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

Simulador de Roteador

1. INTRODUÇÃO

1.1. Linguagens e Ferramentas

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 na 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 VMs 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/***.

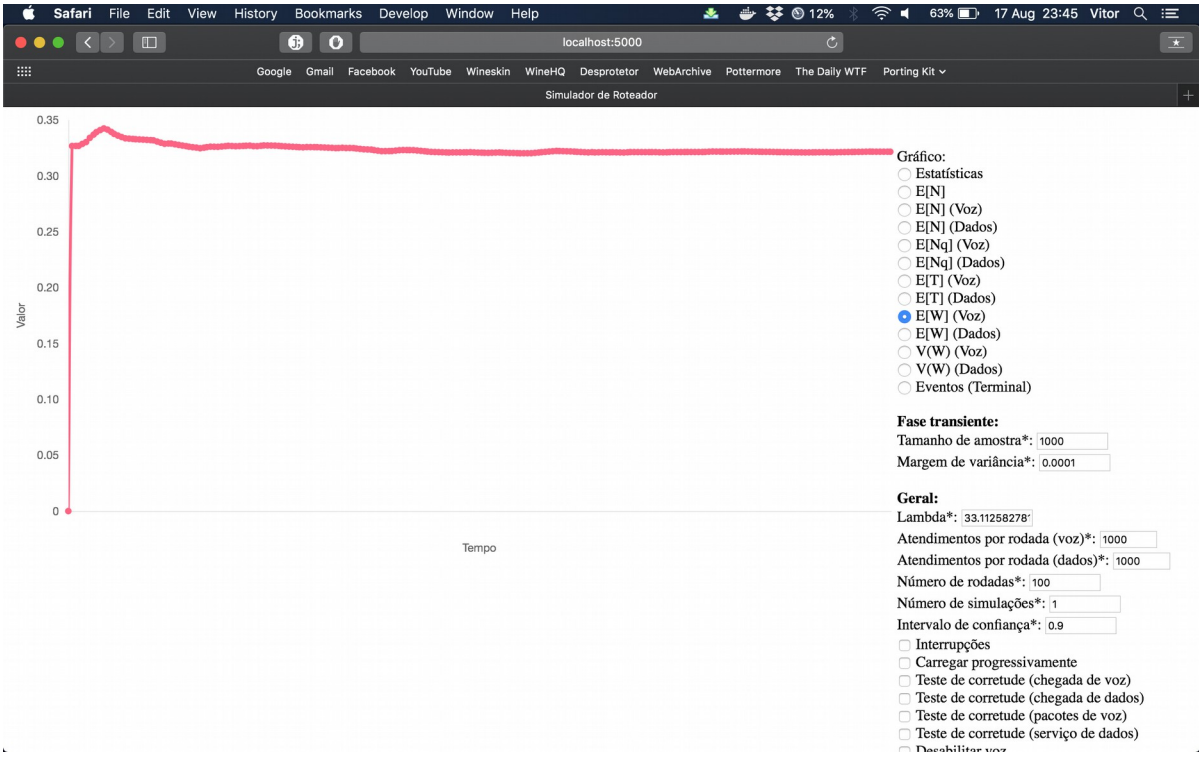
Devido à natureza de poder funcionar através de um Docker, a aplicação pode ser executada virtualmente falando a partir de qualquer sistema operacional direcionado a computadores, como macOS, Windows, e múltiplas distribuições Linux. Ao ser executado

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 a 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 do simulador (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 marcada, o gráfico será plotado aos poucos, sem que haja a necessidade de aguardar até o fim da execução da aplicação do backend. Este recurso é utilizado apenas ao plotar gráficos, e não na geração de relatórios, que é o caso das opções *Estatísticas* e *Eventos*.

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 (cada uma contendo o valor selecionado como medido após cada evento, acompanhado do índice da rodada, o qual será usado para determinar a cor) . Esses gráficos contam com os valores médios, também podendo plotar os intervalos se a

caixa for marcada.

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.

Em uma execução não-progressiva, o ChartJS apenas plota 100 das linhas que receber da aplicação para cada rodada, sendo estar distribuídas simetricamente entre as linhas recebidas. Isto é, se existem 3.001 linhas, serão plotadas as linhas 1, 31, 61, … até 2971 e 3001. Isso é feito para não exigir muito tempo de renderização com linhas que não fariam nada além de dificultar a visão do que é relevante no gráfico.

1.3. Funcionamento do simulador (Backend)

1.3.1. Filas

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.

Pacotes de dados só são atendidos se chegarem em um sistema vazio; caso contrário, vão para o fim da fila de dados. Em um sistema com interrupção, a chegada de um pacote de voz faz um pacote de dados em atendimento voltar para a fila, reiniciando seu serviço quando a fila de voz ficar novamente vazia.

Por consequência, pacotes de voz são atendidos logo ao chegarem se não houver outro pacote de voz no sistema cajo haja interrupção no sistema. Em quaisquer outro caso, ele vai para o fim da fila de voz, onde ele terá prioridade sobre os pacotes de dados na hora de ser atendido.

1.3.2. Eventos

Para que um novo evento seja processado, é preciso que ele seja adicionado à lista de eventos. A simulação começa com 31 eventos de chegadas de pacotes (1 de dados e 30 de voz), sendo que os eventos de voz tem um período de silêncio antes de entrarem na fila, e estes eventos desencadeiam os demais.

Logo que um pacote de voz chega, um evento de chegada de pacote de voz é adicionado à lista de eventos, agendando assim a chegada do próximo pacote de voz (isso não ocorre se este for o último pacote de voz de um serviço; neste caso, quem pode lançar esse evento é o evento de fim do serviço deste mesmo pacote, pois este serviço já acabou).

Quando um pacote de voz termina o seu serviço, as duas filas (voz e dados) são verificadas para ver quem será o próximo a ser atendido. Se houver alguém na fila de voz, ela terá prioridade. Caso não tenha, o primeiro da fila de dados será atendido. Se não houver ninguém, nenhum novo evento será lançado (com exceção do evento mencionado anteriormente neste parágrafo, se este for o último pacote de voz de um serviço).

Quando um pacote de dados chega, um evento de chegada de pacote de dados é adicionado à lista de eventos, agendando assim a chegada do próximo pacote de dados. Quando um pacote de dados termina o seu serviço, as duas filas (voz e dados) são verificadas para ver quem será o próximo a ser atendido. Se houver alguém na fila de voz, ela terá prioridade. Caso não tenha, o primeiro da fila de dados será atendido. Se não houver ninguém, nenhum novo evento será lançado.

Cada objeto evento na fila possui o tempo em milisegundos que é preciso aguardar para que ele ocorra. Os eventos são ordenados do que possui menos tempo para o que possui mais. O evento que possuir o menor tempo é o primeiro na ordem cronológica, e por isso ocorre primeiro, sendo chamado para executar. Após sua execução, a variável de tempo atual tem o seu valor de tempo somado, e esse mesmo valor de tempo é deduzido do tempo faltante dos demais eventos. Depois disso o processo se repete, até que não hajam mais eventos ou que a execução seja interrompida.

Em relação à chegada de novos pacotes de voz: para cada serviço, composto por um certo número de pacotes, a chegada do último pacote é seguida de uma espera de 16 ms, mais uma espera exponencial com média de 650 ms, o qual é gerado com o mesmo método usada para determinar as chegadas de pacotes de dados, mas com lambda = 1/650 (um pacote a cada 650 ms).

1.3.3. Números aleatórios

Para gerar valores de distribuições exponenciais, foi usado o método *random.expovariate* do Python, o qual funciona ao receber um valor lambda (ocorrências/tempo, ou no caso utilizado, pacotes/ms). Essas distribuições são usadas no tempo para a chegada de um pacote de dados e no tempo para a chegada do primeiro pacote de voz após um período de silêncio e no início da execução.

Cada canal trabalha de maneira independente, iniciando após um período de silêncio com distribuição exponencial de média 650, onde inicia-se a chegada de uma remessa de pacotes de mesmo tamanho, com intervalo de chegada de 16ms entre si. Após a chegada

do último pacote dessa remessa, aguardam-se 16ms, e então começa outro período de silêncio, após o qual outra remessa é iniciada.

Para gerar valores de distribuições geométricas, foi usado o método *numpy.random.geometric* disponível no pacote NumPy, o qual também funciona ao receber um valor ocorrências/tempo. Essa distribuição é usada para gerar a quantidade de pacotes de voz em um intervalo, valor este que é arredondado para o mais próximo por precisar ser um inteiro.

No caso da escolha do tamanho do pacote de dados, uma distribuição mista com pdf, foi criado uma método manual. Para cada valor de 64 a 1500, existe uma condição que determina a sua escolha baseando-se na fórmula passada no enunciado. Primeiro um número aleatório entre 0.0 e 1.0 é gerado:

- **Se o número é menor que 0.3 + 3/14360,** retorna 64
- **Senão,** subtrai 0.3 + 3/14360 do número gerado
- **Se o número é menor que 1341/14360,** retorna 65 + (o número)*14360/3
- **Senão,** subtrai 1341/14360 do número gerado
- **Se o número é menor que 0.1 + 3/14360,** retorna 512
- **Senão,** subtrai 0.1 + 3/14360 do número gerado
- **Se o número é menor que 2961/14360,** retorna 513 + (o número)*14360/3
- **Senão,** retorna 1500

Essa foi a maneira mais eficiente encontrada para trazer a resposta de forma acurada.

1.3.4. Método de simulação

O método adotado para a avaliação do simulador foi o método Batch, para comparar rodadas após uma fase transiente, que pareceu ser a proposta mais interessante.

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 diferença de pelo menos 100.000.000 das sementes geradas anteriormente; caso contrário, uma nova semente é gerada.

No caso, cada semente é única por simulação. Se a aplicação for executada requisitando mais de uma simulação com os mesmo parâmetros, a semente será trocada entre uma simulação e outra, criando várias simulações Batch consecutivas e não-relacionadas.

1.3.5. Final de rodada

Os valores passados de número de eventos de serviço de voz e dados por rodada definem quando uma rodada termina, e o número de rodadas que devem ser executadas também é definido como parâmetro. Se, digamos, 1.000 eventos de término de dados e 1.000 eventos de término de voz estão definidos, então a chegada do primeiro evento após o 1.000º término de voz ou de dados (o que vier por último) iniciará a rodada seguinte, e isso se repetirá até que não hajam mais rodadas.

Assim que um pacote entra no sistema, é armazenado nele o índice da rodada ao qual ele pertence, e uma referência a ele é guardada em uma lista exclusiva dos pacotes de sua rodada. Se a rodada acabar e outra se iniciar, ele passa a ser um pacote de uma rodada anterior.

1.4. Parâmetros utilizados

Ignorando os testes enquanto o simulador estava incompleto ou defeituoso, os parâmetros acabaram definidos como:

Para simulações sem interrupção:
- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.000** atendimentos de voz por rodada
- Mínimo de **1.000** atendimentos de dados por rodada
- **100** rodadas

Para simulações com interrupção, e para simulações sem pacotes de dados:
- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **20.000** atendimentos de voz por rodada
- Mínimo de **0** atendimentos de dados por rodada
- **100** rodadas

1.4.1. Escolha dos parâmetros

Após executar o simulador com os parâmetros para não-interrupção, habilitando todos os teste de corretude em uma execução normal (ambos com lambda para p = 0.1), observou-se imediatamente que havia uma diferença nos resultados. O tempo de fila médio para pacotes de dados estava bem maior que o esperado (0.450710 contra 0.062114), e o tempo de fila médio para pacotes de voz estava levemente menor (0.322180 contra 0.362431), apesar de, neste último caso, estar dentro do intervalo de confiança do teste.

