UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Gustavo Kenji Kuroda de Oliveira

Extensão funcional para simulação de redes de fila

São José do Rio Preto
2023

Gustavo Kenji Kuroda de Oliveira

Extensão funcional para simulação de redes de fila

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para a aprovação na disciplina Trabalho de Conclusão de Curso.

Orientadora:

Profa. Dra. Renata Spolon Lobato

São José do Rio Preto
2023

O48e

Oliveira, Gustavo Kenji Kuroda de
Extensão funcional para simulação de redes de fila / Gustavo Kenji Kuroda de Oliveira. -- São José do Rio Preto, 2022
53 p. : il., tabs.

Trabalho de conclusão de curso (Bacharelado - Ciência da Computação) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto
Orientadora: Renata Spolon Lobato

1. Simulação de eventos discretos. 2. Python. 3. Extensão funcional. 4. Redes de fila. 5. SMPL. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Gustavo Kenji Kuroda de Oliveira

Extensão funcional para simulação de redes de fila

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para a aprovação na disciplina Trabalho de Conclusão de Curso.

Banca Avaliadora:

Profa. Dra. Renata Spolon Lobato
UNESP – Câmpus de São José do Rio Preto
Orientadora

Prof. Dr. Aleardo Manacero Junior
UNESP – Câmpus de São José do Rio Preto

Prof. Dr. Geraldo Francisco Donegá Zafalon
UNESP – Câmpus de São José do Rio Preto

São José do Rio Preto
11 de Janeiro de 2023

À minha família e amigos.

"...enquanto você estiver progredindo, isso é mais, do que perfeito."

- Paulo Muzy

Agradecimentos

Agradeço primeiramente e especialmente à minha mãe por todo o apoio durante toda a graduação e antes dela.

Agradeço também à minha orientadora Profa. Dra. Renata Spolon Lobato, que aceitou me orientar e teve toda a paciência e disponibilidade para a conclusão do trabalho.

Agradeço todos os professores que fizeram parte da minha graduação, por passar conhecimentos técnicos, mas além disso, ensinamentos de vida

Resumo

O conceito de fila está presente no cotidiano das pessoas seja em supermercados, bancos ou parques de diversão. O estudo de redes de filas tem como objetivo otimizar os modelos a ser implementados ou os já implementados, reduzindo o tempo de espera e possíveis perdas financeiras por parte das empresas. Nesse sentido, a simulação de eventos discretos tem papel fundamental em complementar o estudo de modelos de fila, podendo modelar e simular, com um certo grau de fidelidade, cenários da vida real. Assim, esse trabalho propõe o desenvolvimento de uma extensão funcional (biblioteca) em Python para a simulação de eventos discretos, capaz de reproduzir e realizar estudos de desempenho de diferentes modelos presentes na literatura, em diferentes sistemas computacionais. Para validação da solução obtida, utilizou-se a extensão funcional SMPL, originalmente desenvolvida na linguagem C por Myron H. MacDougall. Analisando e comparando os resultados das soluções, chegou-se a conclusão de que a extensão funcional desenvolvida nesse trabalho foi capaz de reproduzir os mesmos resultados de uma implementação consolidada, porém utilizando uma linguagem de programação mais atual, de fácil aprendizado e aplicável.

Palavras-chave: Simulação de eventos discretos, Python, SMPL, Extensão funcional, Filas, Redes de filas.

Abstract

The concept of queue is present in everyday life of people, whether in supermarkets, banks or amusement parks. The study of queue networks has the objective to optimize models to be implemented or those already implemented, decreasing the waiting time and possible financial losses by companies. In that regard, the discrete event simulation has a fundamental role to complement the study of queue models, being able to model and simulate, with a certain degree of fidelity, real life scenarios. Therefore, this work propose the development of a functional extension (library) in Python for discrete event simulations, capable of reproduce and perform performance studies of different models available in the literature, in different computational systems. To validade the reached solution, was used the SMPL functional extension, originally developed in C language by Myron H. MacDougall. Analyzing and comparing the solutions results, the reached conclusion was that the developed functional extension in this work was able to reproduce the same results of a consolidated implementation, however utilizing a modern, easy to learn and applicable programming language.

Keywords: Discrete event simulation, Python, SMPL, Functional extension, Queues, Network queues.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O conceito de fila está presente no cotidiano de todos, seja em atendimentos em bancos, supermercados ou parques de diversão, é necessário esperar por um serviço. Contudo, busca-se otimizar esse processo, devido à aleatoriedade envolvida, afim de evitar longos períodos de espera, perdas financeiras por parte das empresas devido a desistências, entre outros fatores.

Segundo Magalhães (MAGALHÃES, 1996), o estudo dos modelos de filas tem como objetivo melhorar o desempenho de sistemas de filas em diferentes aspectos, como gerenciamento de recursos, menor tempo de espera, rapidez e eficiência no atendimento. O pioneiro à estudar os modelos de filas foi A. K. Erlang na congestão de linha telefônicas, na época em que trabalhava como engenheiro da companhia dinamarquesa de telefonia.

Para facilitar o processo de otimização, pode ser utilizado a simulação ao invés de implementar e testar os modelos na prática, o que pode ser custoso e incerto. Nesse contexto, os ambientes de simulação de eventos discretos são muito importantes, pois permitem a avaliação de sistemas sem interferir no sistema real. A partir de vários testes e análises de desempenho pode-se auxiliar em tomadas de decisão para processos operacionais e gerenciais do sistema, e assim implementá-los de fato.

Este trabalho foi desenvolvido na linguagem de programação Python, por ser uma

linguagem fácil de aprendizado e de sintaxe simplificada, o que a torna muito popular. Toda essa popularidade gera uma grande quantidade de materiais online para aprendizado, além de diversas bibliotecas *open source* disponibilizadas por outros desenvolvedores.

1.1 Objetivos

O objetivo deste trabalho é criar uma extensão funcional (biblioteca) para a simulação de eventos discretos para a linguagem Python e fazer um estudo de desempenho, com resolução de modelos através da simulação, em diferentes sistemas computacionais. Esses modelos poderão ser obtidos por meio da literatura. A extensão deve ser capaz de permitir a modelagem de diferentes modelos de sistemas de filas e oferecer relatórios de desempenho.

1.2 Motivação

Dada a importância da otimização de sistemas de filas e o papel fundamental que a simulação de eventos discretos desempenha nesse processo, surge a oportunidade para o desenvolvimento de uma biblioteca para a linguagem Python que implemente funções que auxiliem no desenvolvimento de modelos de sistemas de filas. O fato de a linguagem Python ser uma linguagem emergente e de fácil aprendizado simplifica a modelagem e a simulação de sistemas de redes de fila, além de tornar a compreensão dos dados, provindos da simulação, simples.

1.3 Justificativa

O processo de modelagem de sistemas reais quando combinado com a simulação de eventos discretos, permite a abstração desses sistemas para a realização de ajustes.

Por meio da aplicação de testes e análises dos resultados, o processo de tomada de decisão pode ser realizado de maneira mais segura e confiável. Portanto o processo de simulação possui extrema importância no processo avaliativo e de tomada de decisão para alteração de sistemas reais.

1.4 Metodologia

A metodologia adotada para o desenvolvimento do trabalho consiste na etapa de pesquisa e embasamento teórico, abordando temas como redes de filas, simulação, simulação de eventos discretos, modelagem, SMPL e Python. A etapa seguinte é a implementação da extensão funcional e a modelagem de diferentes modelos presentes na literatura. A etapa final é a realização da avaliação e validação do modelo. O material de pesquisa e embasamento teórico tem origem em principalmente em livros, monografias e artigos científicos, além de documentações online de acesso gratuito.

1.5 Exequibilidade

A realização do trabalho utilizou os próprios recursos de *hardware* do autor, bem como a linguagem de programação Python como citado anteriormente. Toda a documentação da linguagem Python, como a grande maioria dos livros e artigos científicos podem ser acessados de forma gratuita e online ou através da biblioteca. Portanto o trabalho pode ser desenvolvido no prazo estipulado sem o uso de recursos adicionais.

1.6 Organização do trabalho

No capítulo 2 é realizada a revisão bibliográfica, abordando conceitos das áreas de redes de filas, simulação de eventos, modelagem de sistemas e da extensão funcional SMPL para a linguagem C. No capítulo 3 é descrito o desenvolvimento das

funções que compõe a extensão funcional e do modelo mais simples de sistema de filas. No capítulo 4 são apresentados os testes aplicados sobre modelos de sistemas computacionais (os quais serão obtidos da literatura). No capítulo 5 são apresentadas as conclusões do trabalho desenvolvido, assim como propostas de trabalhos futuros.

Capítulo 2

Revisão bibliográfica

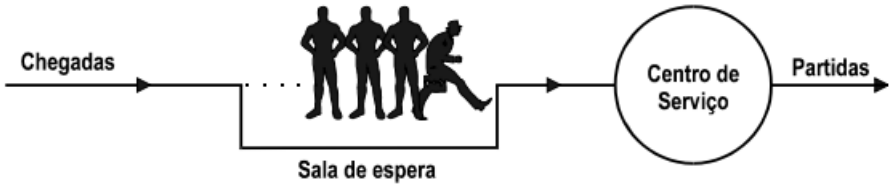
Neste capítulo é descrito o embasamento teórico utilizado para o desenvolvimento do trabalho, apresentando e definindo conceitos envolvendo redes de filas, simulação e modelagem de sistemas, simulação de eventos discretos, Lista de Eventos Futuros, extensão funcional SMPL e trabalhos relacionados do estado da arte.

