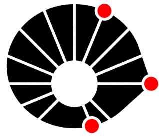
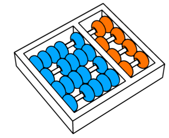




Um estudo sobre placement com métricas de consumo de energia para sistemas auto-distribuídos

J. A. M. Seródio   *L. F. Bittencourt*   *R. R. Filho*

Relatório Técnico - IC-PFG-22-38

Projeto Final de Graduação

2022 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Um estudo sobre placement com métricas de consumo de energia para sistemas auto-distribuídos

João Alberto Moreira Seródio* Luiz Fernando Bittencourt*
Roberto Rodrigues Filho†

Resumo

Sistemas auto-distribuídos são capazes de alterar sua composição em tempo de execução para se adaptar ao ambiente que se encontram e melhor atender aos seus requisitos. Para tanto, utilizam-se de métricas coletadas durante a execução dos componentes, como tempo de execução. Este trabalho tem como objetivo habilitar a utilização de métricas de consumo energético para sistemas auto-distribuídos, para que o *placement* dos componentes possa ser realizado de maneira a atingir uma configuração energeticamente eficiente, e então verificar o impacto que isso tem no sistema.

1 Introdução

A crescente complexidade dos sistemas de software modernos fez com que maneiras de reduzir a necessidade de interação humana nos processos de implantação, configuração e otimização de sistemas fossem desenvolvidas. Nisso, a Computação Autonômica e Sistemas Auto-adaptativos surgem como base para sistemas que conseguem se auto-gerenciar a partir de políticas e objetivos de alto nível descritos pelo administrador do sistema. [1]

Partindo da autogestão da Computação Autonômica, Filho [2] introduz o conceito de Sistemas de Softwares Emergentes, no qual sistemas são compostos por unidades de software capazes de auto-composição e auto-otimização, podendo assim se adaptar ao ambiente em que se encontram e reagir a mudanças. Para a implementação desses sistemas, a linguagem de programação Dana [3] pode ser utilizada, tendo em vista que possui a capacidade de se adaptar em tempo de execução com a troca de seus componentes de maneira rápida. No contexto de sistemas distribuídos, há a questão da auto-distribuição [4] onde o *placement* e configuração do sistema é realizado durante sua execução e leva em conta o ambiente que se encontra.

Em um ambiente de nós heterogêneos, como realizar o *placement* dos serviços é um problema complexo, e pode fazer uso de métricas como tempo de resposta/execução e carga do sistema para escolha do nó. Uma métrica que nem sempre é empregada para a tomada de decisão é o consumo energético, o que pode ser atribuído à dificuldade de obtenção. Porém,

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Instituto de Informática, Universidade Federal de Goiás, 74690-900 Goiânia, GO

dado o gasto energético de sistemas computacionais [5], utilizar métricas de consumo de energia podem ajudar na obtenção de sistemas mais eficientes.

Com isso, este trabalho busca habilitar o uso de métricas de consumo energético no *placement* de serviços de um sistema auto-distribuído, possibilitando estratégias para tornar o sistema mais eficiente energeticamente. Para isso, ferramentas como PAPI [6] e modelos matemáticos são utilizados para a estimação do consumo energético dos nós.

Este relatório técnico é dividido da seguinte maneira. A Seção 2 provê uma breve apresentação de conceitos utilizados na elaboração do projeto. Na Seção 3 o objetivo geral do relatório é apresentado, assim como algumas perguntas. A Seção 4 discorre sobre como esse trabalho busca responder às perguntas levantadas na Seção 3, apresentando como os experimentos foram realizados. Os resultados dos experimentos estão presentes na Seção 5. Por último, a Seção 6 apresenta a conclusão, com alguns possíveis trabalhos futuros que derivam deste projeto.

