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

DANIEL CARDOSO CRUZ DE SOUZA
GUSTAVO MUZY FRAGA

ANÁLISE DO CONSUMO DE ENERGIA EM DIFERENTES IMPLEMENTAÇÕES DO ALGORITMO DE CONSENSO RAFT EM SISTEMAS DISTRIBUÍDOS

RIO DE JANEIRO
2025

DANIEL CARDOSO CRUZ DE SOUZA
GUSTAVO MUZY FRAGA

ANÁLISE DO CONSUMO DE ENERGIA EM DIFERENTES IMPLEMENTAÇÕES DO ALGORITMO DE CONSENSO RAFT EM SISTEMAS DISTRIBUÍDOS

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

Orientadora: Prof. Silvana Rossetto

RIO DE JANEIRO
2025

Souza, Daniel Cardoso Cruz de
S729a Análise do consumo de energia em diferentes implementações do algoritmo de consenso raft em sistemas distribuídos / Daniel Cardoso Cruz de Souza e Gustavo Muzy Fraga. – Rio de Janeiro, 2025.

51 f.

Orientadora: Silvana Rossetto.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação)- Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Consumo de energia. 2. Raft. 3. Consenso. 4. Sistemas distribuídos. I. Fraga, Gustavo Muzy. II. Rossetto, Silvana (Orient.). III. Universidade Federal do Rio de Janeiro, Instituto de Computação. IV. Título.

DANIEL CARDOSO CRUZ DE SOUZA
GUSTAVO MUZY FRAGA

ANÁLISE DO CONSUMO DE ENERGIA EM DIFERENTES IMPLEMENTAÇÕES DO ALGORITMO DE CONSENSO RAFT EM SISTEMAS DISTRIBUÍDOS

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

Aprovado em 16 de dezembro de 2025

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 05/02/2026 12:35:04-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto
D. Sc. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
MARCOS TOMAZZOLI LEIPNITZ
Data: 05/02/2026 16:39:17-0300
Verifique em https://validar.iti.gov.br

Marcos Tomazzoli Leipnitz
D. Sc. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
CAROLINA GIL MARCELINO
Data: 05/02/2026 16:08:57-0300
Verifique em https://validar.iti.gov.br

Carolina Gil Marcelino
D. Sc. (Instituto de Computação - UFRJ)

Eu, Gustavo Muzy dedico este TCC a Rhyan Sena Seabra, cuja ausência é e sempre será um monumento a todas as boas experiências que vivemos juntos, em sua breve, porém marcante, passagem. Que esta saudade que sinto hoje seja a prova de que foi um grande amigo. Esta dedicatória é o mais próximo que consigo expressar com palavras o que sinto.

(Gustavo Muzy Fraga)

Dedico esta jornada àqueles que sempre estiveram comigo desde o início, meus pais Everaldo e Cristiane, que nunca pararam de buscar um futuro melhor para seus filhos, por caminhos que eles mesmos não puderam percorrer. Dedico também ao meu irmão, Christian, que dentre as brincadeiras da juventude encontrava tempo para ir me buscar na escola de bicicleta, e à minha irmã, Evelyn, a quem eu tentava ser exemplo, mas que se formou antes de mim.

Por fim, mas não menos importante, dedico à minha parceira e esposa, Nicole, que fez o papel principal na mudança da minha vida no decorrer da minha graduação, e é mãe dos dois pequenos a quem dedico também esse trabalho, e também fazem parte dessa jornada, o Felix e o Oliver. Talvez um pouco da minha paixão por números e computadores continue com eles.

(Daniel Cardoso Cruz de Souza)

AGRADECIMENTOS

Agradecemos ao Instituto de Computação da UFRJ pela formação acadêmica que nos trouxe até aqui e nos abriu diversas portas, academicamente e profissionalmente. Agradecemos aos professores e orientadores que, com muita paciência, continuaram nos guiando, mesmo depois de nós termos ultrapassado o tempo comum de formação.

"Não há longa noite que não encontre um dia."

William Shakespeare

RESUMO

O consumo de energia decorrente de sistemas de informação e comunicação cresce ano após ano, o que provoca a demanda por uso mais eficiente dessa energia para que esses sistemas sejam mais sustentáveis. Programas de larga escala que funcionam de forma distribuída são agentes principais no consumo de energia dos sistemas de informação modernos, e algoritmos de consenso distribuídos compõem parte fundamental desses programas. Motivado pela busca de abordagens energeticamente mais eficientes e sustentáveis, este trabalho analisa comparativamente o consumo energético associado à execução de diferentes implementações do algoritmo de consenso distribuído Raft, que é utilizado em sistemas distribuídos modernos. As implementações são avaliadas sob um teste de carga unificado, seu consumo de energia é medido com a ferramenta Alumet e os resultados são armazenados num banco de dados InfluxDB. Os resultados demonstram que cada implementação provoca uma sobrecarga diferente no consumo de energia da aplicação distribuída testada, que representa um aumento considerável no consumo total. Concluí-se que a escolha da implementação não pode ser negligenciada, pois pode representar um grande aumento de consumo em aplicações onde o Raft é comumente usado.

Palavras-chave: consumo de energia; raft; consenso; sistemas distribuídos.

ABSTRACT

The energy consumption resulting from information and communication systems grows each year, driving the demand for more efficient energy use to make these systems more sustainable. Large-scale programs that work in distributed manner are key agents of energy consumption from modern information systems, and distributed consensus algorithms are fundamental to these programs. Motivated by the search for energy-efficient and sustainable approaches, this work comparatively analyzes the energy consumption associated with running different implementations of the Raft distributed consensus algorithm, which is used in modern distributed systems. The implementations are evaluated under a unified load test; their energy consumption is measured with the Alumet tool, and the results are stored in the InfluxDB time series database. The results demonstrate that each implementation introduces a different overhead in the energy consumption of the distributed application evaluated, representing a considerable increase in total energy consumption. It is concluded that the choice of implementation cannot be neglected, as it may lead to a substantial increase in energy consumption in applications where Raft is commonly used.

Keywords: energy consumption; raft; consensus; distributed systems.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

CSV	*Comma-Separated Values*
EDP	*Energy-Delay Product*
EET	*Energy Efficiency Tester*
GSMO	*Green Software Measurement Ontology*
GSMP	*Green Software Measurement Process*
RAPL	*Running Average Power Limit*

SUMÁRIO

1 INTRODUÇÃO

O consumo de energia por sistemas de informação e comunicação cresce ano após ano, e continuará crescendo nos próximos anos, de acordo com Gelenbe e Caseau (2015). Diante desse crescimento, e considerando que grande parte dessa energia consumida provém de fontes não renováveis e por isso é responsável por uma grande quantidade de emissões de dióxido de carbono (CO_2), vê-se a necessidade de garantir que essa energia seja utilizada de forma eficiente, para que o uso de fontes não renováveis possa ser reduzido e assim todo o processo tornar-se mais sustentável.

Um dos caminhos tradicionais de reduzir esse consumo e seu impacto ambiental em sistemas de computação é otimizando os componentes físicos do sistema (por exemplo, com componentes físicos mais eficientes, ou resfriamento mais adequado). Nos últimos anos, pesquisas como as de Pinto e Castor (2017), as de Noureddine et al. (2012), e as de Sahin, Pollock e Clause (2014) trouxeram à tona que os programas executados em um computador são um fator importante ao considerar o consumo de energia de um sistema. Pesquisas como as de Manotas et al. (2016) mostraram que desenvolvedores de software experientes em sistemas de computador com requisitos de consumo de energia se consideram influenciados por preocupações com o consumo de energia ao desenvolver novos programas. Outra informação importante obtida é que esses desenvolvedores experientes consideram que a minimização do consumo de energia em um sistema de computação não é de responsabilidade única dos componentes físicos utilizados: o consenso é que o sistema operacional e os programas utilizados também possuem parte da responsabilidade de reduzir o consumo energético.

1.1 CONSENSO DISTRIBUÍDO SUSTENTÁVEL

Um sistema distribuído, de acordo com van Steen e Tanenbaum (2023), é um sistema de computação em rede no qual processos e recursos são espalhados de forma suficiente em múltiplos computadores de forma a atender alguma demanda, normalmente de escalabilidade. Um exemplo desse tipo de demanda é o serviço de envio de e-mails Google Mail, que é normalmente utilizado nos computadores pessoais de seus usuários indicando o servidor de e-mail de saída do Google para realizar os envios. Na prática, é improvável que apenas um servidor seja capaz de atender as demandas de envio de e-mail dos usuários do Google Mail, que em 2022 foi estimada em 2 bilhões de usuários — a realidade é que esse serviço inteiro foi implementado com vários servidores trabalhando em conjunto, o que forma um sistema distribuído. Essa estrutura de sistema, que envolve vários computadores em rede, o expõe a diferentes problemas que não existem ou são resolvidos muito facilmente em alternativas não distribuídas. Nesse contexto, um dos problemas que precisa ser tratado

é o problema do consenso distribuído, que garante que computadores que compõem um sistema maior possam concordar em valores ou cálculos mesmo quando alguns dos computadores envolvidos falhem. Vários algoritmos foram desenvolvidos para resolver esse problema, como Paxos, Viewstamped Replication e Raft (ONGARO; OUSTERHOUT, 2014), e essas soluções tornaram-se peças fundamentais dos sistemas de larga escala do mundo moderno (WIKIPEDIA CONTRIBUTORS, 2025a) (WIKIPEDIA CONTRIBUTORS, 2025b). Dentre esses algoritmos, Raft resolve o problema do consenso elegendo um líder dentre os computadores participantes e o responsabiliza por todas as operações realizadas no conjunto de computadores. Ele é comumente utilizado como uma biblioteca de programação (ONGARO, 2025), e por isso possui diversas implementações populares em várias linguagens de programação, estando a cargo dos desenvolvedores escolher uma pronta ou desenvolver a própria implementação para resolver o problema do consenso em seus programas.

Diante da variedade de implementações de algoritmos de consenso para uso dos desenvolvedores em seus programas — disponibilizados na forma de bibliotecas de programação — e do fato que sistemas de computação distribuídos de média e larga escala podem demandar alto consumo de energia, realizamos uma análise comparativa do consumo de energia de diferentes implementações populares do algoritmo Raft para auxiliar desenvolvedores conscientes do impacto ambiental dos seus programas na escolha por bibliotecas de programação mais sustentáveis. Analisamos o consumo de energia de cinco implementações populares do algoritmo Raft nas linguagens de programação Go e Rust e concluímos que as implementações avaliadas consomem quantidades muito diferentes de energia e desempenham um papel importante no consumo de energia total dos sistemas aos quais fazem parte.

1.2 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta trabalhos que abordam a questão de como avaliar o consumo energético de um programa de computador e que serviram como motivação para este trabalho. O Capítulo 3 descreve o problema do consenso distribuído e o algoritmo Raft em sua forma original, de acordo com Ongaro e Ousterhout (2014). Ele também explica tipos de falhas que o consenso distribuído busca combater, e apresenta o Raft como solução para as falhas não-bizantinas.

Os capítulos seguintes são mais práticos: o Capítulo 4 detalha o procedimento empírico dos testes realizados. A escolha das implementações do algoritmo Raft é explicada e a metodologia da geração e coleta das métricas de consumo de recursos de hardware e energia é elaborada. O Capítulo 5 detalha os resultados obtidos pelo trabalho e os analisa, indicando o que a diferença entre o consumo de energia de cada implementação estudada pode revelar. O Capítulo 6 conclui o trabalho e apresenta possíveis ameaças à validade

das informações obtidas, assim como possíveis trabalhos futuros.

2 EFICIÊNCIA ENERGÉTICA NA COMPUTAÇÃO

Com o grande crescimento do uso de sistemas de informação e comunicação nas últimas décadas, estudos sobre o consumo de energia desses sistemas e seu impacto ambiental também estão crescendo, e por isso há muitos trabalhos relacionados a esse tema desenvolvidos nos últimos anos. Sobre a área de forma geral, o trabalho de Pinto e Castor (2017) detalha como o campo de pesquisa em eficiência energética de programas de computador está evoluindo nos últimos anos, e cita os problemas de falta de ferramentas para aferição confiável do consumo de energia e falta de conhecimento dos desenvolvedores sobre como medir e otimizar esse consumo. O trabalho de Poy et al. (2025) realiza um mapeamento sistemático do impacto no consumo de energia de padrões de design, técnicas de refatoração, e problemas de coesão (*code smells*) em aplicações.

