UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE FLORESTAL
CIÊNCIA DA COMPUTAÇÃO

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)
VITOR HUGO (3049)

SISTEMAS DISTRIBUÍDOS

TRABALHO PRÁTICO 1

FLORESTAL, MG
17 de novembro de 2020

ATHENA SARANTÔPOULOS (2652)
JOSE GRIGORIO NETO (3046)
VITOR HUGO (3049)

SISTEMAS DISTRIBUÍDOS
TRABALHO PRÁTICO 1

Documentação do 2º trabalho prático de Sistemas Distribuídos que tem como objetivo realizar uma implementação de uma API de Sockets

Orientador: Profa. Dra. Thais Regina de Moura Braga Silva

FLORESTAL, MG

17 de novembro de 2020

Resumo

Este trabalho tem como objetivo realizar uma implementação de uma API Sockets, configurando como o uso de visão de processos para os elementos arquitetônicos.O sistema distribuído que será implementado possui o modelo de arquitetura cliente-servidor. Onde o cliente entrará em contato com 3 servidores: servidor aeroporto, servidor hotel e servidor passeio, gerando assim um sistema distribuído Agência de Turismo.

Palavras-chaves: sistema distribuído, API Sockets, cliente-servidor.

Abstract

This work aims to carry out an implementation of an API Sockets, configuring as the use of process view for the architectural elements. The distributed system that will be implemented has the client-server architecture model. Where the customer will contact 3 servers: airport server, hotel server and tour server, thus generating a distributed Tourism Agency system.

Key-words: distributed system, API Sockets, client-server.

Sumário

1 Introdução

Este trabalho é dividido em três partes. Essa documentação foi realizada para a implementação da primeira parte, que consiste em utilizar a API de Sockets, configurando o uso da visão de processos para os elementos arquitetônicos. O trabalho poderia ser implementado na Linguagem C, Java ou Python. O grupo resolveu utilizar a linguagem Python, por ser mais fácil a aprendizagem de Sockets e de interface gráfica.

O sistema distribuído possui o modelo servidor-cliente, onde possui um cliente e três servidores. O cliente então, precisa entrar em contato com o servidor passagem aérea, hospedagem e passeios, gerando assim um pacote turístico. Ao longo dessa documentação iremos dissertar sobre a escolha de cada parte desse processo de implementação.

2 Como executar

Para executar o sistema criamos um shell simples para automatizar algumas tarefas de verificações de arquivos e a inicialização dos três servidores e os 2 clientes de teste. Abaixo será especificado algumas operações necessárias antes de executar o bash, como também sua execução.

Primeiramente, se for desejado rodar os sistemas em outro dispositivo será preciso liberar as portas que será usada para se comunicar com os sistemas, como usaremos um servidor de client side, não precisaremos nos comunicar com os servidores, assim será preciso liberar apenas as portas que usaremos para teste no client side as 5000 e 5001.

Antes de rodar o shell é preciso ter instalado especificamente o python 3.7, assim caso esteja no linux basta abrir o terminal e executar o comando:

```
$sudo apt-get python3.7
```

Após a instalação do python que será a nossa linguagem usada para programar o sistema, temos que baixar algumas dependências deste, como o caso do gerenciador de pacotes python3-venv, assim para baixar este gerenciador basta executar o comando:

```
sudo apt-get install python3-venv
```

Agora o sistema estará pronto para rodar o shellscript, este está dividido em 4 partes que será explicado em seguida.

```
#!/bin/bash
trap ctrl_c INT

function ctrl_c() {
  fuser -k 5050/tcp
  fuser -k 5051/tcp
  fuser -k 5052/tcp
  fuser -k 5000/tcp
  fuser -k 5001/tcp
  echo "$(tput setaf 5)$(tput bold)I catch U!!!$(tput sgr0)"
}

if [ ! -f tickets/passeios.JSON ] || [ ! -f tickets/aeroportos.JSON
   ] || [ ! -f tickets/hospedagens.JSON ]; then
  echo "$(tput setaf 3)$(tput bold)Gerando mocks$(tput sgr0)"
  cd tickets &&
  python3 ticketsGenerator.py
```

```
17    cd ..
18  fi
19
20  if [ ! -f venv ];then
21    echo "$(tput setaf 3)$(tput bold)Criando venv$(tput sgr0)"
22    python3 -m venv venv
23    source venv/bin/activate
24    pip3 install flask
25  fi
26
27  source venv/bin/activate
28
29  #Verificar se há python 3.7
30  python3.7 ticketServer.py &
31  python3.7 hospServer.py &
32  python3.7 passeioServer.py &
33  python3 main.py -p 5000 &
34  python3 main.py -p 5001
```

O primeiro bloco tem por função criar um listen para quando o usuário desejar terminar os processos dos sistemas(Servidores e Clientes), assim a função crtl_c finaliza os processos relacionados as portas abertas no uso dos sistemas.