2.1 Redes de filas

Um modelo ou sistema de filas é composta por usuários que demandam um determinado serviço de interesse. Por esse serviço não poder atender a requisição do usuário de forma imediata, são formadas as filas de espera. As filas são essenciais para o processo de simulação (MARYANSKI, 1980).

Pode-se dizer que os estudos de redes de filas têm como objetivo melhorar o desempenho do sistema num geral, a partir de um melhor gerenciamento dos recursos de serviço disponíveis, assim como reduzir o tempo de espera das filas e agilizar o atendimento das requisições. Dentre as características básicas das filas temos (MAGALHÃES, 1996): chegada, serviço e disciplina de atendimento. Sua estrutura é apresentada na figura 2.1.

Figura 2.1: Diagrama de uma fila.



Fonte: Extraído de (MAGALHÃES, 1996).

2.1.1 Chegada

O processo do usuário procurar um serviço e a forma como isso é realizado descreve o processo de chegada. Esse se dá em um intervalo de tempo que pode ser fixo (constante ou determinístico) ou então em intervalos aleatórios em um processo estocástico seguindo uma distribuição probabilística. Esse último é mais comumente utilizado, já que supõem se que os intervalos entre as chegadas são independentes e igualmente distribuídos.

Para a modelagem das chegadas, o processo de Poisson é o mais utilizado, por ser um processo de renovação de chegadas com uma distribuição exponencial que mais se assemelha a situações práticas. As chegadas dependem do número de usuários presente na chegada, sendo unitária ou em blocos (*batches*) de tamanho aleatório. Pode-se criar também modelos em que existe um certo tipo de dependência entre as chegadas, no qual um determinado tipo de chegada implicaria em uma probabilidade maior de um outro tipo de chegada ocorrer.

2.1.2 Serviço

Os serviços, igualmente ao processo de chegada, podem ser determinísticos ou aleatórios. O tempo em que um serviço é utilizado vai depender do tipo de usuário ou do estado do sistema, porém o mais simples é pensar na independência, no qual o serviço é um processo de renovação. Utiliza-se a distribuição exponencial, Erlang e

hiperexponencial, preferencialmente.

2.1.3 Disciplina de atendimento

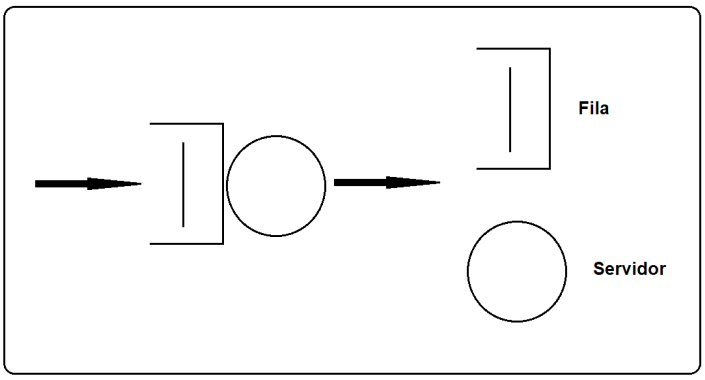
Essa característica diz respeito à maneira como são selecionados os usuários para receber o serviço. Comumente tem-se a disciplina de FCFS (*first come first served*) ou FIFO (*first in first out*) aplicado nas filas, como nas filas de supermercados e cinemas, mas outras disciplinas podem ser aplicadas, como a LCFS (*last come first served*), muito utilizada em sistemas de arquivos ou buscas em discos rígidos. Na computação também é muito aplicada a disciplina de tempo compartilhado, por exemplo no uso da UCP pelos processos, no qual o usuário (processo) recebe doses de atendimento até que sua requisição seja atendida por completo, de maneira sucessiva. Existe ainda a disciplina de prioridade, na qual pode ser implementada uma fila de prioridade e então aplicada a disciplina de FCFS sobre ela, ou ao invés disso, o serviço pode ser interrompido para dar lugar a um outro com prioridade maior.

2.1.4 Modelos baseados em redes de filas

Tem-se na literatura os seguintes modelos (MARYANSKI, 1980; SOARES, 1990):

- O modelo mais simples é composto por um servidor e uma fila (Figura 2.2). O usuário solicita um serviço, caso o serviço esteja ocupado, esse usuário se junta a uma fila e espera até ser selecionado para usar o serviço de acordo com a disciplina de atendimento. Após a realização do serviço, o usuário deixa o sistema. Exemplo: cinema com apenas uma bilheteria.

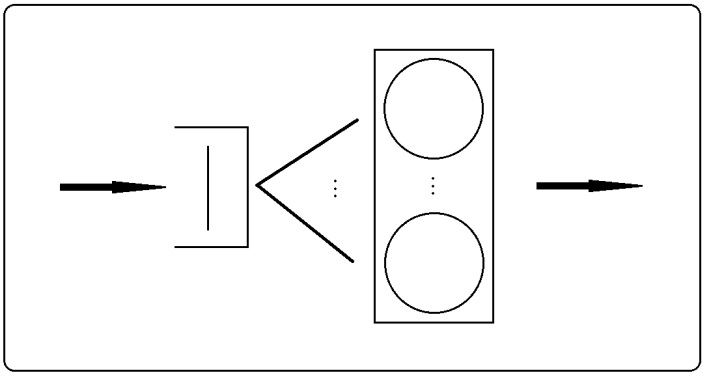
Figura 2.2: Uma fila e um servidor.



Fonte: Elaborado pelo autor.

- Uma variação desse modelo é a adição de vários servidores (Figura 2.3). Exemplo: caixas de uma agência bancária.

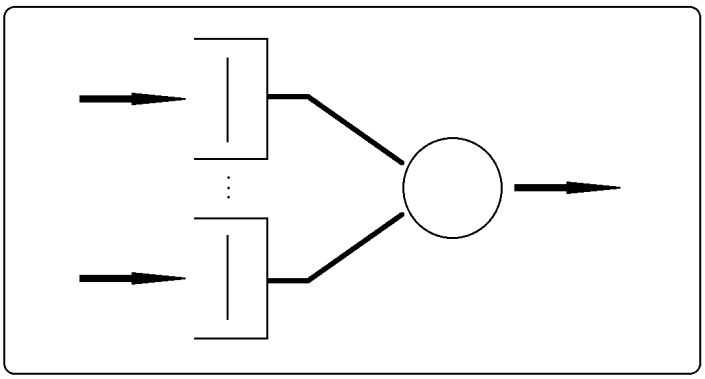
Figura 2.3: Uma fila e n servidores.



Fonte: Elaborado pelo autor.

- Pode-se ter o contrário, várias filas associadas a um único servidor (Figura 2.4). Exemplo: guichê de uma repartição pública com duas filas diferente, uma para aposentados e outra para as demais pessoas. Tem-se preferência a atender a fila de aposentados, só atendendo a outra caso não tenha nenhum aposentado esperando.

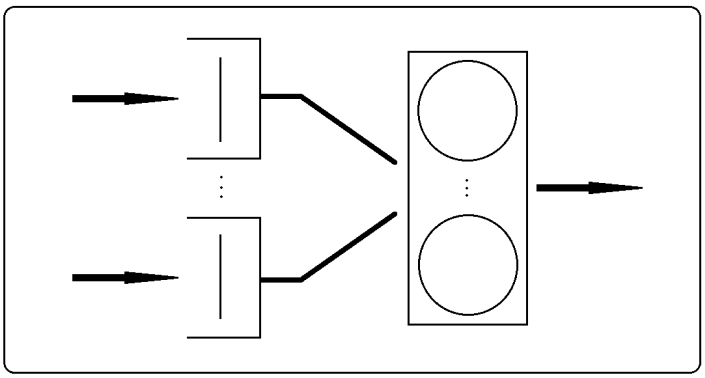
Figura 2.4: N filas e 1 servidor.



Fonte: Elaborado pelo autor.

- O modelo mais complexo envolve múltiplas filas e múltiplos servidores (Figura 2.5). Exemplo: agência bancária com vários caixas que atendem duas filas, uma para idosos, gestantes e portadores de deficiências físicas, e outra para as demais pessoas. A fila com os clientes em geral só é atendida se não houver idosos, gestantes ou portadores de deficiências físicas.

Figura 2.5: N filas e n servidores.



Fonte: Elaborado pelo autor.

Considerando os modelos apresentados, são definidos alguns parâmetros para a caracterização completa do modelo. No modelo mais simples define-se a disciplina da fila, a taxa de chegada dos usuários e o tempo para realizar o serviço. No modelo de uma fila e vários servidores é necessário um parâmetro para a seleção de um servidor

caso exista mais de um ocioso.