Dentre as publicações mapeadas por Poy et al. (2025) e outras que encontramos, consideramos útil organizá-las em três categorias:

a) Trabalhos que investigam a viabilidade da ação de medição do consumo de energia em si, desenvolvem ferramentas para utilização nesse processo, formalizam conceitos utilizados na literatura, e especificam processos para aumentar a precisão dos resultados obtidos por pesquisas relacionadas à consumo de energia em computadores.

b) Trabalhos que investigam o impacto no consumo de energia de implementações em um contexto geral de desenvolvimento de sistemas de computação ou em contextos específicos, como o de dispositivos móveis.

c) Trabalhos que investigam o consumo de energia de programas, ferramentas, ou implementações de forma comparativa, de modo a avaliar a eficiência energética de cada alternativa.

As seções a seguir listam algumas publicações de cada uma dessas categorias que motivaram o estudo empírico deste trabalho e que foram instrumentais na sua elaboração.

2.1 PESQUISAS SOBRE A AFERIÇÃO DO CONSUMO DE ENERGIA

A pesquisa de Ournani et al. (2020) investiga os fatores que podem causar ou aumentar a variabilidade de consumo de energia em avaliações comparativas de sistemas, e define uma série de guias de como contorná-los. O trabalho de Mancebo et al. (2021) formaliza um processo denominado *Green Software Measurement Process* (GSMP) e uma ontologia *Green Software Measurement Ontology* (GSMO) para evitar conflitos de terminologia em pesquisa relacionada a consumo de energia de programas de computador. Ele também apresenta uma ferramenta física denominada *Energy Efficiency Tester* (EET) para medição de consumo de energia de componentes de um computador, e uma ferramenta gráfica

denominada ELLIOT para visualização dos dados obtidos pelo EET. O trabalho de Zhang e Fu (2011) desenvolve *macropower*, um *framework* de cálculo do uso de potência e consumo de energia de uma nuvem de computadores atuando em conjunto como sistema distribuído. A pesquisa de Díaz et al. (2024) desenvolve um medidor de energia físico que armazena os dados obtidos de forma sincronizada e unificada num servidor remoto, possibilitando o uso em contextos distribuídos.

O trabalho de Noureddine et al. (2012) desenvolve a ferramenta e *framework* denominada PowerAPI para aferição do consumo de energia de diferentes componentes do sistema em tempo de execução por meio de modelos matemáticos. Esse *framework* é usado por vários trabalhos citados nas seções 2.2 e 2.3, e está sob melhoria contínua pela comunidade de código-fonte aberto, por exemplo pela ferramenta SmartWatts desenvolvida por Fieni, Rouvoy e Seinturier (2020) que integra o *framework* do PowerAPI com a interface *Running Average Power Limit* (RAPL) da Intel.

O trabalho de Huang et al. (2015) avalia o impacto de novas técnicas de gerenciamento de energia nos processadores com arquitetura Haswell da Intel medindo o consumo de quatro aplicações de referência, fazendo uso dos indicadores embutidos no processador pela Intel, pela interface RAPL. Ele analisa a sobrecarga (*overhead*) do uso do RAPL, com um medidor externo de potência, que nas condições do teste realizado no estudo, conclui que a sobrecarga de monitoramento é muito pequena se a ferramenta utilizada for bem configurada. A pesquisa de Khan et al. (2018) realiza uma série de experimentos para investigar a precisão da interface RAPL, e concluem que as leituras da interface são altamente correlacionadas ao consumo de energia medido na tomada do computador, com precisão significativa, e com sobrecarga insignificante nas medições. Esses resultados sugerem que RAPL é uma ferramenta muito útil para monitorar o consumo de energia de computadores sem precisar de medidores físicos complexos.

O trabalho de Raffin e Trystram (2025) realiza uma análise comparativa das tecnologias de aferição do consumo de energia que fazem uso da tecnologia RAPL, e desenvolve uma ferramenta que faz uso de todas as qualidades identificadas no método de aferição das tecnologias analisadas. A ferramenta de monitoramento de consumo de energia distribuído ALUMET, utilizada neste trabalho, é desenvolvida a partir da pesquisa de Raffin e Trystram (2025), tendo como base a ferramenta ideal que eles desenvolveram nesse artigo.

Os trabalhos citados nesta seção foram os que mais enriqueceram a parte técnica do estudo empírico realizado — as diversas técnicas utilizadas para medir o consumo de energia de diferentes aplicações nos ajudaram a elaborar a estrutura final que utilizamos. Mais detalhes desse processo são expostos na Seção 4.4.

2.2 PESQUISAS SOBRE TÉCNICAS DE PROGRAMAÇÃO E ESCOLHAS DE TECNOLOGIA NO CONSUMO DE ENERGIA

O trabalho de Sahin, Pollock e Clause (2014) avalia o impacto de refatorações de código na eficiência energética de aplicações. Refatorações e aplicações são escolhidas, e a diferença de desempenho entre a versão de cada aplicação antes e após aplicar uma refatoração específica é relatada. Os trabalhos de Ournani et al. (2021b) e Ournani et al. (2021c) também avaliam o impacto de refatorações de código na eficiência energética de aplicações, mas usam outra metodologia: aplicações são escolhidas e o impacto no desempenho energético de refatorações ao longo do desenvolvimento da aplicação são relatadas.

A pesquisa de Cruz e Abreu (2017) avalia se práticas de desenvolvimento recomendadas para melhorar o desempenho de aplicações para dispositivos móveis possuem um impacto no consumo de energia do dispositivo móvel. É relatado que o tratamento de alguns problemas de coesão (*code smells*) melhora o desempenho energético das aplicações em dispositivos móveis. Os trabalhos de Sahin et al. (2016) e Bunse (2018) avaliam o impacto de técnicas de ofuscação (*obfuscation*) de código-fonte no consumo de energia de aplicações de dispositivos móveis.

O trabalho de Sahin et al. (2012) avalia o impacto no consumo de energia ao aplicar diferentes padrões de design (*design patterns*) no código fonte de diferentes aplicações. Outros trabalhos similares são listados no levantamento de Poy et al. (2025).

As publicações citadas nesta seção foram instrumentais na elaboração da fração não técnica do trabalho: como guiar a narrativa em relação à importância da análise do consumo de energia, qual o papel do código fonte nesse processo, e quais investigações seriam interessantes de realizar a partir do resultado desses trabalhos. Além disso, o processo de isolar componentes específicos do código fonte para análise utilizado nesses trabalhos foi de grande valor para a elaboração do ambiente de teste utilizado — mais detalhes desse isolamennto são expostos na Seção 4.1.

2.3 PESQUISAS SOBRE O CONSUMO DE ENERGIA DE DIFERENTES OPÇÕES DE PROGRAMAS

O trabalho de Pérez et al. (2024) avalia o consumo de energia dos motores de jogos virtuais populares Unity e Unreal Engine em diferentes cenários. A pesquisa de Damasceno et al. (2024) realiza uma análise comparativa da eficiência energética de compiladores em diversos domínios da computação de alto desempenho. O trabalho de Manotas et al. (2013) avalia como a escolha de diferentes servidores web afetam o consumo de energia de uma aplicação web. Servidores web diferentes são mais ou menos eficientes em termos de energia dependendo de quais funcionalidades do servidor web são utilizadas pela aplicação. O trabalho de Ournani et al. (2021a) compara o consumo de energia de diferentes

máquinas virtuais da linguagem Java (*Java Virtual Machines*). A pesquisa de Durán et al. (2025) avalia o consumo de energia de diferentes motores e ambientes de execução de modelos de linguagem pequenos.

As publicações citadas nesta seção mostraram que análises comparativas de importantes componentes de sistemas de computação diversos são objetos de pesquisa interessantes e que há demanda e relevância para tais análises. A grande variedade de componentes comparados e as formas utilizadas para medir o consumo de energia de cada um foram motivadores importantes ao estudo empírico realizado, por serem trabalhos com metodologia similar aplicada a vários tipos de sistemas, e por isso foram ótimos exemplos.

2.4 NOSSA CONTRIBUIÇÃO

Este trabalho se enquadra na categoria da Seção 2.3. Ele faz uso de uma ferramenta derivada dos trabalhos da Seção 2.1 para analisar de forma comparativa diferentes implementações de um algoritmo que é amplamente utilizado em sistemas distribuídos para garantir a consistência em nós replicados — o algoritmo Raft. Esse algoritmo foi escolhido devido a sua difusão e a disponibilidade de diferentes implementações de código-fonte aberto. O Raft é utilizado para manter consistentes os dados entre nós de sistemas distribuídos de uso generalizado, como nos bancos de dados *MongoDB*[1], *CockroachDB*[2] e *etcd*[3] (de uso muito popular por ser um componente vital dos sistemas *Kubernetes*[4] e *Docker Swarm*[5]) e pelo software de fila de mensagens *RabbitMQ*[6]. As informações provenientes deste trabalho visam educar desenvolvedores sobre os custos energéticos decorrentes da escolha da implementação desse algoritmo em seus programas. Assim, o trabalho os permite realizar uma escolha mais sustentável ao pesar esse consumo de energia com as outras necessidades de seus programas.

Há também pesquisas em outra direção relacionadas ao algoritmo Raft, como a de Pâris e Long (2015), que analisa como otimizar o consumo de energia do ponto de vista do próprio algoritmo, isto é, reorganiza os passos do algoritmo para resolver o problema do consenso de forma segura e resistente a falhas com menos participantes e consequentemente menos consumo de energia quando comparado ao algoritmo original. Isso é diferente do escopo deste trabalho, que apenas investiga implementações já existentes, sem adentrar no funcionamento do algoritmo.

[1] https://www.mongodb.com/
[2] https://www.cockroachlabs.com/
[3] https://etcd.io/
[4] https://kubernetes.io/pt-br/
[5] https://docs.docker.com/engine/swarm/
[6] https://www.rabbitmq.com/

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta o algoritmo Raft. A Seção 3.1 descreve o problema de consistência gerado pela replicação em sistemas distribuídos e as técnicas utilizadas para enfrentá-lo. A Seção 3.2 apresenta o algoritmo Raft em detalhe: seu funcionamento, e como ele é utilizado para resolver o problema do consenso.

3.1 O PROBLEMA DO CONSENSO DISTRIBUÍDO

Sistemas distribuídos são replicados visando alcançar:

a) Desempenho: replicar o sistema permite que o trabalho total seja dividido com o objetivo de melhorar o desempenho. Além disso, a replicação do sistema pode melhorar o desempenho do sistema por ter partes localizadas mais próximos do cliente.

b) Dependabilidade: é a qualidade que um sistema ou componente tem de proporcionar disponibilidade, confiabilidade, segurança, manutenibilidade, confidencialidade e integridade. A replicação permite que o sistema trabalhe como um grupo concordante capaz de sobreviver a falhas em seu funcionamento (ONGARO; OUSTERHOUT, 2014).

A replicação introduz um problema de consistência: sempre que uma réplica é atualizada, essa réplica se torna diferente das outras. Para manter as réplicas consistentes, precisamos propagar as atualizações de forma que as inconsistências temporárias não sejam percebidas. Consequentemente, o sistema se torna sujeito a falhas que ameaçam a sua consistência, por isso, a detecção e tratamento de falhas é um assunto importante para o desenvolvimento de sistemas distribuídos. De acordo com van Steen e Tanenbaum (2023), a tolerância a falhas é uma das técnicas de tratamento de falhas pelas quais um sistema pode mascarar a ocorrência de falhas e se recuperar delas. Em outras palavras, um sistema é tolerante a falhas se puder continuar operando na presença de falhas. Existem diferentes tipos de falhas:

a) Falhas de Parada (*Crash*): Uma falha de *crash* ocorre quando um servidor para de funcionar repentinamente, embora estivesse operando corretamente até então. Após o travamento, ele deixa de responder completamente. Um exemplo típico é um sistema operacional que congela e precisa ser reiniciado.

b) Falhas de Omissão: ocorrem quando o servidor não responde a uma requisição. Isso pode acontecer por dois motivos principais:

- Falha de Recepção-Omissão: o servidor nunca recebe a requisição, por exemplo, porque não havia uma *thread* escutando. Nesse caso, o estado do servidor não é alterado, pois não está ciente das mensagens enviadas a ele.
- Falha de Envio-Omissão: o servidor processa a requisição, mas falha ao enviar a resposta (por exemplo, devido a um *buffer* cheio). Nesse caso, o servidor precisa estar preparado para que o cliente reenvie sua solicitação anterior.