O segundo bloco realiza a verificação de três arquivos que são os mocks gerados para as passagens de avião, hospedagem e passeios, assim esse bloco verificar se esse mocks já foram gerados, caso contrário ele executa um programa também em python ticketsGenerator, para gerar esse mocks.

O terceiro bloco verifica se o ambiente virtual que será usado para rodar os sistemas já está criado e com as devidas dependências, caso contrário ele realiza a criação de um ambiente virtual e instala as dependências necessárias, como o caso do Flask.

Por fim, o quarto bloco inicializa os três servidores e os dois clients side, um na porta 5000 e outro na porta 5001.

3 Implementação

O sistema foi criado visando os conteúdos visto em sala de aula, dessa forma utilizamos o paradigma cliente-servidor com 3 camadas demonstradas na figura abaixo:

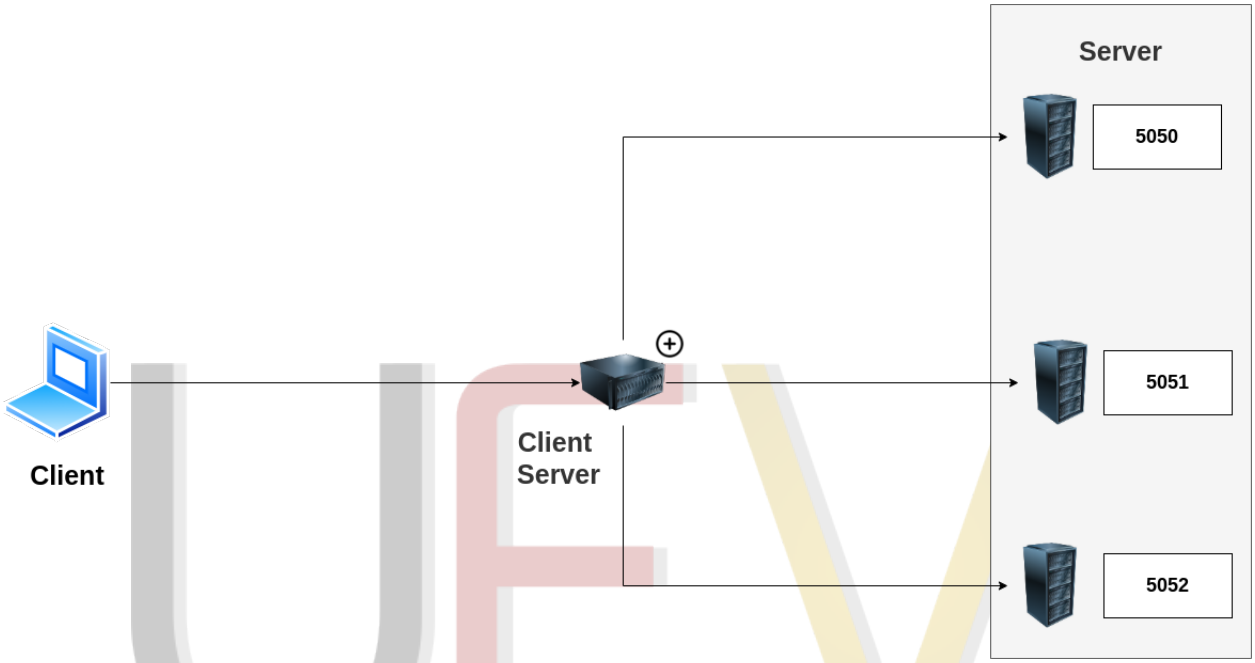

Figura 1 – Logo da linguagem

Da direita para esquerda temos os três servidores, dos quais a porta 5050 corresponde ao servidor do voos, a porta 5051 ao servidor de hospedagem e o 5052 ao servidor de passeio.

O client server, será responsável por controlar a interface e conexão com os usuário. Nessa imagem está demonstrado com um icon de plus, pois será possível criar mais de um instância deste, com diferentes portas e cada instância será uma nova conexão no server, posteriormente discutiremos mais sobre essa decisão de projeto.

O client por sua vez representa o cliente que sempre fará uma conexão no client server para poder acessar ao sistema web.

3.1 Hierarquia de pastas

O projeto é composto por 5 pastas e 7 arquivos na raiz. As 5 pastas são o socket que possui 2 arquivos contendo as classes de abstração do socket, a static que possui arquivos de imagens e css usados no html(template do Flask), o template que possui os

Templates das páginas web, o tickets que possui os arquivos jsons gerados pelo gerador que também está nessa pasta e a última pasta a venv que é o ambiente virtual do python.

Os 7 arquivos principais são os:

- hospServer : Server que gerencia as hospedagens;
- main.py: Inicializa uma nova conexão com os server e uma nova instância do client server(flask);
- passeioServer : Server que gerencia os passeio;
- ticketServer.py : Server que gerencia os voos;
- run.sh : shell script que realiza algumas verificações e inicializa os servidores e o client server;
- run_clientSide : Executa duas instâncias do client server em portas diferentes;
- run_server : Executa os 3 servidores.

