UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS TRINDADE
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Ricardo do Nascimento Boing

Avanços na Orquestração de Carga Descentralizada em Redes MEC para Garantir a Qualidade de Serviço (QoS) de Sistemas de Tempo Real

Florianópolis
2023

Ricardo do Nascimento Boing

Avanços na Orquestração de Carga Descentralizada em Redes MEC para Garantir a Qualidade de Serviço (QoS) de Sistemas de Tempo Real

Dissertação submetida ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do título de Mestre em Ciências da Computação.
Orientador: Prof. Carlos Becker Westphall, Dr.
Coorientador: Hugo Vaz Sampaio, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Boing, Ricardo do Nascimento
Avanços na orquestração de carga descentralizada em redes MEC para garantir a qualidade de serviço (qos) de sistemas de tempo real / Ricardo do Nascimento Boing ; orientador, Carlos Becker Westphall, coorientador, Hugo Vaz Sampaio, 2023.
97 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciência da Computação. 2. Multi-access Edge Computing (MEC). 3. Fog Computing. 4. Sistemas de tempo real. 5. Orquestração de carga descentralizada. I. Westphall, Carlos Becker. II. Sampaio, Hugo Vaz. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

Ricardo do Nascimento Boing

Avanços na Orquestração de Carga Descentralizada em Redes MEC para Garantir a Qualidade de Serviço (QoS) de Sistemas de Tempo Real

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina - UFSC

Prof.(a) Janine Kniess, Dra.
Universidade do Estado de Santa Catarina - UDESC

Prof. Douglas Dyllon Jeronimo de Macedo, Dr.
Universidade Federal de Santa Catarina - UFSC

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciências da Computação.

Documento assinado digitalmente
PATRICIA DELLA MEA PLENTZ
Data: 16/09/2023 09:25:52-0300
CPF: ***.243.700-**
Verifique as assinaturas em https://v.ufsc.br

Coordenação do Programa de
Pós-Graduação

Documento assinado digitalmente
Carlos Becker Westphall
Data: 15/09/2023 17:01:23-0300
CPF: ***.747.990-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Carlos Becker Westphall, Dr.
Orientador

Florianópolis, 2023.

Dedico este trabalho a Jesus Cristo, meu SENHOR e Salvador, aos meus pais, Nazir e Cristiane, e à minha irmã Eduarda. Também dedico esta pesquisa a todos os brasileiros que a financiaram através dos impostos pagos a união, e a todos os pesquisadores que trabalham diariamente para o avanço da ciência no Brasil.

AGRADECIMENTOS

Primeiramente eu quero expressar a minha gratidão a Deus por essa conquista. Reconheço que não cheguei até aqui apenas pelo meu próprio esforço, mas porque isso fazia parte dos planos de Deus. É o SENHOR quem concede e retira, eleva e humilha, e nada acontece sem a sua permissão. Agradeço ao Deus Pai, e ao seu Filho, o Cristo, que se sacrificou na cruz pela remissão dos nossos pecados.

Em segundo lugar eu quero agradecer a meus pais, Nazir Boing e Cristiane Nascimento. Acredito que, salvo pequenas e tristes exceções, não existe amor terreno que supere o amor de um pai e de uma mãe por seus filhos. Considero os meus pais como sendo os anjos que Deus colocou em minha vida, sempre prontos para me ajudar, aconselhar e socorrer. Agradeço também a minha irmã, Eduarda Boing, pelos bons momentos que vivemos durante a infância, e até pelos inevitáveis conflitos entre irmãos.

Expresso a minha gratidão a todos os colegas, amigos e professores que contribuíram de alguma forma para essa e outras conquistas. Em particular, agradeço ao meu orientador Dr. Carlos Westphall pela oportunidade, aos Dr. Hugo Sampaio e Dr. Fernando Koch pelas grandes contribuições, e ao Me. René Nolio pela parceria ao longo de todo o mestrado. Também gostaria de agradecer ao Me. Cristiano Souza, Me. Rodolfo Borges, João Vitor, e a todos aqueles que participaram das boas partidas de ping-pong que aliviaram o estresse gerado pela rotina de estudos.

Muitas outras pessoas estiveram comigo durante esta longa caminhada. Algumas entraram em minha vida através de laços familiares, enquanto outras eu conheci durante o curso técnico e a graduação. Dentre elas, as minhas avós, Eleonora Boing e Maria Nascimento, e os meus avôs, Antônio Nascimento e Eriberto Boing, que criaram e educaram os meus pais, e por consequência habilitaram a minha vinda ao mundo. Dentre os que conheci durante a formação acadêmica, agradeço a Idien Ariel, Raul Missfeldt, Guilherme Corrêa, Fernando Beltrami, João Guilherme, Dr. Cassiano Machado, Lukas Grudtner, Dr. Leandro Loffi, Me. Pedro Côrte, e a professora Dra. Carla Westphall.

Também agradeço à banca examinadora, formada pela professora Dra. Janine Kniess e os professores Dr. Frank Siqueira e Dr. Douglas Dyllon, por aceitarem julgar este trabalho. Agradeço à universidade pela vaga disponibilizada, e à conferência ICOIN, que, em conjunto com a plataforma IEEE Xplore, aceitaram e publicaram o artigo que habilitou a defesa desta dissertação.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), sob o código de financiamento 001.

“Ouça o sábio e cresça em prudência; e o instruído adquira habilidade
para entender provérbios e parábolas, as palavras e enigmas dos sábios.
O temor do SENHOR é o princípio do saber, mas os loucos desprezam a sabedoria e o ensino.
Filho meu, ouve o ensino de teu pai e não deixes a instrução de tua mãe.
Porque serão diadema de graça para a tua cabeça e colares, para o teu pescoço.”
(BÍBLIA, Provérbios, 1, 5–9)

RESUMO

O paradigma *Multi-access Edge Computing (MEC)* é um conceito em que recursos de computação, anteriormente fornecidos na *Cloud*, são transferidos para a borda da rede, próximo à fonte de dados. Devido à baixa latência gerada pelo *MEC*, o paradigma se torna estratégico para os sistemas de tempo real (STR), pois este tipo de sistema é caracterizado por possuir restrições de tempo. No entanto, o *MEC* possui recursos limitados quando comparado aos data centers da *Cloud*, podendo ficar sobrecarregado durante o pico de demanda. Na literatura, um dos propósitos do uso de redes *MEC* colaborativas é evitar a sobrecarga dos nós através do compartilhamento da carga de trabalho, de modo a encaminhar parte da carga de nós ocupados para nós ociosos. Para realizar o compartilhamento da carga, estratégias de orquestração se fazem necessárias, a fim de estabelecer critérios para a distribuição da carga entre os nós *MEC*. Neste trabalho, os critérios para a seleção dos nós é baseada nos prazos de resposta pré-estabelecidos pelos sistemas STR. Uma busca exploratória foi realizada para encontrar os trabalhos recentemente propostos na literatura, que após serem analisados, serviram para identificar um *gap* de pesquisa que foi tomado como base para a solução proposta. Um modelo de orquestração de carga foi adaptado através da introdução de um novo modelo de fila de espera, utilizada para alocar as requisições, que se caracteriza pela preferência em atender as requisições com os menores prazos. Um simulador de carga foi elaborado para servir como ambiente de experimentação, no qual foram realizados uma série de experimentos para demonstrar a eficácia da adaptação proposta. Os resultados apontam para a redução do número de encaminhamentos de requisições em até 3,82%, e um aumento no número de prazos cumpridos em até 5,90%.

Palavras-chave: *Multi-access Edge Computing (MEC)*. *Fog Computing*. Sistemas de tempo real. Orquestração de carga descentralizada.

ABSTRACT

The Multi-Access Edge Computing (MEC) paradigm is a concept in which computing resources previously provided in the Cloud, are transferred to the edge of the network, close to the data source. Due to the low latency generated by MEC, the paradigm becomes strategic for real-time systems (RTS), as this type of system is characterized by having time constraints. However, MEC has limited resources when compared to Cloud data centers and may become overloaded during peak demand. In the literature, one of the purposes of using collaborative MEC networks is to avoid overloading nodes by sharing the workload in order to forward part of the load from busy nodes to idle nodes. To carry out load sharing, orchestration strategies are necessary in order to establish criteria for distributing the load between MEC nodes. In this work, the criteria for selecting nodes are based on response deadlines pre-established by RTS systems. An exploratory search was carried out to find recently proposed works in the literature, which, after being analyzed, served to identify a research gap that was taken as the basis for the proposed solution. A load orchestration model was adapted through the introduction of a new queue model used to allocate requests, which is characterized by the preference to serve requests with the shortest deadlines. A load simulator was designed to serve as an experimentation environment in which a series of experiments were carried out to demonstrate the effectiveness of the proposed adaptation. The results point to a reduction in the number of requests forwarded by up to 3.82%, and an increase in the number of deadlines met by up to 5.90%.

Keywords: Multi-Access Edge Computing (MEC). Fog Computing. Real time systems. Decentralized load orchestration.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os sistemas de tempo real (STR) são caracterizados por possuírem restrições em relação ao tempo. Um prazo de resposta é estipulado para cada serviço fornecido por este tipo de sistema, e o desempenho do sistema depende do cumprimento destes prazos (AUSTAD *et al.*, 2023). De acordo com Austad *et al.* (2023) e Kopetz e Steiner (2011), alguns sistemas STR são tolerantes a perda de prazos, enquanto outros se tornam inoperantes diante uma resposta atrasada.

O tempo de resposta é uma variável que está diretamente ligada à capacidade computacional da máquina onde o sistema está sendo executado (CICCONETTI; CONTI; PASSARELLA, 2020). O paradigma *Cloud Computing* é um conceito em que serviços de computação são fornecidos via rede, através de grandes data centers, e proporciona aos usuários a sensação de ilimitação de recursos (MELL; GRANCE, 2011). O alto poder computacional da *Cloud* a torna atrativa para fornecer os serviços a sistemas STR, porém a distância geográfica entre os usuários e os data centers pode inviabilizar o uso do paradigma devido à latência da rede, que deve contribuir para o aumento do tempo de resposta (BATISTA; FIGUEIREDO; PRAZERES, 2022; DEBAUCHE *et al.*, 2022; CICCONETTI; CONTI; PASSARELLA, 2020; MAO *et al.*, 2017).

Para contornar os problemas geográficos, a aproximação entre os recursos de provedores *Cloud*, e dispositivos de usuários, se tornou tópico de pesquisa, dando origem a paradigmas como a *Fog Computing* e o *Multi-Access Edge Computing (MEC)*. Para ambos os paradigmas, dentre os principais objetivos, tem-se a redução dos impactos gerados pela latência da rede no tempo de resposta dos serviços (WU *et al.*, 2023; DEBAUCHE *et al.*, 2022; IORGA *et al.*, 2018). A diferença se dá pela localização dos recursos, pois a *Fog* é um paradigma multi-camadas, podendo os nós *Fog* estarem em qualquer ponto entre os usuários e a *Cloud* (IORGA *et al.*, 2018). Já o *MEC* é um conceito que pode ser visto como uma especialização da *Fog*, pois está limitado a borda da rede, estando os nós *MEC* localizados dentro das estações-base e muito próximos dos usuários (WU *et al.*, 2023; DEBAUCHE *et al.*, 2022).

1.1 PROBLEMA DE PESQUISA

A aproximação entre recursos e usuários pode ser vantajosa em termos de latência, porém manter os recursos na borda da rede é mais custoso do que em data centers centralizados (CICCONETTI; CONTI; PASSARELLA, 2020). A sensação de infinitude de recursos, obtida ao utilizar a *Cloud*, pode não ser alcançada no *MEC* em casos de sobrecarga dos nós (WU *et al.*, 2023). Ao receber um número de requisições que ultrapasse a sua capacidade, um nó *MEC* se tornará sobrecarregado, afetando a qualidade de serviço ao gerar atrasos no tempo de resposta (ZHANG *et al.*, 2022).

Para garantir a qualidade dos serviços prestados a sistemas STR, é essencial a presença de mecanismos para lidar com a carga que será imposta aos nós (provedores) de redes *MEC*. Trabalhos como Beraldi *et al.* (2020a,b,c) buscam utilizar os recursos dos nós provedores de forma colaborativa, transferindo a carga entre os nós sempre que um determinado critério for alcançado. Esta transferência pode ser realizada para atingir diferentes objetivos, como a redução do tempo de resposta das requisições e o evitamento da sobrecarga das redes *backhaul* (MAO *et al.*, 2017; BERALDI *et al.*, 2020a, 2020b, 2020c).

Dentre as estratégias para lidar com a carga de trabalho, as abordagens de balanceamento de carga buscam alcançar o equilíbrio da carga entre os nós provedores (EL-NATTAT *et al.*, 2021; TAHMASEBI-POUYA; SARRAM; MOSTAFAVI, 2022). No entanto, existem abordagens que buscam resolver o problema de distribuição de carga sem considerar o equilíbrio entre os nós. Um exemplo é a abordagem proposta em Beraldi *et al.* (2020a,b,c), que distribui a carga entre os nós de maneira individual e cega, sem garantir o equilíbrio de carga.

A fim de possibilitar a discussão e a comparação entre ambos os tipos de abordagens, neste trabalho é apresentado o conceito de orquestração de carga. Em computação, o termo orquestração é comumente utilizado para se referir a automatização, coordenação e gerenciamento de recursos (DE SOUSA *et al.*, 2019). Neste trabalho, o termo orquestração da carga é utilizado para se referir a orquestração das requisições, recebidas por redes de nós *Fog* (ou *MEC*), que abrange abordagens que buscam, ou não, o balanceamento de carga entre os nós.

Sendo assim, a pergunta de pesquisa deste trabalho é: "como orquestrar a carga de trabalho, em uma rede de nós *MEC* colaborativos, para garantir a qualidade dos serviços, providos a sistemas de tempo real, através do cumprimento dos prazos de resposta pré-estabelecidos?"A hipótese assumida é que a descentralização da decisão de orquestração de carga, realizada na borda da rede, pode reduzir a latência de rede e o sobrecarregamento dos nós *MEC*, e por consequência afetar positivamente o cumprimento dos prazos de resposta das requisições.

1.2 MOTIVAÇÃO E JUSTIFICATIVA

A quinta geração de redes móveis (5g) se caracteriza, principalmente, pelas altas taxas de velocidade e baixa latência de rede. Para que esses benefícios possam ser entregues aos usuários, um dos pilares da tecnologia 5g é o paradigma *MEC*, que deve gerar uma latência muito abaixo daquela obtida na *Cloud*, pois através do *MEC* será possível fornecer os serviços de computação na borda da rede (MAO *et al.*, 2017). Além disso, a aproximação entre os provedores, e as fontes de dados, deve resultar em uma menor sobrecarga das redes backhaul (MAO *et al.*, 2017).

Com a adoção da tecnologia 5g, o *MEC* terá o seu uso impulsionado, motivando

a realização de estudos para melhorar a qualidade dos serviços prestados aos usuários. Dentre os atuais tópicos de pesquisa, o problema do posicionamento da carga de trabalho, em redes formadas por nós *MEC* colaborativos, vem sendo estudado em trabalhos como Beraldi *et al.* (2020a,b,c). No entanto, um gap de pesquisa foi identificado após a realização de uma revisão bibliográfica. De modo geral, constatou-se que o cumprimento dos prazos de resposta, e a redução do uso da rede, foram considerados nas soluções encontradas, porém não de forma simultânea. Portanto, a justificativa para a realização deste trabalho é a necessidade de se propor uma solução que leve em consideração estes dois aspectos.

1.3 OBJETIVOS

O objetivo geral desse trabalho é propor um modelo de orquestração de carga, ou a adaptação de um orquestrador existente, para atuar em uma rede colaborativa de nós *MEC* utilizada para fornecer serviços a sistemas STR. O modelo proposto deve atuar sobre a carga de trabalho, recebida individualmente por cada nó *MEC*, de modo a buscar o cumprimento dos prazos de resposta de cada requisição. Para cumprir os prazos de resposta, o orquestrador deverá considerar, além da carga existente em cada nó *MEC*, a redução da troca de mensagens na rede, a fim de evitar possíveis sobrecarregamentos que possam afetar a latência de comunicação.

1.3.1 Objetivos específicos

Os objetivos específicos deste trabalho são:

1. Realizar uma análise sobre o estado da arte, com o intuito de identificar as abordagens recentes de orquestração de carga que possam ser utilizadas em redes colaborativas de nós *MEC*;
    a) A análise deve considerar a comparação entre os modelos de orquestração de carga encontrados, evidenciando as semelhanças e diferenças entre os modelos, para ser possível identificar os *gaps* de pesquisa a serem explorados na solução a ser proposta;
2. Propor uma abordagem de orquestração de carga inovadora, ou a adaptação de uma abordagem existente, a fim de garantir a qualidade dos serviços providos a sistemas STR, através do cumprimento dos prazos de resposta das requisições, e considerando a evitação do sobrecarregamento da rede;
3. Discutir a relevância do modelo de orquestração de carga proposto, por meio de uma análise sobre as possíveis vantagens e desvantagens da utilização do novo modelo.

1.4 METODOLOGIA DE PESQUISA

Este trabalho se caracteriza como sendo de natureza aplicada e se divide em três etapas. Inicialmente é realizada uma revisão da bibliografia, a fim de apresentar uma definição sobre os conceitos fundamentais e analisar as soluções de orquestração de carga existentes na literatura. Em seguida, uma nova solução de orquestração de carga é elaborada, a qual é posteriormente submetida a uma etapa experimental para avaliar o seu desempenho.

Para a realização da pesquisa bibliográfica, a ferramenta de pesquisa Google Scholar foi utilizada, assim como três bases de dados relacionadas a área da computação: ScienceDirect, IEEE Explorer e Springer. Esta etapa se caracteriza pela busca exploratória por livros, e artigos científicos, utilizados para a definição dos conceitos fundamentais. Em seguida, esta mesma ferramenta, e as bases de dados, foram utilizadas para encontrar os trabalhos que apresentam soluções de orquestração de carga. Após uma leitura de cada trabalho correlato, realiza-se um resumo sobre cada solução, e comparam-se as abordagens encontradas a fim de identificar um *gap* de pesquisa.

Na etapa seguinte, um modelo de orquestração de carga, anteriormente encontrado na literatura, é tomado como base para a proposta de uma nova abordagem de orquestração de carga. A nova abordagem é apresentada com o propósito de cobrir os *gaps* de pesquisa identificados, e se caracteriza por ser uma versão adaptada da abordagem anterior. Em seguida, ambos os modelos de orquestração são submetidos a uma etapa experimental, realizada por meio de um ambiente de experimentação, elaborado neste trabalho, que utiliza um *dataset*, o qual também é elaborado pelo autor para a realização dos experimentos.

O ambiente de experimentação é um simulador de carga, concebido com o propósito de avaliar o desempenho do novo modelo de orquestração de carga e compará-lo com a solução original. Trata-se de um ambiente de experimentação controlado, que considera os objetivos de pesquisa e as delimitações de escopo, deste trabalho, para gerar dados estatísticos sobre o desempenho de cada abordagem de orquestração de carga. A simulação ocorre através da inserção dos dados de entrada, que correspondem aos cenários de experimentação (*dataset*) e as abordagens a serem avaliadas. Os dados estatísticos são gerados com base em métricas de desempenho que são pré-definidas dentro do simulador.

1.5 CONTRIBUIÇÕES

Neste trabalho foram alcançados os seguintes resultados:

1. Uma análise sobre o estado da arte, que compara as abordagens de orquestração de carga encontradas e destaca o *gap* de pesquisa existente entre estas abordagens;

2. Adaptação de um modelo de orquestração de carga descentralizado, para atuar em redes colaborativas de nós *MEC*, a fim de garantir a qualidade dos serviços fornecidos a sistemas STR;
    a) Novo modelo de escalonamento de requisições, não preemptivo, que aloca as requisições na fila de espera de cada nó *MEC* com base nos prazos de resposta de cada requisição;
3. Novo simulador de carga, utilizado como ambiente de experimentação para possibilitar a realização dos testes experimentais sobre a abordagem de orquestração de carga proposta;
4. Um conjunto de *datasets*, o qual é utilizado no ambiente de experimentação para habilitar a avaliação da abordagem de orquestração de carga;
5. Resultados experimentais, e uma discussão acerca destes resultados, os quais evidenciam a relevância da abordagem proposta para o cumprimento dos prazos de resposta.

1.6 DELIMITAÇÃO DE ESCOPO

Neste trabalho é considerado um tempo hipotético, e previsível, para o processamento de cada tipo de serviço fornecido pela rede colaborativa de nós *MEC*. Ao realizar o envio de requisições, os sistemas STR definem um prazo de resposta conforme a importância da informação, e do tipo de serviço requisitado. Considera-se que cada nó *MEC* atenderá uma única requisição por vez, sem preempção, e que todos os dados necessários para atender as requisições são fornecidos pelos próprios sistemas STR. No entanto, nenhum estudo é realizado sobre a segurança dos dados enviados pela rede.

Conforme os prazos, e a sobrecarga dos nós *MEC*, o trabalho busca realizar a distribuição de carga a fim de alcançar os objetivos que foram estabelecidos neste trabalho. Apesar de um dos objetivos ser a redução do número de encaminhamentos de requisições, para reduzir o uso da rede, nenhum estudo é realizado sobre os impactos gerados pela latência para o cumprimento dos prazos de resposta. De modo geral, o ambiente de experimentação utilizado é do tipo controlado, no qual o tempo de processamento dos serviços é conhecido, a latência de rede é nula, a disponibilidade energética é infinita, os nós *MEC* estão disponíveis durante todo o tempo, e outras aplicações do sistema operacional não estão concorrendo pelos recursos utilizados pelos serviços de tempo real.

1.7 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte maneira:

- Capítulo 2: são apresentados os conceitos fundamentais deste trabalho: Sistemas de tempo real (STR), *Cloud Computing*, *Fog Computing*, *Multi-access Edge Computing (MEC)*, e Orquestração de carga centralizada, centralizada por região, e descentralizada;
- Capítulo 3: é apresentado o estado da arte, contendo um relato sobre cada solução encontrada, uma comparação entre essas soluções, e uma análise geral sobre o estado da arte, o qual inclui o gap de pesquisa identificado;
- Capítulo 4: é apresentada a proposta do trabalho. Na parte inicial deste capítulo é explicado o funcionamento do modelo de orquestração de carga original, tomado como base para a solução proposta. Em seguida é definido o mecanismo de escalonamento de requisições, o qual é proposto para adaptar o modelo de orquestração de carga original;
- Capítulo 5: é apresentada uma explicação sobre como o simulador de carga (ambiente de experimentação) funciona, desde a leitura dos dados de entrada até a geração dos dados de saída;
- Capítulo 6: é apresentada uma explicação sobre a geração dos cenários de experimentação, os quais são, na sequência, descritos conforme as suas características individuais;
- Capítulo 7: o capítulo é iniciado por meio de uma explicação sobre as métricas de desempenho utilizadas para avaliar os modelos de orquestração de carga. Na sequência são apresentados os resultados, os quais são descritos mediante uma explicação conforme o cenário de experimentação utilizado. Por fim, o capítulo é encerrado com uma seção de discussão sobre os resultados obtidos nos três cenários de experimentação;
- Capítulo 8: são apresentadas as conclusões do trabalho e uma breve discussão sobre a continuação da pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 SISTEMAS DE TEMPO REAL (STR)

Sistemas de tempo real (STR) são aqueles em que o desempenho está diretamente ligado ao tempo que o sistema levará para entregar uma resposta. Isto é, a corretude de um sistema STR não depende apenas da saída lógica, mas também do cumprimento dos prazos que foram estabelecidos. Múltiplos componentes podem compor um sistema STR, e cada componente possui seu próprio nível de sensibilidade ao tempo. De modo geral, conforme o impacto que uma perda de prazo pode gerar para o sistema, existem três tipos de sistema STR: *Soft real-time*, *Firm real-time* e *Hard real-time* (AUSTAD *et al.*, 2023).

1. Sistemas *Soft real-time*: são sistemas onde o cumprimento de prazos é desejável, porém não é crítico para o sistema (KOPETZ; STEINER, 2011). Uma resposta atrasada continua sendo importante para o sistema, porém a relevância pode diminuir à medida que o atraso se tornar maior (AUSTAD *et al.*, 2023);
2. Sistemas *Firm real-time*: são sistemas capazes de se recuperar diante uma perda de prazo, porém a resposta atrasada se torna irrelevante (AUSTAD *et al.*, 2023);
3. Sistemas *Hard real-time*: são sistemas intolerantes a falhas, de modo que uma perda de prazo resulta na falha total do sistema (AUSTAD *et al.*, 2023).

Independentemente do seu tipo, um sistema STR deve possuir uma estratégia de escalonamento para que as tarefas possam ter o prazo respeitado (LEE, 2022). O algoritmo *Earliest Deadline First (EDF)* é um exemplo de abordagem utilizada por sistemas STR. O algoritmo *EDF* organiza a fila de espera conforme o tempo restante para o término do prazo, de modo a priorizar as tarefas cujo prazo terminará mais cedo. O algoritmo *EDF* possui um ótimo desempenho em casos normais, porém possui um péssimo desempenho em casos de sobrecarga (BARUAH; HARITSA, 1997).

Neste trabalho é introduzido um escalonador de tarefas no Capítulo 4. O escalonador faz parte da proposta deste trabalho, e foi utilizado em um contexto envolvendo tarefas de um sistema de videomonitoramento do tipo *Soft real-time*, no qual as tarefas apresentam diferentes prazos de resposta.

2.2 *CLOUD COMPUTING*

Cloud Computing (Cloud) é definido pelo *National Institute of Standards and Technology (NIST)* como um conceito em que recursos de computação, como processamento e armazenamento de dados, são fornecidos através da rede, e devem estar

acessíveis a partir de qualquer lugar. Estes recursos podem ser desde uma infraestrutura, onde aplicações dos usuários são instaladas, até aplicações ofertadas pelos próprios provedores *Cloud*. Os serviços devem ser fornecidos sem a necessidade dos usuários conhecerem a localização do provedor, e com a sensação de que os recursos fornecidos são ilimitados. Todos os serviços prestados devem ser provisionados de forma automática, sem a interação humana, para serem acessados por equipamentos como telefones celulares, notebooks, tablets e *workstations* (MELL; GRANCE, 2011).

A vantagem da utilização da *Cloud* é que os consumidores pagam apenas pelos serviços que consomem. No modelo tradicional, é necessário investir em recursos de software e hardware, assim como arcar com os custos de manutenção dos equipamentos (LIAQAT *et al.*, 2017). A escalabilidade e disponibilidade são características do modelo *Cloud*, tornando a manutenção imperceptível aos usuários (MELL; GRANCE, 2011).

Apesar do alto poder computacional da *Cloud*, a qualidade de serviço das aplicações de usuários pode ser afetada pela alta latência gerada pela distância geográfica. Isto é, a *Cloud* pode ser definida como um conjunto de nós servidores centralizados, que estão localizados longe dos usuários (BATISTA; FIGUEIREDO; PRAZERES, 2022; DEBAUCHE *et al.*, 2022). Considerando a afirmação de Austad *et al.* (2023), sobre a dependência dos sistemas STR em relação ao tempo, a utilização da *Cloud* pode se mostrar inaceitável para este tipo de sistema.

Para lidar com as diferentes localidades geográficas surgiu o paradigma *Distributed Cloud Computing (DCloud)*. De acordo com Rashid *et al.* (2018), o paradigma *DCloud* reúne os servidores e os usuários distribuídos em diferentes localidades. Dentre os objetivos tem-se a melhora da qualidade dos serviços através da redução do tempo de processamento, redução do tempo de espera e um melhor desempenho na comunicação entre usuários e servidores.

2.2.1 *Fog Computing*

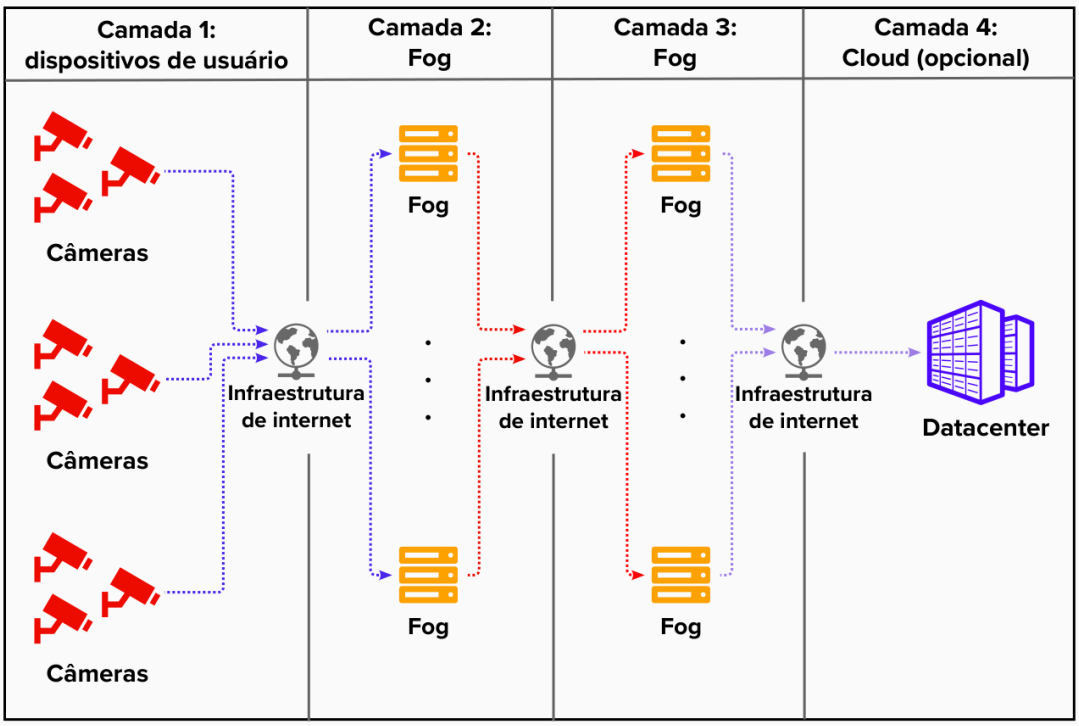
O paradigma *Fog Computing (Fog)* é um conceito em que recursos de computação são fornecidos em camadas intermediárias, entre os dispositivos de usuário e a *Cloud*. Trata-se de um modelo multi-camadas, que pode ser escalável de modo horizontal, entre os nós de uma mesma camada, e/ou de modo vertical, entre os nós de diferentes camadas. O objetivo da *Fog* é proporcionar uma maior qualidade de serviço para as aplicações sensíveis ao tempo de resposta, visto que a proximidade entre os serviços e os usuários acarretará uma menor latência gerada pela rede (IORGA *et al.*, 2018).