Outras falhas por omissão também podem ser causadas por erros de *software*, como *loops* infinitos ou gerenciamento inadequado de memória, fazendo com que o servidor trave.

c) Falhas de Atraso (*Timing*): Ocorre quando a resposta do servidor demora mais que o intervalo de tempo esperado. Isso pode acontecer tanto quando a resposta chega cedo demais (como em *streaming* de vídeo sem espaço no buffer) quanto tarde demais, o que é mais comum, e é caracterizado como uma falha de desempenho.

d) Falhas de Resposta: Ocorre quando o servidor envia uma resposta incorreta. Dois tipos de falha de resposta podem ocorrer:

- Falha de valor: Ocorre quando um servidor retorna ao pedido do cliente um resultado errado.
- Falha de Transição de Estado: Ocorre quando o servidor reage de forma inesperada a uma requisição. Por exemplo, ao receber uma mensagem desconhecida, a falha de transição de estado pode ocorrer se nenhuma medida for tomada para lidar com essa mensagem.

e) Falhas Arbitrárias ou Bizantinas: São as falhas mais graves, pois o servidor pode gerar respostas incorretas que parecem válidas e não podem ser detectadas corretamente.

Logo, replicação do sistema é feita para melhorar o desempenho do sistema e aumentar sua dependabilidade. Todavia, a replicação torna o sistema suscetível as falhas mencionadas. Sendo assim, torna-se importante garantir consistência entre as replicas do sistema (VAN STEEN; TANENBAUM, 2023). A consistência, nesses casos, é formulada em termos do que os processos podem esperar ao ler e atualizar os dados, sabendo que outros processos também os estão acessando. Uma classe importante dos chamados modelos de consistência considera que múltiplos processos acessam simultaneamente dados compartilhados. Entretanto, modelos de consistência para dados compartilhados são difíceis de implementar de forma eficiente em sistemas distribuídos de grande escala. Por isso, muitas vezes modelos mais simples são utilizados, pois são mais fáceis de implementar (VAN STEEN; TANENBAUM, 2023). Uma categoria específica é a dos modelos de consistência centrados no cliente, que tratam da consistência do ponto de vista de um único

cliente (possivelmente móvel). Outro modelo de consistência é o centrado em dados, que é essencialmente um contrato entre os processos e o repositório de dados (*data store*). Usando o termo repositório de dados de forma ampla, o armazenamento de dados pode estar fisicamente distribuído por várias máquinas. Presume-se que cada processo que pode acessar dados do armazenamento tenha uma cópia local (ou próxima) disponível de todo o armazenamento. A consistência, porém, é apenas metade da questão. Também é preciso considerar como ela é implementada, para isso, precisamos de algoritmos de consenso.

Há duas formas comuns de atacar o problema do consenso em sistemas distribuídos: de forma centralizada ou de forma distribuída. Em sistemas distribuídos com consenso centralizado, um nó central tem responsabilidade e poder maiores que os demais nós. Em sistemas de consenso centralizado, quando um nó local coleta dados ou recebe requisição, o mesmo não toma decisões, então envia mensagens para o nó central. Em seguida, o nó central toma a decisão e envia mensagens para os demais nós periféricos. Neste caso, o nó central é o único participante que pode sincronizar o estado dos outros nós. Sem ele o sistema não funciona. Sendo assim, um desempenho confiável é totalmente dependente do nó central. Sistemas como o módulo de controle de um carro à combustão e sistemas de monitoramento de câmeras são exemplos de arquiteturas de sistemas distribuídos com consenso centralizado.

Com o objetivo de garantir alta disponibilidade, existe a necessidade de sistemas distribuídos serem resistentes a falhas em quaisquer um de seus nós, e isso torna a implementação de soluções baseadas em consenso centralizado inadequadas para esses casos de uso. A partir disso, cria-se a necessidade de um esquema onde a necessidade do nó central é reduzida, melhorando a robustez de tomada de decisão crítica — as soluções de consenso que partem dessa premissa são denominadas de consenso distribuído. O sistema distribuído com consenso distribuído tem como objetivo garantir que o sistema geral possa alcançar os acordos sobre os estados unificados, mesmo que a rede sofra de uma quantidade de falhas (YU; ZHANG, 2022). Em consenso distribuído, cada participante deve ser capaz de transmitir e receber comandos para alterar o estado das réplicas, desde que siga protocolos específicos decididos pelo algoritmo de consenso distribuído.

O problema do consenso normalmente é expresso no contexto de máquinas de estado replicadas (SCHNEIDER, 1990). Nessa abordagem, máquinas de estado em uma coleção de servidores executam a mesma sequência de instruções para chegar a um mesmo estado, idêntico entre todos os servidores da coleção, e podem continuar operando mesmo quando alguns dos servidores estão indisponíveis. Esse conceito é usado para resolver vários problemas de tolerância a falhas em sistemas distribuídos. Máquinas de estados replicadas são comumente implementadas nesses algoritmos utilizando um registro de ações replicado. Cada servidor armazena um registro com comandos a serem executados em ordem pela máquina de estados. Os registros em cada servidor contêm os mesmos comandos na mesma ordem, para que cada máquina de estados possa executá-los na mesma ordem.

Como as máquinas de estado são determinísticas, cada servidor sempre computa o mesmo estado.

O trabalho de manter o registro distribuído consistente é do algoritmo de consenso. Um servidor, ao receber comandos de clientes, armazena-os no seu registro e, usando o algoritmo de consenso, comunica-se com os outros servidores do conjunto ao qual faz parte para garantir que todos os registros em todos os servidores eventualmente possuam os mesmos comandos na mesma ordem, mesmo que alguns servidores falhem. Quando algum comando é replicado com sucesso, a máquina de estado de cada servidor os processa na ordem do registro, e as saídas computadas são enviadas aos clientes. Dessa forma, o conjunto de servidores forma uma máquina de estados unificada e resistente a falhas (ONGARO; OUSTERHOUT, 2014).

Algoritmos de consenso como Raft e Paxos são projetados para gerenciar a duplicação confiável de estados e evitar a quebra do sistema causada especialmente por falhas não bizantinas, como falhas de parada e partições de rede (YU; ZHANG, 2022). Dentre os algoritmos que implementam tolerância a falhas não bizantinas, o algoritmo Raft foi escolhido para este trabalho devido à sua compreensibilidade maior comparado a outros algoritmos similares (HOWARD; MORTIER, 2020), tendo sido projetado para tratar a dificuldade de entendimento do algoritmo Paxos (ONGARO; OUSTERHOUT, 2014). Além disso, ele também é considerado mais fácil de implementar quando comparado ao Paxos (ONGARO; OUSTERHOUT, 2014), e possui suporte a bibliotecas modernas e adequação a sistemas distribuídos em ambientes controlados, como *etcd* e *Clickhouse*[1], respectivamente. O algoritmo Raft apresenta ampla adoção em sistemas distribuídos modernos. Além dos mencionados no Capítulo 2, temos o *Neo4j*, por exemplo, o utiliza para garantir consistência e segurança; no *TiDB*, Raft é integrado à *Transactional Key-Value API (TiKV)*; enquanto plataformas como *NATS Messaging*, *Camunda* e *Redpanda* o empregam como mecanismo de replicação de dados (WIKIPEDIA CONTRIBUTORS, 2025b).

3.2 O ALGORITMO RAFT

O algoritmo Raft executa em um conjunto de servidores, também chamados de nós. A função dos servidores do Raft é gerenciar um registro distribuído na forma que é comumente usada no conceito de máquinas de estado distribuídas, descritas na Seção 3.1. Raft implementa consenso com a estratégia de primeiro eleger um servidor líder, e responsabiliza esse servidor com o controle exclusivo do registro distribuído. O líder aceita entradas de registro (comandos da máquina de estado) enviados pelos clientes, replica essas entradas de registro aos outros servidores participantes, e avisa-os quando é seguro aplicar esses comandos em suas respectivas máquinas de estado. Ter um único líder simplifica

[1] https://clickhouse.com/clickhouse/keeper

a gerência do registro distribuído, pois ele pode decidir onde colocar entradas novas no registro sem consultar os outros servidores, e os dados fluem de maneira simples do líder para os outros servidores. Um líder pode falhar ou perder a comunicação com os outros servidores, e nesse caso um novo líder é eleito.

O Raft, como outros algoritmos de consenso com o mesmo nível de resistência a falhas não bizantinas, é projetado para manter-se disponível contanto que a maioria dos servidores participantes estejam operacionais e consigam se comunicar entre si e com os clientes. Nesse sentido, três é o número mínimo de servidores necessário para o algoritmo tolerar a falha de um nó, e cinco servidores são necessários para tolerar a falha de até dois nós (ONGARO; OUSTERHOUT, 2014).

Raft, é similar a outros algoritmos de consenso, porém com várias funcionalidades inovadoras, segundo Ongaro e Ousterhout (2014):

- **Liderança forte:** o Raft usa uma forma mais rígida de liderança comparado aos demais algoritmos de consenso. O fluxo de registros passa apenas pelo líder, que repassa para os demais servidores, simplificando o gerenciamento da replicação de registros.

- **Eleição de líder:** o Raft utiliza temporizadores aleatórios para eleger o líder, enquanto resolve conflitos de forma simples e rápida. Isso é aprofundado na Seção 3.2.1.

- **Mudança de membros:** o mecanismo do Raft para alterar o conjunto de servidores participantes utiliza uma abordagem de *consenso conjunto*, na qual as duas configurações diferentes de participantes se sobrepõem durante a transição, o que permite a operação contínua da máquina de estados nesse período.

Com essa abordagem de líder, o algoritmo Raft decompõe o problema de consenso em três problemas menores:

- **Eleição de líder:** como mencionado anteriormente, deve ocorrer sempre quando houver a ausência de ou um líder existente falhar.

- **Replicação de registros:** deve ocorrer quando um cliente envia um comando ao sistema e o líder o recebe; o líder deve enviar uma entrada no registro correspondente à ação do cliente aos outros servidores e garantir que o registro dos outros servidores concorde com seu próprio registro.

- **Segurança:** garante que se qualquer servidor aplicou em algum momento uma entrada do registro na sua máquina de estados, então nenhum outro servidor pode aplicar um comando diferente para uma entrada de registro com o mesmo índice.

As subseções seguintes entram em detalhes sobre partes do algoritmo cuja compreensão é necessária para que a comparação de diferentes implementações possa ser realizada. A Subseção 3.2.1 entra em detalhes sobre como a eleição de líder é realizada pelo algoritmo, a subseção 3.2.2 descreve como a recepção de comandos dos clientes e as operações necessárias para replicação dos registros correspondentes são realizadas. A Subseção 3.2.3 descreve componentes opcionais do Raft descritos no artigo original que estão presentes em algumas das implementações avaliadas e têm como intuito aumentar o desempenho do algoritmo.

3.2.1 Eleição de líder

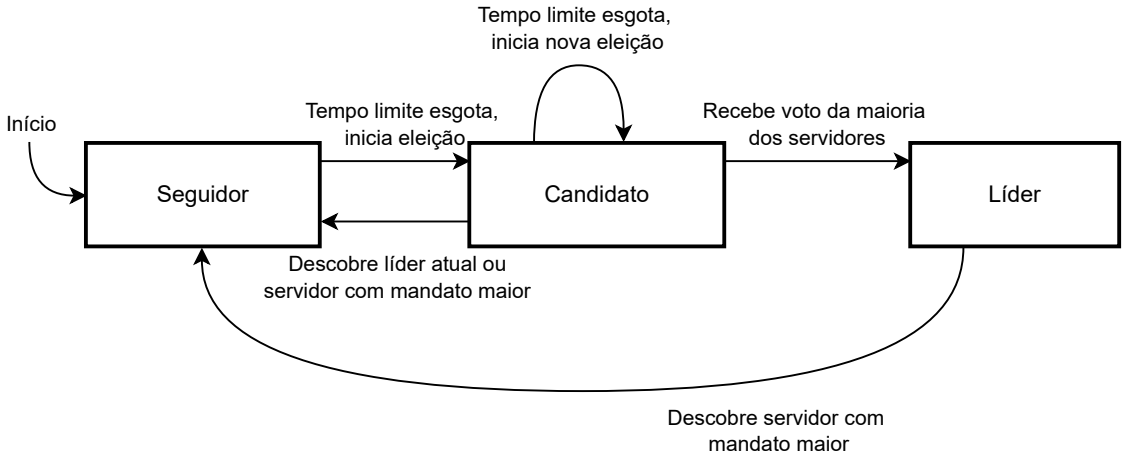
A qualquer momento durante a execução do Raft cada servidor participante está em um de três estados: líder, candidato e seguidor. A Figura 1 apresenta os possíveis estados dos servidores no algoritmo e as possíveis transições entre eles. Durante uma execução típica do Raft, há exatamente um líder e todos os outros servidores são seguidores. Os seguidores são passivos: eles não fazem requisições próprias, apenas respondem a requisições de líder e candidatos. O estado de candidato é utilizado para eleger um novo líder. O líder trata todas as requisições do cliente e se comunica com seus seguidores por meio de pelo menos dois tipos de mensagens, *AppendEntries* e *RequestVote*.

