PeeringDB REST API

Luísa Ribeiro Bezerra - Matrícula: 2015086441

1. INTRODUÇÃO

Uma chamada remota de procedimento (RPC, acrônimo de *Remote Procedure Call*) é uma tecnologia de comunicação entre processos que permite a um programa de computador chamar um procedimento em outro espaço de endereçamento (geralmente em outro computador, conectado por uma rede). O programador não se preocupa com detalhes de implementação dessa interação remota: do ponto de vista do código, a chamada se assemelha a chamadas de procedimentos locais. O RPC é uma tecnologia popular para a implementação do modelo cliente-servidor de computação distribuída. Uma chamada de procedimento remoto é iniciada pelo cliente enviando uma mensagem para um servidor remoto para executar um procedimento específico. Uma resposta é retornada ao cliente. Uma diferença importante entre chamadas de procedimento remotas e chamadas de procedimento locais é que, no primeiro caso, a chamada pode falhar por problemas da rede. Nesse caso, não há nem mesmo garantia de que o procedimento foi invocado. [1]

O objetivo principal deste trabalho é desenvolver um par *cliente-servidor* utilizando chamadas de procedimentos remotos (RPC) em REST. O trabalho se baseou em um sistema de consulta para o *PeeringDB*, um banco de dados de interconexões (peerings) entre redes membro de IXPs (Internet Exchange Points). O sistema possuirá um (1) programa cliente, que faz requisições HTTP para um servidor através de uma API REST, e (2) um programa servidor, que atende as requisições de clientes. Ao final, temos um JSON com dados requisitados e tratados, seguindo de duas análises predeterminadas, discutidas a seguir.

2. INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO DO PROGRAMA

O programa foi feito para executar em **Python 2.7**. Para obter tal versão estável, é recomendado executar a seguinte linha de comando no terminal:

```
sudo apt-get install python2.7
```

2.1. SERVIDOR

Ao iniciar, o servidor lê três arquivos que representam os dados do PeeringDB: *net.json* (contendo informações sobre redes), *ix.json* (contendo informações sobre IXPs) e *netixlan.json* (contendo associações entre redes, IXPs e LANs), fornecidos pelo professor para este Trabalho Prático. Para sua execução, o servidor deve ser executado com a seguinte linha de comando no terminal:

```
python server.py port Netfile Ixfile Netixlanfile
```

Onde [port] é o porto no qual o servidor receberá mensagens, [Netfile] é o caminho do arquivo de redes, [Ixfile] é o caminho do arquivo de IXPs e [Netixlanfile] é o caminho do arquivo de associações. O servidor só termina com comando explícito do usuário (CTRL+C) - o cliente deve terminar imediatamente após gerar a saída da opção de análise.

2.2. CLIENTE

O cliente deve ser executado com a seguinte linha de comando:

```
python client.py IP:port Opt
```

Onde [IP] é o IP do servidor, [port] é o porto que o servidor está utilizando e [Opt] especifica o número da análise: "0" se for a análise de IXPs por rede ou "1" se for a análise redes por IXP.

Outro ponto importante é de que o servidor e o cliente podem se comunicar através do IP 127.0.0.1 se o programa estiverem executando na mesma máquina, mas pode também ser o endereço de uma outra máquina acessível pela rede

3. DESENVOLVIMENTO DO PROBLEMA

Para o desenvolvimento, tivemos muito trabalho com a implementação de tal trabalho. Para tal, utilizamos uma biblioteca permitida - **Flask** [2], porém apenas para o servidor. Por partes, analisaremos o desenvolvimento de cada uma delas - Cliente e Servidor -, para um melhor entendimento de suas lógicas e desenvolvimentos:

3.1. SERVIDOR

A princípio, o servidor recebe como parâmetros o [port], que é o porto no qual o servidor receberá mensagens, o [Netfile], é o caminho do arquivo de redes, o [Ixfile], que é o caminho do arquivo de IXPs, e o [Netixlanfile], que é o caminho do arquivo de associações - como descrito no item 2.1 desta documentação.

O Flask utiliza decorators para demarcar as funções que tratarão os requests, de acordo com a URL e o Método HTTP utilizado. Por exemplo, no código-fonte temos:

```
@app.route('/api/ix')
```

Isto indica que função “decorada”, *ixis()*, atenderá a requests para a raiz do servidor. O Flask também tem um servidor embutido, e o comando do código-fonte abaixo o inicia, tendo como parâmetros o host e o porto (retirado de argumento dado para a execução do programa):

```
app.run(host='0.0.0.0', port=porto)
```

Na parte do servidor, foi-nos requeridos a criação de 03 endpoints - métodos GET descritos abaixo:

1. **`/api/ix`: todos os IXPs**

   @app.route('/api/ix')

   Este endpoint lê do arquivo *ixfile* os IXPs e retorna um JSON, com um campo "data", o qual contém uma lista dos objetos IXPs;
2. **`/api/ixnets/{ix_id}`: identificadores das redes de um IXP**