Para o modelo de várias filas e um único servidor, além dos parâmetros do modelo mais simples, é necessário um parâmetro para definir de qual fila será retirado o próximo usuário, além de um outro para definir como um usuário escolhe uma fila para esperar. No último modelo são necessários todos os parâmetros citados.

Além dos modelos citados, utiliza-se a notação de Kendall (KENDALL, 1953) para descrever esses modelos, que segue a forma A/B/c/K/Z, na qual A representa a distribuição do processo de chegadas, B a distribuição no tempo necessário para concluir o serviço, c o número de servidores, K a capacidade de cada fila e Z a disciplina da(s) fila(s).

Para A e B é possível utilizar as seguintes distribuições:

- M para distribuição exponencial;
- E_k para distribuição de Erlang-k;
- D para distribuição determinística ou degenerada;
- U para distribuição uniforme;
- G para uma distribuição geral, ou seja, não especificada.

Os parâmetros K e Z podem ser omissos, indicando que a(s) fila(s) tem capacidade infinita e disciplina FCFS.

2.2 Simulação

A simulação implica em imitar o comportamento de algo (ROBERTS et al., 1983), geralmente na computação a simulação é utilizada pra simular sistemas reais e avaliar seu desempenho sem precisar implementá-los de fato.

Primeiramente deve-se desenvolver um modelo que descreva o comportamento dinâmico do sistema. Esse modelo representa de maneira simplificada o sistema em

níveis de detalhes e cada nível representa um grau de abstração (MACDOUGALL, 1975). Os modelos são compostos por três unidades básicas: atividades, processos e eventos. Atividade é a menor unidade de trabalho e sempre está associada ao tempo e varia de acordo com o nível de abstração. Processos são conjuntos de atividades relacionadas logicamente e também variam de acordo com o nível de abstração. Eventos são as mudanças de estado das entidades envolvidas no sistema em um ponto isolado no tempo.

Para um programa de simulação ser uma implementação válida do modelo é necessário fazer a verificação e validação do mesmo, afim de o tornar confiável e garantir que este reproduz o comportamento do sistema com fidelidade.

2.2.1 Modelagem de um sistema

A modelagem do sistema pode ser dividida em três fases: fase de desenvolvimento, fase de testes e fase de análise. A fase de desenvolvimento compreende a descrição do sistema, sua abstração em um modelo, a coleta de dados, a escolha de um método para análise e por fim o desenvolvimento do programa de simulação (SPOLON, 1994).

A fase de testes incorpora a depuração do programa para garantir que não existem erros e os resultados são coerentes, assim como a verificação de que o programa de simulação é uma implementação válida. Por fim é feita a validação do sistema por meio de comparações entre o modelo e o sistema. Na fase de análise, são analisadas as saídas da simulação por métodos que determinam quando se deve parar uma simulação. Esses métodos auxiliam na obtenção de uma medida de desempenho do modelo.

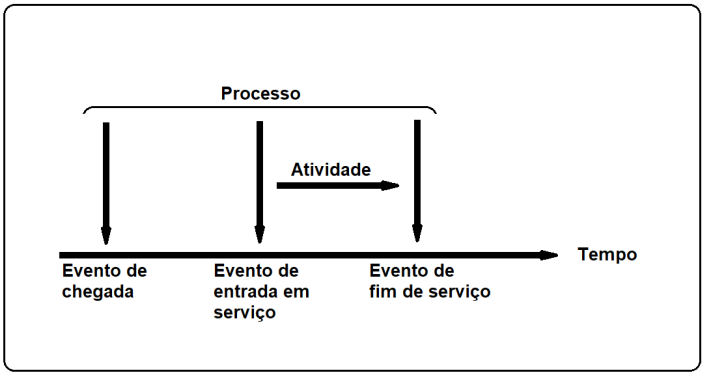
Modelado o sistema, é necessário avaliar o seu desempenho por meio de um programa de simulação que manipula o modelo. Dentre as diferentes ferramentas ressalta-se a solução de modelos por simulação discreta, que possibilita a manipulação de cada componente, a tornando mais flexível e rica em detalhes. Esse tipo de modelo pode ser construído baseado em uma das entidades básicas (atividade, evento e processo):

- Orientação a atividade: São descritas todas as atividades presente para cada entidade do sistema, e o início e fim das atividades é o que desencadeia a realização de novas atividades, baseado no tempo de simulação;

- Orientação a evento: Nesse tipo de orientação, o sistema reage de acordo com as mudanças de estado que ocorrem durante os eventos. Cada evento tem um algoritmo associado. Esses algoritmos ao serem executados de forma ordenada no tempo simulam o sistema;

- Orientação a processo: O sistema é modelado como um conjunto de processos interativos, desencadeados por eventos. Esses processos são executados de forma concorrente (MACDOUGALL, 1975).

Figura 2.6: Relação Evento, Processo e Atividade.



Fonte: Retirado de (SPOLON, 1994).

Na figura 2.6 é apresentada a relação presente entre eventos, processos e atividades. Os eventos representam uma mudança instantânea de estado de alguma entidade do sistema, ocorrendo em um momento isolado, normalmente representando tomadas de decisão. As atividades representam a menor unidade de trabalho do sistema e são sempre associadas à um tempo, dependendo do seu nível de abstração. Já um processo pode ser interpretado como uma coleção de atividades relacionadas, e como as atividades dependem do nível de abstração, os processos também dependem. Por exemplo,

se uma atividade for interpretada como a execução de uma linha de código de um determinado programa, podemos dizer que a execução do programa como um todo seria um processo.

2.2.2 Simulação de redes de filas

Como discutido anteriormente, um sistema (centro de serviço) baseado em filas é descrito pela natureza das chegadas, os serviços adotados, a capacidade e a disciplina da fila. É tomado como referência um sistema simples com uma fila e um servidor (BANKS et al., 2005).

Tem-se que as chegadas ocorrem uma de cada vez de forma aleatória, uma vez que elas entram na fila, elas são atendidas. Os tempos de serviço tem tempo randômico baseado em uma distribuição probabilística que não muda no decorrer do tempo. A capacidade da fila é infinita e sua disciplina é baseada na ordem de chegada (FIFO).

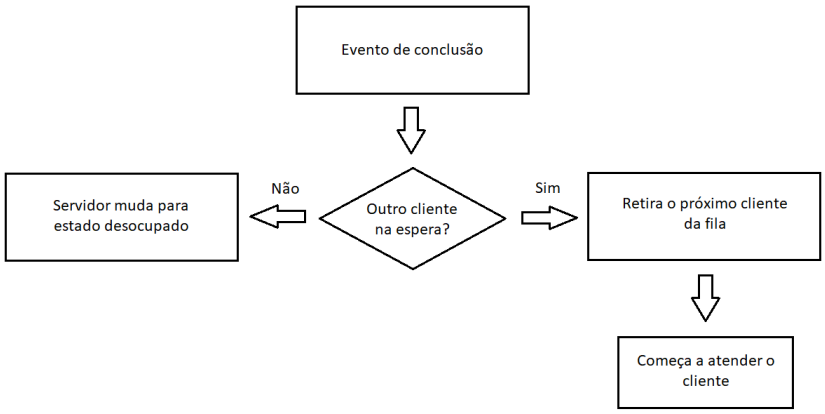
Para qualquer sistema (simples ou com múltiplos canais), a taxa das chegadas deve ser menor que a taxa de execução dos serviços, caso contrário a fila cresce sem limite. Esse tipo de fila é denominado “explosiva” ou instável.

Estado de sistema diz respeito ao número de unidades (clientes) no sistema e o estado do servidor (desocupado ou ocupado). Um evento é um conjunto de ações que geram mudança instantâneas de estado no sistema, no caso para um modelo simples, existem apenas duas possibilidades, eventos de chegada (*arrival*) e de conclusão (*departure*). O relógio de simulação é usado para seguir o tempo de simulação.

No fluxograma 2.7 é apresentado os dois possíveis estados que um servidor adota após um evento de conclusão (*departure*): ocupado ou desocupado. Assim que um evento de conclusão ocorre, é verificado se existem clientes na fila, caso não, o servidor muda para o estado desocupado, caso exista, o cliente é retirado da fila e acessa o serviço. No fluxograma 2.8 é apresentado o processo de chegada, quando uma unidade entra no centro de serviço, caso o servidor esteja desocupado, o cliente é atendido pelo

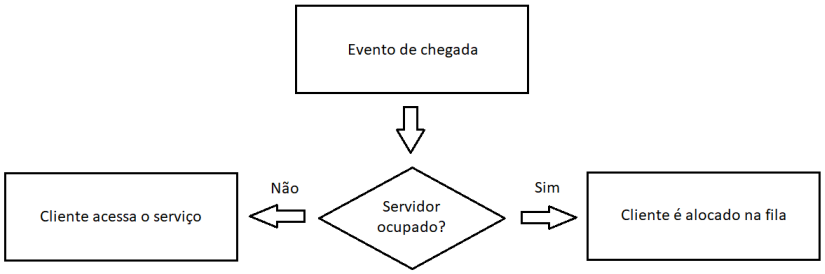
serviço, caso contrário ele é alocado na fila.

Figura 2.7: Fluxograma de um evento de conclusão.



Fonte: Retirado e adaptado de (BANKS et al., 2005).

