UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PEDRO MION BRAGA CORDEIRO

SIMULADOR ONLINE PARA TEORIA DE FILAS

RIO DE JANEIRO
2026

PEDRO MION BRAGA CORDEIRO

SIMULADOR ONLINE PARA TEORIA DE FILAS

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Vinícius Gusmão Pereira de Sá

RIO DE JANEIRO

2026

CIP - Catalogação na Publicação

C794s

Cordeiro, Pedro Mion Braga
Simulador online para teoria de filas / Pedro Mion Braga Cordeiro. -- Rio de Janeiro, 2026.
71 f.

Orientador: Vinícius Gusmão Pereira de Sá.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2026.

1. Simulação. 2. Teoria de filas. 3. Sistema web. 4. Avaliação de desempenho. I. Sá, Vinícius Gusmão Pereira de, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

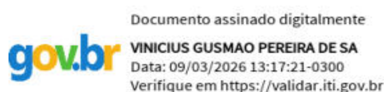
PEDRO MION BRAGA CORDEIRO

SIMULADOR ONLINE PARA TEORIA DE FILAS

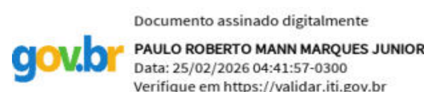
Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 29 de janeiro de 2026

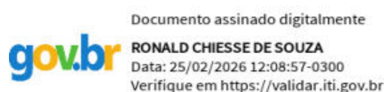
BANCA EXAMINADORA:



Prof. Vinícius Gusmão Pereira de Sá
D.Sc (IC/UFRJ)



Prof. Paulo Roberto Mann Marques Júnior
D.Sc (IC/UFRJ)



Prof. Ronald Chiesse de Souza
D.Sc (IC/UFRJ)

AGRADECIMENTOS

À minha família, especialmente minha mãe, Carla, por sempre me apoiar e por tornar tudo isso possível.

A todos os amigos que me acompanharam ao longo dessa jornada, sendo parte fundamental da minha construção pessoal e profissional.

Por fim, um agradecimento especial a todo o corpo docente com o qual dividi minha trajetória na graduação, enriquecendo meu conhecimento e visão de mundo.

RESUMO

Avaliar a eficiência de um sistema computacional é um desafio desde o surgimento dos computadores. Para encontrar a arquitetura ideal, podem ser utilizadas métricas de desempenho que reflitam o quão preparado o sistema está para lidar com o fluxo de trabalho a ele proposto. Entretanto, existem cenários onde a obtenção de tais métricas a partir de cálculos diretos mostra-se uma tarefa árdua. Pode-se, portanto, contornar essa situação utilizando-se ferramentas computacionais, aliadas à teoria de filas, para simular a execução real do sistema. Dessa forma, a avaliação de seu desempenho via estimativa das métricas necessárias torna-se possível. O presente trabalho cria e avalia o impacto de uma aplicação web que permite a simulação de sistemas computacionais baseada na teoria de filas. A ferramenta proposta possui o potencial de abstrair a complexidade envolvida na execução de simulações computacionais, facilitando sua realização por parte dos usuários.

Palavras-chave: simulação; teoria de filas; sistema web; avaliação de desempenho.

ABSTRACT

The evaluation of the efficiency of a computational system has been a challenge since the advent of computers. To identify the ideal architecture, performance metrics can be used to reflect how well the system is prepared to handle the proposed workload. However, there are scenarios in which obtaining such metrics through direct calculations proves to be a difficult task. This situation can therefore be circumvented by using computational tools, combined with queueing theory, to simulate actual execution of the system. In this way, assessing its performance via estimation of the necessary metrics becomes feasible. The present work develops and discusses the impact of a web application that enables the simulation of computational systems based on queueing theory. The proposed tool has the potential to abstract away the complexity involved in running computational simulations, making their execution easier for users.

Keywords: simulation; queueing theory; web system; performance evaluation.

SUMÁRIO

1 INTRODUÇÃO **8**
1.1 OBJETIVO **8**
1.1.1 Motivação **8**
1.1.2 O que foi feito **10**
1.2 TRABALHOS RELACIONADOS **11**

2 SOBRE A TEORIA DE FILAS **13**
2.1 HISTÓRIA DA TEORIA DE FILAS **13**
2.2 CONCEITOS FUNDAMENTAIS **13**
2.2.1 Noções Básicas **14**
2.2.2 Notação de Kendall **15**
2.2.3 Parâmetros da Simulação **16**
2.2.4 Métricas **18**
2.3 ANÁLISE TEÓRICA **19**
2.3.1 Lei de Little **19**
2.3.2 Lei do Fluxo Forçado (Forced Flow Law) **20**
2.3.3 Lei do Gargalo (Bottleneck Law) **20**
2.3.4 Limite assintótico para sistemas fechados **21**

3 ESTUDOS DE CASO **22**
3.1 SISTEMA M/M/1 . . . . **22**
3.2 UM CASO MAIS COMPLEXO **23**

4 SOBRE A BIBLIOTECA DE SIMULAÇÃO **27**
4.1 ARQUITETURA **27**
4.2 CLASSES **28**
4.2.1 Environment (Ambiente) **28**
4.2.2 Network (Rede) **29**
4.2.3 Server e Job (Servidor e Tarefa) **30**
4.2.4 Metrics (Métricas) **30**
4.2.5 SimulationResults (Resultados da Simulação) **31**
4.2.6 Execution (Execução) **32**
4.3 IMPLEMENTAÇÃO **33**
4.3.1 Configuração do Ambiente **33**
4.3.2 Execução **39**
4.3.3 Testes **45**

4.3.4 Conclusão . . . 47

5 SOBRE A APLICAÇÃO WEB . . . 48
5.1 APLICAÇÃO WEB . . . **48**
5.1.1 Interface gráfica . . . 48
5.1.2 Configuração do ambiente . . . 48
5.1.3 Fluxo de simulação . . . 51
5.2 API . . . **52**
5.3 CONCLUSÃO . . . **53**

6 ANÁLISE DOS RESULTADOS . . . 54
6.1 FILA M/M/1 . . . **54**
6.1.1 Simulação utilizando a biblioteca . . . 55
6.1.2 Simulação utilizando a interface gráfica . . . 55
6.2 SISTEMA FECHADO . . . **57**
6.2.1 Simulação utilizando a biblioteca . . . 57
6.2.1.1 Ambiente . . . 58
6.2.1.2 Servidor A . . . 59
6.2.1.3 Servidor B . . . 59
6.2.1.4 Servidores C . . . 61
6.2.1.5 Servidor D . . . 61
6.2.2 Simulação utilizando a interface gráfica . . . 62
6.2.2.1 Ambiente . . . 62
6.2.2.2 Servidor A . . . 62
6.2.2.3 Servidor B . . . 63
6.2.2.4 Servidores C . . . 63
6.2.2.5 Servidor D . . . 64
6.3 DEMAIS DISTRIBUIÇÕES . . . **64**
6.3.1 Constante . . . 64
6.3.2 Uniforme . . . 65
6.3.3 Normal . . . 66
6.4 COMPARAÇÃO COM SIMPY . . . **67**
6.5 CONCLUSÃO . . . **67**

7 CONCLUSÃO . . . 69
7.1 APRENDIZADO COM O TRABALHO . . . **69**
7.2 CONSIDERAÇÕES PARA O FUTURO . . . **69**

REFERÊNCIAS . . . 71

1 INTRODUÇÃO

Desde o surgimento da computação, profissionais da área trabalham em formas de avaliar e otimizar o desempenho dos programas que constroem. Com a popularização dos computadores na década de 80 e o surgimento da Internet, sistemas computacionais com múltiplos servidores interconectados passaram a ser comuns. Por conta disso, os projetistas responsáveis pela arquitetura de tais sistemas necessitam de maneiras de avaliar a performance de suas criações, visando tanto otimizar métricas de performance quanto minimizar gastos. Para alcançar este objetivo, diversas técnicas são aplicadas a fim de obter métricas que possibilitem avaliar o desempenho de sistemas computacionais.

O presente estudo tem como foco uma dessas técnicas, a teoria de filas, a qual emprega modelos baseados em filas e servidores para a compreensão de ambientes interconectados. Embora existam diversas ferramentas teóricas para a análise desses sistemas, muitos cenários mais complexos apresentam dificuldades significativas quando submetidos a uma abordagem analítica. Nesses casos, entretanto, torna-se viável simular o comportamento do sistema, possibilitando a coleta experimental das métricas de interesse. Dessa forma, arquitetos de sistemas podem avaliar diferentes cenários e configurações de interação entre serviços, servidores e terminais, com o objetivo de otimizar o funcionamento do produto a ser desenvolvido. Assim, o foco deste trabalho concentra-se na simulação de sistemas computacionais sob a perspectiva da teoria de filas.

1.1 OBJETIVO

1.1.1 Motivação

Para que possamos utilizar todo o potencial da teoria de filas no planejamento e execução de sistemas computacionais, precisamos de ferramentas simples e intuitivas, mas que, ao mesmo tempo, nos permitam atingir os resultados esperados e coletar métricas fundamentais para a tomada de decisões. Muitas linguagens oferecem bibliotecas para simulações de eventos discretos, como SimPy[1], em Python, e JaProSim[2], em Java. Entretanto, o ambiente precisa ser manualmente configurado para que as simulações computacionais sejam realizadas com precisão, o que nem sempre se prova um processo simples. Além disso, também é necessário planejar como coletar as métricas necessárias daquela simulação, o que pode ser um impeditivo para usuários sem grande domínio técnico.

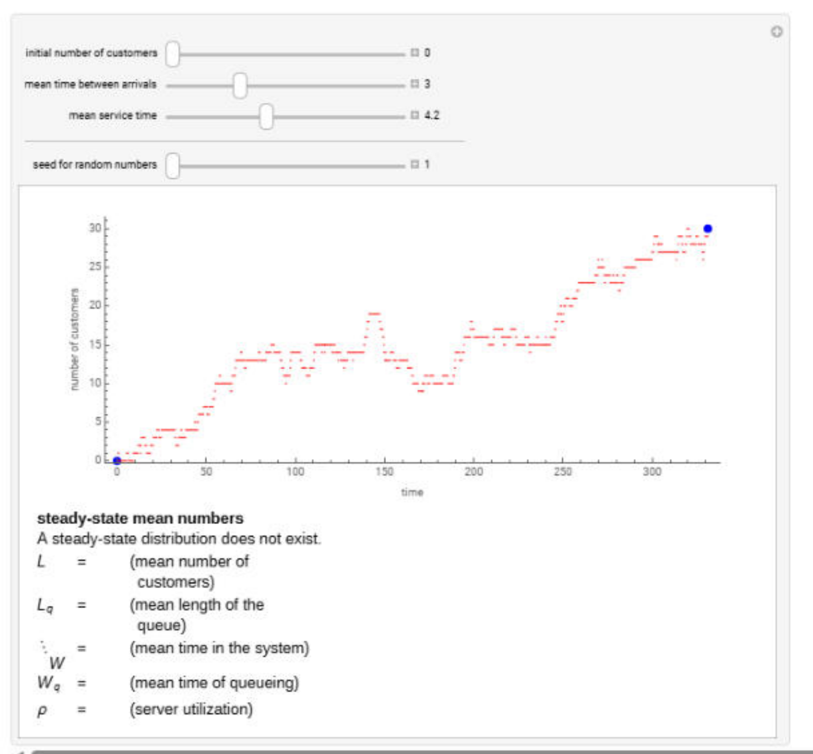
Além das bibliotecas em código, há também aplicações web que oferecem o serviço de simulação de filas computacionais. Entretanto, tais ferramentas não oferecem muitas opções de personalização, restringindo de forma significativa os sistemas passíveis de serem

[1] https://simpy.readthedocs.io/en/latest/
[2] https://sourceforge.net/projects/japrosim/

simulados. Ademais, a maioria não possui uma interface gráfica intuitiva e amigável, dificultando e desmotivando sua utilização.

Figura 1 – Exemplo de ferramenta externa

Simulating the M/M/1 Queue



Fonte: Wolfram (https://demonstrations.wolfram.com/SimulatingTheMM1Queue/)

Dessa forma, a motivação para o desenvolvimento do trabalho surgiu da ausência de ferramentas simples, modulares e já configuradas disponíveis em Python para a simulação de sistemas de filas. Essa característica torna o uso dessas ferramentas menos acessível para fins didáticos ou para análises rápidas, nas quais se deseja obter métricas sem a necessidade de extensas configurações manuais. Diante disso, este trabalho é uma contribuição para o problema de simulações de redes de filas. Uma solução que, em relação às ferramentas existentes, é mais simples e intuitiva foi desenvolvida, auxiliando sobretudo usuários menos familiarizados com os conceitos de redes de filas. Dessa forma, uma interface clara e de fácil utilização é fornecida, ideal para estimular o aprendizado e possibilitar testes simples e rápidos.

Além disso, a interface gráfica desenvolvida desempenha um papel fundamental na democratização do acesso ao recurso, ao viabilizar a realização de simulações de forma intuitiva, ágil e didática por meio da internet. Dessa maneira, os usuários podem explorar diferentes formas de construir, simular e interagir com ambientes baseados em sistemas de filas.

1.1.2 O que foi feito

O primeiro passo consistiu no desenvolvimento de uma biblioteca robusta, contendo diversos parâmetros de personalização, que seja capaz de realizar as simulações por longos períodos de tempo e fornecer as métricas desejadas de forma clara e organizada. A linguagem escolhida foi o Python, visando maximizar a quantidade de potenciais usuários, visto que tal linguagem é conhecida por ser simples e amigável para iniciantes na computação. Através dessa biblioteca, usuários são capazes de definir sistemas de filas e executá-los conforme desejarem, de acordo com alguns parâmetros, obtendo diversas métricas úteis para avaliar o desempenho da arquitetura proposta. A escolha de desenvolver a lógica da simulação como uma biblioteca se dá pela vontade de permitir que usuários com conhecimentos computacionais sejam capazes de interagir diretamente com o código para atender suas necessidades, sem a obrigatoriedade de utilização apenas através da aplicação web, se assim o desejarem.

A segunda etapa consistiu na construção da interface gráfica com a qual o usuário final pode interagir. Através desta interface, o usuário é capaz de montar o sistema que deseja simular, definir os parâmetros que deseja e, enfim, enviar para execução. Após o envio, a aplicação web realiza a simulação com os parâmetros definidos utilizando a biblioteca, através de uma Application Programming Interface (API) desenvolvida com auxílio do framework *FastAPI*[3]. Ao final, o resultado é exposto, possibilitando a coleta das métricas necessárias para análise por parte do demandante.

Diante do cenário apresentado, o capítulo 2 discorre acerca do contexto e dos detalhes técnicos da teoria de filas. Tal base técnica será fundamental para o completo entendimento das decisões tomadas ao longo do estudo, assim como para a compreensão dos parâmetros de entrada e das métricas de saída presentes no software. Desta forma, qualquer usuário poderá usufruir das vantagens oferecidas pelo trabalho.

No capítulo 3, a teoria é posta em prática a partir de dois estudos de caso. No primeiro, um sistema de filas simples é analisado matematicamente, mostrando como a análise das métricas funciona. O segundo exemplo, com maior complexidade, coloca à prova todos os conceitos analisados, mostrando o potencial do sistema computacional criado.

Os capítulos 4 e 5, por sua vez, abordam o software. Assim, as decisões tomadas ao longo do desenvolvimento e os detalhes e nuances de tudo que foi feito são discutidos.

Exemplos de uso são apresentados no capítulo 6, demonstrando como a ferramenta pode ser utilizada na prática. Para atingir este objetivo, os sistemas de filas retratados no estudo de caso são inseridos no sistema e os resultados coletados são comparados com os resultados teóricos, calculados no capítulo 3.

Por fim, o capítulo 7 conclui o trabalho, trazendo os aprendizados obtidos, as dificuldades encontradas e ideias para melhorias no sistema, abrindo margem para potenciais

[3] https://fastapi.tiangolo.com/

trabalhos futuros.

1.2 TRABALHOS RELACIONADOS

Em (LIN; HUANG, 2022), é proposto o uso de simulações baseadas em teoria de filas para modelar e avaliar a confiabilidade de softwares, focando em processos de detecção e correção de falhas. Neste trabalho, é possível ver na prática a aplicação de simulações baseadas em teoria de filas para modelar sistemas complexos de diversas formas diferentes.

Pode-se também observar a dificuldade de modelar por conta própria processos de alto grau de complexidade. Neste contexto, uma aplicação web preparada para realizar tais processos e intuitiva de se utilizar pode se tornar um poderoso aliado na abstração de complexidade, economizando tempo para aqueles que desejam realizar determinados tipos de simulações.

Outro trabalho relevante é (ZINOVIEV, 2024), que mostra como utilizar a biblioteca de simulações discretas SimPy com teoria de filas. O trabalho propõe formas de organizar e executar simulações, assim como demonstra maneiras de construí-las por conta própria.

Apesar deste artigo apresentar a implementação manual de determinadas funcionalidades, conhecer uma biblioteca grande e consolidada como o SimPy é de suma importância para otimizar a forma como o software construído realiza simulações discretas. Tal conhecimento pode ser adquirido através da leitura do artigo, que mostra a partir de exemplos práticos e bem explicados como realizar o que será feito manualmente neste trabalho, com a biblioteca.

Em (ADAN; RESING, 2002), os autores descrevem as ideias por trás da teoria de filas em detalhes, provendo um excelente conhecimento para implementações práticas. Além disso, muitas informações complementares que não foram abordadas neste trabalho podem ser encontradas neste artigo.

A partir deste artigo, também podemos extrair possíveis melhorias no software para trabalhos futuros. Algumas ideias incluem implementar diferentes tipos de processos de chegada e saída, filas finitas e diferentes tipos de prioridades nas filas.

(MARIA, 1997) oferece noções valiosas de simulações de modelagem e simulações que foram aplicadas neste trabalho para otimizar a forma como o software realiza estas funções.

Por fim, (KAMALI et al., 2009) demonstra a aplicação da teoria de filas e de simulação na análise de tráfego de redes computacionais, destacando sua importância na avaliação de sistemas heterogêneos. Tal abordagem evidencia o potencial das simulações como ferramenta de apoio à análise de sistemas computacionais, reforçando a relevância do presente trabalho ao abstrair detalhes de modelagem e potencializar o aprendizado da disciplina de teoria de filas.