Apesar das vantagens da *Fog*, o paradigma não substitui a *Cloud*. O uso dos data centers da *Cloud* se torna opcional na arquitetura *Fog*, e um exemplo é apresentado em Liutkevičius *et al.* (2022). Liutkevičius *et al.* (2022) apresentam uma arquitetura

na qual os nós *Fog* são utilizados para fornecer serviços de tempo real a dispositivos de usuários móveis, havendo a opção de utilizar a *Cloud* como um nó extra com alta capacidade computacional.

Na Figura 1 é apresentada uma arquitetura genérica de *Fog* que apresenta as características mencionadas anteriormente. Isto é, os nós *Fog* são distribuídos de modo vertical e horizontal, na camada 2 e camada 3, e os data centers da *Cloud* (opcionais) são apresentados na camada 4.

Figura 1 – Representação ilustrativa da arquitetura Fog.



Fonte: Elaborado pelo autor.

O problema da *Fog* são os recursos limitados dos nós. É fundamental existir um gerenciamento dos recursos, a fim de compartilhar a carga de nós sobrecarregados com nós que estejam mais ociosos (PEREIRA *et al.*, 2020). Na Sec. 2.3 são discutidos métodos que podem ser utilizados para orquestrar a carga entre os diferentes nós da camada *Fog*, a fim de reduzir a sobrecarga dos nós e aumentar a qualidade dos serviços fornecidos aos usuários.

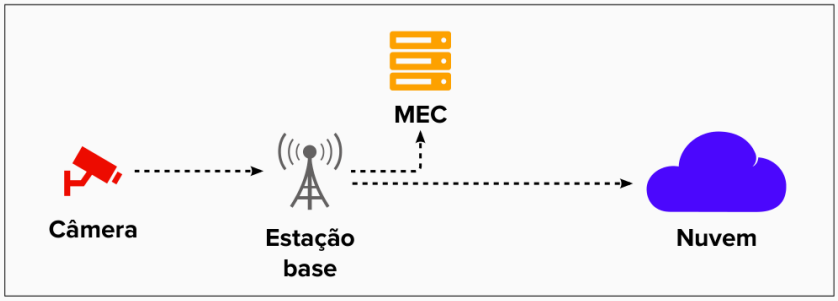
2.2.1.1 *Multi-access Edge Computing (MEC)*

Multi-access Edge Computing (MEC) é um conceito que utiliza as estações-base, das operadoras de telecomunicações, para fornecer recursos de computação com baixa latência (WU *et al.*, 2023; DEBAUCHE *et al.*, 2022). Tecnologias de rede móvel, como a quinta geração de redes móveis (5G), utilizam o *MEC* como um pilar

para atender a demanda de aplicações de sistemas STR, como realidade virtual, jogos online e streaming de vídeo. O *MEC* também contribuirá para reduzir o tráfego das redes de *backhaul*, pois as requisições que eram destinadas à *Cloud* serão atendidas na borda da rede (MAO *et al.*, 2017). Alguns autores, como Toczé e Nadjm-Tehrani (2019), consideram o *MEC* como equivalente à *Fog*, porém o *MEC* é uma especialização que abrange a parte da *Fog* que está mais próxima dos usuários (DEBAUCHE *et al.*, 2022; WU *et al.*, 2023).

Na Figura 2 é apresentada uma ilustração generalizada do conceito *MEC* baseada em Sabella *et al.* (2019). Uma câmera é ilustrada como um dispositivo de usuário, cuja finalidade é capturar imagens do ambiente e enviá-los a estação base mais próxima. A estação base pode alocar a requisição para ser processada no nó *MEC* ou encaminhá-la para a *Cloud*.

Figura 2 – Representação ilustrativa da arquitetura MEC.



Fonte: Elaborado pelo autor.

2.3 ORQUESTRAÇÃO DE CARGA

Em computação, a orquestração é um conceito que envolve a automatização e o gerenciamento de recursos, serviços e sistemas. A automatização se refere à realização de tarefas sem a intervenção humana. O gerenciamento consiste na manutenção e garantia de integridade da infraestrutura. A orquestração é um termo mais amplo que controla a execução de um fluxo de trabalho do início ao fim visando otimizar e automatizar a implantação dos serviços (DE SOUSA *et al.*, 2019).

Diversas áreas da computação costumam utilizar a orquestração, incluindo as áreas de redes de computadores e *Cloud Computing*. O orquestrador deve ser projetado para lidar com a dinamicidade do ambiente em que está inserido, devendo se adaptar às mudanças de contexto para atingir os objetivos ao qual foi atribuído. De modo geral, a orquestração se assemelha ao conceito de orquestra sinfônica, onde a música é dividida em várias partes atribuídas a diferentes instrumentos musicais (DE SOUSA *et al.*, 2019).

Em ambientes *Cloud*, o orquestrador deve lidar com recursos em diferentes localizações, devendo coordenar e selecionar os recursos conforme os objetivos e as políticas de negócios que foram estabelecidas (DE SOUSA *et al.*, 2019). Nesta dissertação, o termo orquestração de carga será utilizado para se referir à orquestração dos recursos de uma rede de nós *MEC* (ou *Fog*) utilizada para receber requisições de sistemas STR. Isto é, um orquestrador de carga deverá reconhecer a carga e a heterogeneidade dos recursos de cada nó, devendo encaminhar as requisições para o nó que esteja mais apto considerando o objetivo que deverá ser alcançado. Por exemplo, em Samir, El-Hennawy e El-Badawy (2022) o objetivo foi minimizar o consumo de energia de uma rede de nós MEC. Já em El-Nattat *et al.* (2021) o objetivo foi reduzir o makespan geral de uma rede de nós Fog.

Parte dos trabalhos que serão relatados no Capítulo 3, incluindo (SAMIR; EL-HENNAWY; EL-BADAWY, 2022) e (EL-NATTAT *et al.*, 2021), mencionados acima, não utilizam o termo orquestração de carga. Dentre os termos utilizados tem-se o balanceamento de carga, que corresponde a uma especialização do termo orquestração de carga fornecido nesta dissertação. O balanceamento de carga busca equilibrar a carga entre os nós (EL-NATTAT *et al.*, 2021; TAHMASEBI-POUYA; SARRAM; MOSTAFAVI, 2022), enquanto o termo orquestração de carga será utilizado para se referir a modelos em que a carga poderá (ou não) ser distribuída em pé de igualdade. O modelo proposto nesta dissertação é um exemplo de orquestrador de carga que não garante a igualdade de carga entre os nós.

Conforme a natureza dos trabalhos encontrados na literatura, nesta dissertação são definidos três tipos de orquestração de carga: orquestração de carga centralizada, centralizada por região e descentralizada. As principais diferenças entre os tipos de orquestração se dão pela localização e a quantidade de nós orquestradores. As características de cada modelo de orquestração são apresentadas a seguir.

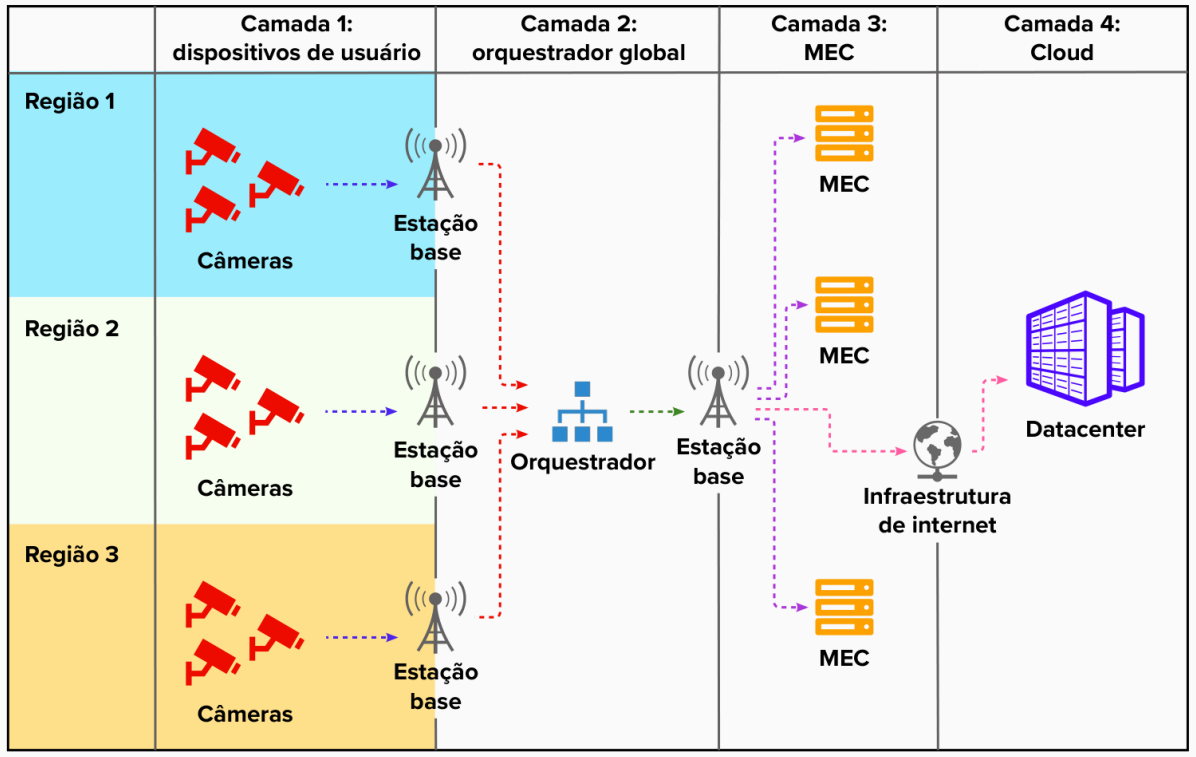
2.3.1 Orquestração de carga centralizada

Serão tipificados como orquestradores de carga centralizados as abordagens que atribuem apenas um único nó orquestrador na rede. Neste modelo, o nó orquestrador recebe as requisições de serviço, criadas pelos dispositivos de usuário, e as encaminha aos nós provedores que irão processá-las. O nó que processará uma determinada requisição será escolhido com base nos objetivos de cada orquestrador. Modelos como os propostos em Batista, Figueiredo e Prazeres (2022) e Batista *et al.* (2018) são exemplos de orquestradores de carga centralizados.

Na Figura 3 é apresentada uma ilustração generalizada dos modelos de orquestração de carga centralizados. Na primeira camada estão localizados os dispositivos de usuário, que devem capturar os dados e enviá-los ao orquestrador de carga da segunda camada. O orquestrador utilizará alguma estratégia para definir onde a carga

deverá ser processada. A carga poderá ser processada em algum nó da terceira camada, ou em data centers da *Cloud* (quarta camada). Os data centers da *Cloud* são opcionais, e servem como um apoio para os nós da terceira camada.

Figura 3 – Representação ilustrativa da arquitetura dos modelos de orquestração centralizados.



Fonte: Elaborado pelo autor.

Este modelo pode apresentar algumas limitações. Dependendo da localização dos dispositivos de usuário, nós provedores e do nó orquestrador, haverá uma maior influência do tráfego da rede sobre o tempo de resposta das requisições. A resiliência também pode ser reduzida, visto que a falha do nó orquestrador inviabilizará o funcionamento do sistema.

2.3.2 Orquestração de carga centralizada por região

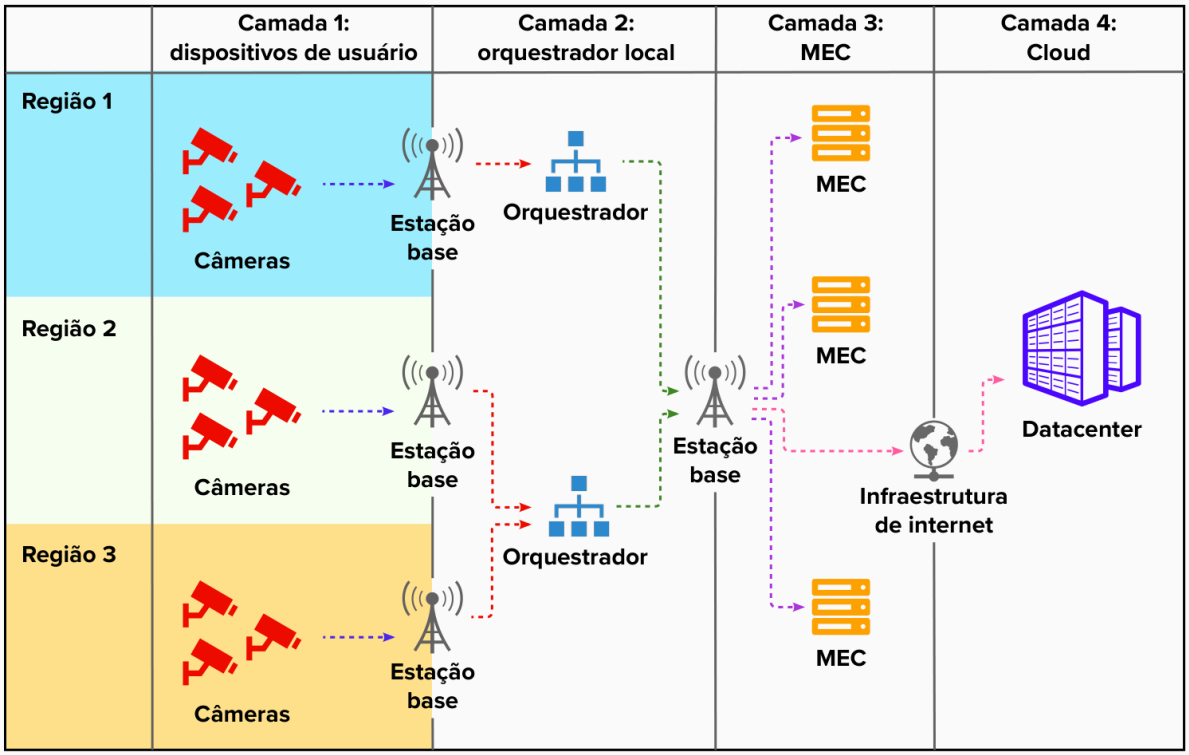
Nesta dissertação, um orquestrador de carga será considerado do tipo centralizado por região quando cada região possuir o seu próprio nó orquestrador. Por exemplo, o modelo proposto em Toczé e Nadjm-Tehrani (2019) considera um nó orquestrador por região, e cada orquestrador se comunica com os orquestradores das demais regiões. Em alguns trabalhos, como Pereira *et al.* (2020) e Malik *et al.* (2022), os nós orquestradores regionais são auxiliados por um nó orquestrador central.

Quando comparada à orquestração centralizada, que possui um nó central para atender todas as regiões, a centralização por região contribui para reduzir o tráfego

gerado na rede principal, reduzindo a sobrecarga da rede. A latência de resposta também deve ser reduzida, pois a tendência é que as requisições sejam atendidas por provedores mais próximos das fontes de dados. A resiliência também deve aumentar, visto que se o orquestrador de uma região falhar as demais regiões não serão afetadas.

Na Figura 4 é apresentada uma ilustração generalizada da orquestração de carga centralizada por região. Os orquestradores da camada 2 receberão as requisições, geradas pelos dispositivos de usuário da primeira camada, e utilizarão alguma estratégia para definir onde a carga deverá ser processada. Isto é, pelos nós da terceira camada, ou pelos data centers da *Cloud*, na quarta camada. Assim como na orquestração de carga centralizada, a quarta camada é opcional, e é utilizada conforme os objetivos do orquestrador de carga. Em resumo, este modelo resulta na redução da sobrecarga da rede, aumento da resiliência e menor latência.

Figura 4 – Representação ilustrativa da arquitetura dos modelos de orquestração centralizados por região.



Fonte: Elaborado pelo autor.

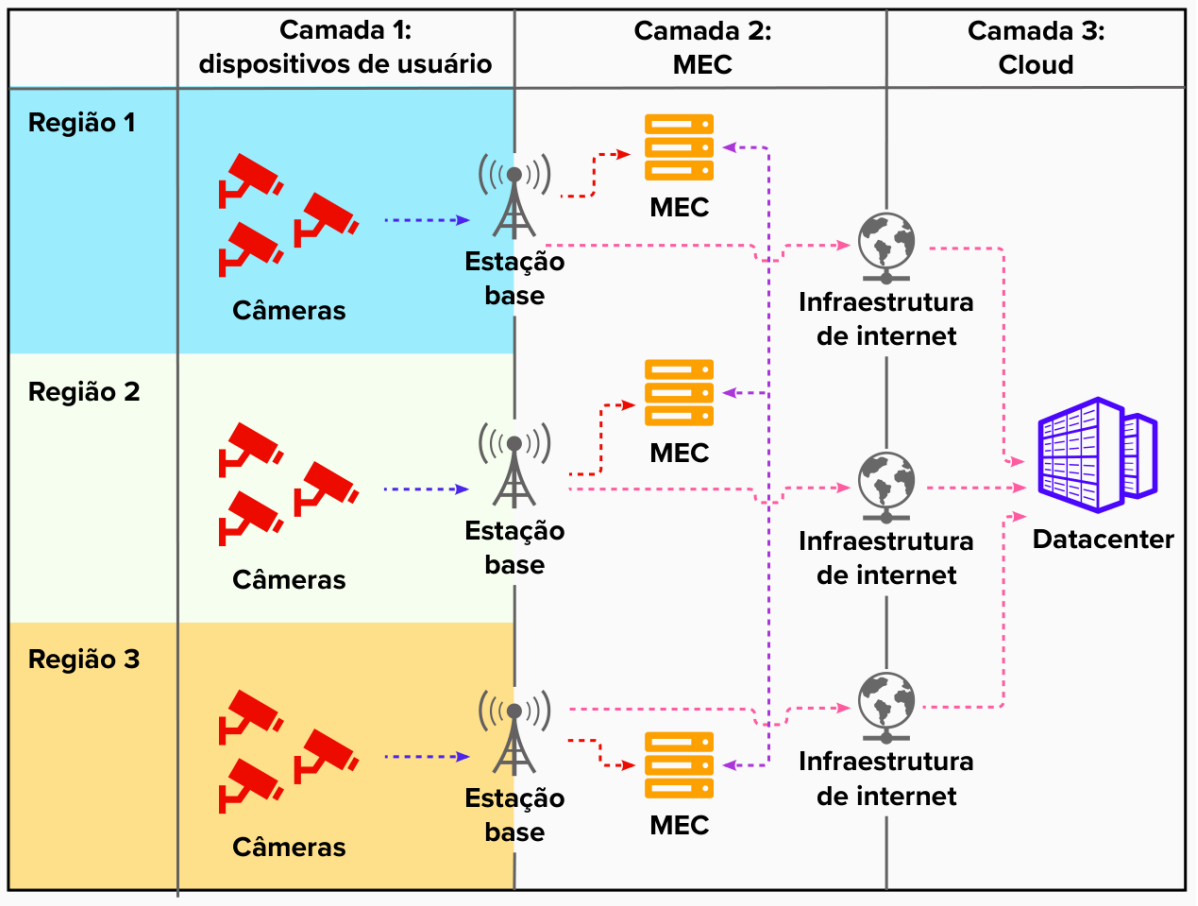
2.3.3 Orquestração de carga descentralizada

Os orquestradores de carga serão tipificados como descentralizados quando atribuírem um agente de orquestração a cada nó da rede. Nesse tipo de orquestração, os nós recebem individualmente as requisições de usuário. O agente de orquestração,

de cada nó, deve decidir entre a alocação da requisição na fila de espera ou encaminhá-la a outro nó da rede. O modelo proposto em Beraldi *et al.* (2020a,b,c) é um exemplo de orquestração de carga descentralizada, onde os agentes de orquestração encaminham as requisições sem conhecer a carga dos demais nós.

Na Figura 5 é apresentada uma ilustração generalizada da arquitetura dos modelos de orquestração de carga descentralizados. Quando comparado as arquiteturas apresentadas na Figura 3 e Figura 4, esta arquitetura possui apenas três camadas. Os dispositivos de usuário enviam os dados para os nós da segunda camada, que deve processar a requisição, encaminhá-la a outro nó da segunda camada, ou enviá-la para os data centers da *Cloud* (terceira camada). O uso da terceira camada é opcional, e dependerá dos objetivos de cada modelo de orquestração de carga.

Figura 5 – Representação ilustrativa da arquitetura dos modelos de orquestração descentralizados.



Fonte: Elaborado pelo autor.

Os orquestradores descentralizados possuem algumas vantagens em relação aos demais. Por exemplo, a sobrecarga da rede deve ser reduzida, visto que a distribuição da carga é feita localmente, sem a necessidade de envio de informações para um ponto central. Em termos de resiliência, se um nó falhar, apenas os usuários mais

próximos serão afetados. A latência tende a ser reduzida, pois as requisições não precisarão ser enviadas a um nó centralizado.

2.3.4 *Uncoordinated Access*

Cicconetti, Conti e Passarella (2020) introduzem uma arquitetura nomeada *Uncoordinated Access (UA)* para atuar em redes *MEC*. Apesar de ser uma arquitetura descentralizada, a arquitetura *UA* difere da arquitetura apresentada na Figura 5, pois na arquitetura *UA* são os dispositivos de usuário quem decidem onde a carga deverá ser processada. A arquitetura *UA* adiciona um nó central na rede, que deve conhecer o endereço de todos os nós *MEC* de uma determinada região. O nó central deve manter uma lista de nós *MEC*, em cada dispositivo de usuário, para que o dispositivo selecione um nó *MEC* para receber as requisições.

3 ESTADO DA ARTE

Neste capítulo é apresentado um estudo sobre 14 trabalhos encontrados na literatura. O estudo foi realizado para identificar um gap de pesquisa, o qual é considerado no modelo de orquestração de carga proposto no Capítulo 4. A obtenção dos trabalhos se deu mediante uma busca exploratória, que foi realizada considerando três bases de dados: IEEE Explorer, ScienceDirect e Springer. Para complementar a busca, também foi considerado o uso da ferramenta de buscas Google Scholar. Após a conclusão das buscas, os trabalhos foram comparados entre si através das seguintes classificações.

1. Prazos: definição de prazos para responder às requisições;
2. Fila: tipo de fila de espera para a alocação de requisições: *Fist In First Out (FIFO)*, prioritária (PR) ou preferencial (PF);
3. *Cloud*: uso da *Cloud* como um nó extra na rede para atuar em conjunto com os nós *MEC* (ou *Fog*): é utilizada (Sim), não é utilizada (Não), ou é opcional (Opcional);
4. Modelo: o modelo de orquestração de carga que os autores utilizaram: centralizado (C), centralizado por região (CR), centralizado por região com nó global (CR-G), descentralizado (D), ou descentralizado com consultor centralizado (D-C);
5. Paradigma: paradigma considerado no trabalho: *Fog*, *MEC* ou *Distributed Cloud (DCloud)*.

A comparação entre os trabalhos é apresentada no Quadro 1. É possível analisar que parte dos trabalhos utilizou a *Cloud* como um nó extra na rede, a fim de amenizar a sobrecarga dos nós *MEC* (ou *Fog*), porém a maioria dos trabalhos não considerou o uso da *Cloud*. Também é possível notar que, dentre os modelos encontrados, 8 trabalhos seguiram a arquitetura de orquestração de carga descentralizada, 2 trabalhos consideraram a arquitetura centralizada, e 4 trabalhos foram baseados na arquitetura centralizada por região. Dentre as abordagens descentralizadas, 1 abordagem considerou a utilização de um nó centralizado como um consultor sobre o estado da carga dos demais nós. Contudo, apesar da existência de um consultor, os autores propuseram que a decisão sobre onde a carga deve ser processada é tomada de maneira descentralizada.

Dentre os paradigmas, a *Fog* foi considerada em 10 dos 14 trabalhos, sendo a mais utilizada. Apenas 3 trabalhos consideraram o paradigma *MEC*, e 1 trabalho considerou o paradigma *Distributed Cloud*. Além disso, todos os trabalhos consideraram que os nós atuam em uma rede colaborativa, cujos nós podem estar distribuídos em diferentes localizações geográficas.

Quadro 1 – Comparativo entre os trabalhos encontrados na literatura e a abordagem proposta no Capítulo 4.

Artigo	**Prazos**	**Fila**	***Cloud***	**Modelo**	**Paradigma**
(EL-NATTAT *et al.*, 2021)	Sim	PR	Não	D	*Fog*
(TAHMASEBI-POUYA; SARRAM; MOSTAFAVI, 2022)	Não	*FIFO*	Sim	D	*Fog*
(GĄSIOR; SEREDYŃSKI, 2017)	Sim	PR	Não	D	*DCloud*
(BERALDI *et al.*, 2020a)	Não	*FIFO*	Não	D	*Fog*
(BERALDI *et al.*, 2020b)	Não	*FIFO*	Não	D	*Fog*
(BERALDI *et al.*, 2020c)	Não	*FIFO*	Não	D	*Fog*
(BATISTA *et al.*, 2018)	Não	*FIFO*	Não	C	*Fog*
(BATISTA; FIGUEIREDO; PRAZERES, 2022)	Não	*FIFO*	Sim	C	*Fog*
(CICCONETTI; CONTI; PASSARELLA, 2020)	Não	*FIFO*	Não	D-C	*MEC*
(TOCZÉ; NADJM-TEHRANI, 2019)	Sim	*FIFO*	Sim	CR	*MEC*
(SAMIR; EL-HENNAWY; EL-BADAWY, 2022)	Não	*FIFO*	Não	CR	*MEC*
(LIUTKEVIČIUS *et al.*, 2022)	Não	*FIFO*	Opcional	D	*Fog*
(PEREIRA *et al.*, 2020)	Não	PR	Sim	CR-G	*Fog*
(MALIK *et al.*, 2022)	Não	*FIFO*	Sim	CR-G	*Fog*
PROPOSTA	Sim	PF	Não	D	*MEC*

Fonte: Elaborado pelo autor.

Em relação à fila de espera de requisições, constatou-se que a maior parte dos trabalhos utilizou o modelo *FIFO* ou não mencionou o uso de uma fila de espera na abordagem proposta. A fim de realizar uma comparação entre todas as abordagens, os trabalhos que não mencionam o uso de uma fila de espera também foram agrupados na categoria de fila *FIFO*. Ao todo, 11 trabalhos foram marcados na categoria de fila de espera *FIFO*, enquanto 3 trabalhos propuseram ou utilizaram outro modelo de fila. Já a atribuição de prazos de resposta, para as requisições, foi considerada em 3 dos 14 trabalhos correlatos, porém apenas 2 dos 3 trabalhos utilizaram uma fila de requisições diferente da *FIFO*.

3.1 TRABALHOS CORRELATOS

Nesta seção é apresentado um resumo para cada trabalho encontrado na literatura. O resumo dos trabalhos se baseou em 7 perguntas, previamente elaboradas, e apresentadas a seguir.

1. Qual o nome do modelo de orquestração de carga (caso tenha sido definido)?
2. Em qual contexto o modelo foi utilizado?
3. Quais os problemas devem ser resolvidos? Isto é, qual o objetivo do que está sendo proposto?
4. Como a proposta funciona?
5. Como foram realizados os experimentos (caso tenham sido realizados)?
6. Com quais abordagens o modelo proposto foi comparado (caso tenha sido)?
7. Quais foram os resultados obtidos (caso tenham sido)?

3.1.1 *Performance Improvement of Fog Environment Using Deadline Based Scheduling Algorithm*

Em El-Nattat *et al.* (2021) é proposto um algoritmo de escalonamento de requisições nomeado *Enhanced Average Makespan (EAM)*. O objetivo foi melhorar o desempenho de uma rede de nós *Fog* ao prover serviços a sistemas STR. Para tanto, consideraram-se as restrições de prazos, pré-definidas pelos usuários, e a redução do *makespan* médio da rede. Para cada servidor *Fog* foi definido o uso de múltiplas máquinas virtuais (*virtual machines - VM*).

O algoritmo *EAM* classifica as requisições em grupos conforme o seu tipo, e posteriormente as organiza de forma descendente conforme os requisitos. A cada grupo é atribuída uma prioridade com base na disponibilidade das *VM*, e os grupos são percorridos conforme a sua prioridade. Ao percorrer um grupo, cada requisição do grupo deverá ser analisada.

A análise de cada requisição consiste em comparar a previsão do tempo de processamento com o prazo pré-definido pelo usuário. A análise é realizada para cada *VM*, e caso o tempo previsto para o processamento seja menor do que o prazo, em alguma *VM*, a requisição é processada por aquela *VM*. Caso contrário, a requisição é encaminhada a outro nó *Fog*. Apesar de considerar a *Cloud* na arquitetura de referência, o algoritmo não menciona o encaminhamento de requisições para a *Cloud*.

Os experimentos foram realizados em um ambiente simulado, no qual o modelo *EAM* foi comparado com o modelo *Modified Round Robin (MRR)* através de duas métricas de desempenho: tempo médio de resposta e o *makespan* médio da rede. Os resultados apontam para um melhor desempenho do modelo *EAM* quando o número de requisições é alto, enquanto o modelo *MRR* foi superior quando o número de requisições era baixo. As decisões de orquestração de carga são realizadas pelos próprios nós *Fog*. A carga é somente encaminhada quando o nó prever que não conseguirá cumprir o prazo da requisição, portanto não há a garantia de balanceamento da carga entre os nós.

3.1.2 *A Blind Load-Balancing Algorithm (BLBA) for Distributing Tasks in Fog Nodes*

Em Tahmasebi-Pouya, Sarram e Mostafavi (2022) é proposta uma arquitetura de referência, para ambientes *Fog*, e uma abordagem descentralizada de orquestração de carga, nomeada *Blind Load-Balancing Algorithm (BLBA)*. O artigo considera que os nós estão geograficamente distribuídos, e que o destino das requisições é o nó *Fog* mais próximo. O objetivo é reduzir o atraso de resposta dos serviços com custo computacional mínimo.