Figura 2.8: Fluxograma de um cliente chegando ao centro de serviço.



Fonte: Retirado e adaptado de (BANKS et al., 2005).

É de suma importância calcular e coletar medidas de performance dos modelos, principalmente algumas médias:

- Taxa de chegada: $\lambda = \frac{A}{T}$
- Vazão: $X = \frac{C}{T}$
- Utilização do servidor: $U = \frac{B}{T}$
- Tempo médio de serviço por cliente: $T_s = \frac{B}{C}$

Em que A é o número de chegadas, C é o número de eventos completados, B é o tempo médio de servidor ocupado e T um período de tempo de observação.

Lei de Little (*Little's Law*): Seja L o número médio de clientes no sistema, W o tempo médio gasto por eles durante um período de observação e considerando a equação de vazão, tem-se a lei: L = XW, ou assumindo um período suficientemente longo, em que o número de chegadas é aproximadamente igual ao número de eventos completados (C≈A), ou seja, λ = X, é possível reescrever a lei: L = λW. O número médio de clientes no sistema é o produto da vazão pelo tempo médio que um cliente gasta no sistema.

Lei do Tempo de Resposta (*Response Time Law*): Seja N o número de servidores, Z o tempo de resposta de um servidor, X a vazão e R o tempo médio teórico esperado de um servidor, o tempo de resposta R é expresso por: R = ($\frac{N}{X}$) - Z. Essa lei é aplicada principalmente no contexto de sistemas de tempo compartilhado.

2.3 Lista de Eventos Futuros

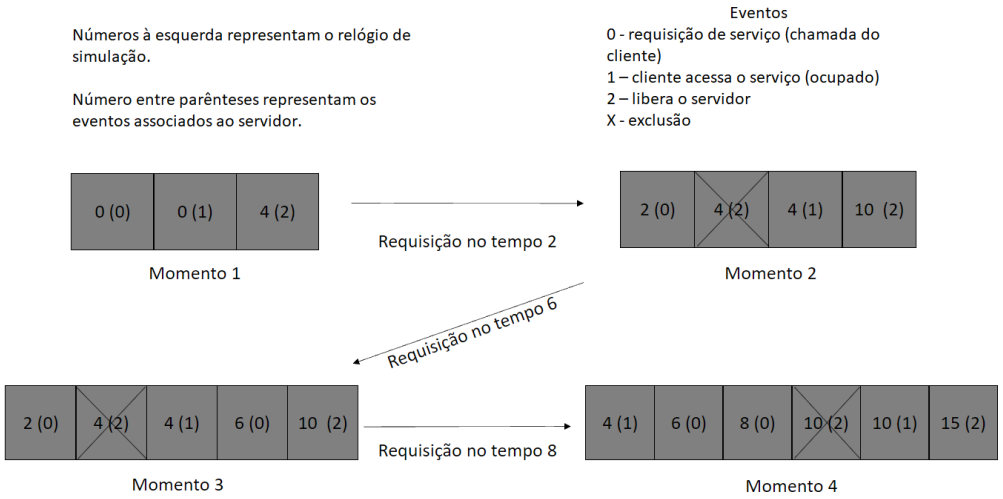
Como citado anteriormente, a simulação discreta pode ser orientada de acordo com as entidades básicas presentes em um modelo (atividade, processo e evento). Especificamente a simulação de eventos discretos, dado um estado inicial e uma lógica de processamento de cada evento, possibilita uma simulação gerenciada pela sequenciação dos eventos. Portanto é necessário a criação da Lista de Eventos Futuros (LEF).

Na LEF cada nó da lista representa um evento com seu identificador (ID) que define as ações que devem ocorrer e qual o seu tempo de ocorrência. Essa lista é ordenada com relação ao tempo de ocorrência dos eventos de maneira crescente, ou seja, após a ocorrência de um evento, o identificador do evento, correspondente as ações, é inserido na lista por meio de um escalonador que garante a ordenação da lista. O uso da LEF junto do relógio de simulação garante que no processo de simulação todos os eventos ocorram em ordem cronológica (MACDOUGALL, 1975; BANKS et al., 2005).

Na figura 2.9 é apresentado um modelo de LEF para um centro de serviço com uma fila e um servidor. Inicialmente o servidor está ocupado até o tempo 4 do relógio de simulação. É feita uma requisição de acesso no tempo 2, que será adicionada na LEF, que gerenciará a requisição até o seu acesso ao servidor.

No momento 2, o servidor é desocupado e a requisição do tempo 2 tem seu acesso garantido. O servidor será liberado no tempo 10. No momento 3 e 4 são feitas requisições nos tempos 6 e 8 respectivamente, que serão enfileiradas na LEF até terem seu acesso concedido. Ainda no momento 4, a requisição do tempo 2 deixa o servidor e a requisição do tempo 6 tem seu acesso garantido, liberando-o no momento 15.

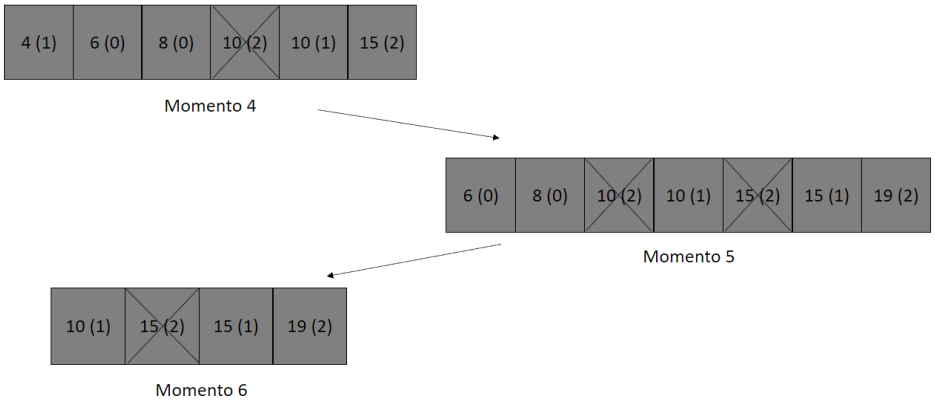
Figura 2.9: Modelo de LEF - Parte 1.



Fonte: Adaptado de (ZAFALON, 2006).

No momento 5 (figura 2.10), a requisição do tempo 6 deixa o servidor e a requisição do tempo 8 tem acesso garantido, deixando-o no momento 19. Ressaltar-se que qualquer nova requisição feita entre os tempos 15 e 19 serão inseridas na LEF. Nota-se que a LEF é uma lista encadeada com operações de inserção ordenada e remoção, garantindo a ocorrência dos eventos de maneira cronológicas, além de ser atualizada com a chegada de um novo evento ou término de um evento em execução. Porém, existem estruturas de dados mais eficientes que a lista encadeada que podem ser utilizadas em seu lugar, como a estrutura de dados *Heap*.

Figura 2.10: Modelo de LEF - Parte 2.



Fonte: Adaptado de (ZAFALON, 2006).

2.4 SMPL

O SMPL é uma extensão funcional da linguagem C para simulação discreta orientada a eventos, no qual as operações de simulação são executadas pela chamada de funções descritas na biblioteca *smpl.h*.

No SMPL são definidos três tipos de entidades: recursos, tokens e eventos (MACDOUGALL; MACDOUGALL, 1987).

- Recursos (*facilities*): representa um recurso do sistema, tanto em questão de hardware, como um barramento ou a UCP, ou um componente de software, como um lock. O SMPL oferece funções para definição, reserva, liberação e preempção, além de informar seus estados.

- *Tokens*: representa as entidades ativas do sistema. Seu deslocamento através do conjunto de recursos promove o comportamento dinâmico do sistema. Pode ser interpretado como uma tarefa em um modelo computacional. O SMPL oferece duas operações básicas para manipulação dos *tokens* pelo sistema: a reserva ou preempção de recursos e pode programar atividades com diversas durações. Caso um *token* tente reservar um recurso ocupado, ele é enfileirado até que o

recurso fique disponível.

- Eventos: representa a mudança de estado de qualquer entidade do sistema. O SMPL oferece funções para programar a ocorrência de eventos e de selecionar a sua execução (causar). Esses eventos são identificados por um número, o instante de tempo em que ele ocorre e seu *token*.

A estrutura de um programa de simulação em SMPL segue uma rotina de inicialização, uma rotina de controle e algumas rotinas de eventos. Na rotina de inicialização define-se as entidades do sistema, assim como o tempo de simulação, tempo de serviço e tempo de chegada e a primeira chegada. Na rotina de controle é realizada a escolha do próximo evento, que passa a ser controlado pela rotina de evento, que escalona um ou mais eventos, voltando o controle para a rotina de controle. No código 1 é apresentado um modelo, em SMPL, de simulação de sistema de fila.

São necessários três passos para a inicialização de um modelo de simulação SMPL: chamada da função *smpl ()* para inicialização do subsistema e para nomeação, chamada da função *facility ()* para criar e nomear um servidor e chamada da função *schedule ()* para escalonar a primeira chegada. Toda a simulação é controlada pelo laço while e a função *time ()* que retorna o tempo atual de simulação.