2 Referencial Teórico

Esta seção apresenta conceitos que servem como fundamentação teórica do projeto, sendo eles Computação Autonômica e Sistemas Auto-adaptativos, Sistemas de Softwares Emergentes e Computação Energeticamente Eficiente. Além disso, há uma breve apresentação de computação em nuvem e na borda, dado que ambos os ambientes estão presentes no projeto. Finalmente, virtualização baseada em contêineres e orquestração de contêineres, que são tecnologias empregadas na execução dos experimentos, são discutidas.

2.1 Computação Autonômica e Sistemas Auto-adaptativos

Com o manifesto publicado pela IBM em 2001, falando sobre uma iminente crise na indústria de TI devido a crescente complexidade dos softwares, soluções para diminuir a dificuldade de gerenciamento de aplicações passaram a surgir, sendo uma delas a computação autonômica. Kephart e Chess [1] discutem sobre essa alternativa, apontando 4 aspectos de auto-gerenciamento desses sistemas:

- Auto-configuração: sistemas autonômicos se configuram de maneira automática seguindo políticas de alto nível;
- Auto-otimização: buscam continuamente maneiras de melhorar sua operação, tornando-os mais eficientes em questões de custo e desempenho;
- Auto-cura: são capazes de detectar, diagnosticar e reparar problemas causados por *bugs* e falhas de software e hardware;
- Auto-proteção: conseguem se defender de problemas de larga escala causados por ataques maliciosos ou falhas em cascata, além de antecipar possíveis problemas e tomar medidas para evitá-los.

Fora esses aspectos, Kephart e Chess ainda apresentam a arquitetura de sistemas autonômicos, que serão formados por elementos autonômicos. Esses elementos autonômicos

por sua vez são responsáveis pela gestão de seu comportamento interno e como se relacionam com os demais elementos do sistema.

2.2 Sistemas de Softwares Emergentes

Filho [2] define Sistemas de Softwares Emergentes como sistemas construídos com unidades de comportamento de software, que são capazes de auto-composição e auto-otimização como resultado de características de seu ambiente de operação. Para possibilitar esses sistemas, o framework *PAL* [2] foi proposto e sua arquitetura pode ser visualizada na Figura 1.

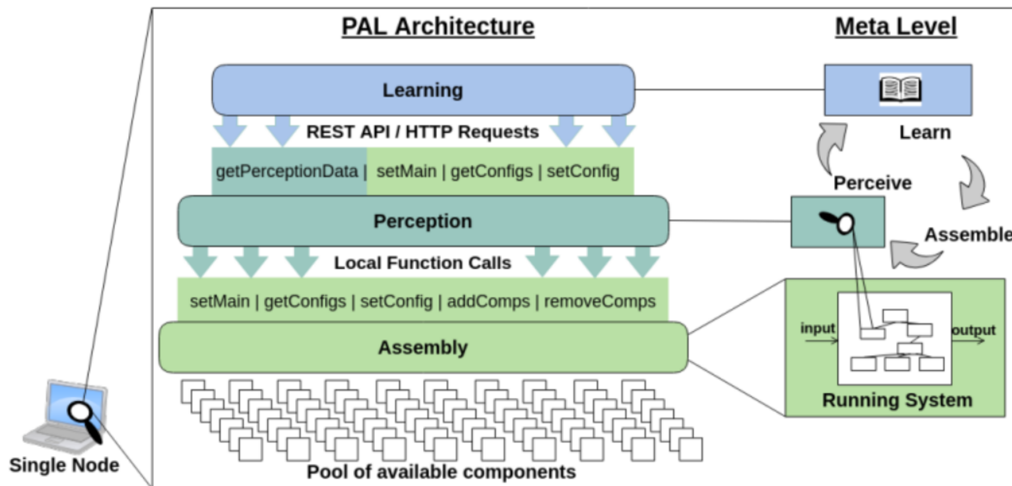


Figura 1: Arquitetura do Framework PAL [2].

