Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 3
Desenvolvimento de Servidor e Cliente HTTP
Criação de uma API REST

Nome: Paula Jeniffer dos Santos Viriato
Matrícula: 2015114240
Data: 11 de dezembro de 2018, terça-feira

1 Introdução

Neste trabalho foi implementado um par cliente-servidor, e tal par utiliza chamadas de procedimentos remotos (RPC) em REST durante sua execução. Com esse par cliente-servidor, um sistema de consulta ao *PeeringDB* foi desenvolvido, sendo o *PeeringDB* um banco de dados de interconexões (peerings) entre redes-membro de IXPs (*Internet Exchange Points*). O sistema possui dois módulos, sendo eles:

(1) Um programa cliente, que faz requisições HTTP para um servidor, através de uma API REST, para capturar consultas ao banco *PeeringDB*;

(2) Um programa servidor, que atende as requisições, realizando as consultas.

O objetivo do trabalho era implementar os lados cliente e servidor da API descrita acima, para os dados do PeeringDB armazenados previamente em arquivos JSON, **mas sem utilizar a biblioteca do PeeringDB para acesso à API original**. Isso significa que os cabeçalhos HTTP deviam ser contruídos manualmente.

O sistema deve ter o seguinte comportamento, ilustrado pela figura 1, onde:

(1) O servidor inicia e carrega os dados dos arquivos JSON do *PeeringDB*;

(2) O servidor exporta um subconjunto dos dados via uma API REST (HTTP);

(3) O cliente consulta os dados do servidor através dos *endpoints* definidos;

(4) O cliente usa os dados recuperados do servidor para gerar algumas análises.

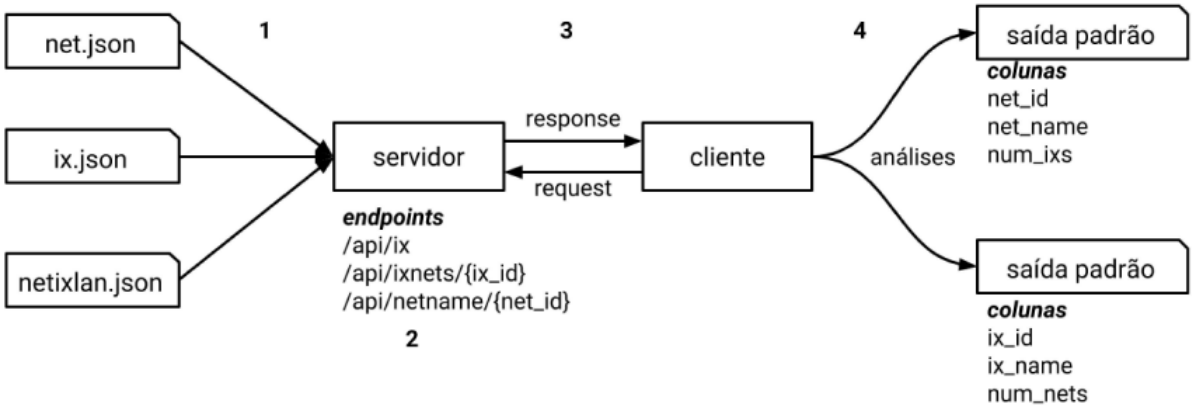

Figura 1 - Esquema do comportamento do sistema

É bom lembrar que um *endpoint* corresponde à um final de URL (*path*) que disponibiliza uma resposta ou um serviço. No exemplo da API original do *PeeringDB*, quando acessada a URL https://www.peeringdb.com/api/ix/1, o *endpoint* é /api/ix/1, e uma resposta de formato JSON é retornada.

2 Desenvolvimento

Os módulos foram desenvolvidos na linguagem *Python*, especificamente no *Python3.6.7*. Todo o processo de desenvolvimento foi realizado no ambiente *Jupyter Notebook*, disponibilizado pelo *Anaconda3 64-bit*. Os gráficos que serão apresentados neste relatório foram gerados pelo *Jupyter Notebook*. Os principais requisitos são:

(a) *Python3*. Comando de instalacao: sudo apt install python3.5

(b) *Pyinstaller*. Comando de instalacao: pip3 install pyinstaller

O código do programa desenvolvido neste trabalho prático está disponível no GitHub em: github.com/PaulaViriato/HTTP_Server_and_Client

2.1 Programa

A primeira tarefa do servidor implementado é carregar os arquivos de dados do *PeeringDB*, cujos caminhos são disponibilizados como parâmetros do programa. Estes arquivos estão no formato JSON, e a biblioteca do JSON disponibiliza métodos para tranformar este formato num dicionário *Python*, o que facilita o acesso. Os dicionário de cada arquivo ficam em variáveis globais.