A arquitetura proposta é dividida em quatro camadas: *Internet of Things (IoT)*;

Fog; servidor *proxy*; *Cloud*. Cada nó *Fog* deve orquestrar a sua própria carga, decidindo se irá processar a requisição, encaminhá-la para um nó vizinho, ou encaminhá-la para a *Cloud*. O encaminhamento para a *Cloud* ocorre quando o atraso de processamento no nó *Fog*, e nos nós vizinhos, for maior do que o atraso gerado ao processar a requisição na *Cloud*. A tomada de decisão ocorre com base em informações obtidas através do uso de uma rede neural pré-treinada, sem que os nós conheçam o estado dos nós vizinhos ou da rede. Além da rede neural pré-treinada, o nó conhece apenas a sua própria carga de trabalho.

Os experimentos foram realizados para comparar o modelo proposto com outras três abordagens. O ambiente de experimentação utilizado foi o simulador *iFogSim*, e os resultados mostraram que o modelo proposto demonstrou ser mais eficaz para reduzir os tempos de processamento, e de resposta, de todas as requisições.

3.1.3 *A Sandpile cellular automata-based scheduler and load balancer*

Em Gąsior e Seredyński (2017) é considerado o contexto de ambientes *Distributed Cloud*. Os recursos computacionais dos nós, e a latência da comunicação da rede, são considerados ao alocar cada uma das requisições na fila de espera. O objetivo do trabalho é reduzir o tempo médio de resposta das requisições, e para isso busca-se o equilíbrio da carga entre os nós de toda a rede.

O modelo proposto é um orquestrador de carga descentralizado, no qual um agente, atuando em cada nó da rede, deve identificar a existência de um desequilíbrio de carga entre os nós. Sempre que um desequilíbrio de carga for detectado, o agente deve iniciar uma intensa troca de carga até que o equilíbrio seja estabelecido entre os nós. Para equilibrar a carga, os nós começam a busca entre os nós mais próximos, podendo alcançar todos os nós da rede mediante um intenso encaminhamento de requisições. Contudo, o encaminhamento de requisições também pode ocorrer sempre que for previsto o descumprimento de prazo de alguma requisição.

A fila de espera utilizada nesse artigo é do tipo *Earliest Deadline First (EDF)*. Os experimentos foram realizados no *software* Matlab, e as métricas de desempenho utilizadas foram baseadas no tempo de resposta das requisições. A abordagem proposta foi comparada com algoritmos clássicos da literatura: *First Come First Served (FCFS)*, *Shortest Job First (SJF)*, *Longest Job First (LJF)*, *Shortest Remaining Time First (SRTF)* e *Heterogeneous Earliest Finish Time (HEFT)*. Experimentos também foram realizados para comparar a abordagem proposta com o modelo proposto em um trabalho citado pelos autores. Os resultados mostraram que a abordagem proposta superou todas as abordagens com a qual foi comparada.

3.1.4 *A Random Walk based Load Balancing Algorithm for Fog Computing*

Em Beraldi *et al.* (2020a) são propostas duas abordagens de orquestração de carga descentralizada: *Sequential Forwarding Algorithm (SFA)* e *Adaptive Forwarding Algorithm (AFA)*. As abordagens também são apresentadas, pelos mesmos autores, em Beraldi *et al.* (2020b) e Beraldi *et al.* (2020c), no qual novos experimentos foram realizados. O contexto considerado, nos três trabalhos, é de orquestração de carga em redes de nós *Fog*, cujos nós são distribuídos em diferentes localidades geográficas. Os nós *Fog* devem fornecer serviços a dispositivos *IoT*, em *Smart Cities*, e as requisições são realizadas aos nós *Fog* que estão geograficamente mais próximos dos dispositivos *IoT*. Uma arquitetura é apresentada, composta por três camadas: camada de sensores; camada *Fog*; camada *Cloud*. Apesar de ser apresentada na arquitetura, a camada *Cloud* não é considerada no restante do trabalho.

Os modelos de orquestração de carga propostos são utilizados visando reduzir o tempo de espera das requisições. Um valor Θ é definido como um limite de alocação para cada um dos nós, de modo que, se o tempo de espera de uma nova requisição for superior a Θ, a requisição é encaminhada a outro nó da rede. O número de encaminhamentos é limitado a M, de modo que o último nó M a receber a requisição não poderá encaminhá-la a outro nó. Sendo assim, se a fila de espera (de tamanho Q) não estiver cheia, o nó M alocará a requisição na fila. Caso a fila estiver cheia, a requisição é descartada.

Para o algoritmo *SFA*, o valor Θ é calculado com base no número de requisições já alocadas na fila de espera. O algoritmo *AFA* é uma adaptação do *SFA*, cuja diferença está no cálculo do limite Θ, que é realizado da seguinte forma: $\Theta = (S * Q)/M$, onde S é o número de encaminhamentos já realizados para a requisição. O objetivo da adaptação é reduzir o número de encaminhamentos, visto que a medida que o número de encaminhamentos cresce, o limite Θ também aumenta, reduzindo a possibilidade da requisição ser encaminhada novamente.

Os experimentos foram realizados em um simulador baseado no *framework* Omnet++3, e o comportamento dos nós *Fog* foram baseados em um protótipo de dispositivo físico. As principais métricas consideradas foram a taxa de requisições descartadas, o número médio de encaminhamentos e o tempo de resposta das requisições. Os modelos de orquestração de carga propostos foram comparados com a não utilização de um orquestrador, e os resultados mostram que a utilização das abordagens propostas reduziram o tempo de espera e o número de requisições descartadas. Os resultados também foram comparados com os valores obtidos mediante um modelo matemático, apresentado pelos autores.

3.1.5 *Load Balancing in the Fog of Things Platforms Through Software-Defined Networking*

Em Batista *et al.* (2018) foi considerada uma rede de nós *Fog* para fornecer serviços a dispositivos *IoT*. O objetivo do trabalho foi realizar a orquestração de carga entre os nós *Fog*, com o intuito de reduzir o número de requisições perdidas, devido à sobrecarga dos nós, e reduzir o tempo de resposta.

O modelo de orquestração de carga proposto considera a existência de um nó centralizado, responsável por orquestrar a carga entre os nós *Fog* da rede. O nó central, isto é, o orquestrador, deve ser mantido informado sobre a disponibilidade de todos os nós *Fog*. O orquestrador realiza uma varredura constante sobre a disponibilidade dos nós, a fim de identificar quais os nós estão sobrecarregados e quais os nós estão disponíveis para receberem novas requisições. Nos casos de sobrecarga de parte dos nós, o orquestrador encaminha a carga para os nós mais disponíveis. Mesmo quando não há nós sobrecarregados, o orquestrador busca balancear a carga entre os nós.

Os experimentos foram realizados mediante um ambiente de experimentação simulado, baseado na ferramenta *Mininet*, e as métricas utilizadas foram o número de amostras perdidas e o tempo de resposta das requisições. A abordagem proposta foi comparada com a não utilização de um orquestrador de carga, e os resultados apontam que, para a maioria dos cenários testados, foi obtido um melhor desempenho quando o modelo proposto foi utilizado.

3.1.5.1 *Load balancing between fog and cloud in fog of things based platforms through software-defined networking*

Em Batista, Figueiredo e Prazeres (2022) foi estendido o trabalho proposto em Batista *et al.* (2018). A rede de nós *Fog* passou a encaminhar requisições para a *Cloud* quando os nós *Fog* não fossem capazes de processar as requisições. Além disto, foi entregue uma extensão do modelo que havia sido proposto em Batista *et al.* (2018). No novo modelo de orquestração de carga, o orquestrador verifica se existe ao menos 1 nó sobrecarregado e 1 nó disponível. Em caso positivo, uma lista de dispositivos *IoT*, atendidos pelo nó sobrecarregado, é obtida para ser repassada ao nó disponível. Os dispositivos *IoT* selecionados são os mais próximos do nó disponível.

O novo modelo também considera a possibilidade de um nó *Fog* ficar em estado de inatividade. Em casos em que os nós disponíveis não possam receber encaminhamentos, ou não existam nós disponíveis, um nó inativo deve retornar ao estado ativo e receber os encaminhamentos. Caso não existam nós inativos, as requisições são encaminhadas para a *Cloud*. Já no caso em que existam nós *Fog* disponíveis, não existam nós sobrecarregados, e parte das requisições estejam sendo enviadas para a *Cloud*, os nós disponíveis passam a receber a carga que era encaminhada para a

Cloud. Para que isso ocorra, também é necessário que os nós disponíveis não estejam com a rede congestionada.

A ferramenta *Mininet* foi utilizada para realizar os experimentos. O modelo foi comparado com as abordagens *Round Robin (RR)*, *Least Connection (LC)* e com a não utilização de um orquestrador de carga. As métricas de desempenho utilizadas foram o tempo de resposta, o número de amostras perdidas, e o tempo de atividade dos nós *Fog*. Os resultados apontam que, para a maior parte dos cenários, o modelo proposto foi superior aos demais.

3.1.6 *Uncoordinated access to serverless computing in MEC systems for IoT*

Em Cicconetti, Conti e Passarella (2020) é proposta uma nova arquitetura de orquestração de carga nomeada *Uncoordinated Access*, para atuar em redes *MEC*, a fim de fornecer serviços a dispositivos *IoT*. Um nó central é atribuído a cada região, devendo conhecer o endereço de todos os nós *MEC* da região onde está localizado. O nó centralizado é utilizado para manter uma lista de nós *MEC*, em cada dispositivo *IoT*, a fim de que o próprio dispositivo *IoT* selecione um dos nós *MEC* para receber as requisições.

Para selecionar o nó *MEC* o dispositivo *IoT* obtêm, periodicamente, um nó da lista fornecida pelo nó central, chamado de nó secundário. Em seguida, o nó secundário é comparado com o atual nó *MEC*, chamado de nó primário. A decisão de continuar com o nó primário, ou trocar pelo nó secundário, dependerá do tempo de resposta de uma requisição enviada a ambos os nós.

Para a realização dos experimentos foi utilizada a ferramenta *Mininet*. O modelo proposto é comparado com outras abordagens, como a *Round Robin (RR)*, através das seguintes métricas de desempenho: o tempo de resposta, o tráfego gerado na rede, e a carga média dos nós *MEC*. Após a análise dos resultados, os autores concluíram que a abordagem proposta é capaz de reduzir o tráfego de rede em mais de 50%, gera 10% a mais de carga nos nós *MEC*, e possui um tempo de resposta próximo, porém superior, ao das demais abordagens.

3.1.7 *ORCH: Distributed Orchestration Framework using Mobile Edge Devices*

Em Toczé e Nadjm-Tehrani (2019) é proposto um modelo de orquestração de carga descentralizado, nomeado ORCH. O modelo é utilizado para orquestrar a carga em uma rede de nós *MEC*, distribuídos geograficamente, a fim de fornecer serviços a dispositivos de cidades inteligentes. Tanto os dispositivos de usuários, quanto os nós *MEC*, podem ser do tipo estacionário ou móvel. As requisições são classificadas de acordo com três tipos: sensíveis, restritas e tolerante a prazos. Além disso, cada requisição possui um número diferente de instruções, portanto levam um tempo diferente para

serem concluídas. De modo geral, o objetivo do trabalho foi resolver dois problemas: onde posicionar a carga de trabalho, e onde posicionar os nós *MEC* móveis.

Uma arquitetura é apresentada, composta por três camadas: camada de dispositivos de usuário; camada de nós *MEC*; camada *Cloud*. Também foram definidas regiões compostas por dispositivos de usuário, da primeira camada, e nós *MEC*, da segunda camada. Cada região foi denominada como uma área de orquestração de carga, e para cada área de orquestração existe um nó *MEC* orquestrador. Qualquer nó *MEC* da região pode ser escolhido para ser temporariamente o orquestrador, inclusive os nós *MEC* móveis. Além disto, os nós *MEC* móveis podem mudar de região conforme for necessário.

O modelo de orquestração é composto por um componente de orquestração de área e um componente de orquestração de borda. O componente de orquestração de área é utilizado para posicionar os nós *MEC* móveis nas regiões que possuam uma maior carga de requisições sensíveis a prazos. Já o componente de orquestração de borda é utilizado para definir qual nó *MEC*, da região do orquestrador, deverá receber as requisições. O componente de borda é composto por outros três componentes, sendo o primeiro componente corresponde ao caracterizador de requisições, que deve definir a qual categoria pertence cada requisição: sensível, restrita ou tolerante a prazos. O segundo componente corresponde ao estimador de capacidade, que, periodicamente, deve consultar todos os nós *MEC* sobre os recursos disponíveis. Já ao terceiro componente é atribuída a tarefa de distribuir as requisições entre os nós *MEC*, a fim de alocar a requisição no nó *MEC* mais próximo do usuário.

Para realizar os experimentos, os autores criaram uma extensão do simulador EdgeCloudSim. Os experimentos foram realizados em três cenários. O primeiro cenário corresponde ao caso em que há apenas nós *MEC* estacionários. No segundo e terceiro cenário foram considerados a adição de nós *MEC* móveis. Como métricas de desempenho, os autores utilizaram duas métricas baseadas na taxa de requisições respondidas com sucesso. A diferença é que a primeira métrica considera a taxa de sucesso por tipo de requisição, enquanto a segunda métrica considera a taxa de sucesso para cada tipo de nó *MEC*.

Para o caso de atribuição de requisições aos nós *MEC*, o modelo proposto foi comparado com a estratégia *First Fit (FF)*. Já para o caso de distribuição dos nós *MEC* móveis, entre as regiões, o modelo proposto foi comparado com outros três modelos apresentados pelos autores. Os resultados apontam que, enquanto a estratégia *FF* gerou 57% de perda de prazos para requisições sensíveis a prazos, o modelo proposto gerou 0% de perdas. O modelo proposto também foi superior aos outros três modelos apresentados pelos autores.

3.1.8 *Orchestration of MEC Computation Jobs and Energy Consumption Challenges in 5G and Beyond*

Em Samir, El-Hennawy e El-Badawy (2022) é proposta uma abordagem de orquestração de carga centralizada por região. O modelo de orquestração é utilizado em uma rede de nós *MEC* colaborativa, e cada nó *MEC* possui uma bateria recarregada por energia solar ou pela rede pública. O objetivo do trabalho é reduzir os custos de energia gerados pela rede.

Uma arquitetura é apresentada, composta por duas camadas. A primeira camada corresponde aos nós *MEC* localizados mais próximo dos usuários, os quais serão responsáveis por processar as requisições. Já na segunda camada estão os nós *MEC* que farão a orquestração da carga da região. Os orquestradores devem se comunicar entre si, a fim de compartilhar a carga entre as regiões, e possuírem uma réplica, para garantir a disponibilidade dos serviços na região onde está inserido. Os experimentos foram realizados por meio de simulações no *software Matlab*, e o modelo proposto foi comparado com a versão apresentada em um trabalho anterior. Os resultados apontam uma melhoria de até 50% nos custos com energia.

3.1.9 *Distributed Agent-Based Orchestrator Model for Fog Computing*

Em Liutkevičius *et al.* (2022) é proposto um modelo de orquestração de carga descentralizado para atuar em uma rede de nós *Fog* geo-distribuídos. Os nós *Fog* devem fornecer serviços de tempo real, a dispositivos de usuários móveis, que enviarão requisições ao nó *Fog* mais próximo. O objetivo do trabalho foi garantir uma melhor resiliência, escalabilidade e desempenho de serviços de tempo real, considerando os recursos computacionais de cada nó *Fog*, os diferentes níveis de bateria e segurança, e a possibilidade de falha dos nós. O uso da *Cloud* é opcional, sendo tratada como um nó extra na rede.

Ao receber uma requisição, diretamente do dispositivo de usuário, o nó *Fog* irá identificar o tipo da requisição e os recursos necessários para processá-la. Em seguida, o nó *Fog* irá identificar qual é o melhor nó para processar a requisição, considerando ele próprio e os nós vizinhos. A seleção do nó é uma decisão baseada em um algoritmo proposto em um trabalho anterior, o qual necessita que os nós enviem, constantemente, o seu estado atual aos demais nós da rede.

A falha de algum nó da rede deve ser considerada ao decidir onde a carga será processada. Para detectar a falha de um nó, os demais nós irão identificar quando foi recebida a última mensagem do nó que falhou. Isto é, se a última mensagem foi recebida em um período considerado antigo, o nó é classificado como indisponível. Além da integridade, o nó também deve possuir recursos disponíveis, tanto em relação à carga quanto ao nível da bateria. Também é necessário que o nó selecionado possua

o serviço requisitado, devendo ativá-lo, caso esteja desativado, ou até baixar da *Cloud*, caso ainda não esteja instalado.

Os experimentos foram realizados em um ambiente de experimentação baseado em *hardware* real. Dois *Raspberry Pi* foram utilizados como nós *Fog*, e dois dispositivos *IoT* serviram como dispositivos de usuário. Todos os dispositivos foram conectados entre si a partir de uma rede local, por meio de um roteador. Um computador pessoal, e outro virtual, também foram utilizados em uma segunda rede, a fim de comparar o desempenho em relação ao obtido com os *Raspberry Pi*.

Os resultados apontam para uma maior sobrecarga computacional do modelo proposto, quando comparado a modelos centralizados. Contudo, os autores concluem que a sobrecarga é aceitável, e destacam a vantagem do modelo proposto ser mais resiliente do que os demais. Isto é, a rede é capaz de se adaptar mais facilmente quando comparado ao uso de um nó central, pois a falha do nó central implicará na indisponibilidade dos serviços. Contudo, os autores não apresentam resultados de comparação do modelo proposto com outros modelos, se limitando ao desempenho apenas do modelo proposto.

3.1.10 *Increasing the efficiency of Fog Nodes through of Priority-based Load Balancing*

Em Pereira *et al.* (2020) é proposta a orquestração de carga centralizada por região, entre nós *Fog* geo-distribuídos, e com um nó orquestrador global. As requisições de serviços são realizadas por dispositivos *IoT* e são classificadas como de alta ou baixa prioridade. O objetivo foi reduzir o tempo de resposta das para ambos os tipos de requisições.

Os autores consideraram uma arquitetura dividida em três camadas. A primeira e a segunda camada estão na rede local, enquanto a terceira camada engloba o restante da *internet*. A primeira camada corresponde aos dispositivos *IoT*, que devem enviar requisições de serviços aos nós *Fog* da segunda camada. Na segunda camada também existe um nó central, para cada rede local, que deve orquestrar a carga entre os nós *Fog* locais. Por fim, na terceira camada estão a *Cloud* e um orquestrador de carga global, que deve atuar em conjunto com os orquestradores de carga de cada localidade.

A alocação das requisições deve considerar a prioridade de cada requisição: alta ou baixa. As requisições de baixa prioridade devem ser alocadas no fim da fila de requisições, enquanto as de alta prioridade são alocadas no início. Os nós com menor carga devem atender às requisições de alta prioridade, enquanto as demais requisições são atendidas pelos nós com maior carga. No caso de todos os nós estarem sobrecarregados, existe a possibilidade do nó central encaminhar as requisições para a *Cloud*.

Os experimentos foram conduzidos em *hardware* real, através da plataforma *Altix UV 2000*. Os resultados apontam que o modelo proposto obteve um melhor desempenho quando comparado ao modelo *Round Robin* e ao caso em que não há orquestração de carga.

3.1.11 *Intelligent Load-Balancing Framework for Fog-Enabled Communication in Healthcare*

Em (MALIK *et al.*, 2022) é considerado o fornecimento de serviços para dispositivos *IoT* utilizados na área da saúde. As requisições são enviadas para a camada *Fog*, formada por nós geo-distribuídos, em que cada nó *Fog* é constituído por um conjunto de *VM*. O objetivo é reduzir o tempo de resposta gerado para as requisições sensíveis a atrasos. Uma arquitetura é apresentada, composta por três camadas: camada *Cloud*; camada *Fog*; camada de sensores e sistemas de saúde. A camada *Fog* é composta por *clusteres* de nós *Fog*, e cada nó *Fog* possui múltiplas *VM* e um orquestrador de *VM*.

Para orquestrar a carga entre os diferentes *clusters*, de cada região, é designado um nó central para a camada *Fog*. A orquestração de carga é realizada pelo orquestrador global, que deve selecionar a melhor *VM* dentre as escolhidas pelos orquestradores locais de *VM*. O critério de escolha da *VM* é feito através da divisão do número de requisições atendidas com sucesso, de cada *VM*, pelo número de requisições designadas para a *VM*. Os autores não realizaram experimentos para validar a proposta.

3.2 GAP DE PESQUISA

Conforme explicado no Capítulo 2, os modelos de orquestração de carga descentralizados apresentam algumas vantagens em relação aos demais, como a redução da sobrecarga e a latência da rede. Conforme apresentado no Quadro 1, a arquitetura descentralizada é utilizada em 8 dos 14 trabalhos que foram citados nesta dissertação. No entanto, após uma análise desses 8 trabalhos descentralizados, concluiu-se que existe um *gap* de pesquisa gerado pela ausência de trabalhos que considerem, ao mesmo tempo, resolver o problema do cumprimento de prazos de resposta e a redução do tráfego de rede.

Apesar de nenhum dos trabalhos conseguirem cobrir todos os objetivos que foram estabelecidos nesta dissertação, os trabalhos El-Nattat *et al.* (2021) e Beraldi *et al.* (2020a,b,c) possuem objetivos muito próximos. O problema de Beraldi *et al.* (2020a,b,c) é a ausência de uma fila de requisições apropriada para os sistemas STR, pois o modelo utilizado não considera os prazos de resposta das requisições. Em El-Nattat *et al.* (2021) os prazos de resposta são considerados, porém, os autores não

consideraram a preferência, mas sim uma prioridade no atendimento das requisições com menor prazo. Isto é, El-Nattat *et al.* (2021) atribuem uma classificação para as requisições, de modo a priorizar um determinado grupo em relação aos demais.

De modo geral, a maior parte dos trabalhos não considera organizar a fila de espera para cumprir o prazo das requisições, e os trabalhos que consideram uma fila diferente da fila *FIFO* não buscam resolver o problema do tráfego de rede. De igual modo, os trabalhos que resolvem o problema de tráfego de rede não lidam com a maximização de requisições com prazo cumprido.

4 PROPOSTA DE APRIMORAMENTO DE UM ORQUESTRADOR DE CARGA DESCENTRALIZADO

A proposta deste trabalho é aprimorar o orquestrador de carga proposto em Beraldi *et al.* (2020a,b,c), nomeado *Sequential Forwarding Algorithm (SFA)*. A decisão pela escolha do *SFA* foi baseada em dois aspectos. O primeiro aspecto é a proximidade dos objetivos de Beraldi *et al.* (2020a,b,c) com aqueles estabelecidos nesta dissertação de mestrado. O segundo aspecto está relacionado com questões de praticidade na realização dos experimentos. Isto é, os objetivos de El-Nattat *et al.* (2021) estão ainda mais próximos daqueles que foram estabelecidos nesta dissertação, porém questões como o uso de máquinas virtuais implicariam em questões que fogem do escopo deste trabalho.

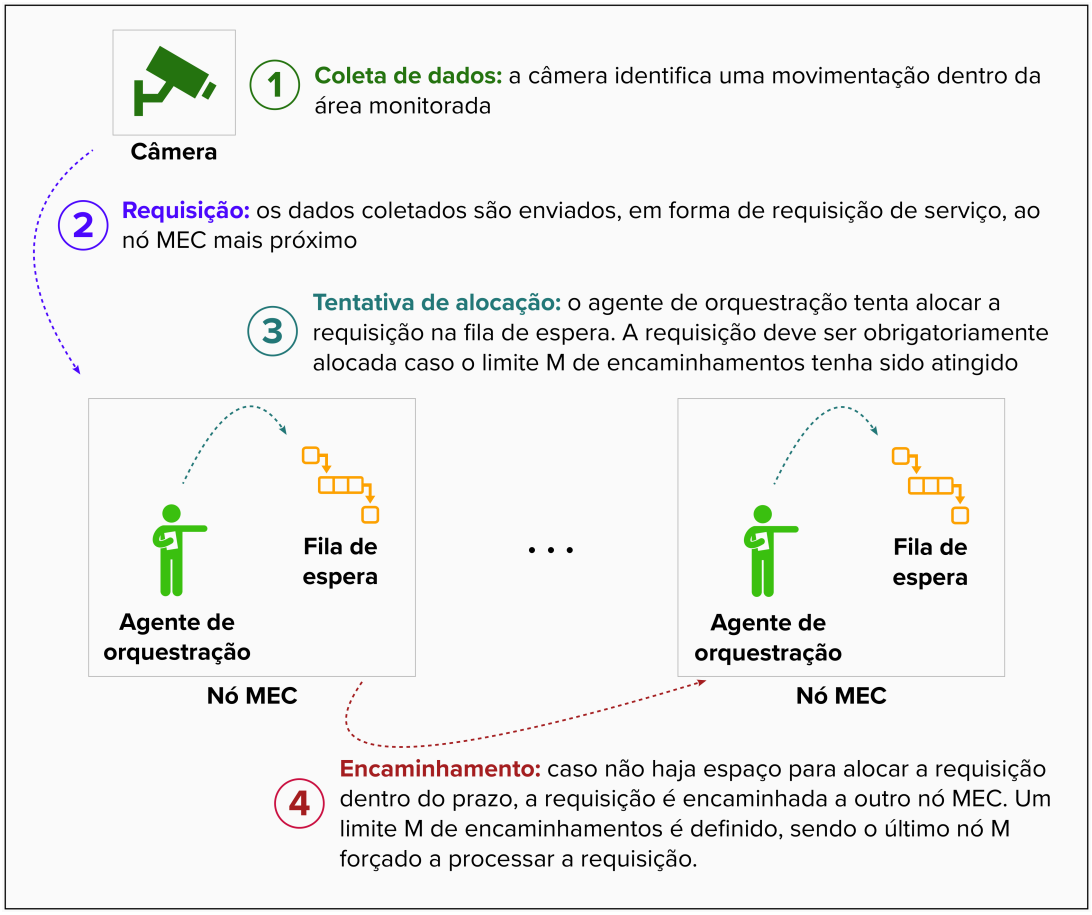
4.1 ADAPTAÇÃO DO MODELO *SEQUENTIAL FORWARDING ALGORITHM (SFA)*

Conforme descrito no Capítulo 3, o modelo *SFA* é um orquestrador descentralizado, que consiste em orquestrar a carga destinada a uma rede de nós *Fog*. Os nós *Fog* devem receber as requisições individualmente, e mediante um agente local, as requisições são alocadas em uma fila de espera. No momento da alocação, o agente fará a verificação da carga do nó *Fog*. Caso identifique que o nó *Fog* será capaz de cumprir o prazo da requisição, o agente aloca a requisição na fila de espera. Caso contrário, o agente encaminha a requisição a outro nó *Fog*. O encaminhamento de uma requisição é limitado a um número *M* de vezes, sendo o último nó *M* forçado a processar a requisição.

Nesta dissertação será considerado o uso do *SFA* para uma rede de nós *MEC*. A modificação proposta é referente ao gerenciamento da fila de espera, que na versão original é baseado no algoritmo *First In, First Out (FIFO)*. Isto é, as novas requisições são alocadas no fim da fila de espera. Nesta dissertação é proposta uma versão alternativa de fila de espera, nomeada Fila de Espera Preferencial (FEP).

Na Figura 6 são ilustrados os passos realizados durante o processo de orquestração de carga. O primeiro passo ocorre dentro das câmeras, que devem detectar uma movimentação dentro do ângulo de visão. O passo 2 corresponde ao envio de requisições de serviço, contendo as imagens capturadas, ao nó *MEC* mais próximo. O terceiro passo é quando o agente de orquestração, localizado no nó *MEC*, tentará alocar a requisição na fila de espera. O passo 4 acontece a partir da falha do passo 3, correspondendo ao encaminhamento da requisição a outro nó *MEC* da rede. Por fim, o passo 4 é repetido até que um nó *MEC* seja capaz de cumprir o prazo da requisição ou o limite *M* seja alcançado.

Figura 6 – Representação ilustrativa do processo de orquestração de carga.



Fonte: Elaborado pelo autor.

4.2 FILA DE ESPERA PREFERENCIAL (FEP): CONCEITUAÇÃO

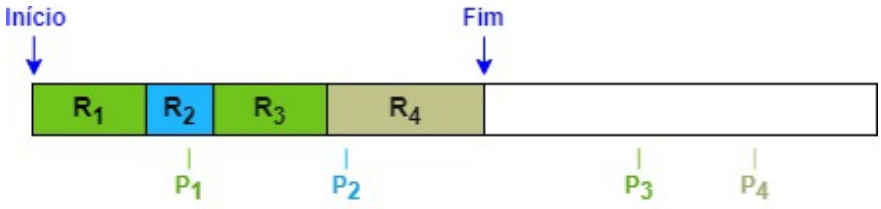
A fila de espera proposta neste trabalho é nomeada de Fila de Espera Preferencial (FEP), sendo definida nesta dissertação como um modelo que busca reorganizar as requisições para maximizar o número de requisições respondidas dentro do prazo.

Primeiramente, ao receber uma nova requisição R_{new}, o alocador fará uma tentativa de alocação de R_{new} no fim da fila. Caso a alocação no fim da fila resulte na violação do prazo P_{new}, o algoritmo de alocação buscará alocar R_{new} na frente de requisições já alocadas anteriormente. A condição para privilegiar R_{new} é não gerar a violação do prazo das demais requisições. A posição escolhida deve ser, dentre aquelas em que o prazo de R_{new} seja respeitado, a mais próxima do fim da fila.

A Figura 7 ilustra a fila preferencial na visão do escalonador. A primeira requisição a ser processada é a requisição mais à esquerda, enquanto a última requisição a ser processada está alocada mais à direita. Cada requisição R_i possui um prazo de resposta P_i e aponta para a requisição seguinte.

A visão do alocador é diferente, conforme ilustrado na Figura 8. Conforme o

Figura 7 – Representação ilustrativa, da fila de espera, na visão do algoritmo de escalonamento.