3.1.1 Flask

3.1.2 Servidores

3.1.3 Geradores de Mock

3.2 Detalhes Gerais

Como realizamos a implementação de um sistema web foi necessário criar um client server(server web) que armazena a instância do mesmo, consideramos que cada instância desse server web seria como um cliente na visão dos servidores.Foi pensado isso, pois facilitaria a escalabilidade do sistema, de forma que poderíamos abrir um porta nova no servidor web para cada quantidade x de usuários, dessa forma não haveria o risco de sobrecarregar um server web já que quando ele estivesse muito ocupado bastaria criar um nova instância em uma nova porta que já se conectaria aos servidores e os novos clientes seriam redirecionados para essa nova instância.

3.3 Python Socket

Para esse trabalho foi utilizado o Python Socket, ele é um biblioteca do Python. Devemos levar em consideração que o Socket não é uma linguagem de programação e sim funções do próprio Sistema Operacional, por isso alguns comportamentos dependem da plataforma utilizada.

No Python Socket a função **socket** retorna um objeto socket cujos métodos implementam várias chamadas do sistema de socket. A primeira operação a ser criado é o socket. Ele cria o socket que recebe como parâmetros a família de endereço, onde o padrão é o AF_INET que corresponde ao IPV4 e o SOCK_STREM que corresponde ao protocolo TCP. A segunda operação é o bind que liga o socket a um endereço. Como foi utilizado o tipo de família de endereço IPV4, é preciso passar como parâmetro para o bind o endereço IP e a porta. Ao longo da documentação iremos falar mais sobre essas operações e outras que foram utilizadas.

3.4 Servidores

3.4.1 Classe EasySocketServer

A classe easySocketServer foi criada com o intuito de abstrair e facilitar o uso de sockets para os servidores criados. Definimos como padrão o tamanho do header como 10, assim toda mensagem antes de ser recebida envia um header com 10 bytes informando qual será o tamanho da mensagem a ser recebida, dessa forma podemos receber uma mensagem de até 1000000000 bytes, sem haver truncamento. Outra definição dessa classe foi o IP sendo 0.0.0.0, pois dessa forma toda rede local consegue se conectar ao servidor através desse IP.

No inicializador da classe se recebe como parâmetro a porta ao qual o socket irá se conectar, assim se cria uma instância do socket passando como parâmetros as opções socket.AF_INET e socket.SOCK_STREAM, que corresponde a utilização de IP e o protocolo TCP. Após isso, se passa como opções a possibilidade de se reusar um address que já está em uso, escolhemos por usar essas opções para caso tenhamos que resetar o client server não haveria conflito de portas em uso. Por fim, com a instância do socket criado, passamos o IP e a PORT e abrimos o server para ficar em modo escuta.

Essa classe possui três métodos que farão a escuta do server para novas conexões, o tratamento de mensagens recebidas e tratamento de mensagens que serão enviadas, na sequência temos listen, recive_message e send.

O listen é uma função async, devido a sua natureza de ficar na espera de um cliente ou de algo que algum cliente envie, todavia poderíamos contornar isso com o uso de threads, uma para receber os cliente e outra para tratar das mensagens de um cliente. Como já dito, esse método é responsável por tratar o recebimento de novos clientes e também de escutar as mensagens enviados por esse cliente. Para toda mensagem recebida é realizado um tratamentos de serialização da mensagem. Esse tratamento é feito no método recive_menssage, caso haja algum erro essa função retorna False e o listen também retorna um False para o programa, sinalizando que ocorreu algum error.

O send realiza praticamente o mesmo trabalho que o recive_menssage, porém ao contrário, de forma a preparar o header e serializar a mensagem para o envio.

3.4.2 Gerador de Mock

Para testar os nossos servidores e popular o banco de dados da aplicação, foi criado um pequeno script em Python que gera objetos de passagens, hospedagens e passeios com dados aleatórios que são armazenados em seus respectivos arquivos .json. Esses objetos são formatados em JSON e suas estruturas, que são as seguintes:

3.4.2.1 Passagem

```
[{
    'horarioIda',
    'chegadaIda',
    'valor',
    'diaIda',
    'diaChegada',
    'deAeroporto',
    'paraAeroporto'
},...]
```

3.4.2.2 Hospedagem

```
[{
    'diaEntrada',
    'numero',
    'qtdQuartos',
    'qtdCamas',
    'valor',
    'nome',
    'endereco',
    'cidade',
    'cafeDaManha'
},...]
```

3.4.2.3 Passeio

```
[{
    'horario',
    'valor',
    'dia',
    'local',
```

```
6                 'endereco',
7                 'cidade'
8             },...]
```

3.4.3 Passagens

O servidor de Passagens aguarda o recebimento de um objeto JSON por parte de algum cliente, contendo as informações que o usuário necessita, no seguinte formato:

```
1         {
2             'dataIda',
3             'dataVolta',
4             'paraAeroporto',
5             'numeroPassagens'
6         }
```

Estes campos simbolizam a data da passagem de ida e data da passagem de volta, qual o aeroporto de destino e o número de passagens que o usuário deseja comprar. Para simplificar o algoritmo, foi decidido utilizar um formato de agência de viagens local, onde o aeroporto de partida sempre será o mesmo, neste caso, o aeroporto de partida é o Aeroporto Internacional de Confins - Tancredo Neves.