Na função *cause ()* é retirado o evento cabeça da LEF, incrementado o tempo de simulação para o tempo de ocorrência do evento retirado, salva-o como evento atual o número desse evento e retorna seu número e o do cliente. O número do evento é utilizado para escolher o próximo evento.

No modelo do código 1 são definidos três eventos. Evento 1 representa a chegada do cliente; é escalonado a requisição do servidor pelo cliente de maneira imediata, e escalona-se a próxima chegada. Evento 2 é a requisição do servidor, caso o servidor esteja desocupado, na função *request ()* é reservado o servidor para quem o requisitou, retornando 0 como indicativo de sucesso. Caso o servidor esteja ocupado, na função *request ()* a requisição é enfileirada e é retornado 1. Evento 3 é a representação da

conclusão do serviço, é chamada à função *release ()* para liberação do servidor de um determinado recurso, as estatísticas do sistema são atualizadas e é decrementado o número de servidores ocupados. Na função *release ()* é desenfileirada uma requisição previamente enfileirada, caso exista, que é colocada como cabeça da LEF, para execução imediata.

Ao finalizar a simulação, é chamada à função *report ()*, que gera um relatório incluindo o período de simulação, a utilização do servidor, o tempo médio de ocupação do servidor, o tamanho médio da fila, número de requisições atendidas (conclusão de serviços) e o número de requisições enfileiradas.

Código 1: Modelo de simulação de sistema de fila em SMPL, retirado de (MACDOUGALL; MACDOUGALL, 1987).

```
#include <smpl.h>
main() {
    real Ta=200.0, Ts=100.0, te=200000.0;
    int client=1, event, server;
    smpl(1,"Queue M/M/1");
    server=facility("server", 1);
    schedule(1, 0.0, client);
    while(time () < te) {
        cause(&event, &client);
        switch(event) {
            case 1:
                schedule(2, 0.0, client);
                schedule(1, expnt(Ta), client);
                break;
            case 2:
                if (request(server, client, 0) == 0)
```

```
                    schedule(3, expntl(Ts), client);
                break;
            case 3:
                release(server, client);
                break;
        }
    }
    report();
}
```

2.5 Trabalhos relacionados

Nesta seção são apresentados trabalhos relacionados do estado da arte, que de alguma forma se relacionam com a proposta desse trabalho.

No artigo (TJAHYADI et al., 2019) é apresentado o conceito de computação em nuvem como um sistema complexo com fatores de alta incerteza. Afim de garantir confiabilidade e performance é necessário um sistema de controle. Os autores apresentam a simulação de um sistema de computação em nuvem com um sistema de controle adaptativo. Foi utilizada a plataforma MATLAB com o auxílio das ferramentas Simulink e SimEvents para a implementação do sistema, composto de uma fila ilimitada, um servidor e um sistema de controle adaptativo simples. Os resultados obtidos mostram que o sistema de controle adaptativo proposto é promissor para uso em sistemas de computação em nuvem.

Sistemas de computação de alta performance (*High Performance Computing Systems (HPC)*), são sistemas de larga escala compostos de diversos nós computacionais, consumindo energia massivamente. No artigo (AHMED; TASNIM; YOSHII, 2020) é apresentado um modelo com mecanismo de leilão (*auction mechanism model*) para

reduzir o consumo de energia nesses sistemas. No modelo inclui-se um esquema de alocação de recursos otimizado para processos HPC baseado em frequência de processador e um modelo de leilão antecipado (*forward auction model*) baseado em Vikery-Clarke-Groove (VCG) para permitir a participação na redução de energia dos usuários de HPC. O simulador de escalonamento de processos e o modelo de leilão antecipado foram implementados baseado no Simulus, um simulador paralelo de eventos discretos em Python. Como resultado, as simulações indicaram que o modelo pode alcançar uma redução geral de energia para um sistema HPC, sem prejudicar a participação dos usuários.

No artigo (LARSEN et al., 2017) é proposto um ambiente para simulação atômica (*atomic simulation environment (ASE)*) composta por diversos módulos para a linguagem Python destinados a configurar, controlar, visualizar e analisar simulações nas escalas atômica e eletrônica. São implementadas classes como *Atoms* que armazenam informações sobre propriedades e e posição de átomos individuais. Desta forma, o ASE funciona como um *front-end* para simulações atomísticas no qual estruturas atômicas e simulações de controle de parâmetros podem ser facilmente definidas. Ao mesmo tempo, todo o poder da linguagem Python está disponível para que o usuário possa controlar diferentes simulações inter-relacionadas de forma interativa e detalhada. A linguagem Python foi combinada com a biblioteca NumPy para possibilitar a simulação de tarefas muito complexas, como cálculos de energia, forças, tensões e outras quantidades, podendo ser realizados com uma simples estrutura de loop *for*. O ASE tornou-se um projeto internacional de código aberto completo com desenvolvedores em vários países.

A indústria automotiva é uma das mais importantes do mundo em termos econômicos. Inicialmente, os carros eram compostos basicamente de componentes mecânicos, porém com o desenvolvimento tecnológico, esses passaram a ser equipados com diversos tipos de componentes eletrônicos. Consequentemente, desenvolveu-se a neces-

sidade da criação de chicotes de fiação automotivos. Chicotes automotivos possuem diversas variantes, projetadas e fabricadas em designs específicos para cada carro. A produção de um grande número de variantes demanda uma enorme quantidade tempo e gera muitos custos. No artigo (GUERRERO et al., 2022) é proposto um simulador de eventos discretos em linguagem Python, com auxílio da biblioteca SimPy, para auxiliar na tomada de decisão para o projeto adequado de linha de montagem e na detecção de possíveis processos sobrecarregados, gerando desequilíbrios da linha de produção. O processo de fabricação de chicotes automotivos possui os seguintes processo: criação de 46 tipos diferentes de fios usando máquinas de corte, crimpagem e vedação (*Cutting, crimping, and seals* (CCS)); torção dos fio por máquinas de torção; soldagem dos fios por máquinas de solda; processo de montagem; testes elétricos e o processo de verificação de qualidade e empacotamento. Foram realizadas 100 simulações para cada uma das 6 configurações diferentes propostas, em que cada configuração representa um cenário diferente de linha de produção. Os resultados indicam que é possível realizar uma melhor distribuição da carga de trabalho entre os processos, pelo aumento do número de recursos destinados aos processos de qualidade de embalagem. Isso melhoraria a capacidade e a produtividade do processo de produção de chicotes automotivos.

Capítulo 3

Desenvolvimento do projeto

Neste capítulo estão descritas as característica das funções implementadas para o desenvolvimento da extensão funcional nomeada *simple_sim*, tomando como referência os conceitos de redes de filas, simulação de eventos discretos, extensão funcional SMPL, entre outros conceitos apresentados no capítulo 2.

3.1 Eventos de chegada, requisição e conclusão

Com o objetivo de abstrair e representar os eventos de chegada, requisição e conclusão de serviço, foi desenvolvida a classe *Event*. Como apresentado na seção 2.4, cada evento possui um tipo (*kind*), podendo ser do tipo 1 que indica a entrada (*arrival*) de um cliente no centro de serviço; do tipo 2 que indica uma requisição de acesso ao servidor ou serviço pelo cliente; e do tipo 3 que indica a conclusão (*completion*) de atendimento ao cliente. Podem existir mais tipos de eventos, dependendo do cenário a ser modelado.

Além do atributo de tipo de evento, na classe existem mais dois atributo, sendo eles o de tempo de ocorrência (*time*), baseado no tempo de simulação, e o atributo *costumer* que associa um cliente ao evento.

Defini-se um construtor simples para seus atributos (*time*, *kind* e *costumer*), e um

método *__lt__ ()* que retorna, entre os dois eventos, o que possui o menor tempo de ocorrência.

3.2 Processo de modelagem e simulação do sistema

Para o processo de modelagem e simulação dos sistema foram desenvolvidas as classes *FEL* e *Model*.

Na classe *FEL* realiza-se a implementação da LEF para armazenar cronologicamente a ocorrência dos eventos, em uma estrutura de dados *heap*, da biblioteca *heapq*. A estrutura *heap* foi adotada por ser mais eficiente, em questão de ordenação, do que uma estrutura de lista encadeada comum. No método *append ()* é adicionado um novo evento na fila, que é ordenada cronologicamente. No método *trigger ()* é disparado o evento cabeça da LEF, que retorna o evento.

A classe *Model* modela um único centro de serviço com os atributos *now*, responsável por armazenar o relógio de simulação, sempre inicializado com o valor 0; *fel*, armazena um objeto da classe *FEL* para alimentar a simulação com os eventos; *req_queue* responsável por implementar a fila do centro de serviço, utilizando a classe *deque* da biblioteca *collections*; *inter_arrival_time*, tempo médio entre duas chegadas de clientes; *service_time*, tempo médio de serviço para cada cliente; *_resources*, armazena os servidores; *costumer*, armazena a quantidade de cliente que entram no centro de serviço, é inicializada com o valor 1; *count*, armazena o número de eventos disparados; *_rand*, armazena o gerador de números pseudo-aleatórios; e *_trace*, atributo lógico para habilitar ou desabilitar o marcador de eventos.