Os três módulos do framework são:

- ***Assembly***: utiliza um *runtime* baseado em componentes para realizar a troca de componentes e composições em tempo de execução, além de gerir as possíveis composições do sistema do sistema e prover essas informações para as módulos acima;
- ***Perception***: monitora o status do sistema e coleta informações do ambiente de operação;
- ***Learning***: faz uso de aprendizado de reforço para adaptar o sistema de acordo com objetivos estabelecidos previamente.

O *runtime* utilizado pelo módulo *Assembly* é o da linguagem de programação Dana [3], que também é utilizado por aplicações analisadas neste trabalho.

2.3 Computação Energeticamente Eficiente

A energia na computação é uma preocupação ambiental séria, e mitigá-la é um importante desafio tecnológico. Assim, é necessário ter uma medição de consumo energético durante

a execução de uma aplicação para utilizar técnicas de minimização de energia no nível da aplicação, e os três métodos populares de medição são: utilização de um medidor de potência externo, utilização de sensores no chip, e modelos preditivos de energia. [7]

Uma ferramenta comumente utilizada para a medição de consumo energético de computadores é a *Performance Application Programming Interface* (PAPI) [6], que permite a obtenção de métricas de energia da execução de uma aplicação a partir de contadores de performance do processador [8]. Outra abordagem está na utilização de modelos para estimar o consumo energético baseado em outras métricas do sistema, como a utilização da CPU. Kaup, Gottschling e Hausheer [9] apresentam modelos energéticos para um Raspberry Pi em diferentes cenários de uso.

Neste trabalho, a otimização que se busca realizar está no nível do sistema (*cluster*), para tanto, será explorado o *placement* de aplicações em diferentes nós, a fim de verificar como o tempo de execução e o consumo energético são impactados pelas diferenças entre os hardwares empregados.

2.4 Computação em Nuvem

A computação em nuvem é um modelo de negócio no qual os clientes utilizam recursos computacionais e serviços providos por uma plataforma de nuvem e pagam apenas pelo que usam. A fim de atender aos clientes, a infraestrutura física do provedor de nuvem é virtualizada, resultando em redes e CPUs virtuais (vCPUs), o que possibilita o compartilhamento de um mesmo recurso entre diferentes clientes (*multi-tenancy*) e resulta em um ambiente elástico e dinâmico.

Os serviços de um provedor de nuvem podem ser divididos em 3 categorias: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS). A primeira consiste dos recursos que são a base de uma plataforma de nuvem, como máquinas virtuais (VMs) e armazenamento (discos); PaaS já provê recursos de uma camada de abstração acima do IaaS, que podem ser consumidos por APIs específicas; e por último, SaaS são aplicações executadas na nuvem que atendem os usuários. [10]

Os recursos computacionais se encontram em grandes *datacenters* de provedores de serviços de nuvem como Google, AWS e Azure, e são distribuídos geograficamente, usualmente em regiões que favorecem a implantação da infraestrutura física.

No projeto a nuvem é utilizada apenas para o gerenciamento dos nós da borda e implantação de aplicações.

2.5 Computação na Borda

A computação na borda é um novo paradigma computacional na qual a computação é realizada na borda da rede, passando a ser mais próxima do usuário final [11]. Ela vem ganhando mais popularidade devido ao crescimento de aplicações de Internet das Coisas, dado que permite reduzir a carga de trabalho e tráfego de dados no núcleo da rede, já que o processamento passa a ser realizado na mesma rede local que o dispositivo e também consegue atender aplicações com requisitos de latência baixa [12].

A borda é utilizada para a execução dos experimentos, visto que computadores com características distintas são empregados e não utiliza-se VMs (acesso a contadores de performance é reduzido em VMs na nuvem).

2.6 Virtualização baseada em contêineres

Smith e Nair [13] dividem as VMs em dois tipos, VMs de processo e VMs de sistema. A primeira é uma virtualização realizada em cima do sistema operacional (SO) que visa somente suportar o processo da aplicação, já a segunda consiste em um ambiente que suporta um SO hospedeiro junto com seus demais processos e utiliza um Monitor de Máquina Virtual (VMM). Atualmente a VM de processo é usualmente denominada contêiner, e teve um crescimento de popularidade nos últimos devido a tecnologias como Docker [14]. Contêineres são leves e possuem as dependências necessárias para a execução da aplicação, assim facilitam consideravelmente a implantação de serviços em ambientes dinâmicos e a portabilidade do software.