Quando o servidor recebe o JSON acima, é feita uma filtragem entre os dados que estão armazenados em 'passagens.json' para retornar ao cliente apenas as passagens que estão dentro destes parâmetros, caso o cliente confirme a compra de alguma das passagens listadas, é feita outra chamada ao servidor que retorna ao cliente a passagem escolhida.

3.4.4 hospedagens

O servidor de hospedagens aguarda o recebimento por parte de algum cliente um objeto JSON contendo as informações que o usuário necessita, no seguinte formato:

```
1         {
2             'dataIda',
3             'cidade',
4             'numeroDePessoas'
5         }
```

Estes campos simbolizam a data de início da hospedagem, qual a cidade onde o usuário vai se hospedar e o número de camas esperadas.

Quando o servidor recebe o JSON acima, é feita uma filtragem entre os dados que estão armazenados em 'hospedagens.json' para retornar ao cliente apenas as hospedagens que estão dentro destes parâmetros entre vários hotéis e pousadas diferentes naquela

cidade específica, caso o cliente confirme a locação de alguma hospedagem, é feita outra chamada ao servidor que retorna ao cliente os dados da hospedagem escolhida.

3.4.5 Passeios

O servidor de Passeios aguarda o recebimento de um objeto JSON por parte de algum cliente, contendo as informações que o usuário necessita, no seguinte formato:

```
1        {
2            'dataIda',
3            'cidade'
4        }
```

Estes campos simbolizam a data em que se é desejado participar de um passeio e qual a cidade onde o passeio irá ocorrer.

Quando o servidor recebe o JSON acima, é feita uma filtragem entre os dados que estão armazenados em 'passeios.json' para retornar ao cliente apenas os passeios que estão dentro destes parâmetros, caso o cliente confirme a presença em algum dos passeios listados, é feita outra chamada ao servidor que retorna ao cliente o passeio desejado.

3.5 Cliente

3.5.1 EasySocketClient

A classe EasySocketClient foi criada com o intuito de facilitar o uso do socket por parte dos clientes, de forma semelhante a vista no easySocketServer. Essa classe possui algumas definições padrões como é o caso do tamanho do header, que será usado para enviar o tamanho da mensagens e o IP que será usado para identificar a máquina de destino(server).

Seu inicializador recebe um porta como parâmetro, essa porta é referente ao server que será feito a conexão, assim ele inicializa um socket passando os parâmetros socket.AF_INET e socket.SOCK_STREAM definindo o uso do IP e o protocolo TCP e em seguida realiza a conexão com o server de destino. Definimos também que ele não será bloqueante,ou seja, não haverá um time out. E por fim mandamos a primeira mensagem de confirmação.

Essa classe possui dois métodos, no qual um irá realizar o envio e recebimento das mensagens e outro que irá realizar o fechamento das conexões com o server, estes são o send e close.

O send realiza a serialização da mensagem que será enviada e após enviar ele fica na espera escutando a resposta do server para também serializar e realizar o retorno da

mensagem.

3.5.2 Flask

Flask é um micro-framework web escrito em Python, ele é baseado na biblioteca Jinja2 e WSGI Werkzeug. Micro-framework são Frameworks que possuem estrutura mais simples que um Framework convencional. O Flask é usado para criações de sites básicos que contém pequenas aplicações com requisitos mais simples. Por esse motivo, o grupo escolheu usar o Flask para a implementação de uma interface gráfica, já que sua utilização seria somente para passar os dados para os servidores e mostrar os dados retornados pelos mesmos.

Para a criação da aplicação em Flask foi necessário criar um ambiente virtual para gerenciar as dependências do projeto. Para isso foi utilizado o módulo venv que está dentro do pacote virtualenv do Python. Segue abaixo como realizar a criação e um ambiente virtual:

No sistema operacional Linux:

```
# Debian, Ubuntu
$ sudo apt-get install python-virtualenv

# CentOS, Fedora
$ sudo yum install python-virtualenv

# Arch
$ sudo pacman -S python-virtualenv
```

No sistema operacional Mac OS X ou Windows é necessário o download do get-pip.py.

