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/***.

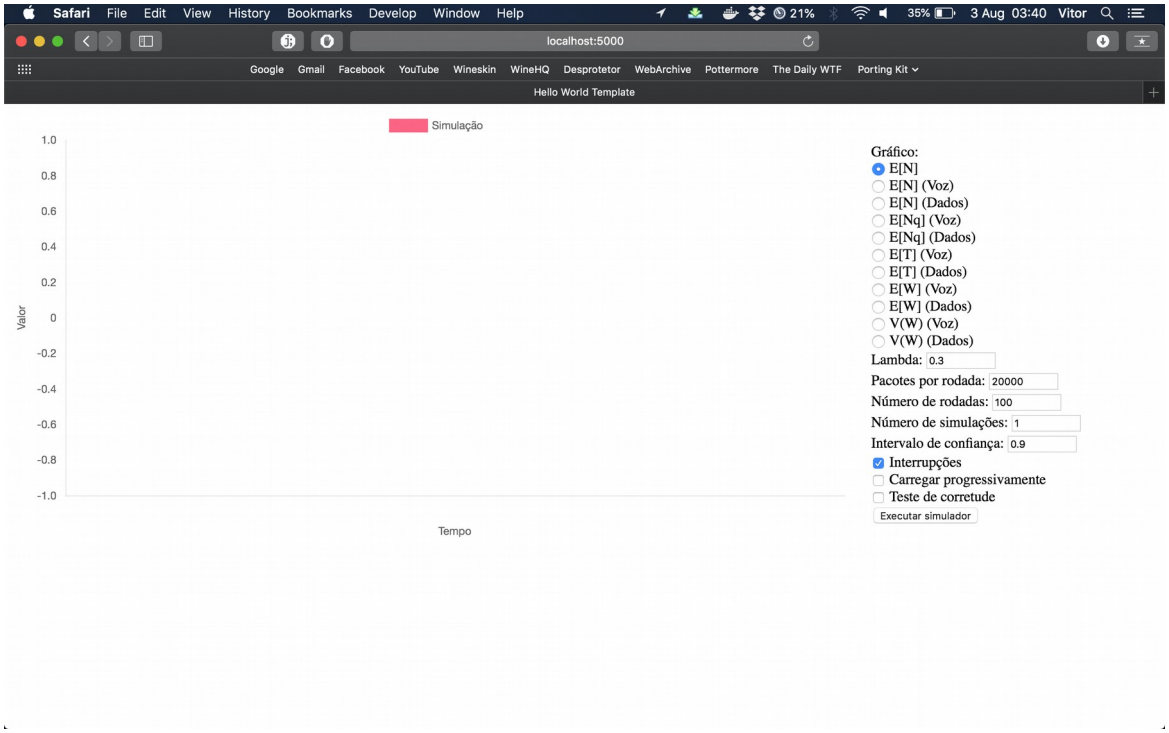
Devido à natureza de poder funcionar através de um Docker, a aplicação pode ser executada virtualmente falando a partir de qualquer sistema operacional direcionado a computadores, como macOS, Windows, e múltiplas distribuições Linux. Ao ser executado

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 valor atribuído, o qual é o tempo em milisegundos até o evento ocorrer (chegar na fila ou finalizar atendimento). No caso da chegada de pacotes de voz, esse valor é armazenado para cada canal, ou seja, 30. Isso totaliza 33 diferentes tipos de eventos que podem ocorrer. O evento que possuir o menor valor é o primeiro na ordem cronológica, e por isso ocorre primeiro. Após sua execução, seu tempo então é deduzido dos tempos dos demais eventos, e então o processo se repete, até que não hajam mais eventos ou que a execução seja interrompida. Lembrando que mais eventos surgem conforme a aplicação avança.

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.

