Trabalho Prático 3 – Redes de Computadores

Comunicação Interplanetária

Daniel Souza de Campos – 2018054664 – 2021/2

Sumário

1 – Objetivo

O objetivo do trabalho é desenvolver três programas que se comunicarão simulando uma troca de mensagens interplanetária. Os três programas são: Emissor, Exibidor e Servidor, cada um deles com uma função específica.

O Servidor deve ser responsável por aceitar conexões dos outros dois programas, também conhecidos como clientes, e gerenciar a comunicação entre eles. Ele deve funcionar com base em comunicação TCP/IP, utilizando a biblioteca *socket* do *Python* e ter sua estrutura montada ao redor da função *select* para monitorar os *sockets* dos clientes.

O Emissor deve ser capaz de receber comandos do usuário, repassá-los para o servidor e, opcionalmente, se conectar a um Exibidor. O Exibidor funciona como um chat ao apenas mostrar mensagens que chegam para ele sendo essas de um Emissor associado ou de outro Emissor que, eventualmente, realize uma comunicação em *broadcast*.

2 – Ferramentas para o trabalho

O trabalho foi desenvolvido utilizando a linguagem Python 3.8.3 no editor de códigos *Visual Studio Code* em uma máquina Windows com 8GB de RAM e processador Intel i5 7 geração. O Github foi usado como repositório e controle de versões e ficará disponível em

https://github.com/Pendulun/planetaryComunnication após a data final de entrega do trabalho dia 14/02/2022.

3 – Instruções para Execução

Como o trabalho foi desenvolvido em Python, não havia a necessidade de um Makefile para o processo de compilação de arquivos. Entretanto, um pequeno Makefile foi criado para a execução caso queira usá-lo. Dentro desse Makefile pode-se ajustar a versão do Python: python ou python3 dependendo do sistema com a variável PYTHON. Dessa forma, as instruções de execução dos três programas são descritas a seguir.

3.1 – Execução do Servidor

Comando para execução em um terminal:

"python server.py <portaParaConexão>" ou "make server ARGS=<portaParaConexão>"

O Servidor deverá ser executado antes dos Exibidores e Emissores. Uma vez executando, o servidor irá imprimir uma mensagem indicando quais os possíveis nomes relativos ao ip que poderão ser usados para se conectar a ele.

Exemplo: "Listening connections on DESKTOP-7657Q79/169.254.89.8/localhost on port 5000". Dessa forma, poderão ser usados como parâmetro de <ipServer> qualquer uma das três opções apresentadas.

3.2 – Execução do Exibidor

Comando para execução em um terminal:

"python exhibitor.py <ipServer>:<porta>" ou "make exhibitor ARGS=<ipServer>:<porta>"

3.3 – Execução do Emissor

Comando para execução em terminal:

"python emitter.py <ipServer>:<porta> [idExibidor]" ou "make emitter ARGS='<ipServer>:<porta> [idExibidor]' "

O idExibidor é um parâmetro opcional. Se informado, um Exibidor que recebeu o mesmo ID deverá estar sendo executado, caso contrário, uma mensagem de erro aparecerá para o emissor. Esse ID poderá ser conhecido após a mensagem HI enviada por um exibidor e descrita na seção 4.1.

4 – Troca de Mensagens

Essa seção aborda questões sobre cada tipo de mensagem a ser trocada pelos três programas.

4.1 – HI

Ao iniciar a execução, os emissores e exibidores enviam uma mensagem de HI automaticamente para o servidor. Quando o servidor receber uma mensagem de HI, ele irá imprimir na tela: "received HI" e irá responder o cliente com o seu novo ID em uma mensagem de OK. Dessa forma, no lado do cliente, será impresso uma mensagem do tipo: "MY ID< 1234".

Dessa forma, um emissor deverá ver qual o ID atribuído ao exibidor por meio dessa mensagem na tela.

Se o Emissor passou como parâmetro um [idExibidor] inválido, ele irá receber uma mensagem de ERROR da resposta da mensagem HI e seu programa terminará. Se não, o emissor estará associado ao exibidor dono do id informado. Se o Emissor não informar nenhum id de exibidor, ele não será associado a nenhum exibidor e isso poderá afetar a execução de outros comandos.

4.2 – ORIGIN

Caso a mensagem HI seja enviada e respondida com sucesso, será necessário o usuário informar de qual planeta o cliente (emissor ou exibidor) está. Para tal, uma mensagem “WRITE ORIGIN PLANET>” aparecerá. Isso significa que está na hora de o usuário digitar o nome de um planeta. O nome do planeta deverá ser apenas uma palavra, ou seja, não poderá conter espaços.