Ao inicializar um objeto *Model*, caso o parâmetro *sequence* for 1, ou seja, ele for o primeiro centro de serviço, é escalonado o primeiro evento de chegada.

O método *init* nomeia o centro de serviço e inicializa atributos de métricas como número de saídas de fila, número de *releases*, tamanho da fila em um dado momento, tempo total em fila e tempo de última alteração.

O método *time* retorna o tempo atual da simulação, o método *trace* habilita ou desabilita o marcador de eventos e o método *resource* aloca a abstração de um recurso, com seus servidores.

No método *schedule* são escalonados os eventos que ocorrem no centro de serviço. Recebe os parâmetros de tipo (*kind*), tempo de ocorrência (*time_of_occur*) e cliente *costumer* de um evento. O método instancia um objeto *Event* com esses parâmetros e chama o método *append* para inseri-lo cronologicamente na LEF do centro de serviço.

No método *request* um servidor é reservado para um determinado cliente. Caso exista um servidor livre, o cliente tem o acesso até sua conclusão (*completion*), é retornado *RESERVED*. Caso o(s) servidor(es) esteja(m) ocupado, o cliente é colocado em uma fila de requisições, diferente da LEF, para aguardar a liberação de um servidor, retornando o *QUEUED*.

No caso de servidor(es) ocupado(s), é calculado o tempo total em fila, o tamanho da fila é incrementado e altera-se o tempo da última alteração.

O método *release* realiza a checagem se um cliente foi enfileirado pelo método *request*. Inicialmente procura-se um determinado cliente, passado como parâmetro, no(s) servidor(es) e caso positivo, libera o servidor e calcula-se o tempo total ocupado desse servidor. Caso uma requisição tenha sido enfileirada, é feito o seu desenfileiramento, adicionado-o como evento cabeça da LEF para execução imediata. Calcula-se o novo tempo total em fila, o tamanho da fila é decrementado e altera-se o tempo da última alteração. O servidor é marcado como ocupado, associa o novo cliente ao servidor e marca o tempo de início do serviço.

O método *cause* é o motor de simulação, ou seja, é responsável por fazer a simulação funcionar adequadamente. É retirado o evento cabeça da LEF, incrementando o tempo de simulação para o tempo de ocorrência desse evento. Retorna o evento, que é utilizado para escolher o próximo evento, e seu cliente.

3.3 Geração do relatório de simulação.

Para a geração do relatório de simulação foram implementadas as classes *ResourceData* e *Resource*, além de métodos na classe *Model*.

Os métodos *U* e *B* da classe *Model* desempenham respectivamente a função de calcular e retornar a utilização do(s) recurso(s), e calcular a média de tempo de servidor(es) ocupado(s).

O responsável pela geração do relatório de simulação é o método *Report*, incluindo dados da utilização, média de tempo de servidor(es) ocupado(s), tamanho médio de fila, número de *releases* e saídas de fila.

A classe *ResourceData* cria um objeto para armazenar os dados de um recurso. Inclui dados como número total de servidores (parâmetro de inicialização), número de servidores ocupados e as instâncias desse recurso em um vetor.

A classe *Resource* representa a instância de um recurso, abstração de um servidor. Inicializado com o parâmetro *index* e possui os atributos *costumer* para indicar o atual cliente, o atributo lógico *busy*, o *busy_time* armazenando o tempo utilizado para atender um cliente e o *total_busy_time* que armazena o tempo total utilizado do servidor.

3.4 Geração de números pseudo-aleatórios.

A classe *Rand* implementa o gerador de números pseudo-aleatórios do SMPL. Defini-se as mesmas 15 *streams* para a geração dos números pseudo-aleatórios, bem como as constantes *A* e *M*, utilizadas nos cálculos dos números pseudo-aleatórios.

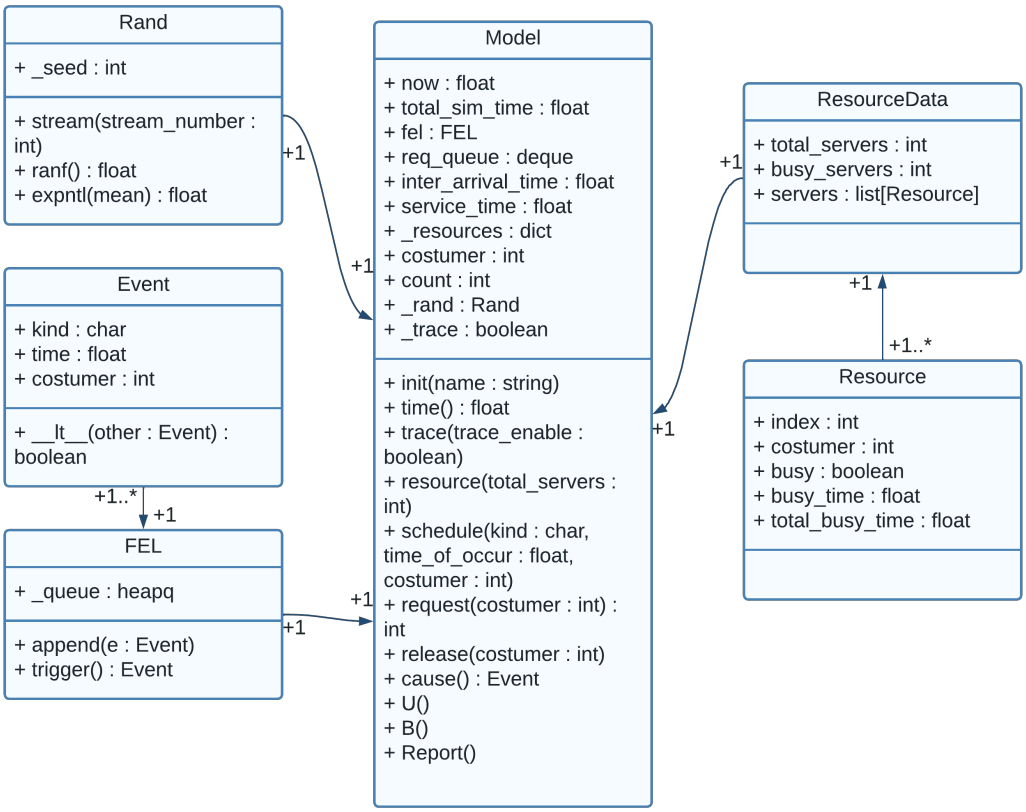
O método *stream* é utilizado para selecionar a *stream* para a geração dos números pseudo-aleatórios, o método *ranf* gera e retorna um número pseudo-aleatório utilizando uma distribuição uniforme em um intervalo entre 0 e 1 e o método *expnlt* gera e retorna um número pseudo-aleatório utilizando uma distribuição exponencial, dado

um valor de média como parâmetro.

3.5 Diagrama de classe e diagrama de atividade

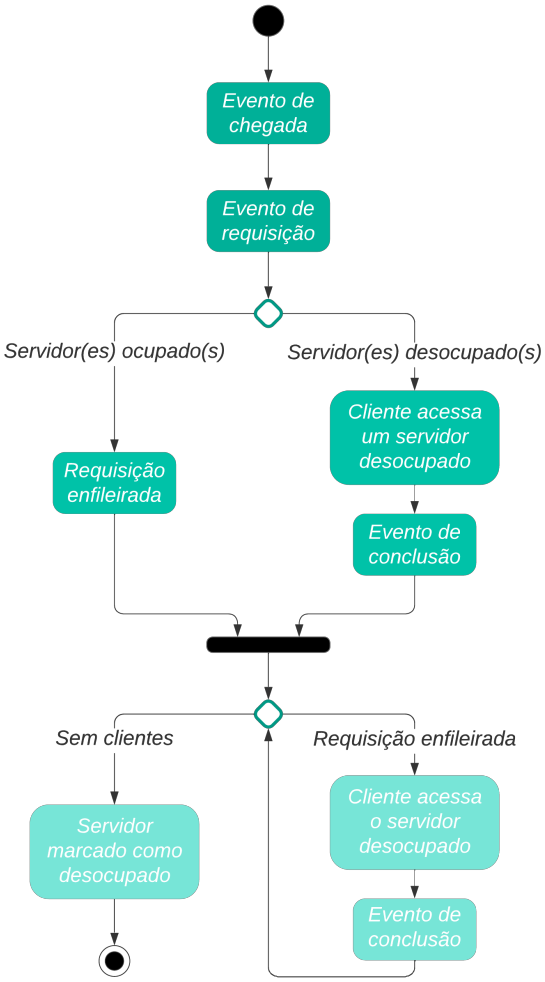
Nessa seção é apresentado o diagrama de classe da extensão funcional, contendo as classes e as suas relações (Figura 3.1), e o diagrama de atividade, demonstrando o fluxo de simulação e utilização das classes (Figura 3.2).

Figura 3.1: Diagrama de classe da extensão funcional *simple_sim*.



Fonte: Elaborado pelo autor.

Figura 3.2: Diagrama de atividade da extensão funcional *simple_sim*.



Fonte: Elaborado pelo autor.

Capítulo 4

Validação

Neste capítulo são apresentados os testes de validação da extensão funcional *simple_sim* desenvolvida para a simulação de eventos discretos. Os testes abrangem a simulação de uma fila M/M/1, os resultados esperados de um sistema e a simulação de filas em série.