No Mac OS X:

```
$ sudo python2 Downloads/get-pip.py
$ sudo python2 -m pip install virtualenv
```

No Windows, se utilizar Python 2.7:

```
> \Python27\python.exe Downloads\get-pip.py
> \Python27\python.exe -m pip install virtualenv
```

Depois da instalação do Virtualenv, é necessário criar e ativar seu ambiente virtual. Para a ativação do ambiente virtual, é necessário somente executar o shell script criado. Para criar o ambiente virtual segue abaixo o comando: No Linux e Mac OS X, dentro da pasta onde será feito o projeto Flask:

```
$ python3 -m venv venv #criação
```

No Windows:

```
1 > py -3 -m venv venv #criação
```

A instalação do Flask é realizada ao executar o shell script.

Para a interface gráfica foi utilizado templates do Flask junto com o Boostrap. O Boostrap é um framework para desenvolvimento de front-end para sites usando HTML, CSS e JavaScript.

Os Templates criados são visualizados através da função render_template(), como parâmetro é passado um arquivo HTML(HyperText Markup Language).

Para utilizar comandos do Python dentro do arquivo HTML é utilizado , ou seja, tudo que estiver dentro das chaves é executado como Python e não como HTML. Para o Python reconhecer os Templates é preciso criar uma pasta com o nome **templates**, com isso o Python o reconhece para ser renderizado pela função render_template.

Para arquivos como imagens, vídeos e estilos(CSS-Cascading Style Sheets), é necessário criar uma pasta **static** e para o Python reconhecer esses arquivos é preciso utilizar a função url_for. Essa função recebe como parâmetro o nome da pasta e o arquivo que será acessado pelo Python dentro do HTML.

Foram criados sete Templates, dois para Aeroporto, dois para Hotéis, dois para Passeio e um de erro. O primeiro Template para o Aeroporto serve para selecionar o aeroporto, o dia da ida, dia da volta e número de passagens. Ao selecionar essas informações, elas são mandadas para o servidor Aeroporto que retorna as passagens que poderão ser reservadas. Com isso, esse retorno as informações são mostradas no outro Template do Aeroporto. Ao enviar a reserva, a página é redirecionada para selecionar informações do Hotel, que são: cidade, dia da chegada, dia da saída e quantidade de pessoas. Ao selecionar essas informações, elas são mandadas para o servidor Hotel que retorna as reservas que poderão ser escolhidas. Com esse retorno, as informações são mostradas no outro Template do Hotel. Ao selecionar o Hotel desejado, a página é redirecionada para o Template do Passeio. Esse Template recolhe informações que são: cidade e dia da chegada. Logo após isso, essas informações são enviadas para o outro Template do passeio, onde será mostrado os passeios disponíveis e selecionados para a reserva.

Segue abaixo imagens do resultado da criação desses seis Templates:

