UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
AVALIAÇÃO E DESEMPENHO (2018-1)
VITOR MARQUES DE MIRANDA

Simulador de Roteador

1. INTRODUÇÃO

1.1. Programação

O simulador foi feito baseando-se no conceito de que: Python seria uma boa opção para a programação por ser uma linguagem mais simples, deixando assim o foco na logística da aplicação; e que a melhor forma de se fazer uma interface gráfica, como adotado hoje por uma grande quantidade de aplicações, seria com HTML5, CSS e Javascript.

Considerando esses dois fundamentos, as escolhas de desenvolvimento foram sendo tomadas. ***Python*** foi pego como a linguagem, e ***pip*** foi escolhido como o gerenciador de pacotes por haver experiência prévia com a ferramenta. O pacote ***SciPy***, o qual vem acompanhado do ***NumPy***, foi selecionado para lidar com questões estatísticas pela mesma razão que Python foi escolhido como linguagem: mais foco no logística e menos em como programa-la.

Isso essencialmente definiria o backend do simulador, no entanto seria necessário uma porta de comunicação entre o backend e o frontend, que seria essencialmente uma página rodando em um browser. Para faze-lo, a tarefa foi deixada nas mãos do ***Flask***, um framework para Python capaz de levantar um serviço.

O desenvolvimento começou em uma máquina virtual com sistema operacional Linux Mint, no entanto as tarefas se demonstraram muito demandadas para a VM, exigindo que tudo fosse testado em uma máquina mais potente. Por falta de opções, um computador da Apple com o sistema operacional macOS Mojave (10.14) foi a escolha. No entanto, o sistema operacional Mojave encontrasse em uma fase de incompatibilidade com as VM da aplicação VirtualBox, que era a usada no outro computador, inferior, que executava o sistema macOS High Sierra (10.13).

Para contornar essa dificuldade, um ***Docker*** foi usado. O Docker em questão era baseado no sistema operacional ***Ubuntu***, e fica responsável não apenas por hospedar o serviço Python do backend, mas também por tornar o frontend acessível de forma mais agil, usufruindo novamente do Flask.

O frontend, por fim, foi feito utilizando Javascript, HTML e CSS bem básicos, em conjunto a um framework chamado ***ChartJS***, para permitir que os gráficos sejam configurados e plotados de uma maneira agradável e eficiente. Uma vez que o Docker é levantado (ou que o Flask seja executado, caso o Docker não seja utilizado), a aplicação pode ser acessada por um navegador convencional usando a URL ***localhost:5000/***.

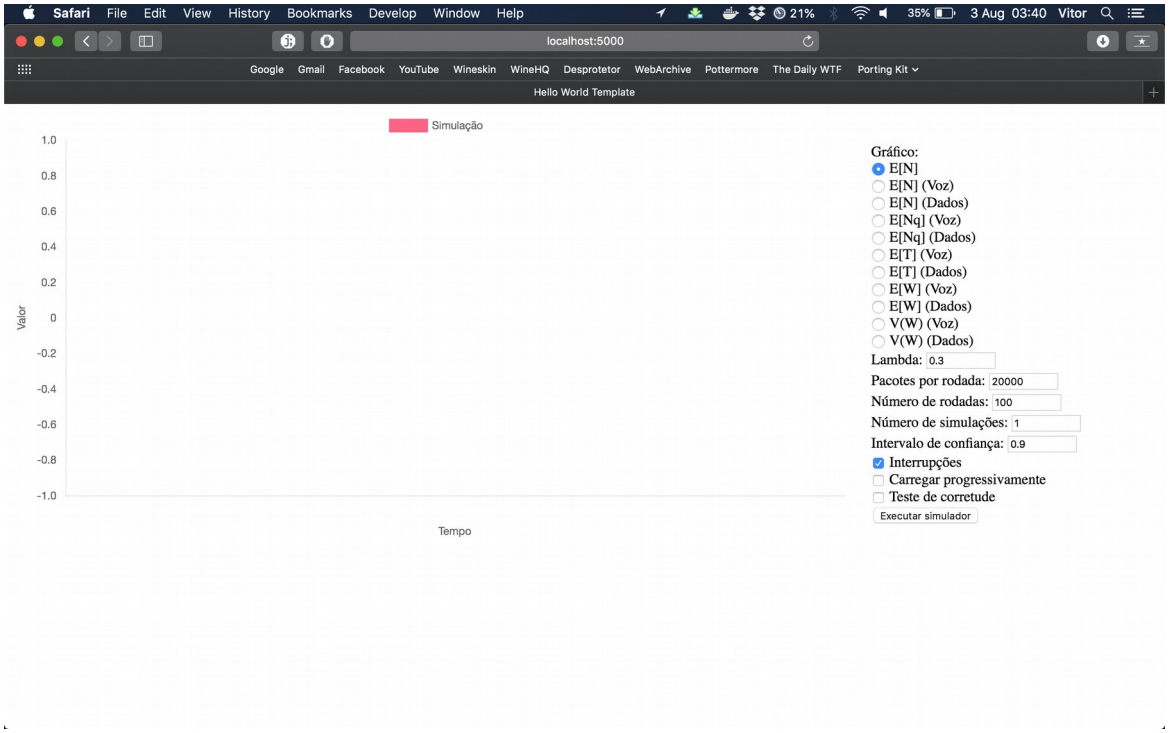
Devido à natureza de poder funcionar através de um Docker, a aplicação pode ser executada virtualmente falando a partir de qualquer sistema operacional direcionado a computadores, como macOS, Windows, e múltiplas distribuições Linux. Ao ser executado

diretamente, por outro lado, só pode ser executado em sistemas operacionais Linux com ***apt-get*** disponível, como Ubuntu e Mint.