Ao executar o mesmo processo, mas com 20.000 atendimentos de cada tipo e 50 rodadas (o qual foi bem demorado), uma boa e uma má notícia. A boa notícia é que quase nada mudou no caso não-teste, indicando que os valores anterior, enquanto bem menores, ainda eram funcionais. A má notícia é que o caso teste não mudou o suficiente, fazendo com que os tempos de fila do teste de corretude de dados (0.441683 contra 0.041048) e de voz (0.321762 contra 0.244345) continuassem diferentes.

Para encontrar a razão da instabilidade, foi feito o mesmo teste inicial, porém sem nenhum pacote de voz, e com 250 rodadas. O resultado foi que no teste de corretude o tempo de fila era zero, enquanto fora do teste era quase zero. No caso de teste: para p = 0.1, lambda = 3.31125827815 pacotes por milissegundo, enquanto o tempo de atendimento de um pacote de dados é de 3.0192 ms, fazendo com que quando o pacote seguinte chegue, o anterior já tenha sido atendido, tornando o resultado plausível. Posteriormente, observou que houve uma falha na implementação do simulador, já que os pacotes de voz iniciais deveriam ter um tempo de silêncio no início da execução, mas não o pacote de dados inicial.

O mesmo foi feito no sentido contrário: desabilitando os pacotes de dados. O resultado foi um tempo de fila bem diferente. 3.712000 ms no teste e 0.024473 fora dele. Apesar disso, o E[X] dos pacotes de voz mantinha-se o esperado. Aumentei então o número de atendimentos mínimo de pacotes de voz para 30.000. Nenhuma mudança.

Forçando os serviços de voz a ter a mesma quantidade de pacotes (22), o problema se manteve exatamente como antes. O mesmo ocorreu ao manter a distância entre serviços de voz constante. No entanto, ao fazer com que apenas o primeiro canal de cada pacote do teste de corretude fosse aleatório, com o resto sendo absoluto, obteve-se o mesmo resultado para ambos. A conclusão para isso foi que: como todos os canais começam ao mesmo tempo, eles constantemente estavam na fila ao mesmo tempo, causando um tempo médio de fila muito maior.

Desta maneira, ficou claro que não era apropriado realizar o teste do tempo de espera de pacotes de voz com uma M/D/1, e em vez disso foi usada uma M/G/1. Uma vez com isso em mente, a situação foi resolvida.

Para os casos com interrupção, tendo a noção de que não é possível esperar que um pacote de dados sempre chegue, devido ao fato de que o tempo de serviço deles aumenta sem parar, os parâmetros usados para este caso não exigem nenhum pacote de dados, mas exigem 20.000 pacotes de voz, na tentativa não só de equilibrar o tempo para lambda de p = 0.1, mas também de conseguir algum pacote de dados.

1.5. Máquina utilizada

Dois computadores foram utilizados para fazer o trabalho. O primeiro foi utilizado por um prazo muito curto, e mal chegou a ter simulações executadas nele. O segundo computador possui as seguintes especificações:

Processador: Intel Core i5 2,6 GHz
Memória RAM: 8Gb
Disco (SSD): 500Gb (41.3Gb available)
Sistema operacional: macOS 10.14.0

O tempo gasto para executar os resultados presentes no relatório com esta máquina se encontra nas tabelas de resultados. Os tempos de execução na primeira máquina foram com versões antigas do simulador, e por isso foram desconsiderados.

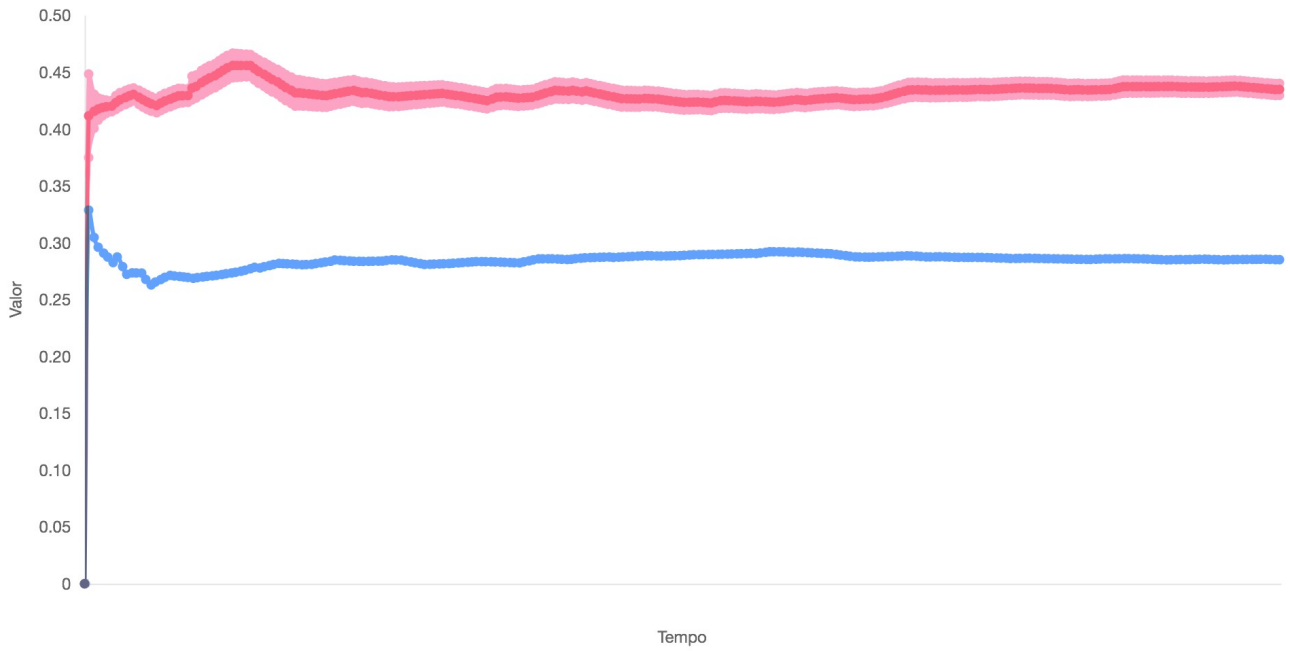
2. TESTES DE CORRETUDE

(Os gráficos deste capítulo foram plotados com lambda para p = 0.1. Todos usam os parâmetros escolhidos no capítulo 1.4, e são as esperanças do sistema conforme as rodadas passam, e não as esperanças por rodada. Azul é o valor do teste de corretude, vermelho é o valor indicado, e as duas faixas rosas indicam o intervalo de confiança do valor indicado.)

Existem múltiplas metodologias para executar testes de corretude. Forçar os valores médios ou forçar a semente são algumas delas, mas generalizar isso se provou inapropriado para o simulador em questão.

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.

Forçar os valores médios (M/D/1) consiste em retornar valores constantes e absolutos sempre, ou seja, o valor médio esperado da variável. Um exemplo: o teste do tempo de serviço de um pacote de dados já foi a média do tamanho dos pacotes de dados (em bits; 6.038,4) dividido pela velocidade do meio (2.000, ao converter 2 Mbps para bits por microsegundo), o que dá 3,0192. Infelizmente, o resultado neste caso divergia muito da simulação.

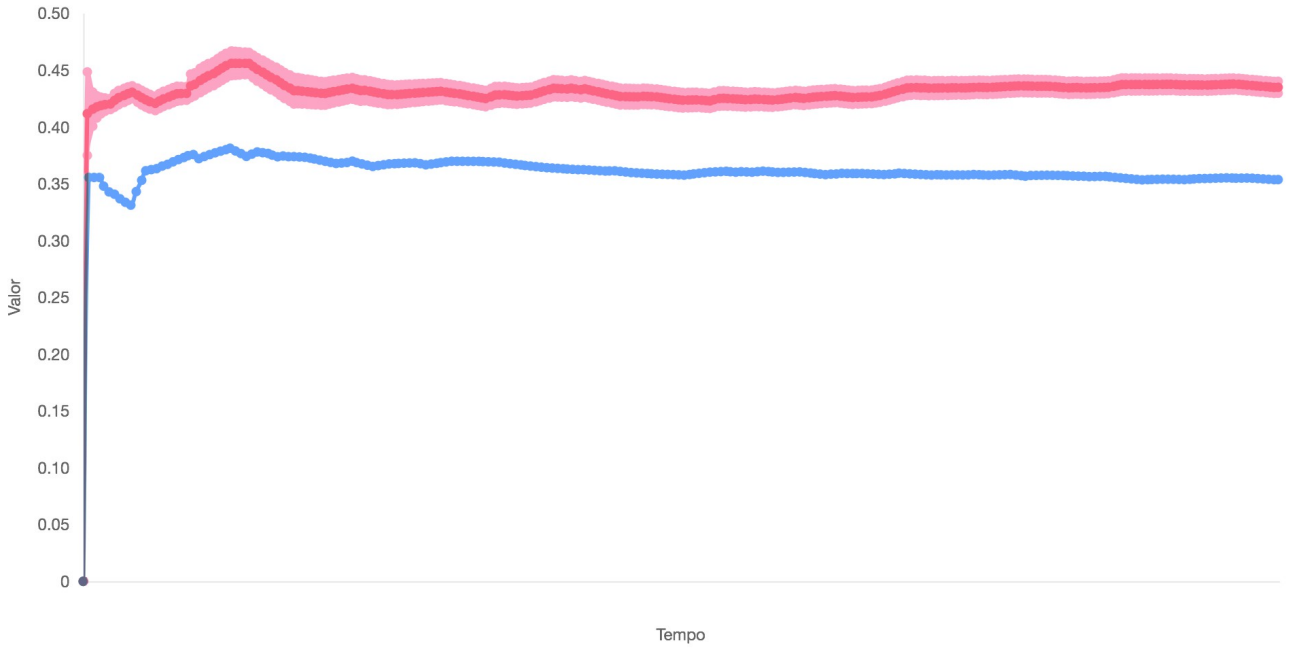


E[W] de pacotes de dados sem interrupções, com teste de tempo de serviço de pacotes de dados M/D/1

Imaginou-se que o ideal seria uma função que pudesse alternar entre 64 e 1500, mas cuja média fosse 755. Para adquirir esse valor, são gerados dois números aleatórios de 0.0 a 1.0 (os chamaremos de valor decisor e valor medidor). Com esses números em mãos:

- Se o valor decisor for abaixo de 0,4812, retorna **755 – 691*valor medidor**
  - Obs.: 0,4812 ≅ (755-64)/(1500-64)
- Caso contrário, retorna **755 + 745*valor medidor**

Assim adquirimos o tamanho do pacote em bytes, com uma geração aleatória que converge para o mesmo valor, apesar de não ter a mesma distribuição. O resultado, apesar de melhor, ainda não era apropriado.