Figura 2 – Template Aeroporto para passar os dados

Figura 3 – Template Aeroporto para mostrar os dados

Figura 4 – Template Hotel para passar os dados

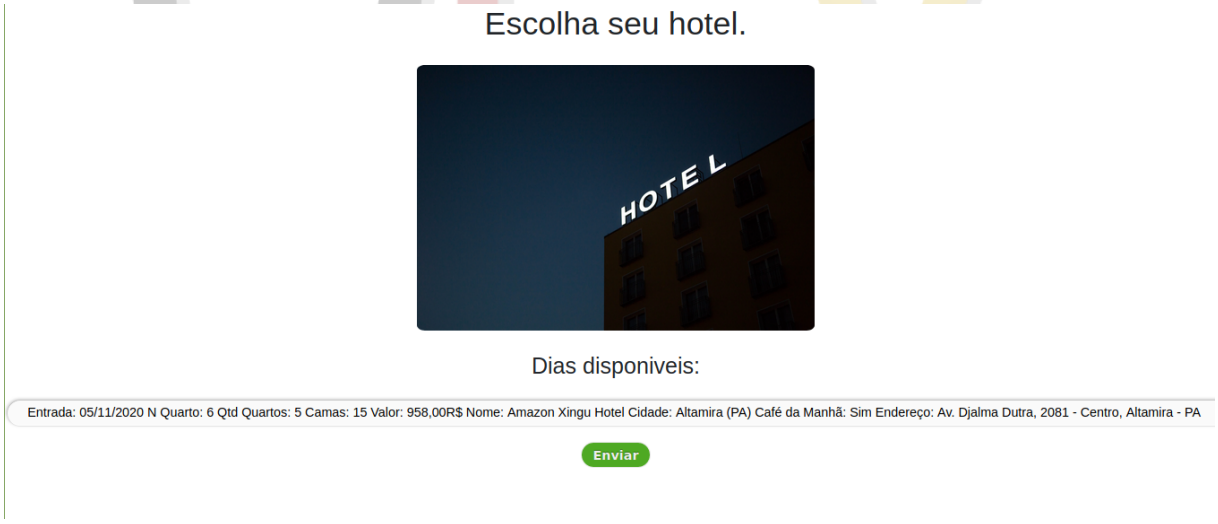

Figura 5 – Template Hotel para mostrar os dados

Figura 6 – Template Passeio para passar os dados

Figura 7 – Template Passeio para mostrar os dados

Figura 8 – Template de erro

3.6 Testes

Os teste foram realizado em três máquinas diferentes, uma executaria os servidores(192.168.1.11), outra o client server(192.168.1.3) e um terceira(192.168.1.10) que faria o papel do cliente. Para fins demonstrativos, foram criados duas instâncias do client server na máquina que executará o client server, dessa forma os servidores terão que lidar com dois clientes que seriam as instâncias web criadas. Uma dessas instâncias seria conectada em uma terceira máquina e outra será conectada na máquina de server, e em ambas serão realizados procedimentos em paralelo. Abaixo segue o teste em mais detalhes:

Na imagem a seguir, se é visto o server sendo iniciado onde cada "listening for connections on ...", representa que os servidores estão prontos para conexão. Além disso, vemos as duas instâncias do client server sendo conectadas a cada um dos servidores.

```
vitor in Agencia_de_Turismo_TPSD on  master [✘!?]
❯ /usr/bin/zsh /home/vitor/Desktop/Facu/PER/SIs\ Distribuidos/Agencia_de_Turismo_TPSD/run_servers.sh
Listening for connections on 0.0.0.0:5051...
Listening for connections on 0.0.0.0:5052...
Listening for connections on 0.0.0.0:5050...
New connection in Voos
Welcome 192.168.1.3:56667
New connection in Voos
Welcome 192.168.1.3:56668
New connection in Hoteis
Welcome 192.168.1.3:56669
New connection in Hoteis
Welcome 192.168.1.3:56670
New connection in Passeios
Welcome 192.168.1.3:56671
New connection in Passeios
Welcome 192.168.1.3:56672
```

Figura 9 – Clientes conectando

No lado do client server se ver as duas instâncias sendo criadas com sucesso e também se conectando nos servidores como visto na imagem anterior. Além disso, é mostrado dois clientes se conectando, por essas duas instâncias estarem no mesmo console não é possível visualizar detalhadamente, mas o cliente 192.168.0.100 está se conectando a instância web da porta 5000 e o 192.168.0.102 está se conectando a instância web da porta 5001.

```
./run_clientSide.sh                                    ..o_TPSD-master
Buscar                                                                OK
vitor in Facu/Sist. Dist/Agencia_de_Turismo_TPSD-master
❯ ./run_clientSide.sh
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Running on http://0.0.0.0:5001/ (Press CTRL+C to quit)
192.168.1.10 - - [04/Nov/2020 20:09:16] "GET / HTTP/1.1" 200 -
192.168.1.10 - - [04/Nov/2020 20:09:16] "GET /style.css HTTP/1.1" 404 -
192.168.1.10 - - [04/Nov/2020 20:09:16] "GET /static/styles/style.css HTTP/1.1" 200 -
192.168.1.10 - - [04/Nov/2020 20:09:16] "GET /static/aviao.jpg HTTP/1.1" 200 -
192.168.1.10 - - [04/Nov/2020 20:09:17] "GET /favicon.ico HTTP/1.1" 404 -
192.168.1.11 - - [04/Nov/2020 20:09:55] "GET / HTTP/1.1" 200 -
192.168.1.11 - - [04/Nov/2020 20:09:55] "GET /style.css HTTP/1.1" 404 -
192.168.1.11 - - [04/Nov/2020 20:09:55] "GET /static/styles/style.css HTTP/1.1" 200 -
192.168.1.11 - - [04/Nov/2020 20:09:55] "GET /static/aviao.jpg HTTP/1.1" 200 -
192.168.1.11 - - [04/Nov/2020 20:09:56] "GET /favicon.ico HTTP/1.1" 404 -
```

Figura 10 – Client Server

Assim já é possível se conectar ao sistema web, como será visto nas imagens a seguir foi conectado em duas máquinas diferentes com as diferentes portas.