Contêineres Docker foram utilizados para a execução das aplicações nos experimentos.

2.7 Orquestração de Contêineres

Orquestradores de contêineres tem como função facilitar o gerenciamento dos contêineres que compõem uma aplicação em um *cluster* de computadores. Uma *engine* muito utilizada para esse propósito é o Kubernetes [15], onde a menor unidade de software implantável é denominada *pod*. A fim de estender as funcionalidades de orquestração do Kubernetes para a borda, o sistema do KubeEdge [16] pode ser empregado, possibilitando a conexão de nós da borda a um cluster Kubernetes provisionado na nuvem.

3 Objetivos

Este projeto tem como objetivo explorar o *placement* de serviços na borda de forma a obter um sistema energeticamente eficiente. Desse modo, maneiras de se obter métricas de consumo energético serão estudadas, assim como estratégias de adaptação de um sistema auto-distribuído para atingir uma configuração energeticamente eficiente utilizando as métricas coletadas. Com isso, as seguintes questões serão trabalhadas:

- Quais métricas utilizar para determinar a implantação de serviços na borda de maneira eficiente?

- Como obter métricas de consumo energético e disponibilizá-las de maneira simples para a aplicação auto-distribuída?

- Como o consumo energético se relaciona com o tempo de execução de uma aplicação e suas características (e.g. computacionalmente intensiva)?

4 Metodologia

A fim de responder às questões da seção de objetivos, uma metodologia empírica foi adotada no desenvolvimento do trabalho. Algumas aplicações com características diferentes, como multiplicação de matrizes, que é computacionalmente intensiva, e um servidor web com operações de escrita e leitura em uma lista, o que reflete uma carga de trabalho comum em serviços web, são instrumentadas para utilizar um serviço que provê métricas de consumo energético e tempo de execução.

Como o projeto busca explorar estratégias de placement, computadores com diferentes processadores e características energéticas foram utilizados para execução das aplicações, resultando em perfis energéticos que podem ser comparados e utilizados na tomada de decisão de onde a aplicação deve ser implantada. Assim, partindo desses dados coletados, uma análise de como diferentes estratégias impactam o sistema em questões de consumo energético e tempo de execução é realizada.

A coleta de métricas referentes ao consumo energético refletem o sistema todo, não há isolamento de processos, e é realizado de maneiras diferentes a depender do processador em questão, podendo ser derivada de contadores de performance presentes na CPU, ou um modelo linear partindo da carga atual do sistema, o que será melhor discutido na seção **Abordagem**. Outra questão importante é como a instrumentação das aplicações é realizada, requisições HTTP são realizadas pela aplicação para marcar o início e final de regiões de interesse do código, assim o serviço de métricas e a aplicação são executadas na mesma máquina, e um perfil de consumo energético das regiões é obtido.

Com isto, nesta seção é apresentado como os experimentos foram realizados, desde o software desenvolvido e aplicações empregadas até o hardware e arquitetura do sistema.

4.1 Serviço de Consumo Energético

O serviço de métricas de consumo energético é uma API REST desenvolvida em Python [17] utilizando o framework FastAPI [18], e como apontado anteriormente, a comunicação com o serviço é realizada por meio de requisições HTTP. Esse modo de coleta de métricas foi preferido dada a facilidade de utilização do serviço *dockerizado*, assim basta executar o contêiner Docker e realizar as requisições. O serviço possui 4 endpoints, sendo 2 relacionados à obtenção de métricas e os outros 2 são utilizados para exportação das métricas coletadas:

- **POST** /region/start: marca o início de uma região no código e retorna um identificador único (UUID) para ser utilizado na requisição que finaliza a região, evitando conflitos caso a região possua o mesmo nome em um cenário multithread;
- **POST** /region/end: marca o final de uma região (identificada pelo UUID), e retorna o tempo de execução em segundos e a energia consumida em joules;
- **GET** /export/power: exporta um arquivo CSV com todas as amostras de potência coletadas durante a execução do serviço;
- **GET** /export/regions: exporta um arquivo CSV contendo todas as regiões perfiladas, com o consumo energético e os *timestamps* de início e fim.