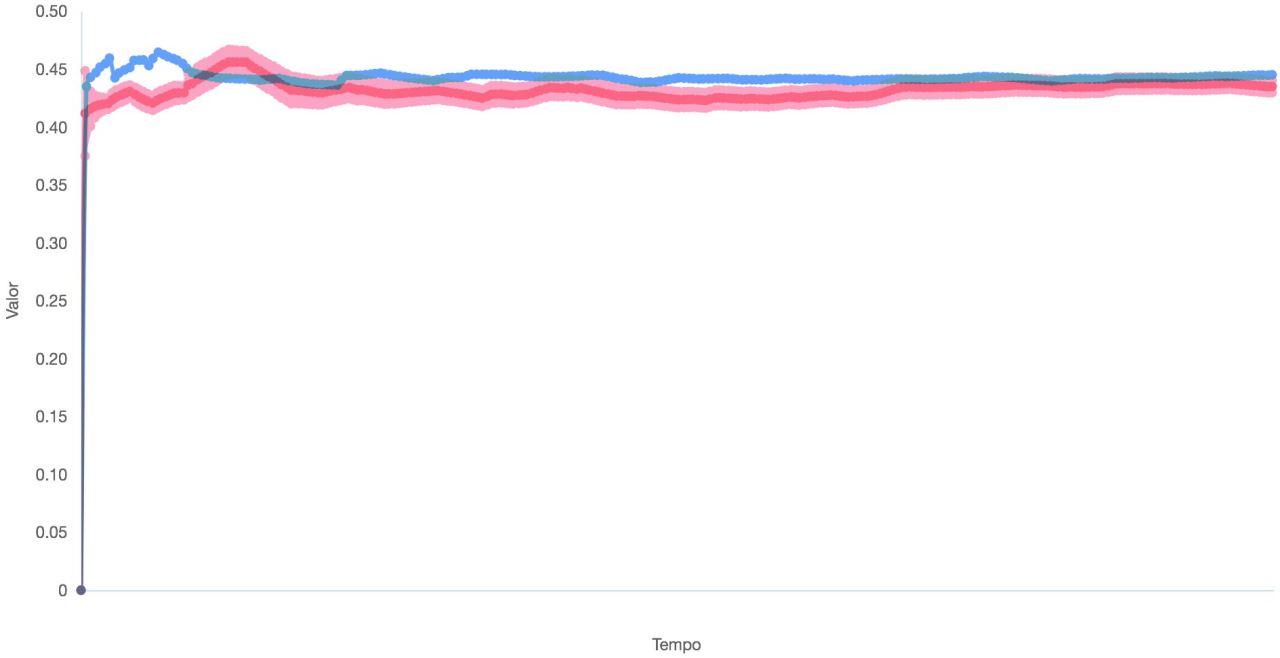
E[W] de pacotes de dados sem interrupções, com teste de tempo de serviço de pacotes de dados customizado

Certamente, a causa disso foi a diferença na distribuição dos valores. Debatendo a questão com outras pessoas, foi sugerida a possibilidade de se integrar a distribuição mista. Mesmo sem saber ao certo o que isso traria, esse processo foi experimentado, e baseado nos métodos descritos na página 33 da apostila da disciplina, foi criado um algoritmo, que será exemplificado abaixo. Antes de apresenta-lo, informo que foi testado o somatório das novas probabilidades, e observou-se que eles beiravam 745 em vez de 1.0. Para resolver a situação, o número aleatório gerado é multiplicado pelo somatório das probabilidades antes do procedimento começar:

- **Se número é menor que 0.3:** retorna 64
- **Senão,** subtrai 0.3 do número
- **Se o número é menor que 447*0.3 + 447!*0.3/1436:**
  - Seta índice como 1
  - **Se o número é menor que 0.3 + índice*0.3/1436:** retorna 64 + índice
  - **Senão,** subtrai 0.3 + índice*0.3/1436 de número, então soma 1 no índice e retorna para o subitem anterior
- **Se o número é menor que 0.3 + 448*0.3/1436 + 0.1:** retorna 512
- **Senão,** subtrai 0.3 + 448*0.3/1436 + 0.1 do número
- **Se o número é menor que 987*0.3 + (1436!/449!)*0.3/1436 + 987*0.1:**
  - Se índice como 449
  - **Se o número é menor que 0.3 + índice*0.3/1436 + 0.1:** retorna 64 + índice
  - **Senão,** subtrai 0.3 + índice*0.3/1436 + 0.1 de número, então some 1 no índice e retorna para o subitem anterior
- Retorna 1500

A média de 20.000 diferentes execuções dessa algoritmo deu um número superior 800, fazendo-me crer que haveria algo errado. Mesmo assim, o código foi testado, e o

resultado conseguiu superar as expectativas:



E[W] de pacotes de dados sem interrupções, com teste de tempo de serviço de pacotes de dados a base da integral da distribuição mista

Já o tempo para chegada de um novo pacote de dados é uma exponencial, e utilizar seu valor médio traz resultados inapropriados se feito em conjunto com o teste da chegada de pacotes de voz, pois desta forma ocorrem chegadas e intervalos regulares, simulando uma circunstância muito improvável em vez de médias esperadas.

Para contornar isso, em vez de uma M/D/1, foi usada uma M/G/1 para testar os casos exponenciais, ou seja, uma distribuição geométrica com a mesma configuração da distribuição exponencial. Isso vale para a chegada de novos pacotes de dados e para a chegada de novos serviços de voz por canal.

Ainda assim, houveram problemas com a chegada de novos pacotes de dados no teste de corretude. Uma análise mais profunda mostrou que como a distribuição geométrica do Python sempre retornava números inteiros, isso tornava os resultados retornados pouco precisos. Para resolver a situação, lambda passou a ser dividido por 1 milhão em vez de mil, e o resultado obtido pela distribuição é transformado em float e então dividido por mil. Com isso, a situação foi resolvida.

A quantidade de pacotes de voz por serviço de canal, por outro lado, é uma M/G/1, então seu teste de corretude foi feito inicialmente com a implementação do valor esperado médio (no caso, 22). Será visto mais a frente que isso foi revisado, e então alterado.

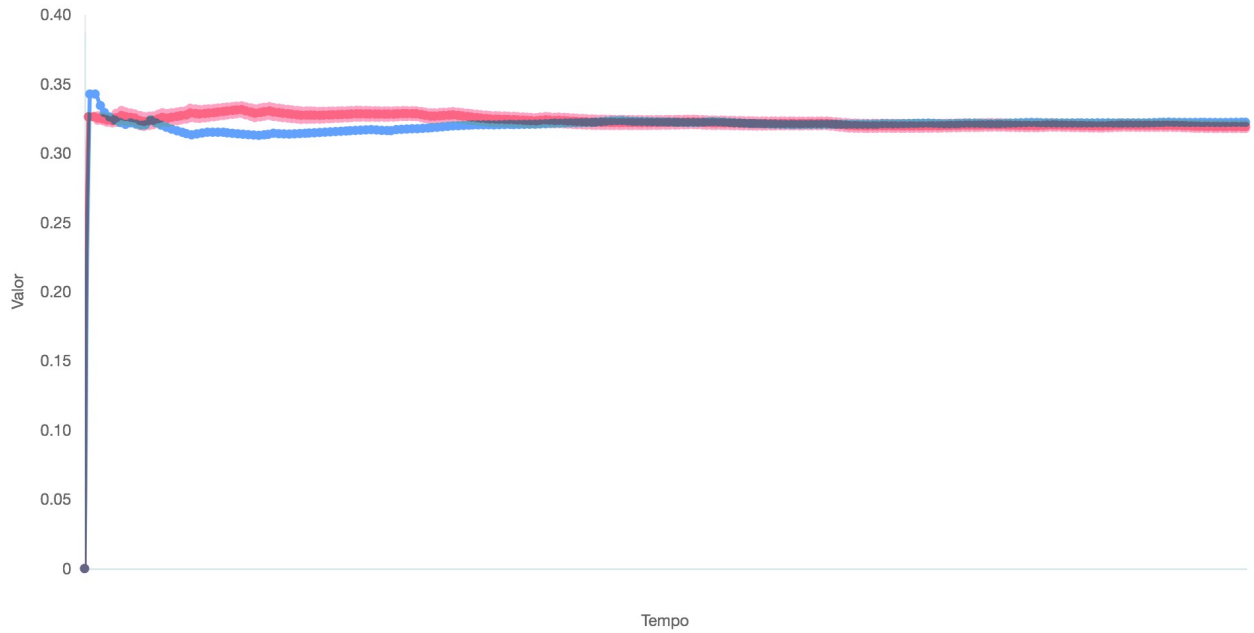
Para permitir uma melhor avaliação dos testes, o frontend permite que seja executado o teste de correção para cada um desses valores de forma individual. Esses testes de corretude isolados serão usados para provar a eficácia dos resultados do simulador mais adiante.

Quando os quatro testes citados acima (chegada de atividade de voz, chegada de pacotes de dados, quantidade de pacotes de voz e serviços de pacotes de dados) foram aplicados em uma execução com os parâmetros padrões com interrupção com lambda para p = 0.1, o resultado veio de maneira bem mais rápida do que na simulação real, levando pouco mais de 90 segundos. Isso indicou que havia uma falha no teste.

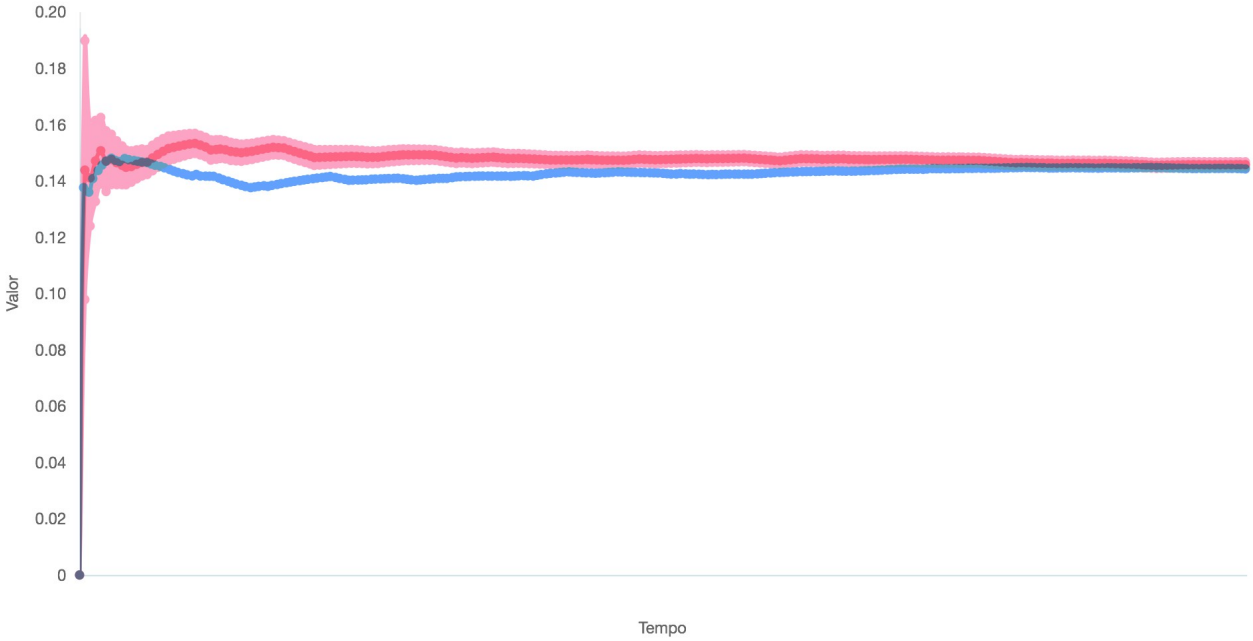
Nessa execução, pacotes de dados tiveram um tempo de sistema (E[T 1]) de 50.514870 ms em média, sendo estes 36.021496 ms de fila e 14.503884 ms de serviço. Todos com intervalos de confiança válidos (com diferença de menos de 10% do valor médio). O mesmo se aplica à média de pacotes de dados na fila (E[Nq1]), com 1.191288.