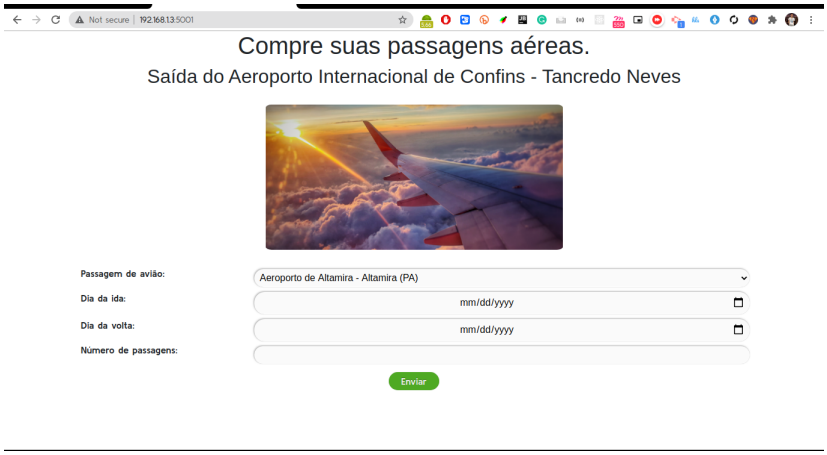

Figura 11 – Conexão com server web porta 5001

Figura 12 – Conexão com server web porta 5000

Por fim,foi realizado o teste de enviar uma requisição para o servidor de voos e como pode ser visto a mensagem foi recebida com sucesso e é exibida para o usuário.

