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/***.

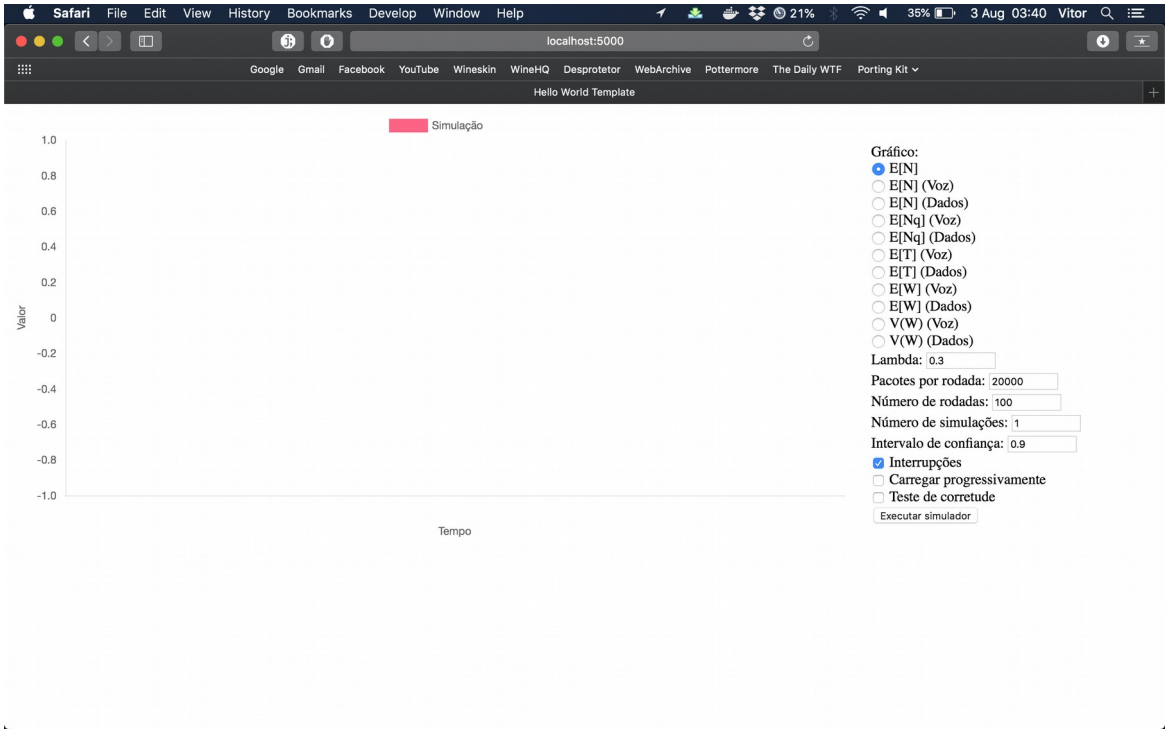
Devido à natureza de poder funcionar através de um Docker, a aplicação pode ser executada virtualmente falando a partir de qualquer sistema operacional direcionado a computadores, como macOS, Windows, e múltiplas distribuições Linux. Ao ser executado

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.

O valor usualmente adotado é 400.000. Em testes com a função *random.expovariate* do Python, a média de 100.000 execuções é bem próxima do valor experado; ou seja, se essa função é chamada 100.000 vezes, os resultados somados e divididos por 100.000 são muito próximos ao valor médio.

No entanto, a função *numpy.random.geometric*, que é usada para distribuições geométricas, necessita de 400.000 execuções para alcançar esse mesmo nível de estabilidade.

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.

Os testes de corretude demoraram a serem aplicados, levando aos problemas descobertos que são citados no fim do subcapítulo 1.3.3. A grande divergência que levaram levou a novos testes que comprovaram a necessidade de aumentar a quantidade de pacotes por rodada.

Apesar disso, mesmo depois de todas as funções geradoras terem sido testatas, ainda se testemunharam divergências no resultado entre o teste de corretude e o resultado final. Em um teste com 20 rodadas e 400.000 pacotes, porém com a chegada de pacotes de dados desabilitada, a média de tempo que um pacote de voz leva no sistema, por exemplo, é de aproximadamente 2.90 ms no teste, mas 2.49 ms fora dele (com um intervalo de confiança de aproximadamente 0.006 ms para cima e para baixo).

Como o tempo de atendimento em ambos os casos é de 0.256 ms, a diferença se encontra na durança do tempo de fila, que só é afetada pelo quantidade de pacotes por serviço e pelo intervalo entre dois servicos; porém ambos foram testados fazendo a média de 100.000 amostras, e ambos chegaram muito próximos do valor médio esperado.