O líder envia mensagens do tipo *AppendEntries* com conteúdo vazio (sem comandos) para todos os seguidores de forma periódica para manter sua autoridade sobre os demais servidores. O não recebimento dessas mensagens no lado dos seguidores por um determinado período de tempo randomizado é utilizado como mecanismo para iniciar uma nova eleição de líder. Esse período de tempo é denominado "tempo limite de eleição" (*election timeout*) e é definido como parâmetro do algoritmo Raft em função do tempo de processamento da máquina de estado e do atraso de comunicação entre os servidores participantes.

Na inicialização de um conjunto de servidores, todos os participantes começam no estado de seguidor, e continuam neste estado enquanto receberem mensagens de um líder. Uma eleição de líder acontece quando um servidor tem o tempo limite de eleição esgotado, ou seja, quando um ou mais seguidores não recebem *AppendEntries* depois de determinado período. Cada servidor tem um contador interno de tempo limite de eleição. Este contador tem um valor pseudo aleatório que fica contando o tempo desde a última mensagem recebida pelo líder. Quando esse tempo acaba, é assumido pelo servidor que não há nenhum líder ativo que seja capaz de liderá-lo e ele começa uma eleição de um novo líder, enviando uma mensagem do tipo *RequestVote* para todos os participantes.

As quantidades de eleições são contadas através de um valor inteiro de "mandato", este valor também é utilizado para validar qual líder é mais atual e a cada eleição de líder o valor do mandato é incrementado em um. Quando um seguidor inicia uma eleição, ele incrementa seu contador interno de mandato e altera seu estado para candidato. Em

Figura 1 – Estados dos servidores no algoritmo Raft.



Fonte: Ongaro e Ousterhout (2014)

seguida, ele vota em si mesmo e envia mensagens do tipo *RequestVote* para os demais participantes. Segundo Ongaro e Ousterhout (2014), um candidato continua neste estado até um desses três resultados acontecer:

a) O servidor ganha a eleição e se torna líder.

b) Outro servidor se estabelece como líder.

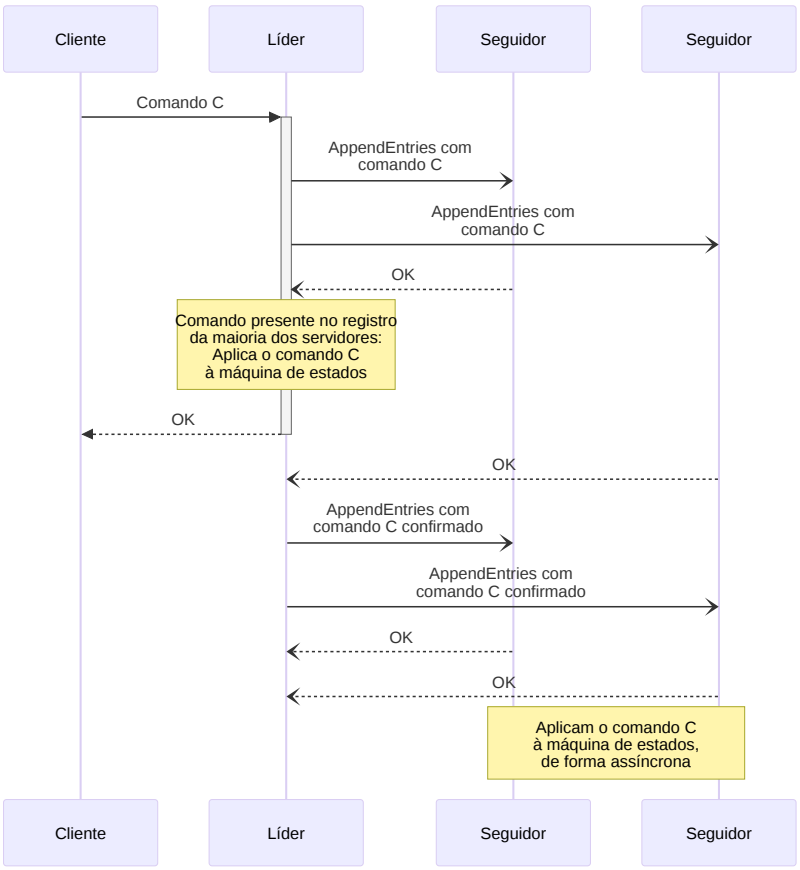
c) Um período de tempo passa sem nenhum servidor vitorioso da eleição.

Os itens b) e c) são casos especiais. Como o objetivo deste trabalho é estudar o funcionamento normal do Raft esses casos excepcionais estão fora do escopo. Sendo assim, para este trabalho, apenas o resultado a) em que um servidor ganha a eleição é relevante. Um candidato vence uma eleição se receber votos da maioria dos servidores do conjunto completo para o mesmo mandato. Cada servidor votará em, no máximo, um candidato por mandato, com base na ordem de chegada das mensagens *RequestVote*. Assim que um candidato vence uma eleição, ele altera seu estado para líder. Em seguida, ele começa a enviar mensagens (*AppendEntries* sem comandos) informando que é o novo líder com o número do mandato incrementado para os demais servidores de forma a estabelecer sua autoridade e evitar novas eleições. Os demais servidores se tornam seguidores deste líder.

3.2.2 Replicação de registros

Após a eleição, o líder começa a processar os registros vindos dos clientes. Cada um dos registros dos clientes contém um comando a ser executado pelas máquinas de estado replicadas dos nós. Ao receber um comando do cliente, o líder adiciona o comando do cliente em seu registro e, então, começa a enviar mensagens do tipo *AppendEntries* para seus seguidores o replicarem. O líder manda as mensagens para seus seguidores até a maioria dos nós tê-las replicado com segurança. Mesmo que alguma falha ocorra em

Figura 2 – Sequência possível de execução de um comando no algoritmo Raft num conjunto de três servidores.



um ou mais seguidores, o líder continua enviando as mensagens até esses eventualmente armazenarem todos os registros enviados. Quando os registros são replicados com sucesso na maioria dos servidores, eles são chamados de confirmados ("*commited*"). Então, o líder aplica as entradas confirmadas à sua máquina de estado e retorna ao cliente o resultado da execução. A partir daí, o líder inclui o índice do maior registro que foi confirmado nas mensagens de *AppendEntries* que envia para seus seguidores que, ao recebê-las, podem aplicar esses registros confirmados às suas máquina de estado. Dessa forma, se um servidor estiver inconsistente ou desatualizado, ele eventualmente replicará os registros e os aplicará em sua máquina de estado local.

A Figura 2 mostra uma sequência possível de execução de um comando enviado por um cliente num sistema distribuído com consenso provido pelo algoritmo Raft com três participantes. O comando é enviado ao líder, que o salva em seu registro e o encaminha aos seguidores enviando mensagens do tipo *AppendEntries*. Assim que um dos seguidores retorna com sucesso que o comando foi incluído em seu registro de entradas, o líder conclui que a maioria dos nós participantes já possui a entrada com o comando (2 de 3 servidores já é maioria) e por isso já pode confirmá-la (marcá-la como *commited*). Ele a marca como confirmada no seu registro, aplica o comando na sua máquina de estado, retorna a

resposta para o cliente que enviou o comando, e envia um novo *AppendEntries* para os seguidores com os valores atualizados de forma que os seguidores entendam que o comando foi confirmado e podem aplicá-lo às suas respectivas máquinas de estado.

O líder resolve inconsistências encontrando a última entrada de registro que ele e o seguidor concordam e exclui todas as entradas após esta. O líder mantém o índice do maior registro para cada seguidor, chamado *nextIndex*. Quando um *AppendEntries* não é respondido devido a alguma inconsistência, o líder decrementa o *nextIndex* e envia novamente a mensagem até eventualmente chegar ao ponto onde os registros coincidem. Quando esse *AppendEntries* for bem-sucedido, quaisquer registros conflitantes do servidor são apagados e a entrada do líder é anexada. Dessa forma, o seguidor se torna consistente com o líder.

3.2.3 Compactação e *snapshots* de registros

Ao longo de sua operação, a lista de registros do algoritmo Raft cresce com as requisições dos clientes. É indesejável que essa lista de registros cresça indefinidamente, pois quanto maior ela se torna, mais espaço seria utilizado para armazená-la e mais tempo levaria para replicá-la em outros nós em casos de quedas ou de mudanças dos participantes. Por conta desses problemas, Ongaro e Ousterhout (2014) sugere mais um tipo de mensagem no algoritmo Raft, que não é necessário para sua corretude, mas auxilia na redução do espaço utilizado a longo prazo e na quantidade de informações que precisa ser trocada entre os participantes afetados por atrasos de rede ou outros problemas. A abordagem sugerida para resolver esse problema é denominada *Snapshotting* (criação de registros de estado): ao invés de manter todos os registros replicados, o estado atual da máquina de estados replicada é gravado em um registro de estado (*snapshot*) e armazenado. Isso possibilita descartar todos os registros de replicação até o momento daquele *snapshot*, e ele também pode ser utilizado para substituir todos os registros anteriores ao momento do *snapshot* na comunicação com outros nós. Cada servidor cria um *snapshot* de forma independente, cobrindo apenas as entradas confirmadas. Os *snapshots* incluem:

- Estado atual da máquina de estado;
- Índice da última entrada;
- Índice do mandato.

Por diversas razões, um participante seguidor pode ficar muito defasado em comparação com o líder, ou um novo servidor ingressar no conjunto de participantes. Quando isso acontece, o líder envia um *snapshot* para esse seguidor, para que este servidor possa sincronizar mais rapidamente com o líder e os demais pares sem precisar replicar todos os registros desde o início do algoritmo. Isso é feito pelo novo tipo de mensagem sugerido, denominado *InstallSnapshot*, que é para o seguidor. O recipiente, então, pode descartar

seus registros, substituindo todos pelo *snapshot* enviado pelo líder, e resumir quaisquer registros já confirmados posteriores ao *snapshot* pelo método mais comum, recebendo mensagens do tipo *AppendEntries* do líder. Se por alguma falha de rede ou retransmissão algum seguidor recebe um *snapshot* que descreve algum prefixo de seus registros, então as entradas até o *snapshot* recebido podem ser descartadas, mas as seguintes são válidas e devem ser mantidas.

4 CENÁRIOS PARA AVALIAÇÃO DO ALGORITMO RAFT

Este capítulo descreve os detalhes do ambiente para a análise de consumo de energia, incluindo as variáveis consideradas e as implementações do algoritmo Raft escolhidas, além da metodologia para a obtenção das métricas de energia.

4.1 VARIÁVEIS EXPERIMENTAIS

Para os fins do estudo empírico englobado por esse trabalho foram escolhidas uma variável dependente, a quantidade de energia consumida por um sistema distribuído cujo consenso é obtido pelo algoritmo Raft, e uma variável independente, a implementação do algoritmo Raft utilizada.

Existem inúmeras variáveis extras que afetam a medição do consumo de energia das implementações do algoritmo que queremos comparar. Num cenário de monitoramento do dia a dia de uma aplicação distribuída, não seria necessário tomar ações para minimizar o impacto dessas variáveis extras na medição final: a medição já apresenta o valor da aplicação sob uso real. No cenário deste trabalho, porém, queremos comparar apenas a diferença no consumo entre as implementações do algoritmo Raft, que é um pequeno componente que compõe um sistema distribuído, e isso quer dizer que devemos minimizar todos os outros aspectos de sistemas distribuídos que podem impactar no consumo de energia total de modo a conseguir obter medições que possam ser usadas para comparar apenas os algoritmos.