   @app.route('/api/ixnets/<ix_id>')

   Este endpoint lê do arquivo *netixlanfile* e retorna um JSON, com um campo "data", o qual contém uma lista dos identificadores das redes do IXP identificado por *'ix_id';*
3. **`/api/netname/{net_id}`: nome de uma rede**

   @app.route('/api/netname/<net_id>')
4. Este endpoint lê do arquivo *netfile* e retorna um JSON, com um campo "data", o qual contém o nome da rede identificada por cada *'net_id'.*

3.2. CLIENTE

O cliente se inicia recebendo os parâmetros [IP], que é o IP do servidor, [port], que é o porto que o servidor está utilizando, e [Opt], que especifica o número da análise: "0" se for a análise de IXPs por rede ou "1" se for a análise redes por IXP.

Para maior facilidade, são definidos dois métodos auxiliares - *recebendo_fragmentos(SOCK)* e *get_dados(servico, HOST, PORT).* O primeiro método, no caso, existe pelo fato de que, na conexão TCP, o dado pode chegar fragmentado - logo, faz-se necessário a existência deste método para acoplar as partes que vierem fragmentadas (por isso a existência do recv, que recebe as mensagens através do socket, que lê os bytes recebidos, retornando-os em uma string, até o limite de *buffer* definido por *buffsize*). O segundo método, no caso, faz conexão TCP, chama o método HTTP GET para determinado endpoint passado como parâmetro, utiliza o primeiro método para recolher os fragmentos da chamada HTTP GET designada, e retorna um pacote JSON, contendo os dados requeridos - ao fim, fecha a conexão TCP e encerra corretamente sua execução.

O cliente também é responsável por realizar duas análises sobre os dados do PeeringDB, que estão descritas no próximo item.

4. ANÁLISES

4.1. ANÁLISE 0 - IXPs por redes

Para a Análise 0, o cliente deve produzir na saída padrão um TSV UTF-8 (tabelas com células separadas por tabulações '\t') com uma rede por linha (em qualquer ordem), contendo as seguintes colunas (nessa ordem):

1. Identificador da rede: esse dado pode ser encontrado no campo id do arquivo net.json; ou no campo net_id do arquivo netixlan.json
2. Nome da rede: esse dado pode ser encontrado no campo name do arquivo net.json

3. Número de IXPs associados à rede: deve ser gerado pelo seu cliente

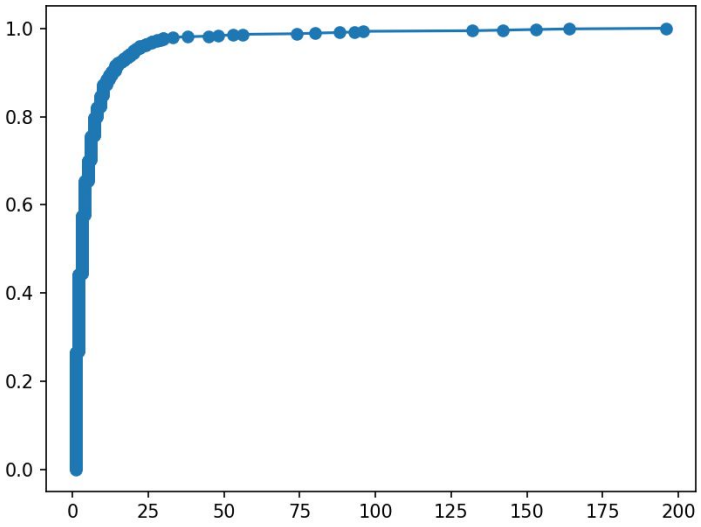
Para a implementação, é executado um dos métodos auxiliares - o *get_dados*, com o endpoint de /api/ix, repassando também o endereço de HOST e PORT, identificados ao inicializar o programa por meio dos argumentos de entrada. Após isso, itera-se em todos ix's para realização da contagem de IXPs para cada rede, *ix_net*, que é obtida por meio também do método *get_dados* para o endpoint /api/ixnets/+ id daquele ix. Por fim, obtém-se os nomes das redes por meio de um *get_dados* para o endpoint /api/netname/+ net, o qual retorna, para cada rede, seu nome, e, por fim, é printado o identificador da rede, o nome da rede e o número IXPs associado à ela para cada uma das redes. Por fim, temos a seguinte visualização:

```
luisa@luisa:~/Downloads/tp3$ python2 client.py 127.0.0.1:5000 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
```

Um IXP (*Internet Exchange Point*) é um ponto de intercâmbio que permite a interligação de redes autônomas bem como a troca de tráfego de dados e conteúdos entre elas. A existência de um IXP em determinado país ou região torna possível que a comunicação entre as redes (não só de provedores) se faça por ligações locais únicas de *latência muito menor*. As ligações locais, além de mais "rápidas" são também mais baratas permitindo redução dos custos operacionais dos provedores, empresas, instituições que se ligarem aos IXP.