No caso da escolha do tamanho do pacote de dados, a medida tomada foi outra. Para cada possível valor, a probabilidade foi calculada, permitindo que, ao gerar um número entre 0.0 e 1.0, o mesmo era comparada com a primeira probabilidade com os diferentes tamanhos em ordem numérica. Se o número gerado for maior, ele é subtraído pela probabilidade e então comparado com o seguinte, repetindo o processo. Se o número gerado for menor, o tamanho do pacote de dados referente àquela probabilidade é escolhido.

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. Se, digamos, 20.000 pacotes por rodada estão definidos, então a chegada do 20.001º pacote iniciará a segunda rodada, o 40.001º pacote iniciará a terceira, e assim por diante.

1.3.4. Testes de corretude

Existem duas metodologias diferentes de executar testes de corretude: forçar os valores médios ou forçar a semente. Os dois métodos estão disponíveis pelo frontend para uso.

Forçar os valores médios consiste em retornar valores constantes e absolutos sempre, ou seja, o valor médio esperado da variável. Um exemplo: o tempo de serviço de um pacote de dados em um teste como esse é sempre a média do tamanho dos pacotes de dados (em bits; 6038,4) dividido pela velocidade do meio (2000, ao converter 2 Mbps para bits por microsegundo), o que dá 3,0192. Outro exemplo é o tempo para chegada de um novo pacote de dados, que passa ser sempre 1000 divido por lambda, já que o valor de lambda é em segundos e o simulador trabalha em microsegundos.

Forçar a semente significa permitir que o usuário escolha a semente determinísticamente. Isso lhe permite reproduzir resultados anteriores usando diferentes gráficos, por exemplo, além de provar que as sementes estão realmente sendo respeitadas pelos geradores de números aleatórios.

1.3.5. Fase transiente

Para determinar quando a fase transiente deve terminar, primeiramente foram observadas várias simulações, no período passado. Percebeu-se que atingir um padrão não necessariamente significa manter um valor constante, mas sim atingir um crescimento constante, e que o valor constante ocorre quando esse crescimento constante é zero.

Portanto, para reconhecer esse momento de crescimento constante, em cada evento é calculado o E[N], para que a cada 1000 eventos seja calculada e comparada a variância dos últimos 1000 eventos e dos 1000 eventos anteriores a esses. Se a diferença entre as duas variâncias for menor que 0.002 (ou 2*10^-3), a fase transiente acaba.

Originalmente, a fase transiente terminava se a diferença entre as duas variâncias fosse menor que 0.0000002 (ou 2*10^-7), pois esse era o valor utilizado no trabalho do período passado. Apesar disso, esse valor provou ser completamente inválido para ester trabalhando, fazendo com que a aplicação levasse tanto tempo para atingir o estado que o Flask retornava *killed*, finalizando o processo sem dar uma resposta.

Acreditasse que esse problema pode ser ocasionado por overflow, ou talvez por que a aplicação estivesse consumindo mundo mais memória do que poderia. De uma maneira ou de outra, a mudança desse valor corrigiu o problema. Os valores aqui ditos podem ser customizados no frontend.

2. RESULTADOS

Utilizando os valores fornecidos para lambda, foram simulados os diferentes valores desejados baseando-se na utilização esperada:

- Lambda = 33.1125827815 => p = 0.1
- Lambda = 66.2251655629 => p = 0.2
- Lambda = 99.3377483444 => p = 0.3
- Lambda = 132.4503311258 => p = 0.4
- Lambda = 165.5629139073 => p = 0.5
- Lambda = 198.6754966887 => p = 0.6
- Lambda = 231.7880794702 => p = 0.7

Lembrando que, como os valores de lambda se dão em pacotes/segundo, o backend divide lambda por 1000 para se ter o valor em pacotes/milisegundo.

Os testes abaixo foram feitos usando 100 rodadas, cada uma com 20.000 pacotes. Apesar de com esses valores haverem algumas sinuosidades nos valores, quantidades muito maiores, como 200.000 pacotes e 100 rodadas, causavam o erro *killed* mencionado anteriormente, então os testes foram feitos sob essa medida.