Embora existam produções que exploram temáticas de certa forma semelhantes, são extremamente raros os trabalhos que apresentam forte correlação com a proposta desenvolvida nesta pesquisa. Nesse sentido, evidencia-se o caráter original e autêntico do presente trabalho.

2 SOBRE A TEORIA DE FILAS

Este capítulo é destinado à explicação de conceitos da teoria de filas fundamentais para a completa compreensão deste trabalho.

2.1 HISTÓRIA DA TEORIA DE FILAS

Conforme descrito em (SZTRIK, 2010), a teoria de filas teve origem nos trabalhos de Agner Krarup Erlang, no início do século XX, voltados para a busca de soluções eficientes na construção de redes de telefonia. Em um de seus estudos, Erlang propôs a modelagem das chegadas de chamadas telefônicas por meio de um processo de Poisson. A modelagem proposta é válida quando o número de usuários é suficientemente grande. Vale ressaltar que, nesse período, os pesquisadores tinham como principal objetivo mitigar o congestionamento das redes telefônicas, recorrendo a modelos matemáticos estocásticos. Esses modelos não apenas se mostraram adequados para prever e testar alternativas ao problema, como também evoluíram em robustez e flexibilidade, fornecendo instrumentos aplicáveis em diferentes áreas e contextos.

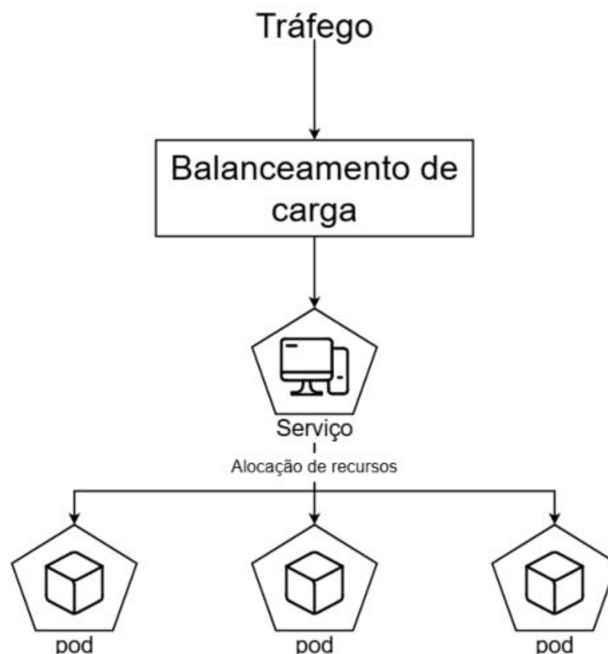
A partir da década de 1960, a teoria de filas passou a despertar interesse na ciência da computação, especialmente com vistas ao aperfeiçoamento de sistemas computacionais integrados. Os modelos e técnicas então disponíveis constituíram ferramentas relevantes para possibilitar a integração de diferentes computadores em uma mesma rede, com troca de informações em tempo real. Desde então, a conexão entre teoria de filas e ciência da computação ampliou-se progressivamente, consolidando a importância da modelagem e simulação de cenários computacionais variados e de alta complexidade para pesquisadores e profissionais da área.

Atualmente, diversos processos do mundo real são modelados por meio da teoria de filas, extrapolando o contexto tradicional das redes de telefonia, com o objetivo de aprimorar a eficiência e a comunicação entre recursos. Ferramentas de computação em nuvem, por exemplo, utilizam esses modelos para gerenciar a entrada de carga em cada instância da aplicação, bem como para otimizar a alocação de recursos e a quantidade de instâncias em operação. Outros exemplos incluem filas virtuais em plataformas de comércio eletrônico, processos de embarque em aeroportos, redes interconectadas de rodovias, entre diversos outros cenários.

2.2 CONCEITOS FUNDAMENTAIS

Nesta seção, serão discutidos os conceitos necessários para realizar consistentemente a modelagem de um sistema computacional em teoria de filas.

Figura 2 – Teoria de Filas em Aplicações em Nuvem



2.2.1 Noções Básicas

Para facilitar a compreensão, pode-se utilizar o exemplo de caixas em um supermercado. Cada caixa possui uma fila independente e atende apenas um cliente por vez. Há também uma fila prioritária, destinada a clientes com determinadas características, que são atendidos antes dos demais. Os operadores de caixa podem apresentar diferentes velocidades de processamento, o que gera variação no desempenho. Cada operador, portanto, possui um tempo médio para atender um cliente. Parte dos clientes se desloca ao mercado de automóvel e, ao finalizar a compra, precisa passar pela fila do estacionamento, enquanto os demais, que foram a pé, deixam o local imediatamente após o pagamento. Embora simplificado, esse exemplo ilustra de forma clara diversos conceitos que serão explorados ao longo deste trabalho. Nesse contexto, os clientes são denominados jobs e os caixas, servidores. Além disso, o tempo de atendimento de cada cliente é denominado tempo de serviço.

No caso de um supermercado, os clientes podem chegar ao caixa a qualquer instante, e não há limite para a quantidade de clientes no sistema. Ademais, não é necessário que um atendimento seja concluído para que novos clientes cheguem e aguardem na fila. Sistemas de filas que apresentam essas propriedades são classificados como abertos. Nesses sistemas, os pontos de entrada de novos jobs são definidos explicitamente no modelo, juntamente com a distribuição probabilística associada e os parâmetros necessários para a geração de amostras durante a simulação.

Por outro lado, quando o ambiente não admite tarefas externas, o sistema é considerado fechado. Nesta configuração, há duas possibilidades: com e sem interação do usuário.

Em sistemas interativos, um terminal com usuários é responsável pelo envio das tarefas. Assim, cada nova tarefa só pode ser produzida quando a tarefa anterior disparada por aquele usuário já foi finalizada. Além disso, tal configuração apresenta uma característica particular: o tempo de pensamento (representado pela variável aleatória Z), que corresponde ao intervalo entre a finalização de um job associado a um usuário e o envio da próxima tarefa, definido pela escolha daquele usuário.

Por sua vez, no modo sem interação, também denominado *batch*, as tarefas são submetidas ao sistema de forma contínua, sem dependência de ações humanas diretas. Nesse cenário, a carga de trabalho é determinada por um conjunto fixo de jobs que circulam internamente, ensejando a inserção de novos jobs na rede de filas tão logo completem sua execução. Não há conceito de tempo de pensamento, pois o sistema opera de maneira autônoma, sem pausas introduzidas por usuários. Esse modo é importante na modelagem teórica de sistemas fechados.

2.2.2 Notação de Kendall

Na maioria dos casos, seis parâmetros são suficientes para descrever um sistema de filas, conforme uma versão estendida da notação de Kendall, definida em (KENDALL, 1953). São eles: (1) distribuição dos intervalos de chegada, (2) distribuição dos tempos de serviço, (3) número de servidores, (4) capacidade do sistema, (5) tamanho da população de clientes e (6) disciplina de atendimento. Por meio dessas seis características, é possível descrever de forma sucinta os principais aspectos do modelo de filas que se deseja representar. Contudo, é comum que apenas os três primeiros parâmetros sejam utilizados, especialmente em situações em que a modelagem reflete um sistema simples.

O primeiro parâmetro, a distribuição dos intervalos de chegada, define qual distribuição estatística será empregada para gerar amostras dos tempos entre chegadas de jobs aos pontos de entrada do sistema. Em sistemas fechados, esse parâmetro se refere ao chamado tempo de pensamento dos usuários, ou seja, o intervalo entre a finalização de uma tarefa e a submissão de uma nova. O segundo parâmetro é análogo, mas aplicado ao tempo de serviço, especificando a distribuição adotada para representar o tempo necessário ao processamento de cada job pelos servidores.

Na maioria dos casos práticos, a escolha mais adequada para descrever o processo de chegada é o processo de Poisson, no qual os intervalos entre chegadas seguem a distribuição exponencial. Na notação de Kendall, essa situação é representada pela letra M, de processo markoviano. Para gerar amostras dessa distribuição, basta conhecer a taxa média, λ, do processo de Poisson associado. Outras distribuições possíveis incluem: a constante, em que todas as amostras possuem o mesmo valor; a uniforme, em que todos os valores de um intervalo têm igual probabilidade de ocorrência; e a normal, utilizada em situações específicas em que a variabilidade é aproximadamente simétrica. Em contextos de simulação, distribuições como a Erlang e a hipergeométrica também podem ser empre-

gadas, oferecendo maior flexibilidade na modelagem de sistemas com diferentes padrões de variabilidade.

O terceiro parâmetro corresponde ao número de servidores do sistema e deve ser explicitado no modelo. Os demais parâmetros, quando omitidos, admitem valores padrão, já que são modificados com menor frequência. O valor padrão para a capacidade do sistema é considerado infinito. Entretanto, se especificado, qualquer job excedente à capacidade máxima será descartado. O parâmetro de capacidade populacional representa o tamanho total da população da qual se originam os jobs e, na ausência de especificação, também é tratado como infinito. Por fim, a disciplina de atendimento determina o critério adotado pelos servidores na escolha do próximo job a ser atendido. O valor padrão é o First Come, First Served (FCFS), em que os jobs são atendidos na ordem de chegada. Entre outras possibilidades, destacam-se o Last Come, First Served (LCFS), no qual o último a chegar é o primeiro a ser atendido, e o Shortest Remaining Time (SRT), que prioriza jobs de menor duração.

Com essa noção, já é possível definir formalmente um sistema de filas. Detalhes adicionais são necessários para viabilizar a execução de simulações, mas, mesmo neste estágio, compreende-se a maior parte das características fundamentais. O exemplo mais simples é o sistema $M/M/1$, constituído por um único servidor, em que tanto as chegadas quanto os tempos de serviço seguem processos markovianos, ou seja, os tempos entre chegadas e os tempos de serviço são variáveis aleatórias exponenciais. Outro exemplo é o $M/D/5$, que descreve um sistema com cinco servidores, chegadas de Poisson e tempo de serviço constante.

Assim, com as noções básicas de teoria de filas formalizadas e sua representação pela notação de Kendall, torna-se possível avançar na análise detalhada dos parâmetros envolvidos na configuração dos ambientes de simulação.

2.2.3 Parâmetros da Simulação

Diversos parâmetros são necessários para a realização de uma simulação completa de um sistema projetado. Esses parâmetros são definidos individualmente para cada componente do sistema, ou seja, cada servidor, job e ponto de entrada possui valores específicos, quando aplicável. Os principais são listados a seguir:

- **Taxa média de chegadas (λ):** Quantidade média de jobs que chegam por segundo em determinado ponto de entrada de um sistema aberto. Exemplo: $\lambda = 2$ jobs/segundo.

- **Tempo médio entre chegadas ($\frac{1}{\lambda}$):** Inverso da taxa média de chegadas, representa o intervalo médio entre duas chegadas sucessivas. Exemplo: $\frac{1}{\lambda} = 0.5$ segundos/job.

- **Tamanho do job (S):** Variável aleatória que representa o tempo necessário para a execução de um job específico em um servidor. Exemplo: $S_i = 2.5$ segundos.

- **Tempo médio de serviço ($E[S]$):** Valor esperado do tamanho de um job, representando o tempo médio de processamento em um servidor. Exemplo: $E[S] = 2$ segundos/job.

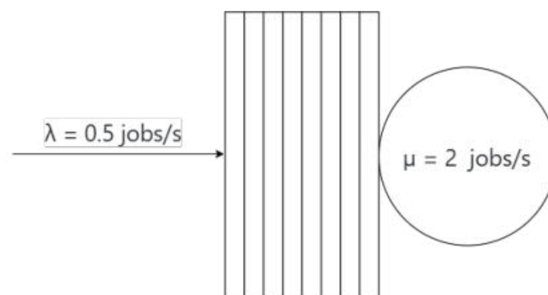
- **Taxa média de serviço ($\mu = \frac{1}{E[S]}$):** Quantidade média de jobs que um servidor é capaz de processar por segundo. Exemplo: $\mu = 0.5$ jobs/segundo.

- **Tempo de pensamento (Z):** Variável aleatória que representa o intervalo de espera entre o término de um job e o envio de uma nova requisição em sistemas fechados. Exemplo: $Z_i = 2$ segundos.

- **Tempo médio de pensamento ($E[Z]$):** Valor esperado do tempo de pensamento, ou seja, o intervalo médio entre a finalização de um job e o envio de uma nova tarefa por parte daquele mesmo usuário de um sistema fechado interativo. Exemplo: $E[Z] = 2$ segundos.

Uma vez definidos os principais parâmetros, é possível compreender exemplos concretos e completos. A figura 3 representa um servidor e sua fila no formato mais simples possível. O parâmetro λ indica que há um ponto de entrada de jobs neste servidor, no qual chegam, em média, 0.5 jobs por segundo. A taxa de serviço do servidor, representada por μ, corresponde à sua capacidade de processamento, que neste exemplo é de 2 jobs por segundo. É fácil, portanto, perceber o comportamento do sistema. Tarefas chegam, são executadas e deixam o sistema, sem acúmulo indeterminado na fila, visto que o servidor possui capacidade de processamento superior à taxa de chegada. Sistemas abertos com essa característica são chamados de estáveis, em oposição a sistemas não estáveis, caracterizados por filas que crescem indefinidamente com a passagem do tempo.

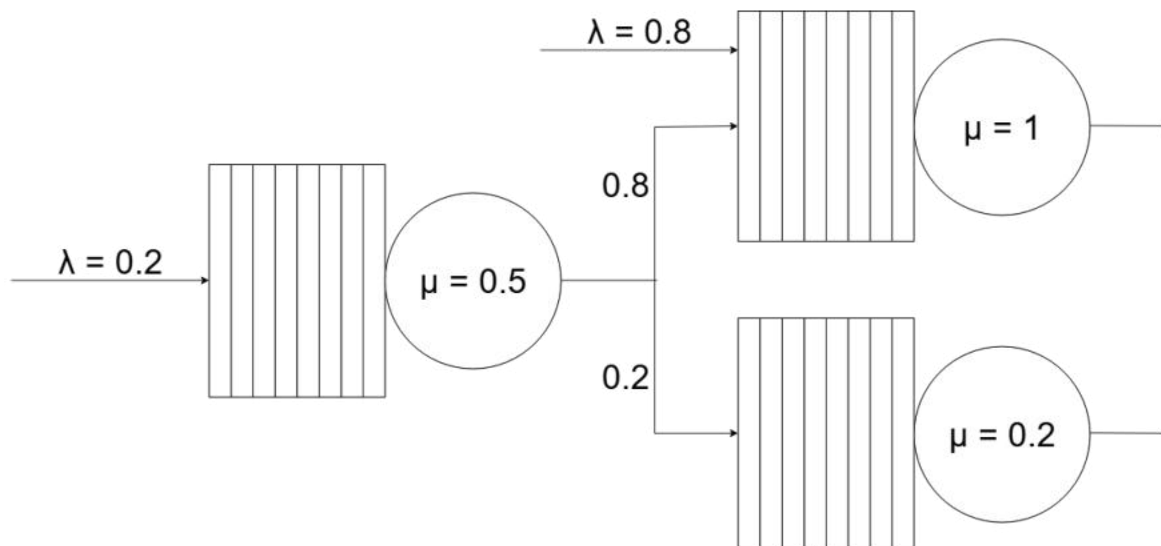
Figura 3 – Fila M/M/1



Em contrapartida, na figura 4 um sistema mais complexo é mostrado. Com dois pontos de entrada e três servidores, um novo fator é acrescentado à rede: conexões entre servidores. Em sistemas complexos, muitas vezes torna-se necessário acoplar componentes

com funções distintas para que possam atuar em uma mesma tarefa. Tal conexão pode ser feita de forma probabilística ou determinística.

Figura 4 – Fila híbrida com 3 servidores



2.2.4 Métricas

Diversos parâmetros são cruciais para a completa análise dos sistemas, principalmente suas características e performance. A partir das métricas de desempenho, os profissionais serão capazes de tomar decisões técnicas que visem a otimização de determinado sistema de servidores. Os parâmetros podem ser obtidos de forma prática, através de simulações, ou de forma teórica, através de cálculos. As principais métricas de performance são listadas a seguir.

- **Utilização do servidor (ρ):** Proporção do tempo em que o servidor está ocupado processando jobs. É dada por $\rho = \frac{\lambda}{\mu}$ em sistemas $M/M/1$ estáveis. Exemplo: $\rho = 0.8$ (80% do tempo ocupado).

- **Tempo médio de resposta ($E[T]$):** Tempo médio total que um job passa no sistema, incluindo fila e serviço. Pode ser obtido subtraindo o tempo de saída do job do sistema pelo tempo de chegada. Exemplo: $T = 2.1$ segundos/job.

- **Tempo médio de espera na fila ($E[T_q]$):** Tempo médio que um job aguarda antes de ser atendido durante toda sua estada no sistema. Exemplo: $T_q = 1.7$ segundos/job.

- **Demanda ($E[D]$):** Valor esperado do tempo que um job gasta em um dispositivo durante sua estada no sistema, ao longo de todas as suas visitas ao dispositivo. Exemplo: $D = 2$ segundos/job.

- **Visitas a um dispositivo ($E[V]$):** Valor esperado da quantidade de visitas de um job a um dispositivo durante sua estada no sistema. Exemplo: $V = 1.2$ visitas/job.

- **Número médio de jobs no sistema ($E[N]$):** Valor esperado da quantidade de jobs presentes no sistema em um momento qualquer. Exemplo: $N = 4.2$ jobs.
- **Número médio de jobs na fila ($E[N_q]$):** Valor esperado da quantidade de jobs aguardando atendimento em alguma fila. Exemplo: $N_q = 3.5$ jobs.
- **Vazão (X):** Taxa média de jobs concluídos por unidade de tempo. Em sistemas estáveis, é igual à taxa média de chegadas: $X = \lambda$. Exemplo: $X = 2$ jobs/segundo.

Com as métricas fornecidas, já é possível realizar inferências sobre o desempenho dos sistemas. Um dispositivo que opera em utilização máxima, por exemplo, representa um gargalo no funcionamento geral. Da mesma forma, tempos de espera excessivamente elevados podem indicar problemas de concepção do sistema ou insuficiência de capacidade para atender à demanda.