Para minimizar as variações nos resultados dos experimentos, várias decisões sobre o teste foram tomadas. Quando possível, cada implementação foi testada com os mesmos parâmetros, descritos na Seção 4.3.3, de modo a minimizar o impacto causado por diferentes parâmetros. Todos os testes foram realizados com todos os processos participantes na mesma máquina física, de modo a minimizar diferenças no consumo de energia entre componentes físicos de computadores diferentes, verificada por Ournani et al. (2020). Não foi provocada ou prevista trocas de líder ou falha de rede, de modo a testar o algoritmo apenas em sua fase mais comum, que é a de execução normal, sem falhas. Frequentes trocas de líderes ou partições de rede tornariam o experimento muito mais difícil de reproduzir de forma consistente, e por isso iriam requerer experimentos com duração muito mais longa para obter resultados confiáveis. Vale observar que as implementações escolhidas neste trabalho podem variar em custo energético de acordo com os parâmetros definidos, ademais, uma implementação pode ser mais eficiente com outra definição de parâmetros, ou sua eficiência energética pode variar em função das falhas que podem ocorrer durante a execução do algoritmo. Apesar disso, a padronização dos parâmetros ainda mostra-se válida para analisar o desempenho das implementações em uma situação

específica, e também como indicativo do desempenho geral das implementações, visto que os parâmetros devem ser ajustados apenas para corresponder com a situação da aplicação sendo testada e da distância entre os nós do conjunto.

4.2 IMPLEMENTAÇÕES

Para avaliar apenas a diferença no consumo de energia causada pela implementação usada, é necessário um sistema distribuído no qual é possível alterar apenas a implementação utilizada, com o mínimo de alterações em outros componentes do sistema para adequar as mudanças de implementação do Raft. Dessa forma, o consumo de energia dos componentes não relacionados ao algoritmo Raft é equivalente em todos os testes, e não impacta o resultado final. Essa abordagem não funciona quando deseja-se avaliar o consumo de energia de implementações Raft desenvolvidas em linguagens diferentes, pois não se pode afirmar que o consumo de energia de um mesmo sistema desenvolvido e compilado em outra linguagem de programação será equivalente.

Foram escolhidas duas linguagens para as quais o desempenho energético dos algoritmos Raft existentes serão avaliados, Go e Rust, e os algoritmos existentes populares dessas linguagens foram obtidos da página web do Raft (ONGARO, 2025), que mostra implementações de código fonte aberto do algoritmo Raft disponíveis na plataforma GitHub[1]. Dentre essa lista, foram escolhidas implementações "puras", isto é, projetos de código aberto cujo objetivo é prover o algoritmo Raft como biblioteca. Além das implementações escolhidas, uma versão da máquina de estados, descrita na Seção 4.3.1, foi avaliada para cada linguagem utilizada sem o uso de Raft, para estimar o consumo de energia de toda a infraestrutura dos testes sem o componente do algoritmo. Todo o resto manteve-se igual: o mesmo número de processos participantes foi utilizado, mas em vez dos três compartilharem a máquina de estados por meio do algoritmo Raft, apenas um deles computava os valores, e os outros serviram apenas de "proxy", isto é, apenas encaminharam as requisições dos clientes para o servidor que de fato implementava a máquina de estados e retornavam a resposta do servidor para esses clientes. O Quadro 1 indica as implementações populares do algoritmo avaliadas.

4.3 AMBIENTE DE COMPARAÇÃO

Para evitar inconsistências e viés na avaliação de cada implementação, o desempenho energético delas foi medido sob um teste de carga média, gerada de forma externa pela aplicação Grafana k6[2]. Essa ferramenta foi escolhida por ser considerada mais simples pelos autores deste trabalho para o caso de uso de um teste de carga médio, ao invés

[1] https://github.com/
[2] https://k6.io/

Quadro 1 – Implementações do algoritmo Raft consideradas

Implementação	Versão	Linguagem	Seguidores
etcd-io/raft[1]	3.6.0	Go	903
hashicorp/raft	1.7.3	Go	8697
lni/dragonboat	4.0.0	Go	5203
async-raft/async-raft	0.6.1	Rust	1073
databendlabs/openraft	0.9.21	Rust	1615

[1] Sucessor do goraft/raft, que está arquivado com 2430 seguidores

Quadro 2 – Versões das bibliotecas utilizadas

Linguagem	Versão	Biblioteca HTTP	Versão
Go	1.24.3	net/http	1.24.3
Rust	1.87.0	Axum	0.8.4

de aplicações similares como Apache JMeter[3] ou Locust[4]. Um único *script* curto na linguagem JavaScript descreve o cronograma inteiro do teste, permite a reprodução do experimento de forma fácil, e gera carga de forma idêntica independente da linguagem da implementação do algoritmo Raft sendo testada. O Código 1 mostra o *script* utilizado para esse teste. O *script* define o código que chama a ação de incrementar no sistema distribuído sendo avaliado, define o número alvo de requisições por segundo ao decorrer de todo o fluxo de teste, e define o número máximo de conexões em paralelo que podem ser utilizadas para atingir essa taxa desejada. Os valores utilizados são detalhados no Capítulo 5.

[3] https://jmeter.apache.org/
[4] https://locust.io/

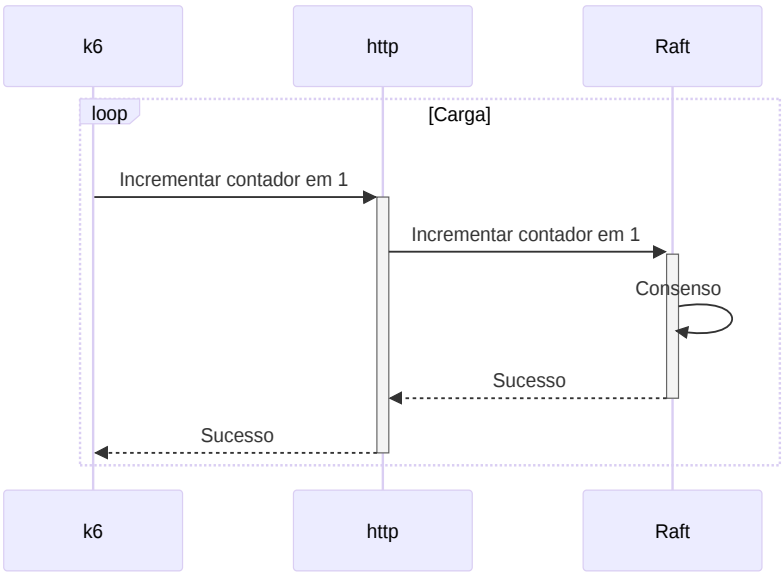
Código 1 – Script k6 do teste de carga médio.

```
import { check, sleep } from 'k6';
import exec from 'k6/execution';
import http from 'k6/http';

export const options = {
  discardResponseBodies: true,
  scenarios: {
    contacts: {
      executor: 'ramping-arrival-rate',
      startRate: 0,
      timeUnit: '1s',
      preAllocatedVUs: 1024,
      maxVUs: 1024,
      stages: [
        { target: 0, duration: '30s' },
        { target: 4096, duration: '30s' },
        { target: 4096, duration: '150s' },
        { target: 0, duration: '30s' },
      ]
    },
  },
};

export default function () {
  let port = 8091 + exec.scenario.iterationInInstance % 3
  let res = http.post(
    `http://${__ENV.REMOTE_HOST}:${port}/increment`,
    null,
    { throw: false, timeout: '1s' });
  check(res, { "status is 200": (res) => res.status === 200 });
}
```

Foi escolhido o HTTP como protocolo de comunicação do teste de carga. Essa estratégia requer que cada conjunto de servidores possua uma interface HTTP para receber a carga gerada pelo Grafana k6. Para implementações em uma mesma linguagem, a mesma implementação de servidor HTTP foi utilizada, de modo a que esse não seja um fator que gere viés em comparações de implementações de algoritmo Raft em linguagens iguais. Como não é possível usar a mesma interface em diferentes linguagens, a comparação entre os resultados da medição de implementações entre diferentes linguagens estará sujeita ao viés da interface HTTP também, além do viés da diferença da linguagem da implementação do algoritmo em si. O Quadro 2 mostra as versões e bibliotecas utilizadas em cada linguagem de programação utilizada por alguma das implementações avaliadas.

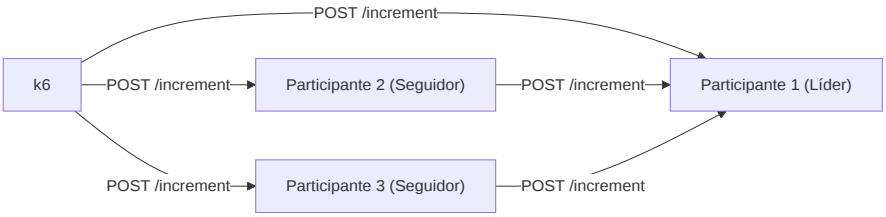
A Figura 3 mostra o fluxo simplificado de uma iteração do teste de estresse. Para

Figura 3 – Diagrama de sequência do teste de estresse.



avaliar o consumo de energia de forma similar entre implementações que não possuem funcionalidade embutida de encaminhamento de requisições realizadas a partir de qualquer nó ao líder, foi implementado um mecanismo simples para realizar esse encaminhamento no ponto de entrada do teste de carga. O k6 foi configurado para enviar as requisições do teste de carga em forma de rodízio para cada servidor do conjunto, e quando um servidor que está no estado de seguidor recebe as solicitações, ele as reenvia para o líder do conjunto. Esse comportamento é descrito pela Figura 4, que mostra o fluxo das requisições oriundas do teste de carga, que no algoritmo Raft realiza o papel de cliente.

Figura 4 – Fluxograma de encaminhamento de requisições do teste de carga.



4.3.1 Máquina de estados

Em todas as linguagens, a mesma máquina de estado foi implementada, com apenas duas ações para minimizar o impacto da sua execução nas medições de consumo de energia. O estado dela é composto por um valor inteiro de 64 bits, e as ações definidas são:

a) Ler o valor do inteiro. Essa ação não modifica o estado, e pode ser realizada pelo algoritmo de forma otimizada, sem comprometimento (commit).

b) Incrementar o valor do inteiro em 1. Essa ação modifica o estado, e é a ação que é utilizada no teste de estresse para avaliação das implementações dos algoritmos.

4.3.2 Idempotência

Algumas implementações avaliadas possuem mecanismos embutidos de idempotência para evitar os problemas de escrita perdida que podem ocorrer ao utilizar o algoritmo Raft com uma máquina de estados não idempotente. Como esses mecanismos não são oferecidos por todas as implementações consideradas, foi escolhido não utilizar nenhum desses mecanismos embutidos, de modo que o consumo de energia mensurado não seja influenciado por uma implementação manual de idempotência na máquina de estados simples desenvolvida pelos autores desse trabalho em apenas algumas das implementações do algoritmo avaliadas.

4.3.3 Parâmetros do algoritmo

Cada uma das implementações avaliadas aceita como parâmetro os valores de tempo limite de eleição e intervalo máximo de *AppendEntries* descritos por Ongaro e Ousterhout (2014), na definição original do algoritmo. Esses valores e o número de nós se manteve em todas as implementações, de modo a minimizar alguma potencial diferença no consumo de energia causada pela divergência desses valores entre as implementações.

Quadro 3 – Parâmetros comuns do ambiente de avaliação e do algoritmo Raft utilizados.

Parâmetro	Valor
Número de nós participantes	3
Intervalo máximo de *AppendEntries*	10ms
Tempo limite de eleição	100ms
Número de registros após *snapshot*	128
Número mínimo de registros para capturar *snapshot*	1024

Além dos parâmetros originais do algoritmo, cada implementação também define outros parâmetros de configuração específicos, como configurações de valores que controlam a frequência dos *snapshots* da máquina de estados. Quando algum desses parâmetros foi identificado em mais de uma implementação, elas foram modificadas para utilizar o mesmo valor para o parâmetro identificado. O Quadro 3 lista os parâmetros comuns entre pelo menos duas implementações que foram usados nos experimentos.