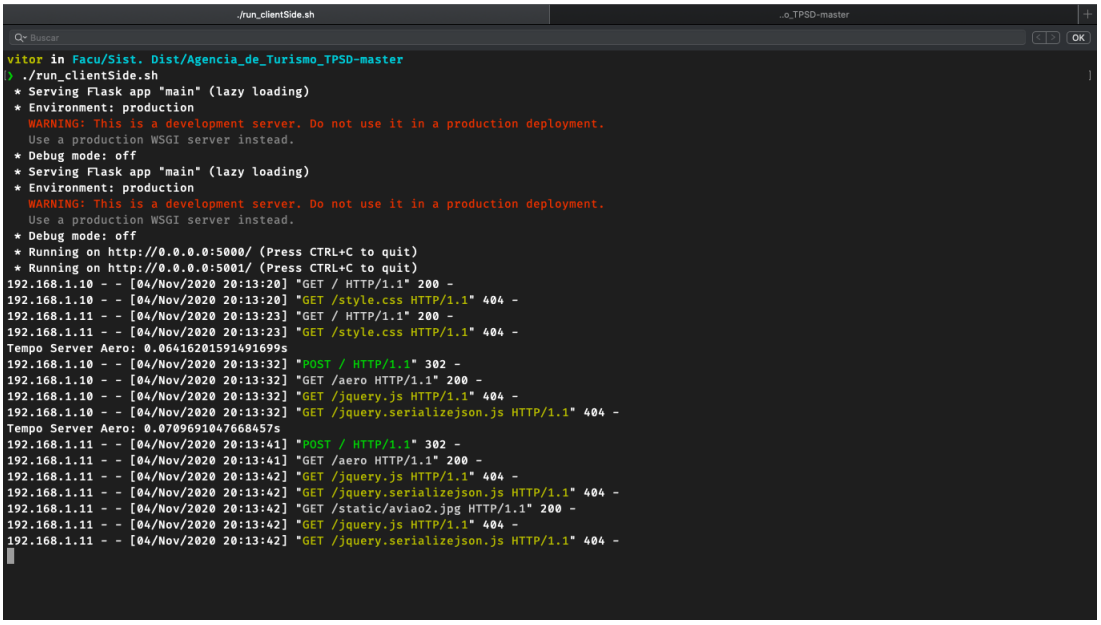

Figura 13 – Tempo de demora para resposta para o usuário no client Server

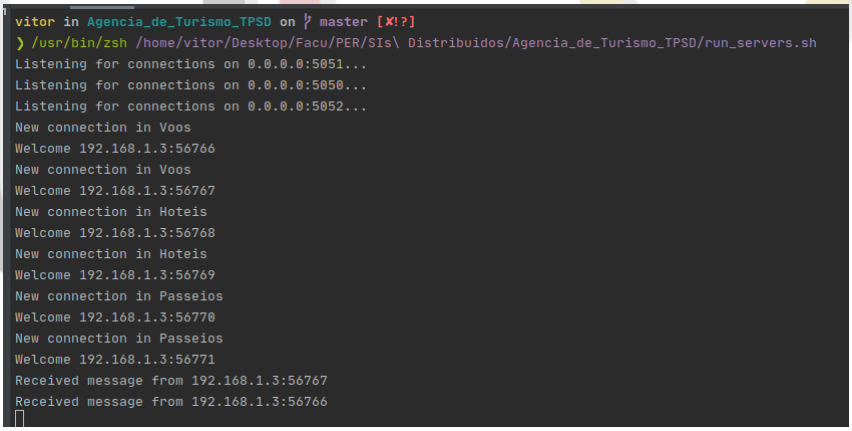

Figura 14 – Mensagens recebidas no servidor

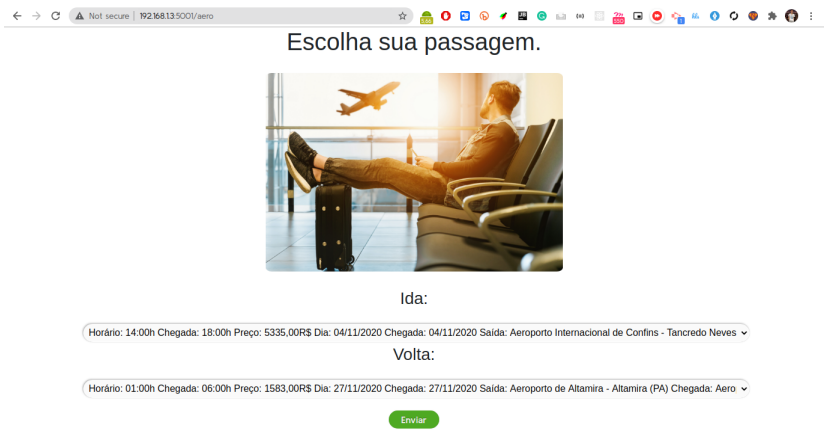

Figura 15 – Mostrando os resultados

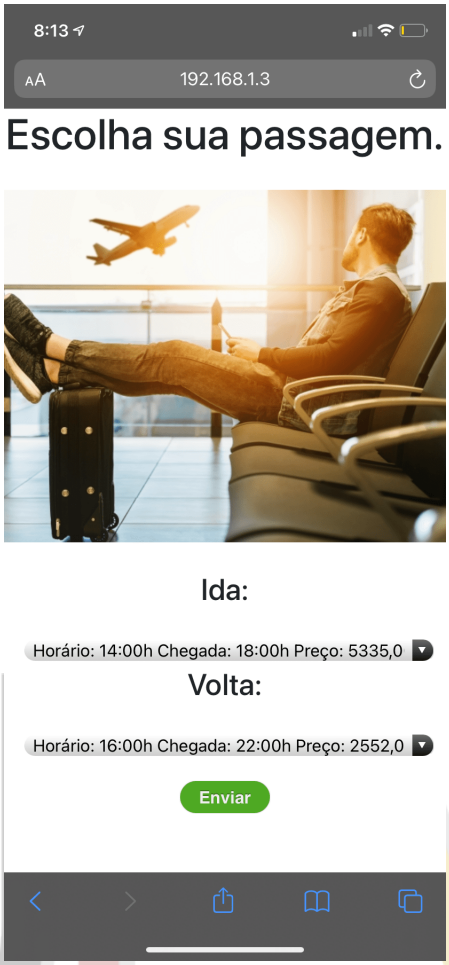

Figura 16 – Mostrando os resultados

As próximas etapas são um ciclo onde o usuário envia os dados para o client server e do client server para o servidor e assim a mensagem é retornada entre eles até chegar ao usuário novamente. Não será mostrado todo o fluxo, pois basicamente o que seria o fluxo principal já foi mostrado de forma que o que mudaria seria as telas e as mensagens enviadas entre os cada requisição.

3.7 Resultados obtidos

Foi calculado uma média de tempo de espera para cada requisição em um total de 5 teste e a média dos resultados é mostrado a seguir:

- Tempo de espera para o servidor de Voos: 0.06756556034088135s
- Tempo de espera para o servidor de Hospedagem: 0.009750843048095703s
- Tempo de espera para o servidor de Passeio: 0.018122196197509766s

Os resultados mostram um tempo relativamente pequeno, isto pode ser devido ao fato de as mensagens entre o client server e o server serem relativamente curtas, além também de todos os dispositivos estarem conectados em uma mesma rede, assim não é possível simular um real atraso de uma conexão com um servidor que está a uma distância consideravelmente longe. Um último ponto a acrescentar é que não foi contabilizado o tempo de resposta para se enviar as páginas HTML com os dados, uma vez que esse não era o enfoque do trabalho.

4 Conclusão

Ao longo deste trabalho, pudemos ver em uma aplicação real alguns dos conceitos de sistemas distribuídos apresentados em aula, e também pudemos aprender mais sobre o uso da linguagem Python em servidores e sobre o uso de Flask como um micro-framework web.

Além disso, criar esses servidores junto ao cliente e implementar nossa própria troca de mensagens entre essas entidades serviu como uma ótima revisão da matéria atual da disciplina.

5 Referências

MCMILLAN, Gordon. HOWTO sobre a Programação de Soquetes. Python. Disponível em: <https://docs.python.org/pt-br/3/howto/sockets.html>. Acesso em: 23 de out. de 2020.

Flask. Disponível em: <https://flask.palletsprojects.com/en/1.1.x/>. Acesso em: 25 de out. de 2020.

Bootstrap. Disponível em: <https://getbootstrap.com/>. Acesso em: 25 de out. de 2020.

ANDRADE,Ana Paula de. O que é Flask?. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-flask/>. Acesso em: 25 de out. de 2020.

Flask. Disponível em: <https://flask.palletsprojects.com/en/1.1.x/installation/#install-instal
Acesso em: 25 de out. de 2020.