Fonte: Elaborado pelo autor.

tamanho do prazo P_i, e do tempo de processamento de R_i, é possível identificar o momento máximo do relógio em que a requisição poderá iniciar o processamento. Com base no tempo máximo para o início e término do processamento da requisição, o alocador consegue visualizar espaços de tempo entre as requisições já alocadas.

Na Figura 8a é realizada a tentativa de alocar uma nova requisição R_5 (R_{new}) no fim da fila. O alocador identifica que o prazo P_5 seria extrapolado caso a alocação fosse feita após R_4. Na sequência, ilustrado pela Figura 8b, o alocador identifica que não existe espaço de tempo entre R_3 e R_4, porém existe entre R_2 e R_3. O alocador identifica que o espaço temporal entre o término do processamento de R_2 e o início do processamento de R_3 não é suficiente para alocar R_5. No entanto, existe uma fatia de tempo que pode ser considerada para a alocação.

Conforme ilustrado pela Figura 8c, o alocador considera o espaço temporal entre R_2 e R_3 e continua buscando por mais espaços temporais. Um espaço temporal entre R_1 e R_2 é encontrado e é maior do que o necessário. Portanto, na Figura 8d o alocador aloca R_5 entre R_2 e R_3 e os espaços disponíveis entre R_1 e R_2, assim como os espaços entre R_2 e R_3, são reduzidos conforme o tempo de processamento e o prazo de R_5.

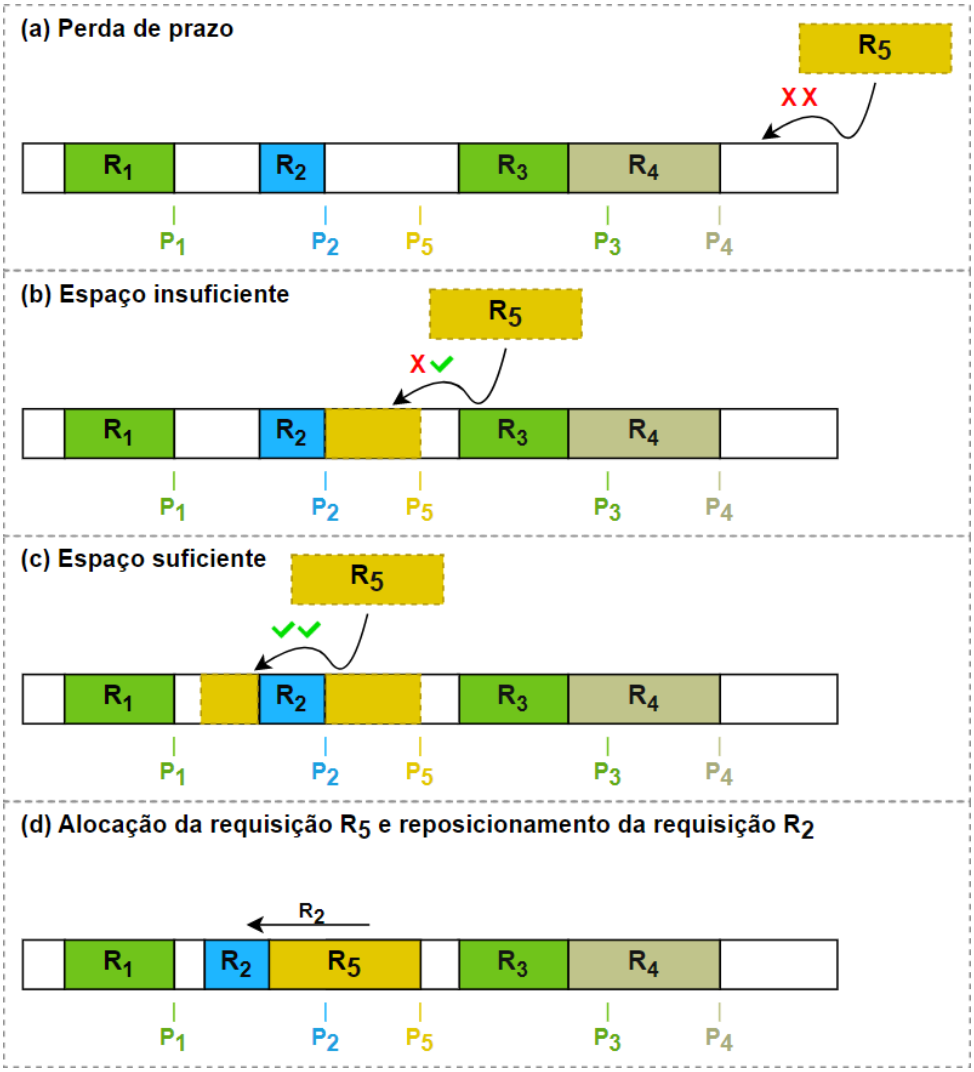
Apesar da requisição R_2 ter sido movida para frente, na Figura 8d o prazo P_2 não acompanhou o bloco R_2. Isto é, o prazo para responder R_2 continua sendo o mesmo, porém não existe espaço suficiente antes de R_2 para haver a alocação de novas requisições que façam R_2 terminar no prazo limite P_2.

4.2.1 Pior caso

O pior caso é alcançado quando o algoritmo de alocação identifica que não existe a possibilidade de alocar a nova requisição R_{new} sem que R_{new}, ou as demais requisições, percam o prazo de resposta. A Figura 9 ilustra a fila, na visão do alocador, para o pior caso. Neste exemplo, a nova requisição R_{new} é a requisição R_6.

No exemplo da Figura 9, o algoritmo realiza uma busca, entre a Figura 9a e Figura 9d, por espaços de tempo que estejam disponíveis para a alocação de R_6. Dois

Figura 8 – Representação ilustrativa, da fila de espera, na visão do algoritmo de alocação.

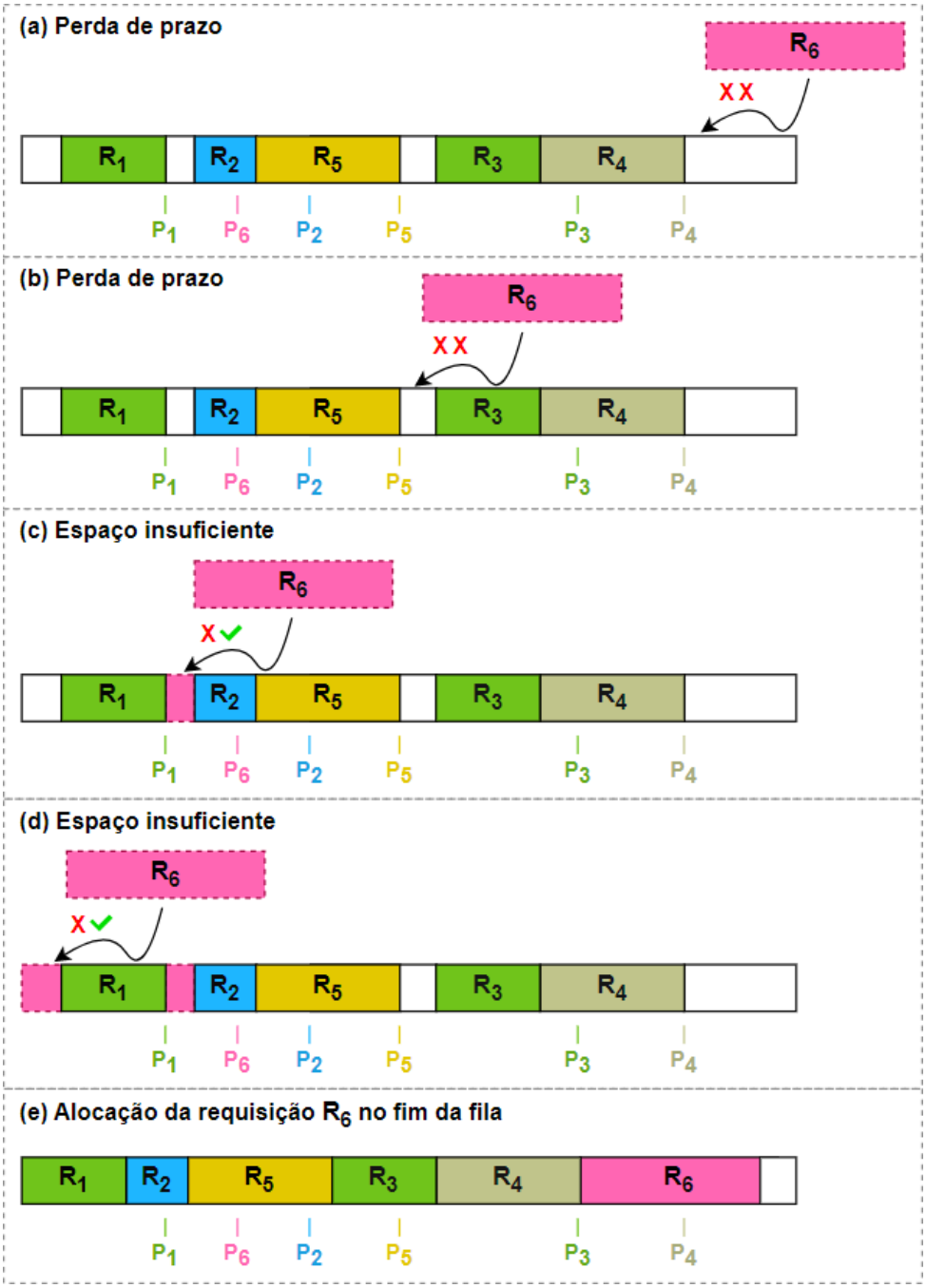


Fonte: Elaborado pelo autor.

espaços de tempo foram encontrados, sendo o primeiro entre R_1 e R_2, e posteriormente entre o início da fila e R_1. Apesar dos espaços encontrados, mesmo quando realizada a soma dos espaços não é suficiente para alocar a nova requisição R_6. Isto é, o tempo de processamento de R_6 é maior do que a soma dos espaços disponíveis. Portanto, o alocador realiza a alocação de R_6 no fim da fila e remove todos os espaços de tempo que estão disponíveis na fila.

O pior caso da fila preferencial é equivalente ao comportamento da fila *FIFO*, pois a falta de espaços de tempo impede, momentaneamente, a realocação de novas requisições, forçando-as a serem alocadas no fim da fila. No entanto, é possível que a remoção dos espaços, e a alocação forçada de R_{new} no fim da fila, não seja caracterizado como o pior caso da fila preferencial. Isto é, o pior caso se caracteriza pela perca de prazo da requisição, que somente ocorrerá caso haja, simultaneamente, a remoção

Figura 9 – Representação ilustrativa do pior caso, da fila de espera, na visão do algoritmo de alocação.



Fonte: Elaborado pelo autor.

de todos os espaços disponíveis, e a alocação no fim da fila. Contudo, é possível que essas duas condições ocorram sem que haja a perca de prazo. O exemplo da Figura 9 se caracteriza como pior somente porque a requisição R_6 (R_{new}) perdeu o prazo.

4.3 FILA DE ESPERA PREFERENCIAL (FEP): ALGORITMOS

Nesta seção é descrita a parte de implementação da fila FEP, a qual é composta pela união de 5 algoritmos. Detalhes teóricos sobre o funcionamento da fila FEP são descritos na seção anterior, e, portanto, não serão discutidos nesta seção. De modo geral, os passos realizados pelos algoritmos se resumem na busca por espaços de tempo disponíveis (e úteis), e a alocação (ou não) da nova requisição.

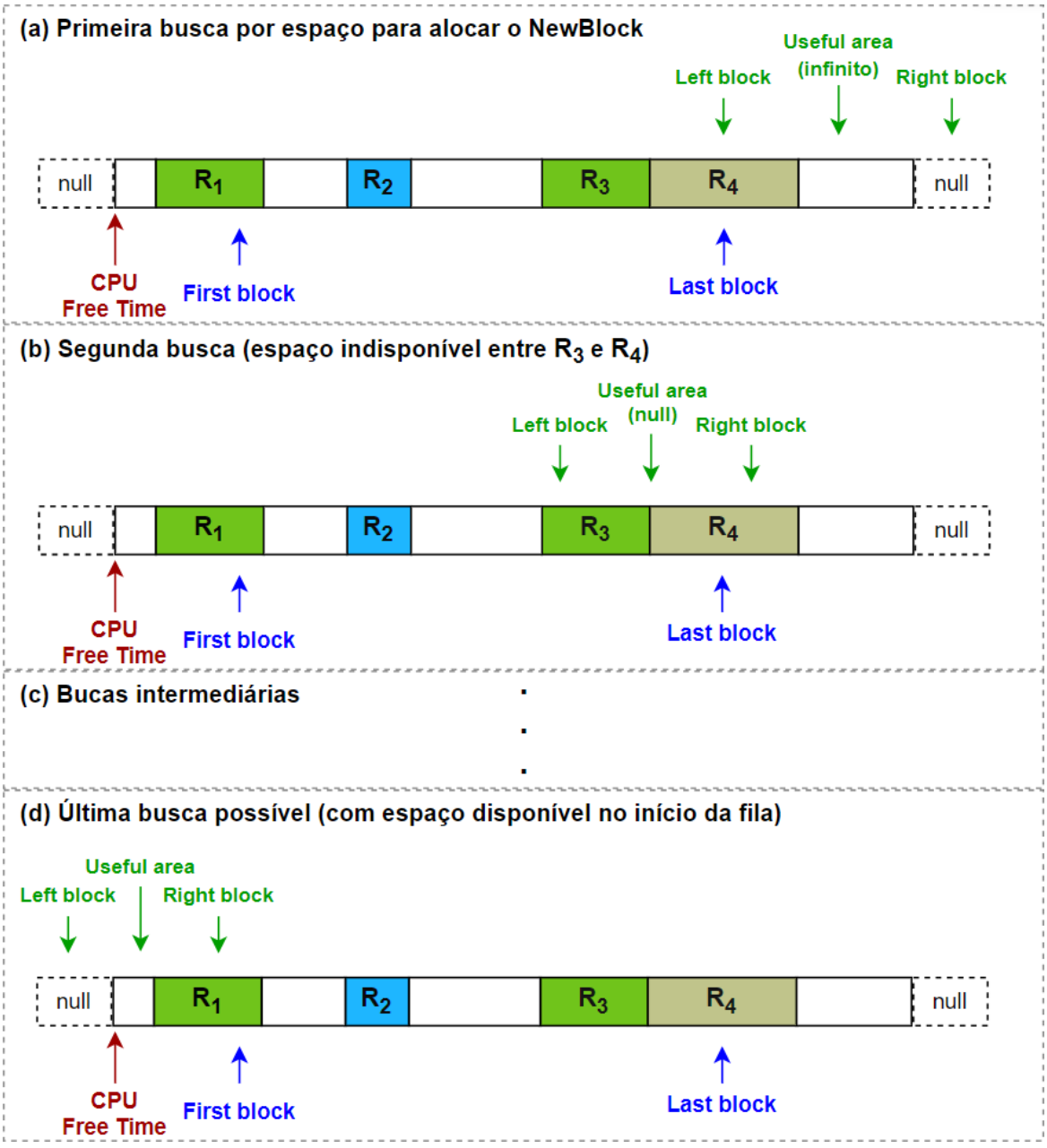
Na Figura 10 é apresentada uma ilustração que resume como os algoritmos se comportam. Ao todo, 6 variáveis são responsáveis por conduzirem as iterações realizadas durante a busca por espaços disponíveis: *cpuFreeTime*, *firstBlock*, *lastBlock*, *leftBlock*, *usefulArea* e *rightBlock*. A definição dessas, e outras variáveis importantes, são apresentadas no Quadro 2.

Quadro 2 – Lista dos tipos e variáveis utilizados pelos algoritmos da fila FEP.

Tipos e variáveis	**Significado**
request	Uma requisição a ser alocada na fila FEP
Block	Uma estrutura que representa um bloco ocupado, ou livre, na fila FEP
RequestBlock	Uma especialização da estrutura *Block* utilizada para encapsular uma *request* na fila FEP
newBlock	Uma instância do tipo *RequestBlock* utilizada para encapsular a nova *request*
leftBlock	Uma instância do tipo *RequestBlock* que, em uma determinada iteração do algoritmo de buscas, está encapsulando a requisição à esquerda
rightBlock	Uma instância do tipo *RequestBlock* que, em uma determinada iteração do algoritmo de buscas, está encapsulando a requisição à direita
usefulArea	Uma instância do tipo *Block* utilizada para representar um espaço de tempo disponível entre o *leftBlock* e *rightBlock*
cpuFreeTime	Momento temporal em que a *CPU* ficará disponível para processar novas *requests*
forcedPush	Condicional utilizada para forçar a alocação da nova *request*, mesmo que o pior caso tenha sido atingido
spaceNeeded	Espaço que ainda está faltando para alocar *newBlock*
hasRightSpace	Valor booleano utilizado para afirmar, ou negar, se as iterações anteriores encontraram algum espaço disponível que seja útil para a alocação de *newBlock*
end	Um valor inteiro correspondente ao fim de *newBlock*, caso seja alocado

Fonte: Elaborado pelo autor.

Figura 10 – Representação ilustrativa das buscas por espaços disponíveis, realizadas pelos algoritmos da fila FEP, durante a tentativa de alocação de uma nova requisição.



Fonte: Elaborado pelo autor.

Conforme ilustrado na Figura 10a, a primeira busca considera a tentativa de alocar a nova requisição após a última requisição alocada (*lastBlock*). Para tanto, considera-se que, caso a nova requisição seja alocada, a requisição que será vizinha, pela esquerda (*leftBlock*), é a última requisição alocada (*lastBlock*). Além disso, considerando que não há outra requisição após *lastBlock*, a nova requisição não teria uma vizinha pela direita (*rightBlock*).

Caso a alocação não ocorra na primeira iteração, o algoritmo segue buscando

por espaços disponíveis, conforme ilustrado pela Figura 10b. Contudo, no caso ilustrado não há espaço disponível entre as requisições *leftBlock* e *rightBlock*, portanto o algoritmo segue realizando novas buscas, pois ainda existem requisições na fila. A última busca possível é ilustrada pelo exemplo da Figura 10d, na qual o *leftBlock* será nulo, e o *rightBlock* será equivalente à primeira requisição da fila (*firstBlock*).

A seguir é apresentada uma explicação detalhada sobre cada um dos 5 algoritmos implementados.

4.3.1 Algoritmo *push_request*

O principal algoritmo é o *push_request* (Algoritmo 1). O retorno da função é um valor *true*, caso a requisição tenha sido alocada, ou *false*, caso contrário. O algoritmo inicia encapsulando a nova requisição em uma estrutura *RequestBlock*, cuja instancia é nomeada *newBlock* (linha 2). Considerando que esta será a primeira iteração realizada para buscar espaços de tempo disponíveis, o *leftBlock* recebe o valor de *lastBlock* (linha 3). Já a requisição à direita (*rightBlock*) será nula (linha 4), pois não há outras requisições alocadas após a última (*lastBlock*).

Algorithm 1: Adicionar uma requisição na fila de espera.

```
1  function push_request(request, cpuFreeTime, forcedPush): boolean is
2      newBlock = RequestBlock(request)
3      leftBlock = this.lastBlock
4      rightBlock = null
5      spaceNeeded = newBlock.get_size()
6      hasRightSpace = false
7      status = search_alloc_space(leftBlock, newBlock, rightBlock, spaceNeeded,
         hasRightSpace, cpuFreeTime, forcedPush)
8      if status == true then
9          return true
10     end
11     if forcedPush == false then
12         return false
13     end
14     if is_empty() == true then
15         start = cpuFreeTime
16     else
17         start = leftBlock.get_end()
18     end
19     end = start + spaceNeeded
20     newBlock.set_end(end)
21     alloc_request(leftBlock, newBlock, rightBlock)
22     return true
23 end
```

Na linha 5 é definido o espaço de tempo necessário para alocar a nova requi-

sição. Já na linha 6 é definido que nenhum espaço suficiente foi encontrado antes da primeira iteração, que ocorre a partir da linha 7, na qual é dado início a busca por espaços disponíveis (e úteis) para alocar *newBlock*.

O algoritmo *search_alloc_space* (linha 7) irá, primeiramente, verificar se a hipótese de que há espaço disponível no fim da fila é verdadeira. Se houver espaço, retorna-se *true*. Caso não haja espaço suficiente, o algoritmo fará novas buscas, como será descrito posteriormente. Caso chegue ao fim de todas as buscas possíveis, e não haja espaço suficiente disponível, a variável *status* receberá o valor *false*.

Nas linhas 8 a 10 é verificado se o retorno da linha 7 é *true*. Caso positivo, a alocação foi bem sucedida e deve-se retornar *true*. Caso contrário, verifica-se, entre as linhas 11 a 13, se a alocação é obrigatória. Caso a alocação não seja obrigatória, retorna-se *false*. Caso contrário, as linhas 14 a 22 são executadas para alocar a requisição no final da fila.

Conforme descrito posteriormente, se a alocação for obrigatória, e não houver espaço suficiente disponível, os espaços insuficientes disponíveis, entre a primeira e última requisição, já serão removidos na linha 7.

4.3.2 Algoritmo *search_alloc_space*

O Algoritmo 2 é utilizado pelo Algoritmo 1 e é responsável por buscar um espaço disponível para alocar o *newBlock*. Caso encontre espaço suficiente disponível, o algoritmo deve invocar o Algoritmo 4 (linha 6) que é responsável pela alocação de requisições. O valor de retorno do Algoritmo 2 é *true*, caso tenha sido encontrado espaço suficiente para a alocação, ou *false*, caso contrário.

Na linha 2 é criado um objeto do tipo *Block*, nomeado *usefulArea*. O objeto *usefulArea* representa o espaço disponível entre os blocos *leftBlock* e *rightBlock*. Na linha 5 é verificado se o espaço disponível no bloco *usefulArea* é suficiente para alocar a nova requisição. Caso positivo, a busca por espaço é encerrada, dando início à etapa de alocação (linha 6). Caso contrário, a busca por espaço continua a partir da linha 9.

Na linha 9 verifica-se se o bloco *leftBlock* é nulo. Caso positivo, então não existem mais requisições na fila de espera, portanto a busca por espaço chegou ao fim. Caso a alocação seja obrigatória (linha 10), nas linhas 11 a 13 é realizado o deslocamento do bloco *rightBlock* (caso não seja nulo) para a esquerda, eliminando todo o espaço disponível entre *leftBlock* e *rightBlock*. Esse deslocamento também é realizado em todos os blocos alocados à direita do bloco *rightBlock* (linhas 25 a 30) através do retorno recursivo da chamada da função.

Caso o bloco *leftBlock* não seja nulo (linha 9), a busca por espaço disponível continua à esquerda do bloco *leftBlock* (linha 25). Entre as linhas 17 e 24 são definidos os parâmetros da nova busca. Mais especificamente, entre as linhas 17 e 19 é definido o parâmetro referente ao espaço disponível entre os blocos *leftBlock* e *rightBlock*, de

Algorithm 2: Verificar a existência de espaços livres, e suficientes, entre dois blocos.

```
1  function search_alloc_space(leftBlock, newBlock, rightBlock, spaceNeeded,
     hasRightSpace, cpuFreeTime, forcedPush): boolean is
2      usefulArea = get_useful_area(leftBlock, newBlock, rightBlock, cpuFreeTime)
3      end = usefulArea.get_end()
4      freeSpace = usefulArea.get_size()
5      if freeSpace ≥ spaceNeeded then
6          shift_or_alloc(leftBlock, newBlock, rightBlock, end, spaceNeeded,
             hasRightSpace)
7          return true
8      end
9      if leftBlock == null then
10         if forcedPush == true and rightBlock ≠ null then
11             shiftValue = rightBlock.get_start() - cpuFreeTime
12             end = rightBlock.get_end() - shiftValue
13             rightBlock.set_end(end)
14         end
15         return false
16     end
17     if freeSpace > 0 then
18         _hasRightSpace = true
19     else
20         _hasRightSpace = hasRightSpace
21     end
22     _freeNeeded = spaceNeeded - freeSpace
23     _leftBlock = leftBlock.get_left_block()
24     _rightBlock = leftBlock
25     status = search_alloc_space(_leftBlock, newBlock, _rightBlock,
         _freeNeeded, _hasRightSpace, cpuFreeTime, forcedPush)
26     if status == false then
27         if forcedPush == true and rightBlock ≠ null then
28             shiftValue = rightBlock.get_start() - leftBlock.get_end()
29             end = rightBlock.get_end() - shiftValue
30             rightBlock.set_end(end)
31         end
32         return false
33     end
34     shift_or_alloc(leftBlock, newBlock, rightBlock, end, spaceNeeded,
         hasRightSpace)
35     return true
36 end
```

modo que, caso não haja espaço disponível, obtém-se o valor referente a existência de espaço a partir do bloco *rightBlock* (linhas 19 a 21).

Na linha 22 é obtida a quantidade de espaço necessária nas próximas buscas.

Isto é, se algum espaço foi encontrado na busca atual, o valor deve ser descontado para a próxima busca. Já nas linhas 22 e 23, respectivamente, são definidos os blocos *leftBlock* e *rightBlock* da próxima busca. A linha 34 é utilizada para empurrar o bloco *rightBlock* para a esquerda, ou para alocar o bloco *newBlock*. A operação (deslocamento ou alocação) dependerá da etapa recursiva em que o algoritmo se encontra.

4.3.3 Algoritmo *get_useful_area*

O Algoritmo 3 é utilizado pelo Algoritmo 2 para obter o espaço disponível entre duas requisições: *leftBlock* e *rightBlock*. O valor de retorno corresponde a um objeto do tipo *Block*. Entre as linhas 2 e 6 é definido o início do bloco de retorno, de modo que, se houver uma requisição à esquerda (linha 2), então o início do bloco de retorno é o fim do bloco da requisição à esquerda (linha 3). Caso contrário, o algoritmo está tentando alocar a nova requisição (*newBlock*) no início da fila, portanto o início do bloco de retorno é o momento em que é prevista a liberação da *CPU*.

Algorithm 3: Criar, e retornar, um bloco do tamanho da área disponível entre dois blocos.

```
1  function get_useful_area(leftBlock, newBlock, rightBlock, cpuFreeTime): Block
   is
2      if leftBlock ≠ null then
3          start = leftBlock.get_end()
4      else
5          start = cpuFreeTime
6      end
7      if rightBlock ≠ null then
8          end = rightBlock.get_start()
9      else
10         end = math.inf
11     end
12     end = min(end, newBlock.get_end())
13     if start > end then
14         start = 0
15         end = 0
16     end
17     width = end - start
18     return Block(width, end)
19 end
```

O fim do bloco de retorno é definido entre as linhas 7 e 11. Se houver uma requisição à direita, o início do bloco da requisição à direita é o fim do bloco de retorno (linha 8). Caso contrário, o algoritmo está tentando alocar a nova requisição (*newBlock*) no fim da fila, portanto o fim do bloco de retorno é infinito (linha 10). Contudo, é possível

que o bloco de retorno possua um tamanho maior do que o necessário para alocar a nova requisição, portanto, na linha 12 é descontado o espaço desnecessário.

O valor descontado na linha 12 pode resultar na criação de um bloco com tamanho negativo. Para contornar este problema, entre as linhas 13 e 16 é verificado se o início do bloco é após o fim. Caso positivo, define-se o início e o fim do bloco com o valor 0 (zero). Por fim, na linha 17 é calculado o tamanho do bloco, e na linha 18 o bloco é criado e utilizado como valor de retorno.

4.3.4 Algoritmo *shift_or_alloc*

O Algoritmo 4 é utilizado pelo Algoritmo 2 possui um valor de retorno nulo. Uma dentre duas operações é realizada: empurrar o bloco *rightBlock* para a esquerda, ou alocar o novo bloco *newBlock* entre os blocos *leftBlock* e *rightBlock*. O deslocamento do bloco *rightBlock* ocorre nas linhas 3 e 4, caso haja espaço disponível entre os blocos mais à direita (linha 2). O deslocamento ocorre conforme o espaço requisitado (*spaceNeeded*). Caso contrário, entre as linhas 6 e 9 é realizada a alocação do bloco *newBlock* entre os blocos *leftBlock* e *rightBlock*.

Algorithm 4: Alocar a requisição ou deslocar o bloco atual para a esquerda.

```
1  function shift_or_alloc(leftBlock, newBlock, rightBlock, end, spaceNeeded,
     hasRightSpace): void is
2      if hasRightSpace == true then
3          end = rightBlock.get_end() - spaceNeeded
4          rightBlock.set_end(end)
5      else
6          if newBlock.get_right_block() == null and newBlock.get_left_block() ==
             null then
7              newBlock.set_end(end)
8              alloc_request(leftBlock, newBlock, rightBlock)
9          end
10     end
11 end
```

4.3.5 Algoritmo *alloc_request*

O Algoritmo 5 é utilizado pelo Algoritmo 4 para alocar uma nova requisição *newBlock* entre *leftBlock* e *rightBlock*, e o valor de retorno do algoritmo é nulo. O algoritmo é iniciado através da verificação se o bloco *leftBlock* é diferente de nulo (linha 2). Caso positivo, o bloco *leftBlock* marca o bloco *newBlock* como o bloco à sua direita (linha 3). Contudo, se o bloco *leftBlock* for nulo, o bloco *newBlock* é definido como o primeiro bloco da fila.

Algorithm 5: Alocar a requisição entre dois blocos já existentes.

```
1  function alloc_request(leftBlock, newBlock, rightBlock): void is
2      if leftBlock ≠ null then
3          leftBlock.set_right_block(newBlock)
4      else
5          this.firstBlock = newBlock
6      end
7      if rightBlock ≠ null then
8          rightBlock.set_left_block(newBlock)
9      else
10         this.lastBlock = newBlock
11     end
12     newBlock.set_left_block(leftBlock)
13     newBlock.set_right_block(rightBlock)
14 end
```

Já na linha 7 é verificado se o bloco *rightBlock* é diferente de nulo. Caso positivo, o bloco *rightBlock* marca o bloco *newBlock* como o bloco à sua esquerda (linha 8). Contudo, se o bloco *rightBlock* for nulo, o bloco *newBlock* é definido como o último bloco da fila. Por fim, as linhas 12 e 13 definem os blocos *leftBlock* e *rightBlock*, respectivamente, como os blocos à esquerda e à direita do bloco *newBlock*.