4.4 COLETA DE INFORMAÇÕES DE CONSUMO DE ENERGIA

Há duas estratégias principais de como medir o consumo de energia de sistemas. A forma mais precisa utiliza medidores de potência, ferramentas físicas que se conectam à fonte de potência do dispositivo ou componente a ser avaliado e mostram o consumo real

em qualquer instante de tempo. Apesar de serem muito precisos, é muito difícil utilizar medidores de potência, pois é bem comum que alterações no sistema a ser medido sejam necessárias para o medidor funcionar. No contexto de sistemas distribuídos, é muito mais fácil e prático utilizar a outra forma, com perfiladores de energia. Eles usam um modelo de estimação do custo de potência de cada componente físico do computador em tempo real, baseando-se em quais componentes estão ativos em um determinado momento para calcular o custo de energia estimado. Como não há necessidade de um medidor físico instalado no computador a ser analisado, os perfiladores de energia são normalmente fornecidos como ferramentas de software que podem ser executadas a partir de qualquer dispositivo para obter o consumo de energia do mesmo (CRUZ, 2021).

Dentre os perfiladores existentes, usamos a ferramenta ALUMET, dos autores de Raffin e Trystram (2025), baseada na tecnologia *Running Average Power Limit* (RAPL), da Intel. Essa tecnologia, presente em processadores modernos tanto da Intel quanto da AMD, também faz uso de estimativas como os outros perfiladores, mas é muito mais precisa por integrar em seus cálculos medidas reais de informações internas do processador (RAFFIN; TRYSTRAM, 2025). A ferramenta ALUMET foi escolhida pois ela possibilita a exposição das métricas obtidas pelo RAPL para formatos compatíveis com OpenTelemetry, que é um dos formatos mais populares e interoperáveis de telemetria (JANES; LI; LENARDUZZI, 2023), e por isso encaixam perfeitamente em arquiteturas de observabilidade já utilizadas em sistemas distribuídos. O ALUMET também suporta outros tipos de saída para as informações coletadas, como arquivos *Comma-Separated Values* (CSV), e bancos de dados tipo OpenSearch, InfluxDB, MongoDB, e Prometheus. O ALUMET é executado em cada servidor, e para avaliar o consumo total do sistema distribuído, deve-se agregar os resultados de alguma forma. Como seria muito trabalhoso recolher arquivos CSV de cada servidor e processar de forma manual ou desenvolver um programa especializado apenas para processá-los, optamos por analisar as opções compatíveis com OpenTelemetry e as de banco de dados. Esses tipos de saída do ALUMET permitem o agrupamento das métricas de consumo de energia colhidas de forma simples e rápida com seus sistemas de consulta poderosos, o que também facilita a análise das métricas obtidas.

O uso do formato OpenTelemetry para o experimento foi testado com o coletor de métricas oficial OpenTelemetry[5] em conjunto com um banco de dados não relacional OpenSearch[6]. Os dados nos formatos OpenTelemetry são úteis para monitoramento contínuo de serviços que ficam disponíveis de forma permanente, e o formato é otimizado para esse tipo de monitoramento. Dados de experimentos, que por definição são efêmeros, isso é, executam por um período curto de tempo de forma não permanente, não são adequados para monitoramento e armazenamento pelo formatos OpenTelemetry — quando um experimento é encerrado, a última aferição do consumo de energia do experimento é

[5] https://opentelemetry.io/docs/collector/
[6] https://opensearch.org/

armazenada de forma repetida no banco de dados de monitoramento, o que pode levar a erros de medição quando o objetivo é obter o consumo total de energia de um único experimento ao invés de acompanhar o consumo de uma aplicação 24 horas por dia, 7 dias por semana. Por essas razões, não exportamos os dados nesse formato para o estudo empírico realizado, apesar dela ser a mais adequada para monitoramento de serviços em ambientes produtivos.

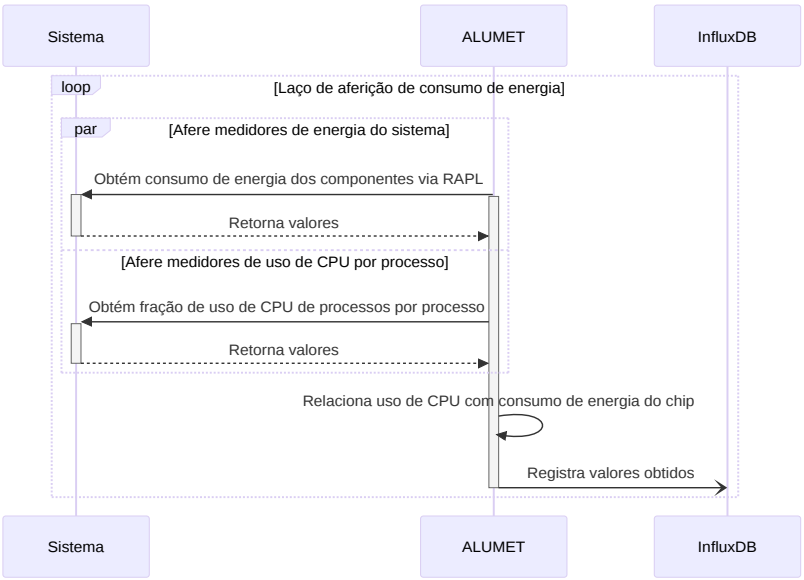
Após a verificação de que essa estratégia de coleta não seria frutífera para as análises estatísticas dos experimentos, avaliamos a possibilidade de armazenar as métricas diretamente no banco OpenSearch. Verificamos que essa estratégia funciona, mas durante os testes foram registrados erros de perdas de leituras da ferramenta por lentidão no armazenamento dos dados coletados, e isso poderia afetar os resultados dos testes. Isso se deu por conta da implementação da integração do ALUMET com o OpenSearch, que é vulnerável a esse tipo de perda de informação em alguns cenários de dessincronização. Para evitar esse problema, decidimos experimentar outro banco de dados, o InfluxDB[7], cuja integração com o ALUMET é mais antiga e aparentou ser mais robusta.

O InfluxDB é um banco de dados de série temporal e foi escolhido para armazenar os dados de consumo de energia dos experimentos. Não ocorreu nenhum erro que pudesse afetar o resultado final dos experimentos do trabalho nos testes preliminares realizado com esse banco. Também é possível utilizar essa forma de armazenamento (com InfluxDB) das métricas de energia numa situação de monitoramento contínuo, porém a necessidade de manter um banco de dados exclusivo para as aferições de energia pode ser oneroso para uma instalação já existente que já possui esteiras de monitoramento compatíveis com OpenTelemetry.

A Figura 5 mostra o fluxo simplificado do registro e armazenamento das aferições de energia de cada experimento. Em suma, o ALUMET amostra repetidamente os medidores de energia expostos pela interface RAPL, e os relaciona com as informações de uso de recursos do sistema por processos obtidas pela funcionalidade *perf_ events* do *kernel* do Linux. Assim, para cada instante de tempo amostrado, o consumo de energia de cada processo é estimado e armazenado no banco de dados InfluxDB.

[7] https://www.influxdata.com/products/influxdb/

Figura 5 – Diagrama de sequência da obtenção das métricas de consumo de energia e CPU dos experimentos.



5 RESULTADOS E AVALIAÇÃO CRÍTICA

Cada implementação foi submetida trinta vezes seguidas ao teste de carga médio de 4096 incrementos por segundo do contador descrito na Seção 4.3.1, com intervalos de "descanso" como recomendado por Ournani et al. (2020). Após 30 segundos, o k6 aumenta de 0 requisições por segundo para 4096 requisições por segundo de forma linear no decorrer de 30 segundos. Nos 150 segundos seguintes, a taxa de 4096 requisições por segundo é mantida, e nos últimos 30 segundos a taxa é reduzida de forma linear para 0 requisições por segundo. Isso dura quatro minutos, para um total de 737279 incrementos realizados, e o ciclo completo dos trinta testes de uma implementação dura aproximadamente duas horas. A Figura 6 indica o número de requisições por segundo durante uma execução do teste de carga médio, conforme o Código 1.

O número de incrementos por segundo foi escolhido de forma a provocar pelo menos 10% de uso de cpu na máquina sendo avaliada, para que haja um consumo de energia suficientemente mensurável (isto é, bem distinto da variância de consumo de energia do computador ocioso) durante a execução do experimento — de forma arbitrária, decidimos alinhar esse valor com uma potência do número dois. Enquanto o processo do experimento estava sendo validado, cada implementação foi submetida a apenas dez iterações do teste de carga médio com duração reduzida, e com essa configuração obtivemos resultados com pouca significância estatística. Para aumentar a significância estatística do resultado final, então, decidimos utilizar mais iterações: testamos trinta iterações de quatro minutos cada, e vimos que com essa configuração já era possível obter resultados com significância estatística. Arcuri e Briand (2011) recomendam pelo menos $n = 1000$ iterações para comparações de algoritmos randomizados como o Raft, mas como $n = 30$ já fez com que nosso experimento demorasse quatorze horas e já foi suficiente para obter significância estatística, consideramos desnecessário, ineficiente e inconveniente realizar o experimento com um número maior de iterações.

Para executar os três participantes do algoritmo Raft, foi utilizado apenas um computador, com 48GB de memória DDR4 e processador AMD Ryzen 7 5700X3D, no sistema operacional Ubuntu 24.04 LTS. O banco de dados InfluxDB e o teste de carga foram executados a partir de outro computador, conectado em rede local ao computador que executou o algoritmo descrito acima, de modo que o computador que executa os algoritmos tenha em execução apenas o algoritmo e o extrator ALUMET. Os *scripts* utilizados nessa orquestração e o código-fonte das implementações testadas estão todos disponíveis publicamente no GitHub[1]. A Figura 7 mostra a alocação dos serviços nos dois computadores utilizados.

[1] https://github.com/DCarts/raft-implementations

Figura 6 – Requisições por segundo durante teste de carga médio.

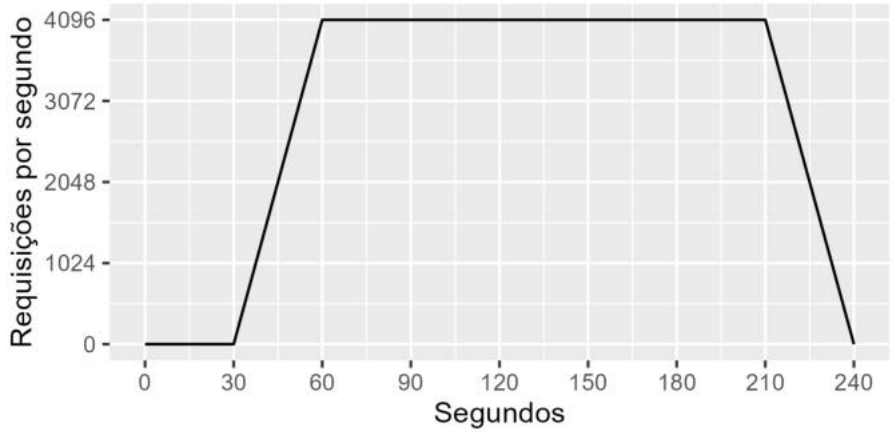
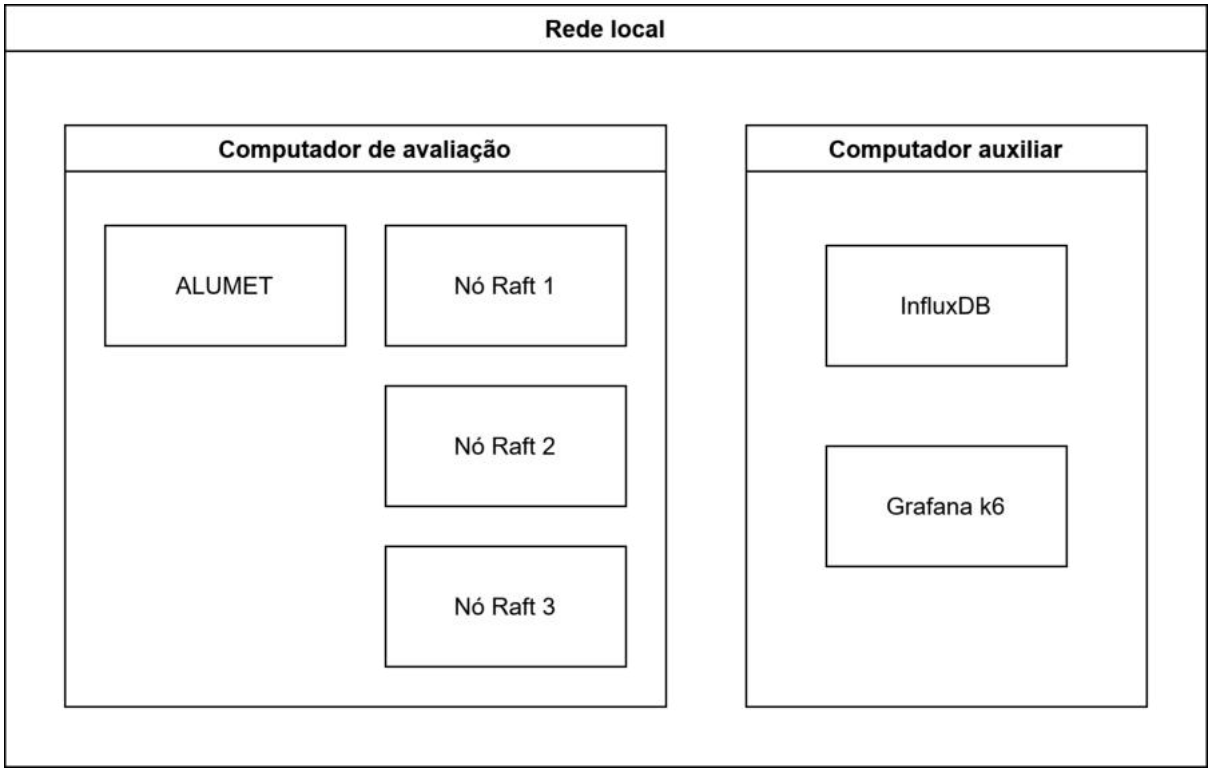


Figura 7 – Alocação dos serviços nos computadores utilizados no experimento.