O serviço é uma aplicação externa à aplicação que está sendo perfilada, assim o consumo energético é obtido a partir de informações do sistema todo, não havendo isolamento de processos nesse quesito. Além disso, como dito na seção anterior, o modo de obtenção de métricas de consumo energético varia com processador. Computadores da Intel possuem contadores de performance dos quais essa métrica pode ser derivada, enquanto que processadores ARM, como é o caso dos Raspberry Pis, não possuem suporte do hardware, então um modelo linear para a potência foi empregado. Para computadores Intel, PAPI foi utilizada, mais especificamente o componente *powercap* que expõe um valor estimado de energia consumida pelo processador. A fim de usar o PAPI na aplicação em Python, um wrapper pré-existente (PyPAPI) foi atualizado para ter suporte às operações da API de baixo nível da biblioteca. Nos Raspberry Pis, o modelo linear baseado na carga da CPU utilizado pode ser visto na Equação 1 e os valores de referência (*stressed* e *idle*) são de [19].

$$P = idle + (stressed - idle) \times CPU_{load} \tag{1}$$

Uma característica do serviço de consumo energético é que novos métodos de obtenção de métricas podem ser facilmente adicionados, basta criar uma classe herdeira de alguma das classes base, e a configuração de qual classe usar é feita por variáveis de ambiente. Assim, o serviço pode ser atualizado para suportar novos processadores de uma maneira simples.

4.2 Aplicações

Os experimentos foram realizados com 2 tipos de aplicação diferentes, multiplicação de matrizes (MatMul) e um servidor web. A multiplicação de matrizes foi implementada em Python, utilizando a biblioteca NumPy, e também uma implementação em Dana foi testada. Já para o servidor web, a aplicação com *Distributor* e *Remote Distributor* desenvolvida por Guardão e Araujo [20] foi utilizada.

4.2.1 Multiplicação de Matrizes

As aplicações de MatMul foram utilizadas para comparar os diferentes computadores em um cenário de computação intensiva, sendo que a implementação em Python faz uso de paralelismo, já que o NumPy faz uso de paralelismo por padrão e é otimizado, e a em Dana é *naive* e não faz uso de paralelismo. A aplicação consiste em um *loop* com uma fase de inicialização das estruturas de dados com valores aleatórios, e a fase de computação, onde a multiplicação é realizada. Assim, em Python matrizes 1000x1000 foram usadas, enquanto que em Dana apenas 50x50; dimensões maiores não foram utilizadas devido a limites do uso de memória e também ao tempo de processamento.

A Figura 2 mostra um esquemático dos componentes de MatMul em Dana. **Main comp** é onde o *loop* é de inicialização de dados e computação é realizado, e as chamadas para início e fim dessas duas regiões são realizadas. **EnergyService** realiza requisições HTTP para o serviço. **MatMul** e **MatMul_Default** é onde a multiplicação é realizada, o segundo componente utiliza uma implementação da biblioteca padrão de Dana. Por último, **InitData** é o componente que inicializa as matrizes com valores aleatórios.