O mesmo processo usando apenas o teste de corretude do tempo de serviço de pacotes de dados resultou em um período de tempo igualmente rápido, mas seus valores foram 67.155227 ms para E[T 1], 52.447560 ms para E[W 1], 14.718442 ms para E[X 1] e 1.754310 para E[Nq1]. Suspeitou-se que essa divergência poderia estar ligada à geração determinística do número de pacotes de voz por intervalo. Ao executar uma simulação com todos os testes, exceto este, o resultado confirmou a teoria, com 64.559062 ms para E[T1], 49.963334 ms para E[W1], 14.623496 ms para E[X1] e 1.672288 para E[Nq1]; todos muitos próximos dos valores anteriores, e com intervalo de confiança que permite alcançá-las. Para evitar isso, o M/D/1 usado em testes da quantidade de pacotes de voz foi substituído por um M/M/1.

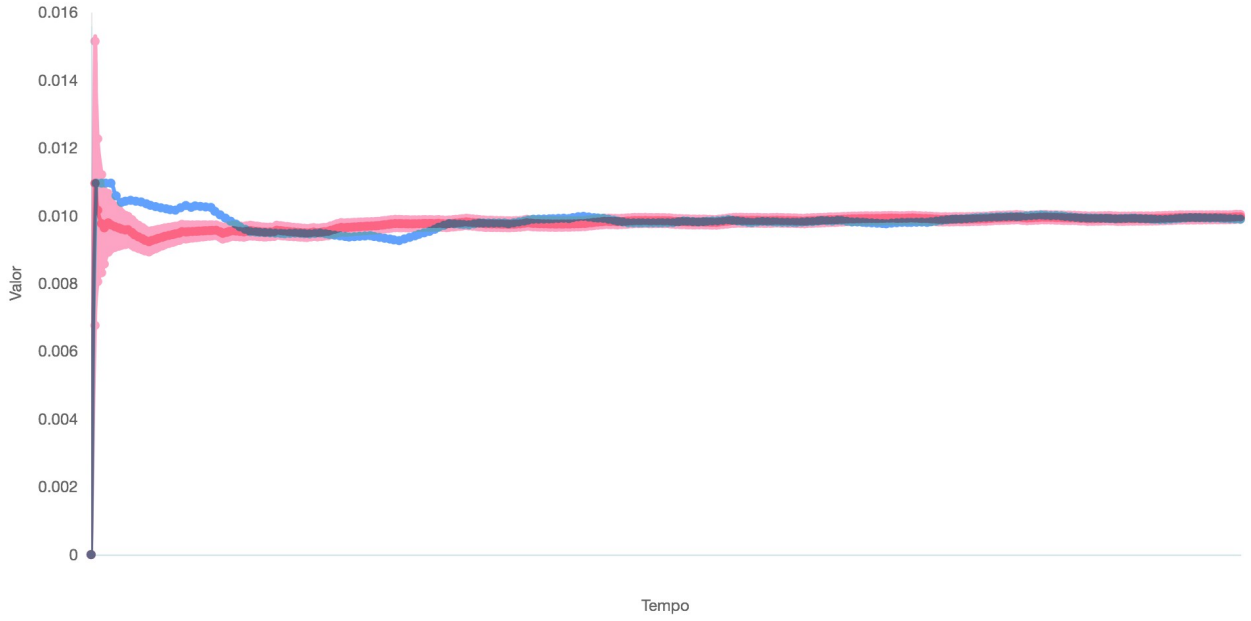
Mesmo assim, quando gráficos foram plotados para avaliar a eficácia do teste de corretude, notou-se que o M/M/1 estava divergindo muito do M/G/1 para este caso. Dois fatores foram dados como suspeitos: a geração de números aleatórios, a qual fazia um exponencial em um caso e uma geométrica na outra, desalinhando a geração aleatória para outros valores, ou o fato de que esse valor é arredondado. O primeiro caso foi testado, e nada mudou. O segundo foi testando arrendondando para cima em vez de para o valor mais próximo. Isso resolveu o problema. O procedimento M/G/1 do Python (`numpy.random.geometric`) gera um número inteiro, enquanto no procedimento M/M/1 (`random.expovariate`) retorna um número de ponto flutuante.



E[W] de pacotes de voz sem interrupções, com teste de quantidade de pacotes de voz

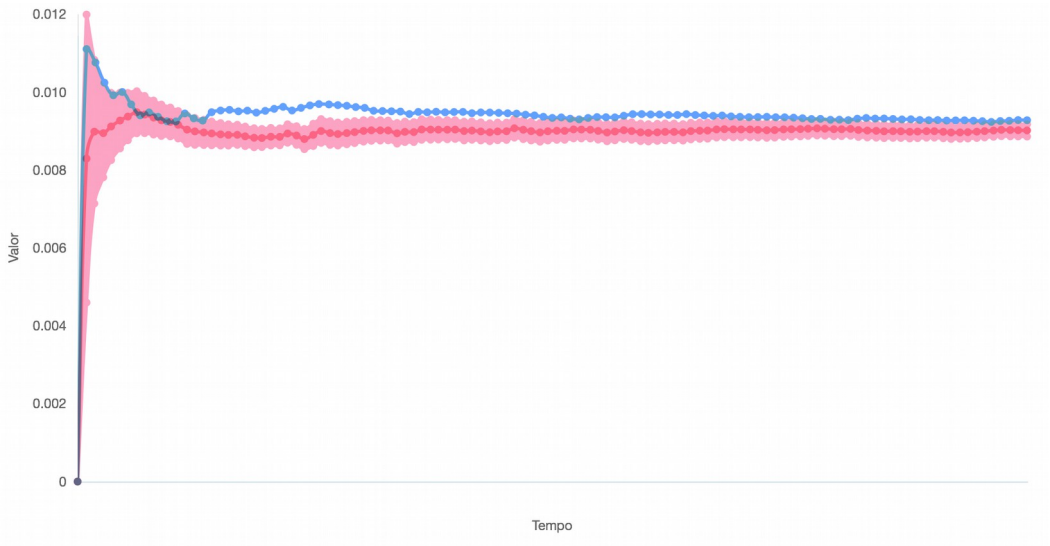


E[Nq] de pacotes de voz sem interrupções, com teste da chegada de pacotes de voz

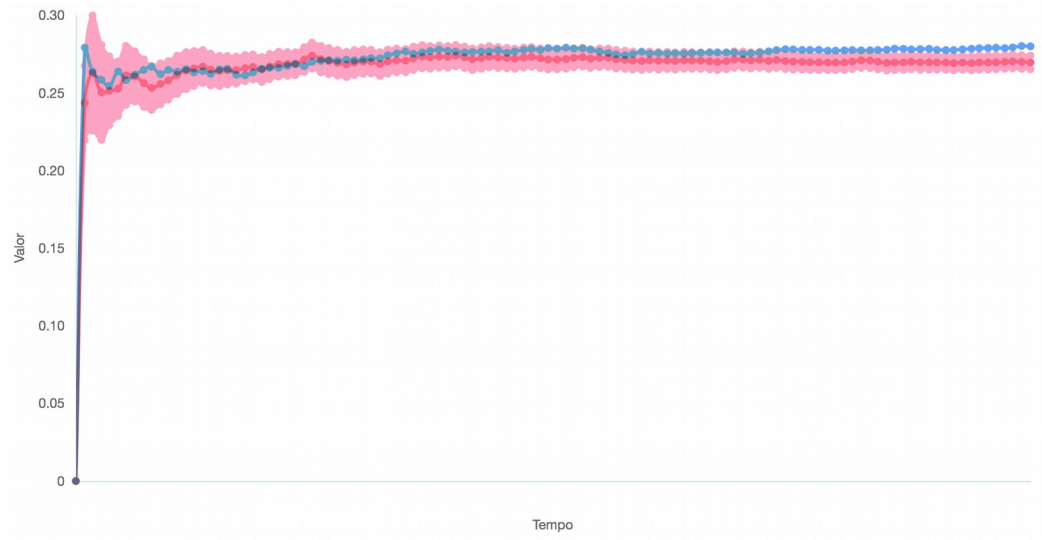


E[Nq] de pacotes de dados sem interrupções, com teste da chegada de pacotes de dados

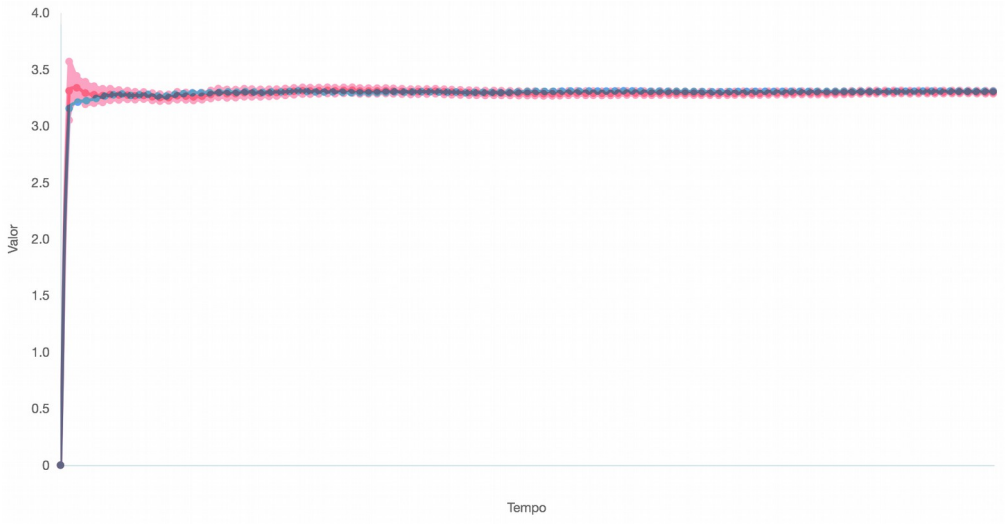
Testes também foram feitos sem pacotes de voz no sistema, para comprovar a eficácia dos testes de corretude. O teste de tempo de chegada de pacotes de dados se mostrou acurado. E[Nq], E[W] e E[T] tiveram todos bons resultados.