Para executar a instalação no Linux, é necessário apenas executar o arquivo ***build-docker.sh***, caso *Docker* esteja disponível no sistema. Caso deseje-se executar sem o Docker, é necessário executar ***build-nodocker.sh*** em vez disso. Lembrando que, caso seja instalado sem Docker, é necessário que pasta ***simulador*** esteja no diretório raiz do sistema, para evitar incompatibilidades com o modo de ***Carregar progressivamente***, que será explicado mais adiante.

1.2. Funcionamento (Frontend)

Ao entrar na URL mencionada no subcapítulo anterior, o usuário irá se deparar com a seguinte tela:



Nessa tela, é possível configurar qual simulação será executada pelo usuário, e com quais parâmetros. Esses parâmetros são convertidos em uma request GET, que chama, via Flask, o simulador feito em Python.

Se a opção ***Carregar progressivamente*** estiver *desmarcada*, como na imagem acima, o gráfico será plotado aos poucos, sem que haja a necessidade de aguardar até o fim da execução da aplicação do backend.

Para fazer isso, o simulador irá criar um diretório ***plot***, dentro do diretório *templates* da pasta *simulador*, onde ele armazenará a saída do simulador em diferentes arquivos, cada um com 100 linhas . Isso ocorre pois, por comportamento do Flask, arquivos inseridos na pasta templates são carregados de maneira dinâmica, o que permite que os arquivos sejam adicionados em tempo de execução, permitindo à aplicação em frontend carrega-los em pedaços; algo que não poderia fazer durante uma única request.

A razão para se utilizar os blocos de 100 linhas é mais simples do que aparenta: para evitar que haja retrabalho por parte do frontend, que tende a gastar cada vez mais

memória conforme a simulação avança, o Javascript procura por novos arquivos continuamente (o que é bem simples, dado que os arquivos seguem o padrão de nome 0.csv, 1.csv, 2.csv, etc). Cada vez que ele lê um desses arquivos, ele combina aos que já carregou previamente, eliminando a necessidade de reler dados que ele já possui, utilizando o mínimo de recursos para isso.

Se, por outro lado, a opção ***Carregar progressivamente*** estiver *marcada*, o frontend irá aguardar até que o backend tenha a resposta completa para a sua requisição, e então retornará tudo de uma vez na própria request GET. Testes demonstram que o retorno é mais rápido neste modo, o que não surpreende, já que ele dispensa o processo de leitura e escrita de arquivos, além de economizar processamento, já que o frontend não é desenhado até que todos os dados estejam em mãos. Apesar disso, em caso de grandes simulações, o carregamento progressivo é recomendado para acompanhar os resultados mais rapidamente.

1.3. Funcionamento (Backend)

1.3.1. Filas e eventos

Baseando-se na estrutura do sistema de duas filas utilizado no período anterior, o backend foi feito pensando-se na ocorrência de 4 diferentes eventos: a chegada de pacotes de dados, o fim do serviço de pacotes de dados, a chegada de pacotes de voz e o fim do serviço de pacotes de voz.

Cada evento possui um timer, que indica quanto tempo falta para que ele aconteça da próxima vez, permitindo assim averiguar e calcular a sequência dos eventos. No caso da chegada de pacotes de voz, temos um timer para cada canal, ou seja, 30. Isso totaliza 33 timers no sistema.

Pacotes de voz tem prioridade, portanto o evento da chegada de um deles verifica se há alguém mais na fila de pacotes de voz além dele. Se não houver mais ninguém nesta fila, e não houver outro pacote de voz sendo atendido, a ação a seguir dependerá se interrupção estiver ativada ou não. Se sim, ele irá direto para o serviço, interrompedo um pacote de dados em atendimento se houver. Se não, ele só irá para os serviço se não houver ninguém sendo atendido; e caso contrário ficará na fila.

A chegada de pacotes de dados, por outro lado, é mais sútil, Eles só são imediatamente atendidos se não houver mais ninguém no sistema, e caso haja eles vão para o fim da fila de pacotes de dados.

O atendimento de pacotes de voz é tão simples quanto. Se houver um pacote de voz na fila, ele será o próxima a ser atendido. Se não houver, o primeiro da fila de dados será atendido (ou voltará a ser atendido, se foi interrompido); a menos que não haja nenhum. Neste caso, haverá um intervalo onde não haverá ninguém no sistema.