Mesmo após inumeros testes e mudanças de parâmetros, os resultados se mantiveram assim. Sendo assim, tudo foi feito levando em conta que essa marge existe. Ironicamente, ao executar o último caso do relatório em modo de corretude (p = 0.7 com interrupção), o valor de E[X] de dados e de E[X] de voz estavam exatamente onde esperados, porém os tempos de fila e cada um continuaram diferentes, o que é bem estranho dado que, em média, os pacotes levaram o mesmo tempo para serem atendidos, indicando que a disparidade só poderia estar no número de pacotes de voz por serviço e no intervalo entre dois serviços de voz, os quais ambos geradores foram testando calculando a média de 100.000 de suas amostras, e comprovadamente produzem os valores esperados.

No entanto, no fim, descobriu que haviam algumas falhas no simulador. Depois que elas foram corrigidas, os resultados do teste de corretude passaram todos a ficar bem próximos do calculado. No teste de p = 0.1, visualizada no capítulo, a maior margem de diferença encontrada foi de 0.29 ms no tempo de fila. Considerando que o teste foi feito com menos amostras, o resultado acabou sendo bem aceitável.

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 se trabalhar, fazendo com que a aplicação levasse tanto tempo para atingir o primeiro estado não-transiente que o Flask retornava *killed*, finalizando o processo sem dar uma resposta.

Contatou-se que esse problema ocorre quando a aplicação consume muito mais memória do que deveria, e isso foi resolvido para outros casos, como será mencionado posteriormente, usando o programa *Memory Clean* para liberar espaço na RAM. De uma maneira ou de outra, a mudança desse valor corrigiu o problema. No entanto, esse problema foi útil para perceber que a simulação estava, desnecessariamente, aguardando um equilíbrio alto demais.

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.

Originalmente, testes foram feitos usando 20 rodadas, cada uma com 400.000 pacotes. Antes usaram-se 100.000 pacotes, apesar de com esse valor haverem algumas sinuosidades nos resultados, pois quantidades grandes, como 200.000 pacotes e 100 rodadas, causavam o erro *killed* mencionado anteriormente. Após usar a aplicação Memory Clean para liberar espaço na RAM, os testes puderão ser feitos com uma medida mais precisa.

Como mencionado no subcapítulo 1.3.4, a quantidade de pacotes não foi escolhida deliberadamente. Em testes isolados, foi com a média de 400.000 amostras que as distribuições geométrica e exponencial chegaram praticamente a seus valores médios esperados.

Apesar disso, a distruição geométrica não chega a ser executada 400.000 vezes, dado que só é chamada quando um novo serviço de voz é iniciado, o qual é composto por vários pacotes de voz. Apesar disso, acaba sendo usada um número suficiente de vezes para a média ficar bem próxima do esperado. No entanto, no final, por uma questão de tempo execução, depois que várias correções foram feitas, foram usadas 100.000 amostras e 50 rodadas.

2.1. Sem interrupção

p_1	E[T1]			E[W1]		
.1	3.350384	3.345120	3.339855	0.327468	0.324820	0.322171
.2	3.749666	3.744633	3.739600	0.727890	0.724671	0.721451
.3	4.274508	4.267663	4.260818	1.256224	1.250665	1.245106
.4	5.005235	4.997278	4.989320	1.985096	1.977891	1.970686
.5	6.081725	6.067150	6.052576	3.061513	3.047951	3.034388
.6	7.796434	7.770721	7.745008	4.774459	4.750051	4.725642
.7	11.130552	11.073663	11.016773	8.112336	8.056192	8.000049

p_1	E[X1]			E[Nq1]		
.1	3.024308	3.020300	3.016292	0.009529	0.009437	0.009346
.2	3.022860	3.019963	3.017065	0.043758	0.043530	0.043302
.3	3.019489	3.016998	3.014506	0.115748	0.115142	0.114536
.4	3.021501	3.019387	3.017273	0.248110	0.247100	0.246091
.5	3.021270	3.019200	3.017129	0.484880	0.482524	0.480167
.6	3.022799	3.020670	3.018542	0.918669	0.913617	0.908565
.7	3.019375	3.017471	3.015566	1.844936	1.831219	1.844936

p_1	E[T2]			E[W2]			E[Nq2]		
.1	0.540134	0.539076	0.538018	0.284134	0.283076	0.282018	0.028395	0.028127	0.027860
.2	0.812944	0.811368	0.809792	0.556944	0.555368	0.553792	0.055086	0.054584	0.054083
.3	1.083242	1.081289	1.079336	0.827242	0.825289	0.823336	0.079408	0.078812	0.078216
.4	1.359877	1.357605	1.355332	1.103877	1.101605	1.099332	0.105793	0.104976	0.104159
.5	1.636599	1.633342	1.630085	1.380599	1.377342	1.374085	0.132823	0.131495	0.130167
.6	1.914016	1.910116	1.906216	1.658016	1.654116	1.650216	0.159595	0.158046	0.156496
.7	2.189275	2.186065	2.182855	1.933275	1.930065	1.926855	0.186915	0.185282	0.183649