Para o servidor foi criada uma classe chamada HTTP, que herda da classe BaseHTTPRequestHandler, e tem como objetivo manipular requisições do protocolo HTTP. Na classe HTTP criada, o método do_GET, que é chamado quando uma requisição do tipo GET é realizada, é sobreescrito, para que os *endpoints* sejam estabelecidos e suas requisições tratadas.

O módulo servidor HTTP é composto por 4 *endpoints*, sendo 3 os especificados pelo enunciado oficial do trabalho prático, e um outro criado pela a impossibilidade de conseguir um dado necessário para as análises, dados este que não pode ser adquirido com o que os outros 3 propõem, nem mesmo juntos. Todos os 4 *endpoints* são:

(1) **/api/ixids** : retorna os ids de todos os IXs armazenados;

(2) **/api/ixnets/[ix_id]** : retorna os ids das redes vinculadas ao IX de id igual a *ix_id*;

(3) **/api/netname/[net_id]** : retorna o nome da rede com id igual a *net_id*;

(4) **/api/ixname/[ix_id]** [*endpoint* "extra"] : ele retorna o nome do IX de id igual a *ix_id*.

Todos os *endpoints* realizam as consultas nos dicionários *netjson*, *ixjson* e *netixlanjson*, estabelecidos na inicialização do programa servidor.

O módulo cliente conecta-se ao servidor HTTP implementado pelo trabalho e realiza as requisições definidas pelos *endpoints*. O cliente recebe como parâmetros o servidor e a porta em que o serviço HTTP opera, além de um terceiro valor, que pode ser 0 ou 1, para selecionar o tipo de análise que será feita. Para realizar uma requisição, foi desenvolvido o método *conexao (requisicao)*, que conecta-se ao servidor HTTP e realiza a requisição enviada pela variável requisição.

A análise 0 é selecionada com o terceiro parâmetro do programa sendo 0, e chama o método *firstanalise*, que imprime os dados de cada rede existente (id e nome) e a quantidade de IXs aos quais ela está vinculada. Para isto, o método faz a requisição **/api/ixids** e, para cada id de IX encontrado, busca todos os ids de redes vinculadas, com a requisição **/api/ixnets/[ix_id]**. Quando uma rede é encontrada nestas requisições **/api/ixnets/[ix_id]**, ela é armazenada num vetor, e caso ela já exista, o contador de IXs desta rede é incrementado. Por fim é consultado o nome da rede com a requisição **/api/netname/[net_id]**.

A análise 1 é selecionada com o terceiro parâmetro do programa sendo 1, e chama o método *secondanalise*, que imprime os dados de cada IX existente (id e nome) e a quantidade de redes vinculadas ao mesmo. Esta é uma análise mais simples que a primeira, porém não era possível adquirir o nome do IX apenas com os três *endpoints* propostos inicialmente, e por isto foi criado o *endpoint* **/api/ixname/[ix_id]**. O primeiro passo é consultar todos os ids de IXs existentes pela requisição **/api/ixids**, e depois consultar o nome de cada id com a requisição **/api/ixname/[ix_id]** e consultar as redes vinculadas ao mesmo com a requisição **/api/ixnets/[ix_id]** e contá-las. Para cada requisição realizada, se a mesma tratar-se de um array, é necessário retirar as duplicatas, e isto foi feito com a função *sorted(set(var))*. Retirar duplicatas foi necessário tanto para a primeira análise quanto para a segunda.

2.2 Instruções de compilação e execução

Para facilitar a compilação e execução dos módulos, foram desenvolvidos três scripts *Shell*, com todos os comandos necessários no ambiente Ubuntu Linux (e outras distribuições GNU/Linux). À seguir serão explicados os passos para compilação e execução com e sem script.