4.1 Ambiente de implementação

O ambiente para a execução dos códigos se deu em uma máquina com as seguintes configurações, utilizando a versão Python 3.10:

- SO: Windows 11(64-bit) Versão 21H2;
- Processador: Intel Core i7-9700kf;
- Memória principal: 16.0 GB;
- Memória de armazenamento: 512 GB.

4.2 Testes

4.2.1 Fila M/M/1

Para a validação da extensão funcional *simple_sim* em simular uma fila M/M/1, tomou-se como referência a implementação de um modelo de simulação de sistema de filas *SMPL* apresentado em (MACDOUGALL; MACDOUGALL, 1987), junto dos resultados obtidos de sua simulação. O comparativo entre os resultados obtidos da simulação *SMPL* e *simple_sim* está explicito na Tabela 4.1. O diagrama de uma fila M/M/1 está representado na Figura 4.1, no qual tempo médio entre chegadas de clientes e o tempo médio de serviço respeitam uma distribuição exponencial, possui apenas um servidor, a capacidade da fila é infinita e sua disciplina não possui prioridades.

Os parâmetros utilizados foram de tempo de simulação de 200000.0 unidades de tempo (UT), tempo médio entre chegadas de 200.0 UT e tempo médio de serviço de 100.0 UT.

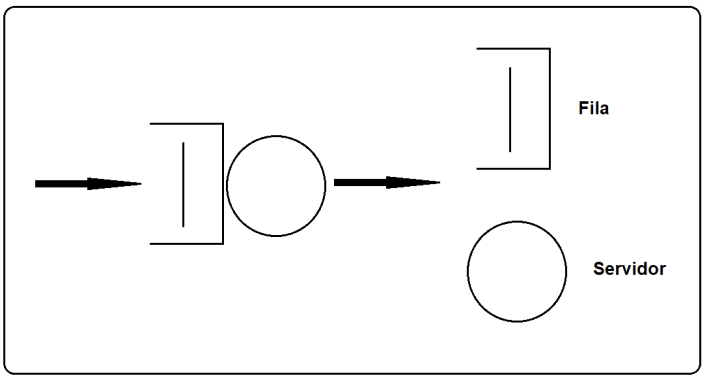
Tabela 4.1: Comparativo entre as extensões funcionais *SMPL* e *simple_sim*.

Métricas	***SMPL***	***simple_sim***
Completions	1025	1025
Utilization	0.53	0.53
Mean Busy Time	103.48	103.48
Mean Queue Length	0.66	0.66

Fonte: Elaborado pelo autor.

Observa-se que a extensão funcional *simple_sim* gerou resultados iguais aos resultados da extensão funcional *SMPL* e portanto atinge o objetivo de criar simulações de eventos discretos e possui a validação, dos resultados obtidos, da extensão funcional *SMPL*. Esses resultados se dão por o *simple_sim* utilizar uma implementação baseada no gerador de números pseudo-aleatórios do *SMPL*.

Figura 4.1: Uma fila e um servidor.



Fonte: Elaborado pelo autor.

4.2.2 Resultados esperados

Ao modelar um sistema e considerar o tempo médio entre chegadas e o tempo médio de serviço é possível prever se haverá ou não a formação de filas de espera. No cenário em que o tempo médio entre chegadas de clientes é alta, ou seja, chegam poucos cliente no sistema, e o tempo médio de serviço é baixo, ou seja, o atendimento é rápido, espera-se a não formação de filas de espera. De modo antagônico, se chegam muitos clientes no sistema e o atendimento desses é demorado, espera-se a formação de longas filas de espera, categorizando o sistema como inviável ou ineficiente.

Para demonstrar esses comportamentos, foi utilizado o mesmo modelo de fila M/M/1 da Seção 4.2.1, alterando os parâmetros de tempo médio entre chegadas (*inter_arrival_time*) e tempo médio de serviço (*service_time*) para criar cenários de formação e não formação de filas. O caso 1 representa o cenário o qual chegam poucos cliente no sistema e o atendimento é rápido. O caso 2 representa o cenário oposto. Os resultados são apresentados na Tabela 4.2.

Os parâmetros utilizados foram de tempo de simulação de 200000.0 UT, para o caso 1: tempo médio entre chegadas de 400.0 UT e tempo médio de serviço de 50.0 UT, para o caso 2: tempo médio entre chegadas de 100.0 UT e tempo médio de serviço de 200.0 UT.

Tabela 4.2: Resultados obtidos das simulações dos casos 1 e 2.

Métricas	**Caso 1**	**Caso 2**
Completions	525	984
Utilization	0.13	1.00
Mean Busy Time	48.75	202.97
Mean Queue Length	0.02	510.76
Queue Exits	78	984

Fonte: Elaborado pelo autor.

Nota-se que no caso 1, apenas 78 dos 525 clientes que acessaram o sistema foram enfileirados, gerando um tamanho médio de fila de apenas 0.02 clientes, representados pelas métricas *Queue Exits* e *Mean Queue Length*, respectivamente. Porém, nota-se que o sistema está sendo subutilizado, com apenas 13% de utilização, ou seja, é possível modificar o sistema para uma melhor utilização do mesmo. No caso 2, todos os clientes que acessaram o sistema foram enfileirados, gerando um tamanho médio de fila de 510.76 clientes. Esse cenário sobrecarrega o sistema (100% de utilização), que pode ser modificado incluindo mais recursos (servidores) para melhorar sua eficiência, por exemplo.

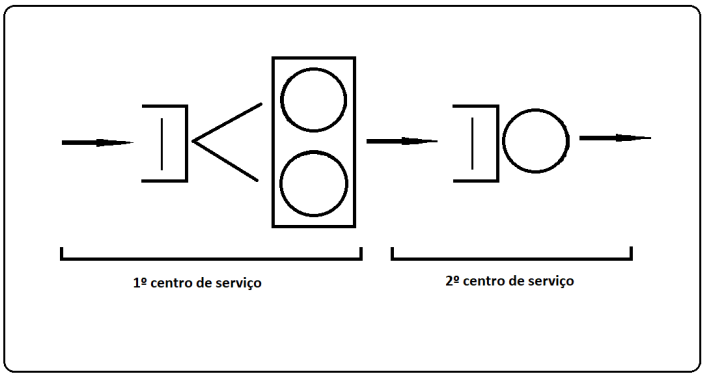
4.2.3 Filas em série

Ao modelar um sistema, é comum a existência de filas em série, ou seja, o modelo possui um centro de serviço seguido de outro centro de serviço, cada um com sua(s) fila(s) e servidor(es), e assim por diante, dependendo do sistema a ser modelado. Para validação da solução desenvolvida, modelou-se um sistema de filas em série com dois centros de serviço, no qual o primeiro centro de serviço possui uma fila e dois servidores (M/M/2) e o segundo uma fila e um servidor (M/M/1), representado pelo diagrama na Figura 4.2. Os resultados são apresentados na Tabela 4.3.

Os parâmetros utilizados foram de tempo de simulação de 200000.0 UT, para o primeiro centro de serviço: tempo médio entre chegadas de 100.0 UT e tempo médio

de serviço de 200.0 UT, para o segundo centro de serviço: tempo médio entre chegadas é desprezível e o tempo médio de serviço de 100.0 UT.

Figura 4.2: Modelo de filas em série.



Fonte: Elaborado pelo autor.

Tabela 4.3: Resultados obtidos da simulação do modelo de filas em série.

Métricas	**1º Centro de serviço**	**2º Centro de serviço**
Completions	1924	1919
Utilization	1.87	0.95
Mean Busy Time	194.76	99.20
Mean Queue Length	11.96	0.33
Queue Exits	1731	512

Fonte: Elaborado pelo autor.

No primeiro centro de serviço nota-se que a maioria dos cliente são enfileirados, sua utilização é alta (soma da porcentagem da utilização de cada servidor) e o tamanho da fila é mediano. No segundo centro de serviço, a utilização também é alta, mas diferente do primeiro, poucos clientes são enfileirados. Podemos concluir que o segundo centro de serviço é mais equilibrado que o primeiro. Uma possível solução para equilibrar o primeiro centro de serviço seria a adição de mais um servidor, reduzindo o tamanho médio da fila de espera.

Capítulo 5

Conclusão

Neste capítulo são apresentadas as conclusões do trabalho desenvolvido, baseado no contexto apresentado no Capítulo 1, no embasamento teórico apresentado no Capítulo 2 e nos testes apresentados no Capítulo 4, além de propostas de trabalhos futuros.

5.1 Conclusões

Conclui-se ao final desse trabalho que a simulação de eventos discretos tem papel fundamental para a otimização de recursos e tomadas de decisão em diferentes problemas que envolvem a estrutura de fila, como computação em nuvem (TJAHYADI et al., 2019), sistemas computacionais de alta performance (AHMED; TASNIM; YOSHII, 2020), linhas de montagem automotiva (GUERRERO et al., 2022), entre outros cenários. Nota-se também o papel atual da linguagem Python, por ser uma linguagem poderosa, versátil e de fácil entendimento.