Após digitar o nome do planeta e enviar, o servidor irá imprimir uma mensagem da forma: “Received EARTH from 1376” em que o número é o ID do cliente que enviou a mensagem. Além disso, o servidor responderá o cliente com uma mensagem OK ou de ERROR.

4.3 – KILL

Para enviar uma mensagem KILL, o emissor deverá digitar o comando KILL no terminal. Dessa forma, ao receber essa mensagem, o servidor irá verificar se o emissor possui algum exibidor associado. Se sim, o servidor irá enviar uma mensagem KILL para esse exibidor e, nesse caso, uma mensagem “< KILL” irá aparecer para ele. O emissor e o exibidor irão imprimir na tela a mensagem “Disconnecting from server!”. Além disso, o servidor irá imprimir na tela a mensagem “Received KILL from 1376” em que o número é o ID do emissor que enviou o comando KILL. Se a qualquer momento um cliente se desconectar, via comando KILL ou ctrl+c, por exemplo, o servidor irá imprimir algo como “closing ('127.0.0.1', 49855)”.

4.4 – MSG

Um emissor poderá enviar uma mensagem para um outro cliente de acordo com o seguinte comando: MSG <idCliente> <mensagem>. O idCliente deverá ser um id válido, caso contrário, o servidor irá responder o emissor com uma mensagem de erro. Se esse id for o número 0, todos os exibidores irão receber a mensagem. Se esse id for o id de um exibidor, apenas esse exibidor irá receber a mensagem. Se esse id for o de um emissor, a mensagem será redirecionada pelo servidor ao exibidor associado a esse emissor. Caso esse emissor não tenha um exibidor associado, o emissor de origem receberá um OK mas a mensagem não será de fato enviada para um exibidor.

4.5 – CREQ/CLIST

Um emissor poderá enviar uma mensagem CREQ para enviar os ids de todos os clientes para um cliente específico via comando: CREQ <idCliente>. Da mesma forma que o comando

MSG, o idCliente deverá ser um id válido e, caso contrário, o servidor irá responder o emissor com uma mensagem de erro.

Primeiramente, se esse id não for um inteiro de tamanho máximo representado por 2 bytes, uma mensagem “SOMETHING WRONG WITH THE COMMAND! Usage: CREQ <destinyID>” irá aparecer. Essa é uma verificação do lado do cliente. Se o id for um inteiro correto, mas não for um id válido do lado do servidor, então uma mensagem “ERROR! SOMETHING WENT WRONG!” deverá aparecer.

Se o id for um id válido, então o servidor irá imprimir algo como “Received CREQ from 3165 to 3165”. Se o id for de um exibidor, então o servidor irá enviar uma mensagem CLIST para esse exibidor contendo a lista de ids dos clientes. Nesse caso, o exibidor irá imprimir algo como “CLIST< '6860 3165'”. Se o id for o id de um emissor e ele possuir um exibidor associado, então a mensagem CLIST será enviada para esse exibidor. Caso ele não possua um exibidor associado, o servidor irá responder o emissor com uma mensagem OK mas o CLIST não será enviado para nenhum exibidor.

4.5 – PLANET

Um emissor poderá enviar uma mensagem PLANET para descobrir o planeta de um cliente via comando: PLANET <idCliente>. Também existe uma verificação do idCliente do lado do cliente e do servidor. Caso o idCliente não exista, o servidor enviará uma mensagem de erro de volta ao emissor. Caso o idCliente seja de um cliente válido, mas o emissor que enviou a mensagem não possuir um exibidor associado, o servidor irá responder o emissor de origem com um erro. Caso o idCliente seja de um cliente válido e o emissor de origem possua um exibidor associado, o servidor irá redirecionar a mensagem de resposta para esse exibidor.

4.6 – PLANETLIST

Um emissor poderá enviar uma mensagem PLANETLIST para receber os planetas de todos os clientes via comando: PLANETLIST. Caso o emissor possua um exibidor associado, a resposta do servidor irá ser direcionada a esse exibidor. Caso contrário, o servidor enviará uma mensagem de erro para o emissor de origem.

5 – Sobre o código de implementação

O arquivo *common.py* possui algumas classes importantes. A primeira delas é a classe *Communicator*. Essa é a classe pai tanto da classe *Client*, também definida em *common.py*, e a classe *Server* definida em *server.py*. A classe *Communicator* apenas possui algumas constantes que são usadas tanto por clientes quanto pelo servidor.