2.3 ANÁLISE TEÓRICA

Com o conhecimento adquirido até aqui, algumas fórmulas, leis e conceitos teóricos podem ser explorados. Estas ferramentas serão essenciais para verificar a corretude do software desenvolvido, uma vez que os valores obtidos por meio de simulação devem apresentar consistência quando comparados com os resultados previstos pela análise teórica.

Entretanto, cenários mais complexos, que envolvem distribuições de tempo mais elaboradas e disciplinas de atendimento não convencionais, podem não dispor de ferramental teórico suficiente para uma análise analítica adequada, tornando necessária, portanto, a utilização de simulações.

2.3.1 Lei de Little

A Lei de Little (LITTLE; GRAVES, 2008) é possivelmente a mais conhecida e aplicada na teoria de filas. Ela estabelece uma relação direta entre o tempo médio de permanência de entidades no sistema e o número médio de entidades presentes em determinado instante. A formulação é válida tanto para sistemas abertos quanto para sistemas fechados.

Teorema 1 *Para todo sistema aberto estável, vale que*

$$E[N] = \lambda \cdot E[T]$$

Teorema 2 *Para todo sistema fechado estável, vale que*

$$N = X \cdot E[T]$$

O Teorema 1 apresenta a formulação da Lei de Little para sistemas abertos. Ressalta-se que a validade do resultado independe das disciplinas de chegada, de serviço e de fila, o que o torna altamente genérico. O Teorema 2, por sua vez, apresenta a versão para

sistemas fechados. Como N é constante nesse caso, não há necessidade de considerar o valor esperado.

2.3.2 Lei do Fluxo Forçado (Forced Flow Law)

A Lei do Fluxo Forçado, proposta por Denning e Buzen (DENNING; BUZEN, 1978), estabelece a relação entre a taxa de vazão global do sistema e a taxa de visitas a cada recurso. Ela é fundamental para a análise de sistemas fechados, pois permite inferir o comportamento de cada servidor a partir da carga global.

Teorema 3 *Seja X a vazão do sistema e $E[V_i]$ o número médio de visitas de um job ao servidor i. A vazão do servidor i, denotado por X_i, é dada por:*

$$X_i = E[V_i] \cdot X$$

O teorema apresentado mostra que a vazão em cada servidor é proporcional à vazão do sistema, ponderado pelo número médio de visitas que cada job realiza nesse servidor. Assim, uma vez conhecida a vazão global, é possível derivar a vazão de cada recurso individual.

Essa lei é de grande importância prática, pois dispensa o detalhamento de processos internos de cada servidor. Basta conhecer as taxas médias de visitação ($E[V_i]$) para determinar, de forma direta e precisa, a carga de cada recurso do sistema.

2.3.3 Lei do Gargalo (Bottleneck Law)

A Lei do Gargalo estabelece a relação entre a vazão de um dispositivo e sua utilização, possibilitando a identificação do recurso mais sobrecarregado do sistema. A partir dessa informação, torna-se viável propor alterações estruturais com o objetivo de melhorar a vazão total.

Teorema 4 *Seja $E[D_i]$ a demanda média de serviço de um job no servidor i, e seja X a vazão de um sistema fechado. A utilização do servidor i, denotada por ρ_i, é dada por:*

$$\rho_i = X \cdot E[D_i]$$

É importante observar que, embora a relação permaneça formalmente válida, o teorema não apresenta o mesmo impacto prático em sistemas abertos, nos quais o foco principal recai sobre a verificação da estabilidade do sistema, e não sobre a determinação de sua vazão máxima.

Ainda assim, a Lei do Gargalo constitui uma ferramenta essencial no planejamento de capacidade e na análise de desempenho de sistemas computacionais.

2.3.4 Limite assintótico para sistemas fechados

A partir da Lei do Gargalo, em conjunto com os conhecimentos referentes a sistemas fechados, é possível estabelecer um limite superior teórico para a vazão total nesses sistemas, conforme descrito em (HARCHOL-BALTER, 2013).

Teorema 5 *Seja i o servidor cuja demanda é a maior entre todos os servidores do sistema, e seja $D_{max} = E[D_i]$ essa demanda. Adicionalmente, seja D a soma das demandas de todos os servidores presentes no sistema, e seja $E[Z]$ o tempo médio de pensamento dos terminais. O limite superior para a vazão de um sistema fechado, denotado por X_{max}, é dado por:*

$$X_{max} = \min\left(\frac{N}{D + E[Z]}, \frac{1}{D_{max}}\right)$$

O teorema demonstra que o desempenho máximo de um sistema nem sempre é limitado exclusivamente pela maior demanda de serviço. O ponto de interseção entre ambas as curvas, representado por N^*, indica a partir de qual valor de N o gargalo passa a ser o principal fator limitante da rede. Para quantidades de terminais inferiores a esse valor, melhorias em qualquer servidor do sistema podem resultar em aumento da vazão total.

Por fim, ressalta-se que o limite superior apresentado é de natureza teórica, de modo que nem sempre é possível atingi-lo em um sistema real.

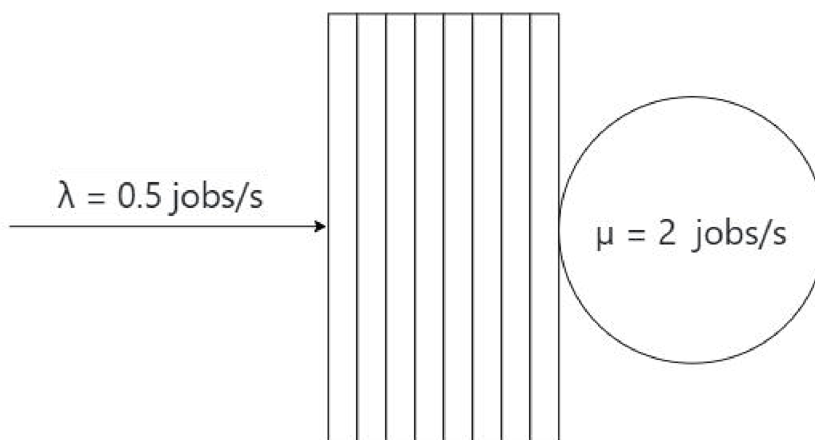
3 ESTUDOS DE CASO

Este capítulo é destinado à aplicação dos conceitos revisados nos capítulos anteriores, demonstrando como analisar redes de filas de maneira teórica.

3.1 SISTEMA M/M/1

Sistemas de filas M/M/1, ou seja, com apenas um servidor, um ponto de entrada de jobs, e distribuições exponenciais, são frequentemente usadas dentro do contexto da teoria de filas. Desta forma, uma análise teórica pode ser realizada utilizando as ferramentas já discutidas neste trabalho.

Figura 5 – Fila M/M/1



Considerando o sistema exibido na figura 5, inicialmente podemos extrair as seguintes métricas.

- **ρ:** Dada por $\rho = \frac{\lambda}{\mu}$, logo, $\rho = 0.25$ (25% do tempo ocupado).
- **V:** Todos os jobs chegam pelo servidor e, após serem servidos, terminam. Dessa forma, $V = 1$ visitas/job.
- **X:** Como o sistema é equilibrado ($\mu > \lambda$), $X = 0.5$ jobs/segundo.

Conforme descrito em (HARCHOL-BALTER, 2013), para sistemas M/M/1, é possível extrair uma fórmula fechada para a quantidade esperada de jobs no sistema, devido à natureza das distribuições de chegada e serviço. Assim, usando a lei de Little, torna-se viável extrair uma fórmula para o tempo médio gasto por cada job no sistema.

Teorema 6 *Para todo sistema de filas M/M/1, vale que*

$$E[T] = \frac{E[N]}{\lambda} = \frac{1}{\mu-\lambda}$$

Com esse resultado, mais algumas métricas podem ser obtidas:

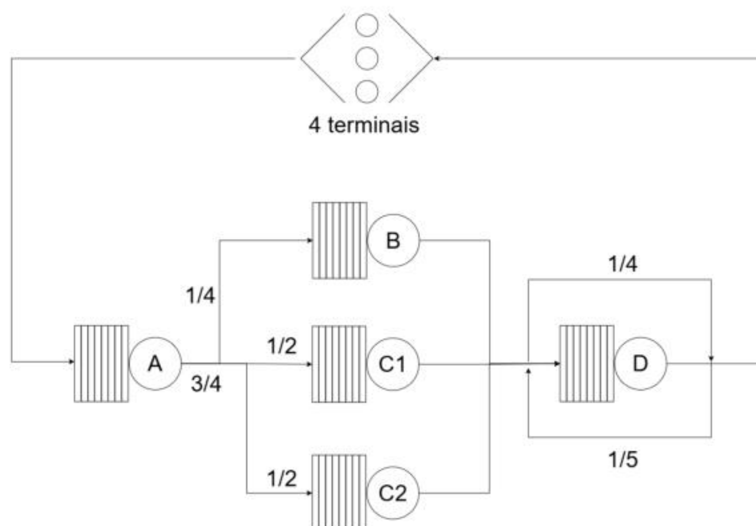
- $\mathbf{E[T]} = \frac{1}{\mu-\lambda} = \frac{1}{2-0.5} \approx 0.666$ s/job.
- $\mathbf{E[T_q]} = E[T] - E[S] = 0.666 - 0.5 \approx 0.166$ s/job.
- $\mathbf{E[N]} = \lambda \cdot E[T] = 0.5 \cdot 0.666 \approx 0.333$ jobs.
- **D**: Como cada job só passa uma vez pelo servidor, $D = 0.5$ s/job.

Assim, é possível extrair todas as métricas cruciais definidas no capítulo anterior apenas com os parâmetros λ e μ. Em casos mais complexos, entretanto, nem sempre será possível determinar as métricas desejadas de forma clara sem realizar experimentos.

3.2 UM CASO MAIS COMPLEXO

Para elucidar de forma mais clara a utilização das técnicas teóricas estudadas na extração de métricas em sistemas robustos, com múltiplos dispositivos e diferentes distribuições de entrada (tempo entre chegadas), de serviço, e disciplinas de fila, torna-se necessário analisar uma rede com mais elementos.

Figura 6 – Sistema Fechado



Considere o sistema fechado ilustrado na figura 6. O sistema é composto por 4 terminais e 5 servidores, todos com tempos de serviço distribuídos exponencialmente. Todos os jobs passam inicialmente pelo servidor A e, após essa etapa, são encaminhados ao servidor B com probabilidade de 25%, ou aos servidores C1 e C2 com probabilidade de 75%. Os servidores C1 e C2 compartilham a carga total de processamento, sendo que cada job possui 50% de chance de ser direcionado a um deles.

Jobs que passam pelos servidores do tipo C não passam pelo servidor B, e vice-versa. Após essas etapas, todos os jobs convergem novamente para serem processados pelo servidor D. No entanto, 25% dos jobs contornam o servidor D, seguindo diretamente para o final do sistema. Além disso, 20% das tarefas processadas pelo servidor D retornam para uma nova execução, independentemente do número de vezes em que já foram atendidas. Por fim, jobs que não passaram pelo servidor D possuem probabilidade nula de retorno, assim como aqueles que retornam não podem contornar o servidor.

Além das características do sistema, as seguintes informações são conhecidas:

- $E[Z] = 0$ segundos. Não há espera para o envio de um novo job quando um atual termina sua execução (sistema em *batch*).
- $E[S_A] = 8$ segundos/job.
- $E[S_C] = 6$ segundos/job, para ambos os servidores.
- $E[T_q] = 17$ segundos.
- $E[N_B^Q] = \frac{1}{40}$ jobs.
- $X_B = \frac{1}{33}$ jobs/segundo.
- $X_D = \frac{1}{9}$ jobs/segundo.
- $\rho_D = 22\%$.

Considerando apenas os jobs processados pelo servidor D, observa-se que, a cada passagem, a probabilidade de não retornar é de 80%. Assim, esse comportamento pode ser modelado por uma distribuição geométrica, cuja probabilidade de sucesso é igual a 0.8. Como o valor esperado de uma variável aleatória geométrica é dado pelo inverso da probabilidade de sucesso, a quantidade média de vezes que um job é atendido pelo servidor D, dado que foi processado ao menos uma vez, é igual a $\frac{1}{0.8} = \frac{5}{4}$.

Sabendo que 75% dos jobs são processados pelo servidor D pelo menos uma vez, tem-se:

$$E[V_D] = \tfrac{3}{4} \cdot \tfrac{5}{4} = \tfrac{15}{16} \text{ visitas/job.}$$

De posse de $E[V_D]$ e de X_D, pode-se utilizar a Lei do Fluxo Forçado para determinar a vazão total do sistema:

$$\begin{aligned} X_D &= V_D \cdot X \\ \frac{1}{9} &= \frac{15}{16} \cdot X \\ X &= \frac{16}{135} \text{ jobs/segundo.} \end{aligned}$$

Em sistemas fechados, o tempo de permanência total de um job inclui o tempo de pensamento. Por essa razão, $E[T]$ é decomposto em $E[R] + E[Z]$, onde R é a variável aleatória que representa o tempo em filas ou sendo executado por servidores, enquanto Z representa o tempo de pensamento. Entretanto, como o tempo de pensamento é zero no exemplo analisado, é razoável considerar apenas o tempo na rede, comportamento consistente com um sistema em *batch*.

Por se tratar de um sistema fechado, também é possível concluir que há exatamente 4 jobs presentes no sistema em qualquer instante. Assim, conhecendo-se N e X, aplica-se a Lei de Little para calcular o tempo médio total de permanência de um job no sistema:

$$\begin{aligned} N &= X \cdot E[T] \\ 4 &= \frac{16}{135} \cdot E[T] \\ E[T] &= 33.75 \text{ segundos.} \end{aligned}$$

Dessa forma, a principal métrica do sistema, o tempo médio total de permanência de cada job, pôde ser determinada.

Com as informações gerais do sistema estabelecidas, torna-se mais simples extrair métricas específicas de cada dispositivo.

Cada job é processado por um dos servidores do tipo C com probabilidade igual a 75%. Como o roteamento entre eles é equiprovável, cada servidor C recebe, em média, 37.5% dos jobs totais, correspondendo a $E[V_{C_1}]$ e $E[V_{C_2}]$. A partir disso, é possível calcular a vazão de C1 e C2:

$$\begin{aligned} X_{C_i} &= X \cdot E[V_{C_i}] \\ X_{C_i} &= \frac{16}{135} \cdot \frac{3}{8} \\ X_{C_i} &= \frac{2}{45} \text{ jobs/segundo.} \end{aligned}$$

Além de aplicar a Lei do Fluxo Forçado considerando os servidores como um todo, também é possível utilizá-la especificamente para as filas, de modo a obter métricas relacionadas exclusivamente ao tempo de espera em cada dispositivo. Como todos os jobs processados por um servidor necessariamente passam por sua fila, ainda que não haja espera efetiva, a quantidade média de visitas por job, $E[V]$, permanece a mesma.

$$X_{B_Q} = X \cdot E[V_{B_Q}]$$
$$X_{B_Q} = \frac{16}{135} \cdot \frac{1}{4}$$
$$X_{B_Q} = \frac{4}{135} \text{ jobs/segundo.}$$
$$E[T_B^Q] = \frac{E[N_B^Q]}{X_{B_Q}}$$
$$E[T_B^Q] = 0.84 \text{ segundos/job (para os jobs que passam por B)}$$
$$E[T_B^Q] = 0.21 \text{ segundos/job, considerando que apenas 25\% dos jobs passam por B.}$$

Por fim, é possível utilizar os resultados obtidos para calcular as demandas de todos os dispositivos. Com o conhecimento dessas demandas, torna-se viável identificar qual servidor representa o gargalo do sistema. Assim, o tempo total de processamento pode ser reduzido a partir da melhoria da capacidade do dispositivo crítico, resultando em uma maior produtividade do sistema.

- $E[D_D] = \frac{\rho_D}{X} = \frac{297}{160}$ segundos/job.
- $E[D_{C_i}] = E[V_{C_i}] \cdot E[S_{C_i}] = \frac{9}{4}$ segundos/job.
- $E[D_A] = E[V_A] \cdot E[S_A] = 8$ segundos/job.
- $E[D_B] = 2.40$ segundos/job (tempo total subtraído das demais demandas e do tempo em filas).

Com base nesses resultados, verifica-se que o servidor A é o gargalo do sistema, segurando os jobs com altos tempos de fila. Portanto, caso seja necessária uma otimização da rede, a ação mais adequada seria aumentar a capacidade de processamento desse dispositivo, visto que o número de jobs é grande o suficiente para manter o gargalo com utilização máxima. Note que existem casos onde aumentar a velocidade do gargalo não solucionará muitos problemas, caso os demais servidores também possuam altas demandas. Nesses cenários, uma redistribuição de carga, organizando a rede de uma forma mais eficiente, pode ser o suficiente, oferecendo uma alternativa com custo reduzido.

Evidencia-se, assim, que é possível analisar teoricamente uma ampla variedade de sistemas com diferentes configurações. No entanto, a ausência de informações completas sobre o sistema, bem como a presença de distribuições de chegada e serviço ou disciplinas de fila distintas do padrão, pode tornar as análises teóricas significativamente mais complexas. Dessa forma, o software desenvolvido neste trabalho auxilia na obtenção de métricas fundamentais por meio de simulações, viabilizando análises detalhadas e de caráter prático.

4 SOBRE A BIBLIOTECA DE SIMULAÇÃO

Este capítulo é dedicado à apresentação e discussão da biblioteca desenvolvida no âmbito deste trabalho, detalhando as ideias que nortearam o processo de concepção e as principais decisões técnicas tomadas durante sua implementação. O objetivo é expor não apenas os aspectos funcionais dos artefatos, mas também as motivações que levaram às escolhas de arquitetura e tecnologia.

A biblioteca de simulação é o principal produto desenvolvido neste trabalho. Trata-se do componente responsável pela execução das simulações, pelo controle dos eventos e pela coleta e processamento das métricas resultantes.