p_1	E[Δ] (segundos)			V(Δ) (segundos)		
.1	0.702420	0.685930	0.669440	0.104188	0.099074	0.093959
.2	1.260389	1.170206	1.080024	0.489474	0.422376	0.355278
.3	1.633380	1.494764	1.356149	0.744981	0.624355	0.503730
.4	2.555894	2.353453	2.151013	1.773974	1.541467	1.308959
.5	1.595043	1.428972	1.262901	0.778132	0.626545	0.474957
.6	1.946686	1.820392	1.694098	0.970790	0.861824	0.752859
.7	2.282807	2.045033	1.807259	1.524210	1.264761	1.005311

2.2. Com interrupção

p_1	E[T1]			E[W1]		
.1	3.657201	3.651150	3.645098	0.630882	0.627440	0.623997
.2	4.210456	4.204629	4.198801	1.192714	1.187591	1.182468
.3	5.024049	5.015275	5.006502	1.998310	1.990250	1.982191
.4	6.195577	6.179569	6.163561	3.173039	3.158161	3.143283
.5	8.090509	8.061654	8.032798	5.072900	5.044839	5.016778
.6	11.974460	11.897732	11.821004	8.954872	8.879281	8.803691
.7	23.012043	22.737749	22.463455	19.994741	19.720814	19.446887

p_1	E[X1]			E[Nq1]		
.1	3.027573	3.023710	3.019847	0.012772	0.012676	0.012580
.2	3.019491	3.017037	3.014584	0.059929	0.059608	0.059286
.3	3.027994	3.025025	3.022056	0.165126	0.164417	0.163708
.4	3.023758	3.021408	3.019058	0.370094	0.368050	0.366005
.5	3.019182	3.016815	3.014447	0.770823	0.766065	0.761307
.6	3.020335	3.018450	3.016566	1.688435	1.673164	1.657893
.7	3.018864	3.016935	3.015006	4.520064	4.455738	4.391412

p_1	E[T2]			E[W2]			E[Nq2]		
.1	0.267904	0.267732	0.267559	0.011904	0.011732	0.011559	0.002628	0.002585	0.002543
.2	0.267907	0.267734	0.267561	0.011907	0.011734	0.011561	0.002611	0.002568	0.002525
.3	0.267533	0.267373	0.267214	0.011533	0.011373	0.011214	0.002528	0.002487	0.002446
.4	0.267772	0.267604	0.267435	0.011772	0.011604	0.011435	0.002582	0.002545	0.002507
.5	0.268085	0.267906	0.267726	0.012085	0.011906	0.011726	0.002651	0.002608	0.002564
.6	0.267859	0.267677	0.267494	0.011859	0.011677	0.011494	0.002589	0.002544	0.002500
.7	0.268194	0.268005	0.267816	0.012194	0.012005	0.011816	0.002691	0.002644	0.002597

p_1	E[Δ] (segundos)			V(Δ) (segundos)		
.1	0.827030	0.781979	0.736929	0.157309	0.142731	0.128154
.2	1.558618	1.437259	1.315899	0.517551	0.453747	0.389944
.3	0.716605	0.676321	0.636036	0.127628	0.114651	0.101675
.4	0.710379	0.700470	0.690562	0.105369	0.100905	0.096442
.5	0.660802	0.629438	0.598073	0.098968	0.090609	0.082251
.6	0.707998	0.690032	0.672066	0.110837	0.103596	0.096355
.7	0.639704	0.614751	0.589799	0.092585	0.086042	0.079500

2.3. Avaliação

Por precisar de um tempo de serviço em média quase 12 vezes maior, e por menos prioridade que os pacotes de voz, o aumento 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.

Curiosamente, E[Δ] parece ter um valor basicamente “aleatório”; o mesmo se aplica a V(Δ). É difícil dizer se há algum padrão ou se a redução/aumento de alguns dos valores é apenas coincidência.

Sobre o tempo de transmissão dos pacotes de dados: considerando que os pacotes pesam em média 512 bits e os codificadores são de 32 Kbps, a transmissão levaria cerca de 16 milisegundos em cada lado em média, ou seja, 32 ms em média, deixando ainda 168 ms para o pacote entrar na fila e ser atendido. Com ou sem interrupção, pacotes de voz ficam bem longe de atingir esse caso, chegando a no máximo aproximadamente 2 ms em média quando não há interrupção, devido à prioridade dos pacotes de voz.

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 levará aproximadamente 2x mais tempo para ocorrer no pior dos casos. enquanto os pacotes de voz ficarão, nessa mesma circunstância, aproximadamente 8x mais rápidos.

Considerando isso, a escolha de um ou do outro dependerá muito do que deseja-se dar prioridade.