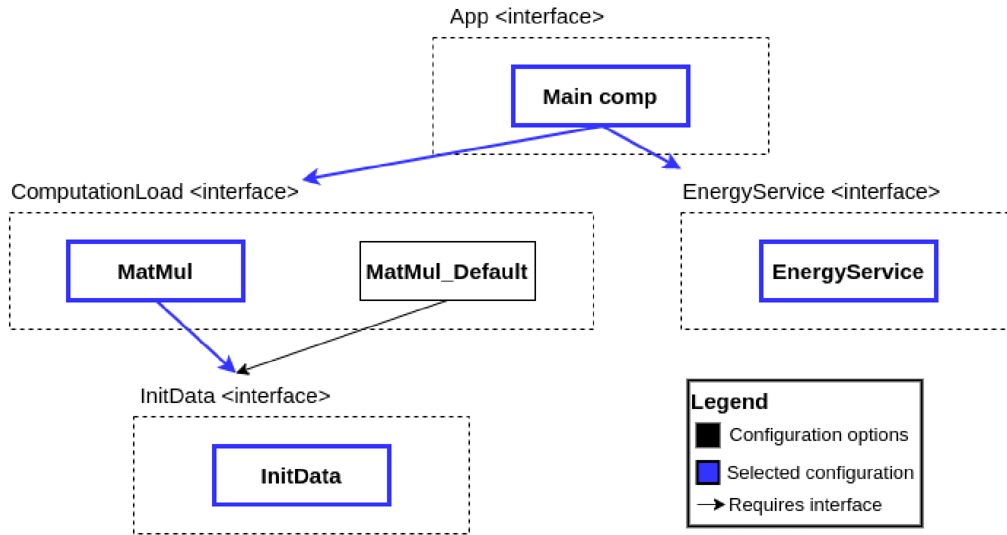


Figura 2: Componentes da Aplicação de Multiplicação de Matrizes em Dana

4.2.2 Servidor Web

O servidor web de Guardão e Araujo [20] é implementado em Dana e realiza operações em uma lista, como escrita (inserção e remoção) e leitura de elementos. O componente da lista utilizada nos experimentos possui um processamento associado a essas operações, para a escrita a complexidade é de $O(log(n))$, dado que a esrutura de dados empregada é um *heap*. Já para a leitura, os elementos são ordenados utilizando o algoritmo *MergeSort*, resultando em uma complexidade de $O(n.log(n))$.

A aplicação analisada consiste em um servidor (*Distributor*) e duas listas remotas (*Remote Distributors*) em uma mesma rede local, mas em máquinas distintas. Assim o *Distributor* se comunica com o *Remote Distributor* para realizar as operações na lista remota; e duas configurações de lista são executadas, uma na qual a lista é replicada resultando em duas listas remotas iguais(*propagate*) e a outra faz uso de *sharding*, assim a lista é fragmentada e cada lista remota possui apenas parte dos elementos da lista.

As regiões de interesse perfiladas no *Remote Distributor* foram as chamadas das funções de inserção (*add_item*), remoção (*remove_item*) e leitura (*get_items*).

4.3 Arquitetura do Sistema

O *cluster* utilizado para execução dos experimentos é formado por um *cluster* provisionado na nuvem usando a *Google Kubernetes Engine* (GKE) e um conjunto de computadores na borda em uma mesma rede local. O gerenciamento dos nós da borda foi realizado utilizando o KubeEdge, que estende as funcionalidades do Kubernetes para a borda, e é composto pelo *CloudCore* e *EdgeCore*. Um esquemático da arquitetura do sistema pode ser visto na Figura 3.

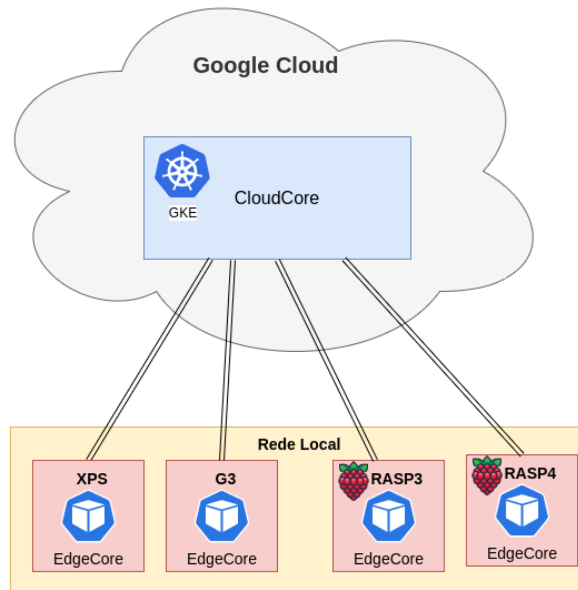


Figura 3: Arquitetura do Sistema.