5 AMBIENTE DE EXPERIMENTAÇÃO BASEADO EM SIMULAÇÃO

Após a proposta de adaptação de um orquestrador de carga, descrito no Capítulo 4, a próxima etapa é a realização de experimentos para avaliar o novo modelo e compará-lo com a versão original. Para habilitar a etapa experimental foi necessária a utilização de um ambiente de experimentação, seja um ambiente existente, ou um novo ambiente a ser elaborado. Por conveniência, devido a dificuldades do autor em entender os simuladores existentes, optou-se pelo desenvolvimento de um novo simulador de carga[1].

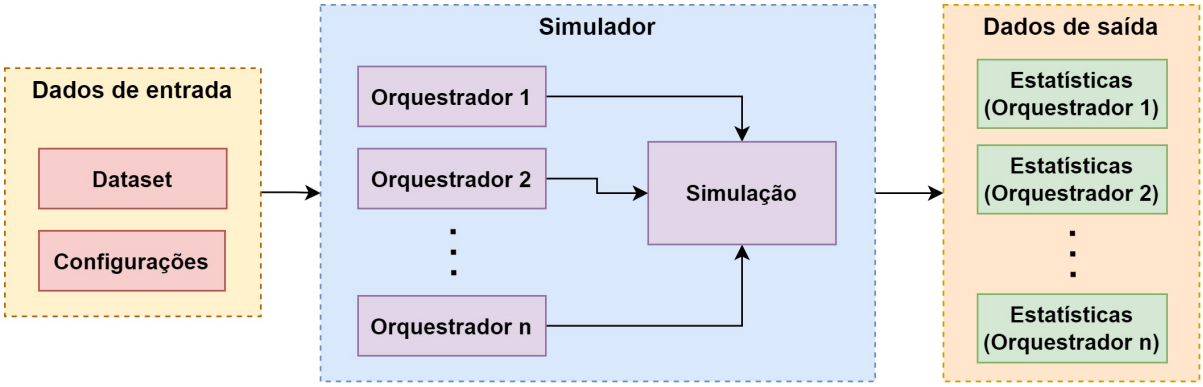
O novo simulador foi desenvolvido utilizando a linguagem de programação Python e se caracteriza por não receber a interferência dos recursos computacionais da máquina onde está sendo executado. Isto é, considerando duas máquinas, máquina A e máquina B, o resultado obtido em uma das máquinas deve ser muito semelhante ao obtido na outra, mesmo que uma das máquinas possua recursos computacionais inferiores. As diferenças entre os resultados obtidos, entre as duas máquinas, ocorre apenas devido à aleatoriedade de algumas variáveis envolvidas na simulação. Isto é, a cada simulação de um mesmo modelo de orquestração de carga, utilizando o mesmo cenário de experimentação, uma mesma máquina também irá gerar resultados distintos, porém estatisticamente próximos.

O início da simulação consiste em uma leitura prévia de dois arquivos de entrada, sendo um arquivo contendo um *dataset*, e outro contendo configurações do simulador. O *dataset* deve ser formado por um conjunto de requisições de serviço (r_i) constituídas por três informações: o código do serviço requisitado (s_y), o código nó *MEC* que deverá receber a requisição (m_k), e o horário, em milissegundos (ms), em que a requisição será enviada ao nó *MEC* (h). Já o arquivo de configuração deve conter o conjunto de nós *MEC* ($M = \{m_1, m_2, ..., m_k\}$) e o conjunto de serviços disponíveis ($S = \{s_1, s_2, ..., s_y\}$), sendo que cada serviço deve conter um tempo de processamento (p) e um prazo de resposta (d). Conforme ilustrado na Figura 11, após a conclusão de todas as simulações, o simulador fornecerá dados estatísticos sobre o desempenho de cada orquestrador de carga. A ilustração também destaca a localização dos orquestradores de carga, os quais foram previamente implementados dentro do simulador.

Na Figura 12 é apresentada uma ilustração sobre o funcionamento interno do simulador, o qual é dividido em quatro etapas. Na etapa 1 é inicializado o simulador, enquanto na etapa 2 são realizados os procedimentos que preparam o simulador para a nova simulação. Cada simulação é realizada na etapa 3 e representa a realização de testes experimentais sobre um determinado orquestrador de carga (o_x). De modo geral, cada simulação se resume a um escalonamento de eventos, os quais são ordenados em uma fila (E) conforme o horário que estão programados para acontecerem. Por fim,

[1] https://github.com/ricardoboing/MEC-LO-Simulator

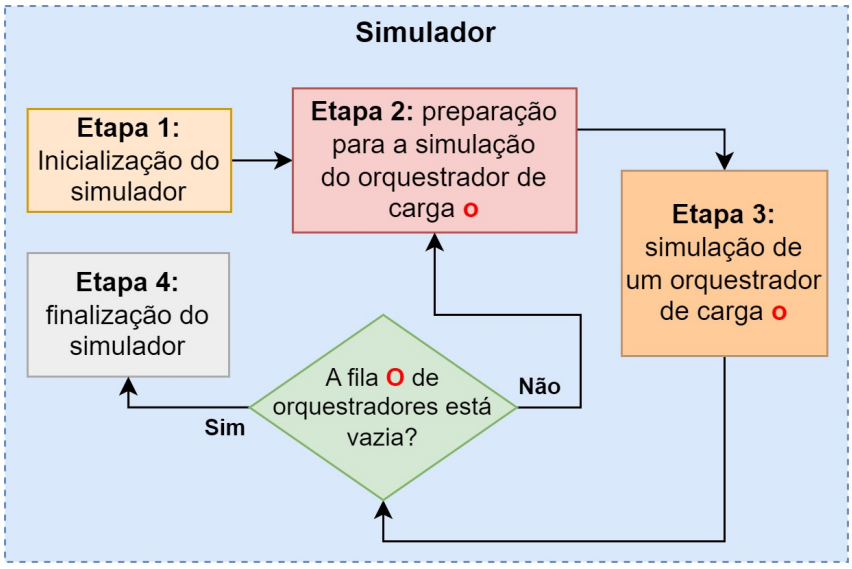
Figura 11 – Representação ilustrativa, do simulador de carga, em uma perspectiva generalizada.



Fonte: Elaborado pelo autor.

após a conclusão de todas as simulações, o simulador é finalizado na etapa 4, na qual são gerados os dados de saída (em formato gráfico). Um maior detalhamento sobre os tipos de eventos é apresentado na Seção 5.3, e o restante deste capítulo se destina a detalhar cada uma das quatro etapas ilustradas na Figura 12.

Figura 12 – Representação ilustrativa do funcionamento interno do simulador.



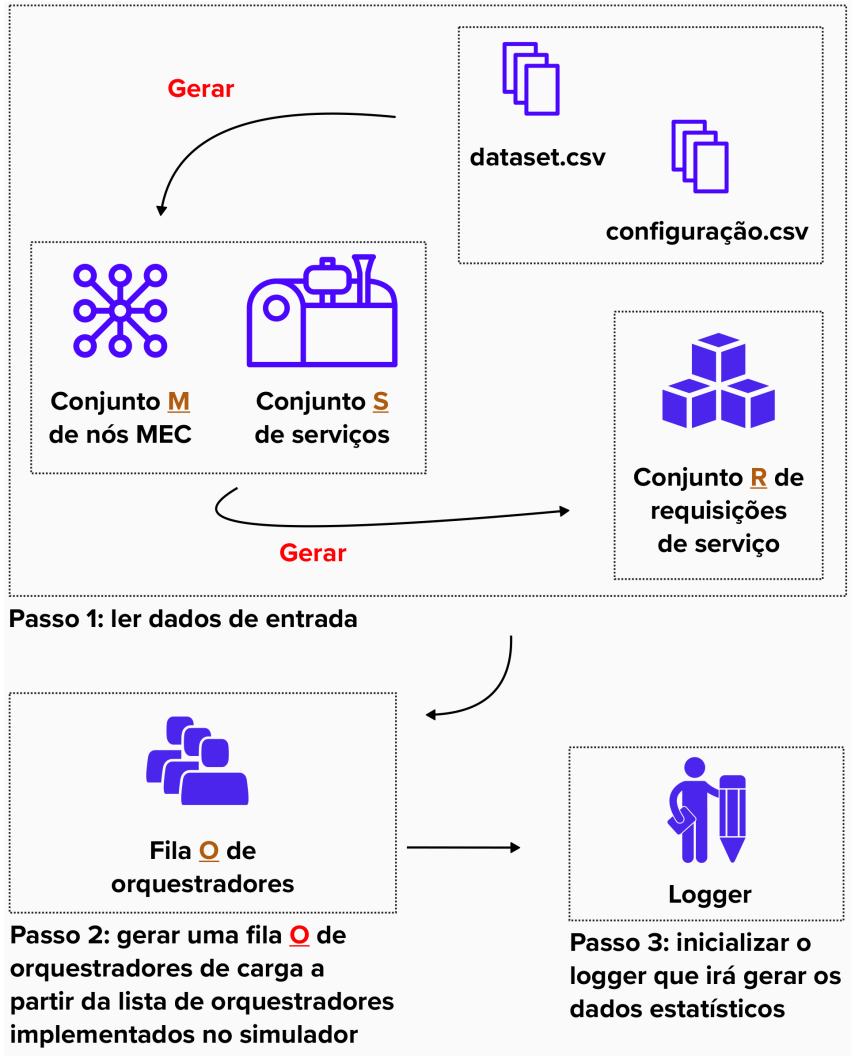
Fonte: Elaborado pelo autor.

5.1 ETAPA 1: INICIALIZAÇÃO DO SIMULADOR

Conforme ilustrado na Figura 13, a primeira etapa corresponde a inicialização do simulador para habilitar a simulação dos orquestradores de carga (o_X). Primeiramente é realizada a leitura dos arquivos contendo o *dataset* e as configurações do simulador,

e em seguida são gerados o conjunto de nós *MEC* (*M*), e o conjunto de serviços (*S*). Os conjuntos *M* e *S* são utilizados para auxiliar a criação do conjunto de requisições (*R*), o qual é utilizado na etapa 2 para gerar os eventos iniciais da simulação (e_f). Nesta etapa também é gerada a fila de orquestradores de carga (*O*), criada a partir do conjunto de orquestradores que foram previamente implementados dentro do simulador. Por fim, é realizada a inicialização do *logger* que fará o registro dos dados estatísticos de cada simulação, os quais serão utilizados, posteriormente, para gerar os dados de saída em formato gráfico.

Figura 13 – Representação ilustrativa sobre o funcionamento da etapa 1 do simulador.



Fonte: Elaborado pelo autor.

Na Figura 14 é apresentado o trecho inicial do código-fonte do simulador. Entre as linhas 8 e 12 é realizada a leitura dos arquivos de entrada. Na linha 14 é gerado o conjunto de orquestradores (*O*), o qual contêm os modelos de orquestração *SFA* e *IcoinBoingSFA*. Na linha 17 é invocado o método *init* da classe *Simulator*, o qual inicializa o *Logger* que fará o registro dos dados estatísticos da simulação. Por fim, na

linha 18 é invocado o método *simulate*, da classe *Simulator*, que iniciará a etapa 2 após utilizar o método *_read_files* para gerar os conjuntos *M*, *S* e *R* através dos arquivos de entrada lidos nas linhas 8 e 12. O método *init*, e a primeira linha do método *simulate*, são apresentados na Figura 15, enquanto a Figura 16 apresenta o método *_read_files*.

Figura 14 – Implementação da parte principal da etapa 1 do simulador de carga (na linguagem de programação Python).

```
7   def main():
8       datasetFileName = INPUT_FILE_NAME_DATASET
9       settingsFileName = INPUT_FILE_NAME_SETTINGS
10
11      datasetSrc = get_input_file_src(datasetFileName)
12      settingsSrc = get_input_file_src(settingsFileName)
13
14      loadOrchestratorTypeList = [Sfa, IcoinBoingSfa]
15
16      print("Simulating a scenario...")
17      Simulator.init()
18      Simulator.simulate(datasetSrc, settingsSrc, loadOrchestratorTypeList)
```

Fonte: Elaborado pelo autor.

Figura 15 – Implementação do método *init* e da primeira linha do método *simulate* (na linguagem de programação Python). Ambos os métodos pertencem a classe *Simulator*

```
63      @staticmethod
64      def init():
65          Simulator.simulatorLogger = SimulatorLogger()
66
67      @staticmethod
68      def simulate(datasetSrc, settingsSrc, loadOrchestratorTypeList):
69          mecList, serviceRequestList = Simulator._read_files(settingsSrc, datasetSrc)
```

Fonte: Elaborado pelo autor.

5.2 ETAPA 2: PREPARAÇÃO PARA A SIMULAÇÃO DE UM ORQUESTRADOR DE CARGA

Conforme ilustrado na Figura 17, na segunda etapa são realizados os procedimentos básicos, e individuais, que antecipam o processo de simulação de um determinado orquestrador o_x obtido na fila *O*. O conjunto de simulações é realizado de maneira sequencial, isto é, simula-se um único orquestrador por vez. O primeiro passo a ser realizado é a obtenção do primeiro orquestrador o_x na fila *O*. Em seguida, o conjunto de nós *MEC* (*M*) é re-configurado para começar a utilizar o orquestrador o_x

Figura 16 – Implementação do método *_read_files* da classe *Simulator* (na linguagem de programação Python)

```
45      @staticmethod
46      def _read_files(settingsSrc, datasetSrc):
47          settingsReader = SettingsReader(settingsSrc)
48          mecList = settingsReader.get_mec_list()
49          mecDict = settingsReader.get_mec_dict()
50          network = settingsReader.get_network()
51          serviceDict = settingsReader.get_service_dict()
52
53          datasetReader = DatasetReader(datasetSrc, mecDict, serviceDict, network)
54          serviceRequestList = datasetReader.get_request_list()
55
56          return mecList, serviceRequestList
```

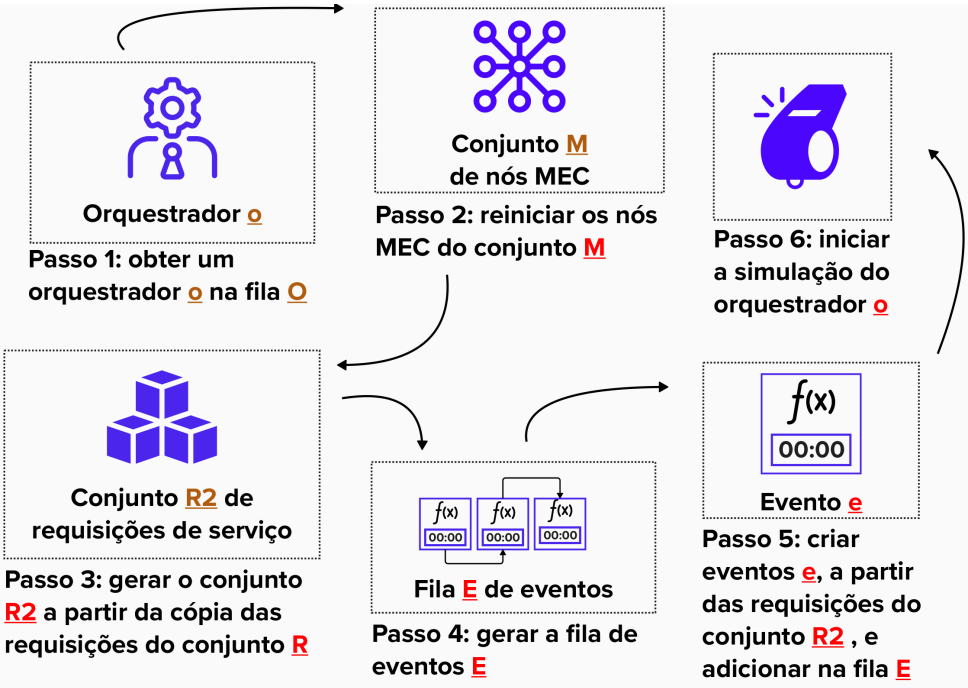
Fonte: Elaborado pelo autor.

para orquestrar a carga. O terceiro passo é gerar o conjunto R_2, a partir da clonagem das requisições do conjunto R, para que as requisições recebidas pelos nós *MEC* (m_k), durante a simulação de um orquestrador o_x, sejam idênticas àquelas recebidas durante a simulação dos demais orquestradores. Após gerar o conjunto R_2 é realizada a criação da fila ordenada de eventos (E), que deve ser inicializada com uma lista inicial de eventos, gerados a partir das requisições do conjunto R_2, que correspondem ao envio da requisição a um nó *MEC* no horário especificado no *dataset*. Por fim, da-se início a simulação do orquestrador o_x.

Na Figura 18 é apresentado o código-fonte do método *simulate* da classe *Simulator*. Com a excessão da linha 69, que faz parte da etapa 1, o restante do método *simulate* corresponde ao código principal da etapa 2. Na linha 71 é possível observar a seleção do orquestrador o_x, que é posteriormente utilizado para a reconfiguração dos nós *MEC* (linha 74). A construção do conjunto R_2 é realizada na linha 76, e na linha 81 são criados os eventos e_f que são inseridos na fila E. Por fim, a simulação é iniciada na linha 82. Um maior detalhamento sobre os métodos *generate_user_send_request_event_list* e *start*, da classe *Simulation*, é apresentado na Seção 5.3.

Os eventos e_f são formados pela tupla (*function*, *parameters*, *temporalMoment*). A variável *function* corresponde a uma função que deve ser invocada para que o evento aconteça. A variável *parameters* contêm os parâmetros que a função *function* receberá quando for invocada. Já a variável *temporalMoment* contêm o horário em que o evento deve ocorrer. Um evento e_f é uma instância da classe *SimulationEvent*, apresentada na Figura 19, onde é possível observar a existência de três métodos: *happen*, utilizado para invocar o evento, *get_time*, utilizado para obter o horário do evento, e *generate_event*, utilizado para auxiliar na criação de novos eventos e inseri-

Figura 17 – Representação ilustrativa sobre o funcionamento da etapa 2 do simulador.



Fonte: Elaborado pelo autor.

Figura 18 – Implementação do método *simulate* da classe *Simulator* (na linguagem de programação Python).

```
67      @staticmethod
68      def simulate(datasetSrc, settingsSrc, loadOrchestratorTypeList):
69          mecList, serviceRequestList = Simulator._read_files(settingsSrc, datasetSrc)
70
71          for loadOrchestratorType in loadOrchestratorTypeList:
72              currentApproach = loadOrchestratorType.__name__
73
74              Simulator._reset_mec_list(mecList, loadOrchestratorType)
75
76              cloneServiceRequestList = Simulator._clone_requests_from_list(serviceRequestList)
77              Simulator.currentSimulation = Simulation(mecList)
78
79              Simulator._notify_created_simulation(currentApproach)
80              Simulator._notify_started_simulation()
81              Simulator.currentSimulation.generate_user_send_request_event_list(cloneServiceRequestList)
82              Simulator.currentSimulation.start()
83              Simulator._notify_finished_simulation()
84          Simulator._notify_finished_all_simulation()
```

Fonte: Elaborado pelo autor.

los na fila *E*.

5.3 ETAPA 3: SIMULAÇÃO DE UM ORQUESTRADOR DE CARGA

Conforme ilustrado na Figura 20, a terceira etapa corresponde a realização da simulação do orquestrador de carga o_x. O primeiro passo é reiniciar o relógio para que

Figura 19 – Implementação da classe *SimulationEvent* (na linguagem de programação Python).

```python
class SimulationEvent:
    def __init__(self, function, parameters, temporalMoment):
        self.function = function
        self.parameters = parameters
        self.time = temporalMoment

    def happen(self):
        return self.function( self.parameters )

    def get_time(self):
        return self.time

    @staticmethod
    def generate_event(function, parameters, time):
        from simulation.Simulator import Simulator

        simulationEvent = SimulationEvent(function, parameters, time)
        Simulator.get_current_simulation().add_simulation_event(simulationEvent)
```

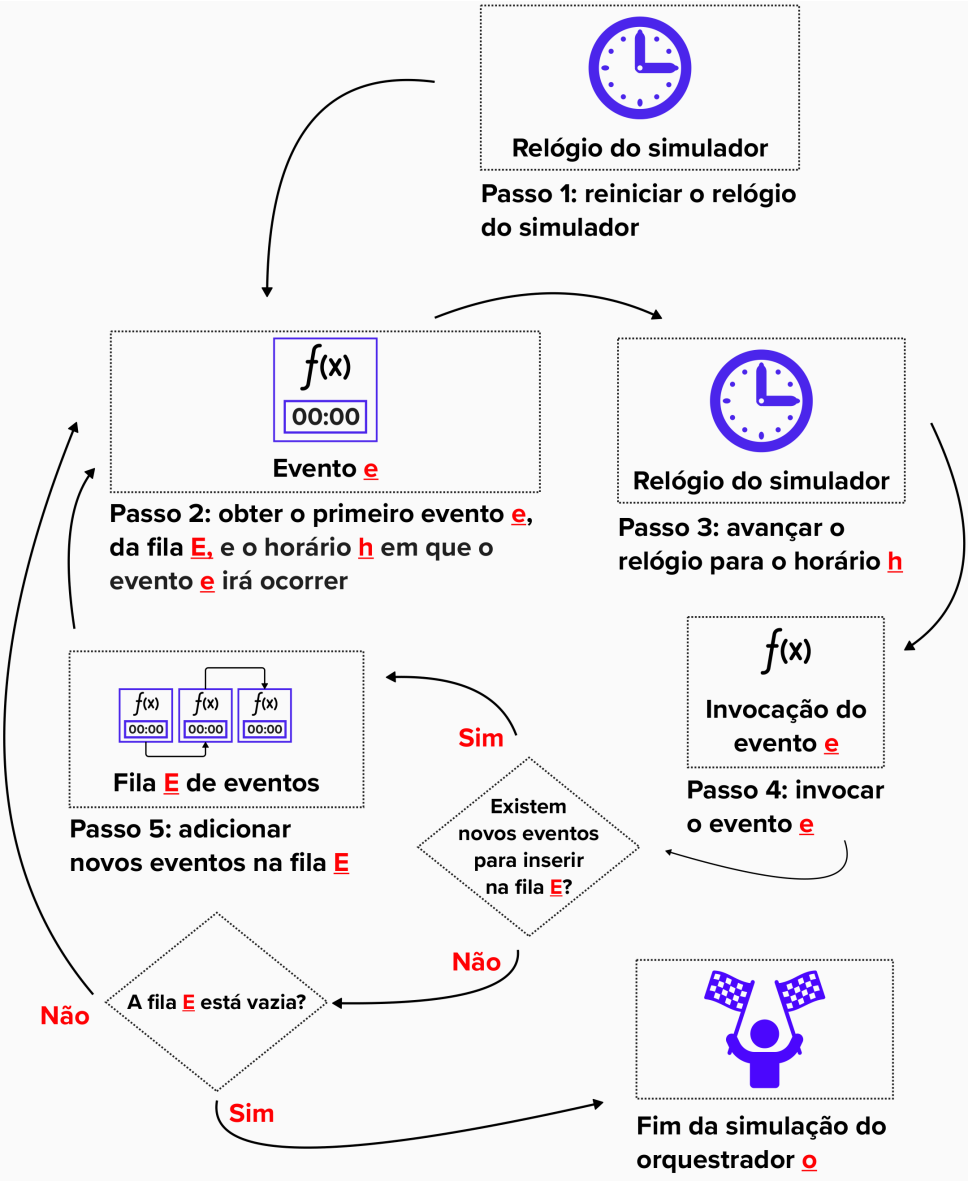
Fonte: Elaborado pelo autor.

o ponteiro aponte para o horário 0*ms* (zero milissegundos). Em seguida, o primeiro evento e_f é obtido da fila E, que está ordenada (de forma crescente) pelo horário de acontecimento dos eventos e_f. Considerando que o tempo entre o término de um evento e_{f-1}, e o início do evento e_f é irrelevante, o simulador avança o relógio para o horário h em que o evento e_f será iniciado. Na sequência é realizada a invocação do evento e_f, no qual, durante a sua realização, é possível acarretar geração de novos eventos que deverão ser adicionados na fila E. Todos os novos eventos deverão ser gerados para acontecerem em um horário h posterior ao atual horário do relógio, e a simulação chegará ao fim quando, simultaneamente, todos os eventos forem realizados e não houver novos eventos a serem inseridos na fila E.

Na Figura 21 é apresentado o trecho de código referente a implementação da classe *Simulation*, que possui métodos como *start*, *add_simulation_event* e *generate_user_send_request_event_list*. No método *start* é iniciada a simulação, que é uma instância da classe *Simulation*, cuja criação ocorreu na etapa 2. O relógio é reinicializado na linha 11, e na linha 13 é iniciado o percorrimento da fila de eventos E. A linha 14 obtêm o evento e_f da fila E, e a linha 15 obtêm o horário h em que o evento e_f deve ocorrer. O ponteiro do relógio é atualizado para o horário h na linha 17, e a invocação do evento e_f é concretizada na linha 18. Já o método *add_simulation_event* é utilizado em outras partes do simulador para adicionar um novo evento na fila E.

O método *generate_user_send_request_event_list* (mencionado na seção anterior) é invocado na etapa 2 e se destina a criação dos eventos de envio de requisições

Figura 20 – Representação ilustrativa sobre o funcionamento da etapa 3 do simulador.



Fonte: Elaborado pelo autor.

aos nós *MEC*. Na linha 27 é realizado o percorrimento das requisições r_i contidas na lista de requisições R. Para cada requisição r_i, na linha 28 é obtido o nó usuário (da rede) que está enviando a requisição r_i a um determinado nó *MEC* m_k. A linha 29 é utilizada para invocar o método *send_service_request*, da classe *User*, que na sequência invoca o método *send_network_message*, da classe *Nic*, para enviar a requisição via rede. Apesar da invocação do método *send_network_message* aparentar o envio imediato da requisição, o que ocorre é um agendamento para que o envio ocorra no horário h, conforme será explicado no decorrer desta seção. A implementação da classe *User* é apresentada na Figura 22.

A implementação do método *send_network_message*, da classe *Nic*, pode ser

Figura 21 – Implementação da classe *Simulation* (na linguagem de programação Python).

```
5   class Simulation:
6       def __init__(self, mecList):
7           self.mecList = mecList
8           self.eventScheduler = EventScheduler()
9
10      def start(self):
11          Clock.reset()
12
13          while not self.eventScheduler.is_empty():
14              event = self.eventScheduler.get_next_event()
15              time = event.get_time()
16
17              Clock.set_time(time)
18              event.happen()
19
20      def get_mec_list(self):
21          return self.mecList
22
23      def add_simulation_event(self, event):
24          self.eventScheduler.push_event(event)
25
26      def generate_user_send_request_event_list(self, serviceRequestList):
27          for serviceRequest in serviceRequestList:
28              recipient = serviceRequest.get_recipient()
29              recipient.send_service_request(serviceRequest)
```

Fonte: Elaborado pelo autor.

observada na Figura 23a. Na linha 17 é criado o pacote que será enviado pela rede através do método *generate_network_packet_receive_event* da classe *NicEventGenerator* (linha 18), invocado para criar um evento e_f no qual o nó de destino receberá o pacote no horário *h*. A Figura 23b (linhas 5 a 9) apresenta a implementação do método *generate_network_packet_receive_event*, o qual é explicado a seguir.

5.3.1 Eventos

Os eventos correspondem a uma ação de alguma entidade do simulador durante a etapa de simulação, que pode estar relacionado a classes como *User*, *MecNode*, *Cpu* e *Nic*. Existem várias ações que podem ser realizadas por estas entidades, como o envio e recebimento de pacotes, inicialização e finalização do processamento de uma requisição, adição ou remoção de uma requisição da fila de espera, dentre outras. No entanto, apenas 2 (duas) ações precisam ser programadas para acontecerem em um momento posterior, e são essas ações que são transformadas em eventos para

Figura 22 – Implementação da classe *User* (na linguagem de programação Python).

```python
class User(NetworkNode):
    def __init__(self):
        super().__init__( UserLogger() )
        self.nic.attach(self)

    def send_service_request(self, serviceRequest):
        time = serviceRequest.get_created_time()
        destination = serviceRequest.get_first_destination()

        self.nic.send_network_message(destination, serviceRequest, time)

    def _receive_responsed_request(self, serviceRequestResponse):
        serviceRequest = serviceRequestResponse.get_service_request()
        serviceRequest.set_responsed()

        self.notify_all(
            NotifyParameter(serviceRequest, NotifyType.RESPONSED_REQUEST)
        )

    def observer_update(self, parameters):
        logType = parameters.get_type()
        content = parameters.get_content()

        if logType == NotifyType.NIC_RECEIVE and type(content) == ServiceRequestResponse:
            self._receive_responsed_request(content)
            return
        raise
```

Fonte: Elaborado pelo autor.

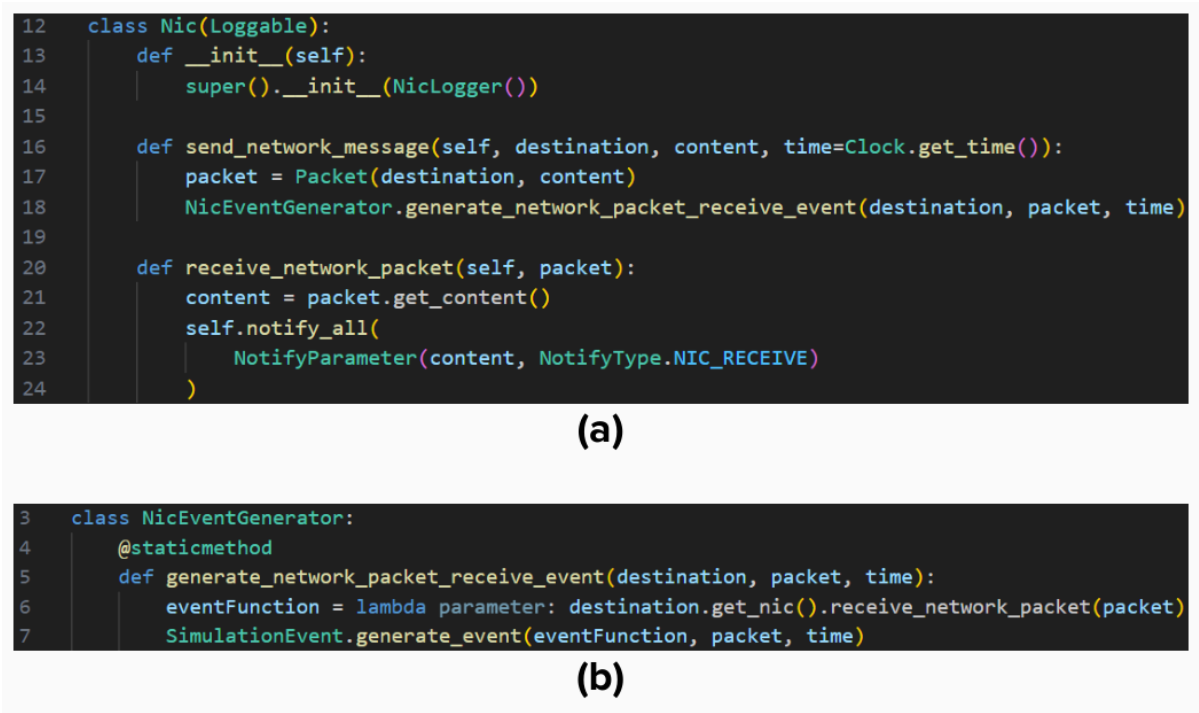
serem adicionadas a fila *E*. As demais ações derivam destas 2 e ocorrem de forma imediata, sem a necessidade de serem transformadas em eventos e adicionadas na fila *E*.