A classe *Client*, definida em *common.py*, é a classe pai da classe *Emitter*, definida em *emitter.py*, e da classe *Exhibitor*, definida em *exhibitor.py*. A classe *Client* possui alguns atributos comuns para todos os clientes e também todo o processo de inicialização para se conectar ao servidor, enviar a mensagem HI e a mensagem ORIGIN.

A classe *BaseHeader*, definida em *common.py*, possui o *header* básico de uma mensagem. Ela é capaz de codificar e descodificar uma mensagem que possua o cabeçalho. Especificamente, é ela quem produz os *bytes* a serem enviados pelas mensagens que só precisam do *header* como OK e ERROR.

A classe *Parameter2BMessage*, definida em *common.py*, representa uma mensagem que, além do cabeçalho base, possui um parâmetro a mais de 2 *bytes* e uma mensagem após esse parâmetro. Essa classe que codifica e descodifica mensagens como MSG, PLANET e PLANETLIST.

A classe *Exhibitor*, definida em *exhibitor.py*, é a classe que representa um exibidor. Ela herda da classe *Client* e trata todas as mensagens que chegam do servidor.

A classe *Emitter*, definida em *emitter.py*, é a classe que representa um emissor. Ela herda da classe *Client* e é responsável por tratar as entradas do usuário, enviar mensagens para o servidor e esperar suas respostas.

A classe *Server*, definida em *server.py*, é a classe que representa o servidor. Ela herda da classe *Communicator* e é responsável por: gerenciar a conexão de clientes e suas informações como planeta de origem, identificador e exibidor associado caso o cliente seja um emissor, receber mensagens de todos os clientes, tratá-las e enviar respostas de volta e gerar identificadores para os clientes. Como requisitado na especificação, o servidor funciona utilizando a função *select* para gerenciar as mensagens que chegam e que devem ser enviadas para os *sockets* tanto do próprio servidor quanto dos clientes. A estrutura base do tratamento de mensagens que chegam e que devem ser enviadas foi criado seguindo aquela descrita em https://pymotw.com/3/select/index.html. Dessa forma, o código geral do servidor pode se parecer um pouco com os exemplos desse link.

6 – Conclusão

Este trabalho tinha como objetivo desenvolver três programas que simulassem três partes importantes de um sistema de troca de mensagens: Servidor, Emissor e Exibidor. Além disso, esses programas deveriam funcionar sob TCP/IP com a biblioteca *socket* em Python e a função *select*. Pode-se dizer que esse objetivo foi alcançado visto que o sistema implementado atende a todos os requisitos de tipos de mensagens a serem trocadas entre os programas clientes e servidor.

Em todas as partes do sistema, estruturar as mensagens a serem enviadas e recebidas foi o primeiro passo e, portanto, a primeira dificuldade geral. Uma vez que isso foi definido, ao implementar as classes *BaseHeader* e *Parameter2BMessage*, o tratamento de mensagens a serem enviadas e *bytes* recebidos não foi mais um problema.

Não houve muita dificuldade em implementar a estrutura de recebimento e envio de mensagens por parte do servidor visto que, como já dito antes, os exemplos em https://pymotw.com/3/select/index.html foram muito bem explicados, o que facilitou o seu entendimento e adaptação para esse trabalho. A principal dificuldade por parte do servidor foi o gerenciamento das estruturas de dados que armazenavam informações sobre os clientes como identificador, *socket* associado e planeta de origem.

Não houveram dificuldades significativas na implementação do exibidor dado que ele possui apenas uma estrutura de repetição simples que trata mensagens recebidas pelo servidor e, ocasionalmente, o responde. O mesmo aconteceu para o emissor já que ele também possui uma estrutura de repetição que trata as mensagens digitadas pelo usuário e envia mensagens para o servidor.

Para finalizar, com este trabalho, foi possível praticar a implementação de um sistema de troca de mensagens entre um servidor e, nesse caso, dois tipos de clientes diferentes seguindo um protocolo de aplicação pré-estabelecido, mas com espaço para adaptações e outras definições específicas de funcionamento. Dessa forma, agregou em muito ao meu conhecimento visto que me obrigou a pesquisar sobre o funcionamento de um servidor e que me fez decidir regras não antes estabelecidas que pudessem influenciar na performance e manutenção geral do sistema.