A motivação para o seu desenvolvimento surgiu da ausência de ferramentas com configuração simples e didáticas disponíveis em Python para simular sistemas de filas. Embora existam bibliotecas consolidadas, como SimPy, essas soluções tendem a exigir uma configuração detalhada e um conhecimento prévio considerável sobre o ambiente de simulação, especialmente em cenários envolvendo múltiplos servidores, filas interconectadas ou políticas complexas de roteamento.

Outro diferencial importante é a coleta estruturada de métricas. Durante a execução, a biblioteca monitora variáveis relevantes, como tempo médio de espera em fila, tempo médio de serviço, tempo total no sistema, número médio de jobs e taxa de utilização de cada servidor. Ao término da simulação, essas informações são organizadas em um formato padronizado e prontas para análise ou visualização. Isso elimina a necessidade de cálculos manuais e garante consistência entre diferentes experimentos.

Por fim, a arquitetura proposta favorece o uso para sistemas variados, permitindo a utilização de diversos tipos de políticas de atendimento e distribuições de chegada ou serviço, todas de forma encapsulada. Dessa forma, a biblioteca pode evoluir de acordo com as necessidades de novos estudos e aplicações, mantendo-se como uma ferramenta robusta, acessível e de fácil integração com diferentes contextos de pesquisa e ensino.

4.1 ARQUITETURA

Conforme citado anteriormente, a linguagem escolhida para o desenvolvimento da biblioteca foi o Python. Apesar de possuir suporte nativo à orientação a objetos, a linguagem não tem essa característica em particular como seu ponto forte, sendo utilizada com maior frequência em projetos de computação científica. Entretanto, Python é uma linguagem amplamente conhecida e difundida, com um grande número de programadores que a utilizam. Além disso, é reconhecida por sua sintaxe simples, característica que auxilia desenvolvedores menos experientes. Dessa forma, visando maximizar a usabilidade, esta linguagem foi escolhida.

Os princípios fundamentais da orientação a objetos, assim como as melhores práticas no desenvolvimento de software, foram respeitados. O código foi modularizado em classes, cada uma desempenhando um papel essencial no resultado final do artefato. O cliente, entretanto, precisa ter acesso apenas a uma dessas classes para iniciar o processo de configuração do ambiente. Dessa forma, a lógica de execução permanece encapsulada na biblioteca.

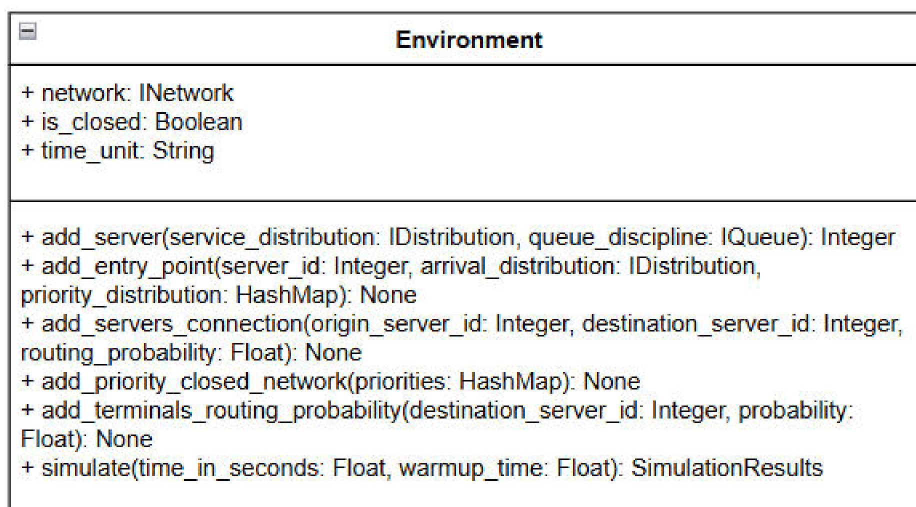
4.2 CLASSES

4.2.1 Environment (Ambiente)

A primeira classe, e também a que orquestra as demais, é a classe *Environment*. Ao ser instanciada, pode receber como parâmetros a quantidade de terminais, a distribuição do tempo de pensamento e a unidade de tempo desejada. Caso os dois primeiros parâmetros sejam fornecidos, a rede criada será fechada. Em contrapartida, caso pelo menos um deles não seja informado, a rede criada será aberta. Por fim, a unidade de tempo pode ser definida para facilitar a visualização dos resultados ao final da simulação, abrangendo situações em que o cliente não fornece os parâmetros de entrada em segundos. Este, entretanto, é o valor padrão quando o parâmetro não é fornecido.

Esta é a única classe que necessita ser instanciada pelo cliente. É responsável por gerenciar o estado da rede, fornecer os parâmetros adequados à simulação e retornar os resultados.

Figura 7 – Classe Environment



A figura 7 apresenta a modelagem UML da classe. A variável *network* possui o tipo *INetwork*, uma interface criada para permitir o uso tanto de redes abertas quanto de redes fechadas. O método *simulate* é responsável por fornecer a configuração definida pelo usuário a uma instância da classe *Execution*, encarregada de executar a simulação e retornar os resultados. O estado dos resultados é armazenado na classe *SimulationResults*,

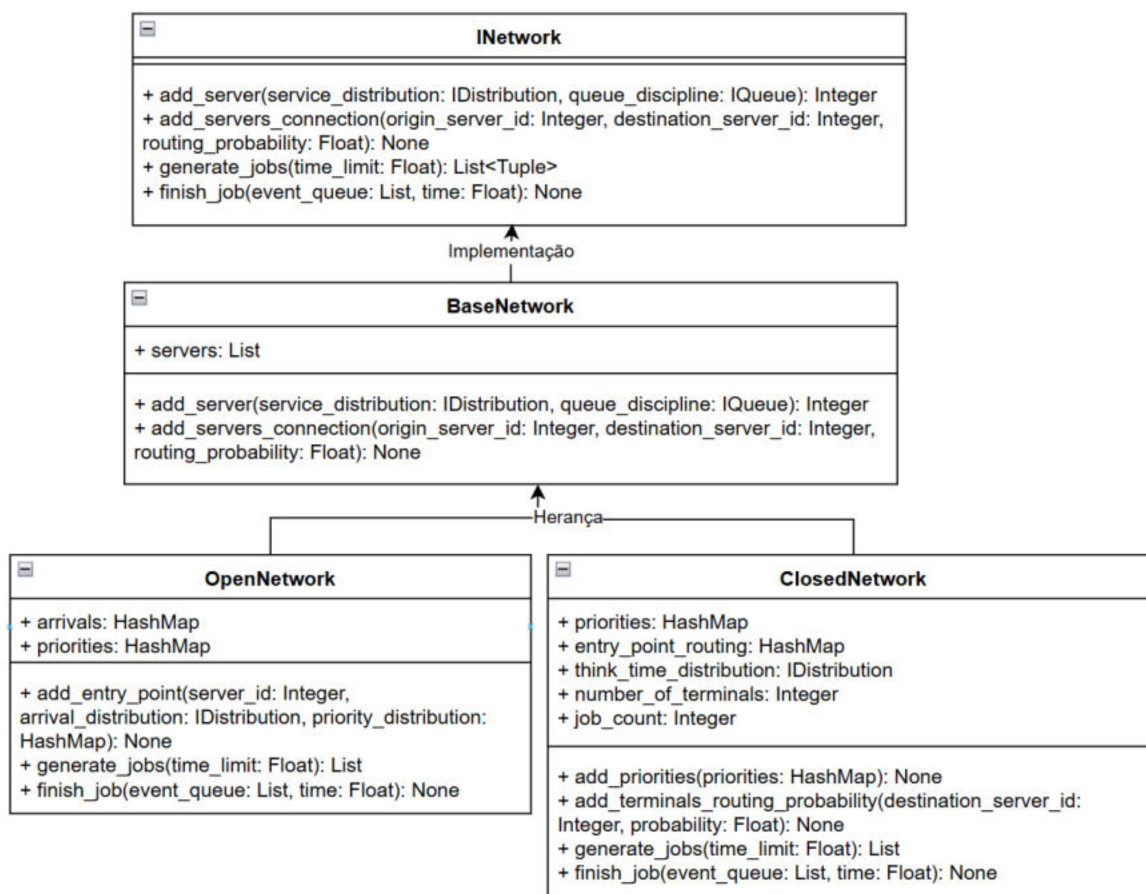
que disponibiliza métodos para acesso aos dados, sem necessidade de manipulação direta das variáveis. O ciclo de vida da instância do objeto *Execution* corresponde à duração da simulação, sendo criada uma nova instância a cada execução.

Ao organizar o código em diferentes classes, mantém-se a limpeza e modularidade, facilitando a evolução do projeto, a adição de novas funcionalidades e a realização de testes.

4.2.2 Network (Rede)

O próximo conjunto de classes refere-se à rede utilizada na simulação. A escolha por utilizar uma interface deve-se à necessidade de empregar os dois tipos de rede, aberta e fechada, na classe que controla o ambiente. Dessa forma, é garantido que os métodos necessários estarão implementados em ambas as classes, permitindo a abstração completa da implementação relacionada às redes no código cliente, ou seja, na classe *Environment*.

Figura 8 – Classes Network



A figura 8 mostra a modelagem UML da interface, assim como das classes relacionadas. Para extrair o máximo proveito da orientação a objetos e evitar a repetição de código, foi criada uma classe base com o objetivo de conter os métodos e variáveis em comum entre as duas implementações. Note que a classe base não implementa a interface por

completo, apenas parte de seus métodos. Dessa forma, apenas as classes *ClosedNetwork* e *OpenNetwork* implementam integralmente a interface.

As classes de rede também são responsáveis por gerenciar o estado dos servidores, que alteram as propriedades dos jobs, bem como por gerar novas tarefas. As classes que representam os servidores e os jobs são fundamentais para a biblioteca, já que é nesses artefatos que está o maior interesse na coleta de métricas.

A presença de métodos na interface que são necessários apenas em um dos tipos de rede é indispensável para que a generalização possa substituir ambas as implementações concretas na classe cliente. Para manter as implementações consistentes, chamadas a métodos referentes a um tipo de rede em uma instância de outro tipo disparam uma exceção, indicando ao usuário que o uso está incorreto. Como o usuário final não tem acesso direto às classes de rede, o único cenário que possibilita esse erro é a ocorrência de um bug na própria biblioteca.

4.2.3 Server e Job (Servidor e Tarefa)

As próximas classes que merecem destaque são aquelas que despertam o maior interesse em relação às métricas: *Server* e *Job*. A classe *Server*, para ser instanciada, precisa de uma distribuição de serviço e de uma disciplina de fila, além de receber um identificador numérico para que seja possível adicionar conexões e identificá-la nas métricas. Tanto as distribuições quanto as disciplinas de fila são utilizadas por meio de uma interface. Com esse padrão, a adição ou remoção de opções torna-se extremamente simples, bastando criar uma nova classe que implemente os métodos e variáveis necessários. A classe *Server* utiliza ainda outra classe, responsável por gerenciar o estado de execução.

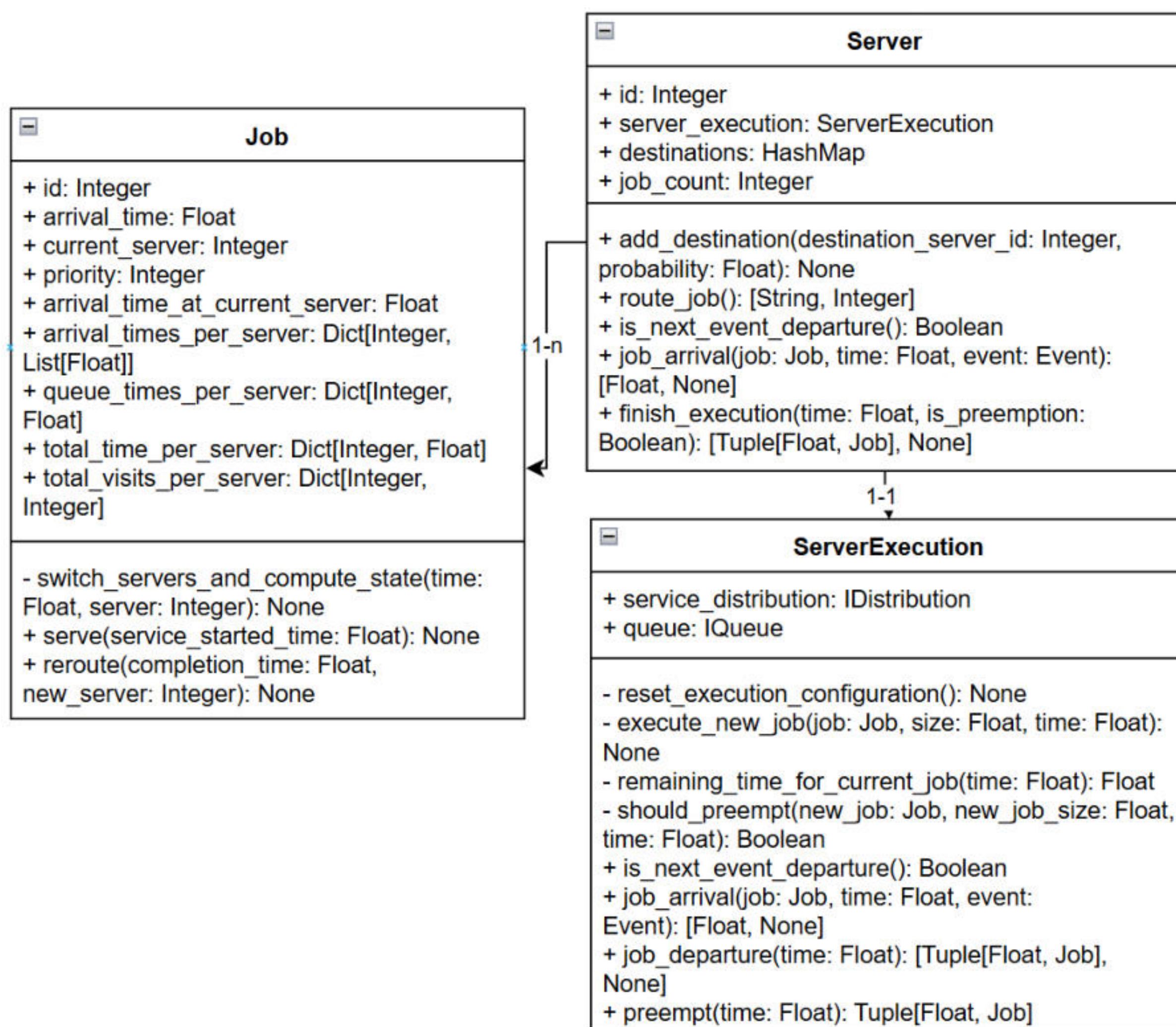
Já a classe *Job*, que representa as tarefas executadas no sistema, é relativamente simples. Sua principal função é preservar o estado do *job* enquanto ele percorre os dispositivos. Por essa razão, é importante registrar os dispositivos pelos quais passou, quantas vezes o fez, o tempo total gasto em cada servidor e em sua fila, entre outras métricas. Para o armazenamento dos dados históricos, foram utilizados *hashmaps*, organizando as informações por servidor.

4.2.4 Metrics (Métricas)

Os dados da simulação são armazenados e gerenciados pelas classes de métricas. Uma classe genérica contém os métodos e propriedades que podem ser utilizados por todas as métricas: ambiente, servidores e prioridade. Assim, as classes *EnvironmentMetrics*, *ServerMetrics* e *PriorityMetrics* herdam da classe *GenericMetrics*, adicionando os métodos exclusivos necessários.

Também são fornecidos métodos que retornam todos os dados coletados já no formato desejado, garantindo tanto a exclusividade de acesso às variáveis de instância da própria

Figura 9 – Classes Server e Job



classe quanto a simplicidade na obtenção das métricas por parte do código cliente.

4.2.5 SimulationResults (Resultados da Simulação)

A classe responsável por controlar o estado das métricas é a *SimulationResults*. Ela dispõe dos métodos que são expostos para adicionar novos dados aos resultados da simulação.

Para evitar repetição de código, os objetos de métricas são públicos, permitindo o acesso direto aos dados. Entretanto, quando a biblioteca é utilizada pelo cliente, a classe também oferece um método para exibir os resultados de forma organizada no console, facilitando a visualização das informações.