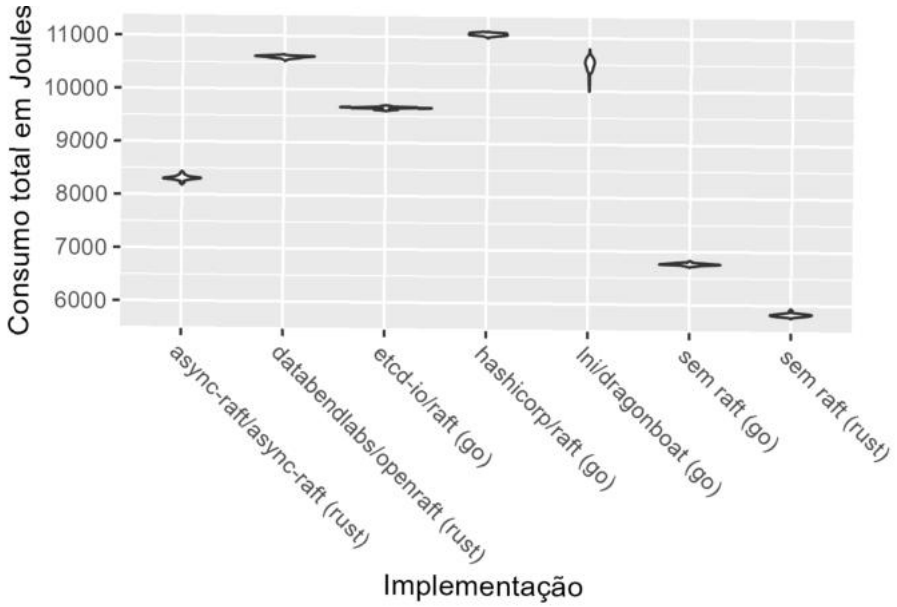


O tempo total dos processos de sistema das implementações avaliadas durante o teste de carga médio oscilou entre 240 e 243 segundos, o que acarretou numa oscilação do número de registros coletados de cada teste: a quantidade de registros mais comum foi 2178, a maior foi 2193 e a menor foi 2160. Essa variação não afeta os resultados de forma significativa, pois ela ocorreu no período de descanso do teste de carga.

O consumo de energia total de cada teste de carga médio pelos algoritmos Raft pode ser visualizado na Figura 8. É importante notar que esses resultados foram obtidos sob um comportamento uniformizado do algoritmo Raft, no qual não ocorreu troca de líder

nem outros fatores que poderiam afetar a reprodutibilidade do teste. A busca por uniformização do funcionamento do algoritmo Raft pode introduzir um viés nos resultados obtidos; contudo, é possível extrair dos resultados que o consumo de energia do sistema proveniente da escolha de implementação do Raft é impactante o suficiente para ser notado alheio a esse viés de metodologia. Pelos resultados obtidos, pode-se confirmar que as versões do experimento sem o Raft possuem maior eficiência energética, como esperado: tanto a versão Go quanto a versão Rust são as que consumiram menos energia no total. O aumento no consumo de energia entre a versão sem Raft e a versão com Raft mais eficiente é de 43% nas duas linguagens, 42,8% para a implementação etcd-io/raft em Go, e 42,6% para a implementação async-raft/async-raft em Rust. Esses números confirmam que há um custo energético em tornar uma aplicação resistente à falhas de parada por meio da implantação do algoritmo Raft, e indicam que pode existir uma espécie de "piso" associado a esse custo.

Figura 8 – Gráfico de violino do consumo de energia total por implementação.



Para verificar se os valores de consumo de energia obtidos de cada implementação são de fato distintos entre si, utilizamos o teste U de Mann-Whitney par a par, como recomendado por Arcuri e Briand (2011), apesar de não ser estritamente necessário, visto que pelo gráfico de violino já é possível notar diferença significativa entre o consumo de energia de cada implementação. Esse teste faz uso das trinta amostras de consumo de energia que obtivemos do teste de carga de cada implementação para avaliar a probabilidade das amostras terem sido obtidas de uma mesma distribuição de probabilidade, sem fazer suposições sobre a distribuição dos dados ou parâmetros da população. Nesse cenário, como descrito em 4.1, a variável dependente sob análise é a quantidade de energia consumida pela implementação Raft analisada, que é nossa variável dependente. A Tabela 1 indica os valores-p par a par calculados, e a Tabela 2 mostra a média, a mediana,

e o desvio padrão de cada implementação no experimento, em joules. Os valores obtidos indicam que a mediana do consumo total de energia das implementações Raft avaliadas e das versões desenvolvidas sem o algoritmo em Go e Rust são todas distintas entre si, com alta confiança (valores-p baixíssimos).

Tabela 1 – Valores-p do consumo de energia das implementações.

-	async-raft	openraft	etcd	hashicorp	dragonboat	sem raft, go
openraft	3.55e-16	-	-	-	-	-
etcd	3.55e-16	3.55e-16	-	-	-	-
hashicorp	3.55e-16	3.55e-16	3.55e-16	-	-	-
dragonboat	3.55e-16	4.37e-05	3.55e-16	3.55e-16	-	-
sem raft, go	3.55e-16	3.55e-16	3.55e-16	3.55e-16	3.55e-16	-
sem raft, rust	3.55e-16	3.55e-16	3.55e-16	3.55e-16	3.55e-16	3.55e-16

Tabela 2 – Média, mediana, e desvio padrão do consumo de energia das implementações, em joules.

Implementação	Média	Mediana	Desvio padrão
asyncraft	8309.15	8308.34	39.91
openraft	10603.74	10609.40	18.93
etcd	9657.98	9661.99	19.87
hashicorp	11060.52	11058.30	21.76
dragonboat	10501.36	10532.40	118.79
sem raft, go	6766.66	6764.57	18.71
sem raft, rust	5824.56	5823.19	25.22

Em valores absolutos, a implementação que menos consumiu energia foi escrita em Rust, e a versão de melhor desempenho energético sem o algoritmo também foi escrita em Rust. Isso pode indicar que a linguagem da implementação é um fator importante no consumo de energia, mas não é um fator decisivo: uma das implementações em Rust consumiu mais energia no total do que duas implementações em Go.

O consumo de energia da versão sem Raft em ambas as linguagens representa mais da metade do consumo de energia de todas as implementações analisadas. Isso indica que o algoritmo utilizado para consenso não é a parte responsável pela maior parte do consumo de energia das aplicações utilizadas, e por isso talvez seja mais eficiente investir esforços em otimizar outros componentes de uma aplicação para reduzir a sua pegada energética. Apesar disso, a parte do algoritmo Raft não é insignificante, e os resultados obtidos pelo experimento mostram que é possível sim reduzir de forma significativa o consumo total de uma aplicação escolhendo cuidadosamente a implementação do algoritmo Raft utilizada.

6 CONCLUSÃO

O uso do algoritmo Raft para obter qualidades como alta disponibilidade e resistência à falhas não bizantinas em sistemas distribuídos provoca um aumento significante no consumo de energia. Esse consumo energético é mensurável por ferramentas de monitoramento de consumo de energia como o ALUMET, que pode ser usado para monitoramento contínuo de sistemas distribuídos, ou de forma pontual para experimentos científicos como o realizado neste trabalho. O experimento realizado com cinco implementações do algoritmo Raft indicam que elas consomem quantidades diferentes de energia em sua execução quando submetidas a uma mesma carga de escrita em um intervalo de tempo fixo e sem a influência de falhas, logo a escolha da implementação do algoritmo em um sistema distribuído é um fator que pode impactar o consumo de energia total. Os resultados também auxiliam na escolha de implementações do algoritmo Raft para utilização no desenvolvimento de aplicações novas, ou aplicações que precisam migrar alguma funcionalidade para funcionar em contextos distribuídos com eficiência energética.

Numa situação real com falhas, que podem provocar retransmissões ou até troca de líder, o consumo de energia pode ser maior, mas por definição o algoritmo só funciona bem se essas falhas são suficientemente infrequentes, o que diminui o impacto delas no consumo de energia total do sistema a longo prazo. Apesar desse resultado, com base nos números obtidos do experimento, conclui-se que o fator predominante no consumo de energia é o restante da aplicação que não engloba o algoritmo Raft, e por isso talvez seja mais produtivo investir esforços em otimizar o consumo de energia nessa parte da aplicação.

Em aplicações de larga escala que utilizam Raft como algoritmo de consenso, a escolha de implementação pode exacerbar as diferenças energéticas apresentadas neste trabalho, assim podendo representar um grande impacto não só energético, mas também ambiental. Portanto, em termos de desenvolvimento sustentável, há importância na escolha da implementação e linguagem utilizada — inclusive na escolha do próprio algoritmo de consenso, que, embora fora do escopo deste trabalho, também deve ser considerado.

A necessidade de executar os experimentos por um longo período de tempo para obter medições precisas do consumo de energia de cada implementação foi desafiador, pois é difícil manter inalteradas as condições externas ao sistema por longos períodos de tempo sem grandes custos de infraestrutura, como fontes de energia e de resfriamento dedicadas. Outro desafio foi definir um teste que funcione em implementações Raft em linguagens de programação diferentes sem enviesar os resultados por conta da sobrecarga (*overhead*) das adaptações feitas para adequar o teste. Foi mais simples disponibilizar os resultados do teste sem o Raft para cada linguagem de modo a visualizar de forma separada essa sobrecarga.

6.1 AMEAÇAS À VALIDADE

Os experimentos desse trabalho abrangem apenas cinco implementações populares do algoritmo Raft, em apenas duas linguagens de programação, o que não representa uma amostragem completa das implementações de código aberto do algoritmo Raft, nem cobre todas as linguagens nas quais alguma implementação está disponível. É possível que outros resultados sejam obtidos ao analisar outras implementações populares em outras linguagens.

A máquina de estados utilizada no experimento é outro fator de viés que pode ter impactado os resultados obtidos. Por ser uma máquina de estados demasiadamente simples, o perfil do consumo de energia em aplicações reais pode ser muito diferente dos medidos no experimento, e por isso a mesma análise feita em outras aplicações (outras máquinas de estado replicadas) pode obter diferentes resultados. O teste realizado neste trabalho foi focado em escritas na máquina de estados replicada, e mesmo utilizando a mesma máquina de estados, pode ser possível que diferentes resultados sejam obtidos caso outro tipo de teste fosse realizado, como por exemplo um que realiza somente leituras, ou uma mistura de leituras e escritas. Outros conjuntos de parâmetros do algoritmo Raft também poderiam gerar outro conjunto de resultados, pois algumas implementações podem ser especialmente sensíveis a alguns parâmetros, e o consumo de energia total escalar de forma diferente com alguns parâmetros em diferentes implementações.