Foi de suma importância a realização da revisão bibliográfica, que trouxe a tona conceitos indispensáveis para o desenvolvimento da extensão funcional, assim como para o entendimento da simulação de eventos discretos. Permitiu ainda o entendimento e aplicação das métricas de avaliação de sistemas de redes de fila.

O desafio de desenvolver uma solução com aplicação na vida real, para auxiliar

na tomada de decisões, foi produtiva, principalmente para o desenvolvimento pessoal. O aprendizado da linguagem Python e o entendimento do processo de simulações de eventos discretos puderam ser realizados com sucesso para o desenvolvimento do trabalho.

Os resultados dos testes validam a solução desenvolvida, a qual foi capaz de reproduzir os resultados de uma solução consolidada, a biblioteca *SMPL*. Foi capaz de reproduzir comportamentos esperados em uma fila e de modelar e simular um sistema de redes de fila, por meio da modelagem de filas em série.

5.2 Trabalhos futuros

Considerando a biblioteca *SMPL* apresentada e utilizada para a validação da solução desenvolvida, como sugestão de trabalhos futuros, cita-se modificações das funções já implementadas, além de novas funções que complementem a biblioteca para a simulação de múltiplos processos e recursos compartilhados. Nesse sentido, considera-se funções de preempção, suspensão, entre outras. Outra sugestão é a implementação de funções que possam modelar sistemas com mais de uma fila, filas com prioridade e recursos (servidores) específicos para atender as filas com prioridade.

Afim de enriquecer a fidelidade da simulação, sugere-se a implementação de funções geradoras de números pseudo- aleatórios de outras naturezas, como de distribuição uniforme, Erlang, hiperexponencial, normal e até mesmo randômica.

Referências Bibliográficas

AHMED, K.; TASNIM, S.; YOSHII, K. Simulation of auction mechanism model for energy-efficient high performance computing. In: *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York, NY, USA: Association for Computing Machinery, 2020. (SIGSIM-PADS '20), p. 99–104. ISBN 9781450375924. Disponível em: https://doi.org/10.1145/3384441.3395991.

BANKS, J.; CARSON, J.; NELSON, B.; NICOL, D. *Discrete-event System Simulation*. [S.l.]: Pearson Prentice Hall, 2005. (Prentice-Hall international series in industrial and systems engineering). ISBN 9780131446793.

GUERRERO, R.; SERRANO-HERNANDEZ, A.; PASCUAL, J.; FAULIN, J. Simulation model for wire harness design in the car production line optimization using the simpy library. *Sustainability*, v. 14, n. 12, 2022. ISSN 2071-1050. Disponível em: https://www.mdpi.com/2071-1050/14/12/7212.

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*, Institute of Mathematical Statistics, v. 24, n. 3, p. 338 – 354, 1953. Disponível em: https://doi.org/10.1214/aoms/1177728975.

LARSEN, A. H.; MORTENSEN, J. J.; BLOMQVIST, J.; CASTELLI, I. E.; CHRISTENSEN, R.; DUŁAK, M.; FRIIS, J.; GROVES, M. N.; HAMMER, B.; HARGUS, C.; HERMES, E. D.; JENNINGS, P. C.; JENSEN, P. B.; KERMODE, J.; KITCHIN, J. R.; KOLSBJERG, E. L.; KUBAL, J.; KAASBJERG, K.; LYSGAARD, S.; MARONSSON, J. B.; MAXSON, T.; OLSEN, T.; PASTEWKA, L.; PETERSON, A.; ROSTGAARD, C.; SCHIØTZ, J.; SCHüTT, O.; STRANGE, M.; THYGESEN, K. S.; VEGGE, T.; VILHELMSEN, L.; WALTER, M.; ZENG, Z.; JACOBSEN, K. W. The atomic simulation environment—a python library for working with atoms. *Journal of Physics: Condensed Matter*, IOP Publishing, v. 29, n. 27, p. 273002, jun 2017. Disponível em: https://doi.org/10.1088/1361-648x/aa680e.

MACDOUGALL, M. System level simulation. Computer Science Press, p. 1–115, 1975.

MACDOUGALL, M.; MACDOUGALL, M. *Simulating Computer Systems: Techniques and Tools*. [S.l.]: MIT Press, 1987. (Computer Systems Series). ISBN 9780262132299.

MAGALHÃES, M. *Introdução à Rede de Filas.* [S.l.]: ABE- Associação Brasileira de Estatística, 1996.

MARYANSKI, F. *Digital Computer Simulation*. [S.l.]: Hayden Book Company, 1980. ISBN 9780810451186.

ROBERTS, N.; ANDERSEN, D.; DEAL, R.; SHAFFER, W. *Introduction to Computer Simulation: The System Dynamics Approach*. [S.l.]: Addison-Wesley, 1983. ISBN 9780201064148.

SOARES, L. F. G. *Modelagem e Simulação Discreta de Sistemas*. [S.l.: s.n.], 1990.

SPOLON, R. *Um Editor Gráfico para um Ambiente de Simulação Automático.* 1994. Dissertação (Mestrado).

TJAHYADI, H.; ARIBOWO, A.; YUGOPUSPITO, P.; LAZARUSLI, I.; MURWANTARA, I. M. Simulation of adaptive control for cloud computing as queueing system. In: *Proceedings of the 2019 3rd International Conference on Advances in Artificial Intelligence*. New York, NY, USA: Association for Computing Machinery, 2019. (ICAAI 2019), p. 68–72. ISBN 9781450372534. Disponível em: https://doi.org/10.1145/3369114.3369155.

ZAFALON, G. *Interface Gráfica, Particionamento da Aplicação e Dispositivo Gerador de Arcabouços para a CMB-Simulation.* 2006. Monografia (Bacharelado em Ciência da Computação).

Apêndice A

Simulação de uma fila M/M/1 utilizando a biblioteca *simple_sim*

A.1 Introdução

Esse apêndice descreve o desenvolvimento de uma simulação de uma fila M/M/1 utilizando a extensão funcional *simple_sim*. O código fonte da extensão funcional, como o código para a simulação se encontram no repositório do GitHub: https://github.com/GustavoKuroda/simple_sim.

A.2 Uso da biblioteca

Para a criação de uma simulação utilizando a biblioteca *simple_sim.py*, é necessário importar alguns símbolos: *Model*, *RESERVED* e, se necessário, *QUEUED*. Utilize o comando:

```
from simple_sim import Model, RESERVED, QUEUED
```

Afim de reproduzir a simulação, é necessário instanciar um objeto *Model* com alguns parâmetros: *total_sim_time*, *inter_arrival_time*, *service_time* e *sequence*.

```
model = Model(total_sim_time, inter_arrival_time,
                service_time, sequence)
```

Para a inicialização do modelo é utilizado o método *init* junto do argumento que nomeia o modelo. Além de nomear o modelo, o método *inti* inicializa atributos estatísticos.

```
model.init('Simulation model name')
```

Porém, ainda é necessário indicar a quantidade de recursos (servers) utilizando o método *resource*.

```
model.resource(1) # total_servers
```

Para rastrear os evento e seus respectivos tempos de ocorrência, o método *trace* pode ser utilizado com o argumento *True*. Por padrão, o *trace* está desativado.

```
model.trace(True)
```

Por a simulação ser baseada na geração de números pseudo-aleatórios, é necessário indicar uma das quinze *streams* geradoras.

```
# Valid stream numbers range from 1 to 15.
model._rand.stream(1)
```

O código a seguir simula o funcionamento de uma fila M/M/1, gerando um relatório ao final da simulação.

```
while model.time() <= model.total_sim_time
                    and model.fel._queue:
    e = model.cause()
    model.count += 1
    match e.kind:
        case '1': # arrival
            model.schedule('2', 0.0, model.costumer)
```

```
                model.schedule('1', model._rand.expntl(
                                    model.inter_arrival_time),
                                    model.costumer + 1)

            case '2': # request
                if model.request(model.costumer) is RESERVED:
                    model.schedule('3',
                                    model._rand.expntl(
                                    model.service_time),
                                    model.costumer)

            case '3': # completion
                model.release(model.costumer)

    # report of simulation
    model.report()
```

O método *cause* é o coração da simulação, sendo invocado constantemente, desenfileirando um evento e atualizando o tempo de simulação para o tempo de ocorrência desse evento.

Uma simples simulação de eventos discretos possui três tipos de eventos:

- 1 - *arrival*: chegada de um cliente ao centro de serviço;
- 2 - *request*: o cliente requisita o acesso ao serviço (servidor);
- 3 - *release*: conclusão do atendimento do cliente, deixando o serviço (servidor).

Na inicialização do modelo, a primeira chegada é agendada para execução imediata. Cada chegada agenda uma requisição para execução imediata e a próxima chegada, utilizando o método *expntl* do objeto *_rand* da classe *Rand*.

Uma requisição é atendida se existe um recurso (servidor) disponível, agendando um evento *release* utilizando novamente o método *expntl*. Caso não exista um recurso (servidor) disponível, a requisição é enfileirada para futura execução.

Ao final da simulação, o método *report* é invocado para a geração de um relatório com estatísticas de utilização, tempo médio de ocupação, tamanho médio de fila, número total de *releases* e número de saídas de fila.