Figura 10 – Classes Metrics

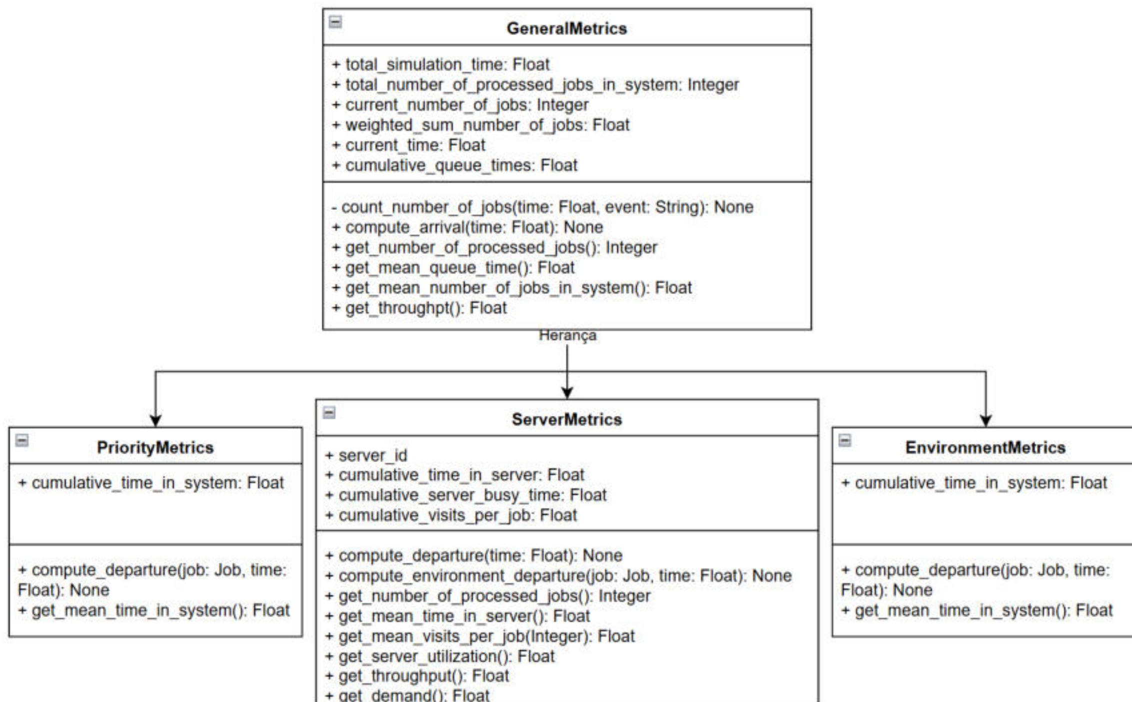
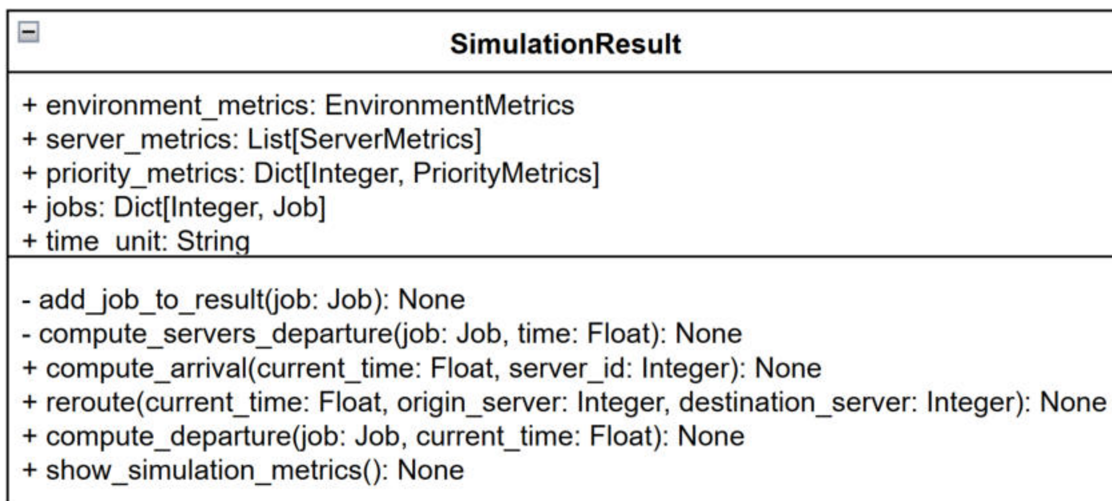


Figura 11 – Classe SimulationResults

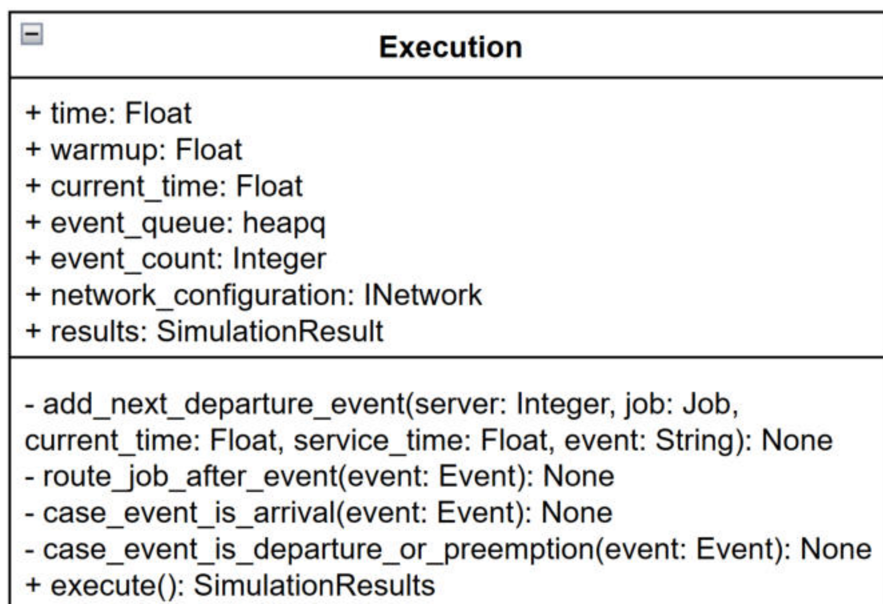


4.2.6 Execution (Execução)

Por fim, a última classe que merece destaque é aquela responsável por orquestrar a execução da simulação. Instanciada pela classe *Environment*, ela recebe as configurações de rede e os eventos, alterando o estado do sistema com base nas características de cada evento.

Todas as classes apresentadas compõem o sistema completo, permitindo a configuração e execução das simulações, bem como a coleta das métricas de interesse por parte dos usuários. O diagrama de classes completo, incluindo todas as classes e suas dependências,

Figura 12 – Classe Execution



pode ser consultado através do link[1].

4.3 IMPLEMENTAÇÃO

Para o desenvolvimento do artefato, foi utilizada a versão 3.13 do Python, lançada em outubro de 2024. A escolha dessa versão se deve ao fato de ser recente, estável e segura, ao mesmo tempo em que já conta com amplo suporte e com versões consolidadas das principais bibliotecas necessárias ao projeto.

A biblioteca *Pydantic* foi empregada para realizar a validação dos tipos de entrada em todas as classes e métodos que interagem com o usuário. Essa medida é fundamental, considerando que a linguagem não realiza checagem de tipos em tempo de execução. Para a anotação dos tipos, também foi utilizada a biblioteca *Typing*.

A geração de amostras das distribuições foi implementada com o auxílio da biblioteca *NumPy*. Além disso, estruturas de dados essenciais, como *heaps*, filas e *hashmaps*, foram utilizadas conforme necessário. Por fim, a biblioteca *abc* foi empregada para garantir a implementação completa de todas as interfaces definidas no projeto.

4.3.1 Configuração do Ambiente

O primeiro passo para realizar uma simulação consiste na instanciação de um objeto da classe *Environment*, fornecendo os devidos parâmetros de configuração.

[1] https://tinyurl.com/5b3ezny5

Código 1 – Método construtor da classe Environment

```
@validate_call(config=dict(arbitrary_types_allowed=True))
def __init__(self, number_of_terminals: Optional[int] = None,
    think_time_distribution: Optional[IDistribution] = None, time_unit:
    Optional[str] = 'seconds'):
    self._network = None
    self._is_closed = False
    self._time_unit = time_unit

    if number_of_terminals is None or think_time_distribution is None or
        number_of_terminals <= 0:
        self._network = OpenNetwork()
    else:
        self._network = ClosedNetwork(think_time_distribution,
            number_of_terminals)
        self._is_closed = True
```

Após a instanciação, o usuário pode adicionar servidores, independentemente do tipo de rede. A chamada para adicionar um servidor é feita na classe *Environment*, que faz a validação dos parâmetros através do *Pydantic*. Caso estejam corretos, a função que adiciona servidores no objeto que implementa INetwork é chamada.

Caso nenhuma disciplina de fila seja especificada, uma fila padrão é atribuída automaticamente, adotando o critério *First Come, First Served* (FCFS), no qual os jobs são atendidos na ordem de chegada, comportamento esperado na maioria dos sistemas de filas.

No momento da criação, jobs que saem do servidor são finalizados com probabilidade igual a um. Conforme novas conexões vão sendo adicionadas, as probabilidades de roteamento fornecidas são subtraídas da probabilidade de encerramento. O método retorna o valor inteiro que identifica aquele servidor.

Código 2 – Método add_server da classe BaseNetwork

```
def add_server(self, service_distribution: IDistribution,
    queue_discipline: Optional[IQueue] = None) -> int:
    if not queue_discipline:
        queue_discipline = QueueDiscipline.fcfs()

    server_id = len(self.servers)
    self.servers.append(Server(server_id, service_distribution,
        queue_discipline))

    return server_id
```

A implementação das distribuições é baseada em princípios de herança, polimorfismo e no padrão de projeto *Factory*. A interface *IDistribution* define os métodos e propriedades

essenciais que todas as distribuições devem implementar para viabilizar a simulação. Cada distribuição concreta, portanto, implementa essa interface.

Com essa abordagem, novas distribuições podem ser adicionadas ou removidas sem que o restante do código seja modificado, em conformidade com os princípios de abertura/fechamento e responsabilidade única descritos por (MARTIN, 2000).

O padrão de fábrica (*Factory*) foi escolhido para simplificar o processo de instanciação e promover o encapsulamento. Assim, o usuário interage apenas com uma classe estática central, a *Distribution*, que contém métodos responsáveis por criar e retornar as instâncias corretas com base nos parâmetros fornecidos (GAMMA et al., 1995).

Dessa forma, o usuário não precisa conhecer a estrutura interna das distribuições, nem realizar importações individuais de cada classe concreta. Todo o processo de criação fica centralizado e automatizado, garantindo maior consistência, modularidade e facilidade de uso.

Código 3 – Classe IDistribution e exemplo de implementação

```
class IDistribution(ABC):
    @abstractmethod
    def sample(self):
        pass

class ConstantDistribution(IDistribution):
    def __init__(self, value: float):
        self._value = value

    def sample(self) -> float:
        return round(self._value, 4)
```

Código 4 – Classe que implementa o padrão de fábrica

```
class Distribution():
    def __new__(cls, *args, **kwargs):
        if cls is Distribution:
            raise TypeError("Cannot instantiate this class directly.")
        return super().__new__(cls, *args, **kwargs)

    @staticmethod
    @validate_call
    def constant(value: float) -> ConstantDistribution:
        return ConstantDistribution(value)

    @staticmethod
    @validate_call
    def exponential(lambda_value: float) -> ExponentialDistribution:
        return ExponentialDistribution(lambda_value)

    @staticmethod
    @validate_call
    def uniform(lower_bound: float, upper_bound: float) ->
       UniformDistribution:
        if upper_bound >= lower_bound:
            return UniformDistribution(lower_bound, upper_bound)
        raise ValueError('Lower bound should be smaller than upper bound
           .')

    @staticmethod
    @validate_call
    def normal(mu: float, sigma: float) -> NormalDistribution:
        return NormalDistribution(mu, sigma)
```

Por ser uma classe estática, *Distribution* não permite instanciação. Além disso, todos os métodos públicos estão devidamente documentados, de modo que o usuário possa compreender claramente suas funcionalidades e a forma correta de utilizar seus parâmetros.

A implementação das filas segue a mesma lógica. A interface *IQueue* define a obrigatoriedade de implementação dos métodos *first_in_line*, *insert* e *with_preemption*, garantindo que as classes concretas sejam suficientemente genéricas. Por fim, o usuário tem acesso a todas as implementações por meio da classe estática *QueueDiscipline*, que disponibiliza métodos para instanciar cada tipo de disciplina de fila.

A adição de arestas entre servidores é realizada de forma semelhante à criação dos próprios servidores. O usuário invoca o método correspondente na classe *Environment*, fornecendo os servidores de origem e destino, além da probabilidade de roteamento. A classe, então, valida os tipos e, caso os parâmetros estejam corretos, delega a criação da conexão ao objeto de rede.

Código 5 – Método add_servers_connection da classe BaseNetwork

```
def add_servers_connection(self, origin_server_id: int,
    destination_server_id: int, routing_probability: float):
    validate_number_params_not_negative_and_not_none(function_name='
        add_servers_connection', origin_server_id=origin_server_id,
        destination_server_id=destination_server_id, routing_probability=
        routing_probability)

    number_of_servers = len(self.servers)

    if origin_server_id < number_of_servers and destination_server_id <
        number_of_servers:
        self.servers[origin_server_id].add_destination(
            destination_server_id, routing_probability)

    else:
        raise ValueError(f'Provided server id is not valid. Received {
            origin_server_id if origin_server_id >= number_of_servers
            else destination_server_id} when only {number_of_servers}
            were created (Index starts at 0).')
```

O método *validate_number_params_not_negative_and_not_none* foi desenvolvido para realizar uma verificação simples de tipos e valores, evitando a sobrecarga de desempenho causada pela validação mais complexa do *Pydantic*. Assim, métodos de validação mais leves, como este, foram criados com o objetivo de aumentar a segurança dos tipos nas partes internas do código sem comprometer a eficiência.

Para configurar a chegada de jobs, são disponibilizados dois métodos distintos. Em redes fechadas, existem apenas os terminais, ou seja, o sistema não possui chegada externa de jobs. Nesse caso, o usuário pode configurar o roteamento a partir do terminal, especificando o servidor de destino e a respectiva probabilidade. Caso a soma das probabilidades não seja igual a 1 no momento da simulação, os valores são automaticamente normalizados.

Além disso, é possível definir uma distribuição de prioridades, caso se deseje utilizá-las na simulação. A entrada adequada para esse parâmetro é um dicionário, em que as chaves representam as prioridades (inteiros positivos) e os valores correspondem à probabilidade de geração de jobs com aquela prioridade. Assim como no caso anterior, os valores são normalizados automaticamente, se necessário.

Código 6 – Método add_terminals_routing_probability da classe ClosedNetwork

```
def add_terminals_routing_probability(self, destination_server_id:
    int, probability: float):
    validate_number_params_not_negative_and_not_none(function_name='
        add_terminals_routing_probability', destination_server_id=
        destination_server_id, probability=probability)

    end_probability = self.entry_point_routing["end"]

    if probability > end_probability:
        raise ValueError("Too many probabilities, values exceeding 1
            ")

    if destination_server_id >= 0 and destination_server_id < len(
        self.servers):
        self.entry_point_routing["end"] -= probability
        self.entry_point_routing[destination_server_id] +=
            probability

        return

    raise ValueError("A server with the provided id was not found")
```

Por fim, a classe *Environment* também permite a adição de pontos de entrada externos de jobs, quando a rede é aberta. De forma análoga às chamadas da função *add_terminals_routing_probability*, uma tentativa de adicionar uma fonte externa de tarefas em uma rede configurada como fechada resultará em uma exceção. Essa validação, assim como a verificação dos tipos dos parâmetros, é realizada no próprio método da classe de ambiente, antes que a chamada à rede seja executada.

Fontes externas de jobs também podem, opcionalmente, possuir distribuições de prioridade, definidas da mesma forma que as utilizadas para os terminais.

Código 7 – Método add_entry_point da classe OpenNetwork

```
def add_entry_point(self, server_id: int, arrival_distribution:
    IDistribution, priority_distribution: Optional[dict] = None):
    validate_number_params_not_negative_and_not_none(function_name='
        add_entry_point', server_id=server_id)
    validate_object_params_not_none(function_name='add_entry_point',
        arrival_distribution=arrival_distribution)

    if server_id >= 0 and server_id < len(self.servers):
        self.arrivals[server_id].append(arrival_distribution)

        if priority_distribution:
            self.priorities[server_id] = priority_distribution

        return

    raise ValueError("The provided server id is not valid.")
```

Com apenas esses poucos métodos, torna-se possível configurar praticamente qualquer sistema de filas de forma simples, expressiva e com alta flexibilidade. Para configurar, por exemplo, o sistema M/M/1 descrito na figura 5, apenas três linhas de código são necessárias.

Código 8 – Definindo fila M/M/1

```
from qpy import Environment, Distribution, QueueDiscipline

env = Environment()
server_id = env.add_server(service_distribution=Distribution.exponential
    (mean_time_between_events=0.5), queue_discipline=QueueDiscipline.fcfs
    ())
env.add_entry_point(server_id=server_id, arrival_distribution=
    Distribution.exponential(mean_time_between_events=2))
```

4.3.2 Execução

Uma vez configurado o ambiente, o usuário pode executar a simulação para obter as métricas desejadas. O método *simulate*, pertencente à classe *Environment*, é responsável por iniciar o processo de simulação.

Durante a execução, a classe *Execution* é instanciada, recebendo como parâmetros o tempo total de simulação, o tempo de aquecimento (warm-up), as configurações da rede e uma fila de eventos, ordenada de forma crescente conforme o instante em que cada evento ocorre. Toda a lógica da simulação se baseia nesses elementos, representados

por objetos da classe *Event*. Esses objetos encapsulam informações essenciais sobre cada acontecimento ao longo da simulação.

O tempo de aquecimento é fundamental para evitar que as métricas sejam contaminadas pelo comportamento transiente do sistema. No início da simulação, o estado da fila tende a ser instável, ainda distante do regime estacionário. Assim, para que as medidas reflitam o desempenho real do sistema em operação, é necessário desconsiderar os dados coletados durante esse período inicial. Dessa forma, as métricas passam a ser registradas apenas após a estabilização da fila, garantindo resultados mais consistentes e representativos.

Código 9 – Classe Event

```
class Event:
    def __init__(self, current_time: float, event_id: int, event_type:
       str, job: Job, server_id: int):
        self.current_time = current_time
        self.id = event_id
        self.type = event_type
        self.job = job
        self.server_id = server_id
        self.server = None
        self.canceled = False
```

Cada evento é composto por sete atributos fundamentais: (i) o tempo decorrido desde o início da simulação até sua ocorrência; (ii) o identificador único do evento, atribuído de forma crescente; (iii) o tipo de evento, que pode ser uma chegada, uma partida ou uma preempção de um job em um dispositivo; (iv), (v) e (vi) os objetos envolvidos no evento, contendo informações sobre o job e o servidor correspondente; e (vii) um indicador que determina se o evento está ativo ou deve ser ignorado, aspecto essencial para o tratamento de preempções.

Antes da criação do objeto *Execution*, o método *simulate* solicita à rede uma lista inicial de eventos. Em redes abertas, todos os eventos de chegada de jobs são previamente gerados e adicionados à fila inicial. Já em redes fechadas, a saída de um job do sistema resulta na criação de um novo evento de chegada, após o período de tempo de pensamento definido.

A lista de eventos é, na prática, uma fila de prioridade, visto que é imprescindível respeitar a ordem cronológica dos acontecimentos para a consistência da simulação. Assim, optou-se pela implementação dessa fila como uma *heap* de mínimo, estrutura que permite a extração eficiente do próximo evento a ser processado.

Cada objeto presente na *heap* é composto por 3 atributos: Tempo associado à ação, identificador único do acontecimento em questão, e o objeto do tipo *Evento*, responsável por armazenar os dados daquele evento. É importante que o tempo seja o primeiro

atributo para que a ordenação seja consistente. Além disso, identificadores únicos se fazem necessários para casos onde hajam duas entradas com o mesmo tempo, configuração que pode acontecer caso muitos eventos sejam gerados.