E[Nq] de pacotes de dados, com teste da chegada de pacotes de dados (sem pacotes de voz no sistema)

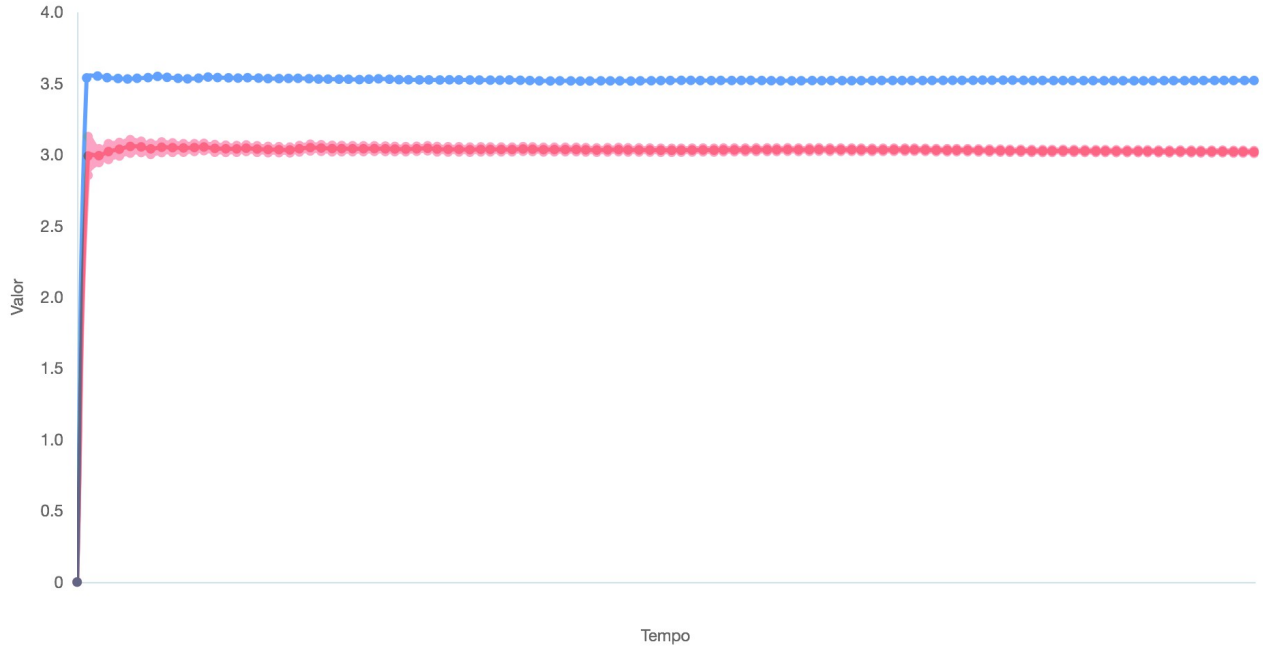


E[W] de pacotes de dados, com teste da chegada de pacotes de dados (sem pacotes de voz no sistema)



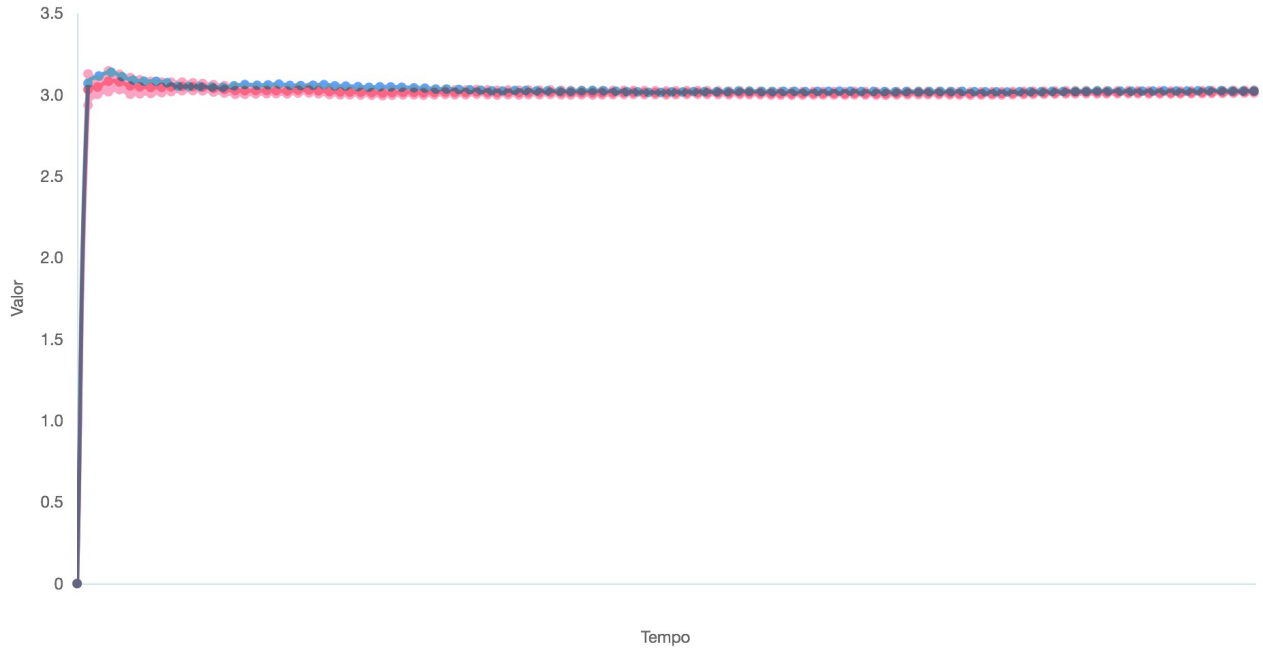
E[T] de pacotes de dados, com teste da chegada de pacotes de dados (sem pacotes de voz no sistema)

No entanto, as mesmas condições não tiveram a mesma acuracia para o teste de tempo de serviço de pacotes de dados, causando um erro de meio milésimo no cálculo de E[X].

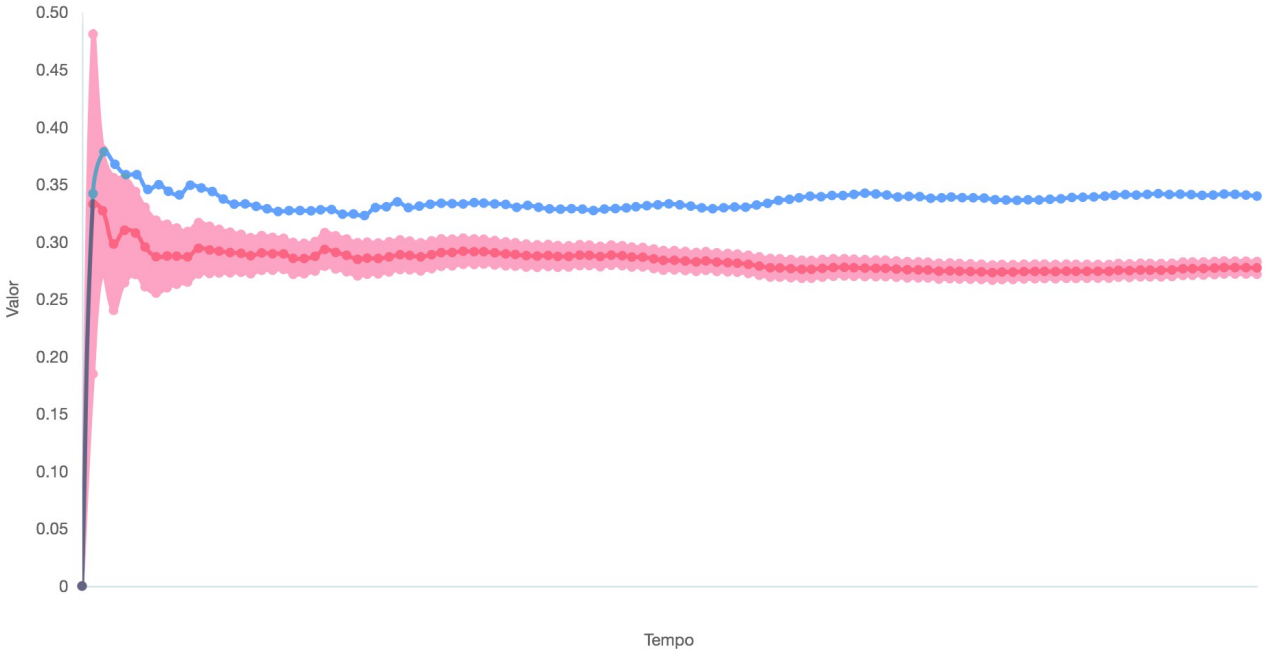


E[X] de pacotes de dados, com teste de serviço de pacotes de dados (sem pacotes de voz no sistema)

É possível atingir um valor médio equivalente para E[X] usando uma distribuição geométrica ao redor de 755, mas essa mesma progressão, por outro lado, não é acurada para o tempo de fila.



E[X] de pacotes de dados, com teste de serviço de pacote de dados com distribuição geométrica (sem pacotes de voz no sistema)



E[W] de pacotes de dados, com teste de serviço de pacote de dados com distribuição geométrica (sem pacotes de voz no sistema)

Os testes de corretude demoraram a serem aplicados, e isso causou alguns problemas durante o desenvolvimento. Na escolha de parâmetros, no subcapítulo 1.4, houveram vários casos dos testes de corretude sendo utilizados como métricas.

Resumo:

- **Chegada de atividade de voz:** M/M/1 se torna M/G/1
- **Chegada de pacote de dados:** M/M/1 se torna M/G/1
- **Quantidade de pacote de voz:** M/G/1 se torna M/M/1
- **Serviço de pacotes de dados:** Distribuição mista se torna sua integral

3. FASE TRANSIENTE

Para determinar quando a fase transiente deve terminar foram observadas várias simulações. Percebeu-se que atingir um padrão não necessariamente significa manter um valor constante; também é possível que haja um crescimento constante.

Portanto, para reconhecer esse momento de crescimento constante, em cada evento é calculado o E[W] de pacotes de voz, para que a cada 1.000 eventos seja calculada e comparada a variância dos últimos 1.000 eventos e dos 1.000 eventos anteriores a esses. Se a diferença entre as duas variâncias for menor que 0.0001 (ou 10^-4), a fase transiente acaba.