2.1. Sem interrupção

p_1	E[T1]			E[W1]		
.1	6.000744	5.952265	5.903787	2.971171	2.924475	2.877779
.2	7.075676	7.025008	6.974340	4.066733	4.019093	3.971453
.3	8.788036	8.723869	8.659701	5.767917	5.706166	5.644415
.4	11.378513	11.268301	11.158089	8.349243	8.242173	8.135103
.5	16.505960	16.299912	16.093863	13.485013	13.280624	13.076234
.6	31.332433	30.568516	29.804599	28.306471	27.545400	26.784329
.7	651.349513	594.372372	537.395230	648.338167	591.361052	534.383936

p_1	E[X1]			E[Nq1]		
.1	3.041715	3.027791	3.013867	0.143355	0.141461	0.139567
.2	3.016502	3.005915	2.995329	0.359488	0.355508	0.351528
.3	3.026312	3.017702	3.009092	0.708421	0.700883	0.693345
.4	3.033272	3.026128	3.018985	1.285684	1.269701	1.253718
.5	3.025552	3.019288	3.013024	2.468683	2.431395	2.394106
.6	3.029659	3.023116	3.016573	5.930121	5.768802	5.607484
.7	3.016530	3.011320	3.006110	151.551375	138.251703	124.952031

p_1	E[T2]			E[W2]			E[Nq2]		
.1	2.713307	2.698456	2.683605	2.457307	2.442456	2.427605	3.827011	3.802635	3.778259
.2	2.938525	2.923131	2.907738	2.682525	2.667131	2.651738	3.893727	3.869787	3.845847
.3	3.113177	3.097454	3.081731	2.857177	2.841454	2.825731	3.892504	3.868410	3.844317
.4	3.282206	3.267967	3.253729	3.026206	3.011967	2.997729	3.873690	3.851750	3.829809
.5	3.392019	3.378433	3.364847	3.136019	3.122433	3.108847	3.795824	3.771910	3.747997
.6	3.537479	3.525496	3.513514	3.281479	3.269496	3.257514	3.751828	3.732442	3.713056
.7	3.632386	3.619803	3.607220	3.376386	3.363803	3.351220	3.631957	3.611816	3.591675

p_1	E[Δ]			V(Δ)		
.1	2.666281	2.546404	2.426528	1258.778072	1052.634887	846.491703
.2	2.884816	2.759523	2.634230	1233.537746	1095.084172	956.630598
.3	2.981673	2.826874	2.672075	2003.183615	1580.722344	1158.261073
.4	3.059271	2.917473	2.775675	2209.912859	1805.331687	1400.750514
.5	2.770807	2.619905	2.469004	1818.598917	1475.365210	1132.131502
.6	2.968821	2.802883	2.636945	2590.003464	2154.657279	1719.311095
.7	2.449250	2.289488	2.129726	1755.729022	1442.532338	1129.335653

2.2. Com interrupção

p_1	p_1*	E[T1]			E[W1]		
.1	0.184795	9.231191	9.157888	9.084585	6.206685	6.136793	6.066901
.2	0.374823	12.026691	11.909559	11.792427	8.986847	8.874085	8.761324
.3	0.560391	16.642420	16.444058	16.245697	13.632180	13.438468	13.244756
.4	0.761559	30.261921	29.586872	28.911822	27.233793	26.561356	25.888919
.5	0.958089	172.446798	158.950769	145.454740	169.431103	155.937558	142.444013
.6	1.143331	7432.436896	6998.229436	6564.021976	7429.449603	6995.241274	6561.032945
.7	?	?	?	?	?	?	?

