Trabalho Prático 2
Transmissão de Arquivos

Arthur Souto Lima

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

1. Introdução

O projeto consiste em implementar um sistema de transmissão de arquivos entre servidor e cliente por meio de soquetes. O protocolo de transferência está definido na especificação, de forma que o controle dessa transmissão deve ser feita por um soquete TCP e arquivo em si deve ser enviado por um soquete UDP. Uma janela deslizante auxilia nesse processo e garante resistência a adversidades e a instabilidades da rede. O trabalho foi implementado em Python, utilizando sobretudo as bibliotecas *socket* e *threading*, bem como outras bibliotecas-padrão.

2. Implementação

2.1. Execução

O programa foi implementado em Python 3 (3.8). No código-fonte entregue, além dos módulos do cliente (*cliente.py*) e servidor (*servidor.py*), também há alguns módulos auxiliares. Tal como requerido pela especificação, podemos iniciar o programa usando os comandos abaixo. Convém ressaltar que pode ser necessário fornecer permissão de execução para esses dois scripts principais.

```
./servidor.py 51511
./cliente.py 127.0.0.1 51511 teste.txt
```

2.2. Resumo do Projeto

O passo inicial do projeto foi adaptar os conceitos e o código em C apresentado em [Cunha 2020] para a realidade do Python. Nessa etapa, muitos das implementações de [Verma 2020] foram adaptadas para servir de "*boilerplate*" para as implementações das funcionalidades seguintes.

Como recomendado pela especificação, o trabalho foi construído por partes. Inicialmente garantia-se a conexão servidor-cliente via TCP, em seguida adotava-se o multithreading do servidor para suportar vários clientes. Após isso, as mensagens básicas do protocolo definido na especificação foram implementadas. Então, o soquete UDP entre servidor e cliente foi implementado sendo seguido, finalmente, pelo envio do arquivo.

Quando o projeto estava funcionando em situações normais de rede, utilizando o *traffic control*, foram simuladas situações adversas, como taxas altas de perda, reordenação dos pacotes, pacotes duplicados e atrasos. O programa deveria funcionar mesmo em situações difíceis.

3. Funcionalidades

A seguir são discutidas algumas funcionalidades requeridas, bem como as respectivas soluções implementadas.

3.1. Janela Deslizante

A especificação requeria uma implementação de uma janela deslizante para gerenciar o envio dos pacotes do arquivo pelo cliente e seu recebimento pelo servidor. Foi implementado o paradigma *Go-back-N*, ou seja, caso algum não fosse recebido, toda a janela era reenviada e ela só avançava após uma certa quantidade de acks chegarem. Para o cliente, foi seguido o modelo visto nas aulas e também foi adaptado de [Bhargava 2017]. Para a janela deslizante do receptor (servidor), usou-se uma ideia parecida, como visto nas aulas.

Um detalhe sobre essa implementação é que ela leva em conta que os acks sempre chegam, visto que eles estão trafegando via TCP, que garante sua chegada, em ordem, sem erros. Assim, quando o servidor envia o acknowledge de um pacote, ele pode ter certeza que o cliente o receberá, pode demorar um pouco dada as condições da rede, mas chegará. O mesmo não pode ser garantido pelo UDP, por isso a janela deslizante.

Durante o envio do arquivo, pode acontecer de o cliente demorar muito para receber o ack de um pacote. Isso pode ter várias causas, como discutido na seção 3.5, por exemplo, o pacote de arquivo enviado nunca chegou ou o pacote de ack está atrasado. Há um timer no cliente sempre que uma janela é enviada, e, caso esse timer termine, é configurado um timeout e o cliente reenvia toda a janela, como prevê a estratégia *Go-back-N*.

3.2. Protocolo

Todas as mensagens do protocolo descritas na especificação foram implementadas via funções auxiliares. Assim, os detalhes de como cada pacote funcionava eram abstraídos do cliente e do servidor, de forma que estes apenas chamavam funções de *encode* e *decode* para cada tipo de mensagem para que uma função auxiliar, por exemplo, retornasse o pacote montado e pronto para enviar (*encode*) ou então retornasse as informações importantes contidas naquele pacote (*decode*). Para manejar os bytes das mensagens, utilizou-se a estrutura *ByteArray*, além de diversas conversões de tipos primitivos para bytes.