Já o atendimento de pacotes de dados varia se o sistema de interrupção estiver ativado ou não. Havendo interrupções, nunca haverá um pacote de voz aguardando, pois este teria interrompido o pacote de dados; o que significa que o próximo pacote a ser atendido, se houver, será um pacote de dados. No entanto, se não houver interrupções, pode houver um pacote de voz na fila, e se houver, ele deverá ter prioridade em relação a pacotes de dados.

1.3.2. Números aleatórios

Apesar de haver uma geração de números aleatórios disponível em Python, alguns completementos foram adicionados para garantir “variedade” nos valores gerados. As sementes usadas para gerar os valores via `expovariate`, o qual gera os valores aleatórios baseados em lamba, apenas são aceitam se elas possuem uma determinada distância das sementes geradas anteriormente.

1.3.3. Método de execução

O método adotado é o Batch, para comparar sistemas já estabilizados, que pareceu ser a proposta mais interessante, além de já ser a linha adotada na estrutura base mencionada no início do subcapítulo 1.3.1.

Os valores passados de número de rodadas e de pacotes por rodada definem quando uma rodada termina, e por quantas rodadas a execução deve passar.

2. USOS

2.1. Utilização de Dados

Diferentes valores de lambda foram usados tentando analizar qual traria as utilizações desejadas para os testes do trabalho.

Houve também a tentativa de averiguar o lambda baseado no valor esperado de E[X]. Baseando-se no enunciado, temos que p = ƛ E[X]. Com isso, ƛ = p / E[X] . Com 754.8 bytes de tamanho médio para pacotes de dados, e com 2MBps de velocidade, o tempo esperado de execução sem considerar interrupções é de 754.8 / 2*1024*1024 segundos, ou 0.359916687012 milisegundos . Considerando isso, é difícil de calcular diretamente esse valor, tornando as tentativas a melhor opção.

Seguem abaixo os valores descobertos por tentativa e erro:

- Lambda = 0.45210 => p = 0.1
- Lambda = 0.91098 => p = 0.2
- Lambda = 1.3499 => p = 0.3
- Lambda = 1.83506 => p = 0.4

Os valores para p igual ou maior que 0.5 não foram calculados ou encontrados.

p_1	E[T1]			E[W1]		
.1	0,50	0,49	0,48	0,27	0,27	0,26
.2	0,63	0,61	0,59	0,40	0,39	0,38
.3	0,82	0,83	0,81	0,61	0,60	0,59
.4	1,56	1,55	1,53	1,46	1,45	1,43
.5						
.6						
.7						

p_1	E[X1]			E[Nq1]		
.1		0,2211		0,0081	0,059	0,0036
.2		0,2195		0.001750	0.000793	0
.3		0,2222		0.0022	0.00138	0.000492
.4		0,2179		0.00114	0.00031	0
.5						
.6						
.7						

p_1	E[T2]			E[W2]			E[Nq2]		
.1	0,0337	0,0336	0,0335	0,0032	0,0031	0,0030	0,053	0,050	0,048
.2	0,03095	0,03094	0,03092	0.00044	0.000423	0.00040	0,23	0,22	0,21
.3	0.03129	0.03125	0.03122	0.00077	0.00074	0.00070	0,64	0,63	0,62
.4	0.03069	0.03068	0.03067	0.000174	0.000166	0.000157	2,18	2,20	2,22
.5									
.6									
.7									

p_1	E[Δ]			V(Δ)		
.1						
.2						
.3						
.4						
.5						
.6						
.7						

3. CONCLUSÕES

A escolha de usar o trabalho anterior como uma base para o sistema foi feita para tentar poupar tempo de desenvolvimento. Apesar disso, erros no desenvolvimento do trabalho passado acabaram por afetar o desenvolvimento deste trabalho; se a idéia de usá-lo como uma base foi no fim benéfica ou não, é difícil dizer pelo tempo gasto corrigindo seus problemas.

Por outro lado, o uso de um Docker provou-se muito útil, permitindo que o trabalho fosse transmitido para um computador mais potente. Além disso, o programa poderia ser facilmente implantado em um servidor na nuvem na forma que foi concebido, permitindo o uso de um enorme processamento sem muito esforço.

O ChartJS provou ser uma escolha bem melhor que o matplotlib, permitindo o desenho dos gráficos em tempo real por exemplo. Apesar disso, o desenho de um gráfico provou ser sempre um gargalo na hora de exibi-lo, devido ao processamento necessário para desenhar tudo.

O uso de Python novamente, no entanto, reafirmou o problema de ser um linguagem lenta, causando lentidão ao realizar os cálculos. Apesar disso, uma máquina mais potente conseguiu resolver esse problema no caso dos cálculos estatísticos (não sendo suficiente, porém, para que os cálculos necessários para gerar os gráficos ocorressem em um curto espaço de tempo).

O Flask também provou-se muito útil, especialmente combinado a um Docker, criando essencialmente um mini-servidor móvel, o que permitiu o uso do ChartJS sem muitas compilicações (apesar da falta de documentação para o mesmo ter ocasionado um pouco de lentidão no processo).