Esse valor foi determinado ao realizar a execução (com lambda para p = 0.1), e a cada 1.000 eventos informar a diferença das variâncias. Os resultados foram:

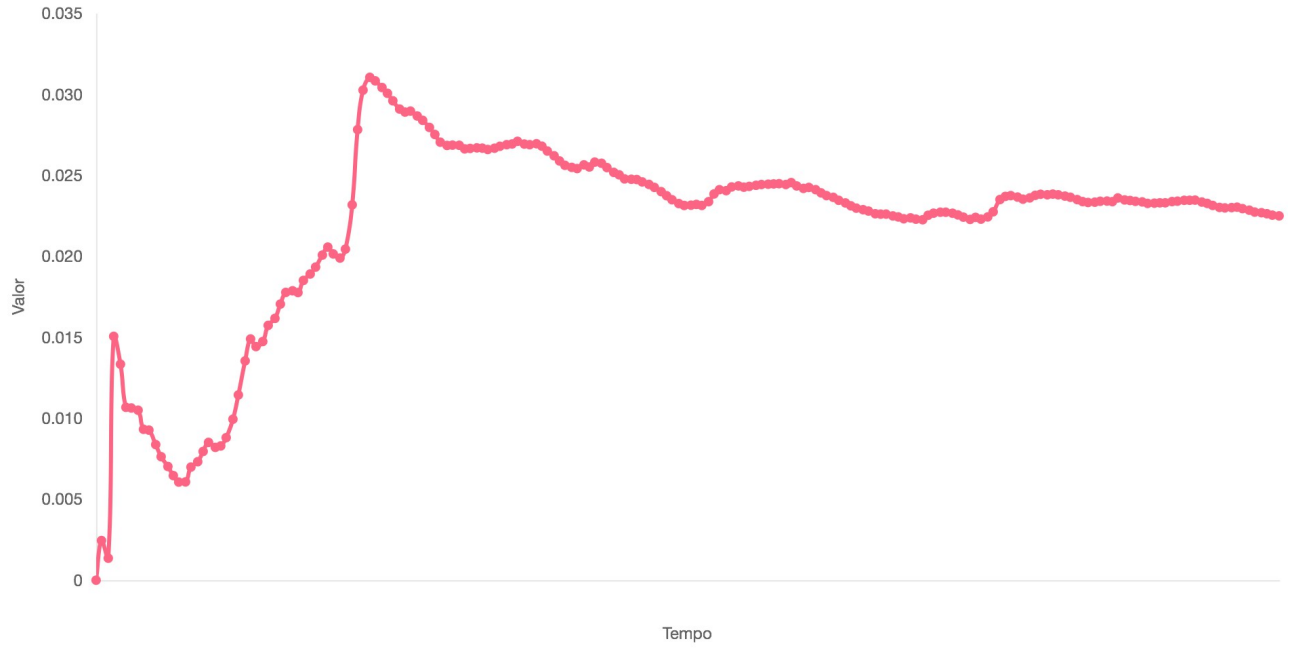
- 0.00337004240804
- 3.98884766549e-06
- 1.60696238357e-06
- 6.82903730393e-07
- 9.01319591112e-08

Indicando que a variância de E[W] decaía conforme a execução avançava, propiciando a detecção de quando os pacotes de voz encontram-se em equilíbrio. Neste primeiro momento, o valor de quebra foi definido arbitrariamente como 0.0000001 (ou 10^-7). No entanto, ao executar sem interrupção, o resultado foi bem diferente:

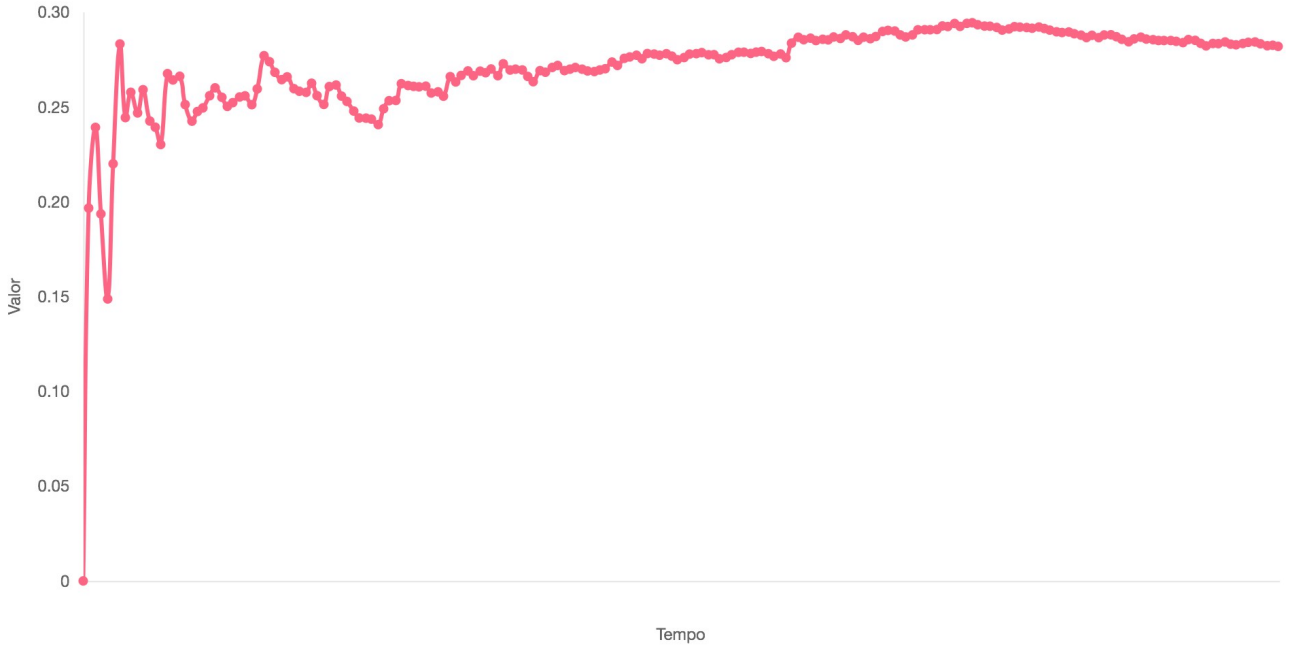
- 0.00117488103793
- 0.00556273913916
- 0.000190403724557
- 6.07831136322e-06
- 1.70131856505e-05
- 1.52041628296e-05
- 2.69563104186e-05
- 9.31703985056e-05
- 1.86793162235e-05
- 1.83764760317e-05

O qual demonstra uma estabilidade abaixo de 0.0001 (10^-4) apenas, e por isso a troca de valores foi realizada. Mesmo executando para lambda de p = 0.7, foi possível finalizar a fase transiente com essa margem:

- 0.272969680647
- 0.00402220652794
- 0.000613103128084
- 0.000629185252734
- 0.000314956800435
- 0.00112658258116
- 0.000179663935408
- 0.000175045284722
- 7.53366395696e-06

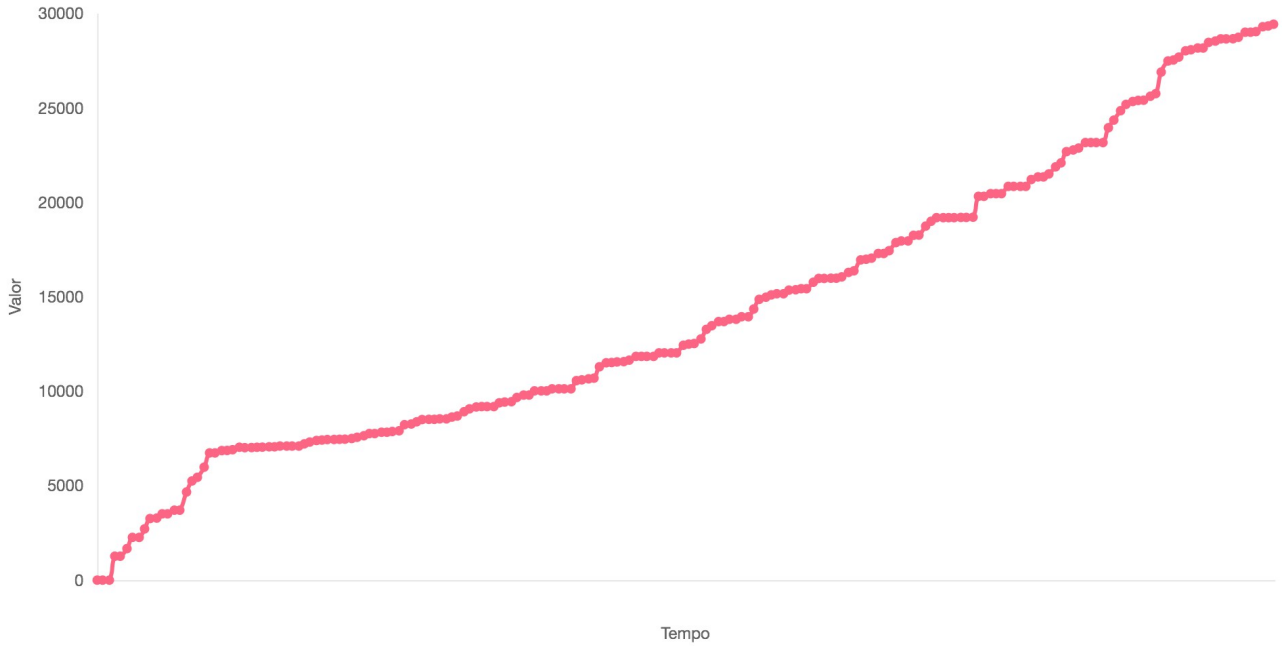


E[W] de pacotes de voz com interrupções após 20.000 serviços de pacotes de voz

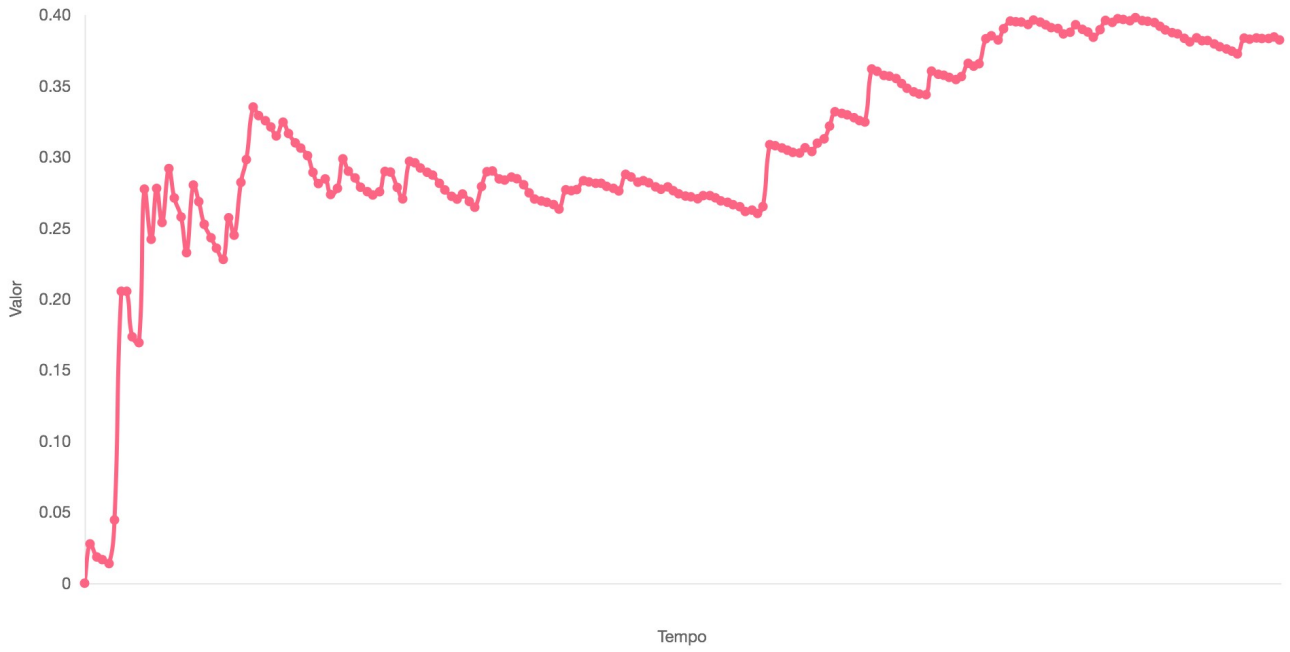


E[W] de pacotes de voz sem interrupções após 1.000 serviços de pacotes de voz e 1.000 serviços de pacotes de dados

Antes disso, tentou-se usar o E[W] de pacotes de dados, mas como em casos com interrupções eles são frequentemente retransmitidos, o tempo médio de fila não possui uma variância de crescimento constante.