Um exemplo de ação, derivada do evento de liberação da *CPU*, é a remoção da primeira requisição da fila de espera. Após a liberação da *CPU*, a fila de espera é consultada a fim de verificar a existência de requisições que estejam aguardando para serem processadas. Caso a fila não esteja vazia, a primeira requisição é removida da fila e encaminhada para a *CPU*, que na sequência inicia o processamento desta requisição. Já o evento de recebimento de pacotes de rede acarreta tentativa de alocação de uma requisição, na fila de espera, ou em um possível encaminhamento da requisição a outro nó *MEC*.

5.3.1.1 Recebimento de pacotes de rede

O evento de recebimento de pacotes é criado pela classe *NicEventGenerator* através do método *generate_network_packet_receive_event*, que é utilizado, por exemplo, na classe *Simulation* através do método *generate_user_send_request_event_list*. Os parâmetros de entrada do método *generate_network_packet_receive_event* são o

Figura 23 – Implementação das classes *Nic* e *NicEventGenerator* (na linguagem de programação Python).



Fonte: Elaborado pelo autor.

endereço do nó destinatário (*destination*), o pacote contendo a requisição (*packet*), e o horário em que a requisição deve ser recebida pelo nó destinatário (*time*). A implementação do método *generate_network_packet_receive_event* pode ser observada na Figura 23b, entre as linhas 4 e 7. Na linha 6 é definida a função *lambda* que deverá ser invocada para que o novo evento e_f aconteça, a qual corresponde a invocação do método *receive_network_packet*, da classe *Nic*, para que um pacote seja recebido por uma instância da classe *Nic* e posteriormente repassado a uma instância da classe *User* ou *MecNode* (que contêm a instância da classe *Nic*). A função lambda é passada como parâmetro na invocação do método *generate_event* (linha 7), da classe *SimulationEvent*, utilizado para criar um novo evento e_f e adicioná-lo na fila de eventos E. A classe *SimulationEvent* é apresentada na Figura 19, e o método *generate_event* está implementado entre as linhas 13 e 18.

5.3.1.2 Liberação da *CPU*

O evento de liberação da *CPU* ocorre após a conclusão do processamento de uma requisição em uma instância da classe *MecNode*. Esse evento é gerado pela classe *CpuEventGenerator* através do método *generate_cpu_release_event*, conforme apresentado na Figura 24. Na linha 6 é gerada a função *lambda*, a qual invoca o método *release*, de uma instância da classe *Cpu*, para que a *CPU* seja liberada. Na linha 7 é

obtido o horário *h* em que essa liberação deve ocorrer, e na linha 10 é gerado o evento e_f a partir do método *generate_event* da classe *SimulationEvent*.

Figura 24 – Implementação da classe *CpuEventGenerator* (na linguagem de programação Python).

```
3   class CpuEventGenerator:
4       @staticmethod
5       def generate_cpu_release_event(cpu):
6           eventFunction = lambda parameter: cpu.release()
7           time = cpu.get_release_time()
8           parameters = None
9
10          SimulationEvent.generate_event(eventFunction, parameters, time)
```

Fonte: Elaborado pelo autor.

5.4 ETAPA 4: FINALIZAÇÃO DO SIMULADOR

Após a simulação de todos os orquestradores contidos na fila *O*, o simulador deve gerar os dados de saída e ser finalizado. Os dados de saída correspondem a figuras gráficas, contendo informações sobre o desempenho de cada orquestrador ao longo da etapa experimental. Para acessar os gráficos é necessário que o simulador seja executado em uma *Integrated Development Environment (IDE)* que forneça extensões do *Jupyter Notebook*. Para o desenvolvimento deste simulador foi utilizada a *IDE Visual Studio Code (VS Code).*

6 DATASET

No Capítulo 5 é apresentado um simulador de carga elaborado para habilitar a avaliação dos orquestradores de carga durante a etapa experimental. Para funcionar, o simulador requer, dentre outras informações de entrada, um conjunto de requisições que devem ser enviadas a cada um dos nós *MEC* ao longo da simulação. Esse conjunto de requisições representa um cenário de experimentação no qual os orquestradores serão testados e avaliados, sendo informado ao simulador mediante um arquivo chamado *dataset*.

O arquivo *dataset* é composto por uma tabela semelhante a apresentada na Figura 25. Cada linha representa uma requisição de serviço, e as três colunas, nomeadas *mec*, *timestamp* e *service*, representam, respectivamente, o código do nó *MEC* que receberá a requisição, o horário em que a requisição será recebida pelo nó *MEC*, e o serviço que está sendo requisitado. Cada um dos *datasets*, utilizados no Capítulo 7, foram obtidos mediante um gerador de *dataset* que foi elaborado nesta dissertação de mestrado. O gerador recebe alguns dados de entrada, sobre as características do cenário de experimentação que está sendo considerado, e gera o arquivo *dataset* como saída.

Figura 25 – Captura de tela de um arquivo *dataset*

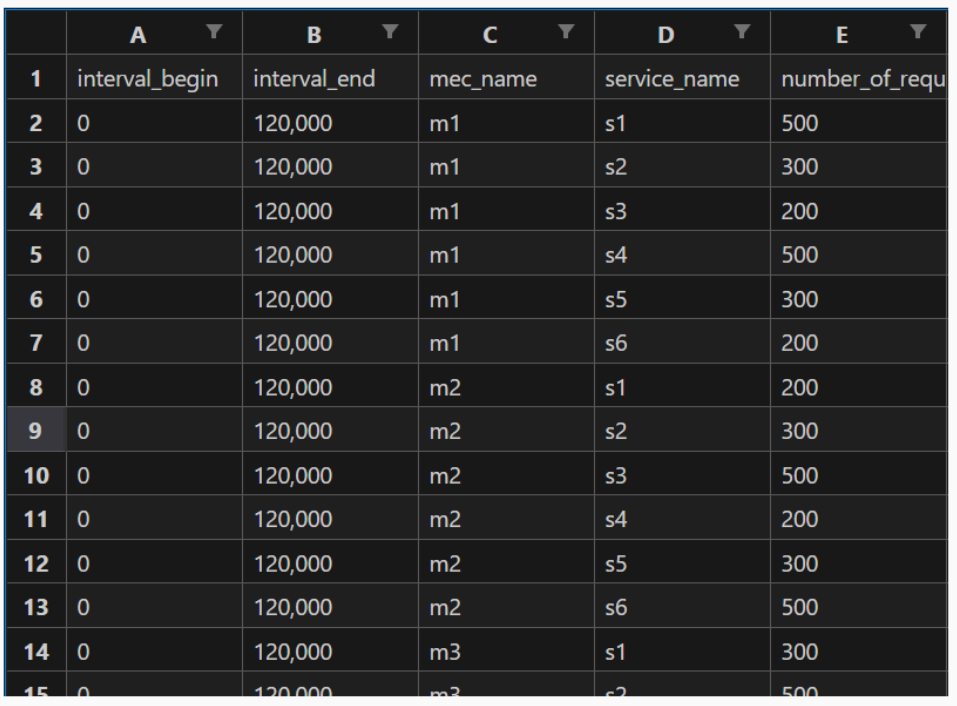
	A	B	C
1	mec	timestamp	service
2	m1	34,835	s1
3	m1	118,792	s1
4	m1	5,888	s1
5	m1	23,173	s1
6	m1	61,878	s1
7	m1	64,970	s1
8	m1	103,676	s1
9	m1	24,097	s1
10	m1	5,547	s1
11	m1	61,202	s1
12	m1	78,419	s1
13	m1	20,586	s1

Fonte: Elaborado pelo autor.

Um exemplo de arquivo de entrada, inserido no gerador de *dataset*, é apresentado pela Figura 26. É possível observar que os dados de entrada definem que cada nó

MEC receberá, em um determinado intervalo de tempo, um número específico de requisições para cada tipo de serviço. O intervalo é definido pelas colunas *interval_begin* e *interval_end*, cujos valores são apresentados em milissegundos (ms). As colunas *mec_name*, *service_name*, e *number_of_requests* representam, respectivamente, o código do nó *MEC*, o código do serviço requisitado e o número de requisições a serem realizadas ao nó *MEC*.

Figura 26 – Captura de tela de um arquivo utilizado como entrada para o gerador de *dataset*.



	A	B	C	D	E
1	interval_begin	interval_end	mec_name	service_name	number_of_requ
2	0	120,000	m1	s1	500
3	0	120,000	m1	s2	300
4	0	120,000	m1	s3	200
5	0	120,000	m1	s4	500
6	0	120,000	m1	s5	300
7	0	120,000	m1	s6	200
8	0	120,000	m2	s1	200
9	0	120,000	m2	s2	300
10	0	120,000	m2	s3	500
11	0	120,000	m2	s4	200
12	0	120,000	m2	s5	300
13	0	120,000	m2	s6	500
14	0	120,000	m3	s1	300
15	0	120,000	m3	s2	500

Fonte: Elaborado pelo autor.

A criação do conjunto de requisições ocorre através do método *read_scenario*, apresentado na Figura 27. Nas linhas 14 e 15 é realizada a leitura do arquivo de entrada, o qual é transformado em uma lista de linhas (*rowList*). Esta lista é percorrida entre as linhas 17 e 32 para gerar cada uma das requisições que serão inseridas no arquivo *dataset*. Entre as linhas 18 e 22 são obtidas as informações sobre as requisições que serão criadas, como o nó *MEC* que deverá receber estas requisições, o serviço que está sendo requisitado, o intervalo de tempo no qual o nó *MEC* deverá recebê-las, e o número de requisições que deverão ser realizadas dentro desse intervalo. As linhas 24 a 28 são utilizadas para armazenar informações que servem apenas para auxiliar a escrita do arquivo de saída, pela classe *DatasetWriter*, a qual não será descrita nesta dissertação por não ser relevante para a geração dos dados do *dataset*. Por fim, nas linhas 30 a 32 as requisições são finalmente geradas e adicionadas em uma lista.

A criação das requisições é realizada através do método *_create_request*, apre-

Figura 27 – Implementação do método *read_scenario* (na linguagem de programação Python).

```
13      def read_scenario(self):
14          requestReaderSheet = self._read_request_reader_sheet()
15          rowList = requestReaderSheet.get_row_list()
16
17          for row in rowList:
18              intervalBegin = row["interval_begin"]
19              intervalEnd = row["interval_end"]
20              mecName = row["mec_name"]
21              serviceName = row["service_name"]
22              numberOfRequest = row["number_of_request"]
23
24              if mecName not in self.mecNameList:
25                  self.mecNameList.append(mecName)
26
27              if serviceName not in self.serviceNameList:
28                  self.serviceNameList.append(serviceName)
29
30              for i in range(0, numberOfRequest):
31                  request = self._create_request(mecName, serviceName, intervalBegin, intervalEnd)
32                  self.requestList.append(request)
```

Fonte: Elaborado pelo autor.

sentado na Figura 28. A requisição é uma lista contendo o código do nó *MEC* (*mecName*), o horário em que a requisição será recebida pelo nó *MEC* (*time*), e o tipo de serviço que está sendo requisitado (*serviceName*). Com a excessão da variável *time*, todas as informações são obtidas através da chamada do método *_create_request* na linha 31 do método *read_scenario*. Para gerar o valor de *time* é considerado o intervalo de criação, o qual é utilizado para criar um horário randômico em um momento entre o início (*intervalBegin*) e o fim (*intervalEnd*) do intervalo especificado.

Figura 28 – Implementação do método *_create_request* (na linguagem de programação Python).

```
49      def _create_request(self, mecName, serviceName, intervalBegin, intervalEnd):
50          time = random.randint(intervalBegin, intervalEnd)
51          request = [mecName, time, serviceName]
52
53          return request
```

Fonte: Elaborado pelo autor.

O restante deste capítulo se destina a explicação de cada cenário de experimentação elaborado. No decorrer do texto são utilizadas algumas nomenclaturas, as quais são apresentadas no Quadro 3.

Quadro 3 – Siglas utilizadas no Capítulo 6.

Sigla	Significado
ms	milissegundos
c_j	Cenário *j*
m_k	Nó *MEC k*
s_y	Serviço *y*
$I_{pc}(c_j)$	Índice Prazo Total sobre Carga Total
$C_t(c_j)$	Carga Total submetida ao cenário c_j
$C_t(m_k)$	Carga Total submetida ao nó *MEC* m_k
$P_t(c_j)$	Prazo Total estabelecido no cenário c_j
$P_t(m_k)$	Prazo Total estabelecido no nó *MEC* m_k
$n(S(m_k))$	Número de serviços fornecidos pelo nó *MEC* m_k
$n(M(c_j))$	Número de nós *MEC* no cenário c_j
$n(R(m_k,s_y))$	Número de requisições do serviço s_y submetidas ao nó *MEC* m_k
$p(s_y,m_k)$	Tempo de execução do serviço s_y no nó *MEC* m_k
$d(s_y)$	Prazo de resposta estabelecido para o serviço s_y

Fonte: Elaborado pelo autor.

6.1 CENÁRIOS DE EXPERIMENTAÇÃO

Um total de 3 cenários de experimentação foram elaborados para avaliar os modelos de orquestração de carga. Em ambos os cenários considerou-se o fornecimento de 6 tipos de serviços pelos nós *MEC*, conforme apresentado na Tabela 1. A primeira coluna contêm o código dos serviços (s_y) providos pelos nós *MEC*, enquanto na segunda coluna é apresentado o tempo de execução dos serviços (p), e na terceira coluna é apresentado o prazo máximo (d) estabelecido para que o nó *MEC* retorne uma resposta.

Tabela 1 – Características dos serviços fornecidos pelos nós *MEC*.

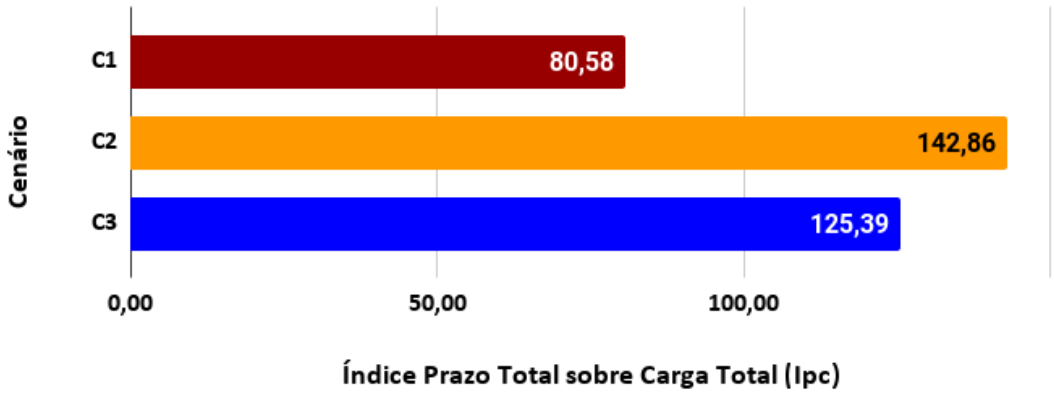
Serviço	Tempo de execução (p)	Prazo (d)
s_1	180*ms*	9.000*ms*
s_2	40*ms*	9.000*ms*
s_3	20*ms*	9.000*ms*
s_4	180*ms*	4.000*ms*
s_5	40*ms*	4.000*ms*
s_6	20*ms*	4.000*ms*

Fonte: Elaborado pelo autor.

A diferença entre os cenários de experimentação se deve a dois fatores: o número de nós *MEC* presentes na rede e o número de requisições realizadas a cada nó *MEC*. Considerando as características de cada serviço, isto é, o tempo de processamento (p), e o prazo de resposta (d), os cenários (c_j) são comparados entre si

através da carga de trabalho total (C_t) e o prazo total de resposta (P_t). A comparação entre os cenários é apresentada na Figura 29, no qual é apresentado o valor do Índice Prazo Total sobre Carga Total (I_{pc}). Três tipos de classificações foram definidas para o índice I_{pc}: alto, médio e baixo. O cenário 1 é aquele cujo I_{pc} é o menor, portanto foi classificado na categoria baixo. Já o cenário 2 foi classificado como alto, visto que o I_{pc} é o maior dentre os 3 (três) cenários. Por fim, o cenário 3 foi classificado como médio, pois o valor I_{pc} está entre os obtidos no cenário 1 e cenário 2.

Figura 29 – Comparação gráfica entre os cenários de experimentação em relação ao índice I_{pc}.



Fonte: Elaborado pelo autor.

Através do índice I_{pc} é possível constatar a proporção entre os prazos, pré-estabelecidos para cada serviço s_y, e a carga total C_t, recebida pelo conjunto de nós *MEC*. Quando comparado ao cenário 2 e cenário 3, o cenário 1 recebeu mais requisições com tempo de processamento alto ou com prazo curto, tornando o cenário 1 mais desafiador para cumprir os prazos que foram pré-estabelecidos. O cálculo do índice I_{pc} é realizado através da Eq. (1), onde $P_t(c_j)$ é o prazo total do cenário c_j e $C_t(c_j)$ é a carga total do cenário c_j.

$$I_{pc}(c_j) = \frac{P_t(c_j)}{C_t(c_j)} \tag{1}$$

Para calcular $P_t(c_j)$ é utilizada a Eq. (2), onde $n(M(c_j))$ é o número de nós *MEC* do cenário c_j e $P_t(m_k)$ é o prazo total para o nó *MEC* m_k. Para calcular $P_t(m_k)$ é utilizada a Eq. (3), onde $n(S(m_k))$ é o número de serviços fornecidos pelo nó *MEC* m_k, $n(R(m_k,s_y))$ é o número de requisições enviadas ao nó *MEC* m_k, para o serviço s_y, e $d(s_y)$ é o prazo estabelecido para responder o serviço s_y.

$$P_t(c_j) = \sum_{k=1}^{n(M(c_j))} P_t(m_k) \tag{2}$$

$$P_t(m_k) = \sum_{y=1}^{n(S(m_k))} n(R(m_k,s_y)) \times d(s_y) \tag{3}$$

Já o cálculo da variável $C_t(c_j)$ é realizado através da Eq. (4), onde $C_t(m_k)$ é a carga total submetida ao nó *MEC* m_k. O cálculo de $C_t(m_k)$ é realizado pela Eq. (5), onde $p(s_y,m_k)$ é o tempo que o nó *MEC* m_k levará para executar o serviço s_y, o qual é o mesmo para todos os nós *MEC*, conforme apresentado pela Tabela 1.

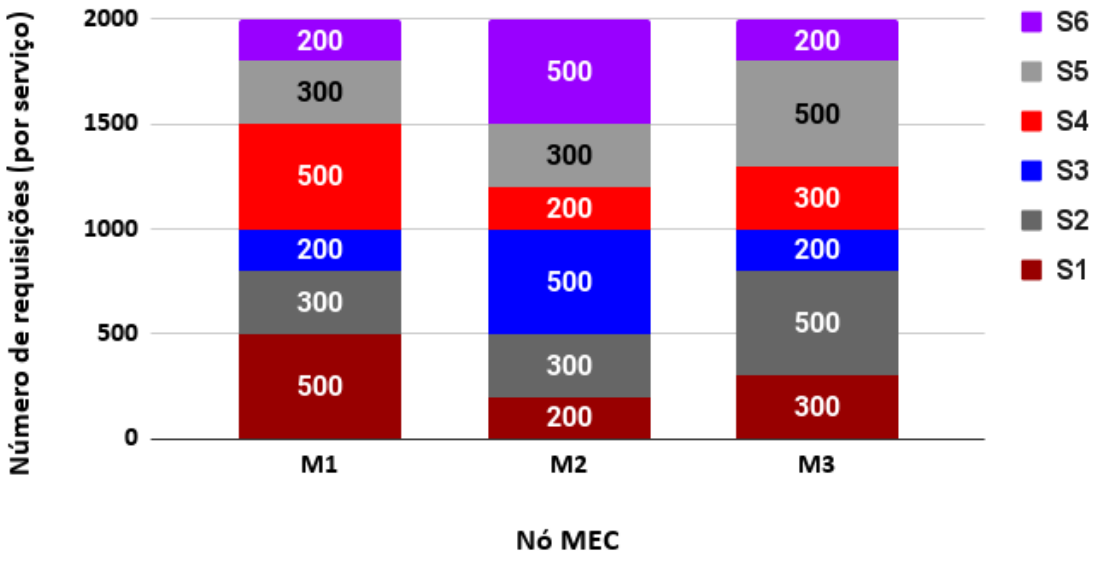
$$C_t(c_j) = \sum_{k=1}^{n(M(c_j))} C_t(m_k) \tag{4}$$

$$C_t(m_k) = \sum_{y=1}^{n(S(m_k))} n(R(m_k,s_y)) \times p(s_y,m_k) \tag{5}$$

6.1.1 Cenário 1

Conforme apresentado na Figura 30, o cenário 1 é constituído por um total de 6.000 requisições realizadas a cada um dos 3 nós *MEC*. Este cenário é aquele que possui o menor número de requisições, e de nós *MEC*, porém possui a maior carga de trabalho devido à proporção de requisições para os serviços s_1 e s_4. Conforme mostrado na Tabela 1, os serviços s_1 e s_4 são aqueles que possuem os maiores tempos de execução.

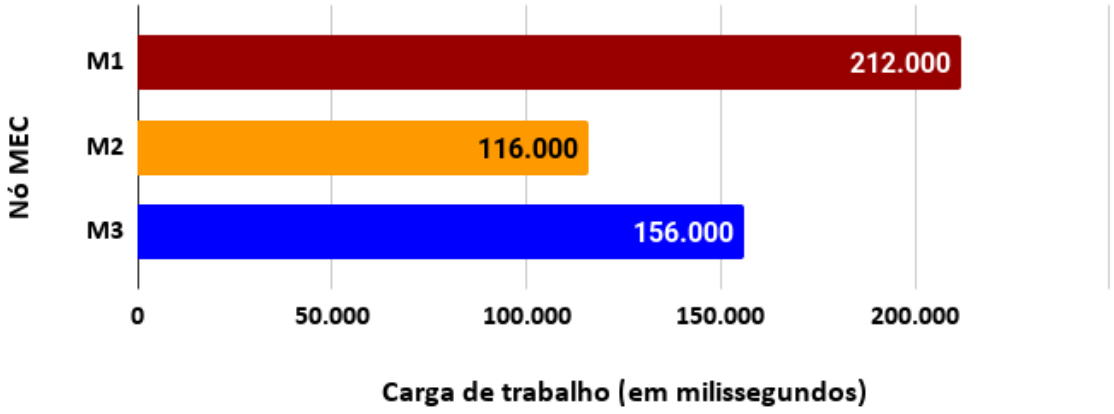
Figura 30 – Número total de requisições realizadas, no cenário 1, para os nós *MEC* m_1, m_2 e m_3.



Fonte: Elaborado pelo autor.

Conforme apresentado pela Figura 31, o nó *MEC* m_1 é aquele que deve receber o maior nível de carga no cenário 1, totalizando 212.000ms de tempo de execução. Em seguida tem-se o nó m_3, com 156.000ms, e o nó m_2, com 116.000ms. Esta diferença de carga é útil para compreender o comportamento dos modelos de orquestração de carga, visto que, para aumentar o cumprimento de prazos, o orquestrador deve possuir alguma regra de distribuição de carga entre os nós *MEC*.

Figura 31 – Comparação gráfica da carga de trabalho total, submetida ao longo de toda a simulação do cenário 1, aos nós *MEC* m_1, m_2 e m_3.



Fonte: Elaborado pelo autor.

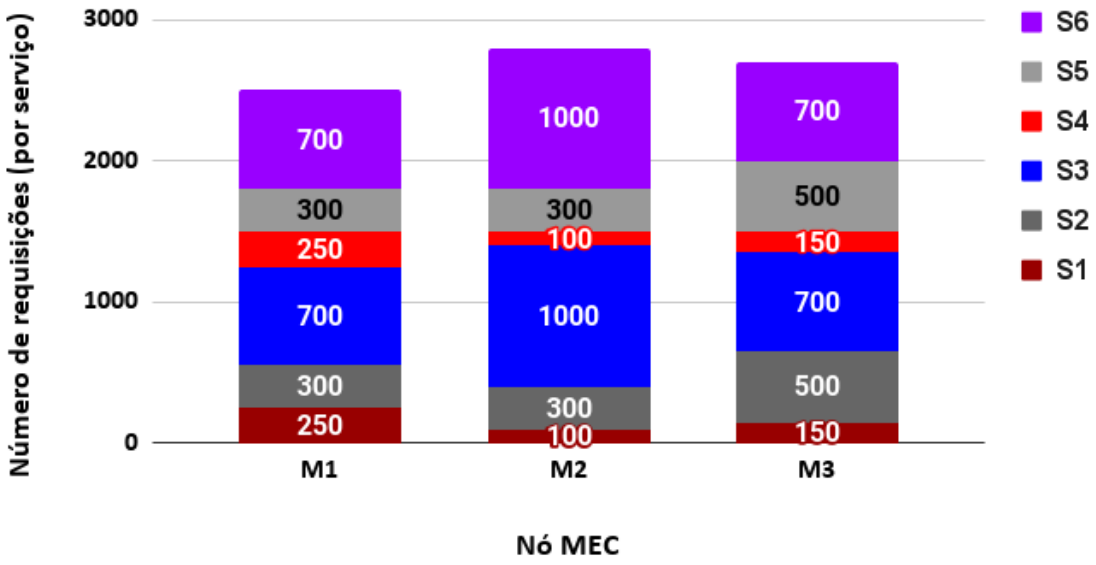
Em relação ao índice I_{pc}, o cenário 1 possui, conforme apresentado na Figura 29, o menor valor dentre todos os cenários elaborados. O baixo I_{pc} faz com que o cenário 1 seja aquele que se aproxima mais de um pico de demanda, sendo útil para compreender o comportamento dos modelos de orquestração quando os níveis de carga são altos e os prazos são curtos.

6.1.2 Cenário 2

O cenário 2 se caracteriza por possuir o mesmo número de nós *MEC* do cenário 1, porém com um número maior de requisições. Na Figura 32 é possível observar o envio de 8.000 requisições aos nós *MEC*, porém a maior parte das requisições se referem a serviços com o menor tempo de execução (s_3 e s_6). Os serviços s_1 e s_4, que demoram mais para serem executados, são os menos requisitados. Estas características resultam, ao cenário 2, o maior índice I_{pc}, conforme apresentado pela Figura 29.

Na Figura 33 é possível observar o impacto gerado pela predominância das requisições para os serviços s_3 e s_4. As cargas totais, submetidas a cada nó *MEC*, são menores do que aquelas obtidas no cenário 1. Ao nó *MEC* m_1 foi submetida uma

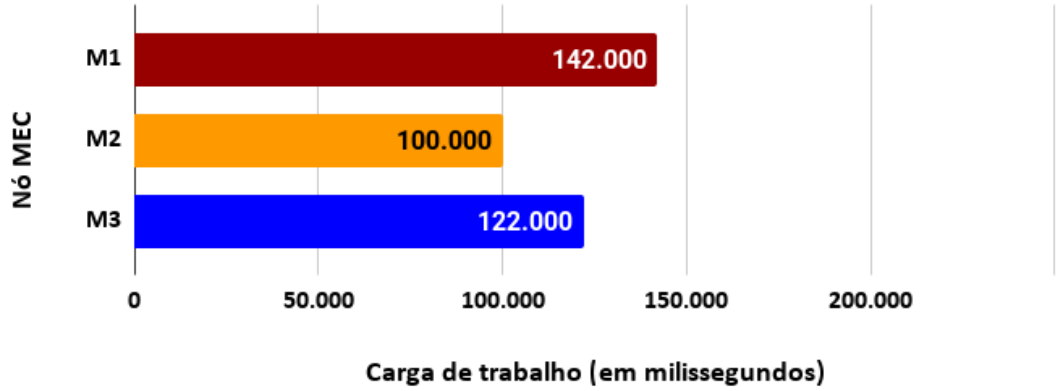
Figura 32 – Número total de requisições realizadas, no cenário 2, para os nós *MEC* m_1, m_2 e m_3.



Fonte: Elaborado pelo autor.

carga de 142.000ms, seguido do nó m_3, que recebeu 122.000ms, e por último o nó m_2, que recebeu 100.000ms de carga de trabalho.

Figura 33 – Comparação gráfica da carga de trabalho total, submetida ao longo de toda a simulação do cenário 2, aos nós *MEC* m_1, m_2 e m_3.