2.2.1 Instruções com script

Primeiramente serão citados os código dentro dos scripts:

```
# compile.sh
#!/bin/bash
cd code;
pyinstaller servidor.py;
pyinstaller cliente.py;
rm -rf build;
rm -rf __pycache__;
rm cliente.spec;
rm servidor.spec;
cd ..;
chmod 777 servidor.sh;
chmod 777 cliente.sh;

# servidor.sh
#!/bin/bash
if [ "root" != `whoami` ]
    then echo "Necessario sudo para executar este script!"
    exit
fi
if [ -z $4 ]; then
    echo "Sao necessarios 4 parametros!"
    echo "Formato: porta netfile ixfile netixlanfile"
    exit
else
    sudo ./code/dist/servidor/servidor $1 $2 $3 $4;
fi

# cliente.sh
#!/bin/bash
if [ -z $2 ]; then
    echo "Sao necessarios 2 parametros!"
    echo "Formato: servidor:porta opcao"
    exit
else
    ./code/dist/cliente/cliente $1 $2;
fi
```

E à seguir temos os passos para execução do sistema:

(a) Compilação: ./compile.sh

(b) Execução do servidor:

- Necessita estar como usuário sudo;
- Primeiro plano: ./servidor.sh porta netfile ixfile netixlanfile
- Segundo plano: nohup ./servidor.sh porta netfile ixfile netixlanfile > servidor.error 2> servidor.log &

(c) Execução do cliente: ./cliente.sh servidor:porta opcao

(d) Terminar a execução do servidor:

- Primeiro plano: ctrl + c
- Segundo plano: fg 1 + ctrl + c

2.2.2 Instruções sem script

À seguir temos os passos para execução do sistema sem script:

(a) Compilação:

- Comando inicial: cd code
- Comando para o servidor: pyinstaller servidor.py
- Comando para o cliente: pyinstaller cliente.py
- Comando inicial: cd ..

(b) Execução do servidor:

- Necessita estar como usuário sudo;
- Primeiro plano: ./code/dist/servidor/servidor porta netfile ixfile netixlanfile
- Segundo plano: nohup ./code/dist/servidor/servidor porta netfile ixfile netixlanfile > servidor.error 2> servidor.log &

(c) Execução do cliente: ./code/dist/cliente/cliente servidor:porta opcao

(d) Terminar a execução do servidor:

- Primeiro plano: ctrl + c
- Segundo plano: fg 1 + ctrl + c

3 Análise Experimental

3.1 Funcionamento

O programa funcionou como esperado, retornando os resultados corretamente. Os resultados mostraram-se coerentes com os exemplos documentados no enunciado do trabalho prático. À seguir uma imagem de trechos retornados pelo programa:

127.0.0.1:555 0

2	Akamai Technologies	153
3	DALnet IRC Network	17
4	Limelight Networks Global	80
5	Swisscom	30
7	RCN	6
8	Verizon Managed Router Service	9
13	XO Communications	8
14	GTT Communications (AS3257)	6
16	XS4ALL Internet B.V.	3
17	BBC	6
19	Webpass	5
20	20C	2
21	Renesys	7
22	Rackspace	11
23	KT Corporation (Korea Telecom)	6
24	DSLExtreme	1
25	MCI - MAE.net	1
26	Stealth Communications	3
27	Yahoo!	88
29	Bouygues Telecom ISP	7
30	Cambridge Bandwidth Consortium	1
31	Telecom Italia Sparkle	8
33	Cologix	7
35	Internap	18
37	Mnet srl	1
72	Pacnet	22
75	Level 3 AS 3549	12
76	TELUS Communications	3
77	Semaphore Corporation	1
86	GTT Communications (AS4589)	15
88	BT Ireland	2

127.0.0.1:555 1

1	Equinix Ashburn	110
2	Equinix Chicago	82
3	Equinix Dallas	50
4	Equinix Los Angeles	41
5	Equinix San Jose	63
7	Equinix Palo Alto	69
9	Equinix Atlanta	9
10	Equinix Vienna (VA)	2
11	Equinix Seattle	13
12	Equinix New York	50
13	SIX Seattle	77
14	NYIIX	66
15	MAE East	0
16	MAE West	1
17	Equinix Miami (formerly NOTA)	57
18	LINX LON1	210
20	MAE Central	0
21	IX Australia WA	16
22	Digital Realty Atlanta	36
23	LAIIX	22
24	TorIX	44
26	AMS-IX	243
29	Equinix Zurich	28
30	JPIX TOKYO	38
31	DE-CIX Frankfurt	224
32	SFIX	1
33	CIXP	19
34	SFINX	29
35	MIX-IT	40
40	PARIX	7
41	NYCX	3