Por fim, o consumo de energia das implementações foi mensurado durante o estado mais comum do algoritmo, que é sem falhas e sem transições de termos ou eleições de líderes. Por mais que essas operações sejam pouco frequentes, e por isso possivelmente não impactem o consumo total numa execução longa o suficiente do algoritmo, esse resultado não foi confirmado por esse experimento: é possível que essas operações consumam energia de forma diferente nas implementações a ponto de impactar o consumo de energia total de forma significante. Realizamos experimentos preliminares para verificar a viabilidade de mensurar o consumo de energia do algoritmo sob o efeito dessas transições de estado, e concluímos que é possível, porém torna-se mais difícil isolar o impacto de cada variável nova no cálculo final do consumo de energia. Isso poderia ser contornado prolongando o tempo total do experimento, mas o ambiente de testes utilizado não é capaz de manter as condições externas (por exemplo, temperatura) estáveis o suficiente por esse período maior do experimento, o que poderia impactar os resultados obtidos.

6.2 TRABALHOS FUTUROS

Há diversos caminhos para expandir e aprofundar os resultados dos experimentos desse trabalho. Pode-se analisar mais implementações do algoritmo Raft, o que geraria um panorama melhor do consumo de energia das implementações de código aberto disponíveis.

A partir daí, também é possível incluir na comparação outros algoritmos de consenso, incluindo algoritmos resistentes a falhas bizantinas, e suas diversas implementações.

Pode-se também analisar o desempenho das implementações sob diferentes condições, como latência maior entre nós, perda de pacotes, diferentes quantidades de nós participantes, e diferentes parâmetros do algoritmo. Isso pode evidenciar que algumas implementações podem ser mais eficientes em alguns cenários do que outros. Nesse sentido, também é possível analisar diferentes máquinas de estado, que possuem, por exemplo, tamanho médio de entrada de registro de transação diferentes, e diferentes perfis de escrita e leitura. Isso também auxiliaria a identificar o desempenho de cada implementação em cenários diferentes.

Há espaço também para ampliar o escopo da análise realizada, para não englobar apenas o consumo de energia total do sistema, mas também uma avaliação da eficiência em transformar o consumo de energia em trabalho útil. Isso pode ser obtido através do estudo de outras métricas, como commits/joule e commits/segundo, além de métricas relacionadas à latência da comunicação entre os nós. É possível até sintetizar o *trade-off* de energia e tempo de execução em um único indicador, com métricas compostas como *Energy-Delay Product* (EDP).

O código-fonte das implementações utilizadas e dos scripts de teste estão disponíveis publicamente no GitHub[1], o que permite a expansão do trabalho com outras implementações ou outros modelos de teste com esforço inicial reduzido: é possível iniciar esses trabalhos futuros a partir desses códigos já existentes.

[1] https://github.com/DCarts/raft-implementations

REFERÊNCIAS

ARCURI, A.; BRIAND, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2011. (ICSE '11), p. 1–10. ISBN 9781450304450. Disponível em: https://doi.org/10.1145/1985793.1985795.

BUNSE, C. On the Impact of Code Obfuscation to Software Energy Consumption. In: OTJACQUES, B. et al. (Ed.). **From Science to Society**. Cham: Springer International Publishing, 2018. p. 239–249. ISBN 978-3-319-65687-8.

CRUZ, L. **Tools to Measure Software Energy Consumption from your Computer**. 2021. Disponível em: http://luiscruz.github.io/2021/07/20/measuring-energy.html. Acesso em: 20 set. 2025.

CRUZ, L.; ABREU, R. Performance-Based Guidelines for Energy Efficient Mobile Applications. In: **2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**. [s.n.], 2017. p. 46–57. Disponível em: https://ieeexplore.ieee.org/document/7972717.

DAMASCENO, E. et al. Comparative analysis of compiler efficiency: Energy consumption metrics in high-performance computing domains. In: **Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho**. Porto Alegre, RS, Brasil: SBC, 2024. p. 252–263. ISSN 0000-0000. Disponível em: https://proceedings-sol.sbc.org.br/index.php/sscad/article/view/31002.

DÍAZ, A. F. et al. Vampire: A smart energy meter for synchronous monitoring in a distributed computer system. **Journal of Parallel and Distributed Computing**, v. 184, p. 104794, fev. 2024. ISSN 0743-7315. Disponível em: https://www.sciencedirect.com/science/article/pii/S0743731523001648.

DURÁN, F. et al. Insights into resource utilization of code small language models serving with runtime engines and execution providers. **Journal of Systems and Software**, v. 230, p. 112574, dez. 2025. ISSN 01641212. ArXiv:2412.15441 [cs]. Disponível em: http://arxiv.org/abs/2412.15441.

FIENI, G.; ROUVOY, R.; SEINTURIER, L. Smartwatts: Self-calibrating software-defined power meter for containers. In: **2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)**. [S.l.: s.n.], 2020. p. 479–488.

GELENBE, E.; CASEAU, Y. The impact of information technology on energy consumption and carbon emissions. **Ubiquity**, v. 2015, n. June, p. 1:1–1:15, jun. 2015. Disponível em: https://dl.acm.org/doi/10.1145/2755977.

HOWARD, H.; MORTIER, R. Paxos vs raft: have we reached consensus on distributed consensus? In: **Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data**. New York, NY, USA: Association for

Computing Machinery, 2020. (PaPoC '20). ISBN 9781450375245. Disponível em: https://doi.org/10.1145/3380787.3393681.

HUANG, S. et al. Measurement and characterization of Haswell power and energy consumption. In: **Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing**. New York, NY, USA: Association for Computing Machinery, 2015. (E2SC '15), p. 1–10. ISBN 978-1-4503-3994-0. Disponível em: https://doi.org/10.1145/2834800.2834807.

JANES, A.; LI, X.; LENARDUZZI, V. Open tracing tools: Overview and critical comparison. **J. Syst. Softw.**, Elsevier Science Inc., USA, v. 204, n. C, out. 2023. ISSN 0164-1212. Disponível em: https://doi.org/10.1016/j.jss.2023.111793.

KHAN, K. N. et al. RAPL in Action: Experiences in Using RAPL for Power Measurements. **ACM Trans. Model. Perform. Eval. Comput. Syst.**, v. 3, n. 2, p. 9:1–9:26, mar. 2018. ISSN 2376-3639. Disponível em: https://doi.org/10.1145/3177754.

MANCEBO, J. et al. FEETINGS: Framework for Energy Efficiency Testing to Improve Environmental Goal of the Software. **Sustainable Computing: Informatics and Systems**, v. 30, p. 100558, jun. 2021. ISSN 2210-5379. Disponível em: https://www.sciencedirect.com/science/article/pii/S2210537921000494.

MANOTAS, I. et al. An empirical study of practitioners' perspectives on green software engineering. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (ICSE '16), p. 237–248. ISBN 978-1-4503-3900-1. Disponível em: https://dl.acm.org/doi/10.1145/2884781.2884810.

MANOTAS, I. et al. Investigating the impacts of web servers on web application energy usage. In: **Proceedings of the 2nd International Workshop on Green and Sustainable Software**. San Francisco, California: IEEE Press, 2013. (GREENS '13), p. 16–23. ISBN 978-1-4673-6267-2.

NOUREDDINE, A. et al. A preliminary study of the impact of software engineering on GreenIT. In: **2012 First International Workshop on Green and Sustainable Software (GREENS)**. [s.n.], 2012. p. 21–27. Disponível em: https://ieeexplore.ieee.org/document/6224251.

ONGARO, D. **Raft Consensus Algorithm**. 2025. Disponível em: https://raft.github.io/. Acesso em: 20 set. 2025.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 USENIX Annual Technical Conference (USENIX ATC 14)**. Philadelphia, PA: USENIX Association, 2014. p. 305–319. ISBN 978-1-931971-10-2. Disponível em: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

OURNANI, Z. et al. Taming Energy Consumption Variations In Systems Benchmarking. In: **Proceedings of the ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICPE '20), p. 36–47. ISBN 978-1-4503-6991-6. Disponível em: https://doi.org/10.1145/3358960.3379142.

OURNANI, Z. et al. Evaluating the Impact of Java Virtual Machines on Energy Consumption. In: **15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. Bari, Italy: [s.n.], 2021. Disponível em: https://inria.hal.science/hal-03275286.

OURNANI, Z. et al. Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption. In: **ICSOFT 2021 - 16th International Conference on Software Technologies**. Virtual, France: [s.n.], 2021. Disponível em: https://hal.science/hal-03202437.

OURNANI, Z. et al. Tales from the Code #2: A Detailed Assessment of Code Refactoring's Impact on Energy Consumption. In: FILL, H.-G.; SINDEREN, M. v.; MACIASZEK, L. A. (Ed.). **ICSOFT 2021 : Software Technologies**. Virtual Event, France: Springer, 2021. (Communications in Computer and Information Science, v. 1622), p. 94–116. Disponível em: https://hal.science/hal-03774328.

PÂRIS, J.-F.; LONG, D. D. E. Reducing the Energy Footprint of a Distributed Consensus Algorithm. In: **2015 11th European Dependable Computing Conference (EDCC)**. [s.n.], 2015. p. 198–204. Disponível em: https://ieeexplore.ieee.org/abstract/document/7371967.

PÉREZ, C. et al. **A Comparative Analysis of Energy Consumption Between The Widespread Unreal and Unity Video Game Engines**. 2024. Disponível em: https://arxiv.org/abs/2402.06346.

PINTO, G.; CASTOR, F. Energy efficiency: a new concern for application software developers. **Commun. ACM**, v. 60, n. 12, p. 68–75, nov. 2017. ISSN 0001-0782. Disponível em: https://dl.acm.org/doi/10.1145/3154384.

POY, O. et al. Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study. **Journal of Systems and Software**, v. 222, p. 112303, abr. 2025. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121224003479.

RAFFIN, G.; TRYSTRAM, D. Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis. **IEEE Trans. Parallel Distrib. Syst.**, v. 36, n. 1, p. 96–107, jan. 2025. ISSN 1045-9219. Disponível em: https://doi.org/10.1109/TPDS.2024.3492336.

SAHIN, C. et al. Initial explorations on design pattern energy usage. In: **2012 First International Workshop on Green and Sustainable Software (GREENS)**. [s.n.], 2012. p. 55–61. Disponível em: https://ieeexplore.ieee.org/abstract/document/6224257.

SAHIN, C.; POLLOCK, L.; CLAUSE, J. How do code refactorings affect energy usage? In: **Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: Association for Computing Machinery, 2014. (ESEM '14), p. 1–10. ISBN 978-1-4503-2774-9. Disponível em: https://doi.org/10.1145/2652524.2652538.

SAHIN, C. et al. How does code obfuscation impact energy usage? **Journal of Software: Evolution and Process**, v. 28, n. 7, p. 565–588, 2016. ISSN 2047-7481. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1762. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1762.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/98163.98167.

VAN STEEN, M.; TANENBAUM, A. S. **Distributed Systems**. 4. ed. Maarten van Steen, 2023. ISBN 978-90-815406-4-3. Disponível em: distributed-systems.net.

WIKIPEDIA CONTRIBUTORS. **Paxos (computer science) — Wikipedia, The Free Encyclopedia**. 2025. Disponível em: https://en.wikipedia.org/w/index.php?title=Paxos_(computer_science)&oldid=1311532955#Production_use_of_Paxos. Acesso em: 20 set. 2025.

WIKIPEDIA CONTRIBUTORS. **Raft (algorithm) — Wikipedia, The Free Encyclopedia**. 2025. Disponível em: https://en.wikipedia.org/w/index.php?title=Raft_(algorithm)&oldid=1301374802#Production_use_of_Raft. Acesso em: 20 set. 2025.

YU, D.; ZHANG, L. Centralized and distributed consensus in wireless network: An analytical comparison. In: **2022 IEEE 20th International Conference on Embedded and Ubiquitous Computing (EUC)**. [S.l.: s.n.], 2022. p. 81–89.

ZHANG, Z.; FU, S. Macropower: A coarse-grain power profiling framework for energy-efficient cloud computing. In: **Proceedings of the 30th IEEE International Performance Computing and Communications Conference**. USA: IEEE Computer Society, 2011. (PCCC '11), p. 1–8. ISBN 978-1-4673-0010-0. Disponível em: https://doi.org/10.1109/PCCC.2011.6108061.

GLOSSÁRIO

ALUMET é uma ferramenta modular para obtenções de métricas de forma local ou distribuída

Go é uma linguagem de programação

InfluxDB é uma banco de dados de série temporal.

k6 é uma ferramenta de código-fonte aberto para testes de carga desenvolvido pela Grafana Labs

OpenTelemetry é um conjunto de padrões e tecnologias projetado para facilitar a geração, exportação e coleta de dados de telemetria

Raft é um algoritmo de consenso distribuído

Rust é uma linguagem de programação