Código 10 – Método simulate da classe Environment

```
@validate_call
def simulate(self, time_in_seconds: float, warmup_time: float) ->
    SimulationResults:
     queue = self._network.generate_jobs(time_limit=time_in_seconds +
         warmup_time)

     new_execution = Execution(time_in_seconds, warmup_time, queue,
        self._network, self._time_unit)

     return new_execution.execute()
```

De posse da fila de eventos, a simulação ocorre de forma discreta, por meio de um laço de repetição. A cada iteração, o evento com menor tempo é processado, podendo gerar novos eventos que são imediatamente inseridos na fila, preservando a ordem temporal. Esse ciclo se repete até que todos os eventos tenham sido processados ou que o tempo do próximo evento exceda o tempo total configurado para a simulação.

Código 11 – Método simulate da classe Execution

```
def execute(self) -> SimulationResults:
    end_time = self.warmup + self.time

    while len(self.event_queue) > 0:
        top_time = self.event_queue[0][0]

        if top_time > end_time:
            break

        next_event = heapq.heappop(self.event_queue)[2]

        if next_event.canceled:
            continue

        next_event.server = self.network_configuration.servers[
            next_event.server_id]

        self.current_time = next_event.current_time

        if next_event.type == 'arrival':
            self._case_event_is_arrival(next_event)
        else:
            self._case_event_is_departure_or_preemption(next_event)

    return self.results
```

Durante a execução, cada evento é processado de forma individual. Caso o evento não tenha sido cancelado e esteja dentro do tempo total estipulado para a simulação, considerando também o período de warm-up, o fluxo segue caminhos distintos conforme o tipo do evento: chegada ou partida/preempção.

No caso de um evento de chegada, o job tem seu estado atualizado para refletir a entrada em um novo dispositivo. Em seguida, ele é encaminhado ao servidor de destino, sendo passado como parâmetro para o método *job_ arrival* do objeto *Server* correspondente.

O servidor, então, gera uma amostra da distribuição de serviço, representando o tempo necessário para processar aquele job. Antes de iniciar o atendimento, o servidor verifica se já há alguma tarefa em execução.

Se não houver, o job atual inicia imediatamente sua execução, e o tempo de serviço calculado é retornado. Caso o servidor possua uma fila Round Robin (RR), o tempo de preempção deve ser checado, visto que após o tempo determinado deve haver a troca de tarefas. Dessa forma, caso o tempo de serviço seja superior ao de preempção, o evento gerado será a troca de tarefas.

O objeto *Execution*, por sua vez, adiciona à fila o evento correspondente à partida do job ou à preempção, cujo instante de ocorrência é igual ao tempo atual somado ao tempo de serviço.

Se já houver uma tarefa em execução, o servidor avalia a necessidade de preempção, que pode ocorrer caso o novo job possua maior prioridade e o usuário determine que a preempção deve ser realizada. Caso não seja necessária, o novo job e seu tempo de serviço são apenas inseridos na fila interna do servidor, e nenhum novo evento é gerado.

Quando ocorre uma preempção, o job em execução retorna à fila do servidor, acompanhado do tempo restante para sua conclusão, enquanto o novo job passa a ser processado. Nesse cenário, o tempo de serviço do novo job é retornado e um evento de partida é adicionado à fila de eventos globais.

Além disso, é necessário cancelar o evento de partida anterior, uma vez que a tarefa preemptada retornou à fila. Para isso, o objeto *Environment* mantém uma lista que registra o evento de partida ativo de cada servidor. Assim, sempre que um novo tempo de serviço é retornado e o servidor possui um evento ativo, entende-se que ocorreu uma preempção, e o evento anterior é marcado como cancelado. Essa estratégia é mais eficiente do que remover o evento da fila, preservando o desempenho da simulação.

Por fim, caso o período de warm-up já tenha transcorrido, as métricas associadas ao evento são atualizadas, garantindo que apenas dados representativos do estado estável do sistema sejam considerados.

Código 12 – Método _case_event_is_arrival da classe Execution

```
def _case_event_is_arrival(self, event: Event):
    event.job.reroute(self.current_time)
    service_time = event.server.job_arrival(event)

    if service_time:
        if self.next_departure_event_by_server[event.server_id]:
            self.next_departure_event_by_server[event.server_id].
                canceled = True

        self._add_next_departure_event(event.server_id, event.job, self.
            current_time, service_time, event_type='departure' if event.
            server.is_next_event_departure() else 'preemption')
    if event.job.arrival_time > self.warmup:
        self.results.compute_arrival(self.current_time, event.server_id)
```

Quando o evento não é de chegada, o tratamento necessário difere. O método *finish_execution* da classe *Server* é chamado, recebendo um parâmetro que indica o tipo de finalização: preempção ou término da execução.

No caso de término da execução, o servidor verifica se há jobs na fila aguardando processamento. Se houver, o job e seu tempo de execução são retornados, possibilitando

a inserção do evento de partida na fila de prioridades da execução. Caso contrário, o servidor permanece ocioso e nada é retornado.

Se o evento for de preempção e a fila não estiver vazia, um novo job será selecionado para execução. O job atual, por sua vez, é adicionado à fila do servidor juntamente com o tempo restante para sua conclusão. Caso a fila esteja vazia, o limite de tempo de execução da tarefa atual é renovado. Em todos os casos, a função retornará um job e o tempo necessário até o próximo evento.

O job retornado, caso exista, terá seu próximo evento adicionado à fila de prioridades, seja ele de preempção ou de término de execução.

Código 13 – Método _case_event_is_departure_or_preemption da classe Execution

```
def _case_event_is_departure_or_preemption(self, event: Event):
    new_job_being_executed = event.server.finish_execution(self.
        current_time, is_preemption=(event.type == 'preemption'))
    self.next_departure_event_by_server[event.server_id] = None

    if new_job_being_executed:
        new_job_service_time = new_job_being_executed[0]
        new_job = new_job_being_executed[1]

        self._add_next_departure_event(event.server_id, new_job, self.
            current_time, new_job_service_time, event_type='departure' if
             event.server.is_next_event_departure() else 'preemption')

    self._route_job_after_event(event)
```

Por fim, o job alvo do evento atual é roteado, caso aplicável. O método *route_job* da classe *Server* seleciona aleatoriamente um dos destinos de saída possíveis, conforme configurado pelo usuário.

O método *reroute* da classe *Job* assegura que todos os dados referentes à execução e ao tempo de espera no servidor anterior sejam corretamente registrados. Caso um novo servidor seja especificado, o estado é atualizado para refletir a mudança; caso contrário, apenas a coleta dos dados é realizada.

Se um novo servidor for o destino da tarefa, o procedimento de chegada é executado exatamente da mesma forma que no tratamento de um evento de chegada.

Por fim, as métricas são coletadas somente se o tempo atual da simulação for superior ao tempo de aquecimento. Todo o fluxo de execução da simulação pelo usuário pode ser observado no diagrama de sequência[2]. Além disso, um diagrama de atividades completo representando as etapas do processamento de um evento também foi construído[3].

[2] https://tinyurl.com/2226mzf5
[3] https://tinyurl.com/mmhuex49

Código 14 – Método _route_job_after_event da classe Execution

```
def _route_job_after_event(self, event: Event):
    route = event.server.route_job()

    if route != 'end':
        event.job.reroute(self.current_time, route)
        destination_server = self.network_configuration.servers[route]

        if event.job.arrival_time > self.warmup:
            self.results.reroute(self.current_time, event.server_id,
                route)
            self.results.compute_arrival(self.current_time, route)

        new_event = Event(self.current_time, self.event_count, 'arrival'
            , event.job, route)
        self.event_count += 1

        service_time = destination_server.job_arrival(new_event)

        if service_time:
            self._add_next_departure_event(route, new_event.job, self.
                current_time, service_time, event_type='departure' if
                destination_server.is_next_event_departure() else '
                preemption')
    else:
        event.job.reroute(self.current_time)
        if event.job.arrival_time > self.warmup:
            self.results.reroute(self.current_time, event.server_id,
                destination_server=None)
            self.results.compute_departure(event.job, self.current_time)
        self.network_configuration.finish_job(self.event_queue, self.
            current_time)
```

4.3.3 Testes

Com o objetivo de aumentar a confiabilidade do software e garantir o funcionamento esperado das funções individuais, diversos testes foram desenvolvidos. Foram explorados dois tipos principais: testes de unidade e testes de comportamento. Ao todo, 233 testes foram implementados com o auxílio da biblioteca *PyTest*.

Todos os métodos públicos foram devidamente avaliados por meio de testes unitários. Na linguagem Python, não há uma forma explícita de definir a visibilidade de um método. Assim, adotou-se o padrão amplamente reconhecido pela comunidade, que utiliza o caractere de sublinhado (_) no início dos nomes de métodos privados.

Os testes unitários aumentam a confiança na execução correta dos métodos desenvolvidos, conforme o comportamento esperado pelo programador. Além disso, permitem verificar o tratamento de exceções e avaliar a robustez do artefato. Para sua elaboração, foi utilizado o critério de cobertura *Each Choice*, conforme descrito em (AMMANN; OFFUTT, 2017). A adoção de um padrão consolidado contribui para maximizar a detecção de eventuais falhas, garantindo boa cobertura de testes com crescimento controlado do código.

Código 15 – Exemplo de testes unitários

```
"""
Particionamento do espaco de entrada para funcao add_server() da classe
    BaseNetwork utilizando Each Choice Coverage:
     queue_discipline: None | Valido
"""

"""queue_discipline = None (Valido)"""
def test_add_server_when_queue_discipline_is_none_should_use_fcfs(
    base_network_empty):
     server_id = base_network_empty.add_server(service_distribution=
         MOCK_DISTRIBUTION, queue_discipline=None)

     assert server_id == 0
     assert isinstance(base_network_empty.servers[server_id], Server)
     assert base_network_empty.servers[server_id].server_execution.queue.
         discipline == QueueDiscipline.fcfs().discipline


"""queue_discipline Valido (Valido)"""
def test_add_server_when_queue_discipline_is_provided_should_use_it(
    base_network_empty):
     custom_queue = QueueDiscipline.srt()
     server_id = base_network_empty.add_server(service_distribution=
         MOCK_DISTRIBUTION, queue_discipline=custom_queue)

     assert base_network_empty.servers[server_id].server_execution.queue.
         discipline == custom_queue.discipline
```

Os testes de comportamento, por sua vez, avaliam a execução completa de seções do código, assegurando que os resultados produzidos estejam em conformidade com o esperado. Dessa forma, fluxos completos de simulação foram verificados, garantindo que as métricas coletadas estejam próximas dos valores teóricos. Essa verificação é realizada considerando uma margem de erro de aproximadamente 5% para mais ou para menos.

Código 16 – Exemplo de teste de comportamento

```
def test_priority_preemption_when_second_job_has_higher_priority_should
_switch(priority_with_preemption_test_object, job_test_object_1,
    job_test_object_2, job_test_object_3):
     event = Event(TIME, EVENT_ID, 'arrival', job_test_object_1,
        SERVER_ID)
     second_event = Event(SECOND_TIME, EVENT_ID + 1, 'arrival',
        job_test_object_2, SERVER_ID)
     third_event = Event(THIRD_TIME, EVENT_ID + 2, 'arrival',
        job_test_object_3, SERVER_ID)

     priority_with_preemption_test_object.job_arrival(event=event)
     priority_with_preemption_test_object.job_arrival(event=second_event)
     priority_with_preemption_test_object.job_arrival(event=third_event)

     assert priority_with_preemption_test_object.server_execution.
        current_job_being_executed == job_test_object_2
     assert priority_with_preemption_test_object.server_execution.queue.
        first_in_line() == (DISTRIBUTION_TIME - (SECOND_TIME - TIME),
        job_test_object_1)
     assert priority_with_preemption_test_object.server_execution.queue.
        first_in_line() == (DISTRIBUTION_TIME, job_test_object_3)
```

4.3.4 Conclusão

A biblioteca de simulação, núcleo central deste projeto, foi desenvolvida com atenção às melhores práticas de engenharia de software. A qualidade do código é fundamental tanto para seu entendimento e uso por terceiros quanto para facilitar sua manutenção e expansão futura. Todo o fluxo interno foi projetado para reduzir a complexidade de uso pelo usuário final, expondo apenas o código essencial para o funcionamento da ferramenta.

Os testes desempenham papel crucial nesse contexto, aumentando significativamente a confiabilidade do produto final. O código completo, juntamente com o histórico de desenvolvimento e instruções de uso, encontra-se disponível no repositório GitHub do projeto[4].

[4] Disponível em: https://github.com/PedroMion/QPy

5 SOBRE A APLICAÇÃO WEB

Com o objetivo de democratizar o acesso às funcionalidades da biblioteca e facilitar seu uso por parte dos usuários, uma aplicação web foi desenvolvida, atuando como interface gráfica.

Para viabilizar a comunicação entre a interface e a biblioteca, foi construída uma API *REST* (FIELDING, 2000). Como há integração direta com a biblioteca, escrita em Python, a API foi desenvolvida na mesma linguagem, utilizando o framework *FastAPI*[1], amplamente reconhecido por seu desempenho e simplicidade.

5.1 APLICAÇÃO WEB

5.1.1 Interface gráfica

A interface adota um tema escuro e um design minimalista, priorizando a clareza e a experiência do usuário. Na parte superior da tela localiza-se a barra principal, que exibe o logotipo da aplicação e as opções para selecionar entre redes abertas e fechadas. Logo abaixo, encontra-se a área central de desenho, onde o usuário pode montar o sistema de filas de forma visual. Por fim, na parte inferior, uma barra de ferramentas permite adicionar dispositivos e iniciar a simulação.

O design adotado foi inspirado em ferramentas de desenho amplamente consolidadas, como *Miro* e *DrawIO*. A escolha por um modelo visual e interativo busca tornar o processo de modelagem mais intuitivo e acessível, aproximando-o do fluxo de trabalho utilizado em ferramentas de design de processos. Essa abordagem distingue o projeto de outras aplicações de simulação, que usualmente se restringem a configurações textuais.

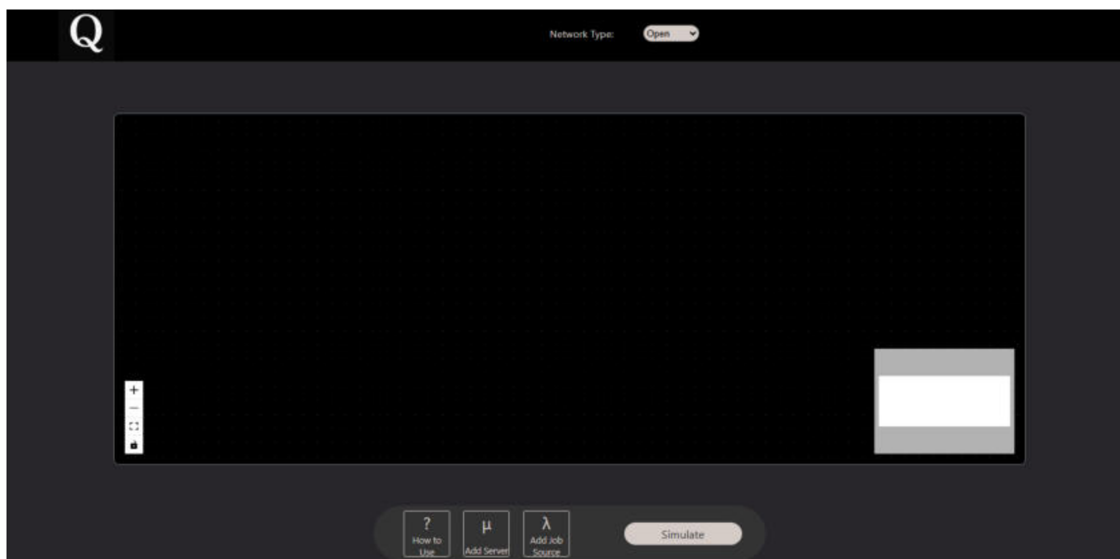
5.1.2 Configuração do ambiente

O primeiro passo para a configuração do ambiente consiste na seleção, na parte superior da interface, do tipo de rede desejado. Caso o usuário opte por uma rede fechada, um objeto representando os terminais é automaticamente inserido na área de desenho. Sempre que um novo tipo de ambiente é selecionado, toda a configuração previamente realizada é reiniciada.

Ao clicar em *Add Server* ou *Add Job Source*, é exibido um modal contendo as opções de personalização do dispositivo. As configurações são atualizadas automaticamente conforme o usuário preenche os campos, refletindo em tempo real a parametrização desejada. Na criação de um servidor, devem ser definidas a distribuição de serviço e as disciplinas de

[1] https://fastapi.tiangolo.com/

Figura 13 – Interface Gráfica



fila. Já na inclusão de fontes de job, é necessário especificar as distribuições dos tempos de chegada e das prioridades.

Diversas partes da interface foram estruturadas como componentes reutilizáveis, prática essencial no *React*, que favorece a organização modular do código e a combinação consistente de HTML, CSS e JavaScript.

Figura 14 – Modal para adicionar servidor

Após a adição dos dispositivos, as conexões podem ser estabelecidas arrastando-se o mouse entre os pontos específicos de cada objeto. A área de desenho foi implementada

com a biblioteca *react-flow*, que oferece suporte à criação de elementos visuais com estado interno, representados por nós e arestas que podem ser arrastados, modificados, conectados e removidos dinamicamente. Essa flexibilidade torna o processo de modelagem mais intuitivo e alinhado ao comportamento esperado de ferramentas visuais interativas.

O estado da aplicação é gerenciado por meio de *hooks*, estruturas nativas do *React* destinadas ao controle de estado e ao acompanhamento do ciclo de vida dos componentes. Cada *hook* encapsula variáveis e funções relacionadas a um contexto específico, fornecendo mecanismos centralizados e consistentes para leitura e atualização dos dados.

No total, foram desenvolvidos cinco *hooks* principais, responsáveis pelo gerenciamento do estado da área de desenho, das arestas, dos modais, bem como das configurações da rede e da simulação. A página principal atua como coordenadora de toda a lógica do sistema, distribuindo valores e funções aos componentes, garantindo que cada um execute corretamente suas responsabilidades.