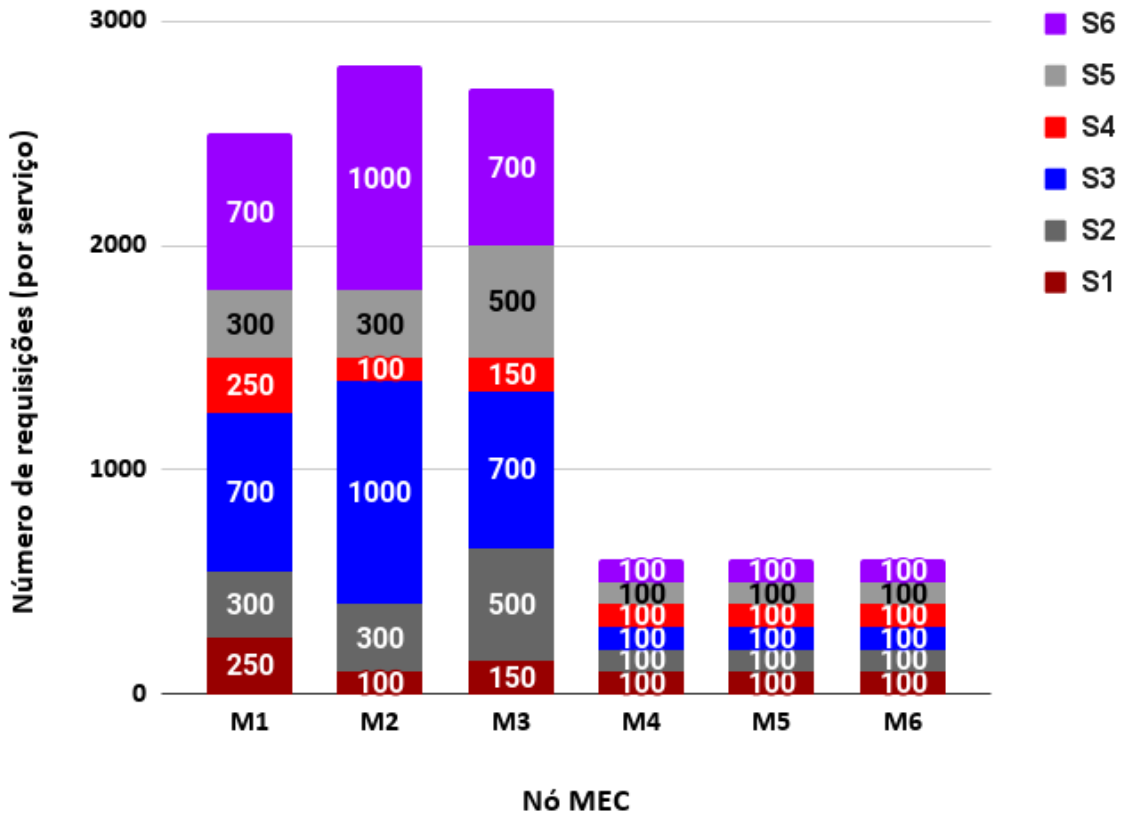
Fonte: Elaborado pelo autor.

Por possuir um nível I_{pc} maior do que o cenário 1, o cenário 2 está mais distante de um caso de pico de demanda. Isto é, a proporção de carga de trabalho é menor em relação aos prazos de resposta, tornando o cenário 2 mais distante do caso de pico de demanda. Portanto, enquanto o cenário 1 representa um caso mais crítico, o cenário 2 representa um cenário de maior flexibilidade aos modelos de orquestração de carga.

6.1.3 Cenário 3

O cenário 3 se caracteriza pelas semelhanças em relação ao cenário 2. A diferença é em relação ao número de nós *MEC*, pois no cenário 3 foram adicionados os nós m_4, m_5 e m_6. Os nós m_1, m_2 e m_3, do cenário 3, são idênticos aos do cenário 2. Na Figura 34 é possível observar o número de requisições enviadas, no cenário 3, para cada um dos 6 nós *MEC*. Para os nós m_4, m_5 e m_6, que são exclusivos do cenário 3, são enviadas 100 requisições para cada um dos 6 serviços, sendo um valor consideravelmente baixo quando comparado aos demais nós (m_1, m_2 e m_3).

Figura 34 – Número total de requisições realizadas, no cenário 3, para os nós *MEC* m_1, m_2, m_3, m_4, m_5 e m_6.

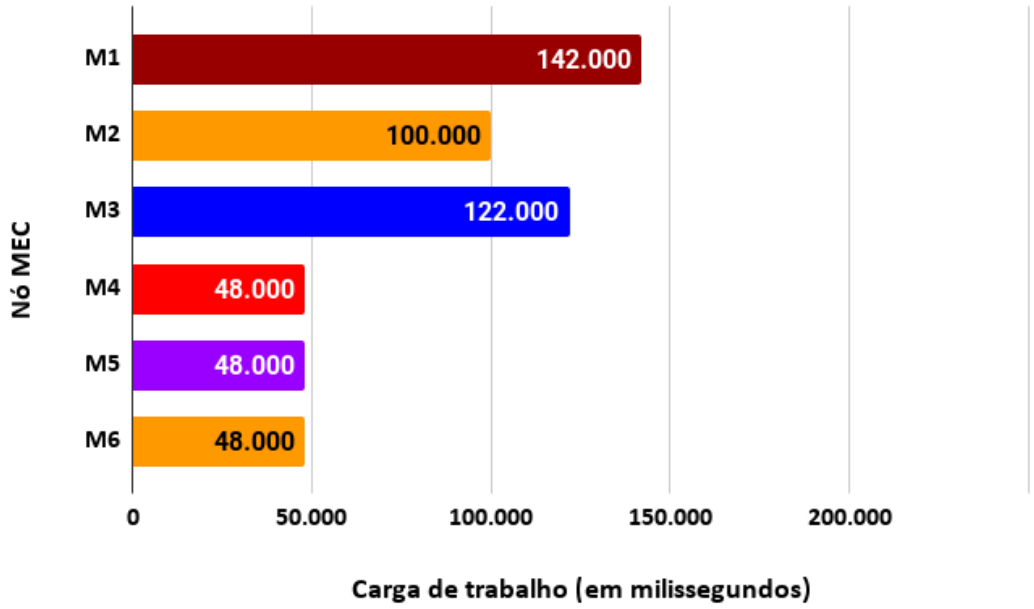


Fonte: Elaborado pelo autor.

É possível observar, na Figura 35, que o número inferior de requisições resultou, aos nós *MEC* m_4, m_5 e m_6, o recebimento de uma carga de trabalho (total) abaixo daquela recebida pelos demais nós. No entanto, considerando que, para os nós m_4, m_5 e m_6 o número de requisições ficou perfeitamente distribuído entre os tipos de serviços, a carga de trabalho ficou proporcionalmente mais elevada em relação aos demais nós. Por exemplo, o nó *MEC* m_2, que recebeu 2.800 requisições, deve receber uma carga de 100.000ms. Já os nós m_4, m_5 e m_6, que recebem individualmente 600

requisições ($\approx$ 21.43% de 2.800), possuem uma carga individual de 48.000ms (48% de 100.000ms).

Figura 35 – Comparação gráfica da carga de trabalho total, submetida ao longo de toda a simulação do cenário 3, aos nós *MEC* m_1, m_2, m_3, m_4, m_5 e m_6.



Fonte: Elaborado pelo autor.

Conforme apresentado na Figura 29, apesar de ser parecido com o cenário 2, o cenário 3 possui um índice I_{pc} menor, cujo valor está entre os obtidos no cenário 1 e cenário 2. Essa característica demonstra que, no cenário 3, há uma proporção maior de carga de trabalho em relação aos prazos de resposta, tornando o cenário 3 menos flexível para o cumprimento de prazos. Contudo, os 3 nós *MEC* adicionais (m_4, m_5 e m_6), que possuem uma carga de trabalho inferior aos demais, pode contribuir positivamente para o cumprimento dos prazos de resposta, visto que ficarão ociosos por mais tempo e poderão receber parte da carga dos demais nós *MEC*. Portanto, o cenário 3 é importante para compreender o desempenho dos modelos de orquestração quando há um alto desequilíbrio de carga entre os nós *MEC*.

7 EXPERIMENTOS E RESULTADOS

Conforme descrito no Capítulo 4, nesta dissertação é proposta a adaptação do modelo de orquestração de carga *SFA* para atuar no fornecimento de serviços a sistemas STR. Para avaliar o desempenho do modelo proposto, comparando-o com o modelo original, um simulador de carga foi elaborado para atuar como um ambiente de experimentação. O simulador é descrito no Capítulo 5, e os experimentos foram conduzidos com base em um conjunto de cenários de experimentação apresentados no Capítulo 6.

Neste capítulo são descritos os resultados obtidos após a realização da etapa experimental. A avaliação dos modelos de orquestração é realizada mediante duas métricas de desempenho, conforme é apresentado na Seção 7.1. De modo geral, os experimentos consideram os seguintes aspectos:

- os usuários enviam as requisições ao nó *MEC* mais próximo;
- os nós *MEC* possuem um prazo para retornar uma resposta;
- os serviços requisitados possuem um tempo máximo de execução conforme os recursos computacionais de cada nó *MEC*;
- os nós *MEC* são homogêneos, isto é, possuem recursos computacionais equivalentes;
- atrasos gerados pela rede foram desconsiderados, assim como os atrasos de agendamento e alocação de requisições;
- cada nó *MEC* irá processar apenas uma requisição por vez;

7.1 MÉTRICAS DE DESEMPENHO

As métricas de desempenho são utilizadas para avaliar o comportamento das abordagens durante a etapa experimental. A definição das métricas está alinhada com o objetivo geral deste trabalho, que é utilizar a orquestração de carga para garantir a qualidade dos serviços fornecidos a sistemas STR. Portanto, duas métricas são definidas: a taxa de requisições com prazo cumprido (T_{rpc}) e a taxa de encaminhamentos de requisições (T_{enc}). A primeira métrica é a principal e está relacionada ao desempenho dos sistemas STR. A segunda métrica está relacionada com a sobrecarga da rede, e, caso não fossem desprezados os atrasos da rede, possuiria influencia direta no resultado da primeira métrica. O restante desta seção se destina a explicação sobre ambas as métricas de desempenho, e no Quadro 4 são apresentadas as variáveis utilizadas para a realização dos cálculos de cada métrica.

Quadro 4 – Lista das variáveis utilizadas para o cálculo das métricas de desempenho T_{rpc} e T_{enc}.

Variável	**Significado**
c_j	Cenário j
r_i	Requisição i
m_k	Nó MEC k
M	Conjunto de nós MEC contidos na rede: $M = \{m_1, m_2, ..., m_k\}$
$E(c_j)$	Conjunto de encaminhamentos realizados no cenário c_j
$E(m_k)$	Conjunto de encaminhamentos realizados pelo nó MEC m_k
$E(r_i)$	Conjunto de encaminhamentos da requisição r_i
$R(c_j)$	Conjunto de requisições realizadas no cenário c_j
$R(m_k)$	Conjunto de requisições realizadas ao nó MEC m_k
$R_{rpc}(c_j)$	Conjunto de requisições, com prazo cumprido, realizadas no cenário c_j
$R_{rpd}(c_j)$	Conjunto de requisições, com prazo descumprido, realizadas no cenário c_j
$R_{rpc}(m_k)$	Conjunto de requisições, com prazo cumprido, realizadas ao nó MEC m_k
$R_{rpd}(m_k)$	Conjunto de requisições, com prazo descumprido, realizadas ao nó MEC m_k
L_{enc}	Número máximo de vezes que cada requisição poderá ser encaminhada
$n(A)$	Número de elementos contidos no conjunto A
$n(M)$	Número de nós MEC contidos na rede
$n(E(c_j))$	Número de encaminhamentos realizados no cenário c_j
$n(E(m_k))$	Número de encaminhamentos realizados pelo nó MEC m_k
$n(E(r_i))$	Número de encaminhamentos da requisição r_i
$n(R(c_j))$	Número de requisições realizadas no cenário c_j
$n(R(m_k))$	Número de requisições realizadas ao nó MEC m_k
$n(R_{rpc}(m_k))$	Número de requisições, com prazo cumprido, realizadas ao nó MEC m_k
$n(R_{rpd}(m_k))$	Número de requisições, com prazo descumprido, realizadas ao nó MEC m_k
$n(R_{rpc}(c_j))$	Número de requisições, com prazo cumprido, realizadas no cenário c_j
$n(R_{rpd}(c_j))$	Número de requisições, com prazo descumprido, realizadas no cenário c_j
$T_{rpc}(c_j)$	Taxa de requisições, com prazo cumprido, no cenário c_j
$T_{enc}(c_j)$	Taxa de encaminhamentos de requisições no cenário c_j

Fonte: Elaborado pelo autor.

7.1.1 Taxa de requisições com prazo cumprido

A taxa de requisições com prazo cumprido $T_{rpc}(c_j)$ é uma das métricas utilizadas para avaliar os modelos de orquestração de carga quando submetidos a cada cenário

de experimentação c_j. O cálculo é realizado através da Eq. (6), onde $R_{rpc}(c_j)$ é o conjunto de requisições respondidas dentro do prazo, no cenário c_j, $R(c_j)$ é o conjunto contendo todas as requisições realizadas no cenário c_j, e $n(A)$ é a função que retorna o tamanho do conjunto A.

$$T_{rpc}(c_j) = 100 \times \left(\frac{n(R_{rpc}(c_j))}{n(R(c_j))} \right) \tag{6}$$

Cada requisição do cenário c_j é enviada de um dispositivo de usuário para um nó *MEC* m_k. Portanto, conforme mostrado na Equação (7), o número $n(R_{rpc}(c_j))$ de requisições do cenário c_j, com prazo cumprido, é calculado através do somatório do número de requisições, com prazo cumprido, de todos os nós *MEC* m_k do cenário c_j. O cálculo do número de requisições $n(R_{rpd}(c_j))$, com prazo descumprido, segue a mesma ideia, conforme mostrado na Equação (8). Além disso, o número de requisições realizadas ao nó *MEC* m_k deve ser a soma do número de requisições com prazo cumprido ($n(R_{rpc}(m_k))$) ao número de requisições com prazo descumprido ($n(R_{rpd}(m_k))$), conforme mostrado na Equação (9).

$$n(R_{rpc}(c_j)) = \sum_{k=1}^{n(M)} n(R_{rpc}(m_k)) \tag{7}$$

$$n(R_{rpd}(c_j)) = \sum_{k=1}^{n(M)} n(R_{rpd}(m_k)) \tag{8}$$

$$n(R(m_k)) = n(R_{rpc}(m_k)) + n(R_{rpd}(m_k)) \tag{9}$$

O conjunto de todas as requisições r_i, contidas no conjunto $R(c_j)$, pertencem, de forma exclusiva, ao conjunto $R_{rpc}(c_j)$ ou $R_{rpd}(c_j)$ do cenário c_j. Isto é, o conjunto $R(c_j)$ pode ser descrito como a união das requisições contidas nos conjuntos $R_{rpc}(c_j)$ e $R_{rpd}(c_j)$, conforme apresentado na Equação (10). Portanto, para calcular o número total de requisições $n(R(c_j))$ realizadas no cenário c_j, basta-se somar o número de requisições com prazo cumprido $n(R_{rpc}(c_j))$ ao número de requisições com prazo descumprido $n(R_{rpd}(c_j))$, conforme apresentado na Equação (11), a qual é posteriormente adaptada para a Equação (12). Os valores de $n(R(c_j))$, para os cenários c_1, c_2 e c_3, também podem ser obtidos, respectivamente, através da Figura 30, Figura 32 e Figura 34.

$$R(c_j) = R_{rpc}(c_j) \cup R_{rpd}(c_j) \tag{10}$$

$$n(R(c_j)) = n(R_{rpc}(c_j)) + n(R_{rpd}(c_j)) \tag{11}$$

$$n(R(c_j)) = \sum_{k=1}^{n(M)} n(R_{rpc}(m_k)) + n(R_{rpd}(m_k)) \tag{12}$$

Com algumas adaptações, a Eq. (6) é transformada na Eq. (13) ao substituir $n(R_{rpc}(c_j))$, e $n(R(c_j))$, utilizando a Eq. (7) e Eq. (12), respectivamente. Portanto, para calcular $T_{rpc}(c_j)$ basta multiplicar por 100 o somatório da divisão do número de requisições, com prazo cumprido, de cada nó *MEC* m_k, pelo número total de requisições de cada nó *MEC* m_k.

$$T_{rpc}(c_j) = 100 \times \left(\sum_{k=1}^{n(M)} \frac{n(R_{rpc}(m_k))}{n(R_{rpc}(m_k)) + n(R_{rpd}(m_k))} \right) \tag{13}$$

7.1.2 Taxa de encaminhamentos de requisições

Para calcular $T_{enc}(c_j)$ é necessária a utilização de outras três variáveis: o número de encaminhamentos $n(E(c_j))$ realizados no cenário c_j, o número de requisições $n(R(c_j))$ realizadas no cenário c_j, e a variável L_{enc}, uma constante que representa o número máximo de encaminhamentos que podem ser realizados para cada requisição r_i. O valor L_{enc}, utilizado nos experimentos dos cenários c_1 e c_2, foi igual a 2, enquanto no cenário c_3 definiu-se $L_{enc} = 5$. O valor de $T_{enc}(c_j)$ é obtido pela Eq. (14).

$$T_{enc}(c_j) = 100 \times \left(\frac{n(E(c_j))}{L_{enc} \times n(R(c_j))} \right) \tag{14}$$

Para calcular $n(E(c_j))$ deve-se considerar que os encaminhamentos, realizados no cenário c_j, referem-se às requisições que foram, anteriormente, enviadas a cada nó *MEC* m_k. Portanto, o número de encaminhamentos, no cenário c_j, é calculado através do número de encaminhamentos em cada nó *MEC* m_k. O cálculo de $n(E(c_j))$ é realizado através da Eq. (15), onde $n(E(m_k))$ é o número de encaminhamentos realizados pelo nó *MEC* m_k. Já o cálculo de $n(E(m_k))$ corresponde ao somatório do número de encaminhamentos $n(E(r_i))$ de cada requisição r_i do nó *MEC* m_k, conforme apresentado pela Eq. (16).

$$n(E(c_j)) = \sum_{k=1}^{n(M)} n(E(m_k)) \tag{15}$$

$$n(E(m_k)) = \sum_{i=1}^{n(R(m_k))} n(E(r_i)) \tag{16}$$

Com algumas adaptações, a Equação (14) pode ser transformada na Equação (17) ao substituir $n(R(c_j))$ e $n(E(c_j))$ utilizando a Equação (9), Equação (12), Equação (15) e Equação (16).

$$T_{enc}(c_j) = \frac{100}{L_{enc}} \times \left(\sum_{k=1}^{n(M)} \frac{\sum_{i=1}^{n(R(m_k))} n(E(r_i))}{n(R(m_k))} \right) \quad (17)$$

7.2 ANÁLISE DOS RESULTADOS

Nesta seção são descritos os resultados obtidos na etapa experimental. A explicação foi organizada de acordo com cada cenário de experimentação, e para cada cenário é apresentada uma avaliação sobre o desempenho dos modelos de orquestração mediante 3 gráficos estatísticos. Através dos 2 primeiros gráficos é discutido o cumprimento dos prazos de resposta, enquanto o terceiro gráfico é utilizado para avaliar o desempenho dos orquestradores em relação ao número de encaminhamentos realizados. A seção é finalizada por meio de uma discussão geral que considera os 3 cenários de experimentação utilizados.

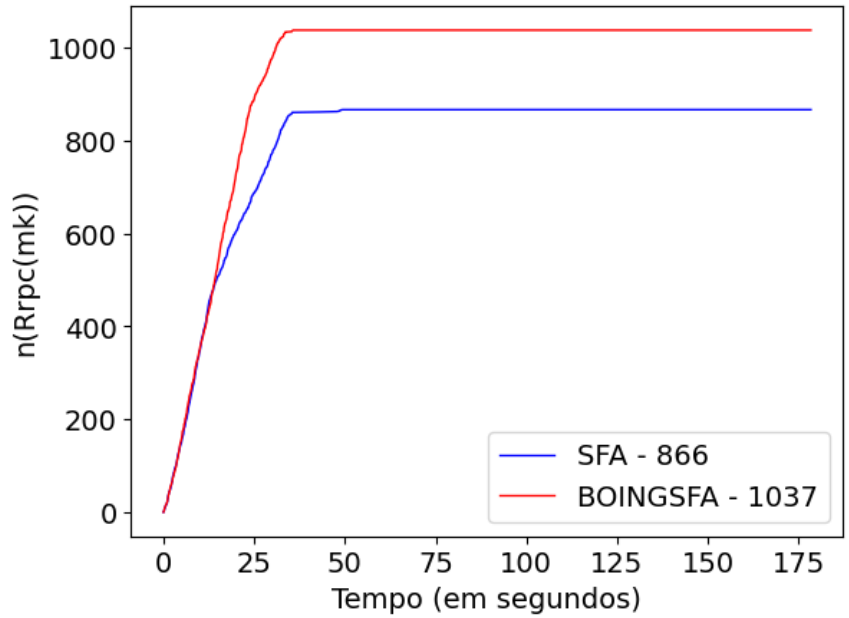
7.2.1 Cenário 1

Na Figura 36 é apresentado um gráfico que mostra o número total de requisições, de cada modelo de orquestração de carga, respondidas dentro do prazo pré-estabelecido. Das 6.000 requisições, o modelo *SFA* original, que possui a fila do tipo *FIFO*, conseguiu responder 866 requisições dentro do prazo. Já o modelo BOINGSFA, que possui a fila FEP proposta, conseguiu responder 1.037 requisições dentro do prazo. Em termos de porcentagem, calculado através da Equação (6), o modelo original *SFA* atingiu uma taxa $T_{rpc} \approx 14{,}43\%$ de prazos cumpridos, enquanto o modelo BOINGSFA, adaptado com a fila FEP, atingiu uma taxa $T_{rpc} \approx 17{,}28\%$ de requisições com prazos cumpridos. A diferença é de aproximadamente 2,85% a mais de prazos cumpridos quando o modelo BOINGSFA foi utilizado.

Também é possível notar, ao observar a Figura 36, que no início da simulação ambos os modelos de orquestração atingiram um pico de prazos cumpridos. Inicialmente, ambos os modelos apresentaram um desempenho quase idêntico, mas em seguida o modelo *SFA* reduziu o cumprimento de prazos e concluiu com um total abaixo do modelo BOINGSFA. Apesar da simulação finalizar próximo aos 175 segundos, ambos os modelos obtiveram o último cumprimento de prazo antes dos 50 segundos.

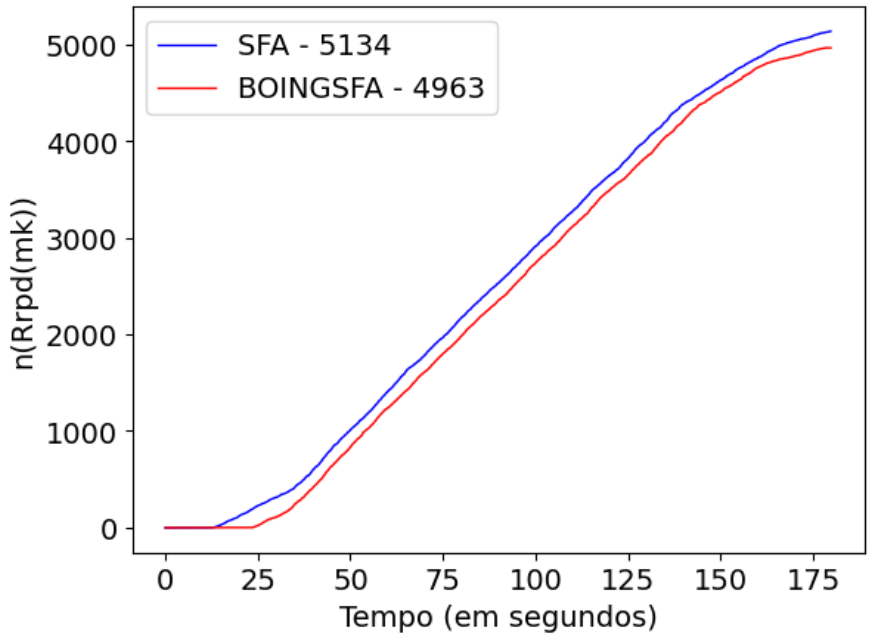
Através da Figura 37 é possível observar o número de requisições respondidas fora do prazo pré-estabelecido. No início da simulação, durante um breve período, ambos os orquestradores não apresentaram nenhuma perda de prazo. Contudo, a medida que o tempo passa, o modelo *SFA* (original) se destaca negativamente por possuir um número maior de prazos descumpridos. O modelo BOINGSFA segue um padrão parecido, porém apresenta uma perda de prazos menor.

Figura 36 – Representação gráfica do número de prazos cumpridos no cenário 1.



Fonte: Elaborado pelo autor.

Figura 37 – Representação gráfica do número de prazos descumpridos no cenário 1.

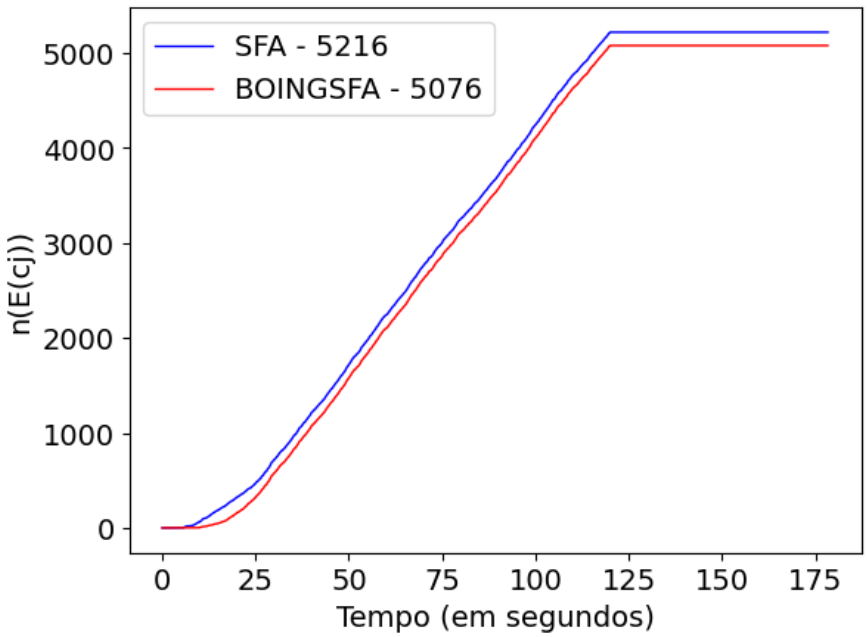


Fonte: Elaborado pelo autor.

Já na Figura 38 é apresentado um gráfico no qual é possível observar o número de encaminhamentos, de requisições, realizados quando foram utilizados os modelos de orquestração *SFA* e BOINGSFA. O modelo *SFA* (original) realizou um total de 5.216 encaminhamentos, número maior do que o gerado pelo modelo BOINGSFA,

que realizou 5.076 encaminhamentos. Ao utilizar a Equação (14) é possível observar uma taxa $T_{enc} \approx 43{,}47\%$, para o modelo *SFA*, e $T_{enc} = 42{,}30\%$, para o modelo BOINGSFA. A diferença é de aproximadamente 1,17% a mais de encaminhamentos quando se utilizou o modelo *SFA*. Para os cálculos foi considerado que o limite de encaminhamentos $L_{enc} = 2$ e o número total de requisições $n(R(c_j)) = 6.000$.

Figura 38 – Representação gráfica do número de encaminhamentos de requisições no cenário 1.



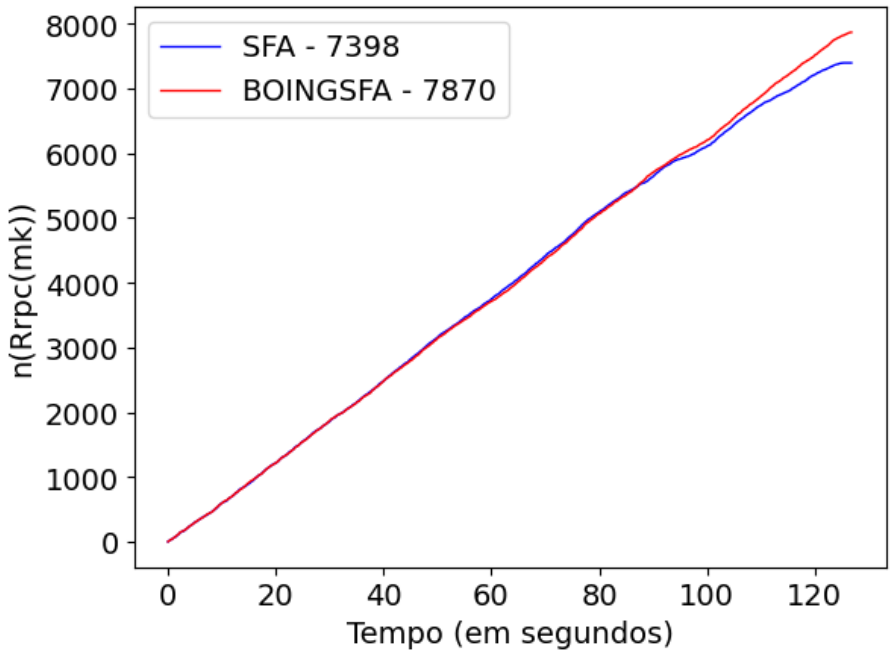
Fonte: Elaborado pelo autor.

7.2.2 Cenário 2

Na Figura 39 é apresentado um gráfico que mostra o número total de requisições, de cada modelo de orquestração de carga, respondidas dentro do prazo pré-estabelecido. Das 8.000 requisições, o modelo *SFA* original, que possui a fila do tipo *FIFO*, conseguiu responder 7.398 requisições dentro do prazo. Já o modelo BOINGSFA, que possui a fila FEP proposta, conseguiu responder 7.870 requisições dentro do prazo. Em termos de porcentagem, calculado através da Equação (6), o modelo original *SFA* atingiu uma taxa $T_{rpc} \approx 92{,}47\%$ de prazos cumpridos, enquanto o modelo BOINGSFA, adaptado com a fila FEP, atingiu uma taxa $T_{rpc} \approx 98{,}37\%$ de requisições com prazos cumpridos. A diferença é de aproximadamente 5,90% a mais de prazos cumpridos quando o modelo BOINGSFA foi utilizado.

Também é possível notar, ao observar a Figura 39, que durante a maior parte da simulação ambos os modelos de orquestração obtiveram um desempenho quase idêntico. O modelo BOINGSFA começa a conquistar um desempenho superior ao

Figura 39 – Representação gráfica do número de prazos cumpridos no cenário 2.



