavam diretamente o funcionamento das pontas, porém, foi mantido alguns logs (comentados) utilizados para verificar a execução do fluxo de comunicação do enlace.

5 Conclusão

De forma geral, esse trabalho prático permitiu de forma bastante interessante a aplicação de toda a teoria vista em sala de aula através de um exemplo prático. Diferentemente do trabalho prático anterior, este foi bem mais desafiador, demandando muito mais tempo e estudo. Porém, mesmo não tendo a total garantia de ter implementado um sistema 100% correto como solicitado no enunciado, ele possibilitou uma aprendizagem mais profunda dos problemas encontrados durante a criação de uma rede, bem como de outros conceitos da disciplina.