Código 17 – Exemplo de hook - Arestas

```
export const useEdges = (onConnect) => {
    const [edgeProperties, setEdgeProperties] = useState({});

    const addEdge = (connection) => {
      onConnect(connection, null);
    }

    const originIsJobSource = (source) => {
      return source.startsWith('job');
    }

    const onAddEdge = (edgeData) => {
      if(originIsJobSource(edgeData.source)) {
        addEdge(edgeData);
      } else {
        setEdgeProperties(edgeData);

        document.getElementById("main-page-edge-properties-wrapper").
           style.display = 'flex';
        document.getElementById("main-page-nav-bar-container").style.
           display = 'none';
      }

  }

  return {
    edgeProperties,
    onAddEdge,
  };
};
```

5.1.3 Fluxo de simulação

Após configurar o sistema de filas, o usuário pode iniciar a simulação por meio do botão *Simulate*. Ao acioná-lo, um modal é exibido solicitando os parâmetros de execução, como o tempo total e o tempo de *warm-up*. Em redes fechadas, o número de terminais e a distribuição do tempo de pensamento também devem ser informados.

Concluída essa etapa, os dados fornecidos são encaminhados ao *hook useQueueSimulation*, responsável por orquestrar o processo de simulação. As informações são convertidas para o formato *JSON*, amplamente utilizado em interfaces *REST*, e enviadas à API. A resposta, igualmente em *JSON*, contém as métricas resultantes da simulação e é apresentada ao usuário após o processamento.

Código 18 – Modal Results

```
function ResultsModal({onModalClosed, simulationResults}) {
  return (
    <div className="results-modal-container">
      <div className='results-modal-header'>
        <div className='results-modal-header-text'>Simulation Results</
          div>
        <div className='results-modal-header-button' onClick={
          onModalClosed}>X</div>
      </div>

      <EnvironmentSection simulationResults={simulationResults} />

      <ServersSection simulationResults={simulationResults} />

      <PrioritySection simulationResults={simulationResults} />
    </div>
  );
}

export default ResultsModal
```

A interface gráfica é, portanto, um componente essencial para a disseminação e acessibilidade da ferramenta. Embora a biblioteca possa ser utilizada diretamente em Python, isso exigiria conhecimentos técnicos e a configuração de um ambiente apropriado. Com a aplicação web, o acesso torna-se simples, rápido e intuitivo, abrindo caminho para que usuários de diferentes perfis possam explorar o simulador.

O código completo da interface gráfica está disponível para visualização no repositório GitHub do projeto[2].

5.2 API

O terceiro artefato desenvolvido foi a API, responsável por conectar a interface gráfica à biblioteca de simulação. A aplicação foi construída com o framework *FastAPI*, que oferece suporte nativo à tipagem, documentação automática e validação de dados, características que garantem robustez e segurança à integração.

A API possui um único endpoint do tipo *POST*, responsável por receber os dados de configuração e retornar os resultados da simulação. As classes de entrada e saída foram implementadas utilizando a biblioteca *Pydantic*, que valida os dados recebidos e assegura que o formato de resposta esteja de acordo com o esperado.

[2] Disponível em: https://github.com/PedroMion/QPy-front

O endpoint principal recebe um objeto *JSON*, validado automaticamente pelo *Pydantic*. Em seguida, uma classe *Mapper* converte os dados para uma instância da classe *Environment* da biblioteca, configurada conforme os parâmetros fornecidos pelo usuário.

O fluxo principal executa a simulação e, após a conclusão, outra classe *Mapper* converte os resultados, instâncias da classe *SimulationResults*, para um formato *JSON* apropriado para resposta. Dessa forma, o ciclo de requisição é encerrado, entregando à interface um retorno pronto para exibição.

Código 19 – Endpoint POST simulate

```
@app.post("/simulate")
async def simulate(request: SimulationRequest) -> SimulationResponse:
    try:
        request_mapper = SimulationRequestMapper(request)

        env = request_mapper.get_environemnt()

        results = env.simulate(float(request.networkParameters.
            simulationTime), float(request.networkParameters.warmupTime))

        response_mapper = SimulationResponseMapper(results)

        return response_mapper.get_response_object()

    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```

5.3 CONCLUSÃO

Esta arquitetura, composta pela biblioteca, API e aplicação web, permite que o sistema seja facilmente expandido. A separação clara entre as camadas garante modularidade e facilita futuras melhorias, como autenticação de usuários, armazenamento de resultados ou integração com bancos de dados. Assim, a ferramenta não apenas oferece uma interface acessível, mas também estabelece uma base sólida para pesquisa e desenvolvimento em simulação de sistemas computacionais.

Visando possibilitar o acesso rápido via internet, foi realizado o *deploy* da interface gráfica através da ferramenta *Netlify*[3], podendo ser acessada através do link[4]. O *deploy* da API foi feito através da *Railway*[5].

[3] https://www.netlify.com/
[4] https://qpy-simulation-library.netlify.app/
[5] https://railway.com/

6 ANÁLISE DOS RESULTADOS

Este capítulo apresenta os experimentos realizados e a análise dos resultados obtidos, com o objetivo de compará-los aos valores teóricos esperados. Foram simulados dois sistemas distintos: a fila M/M/1 e o sistema fechado ilustrados no capítulo 3 (figuras 5 e 6).

Além da execução da fila M/M/1 adotando a distribuição dos tempos de chegada e serviço como exponenciais, as demais distribuições também serão testadas. Disciplinas de fila que diferem da ordem de chegada, no entanto, não serão testadas. Entretanto, todos os testes relacionados ao comportamento esperado de cada componente, incluindo a fila, foram realizados em código, conforme relatado no capítulo 4.

Os experimentos relacionados aos sistemas retratados nas figuras 5 e 6 foram executados tanto por meio da biblioteca desenvolvida quanto por meio da interface gráfica. Já os experimentos que visam validar as demais distribuições foram testados apenas na biblioteca, visando evitar repetição de dados.

Por fim, será realizada uma comparação com a biblioteca SimPy, amplamente utilizada e consolidada como uma ferramenta robusta para simulação de eventos discretos. Ao relacionar ambas as ferramentas, evidencia-se a capacidade de obtenção de resultados consistentes para cenários específicos, mesmo diante de uma redução significativa na complexidade de configuração do ambiente de execução da simulação e do processo de coleta de métricas. Para esse teste, foi utilizada a mesma fila M/M/1.

Em todas as simulações, o critério de parada adotado foi a quantidade total de tarefas processadas no sistema, com o objetivo de possibilitar uma comparação mais clara e direta com os valores teóricos esperados. Em razão da definição prévia de um número fixo de jobs como condição de encerramento da simulação, os valores teóricos utilizados na comparação foram aqueles correspondentes a um limite temporal equivalente a 200.000 segundos. Adicionalmente, todas as simulações consideraram um período de aquecimento (warm-up) de 10.000 segundos, a fim de reduzir os impactos decorrentes do regime transiente inicial.

6.1 FILA M/M/1

A simulação do sistema M/M/1 foi realizada considerando uma taxa de chegada de jobs, λ, igual a $0,5$ jobs por segundo, enquanto a taxa de serviço do servidor, μ, foi definida como 2 jobs por segundo.

Considerando o sistema aberto estável com as características descritas, espera-se, ao longo de um limite temporal de 200.000 segundos, a execução de aproximadamente 100.000

tarefas. Em razão disso, esse quantitativo foi adotado como critério de parada da simulação.

6.1.1 Simulação utilizando a biblioteca

A configuração de um sistema M/M/1 na biblioteca é direta, exigindo apenas a definição de seus parâmetros fundamentais. O Código 8 apresenta o trecho responsável por essa configuração. Os resultados obtidos são apresentados na Tabela 1.

Tabela 1 – Análise comparativa biblioteca - M/M/1

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	0.6666 seg	0.6679 seg	0.2%
$E[T_q]$	0.1666 seg	0.1665 seg	-0.06%
$E[N]$	0.3333 jobs	0.3332 jobs	-0.03%
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	25%	24.93%	-0.28%
X	0.5 jobs/seg	0.4988 jobs/seg	-0.24%
D	0.5 seg/job	0.5014 seg/job	0.28%

Observa-se que as métricas simuladas apresentam forte convergência para os valores teóricos, com margens de erro reduzidas. Quanto maior o tempo total de simulação, maior a estabilização e maior a precisão dos indicadores, conforme esperado em experimentos baseados em processos estocásticos. Não há necessidade de investigar as métricas do ambiente, já que todas serão compartilhadas com o único dispositivo.

A vazão igual à taxa de chegada, característica principal de sistemas equilibrados, pode ser observada nos resultados. O valor da demanda por tarefa confirma o esperado para os tempos de serviço no servidor criado.

Dessa forma, conclui-se que a biblioteca produz resultados consistentes e alinhados aos valores teóricos para o modelo M/M/1.

6.1.2 Simulação utilizando a interface gráfica

Assim como na biblioteca, a configuração da fila M/M/1 através da interface gráfica é extremamente simples.

Os resultados obtidos por meio da interface gráfica reproduzem o comportamento previsto teoricamente e mantêm coerência com a simulação realizada diretamente pela biblioteca. As pequenas diferenças observadas são resultado das particularidades de cada simulação, mantendo-se contidas em uma margem de erro previsível. A interface, por-

Figura 15 – Definição de fila M/M/1 através de interface gráfica

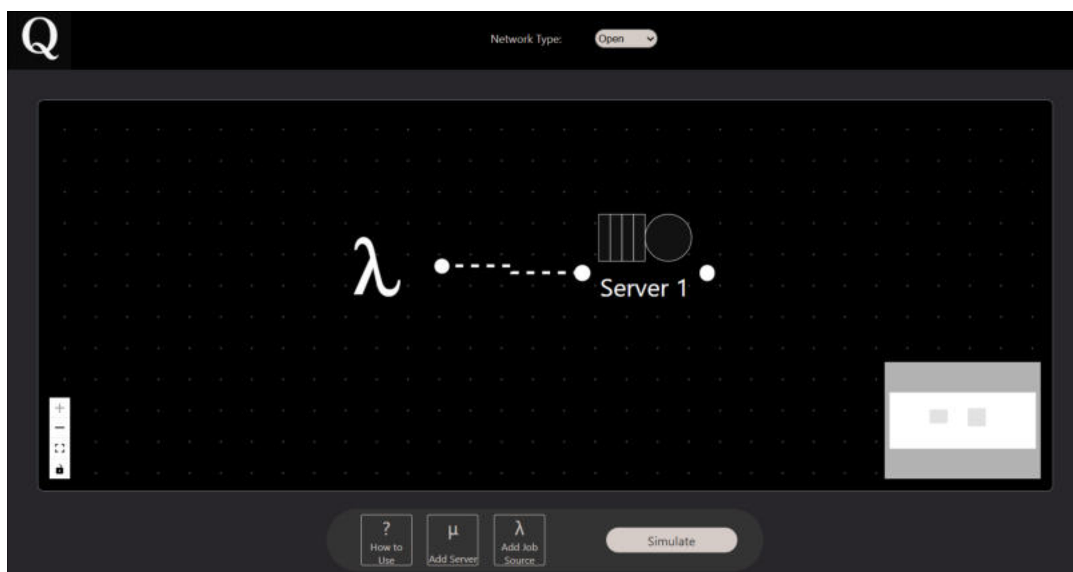


Tabela 2 – Análise comparativa interface gráfica - M/M/1

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	0.6666 seg	0.6656 seg	-0.15%
$E[T_q]$	0.1666 seg	0.1666 seg	0%
$E[N]$	0.3333 jobs	0.3338 jobs	0.15%
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	25%	25.02%	0.08%
X	0.5 jobs/seg	0.5015 jobs/seg	0.3%
D	0.5 seg/job	0.499 seg/job	-0.2%

tanto, cumpre adequadamente seu papel na configuração e execução de experimentos simples.

6.2 SISTEMA FECHADO

O sistema fechado simulado é composto por quatro terminais, todos sem tempo de pensamento, isto é, $E[Z] = 0$, e cinco servidores. Todos os jobs são inicialmente atendidos pelo servidor A, cuja taxa de serviço (μ) é igual a $0,125$ jobs por segundo.

Após o atendimento no servidor A, 25% dos jobs são encaminhados ao servidor B, que possui taxa de serviço igual a $0,1$ jobs por segundo. O restante é direcionado aos servidores C1 e C2, dispositivos idênticos, cada um com μ igual a $0,1667$ jobs por segundo.

Posteriormente, todos os jobs seguem para o servidor D. Nesse estágio, 25% dos jobs não recebem serviço, enquanto, dentre os que são atendidos, 20% podem retornar para uma nova execução. O servidor D possui taxa de serviço μ igual a $0,5$ jobs por segundo. O esquema completo das relações entre os dispositivos pode ser observado na Figura 6.

A vazão do sistema, calculada teoricamente no Capítulo 3, quando multiplicada pelo tempo total de simulação de 200.000 segundos, fornece a quantidade teórica de jobs a serem processados nesse intervalo. Em razão disso, adotou-se como critério de parada o processamento de 23.700 jobs no sistema.

6.2.1 Simulação utilizando a biblioteca

O sistema fechado apresenta maior complexidade estrutural, exigindo atenção à configuração dos roteamentos, capacidades e demandas. Apesar disso, a interface de definição de sistemas fornecida pela biblioteca simplifica o processo. O código 20 apresenta a implementação correspondente.

Código 20 – Definição do sistema fechado retratado na figura 6

```
from qpy import Environment, Distribution, QueueDiscipline

env = Environment(number_of_terminals=4, think_time_distribution=
    Distribution.constant(value=0))

a = env.add_server(service_distribution=Distribution.exponential(
    mean_time_between_events=8), queue_discipline=QueueDiscipline.fcfs())
b = env.add_server(service_distribution=Distribution.exponential(
    mean_time_between_events=10), queue_discipline=QueueDiscipline.fcfs()
    )
c1 = env.add_server(service_distribution=Distribution.exponential(
    mean_time_between_events=6), queue_discipline=QueueDiscipline.fcfs())
c2 = env.add_server(service_distribution=Distribution.exponential(
    mean_time_between_events=6), queue_discipline=QueueDiscipline.fcfs())
d = env.add_server(service_distribution=Distribution.exponential(
    mean_time_between_events=2), queue_discipline=QueueDiscipline.fcfs())

env.add_terminals_routing_probability(destination_server_id=a,
    probability=1.0)

env.add_servers_connection(origin_server_id=a, destination_server_id=b,
    routing_probability=0.25)
env.add_servers_connection(a, c1, 0.375)
env.add_servers_connection(a, c2, 0.375)

env.add_servers_connection(b, d, 0.75)
env.add_servers_connection(c1, d, 0.75)
env.add_servers_connection(c2, d, 0.75)