3.3. Gerenciamento do Arquivo

Com intuito de auxiliar no gerenciamento do arquivo, foram criados dois módulos: *FileDivider* e *FileAssembler*, respectivamente para dividir o arquivo em pacotes e para remontá-lo em disco a partir dessas partes. Um detalhe é que para evitar que arquivos muito grandes sejam subidos de uma só vez para a memória, o *FileDivider* só pega exatamente um intervalo de uma janela de pacotes do arquivo. Por outro lado, o *FileAssembler*, para evitar o mesmo, assim que a janela avança daquela posição específica, ele já escreve aquele pacote em disco, liberando memória alocada.

3.4. Multithreading

Tal como no Trabalho Prático 1, foi necessário o uso de threads para permitir o funcionamento correto do programa. Utilizando a biblioteca padrão *threading*, o servidor conseguia se comunicar com vários clientes concomitantemente e o cliente conseguia enviar os pacotes do arquivo via soquete UDP enquanto aguardava os acks advindos do servidor pelo soquete TCP.

3.5. Testes de Rede

O programa final foi submetido a uma série de testes de situações adversas da rede, como orientado pela especificação, utilizando a ferramenta *tc*. Quanto à reordenação dos pacotes e à duplicatas, a janela deslizante já garante resistência. Quanto à corrupção de pacotes, não foi notada alteração no funcionamento correto do programa, o que leva a concluir que o soquete UDP implementa algum método de detecção de erro, o que é citado na especificação do protocolo UDP. Apesar disso, essa condição deixava o tráfego deveras lento dado que muitos pacotes corrompidos tinham que ser retransmitidos.

Considerando atraso nos pacotes, *delays* de até $1s$ foram testados, sem afetar o envio de arquivos. O máximo que acontecia era a janela do cliente acusar um timeout de recebimento do acks, ou seja, não recebeu o ack de todos os pacotes daquela janela a tempo, e reenviar essa janela. Finalmente, quanto à perda de pacotes, novamente, a janela deslizante auxilia a mitigar os efeitos. Mas é importante notar que taxas de perda acima de 40% começam a afetar mesmo o soquete TCP, adicionando um atraso respeitável inclusive às mensagens mais simples com o Hello.

4. Desafios

Além das questões das várias funcionalidades, bem como as soluções adotadas, já discutidas anteriormente, convém ressaltar alguns outros desafios constantes na implementação do projeto.

Um primeiro que merece ser ressaltado é uma certa dificuldade em se depurar um programa em rede, tal como visto no TP1, já que normalmente são mais de um executando, pelo menos um servidor e um cliente e, no nosso caso, havia, ainda, o multithreading. Desse modo, como inclusive denotado na aula [Cunha 2020], registrar as diversas ações realizadas num log, se mostrou muito útil, já que cliente e servidor deveriam se comunicar via protocolo especificado e então gerenciar um envio de arquivo via uma janela deslizante.

O outro ponto é a programação em *sockets* e *threads* em outra linguagem. A experiência do TP1 certamente foi de muita utilidade para ser possível implementar a questão dos soquetes, agora um TCP e outro UDP, e a questão das threads.

References

Bhargava, S. (2017). Go_back_n_protocol. [1]. GitHub repository, Acesso em: 08 fev 2021.

Cunha, I. (2020). Introdução à programação em redes. [2]. Playlist no Youtube, Acesso em: 04 jan 2021.

Verma, K. (2020). Socket programming in python. [3]. Acesso em: 08 fev 2021.

[1] https://github.com/Bhargavasomu/Go_Back_N_Protocol

[2] https://www.youtube.com/playlist?list=PLyrH0CFXIM5Wzmbv-lC-qvoBejsa803Qk

[3] https://www.geeksforgeeks.org/socket-programming-python/