p_1	E[X1]			E[Nq1]		
.1	5.621536	5.580809	5.540082	0.203396	0.200528	0.197660
.2	5.692166	5.659825	5.627484	0.589077	0.580719	0.572360
.3	5.665452	5.641271	5.617091	1.346364	1.323841	1.301318
.4	5.772736	5.749772	5.726807	3.596013	3.499569	3.403126
.5	5.808675	5.786859	5.765043	28.367295	26.076413	23.785532
.6	5.781967	5.754765	5.727563	15562.143689	14452.311862	13342.480034
.7	?	?	?	62196.961535	59820.390217	57443.818899

p_1	E[T2]			E[W2]			E[Nq2]		
.1	2.492239	2.477406	2.462574	2.236239	2.221406	2.206574	3.772244	3.745248	3.718252
.2	2.494105	2.478258	2.462410	2.238105	2.222258	2.206410	3.753412	3.725201	3.696991
.3	2.491531	2.478769	2.466008	2.235531	2.222769	2.210008	3.756649	3.732375	3.708101
.4	2.501029	2.487321	2.473614	2.245029	2.231321	2.217614	3.769063	3.742997	3.716931
.5	2.501542	2.484329	2.467116	2.245542	2.228329	2.211116	3.771792	3.740766	3.709740
.6	2.491954	2.475822	2.459689	2.235954	2.219822	2.203689	3.773499	3.745192	3.716886
.7	2.469773	2.453989	2.438205	2.213773	2.197989	2.182205	3.734019	3.705137	3.676254

p_1	E[Δ]			V(Δ)		
.1	2.477163	2.373409	2.269655	974.214569	816.902013	659.589456
.2	2.574284	2.463730	2.353175	1164.047178	985.407373	806.767569
.3	2.420681	2.318843	2.217006	775.978087	663.697197	551.416308
.4	2.499819	2.398636	2.297454	1012.844654	836.712328	660.580001
.5	2.551069	2.447046	2.343022	957.677394	840.083170	722.488946
.6	2.424878	2.328999	2.233119	980.670130	796.930780	613.191431
.7	2.359154	2.248789	2.138425	957.633327	756.456819	555.280311

2.3. Avaliação

Antes de comecar, é preciso constar que os testes com p = 0.6 e p = 0.7 para os casos com interrupção foram executados com 90 rodadas em vez de 100, devido à falha *killed*. Os valores em vermelho indicam essas simulações, que por um acaso também são as que possuem resultados instáveis.

Como pode-se observar, o valor da utilização se desvia muito do esperado quando se trata de testes com interrupção. A quantidade de pacotes de voz na fila não costuma oscilar muito, mesmo quando não há interrupção; já seu tempo de fila é levemente afetado.

Por precisar de um tempo de serviço em média quase 12 vezes maior, e por menos prioridade que os pacotes de voz, o aumenta na quantidade de pacotes de dados sempre acaba causando um grande aumento no seu tempo de espera de fila, o que acaba refletindo no seu tempo total. No caso de execuções com interrupção, a situação se agrava ainda mais.

Ficou constatado que, em algumas rodadas, nenhum pacote de dados introduzido na atual rodada conseguia ser servido ainda nela, devido à grande fila já presente e às frequentes interrupções causadas por pacotes de voz. Nesses casos, esses pacotes foram ignorados nos cálculos de esperança, para que algum valor final pudesse ser provido que não fosse infinito. Esse problema ocorreu exclusivamente para p = 0.6 nos casos com interrupção. Em p = 0.7, nem mesmo isso foi suficiente para que um valor pudesse ser provido, pois nenhuma rodada teve o serviço de um pacote de dados dela concluído dentro de si.

3. CONCLUSÕES

3.1. Desenvolvimento

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).

3.2. Resultados

Podemos afirmar com clareza que se a interrupção for permitida nesse caso, a chegada de pacotes de dados será altamente comprometida. Além disso, a interrupção não traz um benefício tão grande aos pacotes de voz ao ponto de compensar o prejuízo aos pacotes de dados.

Para o caso de p = 0.7 sem interrupção, o tempo médio que um pacote de dados leva para ser servido desde a sua chegada é de aproximadamente 0.6 segundos em média. Com interrupção, esse valor nem mesmo pôde ser determinado; mas com p = 0,6, ele já era de 7 segundos.