O gráfico abaixo plotado exemplifica uma **CDF(Cumulative distribution function)** da distribuição da quantidade de IXP por rede. Por meio do gráfico, é nítido que as redes são concentradas em IXPs - ou seja, a enorme maioria das redes participa de poucos IXPs. Por volta de 90% das redes, por exemplo, participam de 15 IXPs ou menos, apenas um pouco acima de 10% das redes participam de mais de 15 IXPs. Uma boa teoria para isso seria a preferência de algumas redes por alguns IXPs, que possivelmente apresentam uma boa interconexão, por exemplo.

Fração acumulada de redes (Y) por IXPs (X)



4.2. ANÁLISE 1

Para a Análise 1, o cliente deve produzir na saída padrão um TSV UTF-8 (tabelas com células separadas por tabulações '\t') com uma rede por linha (em qualquer ordem), contendo as seguintes colunas (nessa ordem):

1. Identificador do IXP: esse dado pode ser encontrado no campo id do arquivo ix.json; ou no campo ix_id do arquivo netixlan.json
2. Nome do IXP: esse dado pode ser encontrado no campo name do arquivo ix.json
3. Número de redes associadas ao IXP: deve ser gerado pelo seu cliente

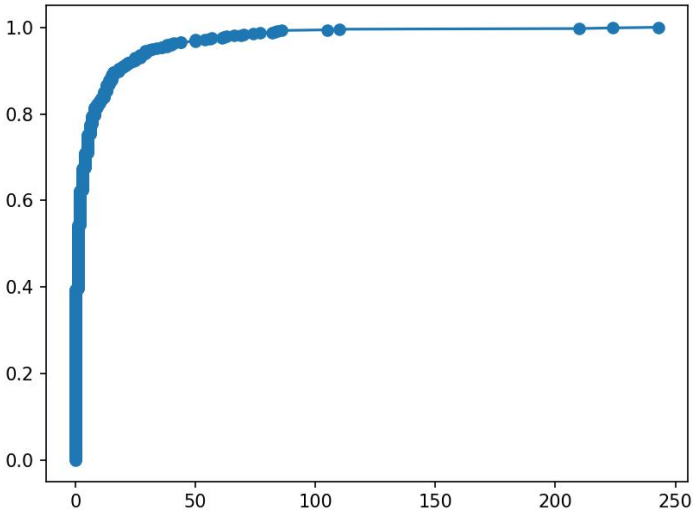
Em seu desenvolvimento, é executado um dos métodos auxiliares - o *get_dados*, com o endpoint de /api/ix, repassando também o endereço de HOST e PORT, identificados ao inicializar o programa por meio dos argumentos de entrada. Para cada IX da resposta, temos novamente a requisição de outro serviço por meio do *get_dados* para o endpoint /api/ixnets/ + id daquele ix, e é feita a contagem do número de redes por IX a eles associados. por fim, printa-se o id do IXP, nome do IXP e número de redes associadas a ele. Exemplo de saída:

```
luisa@luisa:~/Downloads/tp3$ python2 client.py 127.0.0.1:5000 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
```

O gráfico abaixo plotado exemplifica uma **CDF(Cumulative distribution function)** da distribuição da quantidade de redes por IXP. Por meio do gráfico, é nítido que as redes são

concentradas em IXPs. É visível que os IXPs são concentrados em redes, ou seja, a enorme maioria dos IXPs interconecta poucas redes, e também é nítido a existência de alguns pontos do outro lado do gráfico, que são as exceções disso.

Fração acumulada de IXPs (Y) por redes (X)



5. CONCLUSÕES FINAIS

Todos os conceitos englobados neste trabalho foram, por si só, um desafio. Ao todo, tivemos que lidar com também novos frameworks, que possibilitaram uma facilidade na execução do trabalho, bem como a aplicação de melhor qualidade dos mesmos. Analisar os dados segundo a proposta dada no Trabalho requer muito estudo, principalmente de conceitos novos para aplicarmos no nosso projeto final.

É imprescindível mencionar também que o estudo dos IXPs nos deu uma visão de como realmente a troca de tráfego (peering) realizada por IXPs, que oferecem conexão direta, permitem a simplificação do trânsito da Internet, diminuindo o número de redes até um certo destino - o que melhora a qualidade, reduz custos e aumenta a resiliência da rede.

6. REFERÊNCIAS

[1] https://pt.wikipedia.org/wiki/Chamada_de_procedimento_remoto

[2] https://pt.wikipedia.org/wiki/Flask_(framework_web)

[3] http://www.inf.ufg.br/~ricardo/python/programacao.sockets.html

[4] https://en.wikipedia.org/wiki/Cumulative_distribution_function

[5] https://www.menosfios.com/sabe-um-ixp/