Figura 2 - Trechos de retornos do sistema

3.2 CDF da distribuição da quantidade de IXP por rede

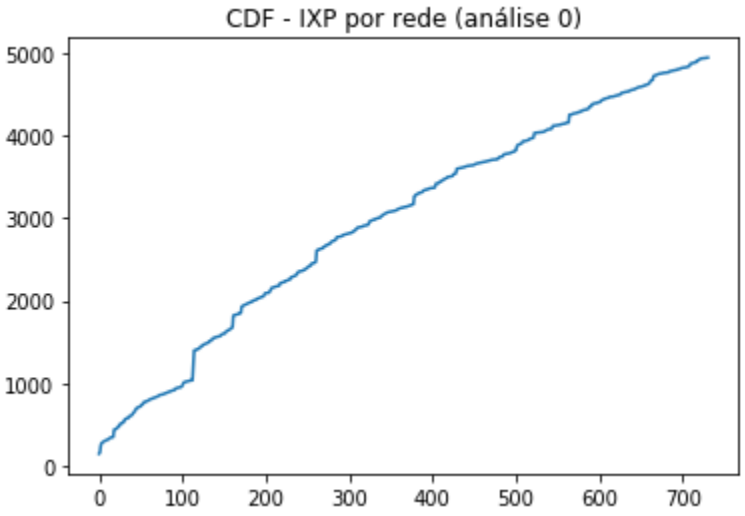

Figura 3 - Gráfico CDF da análise 0

3.3 CDF da distribuição da quantidade de redes por IXP

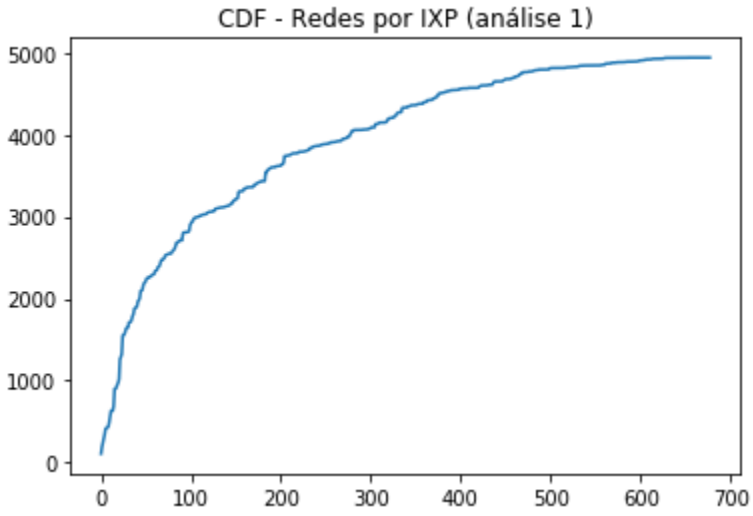

Figura 4 - Gráfico CDF da análise 1

3.4 Discussão dos resultados

Com os gráficos exibidos anteriormente é possível ver que os IXs estão distribuídos razoavelmente proporcionalmente entre as redes relatas, porém este fato não ocorre quanto à distribuição das redes entre os IXs. Os primeiros IXs possuem muito mais redes que os últimos, e quando observa-se a saída do programa, é possível ver que muitos IXs não são associados à nenhuma rede, principalmente entre os últimos.

Este resultado faz sentido, já que os IXs são os *Internet Exchange Points*, ou **Pontos de Troca de Tráfego**, que representam a interconexão entre os provedores de internet e as redes de fornecimento de conteúdo. A distribuição de redes entre os provedores realmente é desigual, alguns provedores possuem muito mais redes do que outros, principalmente na América do Norte. Já uma rede pode utilizar ou não serviços de mais que um provedor, e escolhê-los de forma mais distribuída. Os gráficos possuem coerência.

3.5 Conclusão

Este foi um trabalho muito interessante, porém que com mais tempo poderia ser ainda melhor feito, e assim trazer resultados ainda mais interessantes.

Durante o trabalho foi decidido lidar apenas com a biblioteca HTTP do próprio *Python3*, ao invés de utilizar o Flask. Esta escolha deve-se a um conhecimento prévio da biblioteca, além de ser melhor documentada.

É muito útil aprender como construir um servidor HTTP, em geral os desenvolvedores web não se atentam sobre o quando perto o protocolo HTTP está deles, e o quando é simples manipulá-lo. Programas como *Apache* e *Wamp* auxiliam em serviços como o que o

servidor que foi desenvolvido faz, porém aprender o **como** é feito é muito importante, até para melhoria dos códigos já existentes.

Referências

[1] **GitHub do trabalho**: github.com/PaulaViriato/HTTP_Server_and_Client

[2] **HTTP servers**: docs.python.org/3/library/http.server.html

[3] **HTTP at Python3.7**: github.com/python/cpython/tree/3.7/Lib/http