E[W] de pacotes de dados com interrupções após 100.000 serviços de pacotes de voz



E[W] de pacotes de dados sem interrupções após 1.000 serviços de pacotes de voz e 1.000 serviços de pacotes de dados

4. OBTENÇÃO DE VALORES ESTATÍSTICOS

Na geração de valores estatísticos, diversos valores, os quais foram requisitados, são calculados. Abaixo, o método usado na obtenção deles.

4.1. E[T]

Quando um pacote entra no sistema, o momento em que ele entrou (em milissegundos) do tempo decorrido da simulação é armazenado no próprio pacote. O mesmo ocorre quando um pacote sai do sistema. Ao subtrair o momento de saída pelo momento de entrada, é possível obter o tempo de sistema do pacote. Ao fazer a média do tempo de sistema de todos os pacotes de voz/dados de uma rodada, temos o E[T] para os pacotes de voz/dados dessa rodada. Ao fazer a média dos E[T] de voz/dados das rodadas, temos o E[T] de voz/dados da simulação.

4.2. E[W]

Quando um pacote entra no sistema, o momento em que ele entrou (em milissegundos) do tempo decorrido da simulação é armazenado no próprio pacote. O mesmo ocorre quando um pacote sai do sistema. Ao subtrair o momento de saída pelo momento de entrada, é possível obter o tempo de sistema do pacote. Ao substrair desse valor o E[X], temos o tempo de fila do pacote, considerando mesmo o retorno a fila causado por interrupções no caso de pacotes de dados. Ao fazer a média do tempo de fila de todos os pacotes de voz/dados de uma rodada, temos o E[W] para os pacotes de voz/dados dessa rodada. Ao fazer a média dos E[W] de voz/dados das rodadas, temos o E[W] de voz/dados da simulação.

4.3. E[X]

O tempo de serviço que um pacote precisa passar é determinado logo que ele começa a ser atendido. No caso de pacotes de dados com interrupção, o tempo de serviço é definido quando o pacote é atendido pela primeira vez. Esse tempo de serviço não muda, mesmo quando ocorrem interrupções. O que pode ocorrer de fato é ele não ser concluído totalmente de uma vez só, devido a essas interrupções, mas esses intervalos causados pelas interrupções também contam para E[X]. Para calcular o E[X] de um pacote, subtraímos o momento em que um pacote de dados foi atendido pela primeira vez do momento em que o mesmo pacote completou seu serviço. Ao fazer a média do E[X] de todos os pacotes de voz/dados de uma rodada, temos o E[X] para os pacotes de voz/dados dessa rodada. Ao fazer a média dos E[X] de voz/dados das rodadas, temos o E[X] de voz/dados da simulação.

4.4. E[Nq]

Entre cada par de eventos não ocorre entrada ou saída do sistema, pois isso ocorre durante eventos; isso significa que nesse intervalo de tempo o número de pacotes em cada fila é constante. Por conclusão, se o número de pessoas na fila por intervalo é multiplicado pelo tempo de intervalo em milissegundos, somando-se todos os intervalos, e então dividindo pelo tempo total da rodada em questão em milissegundos, teremos o número médio de pessoas na fila durante aquela rodada. Isso é aplicado para pacotes de

voz e de dados.

4.5. E[Δ]

Há uma classe na aplicação criada unicamente com o objetivo de lidar com o cálculo de intervalos de confiança, a CalculadoraVoz. Logo que uma rodada acaba, a variância da chegada de seus pacotes de voz é calculada, para cada intervalo de cada canal (Δ_j). É calculada então a média dessas variâncias para cada rodada, conseguindo assim o E[Δ_k] da k-ésima rodada. Com a média de todos os E[Δ_k], conseguimos o E[Δ] da simulação.

4.6. V(Δ)

Com base nos valores calculados no subcapítulo 4.5, podemos calcular a variância das variâncias de chegada para cada rodada, subtraindo E[Δ_k] de cada variância, elevando ao quadrado, e então somando os valores obtidos, divindo-os posteriormente pela quantidade de intervalos menos um. Isso proverá V(Δ_k). Para adquirir V(Δ), basta calcular a média de todos os valores de V(Δ_k).

4.7. Intervalo de Confiança

Há uma classe na aplicação criada unicamente com o objetivo de lidar com o cálculo de intervalos de confiança, a CalculadoraIC. Essa classe possui uma única função chamada externamente, a qual recebe uma lista com as amostras das quais deve ser calculado o intervalo de confiança (exemplo: E[T] para cada rodada da simulação; as quais irei me referir apenas como Ti no resto desse subcapítulo para evitar confusão com o E[T] da simulação em si) e o tamanho dessa mesma lista.

Primeiramente, é calculada a média de Ti, obtendo assim E[T]. O grau de liberdade usado para determinar a distribuição T-Student é igual ao número de amostras menos um. Baseando-se no intervalo de confiança passado como parâmetro para a aplicação, e no grau de liberdade recém-calculado, considerando que valor mínimo de n é 30, é possível obter o percentil da distribuição T-Student, e com ele calcular o desvio padrão com:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(X_i - \hat{\mu})^2}{n-1}$$

Dividimos então o desvio padrão pela raiz do número de amostras, e multiplicamos pelo percentil. Isso nos concede a variação do intervalo de confiança. Subtraindo e somando esse valor de E[T], temos respectivamente o mínimo e o máximo do intervalo de confiança.

5. 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 1.000 para se ter o valor em pacotes/milisegundo, para mostrar os resultados de maneira mais precisa.

5.1. Sem interrupção

p_1	p_1*	E[T1] (ms)			E[W1] (ms)		
.1	0.099564	3.441301	3.455660	3.470019	0.441995	0.449103	0.456212
.2	0.200503	3.999157	4.017556	4.035956	0.976929	0.990270	1.003611
.3	0.299517	4.711408	4.743970	4.776532	1.701753	1.729174	1.756595
.4	0.400145	5.812302	5.871855	5.931407	2.795371	2.851324	2.907277
.5	0.499510	7.425251	7.518508	7.611765	4.413192	4.502903	4.592614
.6	0.601890	10.904535	11.164879	11.425222	7.879947	8.137342	8.394737
.7	0.702702	19.175077	19.958436	20.741796	16.149596	16.929377	17.709159

p_1	E[X1] (ms)			E[Nq1] (ms)		
.1	2.996248	3.006837	3.017427	0.010141	0.010351	0.010561
.2	3.018832	3.027596	3.036360	0.049376	0.050235	0.051094
.3	3.004974	3.015142	3.025309	0.137987	0.140746	0.143505
.4	3.012143	3.021092	3.030040	0.319849	0.327547	0.335245
.5	3.008110	3.017043	3.025976	0.658507	0.674350	0.690192
.6	3.020095	3.029511	3.038928	1.474364	1.529280	1.584195
.7	3.022422	3.031657	3.040893	3.660086	3.855170	4.050253

p_1	E[T2] (ms)			E[W2] (ms)			E[Nq2] (ms)		
.1	0.574700	0.577039	0.579378	0.318732	0.321072	0.323412	0.142464	0.144053	0.145641
.2	0.877771	0.881466	0.885161	0.621848	0.625543	0.629239	0.273437	0.276631	0.279825
.3	1.173022	1.179878	1.186734	0.917126	0.923978	0.930829	0.405577	0.411503	0.417429
.4	1.469098	1.476446	1.483794	1.213206	1.220554	1.227901	0.516641	0.524083	0.531524
.5	1.776843	1.785440	1.794036	1.520947	1.529541	1.538136	0.652502	0.661635	0.670769
.6	2.084855	2.095944	2.107033	1.828972	1.840059	1.851145	0.798926	0.812395	0.825865
.7	2.392277	2.407325	2.422373	2.136310	2.151358	2.166406	0.911335	0.927198	0.943061

p_1	E[Δ] (segundos)			V(Δ) (segundos)			Execução (segundos)
.1	0.019479	0.019666	0.019853	0.001837	0.001913	0.001989	95.875753
.2	0.019084	0.019356	0.019627	0.001753	0.001873	0.001993	57.299375
.3	0.019112	0.019398	0.019683	0.001704	0.001806	0.001907	40.176633
.4	0.017564	0.017864	0.018165	0.001380	0.001455	0.001531	37.305377
.5	0.017218	0.017588	0.017957	0.001339	0.001457	0.001575	34.980879
.6	0.017431	0.017807	0.018183	0.001315	0.001417	0.001519	38.254047
.7	0.017083	0.017489	0.017895	0.001348	0.001454	0.001560	29.506318

5.2. Com interrupção

p_1	p_1*	E[T1] (ms)			E[W1] (ms)		
.1	1.330530	10166.967575	12786.699377	15406.431179	10131.322118	12751.585841	15371.849563
.2	2.670555	X	13164.084402	X	X	13135.301909	X
.3	4.432272	X	16772.169259	X	X	16749.523725	X
.4	6.237103	X	21468.686683	X	X	21448.302947	X
.5	9.078343	X	20609.913234	X	X	20576.451839	X
.6	5.664518	X	25679.282729	X	X	25667.942728	X
.7	14.340374	X	28094.597262	X	X	28045.341736	X

p_1	E[X1] (ms)			E[Nq1] (ms)		
.1	39.841783	40.182013	40.522243	17527.793097	18976.515586	20425.238075
.2	X	40.325381	X	64614.753689	69793.359069	74971.964448
.3	X	44.618207	X	111925.263097	120970.715879	130016.168661
.4	X	47.090124	X	157199.341708	169917.200853	182635.059998
.5	X	54.833189	X	203982.421238	220408.772703	236835.124167
.6	X	28.511407	X	251299.707578	271630.834409	291961.961240
.7	X	61.868473	X	298347.329864	322394.061990	346440.794116

p_1	E[T2] (ms)			E[W2] (ms)			E[Nq2] (ms)		
.1	0.280165	0.280498	0.280831	0.024165	0.024498	0.024831	0.015877	0.016125	0.016373
.2	0.280750	0.281116	0.281482	0.024751	0.025116	0.025482	0.016322	0.016586	0.016850
.3	0.280855	0.281205	0.281555	0.024856	0.025206	0.025556	0.016420	0.016672	0.016925
.4	0.280679	0.281015	0.281352	0.024679	0.025016	0.025352	0.016226	0.016478	0.016729
.5	0.280226	0.280553	0.280879	0.024227	0.024553	0.024879	0.016008	0.016250	0.016492
.6	0.280763	0.281142	0.281522	0.024764	0.025143	0.025522	0.016303	0.016577	0.016852
.7	0.280390	0.280742	0.281094	0.024390	0.024742	0.025094	0.016066	0.016322	0.016579

p_1	E[Δ] (segundos)			V(Δ) (segundos)			Execução (segundos)
.1	0.019666	0.019841	0.020015	0.001884	0.001944	0.002005	97.156844
.2	0.019648	0.019836	0.020025	0.001859	0.001930	0.002001	104.403725
.3	0.019620	0.019821	0.020021	0.001835	0.001924	0.002014	110.246694
.4	0.019308	0.019484	0.019659	0.001754	0.001827	0.001899	119.259438
.5	0.019546	0.019733	0.019920	0.001782	0.001847	0.001912	120.193710
.6	0.019492	0.019652	0.019813	0.001864	0.001934	0.002004	124.360016
.7	0.019521	0.019711	0.019901	0.001855	0.001922	0.001990	130.836377