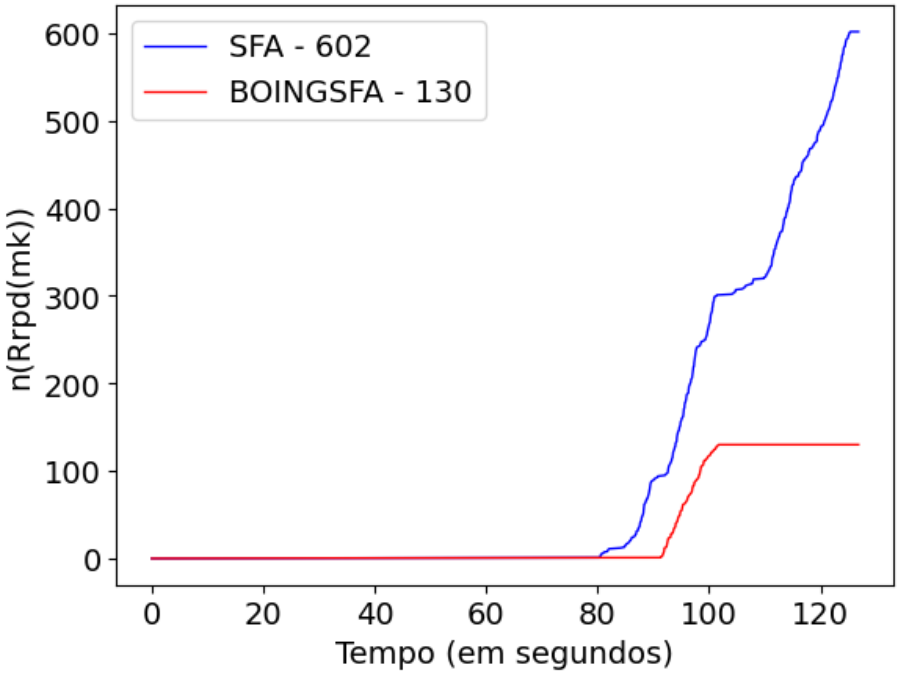
Fonte: Elaborado pelo autor.

modelo *SFA* por volta dos 100 segundos de simulação, quando o modelo *SFA* reduz o número de prazos cumpridos e se distância do modelo proposto até o término da simulação. O término da simulação ocorre por volta dos 130 segundos.

Através da Figura 40 é possível observar o número de requisições respondidas fora do prazo pré-estabelecido. Durante a maior parte da simulação ambos os modelos não registraram perdas de prazo, comportamento que perdurou até um momento entre os 80 a 100 segundos, quando o modelo *SFA* reduz o número de prazos cumpridos (Figura 39). Ainda na Figura 40, nota-se que o modelo *SFA* (original) obteve um rápido crescimento no número de prazos perdidos, enquanto o modelo BOINGSFA registrou perdas de prazo durante um curto período, o qual foi revertido na sequência, quando o modelo BOINGSFA deixou de registrar perdas de prazo até o término da simulação.

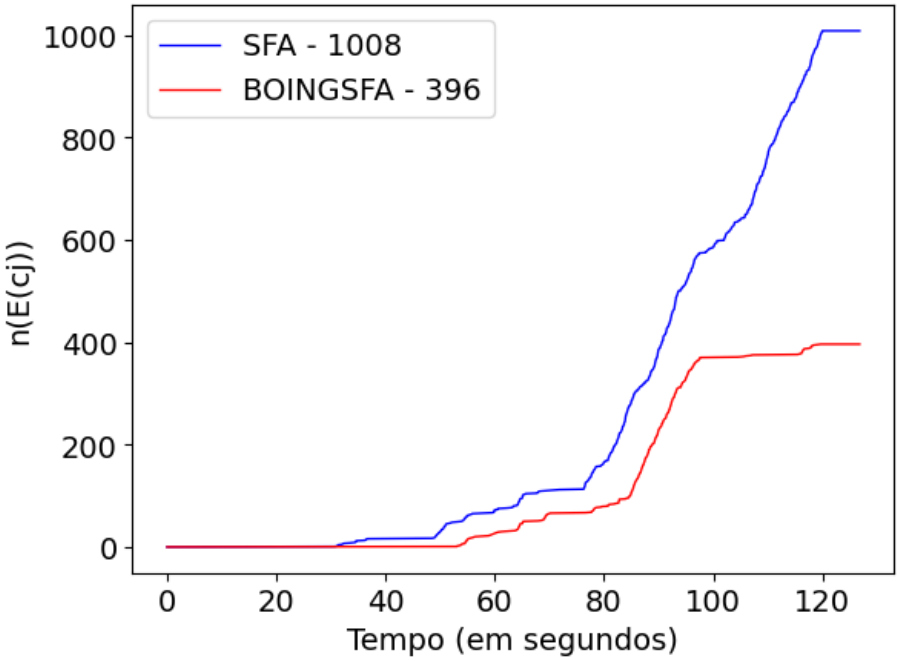
Já na Figura 41 é apresentado um gráfico no qual é possível observar o número de encaminhamentos, de requisições, realizados quando foram utilizados os modelos de orquestração *SFA* e BOINGSFA. O modelo *SFA* (original) realizou um total de 1.008 encaminhamentos, número maior do que o gerado pelo modelo BOINGSFA, que realizou 396 encaminhamentos. Ao utilizar a Equação (14) é possível observar uma taxa T_{enc} = 6,3% quando o modelo *SFA* foi utilizado, e uma taxa $T_{enc} \approx$ 2,47% quando se utilizou o modelo BOINGSFA. A diferença é de aproximadamente 3,82% a mais de encaminhamentos quando o modelo *SFA* foi utilizado. Para os cálculos foi considerado que o limite de encaminhamentos L_{enc} = 2 e o número total de requisições $n(R(c_j))$ = 8.000.

Figura 40 – Representação gráfica do número de prazos descumpridos no cenário 2.



Fonte: Elaborado pelo autor.

Figura 41 – Representação gráfica do número de encaminhamentos de requisições no cenário 2.

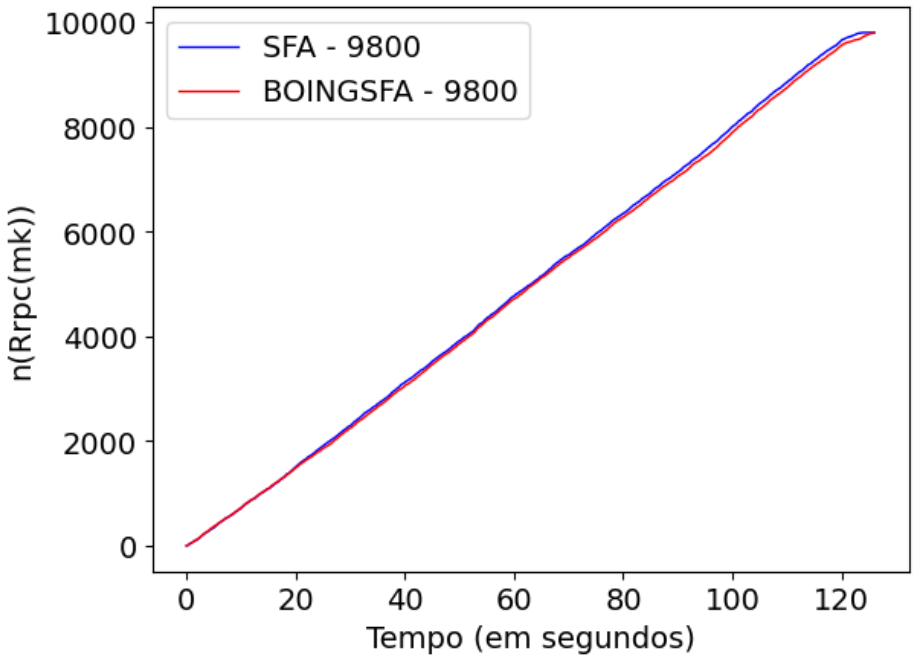


Fonte: Elaborado pelo autor.

7.2.3 Cenário 3

Na Figura 42 é apresentado um gráfico que mostra o número total de requisições, de cada modelo de orquestração de carga, respondidas dentro do prazo pré-estabelecido. Ambos os modelos, *SFA* (original) e BOINGSFA (proposto), conseguiram responder todas as 9.800 requisições dentro do prazo. Isto é, a taxa T_{rpc}, calculada através da Equação (6), foi de 100% para ambos os modelos. No entanto, nota-se que o modelo BOINGSFA obteve um desempenho levemente inferior ao obtido pelo modelo *SFA* após os 40 segundos de simulação, o qual se agravou a partir dos 80 segundos.

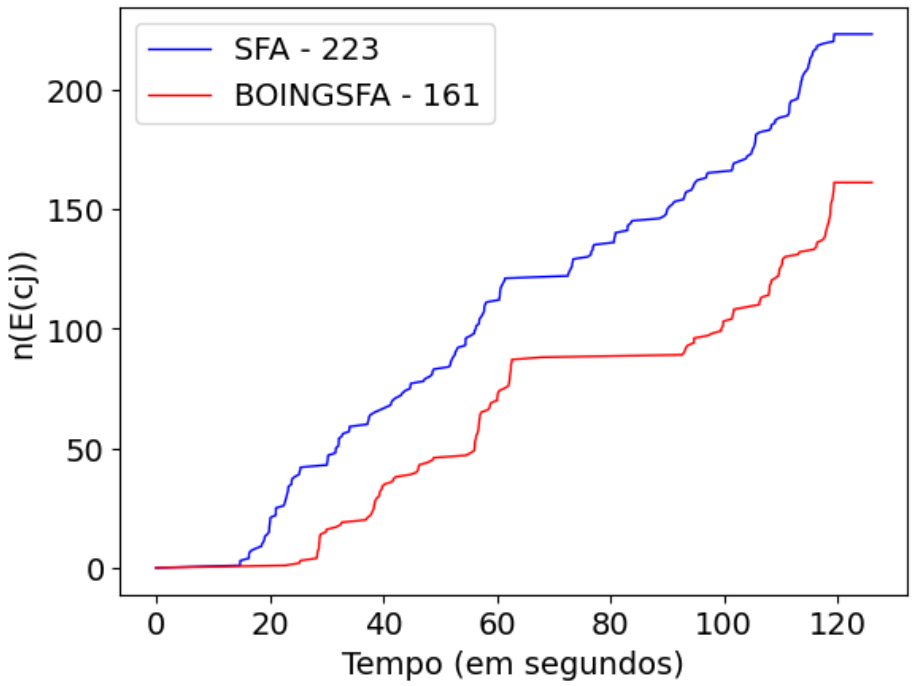
Figura 42 – Representação gráfica do número de prazos cumpridos no cenário 3.



Fonte: Elaborado pelo autor.

Através do gráfico apresentado pela Figura 43 é possível observar o número de encaminhamentos, de requisições, realizados quando foram utilizados os modelos de orquestração *SFA* e BOINGSFA. O modelo *SFA* (original) realizou um total de 223 encaminhamentos, número maior do que o gerado pelo modelo BOINGSFA, que realizou 161 encaminhamentos. Ao utilizar a Equação (14) é possível observar uma taxa $T_{enc} \approx 0{,}45\%$, para o modelo *SFA*, e $T_{enc} = 0{,}33\%$, para o modelo BOINGSFA. A diferença é de aproximadamente 0,13% a mais de encaminhamentos quando se utilizou o modelo *SFA*. Para os cálculos foi considerado que o limite de encaminhamentos $L_{enc} = 5$ e o número total de requisições $n(R(c_j)) = 9.800$.

Figura 43 – Representação gráfica do número de encaminhamentos de requisições no cenário 3.



Fonte: Elaborado pelo autor.

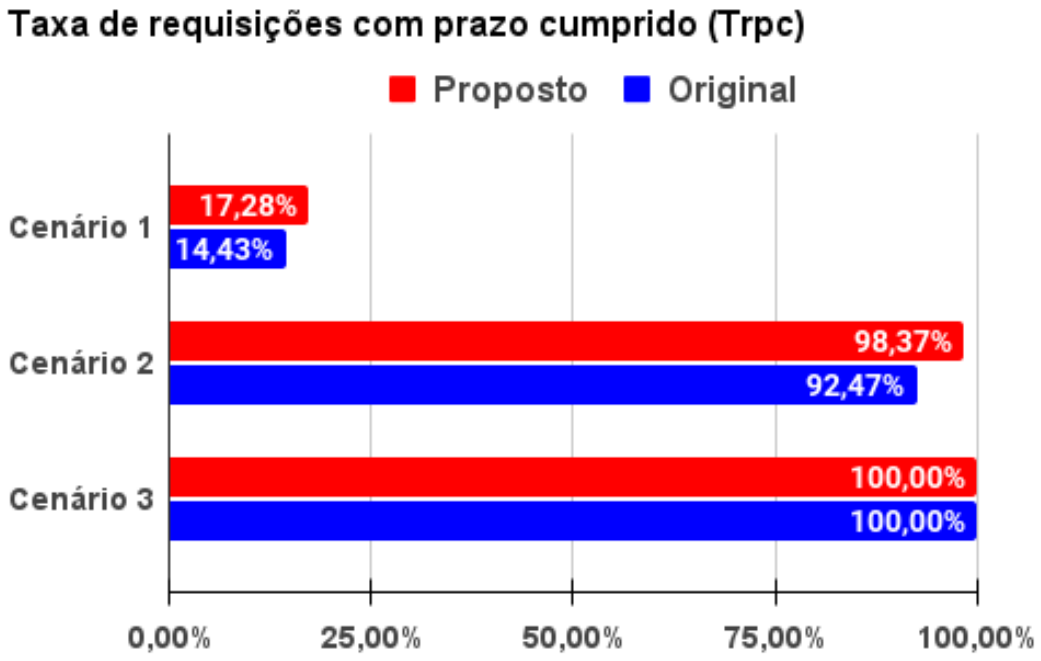
7.2.4 Discussão

No gráfico apresentado na Figura 44 é possível observar as taxas T_{rpc} obtidas pelos modelos *SFA* e BOINGSFA nos 3 cenários de experimentação. No cenário 1 os modelos atingiram uma taxa T_{rpc} abaixo dos 20%, enquanto no cenário 2 a taxa T_{rpc} foi superior a 92%, para ambos, e no cenário 3 ambos os modelos atingiram uma taxa T_{rpc} = 100%. Já na Figura 45 é possível notar uma taxa T_{enc}, para ambos os modelos, superior a 40%, no cenário 1, inferior a 6,5%, no cenário 2, e inferior a 0,5%, no cenário 3.

Conforme discutido no Capítulo 6, o cenário 1 é aquele com o menor índice I_{pc} e o menor número de nós *MEC*, tornando este cenário o mais desafiador para o cumprimento de prazos, visto que, quanto menor o índice I_{pc}, maior é a carga de trabalho total em relação ao total dos prazos de resposta. Um menor número de nós *MEC* também deve impactar o cumprimento de prazos, pois ao ficar sobrecarregado, um nó *MEC* m_k terá poucas opções de nós vizinhos para encaminhar uma determinada requisição r_i. Estas características, do cenário 1, justificam o porque de ambos os modelos atingirem um baixo número de prazos cumpridos (T_{rpc}) e um alto número de encaminhamentos de requisições (T_{enc}).

Apesar de possuir o mesmo número de nós *MEC* do cenário 1, o cenário 2 possui o maior índice I_{pc} dentre os 3 cenários de experimentação. Portanto, o alto

Figura 44 – Representação gráfica do número de prazos cumpridos em todos os cenários.

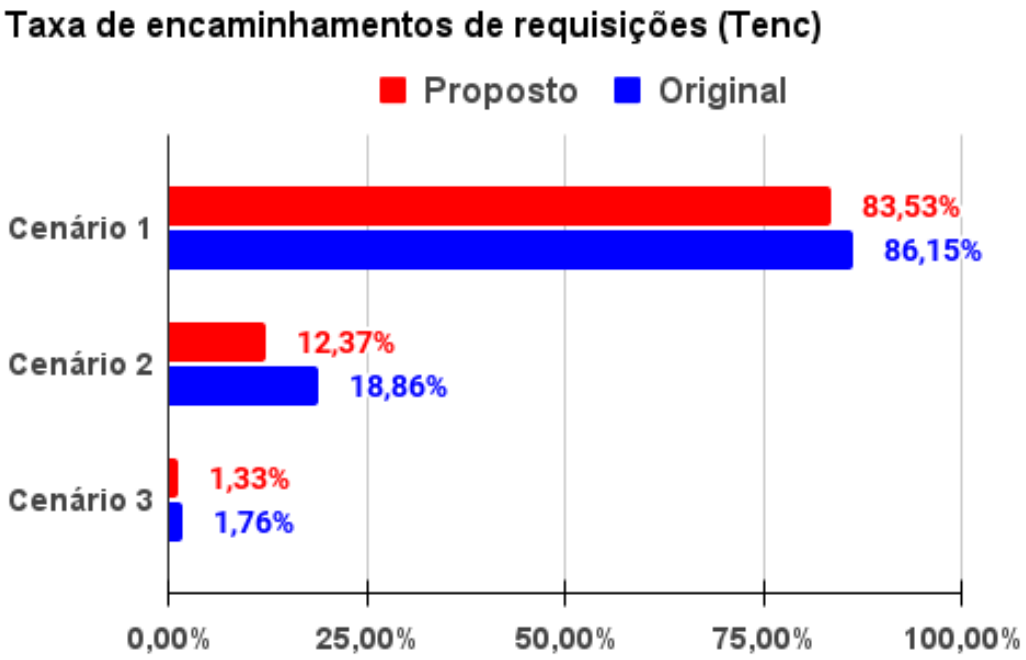


Fonte: Elaborado pelo autor.

desempenho gerado no cenário 2, quando comparado ao cenário 1, se justifica pela maior flexibilidade dos prazos de resposta em relação a carga total gerada aos nós *MEC*. O fato do cenário 3 resultar, a ambos os modelos de orquestração, o maior desempenho em relação as taxas T_{rpc} e T_{enc}, se deve ao maior número de nós *MEC* presentes na rede. Apesar de o cenário 3 possuir um índice I_{pc} menor do que o cenário 2, o maior número de nós *MEC* resultou em mais opções de nós vizinhos para onde as requisições poderiam ser encaminhadas.

De modo geral, o desempenho da abordagem proposta (BOINGSFA) foi superior a original (*SFA*) em relação à taxa de encaminhamentos (T_{enc}) e à taxa de prazos cumpridos (T_{rpc}). A superioridade da abordagem proposta é justificada porque a fila de espera preferencial (FEP), utilizada na versão proposta, se comporta como uma fila *FIFO* no pior caso. Ou seja, as requisições que teriam seu prazo perdido têm a oportunidade de serem alocadas antes de outras já alocadas. Se o prazo das demais requisições for afetado, então a estratégia *FIFO* é utilizada para alocar a requisição no final da fila. Essa possibilidade de realocação, além de aumentar o número de prazos cumpridos, também reduz o número de encaminhamentos, pois os encaminhamentos são feitos apenas quando existe a possibilidade de perda de prazo.

Figura 45 – Representação gráfica do número de encaminhamentos de requisições em todos os cenários.



Fonte: Elaborado pelo autor.

8 CONCLUSÃO

A principal contribuição deste trabalho foi a proposta de um mecanismo de alocação de requisições para atuar na fila de espera de cada nó *MEC*. Trata-se de uma pesquisa de natureza aplicada, na qual foi realizada uma adaptação de um modelo de orquestração de carga, encontrado na literatura, para ser utilizado em redes colaborativas de nós *MEC*. O objetivo foi distribuir a carga de trabalho entre os nós *MEC*, a fim de que seja possível cumprir os prazos de resposta, das requisições, que são pré-estabelecidos pelos sistemas de tempo real (STR).

Após a adaptação do modelo de orquestração de carga, que passou a utilizar o mecanismo de alocação de requisições proposto, foi considerada a metodologia de pesquisa experimental, cujo objetivo foi avaliar o desempenho do novo modelo ao orquestrar as requisições dos sistemas STR. Um simulador foi elaborado para atuar como o ambiente de experimentação, e foi posteriormente utilizado para avaliar o desempenho da abordagem proposta ao submetê-la a três cenários de experimentação distintos.

Para a obtenção dos resultados experimentais, duas métricas de desempenho foram definidas, sendo elas: a taxa de requisições com prazo cumprido, e a taxa de encaminhamentos de requisições. Quando comparada com a abordagem original, a nova versão, que possui o mecanismo de alocação proposto, conseguiu aumentar o cumprimento de prazos em até 5,90%. Já em relação ao número de encaminhamentos, a nova versão conseguiu atingir uma redução de até 3,82%, a qual contribuirá para evitar o sobrecarregamento da rede.

8.1 TRABALHOS FUTUROS

A continuação da pesquisa deve considerar a proposta de uma segunda versão do mecanismo de alocação de requisições, considerando aspectos como a complexidade computacional dos algoritmos, que deve ser analisada de forma teórica e experimental. Outros mecanismos de alocação também devem ser utilizados como base comparativa, como o algoritmo *Earliest Deadline First (EDF)*, específico para sistemas STR. Além disso, estudos devem ser realizados considerando outros tipos de sistemas STR, como o *Firm* e *Hard*, que possuem maiores restrições a perda de prazo quando comparado ao tipo *Soft*.

Para a etapa experimental, é importante serem considerados aspectos como a latência de rede, indisponibilidade energética, falha dos nós *MEC*, novas topologias de rede, e o compartilhamento dos recursos computacionais com outras aplicações do sistema operacional. Outros aspectos também devem ser considerados, como a colaboração dos data centers da *Cloud*, novos cenários de experimentação, e um ambiente de experimentação mais realista, utilizando dispositivos de *hardware*. Também se faz

importante um estudo sobre os impactos de cada arquitetura de orquestração de carga, as quais foram identificadas nesta dissertação de mestrado após uma análise sobre os trabalhos correlatos. Estas arquiteturas são: arquitetura centralizada, centralizada por região, e descentralizada. Trabalhos futuros devem considerar os impactos que cada arquitetura deve gerar no sobrecarregamento da rede, na latência de comunicação entre os nós, no cumprimento dos prazos de resposta, e na escolha do nó *MEC* que processará cada requisição.

REFERÊNCIAS

AUSTAD, Henrik; JELLUM, Erling Rennemo; HENDSETH, Sverre; MATHISEN, Geir; BRYNE, Torleiv Haaland; GREGERTSEN, Kristoffer Nyborg; ALBREKTSEN, Sigurd Morkved; HELVIK, Bjarne Emil. Composable distributed real-time systems with deterministic network channels. **Journal of Systems Architecture**, Elsevier, p. 102853, 2023.

BARUAH, Sanjoy K; HARITSA, Jayant R. Scheduling for overload in real-time systems. **IEEE Transactions on computers**, IEEE, v. 46, n. 9, p. 1034–1039, 1997.

BATISTA, Ernando; FIGUEIREDO, Gustavo; PEIXOTO, Maycon; SERRANO, Martin; PRAZERES, Cássio. Load balancing in the fog of things platforms through software-defined networking. *In*: IEEE. 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). [*S.l.*: *s.n.*], 2018. p. 1785–1791.

BATISTA, Ernando; FIGUEIREDO, Gustavo; PRAZERES, Cassio. Load balancing between fog and cloud in fog of things based platforms through software-defined networking. **Journal of King Saud University-Computer and Information Sciences**, Elsevier, v. 34, n. 9, p. 7111–7125, 2022.

BERALDI, Roberto; CANALI, Claudia; LANCELLOTTI, Riccardo; MATTIA, Gabriele Proietti. A random walk based load balancing algorithm for fog computing. *In*: IEEE. 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC). [*S.l.*: *s.n.*], 2020. p. 46–53.

BERALDI, Roberto; CANALI, Claudia; LANCELLOTTI, Riccardo; MATTIA, Gabriele Proietti. Distributed load balancing for heterogeneous fog computing infrastructures in smart cities. **Pervasive and Mobile Computing**, Elsevier, v. 67, p. 101221, 2020.

BERALDI, Roberto; CANALI, Claudia; LANCELLOTTI, Riccardo; PROIETTI MATTIA, Gabriele. Randomized load balancing under loosely correlated state information in fog computing. *In*: PROCEEDINGS of the 23rd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems. [*S.l.*: *s.n.*], 2020. p. 123–127.

BÍBLIA. Provérbios. *In*: Bíblia Sagrada. Tradução de João Ferreira de Almeida. Revista e Atualizada no Brasil. 2ª Edição. Barueri - SP: Sociedade Bíblica do Brasil, 1993.

CICCONETTI, Claudio; CONTI, Marco; PASSARELLA, Andrea. Uncoordinated access to serverless computing in MEC systems for IoT. **Computer Networks**, Elsevier, v. 172, p. 107184, 2020.

DE SOUSA, Nathan F Saraiva; PEREZ, Danny A Lachos; ROSA, Raphael V; SANTOS, Mateus AS; ROTHENBERG, Christian Esteve. Network service orchestration: A survey. **Computer Communications**, Elsevier, v. 142, p. 69–94, 2019.

DEBAUCHE, Olivier; MAHMOUDI, Said; MANNEBACK, Pierre; LEBEAU, Frederic. Cloud and distributed architectures for data management in agriculture 4.0: Review and future trends. **Journal of King Saud University-Computer and Information Sciences**, Elsevier, v. 34, n. 9, p. 7494–7514, 2022.

GĄSIOR, Jakub; SEREDYŃSKI, Franciszek. A Sandpile cellular automata-based scheduler and load balancer. **Journal of computational science**, Elsevier, v. 21, p. 460–468, 2017.

IORGA, Michaela; FELDMAN, Larry; BARTON, Robert; MARTIN, Michael J.; GOREN, Nedim; MAHMOUDI, Charif. Fog Computing Conceptual Model. National Institute of Standards e Technology, 2018.

KOPETZ, Hermann; STEINER, Wilfried. **Real-time systems: design principles for distributed embedded applications**. [*S.l.*]: Springer, 2011.

LEE, Jinkyu. Schedulability Performance Improvement via Task Split in Real-Time Systems. **Journal of Systems Architecture**, Elsevier, v. 129, p. 102613, 2022.

LIAQAT, Misbah; CHANG, Victor; GANI, Abdullah; AB HAMID, Siti Hafizah; TOSEEF, Muhammad; SHOAIB, Umar; ALI, Rana Liaqat. Federated cloud resource management: Review and discussion. **Journal of Network and Computer Applications**, Elsevier, v. 77, p. 87–105, 2017.

LIUTKEVIČIUS, Agnius; MORKEVIČIUS, Nerijus; VENČKAUSKAS, Algimantas; TOLDINAS, Jevgenijus. Distributed Agent-Based Orchestrator Model for Fog Computing. **Sensors**, MDPI, v. 22, n. 15, p. 5894, 2022.

MALIK, Swati; GUPTA, Kamali; GUPTA, Deepali; SINGH, Aman; IBRAHIM, Muhammad; ORTEGA-MANSILLA, Arturo; GOYAL, Nitin; HAMAM, Habib. Intelligent load-balancing framework for fog-enabled communication in healthcare. **Electronics**, MDPI, v. 11, n. 4, p. 566, 2022.

MAO, Yuyi; YOU, Changsheng; ZHANG, Jun; HUANG, Kaibin; LETAIEF, Khaled B. A survey on mobile edge computing: The communication perspective. **IEEE communications surveys & tutorials**, IEEE, v. 19, n. 4, p. 2322–2358, 2017.

MELL, Peter; GRANCE, Timothy. The NIST definition of cloud computing. National Institute of Standards e Technology, 2011.

EL-NATTAT, Amal; ELKAZZAZ, Sahar; EL-BAHNASAWY, Nirmeen A; EL-SAYED, Ayman. Performance improvement of fog environment using deadline based scheduling algorithm. *In*: IEEE. 2021 International Conference on Electronic Engineering (ICEEM). [*S.l.*: *s.n.*], 2021. p. 1–6.

PEREIRA, Eder Paulo; PADOIN, Edson Luiz; MEDINA, Roseclea Duarte; MÉHAUT, Jean-Francois. Increasing the efficiency of fog nodes through of priority-based load balancing. *In*: IEEE. 2020 IEEE Symposium on Computers and Communications (ISCC). [*S.l.*: *s.n.*], 2020. p. 1–6.

RASHID, Zryan Najat; ZEBARI, Subhi RM; SHARIF, Karzan Hussein; JACKSI, Karwan. Distributed cloud computing and distributed parallel computing: A review. *In*: IEEE. 2018 International Conference on Advanced Science and Engineering (ICOASE). [*S.l.*: *s.n.*], 2018. p. 167–172.

SABELLA, D. *et al.* **Developing Software for Multi-Access Edge Computing**. [*S.l.*: *s.n.*], 2019. Disponível em: `https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp20ed2_MEC_SoftwareDevelopment.pdf`.

SAMIR, Rasha; EL-HENNAWY, Hadia; EL-BADAWY, Hesham M. Orchestration of MEC Computation Jobs and Energy Consumption Challenges in 5G and Beyond. **IEEE Access**, IEEE, v. 10, p. 18645–18652, 2022.

TAHMASEBI-POUYA, Niloofar; SARRAM, Mehdi-Agha; MOSTAFAVI, Seyedakbar. A Blind Load-Balancing Algorithm (BLBA) for Distributing Tasks in Fog Nodes. **Wireless Communications and Mobile Computing**, Hindawi, v. 2022, 2022.

TOCZÉ, Klervie; NADJM-TEHRANI, Simin. ORCH: Distributed orchestration framework using mobile edge devices. *In*: IEEE. 2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC). [*S.l.*: *s.n.*], 2019. p. 1–10.

WU, Chunrong; PENG, Qinglan; XIA, Yunni; JIN, Yong; HU, Zhentao. Towards cost-effective and robust AI microservice deployment in edge computing environments. **Future Generation Computer Systems**, Elsevier, v. 141, p. 129–142, 2023.

ZHANG, Hongxia; YANG, Yongjin; SHANG, Bodong; ZHANG, Peiying. Joint Resource Allocation and Multi-Part Collaborative Task Offloading in MEC Systems. **IEEE Transactions on Vehicular Technology**, IEEE, v. 71, n. 8, p. 8877–8890, 2022.