env.add_servers_connection(d, d, 0.2)
```

6.2.1.1 Ambiente

A análise dos resultados evidencia que as métricas coletadas apresentam boa concordância com os valores teóricos previamente calculados. A convergência observada indica que a biblioteca é adequada para simular sistemas fechados de maior complexidade.

O valor médio de 4 jobs no sistema reflete a modelagem esperada, e a demanda máxima também reflete o valor teórico esperado. As métricas por dispositivo seguem apresentadas nas subseções seguintes.

Tabela 3 – Análise comparativa ambiente - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	33.75 seg	33.82 seg	0.21%
$E[T_q]$	17 seg	16.99 seg	-0.06%
$E[N]$	4.0 jobs	4.0 jobs	0%
X	0.1185 jobs/seg	0.1183 jobs/seg	-0.17%
D_{max}	8.0 seg/job	7.911 seg/job	-1.11%

6.2.1.2 Servidor A

Tabela 4 – Análise comparativa servidor A - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	23.700	23.700	0%
$E[T]$	-	22.13 seg	-
$E[T_q]$	-	14.21 seg	-
$E[N]$	-	2.62 jobs	-
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	94.8%	93.56%	-1.30%
X	0.1185 jobs/seg	0.1183 jobs/seg	-0.17%
D	8 seg/job	7.911 seg/job	-1.11%

A utilização próxima de 100% confirma que o Servidor A constitui o gargalo do sistema, conforme determinado pela análise teórica. Além disso, a demanda de serviço observada na simulação apresentou forte proximidade com o valor calculado. Os tempos de serviço e de fila, assim como a quantidade esperada de jobs no sistema, são mais complexos de serem calculados para servidores específicos em sistemas fechados. Portanto, o ferramental teórico explorado nesse trabalho não possibilita a comparação com os resultados analíticos para as métricas citadas.

6.2.1.3 Servidor B

É possível comparar o tempo médio de espera em fila teórico com o valor observado na simulação para o servidor B, uma vez que a análise teórica apresentada no Capítulo 3 fornece o valor esperado do número de jobs na fila desse dispositivo. Contudo, na ausência

Tabela 5 – Análise comparativa servidor B - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	5.925	5.801	-2%
$E[T]$	-	13.67 seg	-
$E[T_q]$	0.84 seg	0.89 seg	5.9%
$E[N]$	-	0.4 jobs	-
$E[V]$	0.25 visitas/job	0.2447 visitas/job	-2.12%
Utilização	28.44%	29.09%	2.28%
X	0.0303 jobs/seg	0.0289 jobs/seg	-4.6%
D	2.40 seg/job	2.46 seg/job	2.5%

do valor esperado do tempo de processamento de um job, torna-se inviável a obtenção das demais métricas necessárias para uma comparação mais abrangente.

6.2.1.4 Servidores C

Tabela 6 – Análise comparativa servidor C1 - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	8.887	8.935	0.54%
$E[T]$	-	8.0035 seg	-
$E[T_q]$	-	0.7331 seg	-
$E[N]$	-	0.3576 jobs	-
$E[V]$	0.375 visitas/job	0.377 visitas/job	0.5%
Utilização	26.66%	27.08%	1.57%
X	0.0444 jobs/seg	0.0447 jobs/seg	0.67%
D	2.25 seg/job	2.29 seg/job	1.78%

Os dois servidores C apresentaram resultados similares, conforme esperado devido à simetria do modelo.

6.2.1.5 Servidor D

Tabela 7 – Análise comparativa servidor D - Sistema fechado

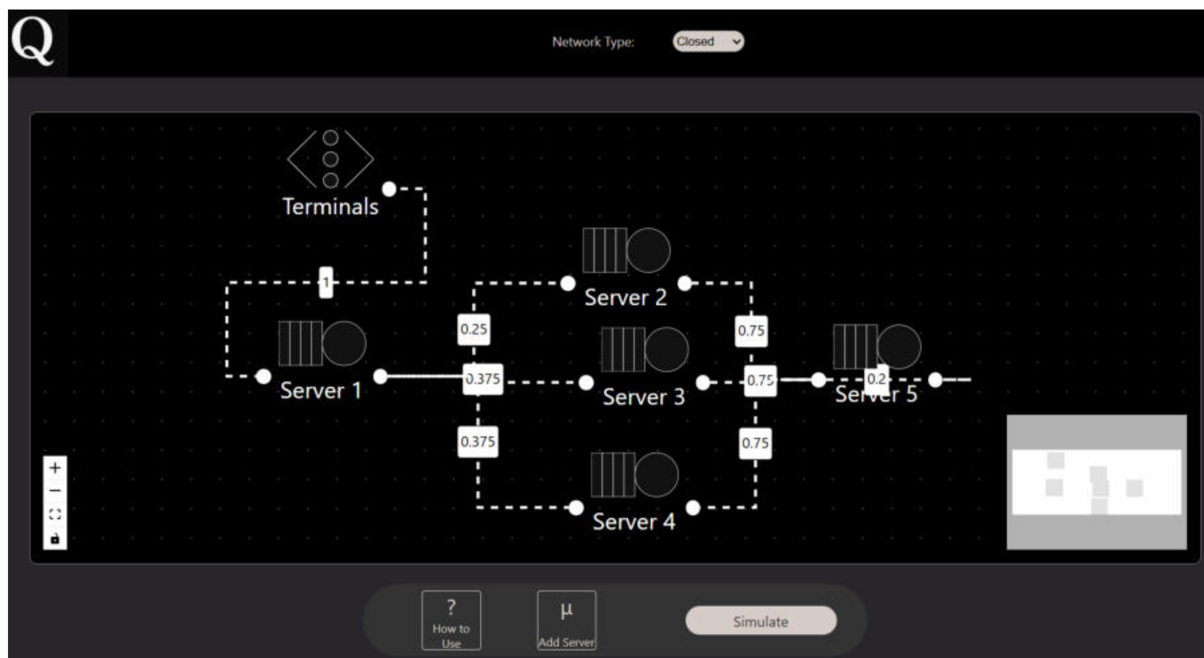
MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	22.218	22.202	-0.07%
$E[T]$	-	2.503 seg	-
$E[T_q]$	-	0.47 seg	-
$E[N]$	-	0.278 jobs	-
$E[V]$	0.9375 visitas/job	0.9387 visitas/job	0.13%
Utilização	21.99%	22.22%	1.05%
X	0.1111 jobs/seg	0.1111 jobs/seg	0%
D	1.8562 seg/job	1.8791 seg/job	1.23%

O número médio de visitas ao Servidor D, calculado com base na modelagem utilizando a distribuição geométrica, também apresentou forte concordância com o valor obtido na simulação. A baixa utilização observada nos servidores em geral, com exceção do Servidor A, evidencia a potencial melhora do sistema como um todo em caso de redução da demanda no dispositivo de gargalo.

6.2.2 Simulação utilizando a interface gráfica

Já na interface gráfica, a configuração é facilitada. A possibilidade de desenhar o sistema de forma simples e interativa torna todo o processo de configuração mais agradável.

Figura 16 – Definição de sistema fechado através de interface gráfica



Os resultados obtidos por meio da interface gráfica reproduziram adequadamente o comportamento do sistema fechado, demonstrando que a ferramenta permite configurar e executar cenários complexos com precisão semelhante à obtida diretamente via código.

6.2.2.1 Ambiente

Tabela 8 – Análise comparativa ambiente via interface gráfica - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	33.75 seg	33.755 seg	0.01%
$E[T_q]$	17 seg	16.89 seg	-0.65%
$E[N]$	4.0 jobs	4.0 jobs	0%
X	0.1185 jobs/seg	0.1185 jobs/seg	0%
D_{max}	8.0 seg/job	7.91 seg/job	-1.12%

6.2.2.2 Servidor A

Tabela 9 – Análise comparativa servidor A via interface - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	23.700	23.700	0%
$E[T]$	-	22.002 seg	-
$E[T_q]$	-	14.09 seg	-
$E[N]$	-	2.61 jobs	-
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	94.8%	93.7%	-1.16%
X	0.1185 jobs/seg	0.1185 jobs/seg	0%
D	8 seg/job	7.91 seg/job	-1.12%

6.2.2.3 Servidor B

Tabela 10 – Análise comparativa servidor B via interface - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	5.925	5.899	-0.44%
$E[T]$	-	13.715 seg	-
$E[T_q]$	0.84 seg	0.87 seg	3.57%
$E[N]$	-	0.4 jobs	-
$E[V]$	0.25 visitas/job	0.249 visitas/job	-0.4%
Utilização	28.44%	29.55%	3.9%
X	0.0303 jobs/seg	0.0295 jobs/seg	-2.64%
D	2.40 seg/job	2.54 seg/job	5.8%

6.2.2.4 Servidores C

Tabela 11 – Análise comparativa servidor C1 via interface - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	8.887	8.856	-0.35%
$E[T]$	-	7.96 seg	-
$E[T_q]$	-	0.72 seg	-
$E[N]$	-	0.352 jobs	-
$E[V]$	0.375 visitas/job	0.374 visitas/job	-0.27%
Utilização	26.66%	26.76%	0.37%
X	0.0444 jobs/seg	0.0443 jobs/seg	-0.22%
D	2.25 seg/job	2.2592 seg/job	0.41%

6.2.2.5 Servidor D

Tabela 12 – Análise comparativa servidor D via interface - Sistema fechado

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
Jobs processados	22.218	22.218	0%
$E[T]$	-	2.5 seg	-
$E[T_q]$	-	0.472 seg	-
$E[N]$	-	0.2783 jobs	-
$E[V]$	0.9375 visitas/job	0.9377 visitas/job	0.02%
Utilização	21.99%	22.15%	0.73%
X	0.1111 jobs/seg	0.1111 jobs/seg	0%
D	1.8562 seg/job	1.8778 seg/job	1.16%

6.3 DEMAIS DISTRIBUIÇÕES

6.3.1 Constante

Na distribuição constante, a mesma amostra é retornada em todas as gerações. Assim, a implementação não depende de aleatoriedade e os resultados tornam-se completamente previsíveis. A configuração adotada segue o padrão utilizado na simulação M/M/1: intervalos de chegada de 2 segundos e tempos de serviço de $0,5$ segundos. Esse cenário implica no mesmo limite de jobs processados empregado na fila M/M/1, isto é, 100.000 tarefas. Os dados coletados na simulação são apresentados na tabela 13.

Tabela 13 – Análise comparativa para distribuição constante - M/M/1

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	0.5 seg	0.5 seg	0%
$E[T_q]$	0.0 seg	0.0 seg	0%
$E[N]$	0.25 jobs	0.25 jobs	0%
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	25%	25%	0%
X	0.5 jobs/seg	0.5 jobs/seg	0%
D	0.5 seg/job	0.5 seg/job	0%

Os resultados são os esperados, ou seja, valores determinísticos e sem variação.

6.3.2 Uniforme

A distribuição uniforme gera valores equiprováveis dentro de um intervalo $[a,b]$. Para defini-la é necessário informar os limites a e b. O valor esperado da distribuição é a média aritmética entre esses limites.

Para manter as médias utilizadas nas demais simulações, definiram-se os limites do tempo de serviço em $0,1$ e $0,9$ segundo por job, enquanto o intervalo de chegada foi fixado entre 1 e 3 segundos por job. Por manter a média nos intervalos de chegada, os mesmos 100.000 jobs são esperados. Os resultados obtidos são exibidos na tabela 14.

Tabela 14 – Análise comparativa para distribuição uniforme - M/M/1

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	0.5 seg	0.5007 seg	0.14%
$E[T_q]$	0.0 seg	0.0 seg	0%
$E[N]$	0.25 jobs	0.2503 jobs	0.12%
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	25%	25.03%	0.12%
X	0.5 jobs/seg	0.4999 jobs/seg	-0.02%
D	0.5 seg/job	0.5007 seg/job	0.14%

Observa-se que o tempo médio de serviço próximo a $0,5$ segundos é compatível com o valor teórico para o servidor configurado. Além disso, o tempo total em filas é nulo: como

o intervalo mínimo entre chegadas é de um segundo e o máximo do tempo de serviço é inferior a esse valor, não ocorre formação de fila nesta configuração.

A utilização e a vazão seguem os valores teóricos esperados, aproximando-se dos resultados obtidos para as demais distribuições testadas.

6.3.3 Normal

A distribuição normal requer dois parâmetros: *mu* (média) e *sigma* (desvio padrão). O parâmetro *mu* determina o ponto central da distribuição, enquanto *sigma* controla a dispersão em torno dessa média. Devido à possibilidade de amostras negativas, que não são plausíveis para tempos, o sistema substitui valores negativos por um limite mínimo (um milésimo de segundo) quando necessário.

Para este teste médias maiores foram escolhidas, mantendo a razão 4:1 entre os intervalos de chegada e os tempos de serviço usada nos demais testes. Para o servidor, foram escolhidos os valores 2 para *mu* e 2 para *sigma*. Já para a distribuição de tempo de serviço, os valores utilizados foram 8 e 2 para *mu* e *sigma*, respectivamente. Em decorrência do aumento nos intervalos de chegada, a quantidade de jobs processados foi reduzida na mesma proporção, estabelecendo-se como critério de parada o processamento de 25.000 jobs. As métricas observadas podem ser observadas na tabela 15.

Tabela 15 – Análise comparativa para distribuição normal - M/M/1

MÉTRICA	TEÓRICO	SIMULAÇÃO	VARIAÇÃO PERCENTUAL
$E[T]$	2 seg	2.1856 seg	9.28%
$E[T_q]$	0.0 seg	0.019 seg	-
$E[N]$	0.25 jobs	0.2731 jobs	9.24%
$E[V]$	1.0 visitas/job	1.0 visitas/job	0%
Utilização	25%	27.07%	8.3%
X	0.1250 jobs/seg	0.1250 jobs/seg	0%
D	2 seg/job	2.1661 seg/job	8.3%

As métricas escalaram conforme esperado em função do aumento das médias, com a vazão diminuindo aproximadamente em um fator 4, enquanto os tempos médios de serviço aumentaram de acordo com a nova expectativa da distribuição.

As demais métricas mantiveram-se próximas aos valores obtidos anteriormente. Entretanto, em virtude do maior desvio padrão utilizado e da natureza da distribuição normal, observam-se flutuações ligeiramente maiores nas medidas amostrais em comparação às outras distribuições.

6.4 COMPARAÇÃO COM SIMPY

Por fim, a comparação entre os resultados obtidos por meio das simulações realizadas com a biblioteca desenvolvida neste trabalho e aqueles provenientes de uma ferramenta consolidada e amplamente utilizada contribui para fortalecer a tese de corretude do artefato proposto.

A simulação conduzida no SimPy empregou os mesmos parâmetros adotados na configuração M/M/1 implementada por meio da biblioteca apresentada neste capítulo, cujos resultados podem ser consultados na Tabela 1. Assim, consideraram-se $\lambda = 0,5$ jobs por segundo, $\mu = 2$ jobs por segundo e o critério de parada fixado em 100.000 jobs processados.

Em razão da natureza generalista da ferramenta externa de simulação de eventos discretos, a configuração do ambiente mostra-se menos direta e intuitiva quando comparada à interface de usuário da biblioteca QPy. O código utilizado para a realização da simulação por meio da biblioteca SimPy encontra-se disponível[1].

Tabela 16 – Análise comparativa biblioteca e SimPy - M/M/1

MÉTRICA	SIMPY	QPY	DIFERENÇA ABSOLUTA
Jobs processados	100.000	100.000	0
$E[T]$	0.6677 seg	0.6679 seg	0.0002 seg
$E[T_q]$	0.1658 seg	0.1665 seg	0.0007 seg
$E[N]$	0.3328 jobs	0.3332 jobs	0.0004 jobs
$E[V]$	1.0 visitas/job	1.0 visitas/job	0 visitas/job
Utilização	24.93%	24.93%	0
X	0.4993 jobs/seg	0.4988 jobs/seg	0.0005 jobs/seg
D	0.5019 seg/job	0.5014 seg/job	0.0005 seg/job

Uma vez que não há um valor esperado de referência, mas sim uma análise comparativa, o uso do erro absoluto mostra-se mais apropriado do que o erro percentual. Observa-se, portanto, uma elevada proximidade entre os dados obtidos a partir de ambas as ferramentas, o que evidencia a qualidade da implementação do artefato desenvolvido neste trabalho.

6.5 CONCLUSÃO

Com a análise dos resultados, evidencia-se um forte indício de corretude da ferramenta, assim como sua capacidade de realização de simulações de redes de filas complexas. Dessa

[1] https://tinyurl.com/yuwse7mh

forma, usuários são capazes de testar o comportamento de determinadas configurações de dispositivos em sistemas de filas e possuem total liberdade para a realização destes testes.

7 CONCLUSÃO

7.1 APRENDIZADO COM O TRABALHO

A construção deste trabalho proporcionou uma experiência abrangente que uniu teoria e prática de forma consistente. Diversos conceitos de teoria de filas, estudados ao longo do curso de Ciência da Computação, foram aplicados na concepção e desenvolvimento da biblioteca de simulação. Contudo, o avanço do projeto exigiu também o aprofundamento em temas complementares e o estudo de novas áreas do conhecimento, ampliando significativamente minha base técnica.

O desenvolvimento da biblioteca resultou em um aprendizado sólido na linguagem *Python*, com especial ênfase em boas práticas de programação e nos princípios de orientação a objetos. Além disso, a necessidade de planejar uma arquitetura modular, escalável e testável reforçou a compreensão sobre o ciclo de desenvolvimento de softwares robustos e a importância da engenharia de software na prática.

A criação da aplicação web e da API consolidou ainda mais essa experiência, ao integrar diferentes camadas tecnológicas em um ecossistema coerente e funcional. Com isso, foi possível desenvolver uma visão sistêmica sobre o processo de construção de soluções completas, que vão desde o núcleo lógico de uma biblioteca até a disponibilização de suas funcionalidades em uma interface amigável ao usuário.

Por fim, o processo de elaboração do trabalho, desde o levantamento teórico até a escrita e documentação, proporcionou um grande amadurecimento acadêmico. A prática da pesquisa científica, o rigor metodológico e a produção de um texto técnico bem estruturado contribuíram para o fortalecimento das competências necessárias à atuação profissional e à continuidade dos estudos na área.

7.2 CONSIDERAÇÕES PARA O FUTURO

O projeto desenvolvido representa uma base sólida para futuras expansões, tanto no âmbito técnico quanto acadêmico. Em versões futuras, a biblioteca poderá ser aprimorada com a inclusão de novos recursos e parâmetros de personalização. A implementação de filas com capacidade finita, por exemplo, ampliaria a aplicabilidade do simulador em cenários reais, nos quais os sistemas possuem limitações físicas. Da mesma forma, a adição de novas distribuições de chegada e de serviço, assim como de diferentes disciplinas de atendimento, permitiria simular um conjunto ainda maior de comportamentos de sistemas.

Na aplicação web, a adição de funcionalidades como o salvamento das configurações de rede, em nuvem ou em arquivos locais, evitaria a repetição de tarefas em execuções futuras, aumentando a praticidade e a atratividade da ferramenta. Outra evolução pro-

missora seria a implementação de uma visualização em tempo real da simulação, exibindo dinamicamente o estado das filas, servidores e conexões, o que traria grande valor didático ao projeto e facilitaria o aprendizado de novos usuários.

Em um contexto mais amplo, o aprofundamento em aspectos teóricos também se apresenta como um caminho natural para trabalhos futuros. Investigações mais detalhadas sobre a modelagem matemática dos sistemas simulados, ou sobre a aplicação de métodos estatísticos avançados para análise dos resultados, poderiam complementar o caráter prático desta pesquisa e contribuir para o avanço do campo da simulação de sistemas computacionais.

REFERÊNCIAS

ADAN, I.; RESING, J. Queueing theory. **Eindhoven University of Technology**, Eindhoven, v. 180, 2002.

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2017.

DENNING, P. J.; BUZEN, J. P. The operational analysis of queueing network models. **ACM Computing Surveys (CSUR)**, Acm New York, NY, USA, v. 10, n. 3, p. 225–261, 1978.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine, 2000.

GAMMA, E. et al. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Deutschland GmbH, 1995.

HARCHOL-BALTER, M. **Performance modeling and design of computer systems: queueing theory in action**. [S.l.]: Cambridge University Press, 2013.

KAMALI, S. H. et al. The monitoring of the network traffic based on queuing theory and simulation in heterogeneous network environment. In: IEEE. **2009 International Conference on Computer Technology and Development**. [S.l.], 2009. v. 1, p. 322–326.

KENDALL, D. G. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. **The Annals of Mathematical Statistics**, JSTOR, p. 338–354, 1953.

LIN, J.-S.; HUANG, C.-Y. Queueing-based simulation for software reliability analysis. **IEEE Access**, IEEE, v. 10, p. 107729–107747, 2022.

LITTLE, J. D.; GRAVES, S. C. Little's law. In: **Building intuition: insights from basic operations management models and principles**. [S.l.]: Springer, 2008. p. 81–100.

MARIA, A. Introduction to modeling and simulation. In: **Proceedings of the 29th conference on Winter simulation**. [S.l.: s.n.], 1997. p. 7–13.

MARTIN, R. C. Design principles and design patterns. **Object Mentor**, v. 1, n. 34, p. 597, 2000.

SZTRIK, J. Queueing theory and its applications, a personal view. In: **Proceedings of the 8th international conference on applied informatics**. [S.l.: s.n.], 2010. v. 1, p. 9–30.

ZINOVIEV, D. Discrete event simulation: It's easy with simpy! **arXiv preprint arXiv:2405.01562**, 2024.