As Tabelas 1 e 2 apresentam as especificações dos computadores utilizados e os softwares para orquestração de contêineres.

Computador	Processador	RAM	SO
Dell XPS	i7-1185G7	16GB LPDDR4X	Ubuntu Desktop 20.04.5 LTS 64-bit
Dell G3	i5-9300H	16GB DDR4 2666MHz	Ubuntu Desktop 22.04.1 LTS 64-bit
Raspberry Pi 3 Model B+	Broadcom BCM2837B0 Quad-core Cortex-A53	1GB LPDDR2	Ubuntu Server 20.04.5 LTS 32-bit
Raspberry Pi 4 Model B	Broadcom BCM2711 Quad-core Cortex-A72	8GB LPDDR4-3200	Ubuntu Server 20.04.5 LTS 32-bit

Tabela 1: Especificações dos computadores utilizados na borda.

Software	Versão	Local
Kubernetes	1.22.15-gke.100	Nuvem (GKE)
KubeEdge	v1.12.0	Borda (nós)

Tabela 2: Softwares para orquestração de contêineres.

A execução dos experimentos é realizada por meio de *Deployments* do Kubernetes, sendo que o nó onde o *pod* será instanciado é selecionado dentre os nós da borda por meio de rótulos. A Figura 4 apresenta o *pod* que é implantado. Nele há 2 contêineres Docker, sendo um da aplicação a ser perfilada e o outro do serviço de consumo energético. Como os contêineres dentro de um mesmo *pod* compartilham a interface de rede *loopback*, a comunicação é facilitada. Um cliente externo ainda pode acessar a API do serviço para obter os CSVs gerados na execução do experimento.

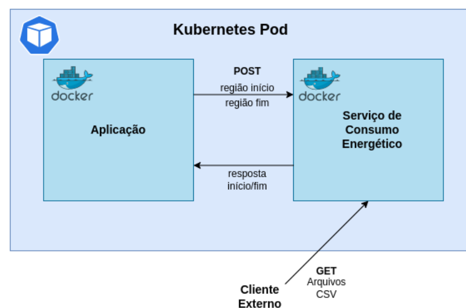


Figura 4: Deployment do Pod.