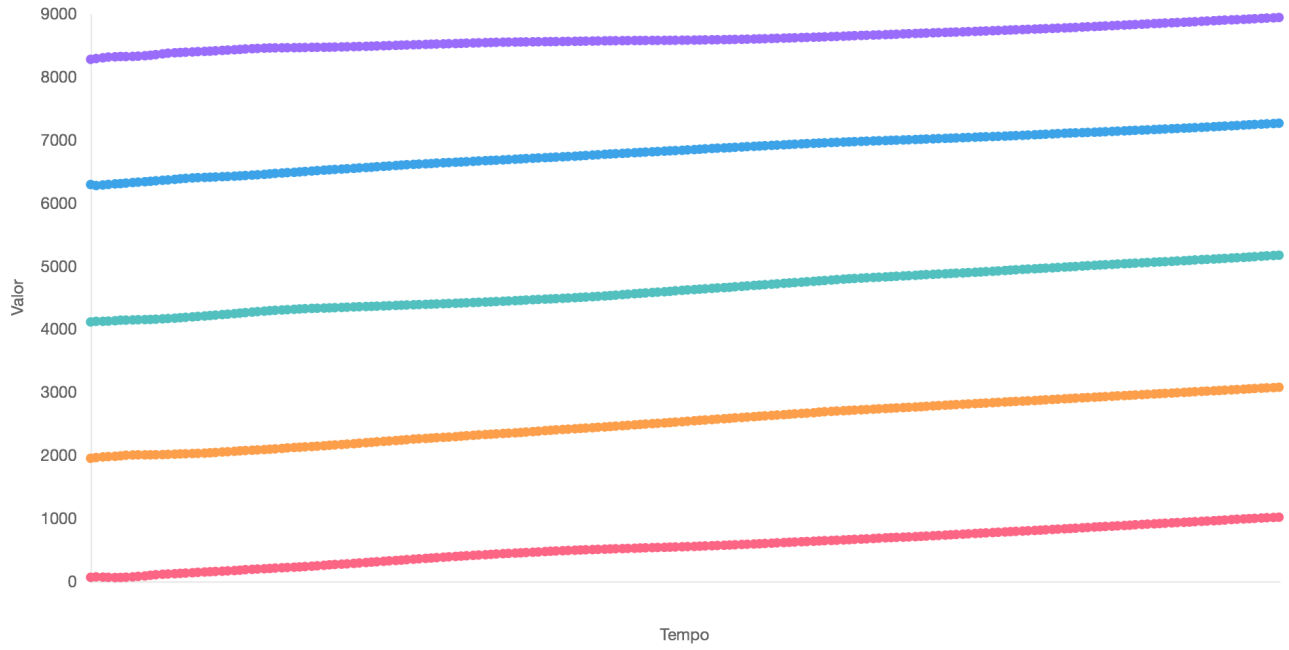
5.3. Sem pacotes de dados

- **E[T2] (ms):** 0.280941 (com IC de 0.280580 a 0.281301)
- **E[W2] (ms):** 0.024941 (com IC de 0.024580 a 0.025301)
- **E[Nq2] (ms):** 0.016489 (com IC de 0.016222 a 0.016756)
- **E[Δ] (segundos):** 0.020005 (com IC de 0.019799 a 0.020211)
- **V(Δ) (segundos):** 0.001985 (com IC de 0.001914 a 0.002057)
- **Execução (segundos):** 78.297746

5.4. Avaliação

Por precisar de um tempo de serviço em média quase 12 vezes maior que os pacotes de voz, e por ter menos prioridade que eles, o aumento na quantidade de pacotes de dados sempre acaba causando um grande aumento no seu tempo de fila, o que acaba refletindo no seu tempo total. No caso de execuções com interrupção, esse tempo grande acaba ocasionando várias interrupções, e isso resulta na instabilidade dos valores relacionados aos pacotes de dados. Essa instabilidade, por sua vez, deixa os intervalos de confiança pouco precisos, quando sequer existem. Os intervalos de confiança marcados com X são aqueles em que apenas uma única amostra pode ser obtida sobre aquele valor em todas as suas rodadas (sempre pertencente à primeira rodada).

Para resolver o problema durante as execuções com interrupção, como mencionado na escolha de parâmetros, foi preciso abrir mão da certeza de qualquer chegada de pacotes de dados. Isso se torna visível na única métrica de pacotes de dados que possui amostras o suficiente para se calcular um intervalo de confiança, por mais que ela também não seja acurada: E[Nq1], a quantidade de pacotes de dados na fila, que tende a estar cada vez maior após cada rodada.



E[Nq] dos pacotes de dados em diferentes rodadas, com 100.000 pacotes de voz por rodada

E[Δ] e V(Δ) se mantém ao redor de uma mesma faixa de valores em testes com interrupção, o que faz sentido, já que a prioridade dos pacotes de voz faz eles agirem da mesma forma, independente de quantos pacotes de dados chegam.

Nos testes sem interrupção, E[Δ] e V(Δ) diminuem conforme p aumenta. Como os pacotes de voz chegam com 16ms de intervalo entre em si em cada canal, e os pacotes de dados levam um certo tempo para serem atendidos, em um sistema sem interrupção os pacotes de voz acabam se enfileirando, fazendo com que cheguem ao serviço de forma mais sequencial.

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. Sem interrupção, é certo que esse tempo seria suficiente, mesmo no pior dos casos encontrados. Agora, considerando a falta de estabilidade em testes com interrupção, é possível afirmar de longe que os pacotes de dados necessitarão de muito mais do que 168ms para passar pela fila e pelo serviço.

Para finalizar, é possível observar que os tempo de pacotes de voz se mantém aproximadamente os mesmos quando não há pacotes de dados, ou quando há, porém com interrupção. Isso é claramente devido ao fato de que a interrupção dá aos pacotes de voz a “impressão” de serem os únicos pacotes do sistema, pois concorrem apenas com eles mesmos.

6. CONCLUSÕES

6.1. 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 apropriada para o mesmo ter ocasionado um pouco de lentidão no processo).

6.2. Otimização da aplicação

Em uma tentativa de otimizar a aplicação, foram feitas algumas mudanças, dentre elas:

- O uso de `xrange` em vez `range`, o qual em um `for` gera o valor a cada loop, e não todos de uma única vez;
- Troca de funções getter's e setter's por acesso direto às propriedades, pois em Python existe lentidão ao acessar funções. Feito na classe evento e em parte da classe pacote.

O resultado com as mudanças acima em testes sem interrupção, para comparar, foi bem animador. Com lambda pra p = 0.1, 1500 atendimentos de cada tipo por rodada em 150 rodadas, o tempo caiu de 345 segundos para 225 segundos. Realizando o mesmo processo descrito por último nas classes de fase e fila, e no resto da classe pacote, a mudança não foi significativa.

Mais otimizações foram então feitas à classe de fase, como remoção de lixo de código e mais remoções de mais getter's. O tempo caiu para 211 segundos, o que não representa uma grande mudança. O uso de generator's também foi considerado, mas houve o problema do mesmo valor ficar se repetindo durante o generator.

Depois de uma mudança na função de adicionarEvento para operações com a métrica de Estatística, o tempo de execução caiu para 208 segundos. Em seguida, a geração do tamanho de pacotes de dados foi otimizada. Anteriormente, havia uma lista com as probabilidades de cada valor possível, a qual era percorrida subtraindo essa probabilidade de um valor aleatório entre 0.0 e 1.0, até que uma probabilidade fosse maior que o número aleatório, indicando que o valor associado àquela probabilidade era o correto.

Isso se mostrou muito ineficaz, e foi substituído pelo método descrito no subcapítulo 1.3.3. Após essa mudança, o tempo de execução caiu para 194 segundos, indicando o máximo que foi possível fazer pela execução do programa.

Fora isso, a construção do conteúdo que é retornado na request do Flask antes era feita na forma de uma string concatenada a cada nova linha. Após uma mudança, passou a haver uma array que recebia as novas linhas, e no fim eram todas concatenadas. Isso reduziu drasticamente o tempo de execução, levando de algo quase interminável para um tempo igual ao da geração de estatísticas.

6.3. Otimização dos parâmetros

Em uma tentativa de otimizar os parâmetros, a aplicação foi executada com diferentes parâmetros para cada um dos valores de p, mas apenas nos casos sem interrupção, já que com interrupção os valores dos pacotes de voz nunca mudam, e os valores dos pacotes de dados crescem indefinidamente, sem nunca entrar em equilíbrio, não tendo sentido otimizar os parâmetros para estes casos.

A métrica que foi definida para dizer que uma simulação atingiu o número ideal de rodadas é o fato da esperança de E[W] dos pacotes de dados mudar em menos de 0.0001 de uma rodada para outra (o que basica considera que, se uma métrica tão instável quanto o tempo de fila entrou em equilíbrio, todas entraram), e foi adicionada uma forma automatizada para determinar isso, enquanto os outros valores foram manuseados manualmente. Para 10 execuções com o método automatizado, o maior número de rodadas atingido é o definido como o mínimo necessário. Com isso, definimos assim valores que possuem qualidade no resultado em menos tempo de execução.

6.3.1. p = 0.1

- **1.000** amostras para fim de fase transiente
- **0.00001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **47 rodadas**
- **Tempo de execução:** 50.923213 segundos

6.3.2. p = 0.2

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **62 rodadas**
- **Tempo de execução:** 38.410992 segundos

6.3.3. p = 0.3

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **73 rodadas**
- **Tempo de execução:** 32.571801 segundos

6.3.4. p = 0.4

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **100 rodadas**
- **Tempo de execução:** 39.833570 segundos

6.3.5. p = 0.5

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **150 rodadas**
- Tempo de execução: 38.001811 segundos

6.3.6. p = 0.6

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **290 rodadas**
- Tempo de execução: 64.602079 segundos

6.3.7. p = 0.7

- **1.000** amostras para fim de fase transiente
- **0.0001** de margem de variância para fase transiente
- Mínimo de **1.200** atendimentos de voz por rodada
- Mínimo de **1.200** atendimentos de dados por rodada
- **410 rodadas**
- Tempo de execução: 80.148778 segundos

Foi analisada a quantidade de pacotes que chegam ao sistema durante cada rodada (e durante a fase transiente), tentando relaciona-las com o tempo que o sistema leva para entrar em equilíbrio. Tomando como exemplo o último caso (p = 0.7), os resultados das simulações variaram muito em relação ao número de rodadas necessárias para atingir o equilíbrio. Apesar disso, esses valores não mostraram possuir nenhuma relação com o número de rodadas necessárias para atingir o equilíbrio.

6.4. Resultados

É possivel afirmar com clareza que o sistema com interrupção jamais será o ideal para esse caso. Os pacotes de dados passam a não ter a certeza de que um dia chegarão, enquanto os pacotes de voz, apesar de afetados, não perdem qualidade de transmissão em um sistema sem interrupção.