5 Resultados

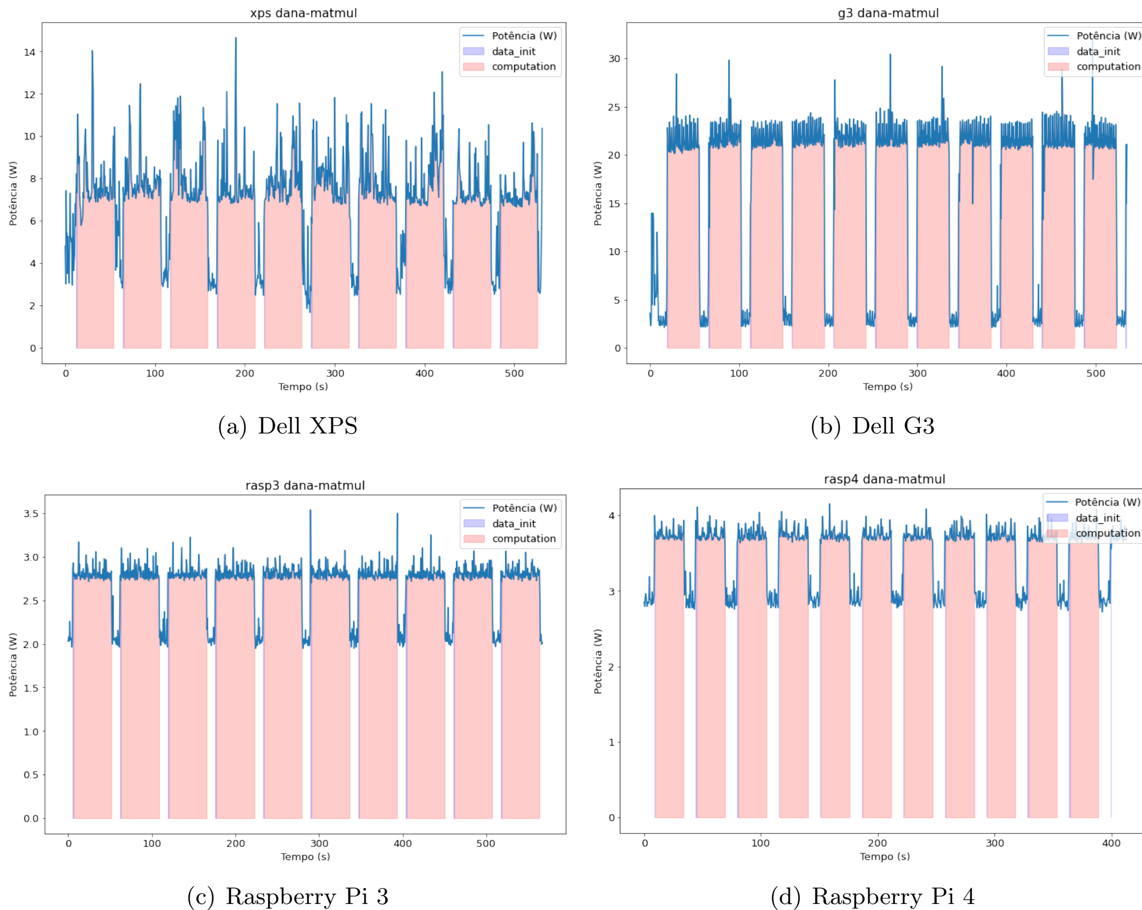
Nesta seção são apresentados os resultados dos experimentos com as aplicações de multiplicação de matrizes em Python e Dana e o servidor web em Dana. Todos os experimentos utilizam a linguagem Dana dado que ela é utilizada para implementação do conceito de sistemas auto-distribuídos, mas ainda se encontra em estágios experimentais. Assim, a linguagem Python também é utilizada já que é bem estabelecida na indústria e apresenta um bom desempenho para operações com matrizes, por usar uma biblioteca BLAS (*Basic Linear Algebra Subprograms*) [21] , servindo de contraste para a aplicação em Dana.

5.1 Métricas de Consumo Energético e Tempo de Execução

Os gráficos apresentados nesta subseção foram gerados a partir das métricas coletadas pelo serviço de consumo energético.

5.1.1 Multiplicação de Matrizes

Os gráficos das Figuras 5 e 6 mostram o perfil de Potência x Tempo da execução das aplicações de multiplicação de matrizes em Dana e Python, respectivamente, nos quatro computadores listados na tabela 1. A linha azul é a potência em Watts, a área hachurada em azul é a região de inicialização dos valores da matriz e a em vermelho a computação da multiplicação, como a inicialização dos dados é extremamente rápida se comparada com a multiplicação, é difícil de notá-la nos gráficos. Com esses gráficos, em especial o da Figura 5, é possível verificar que os níveis de potência empregados pelos processadores quando à uma carga de trabalho computacionalmente intensiva são bem diferentes. Os Raspberry Pis ((c), (d)) ficam em uma faixa de 3 - 4 W, o que é esperado dado que são dispositivos **low power**. Os computadores da Dell apresentaram níveis próximos de 10 W para o XPS e 25W para o G3, o que pode ser justificado pelo fato do processador do XPS ser para notebooks compactos e o do G3 ser próximo de um processador de desktop.



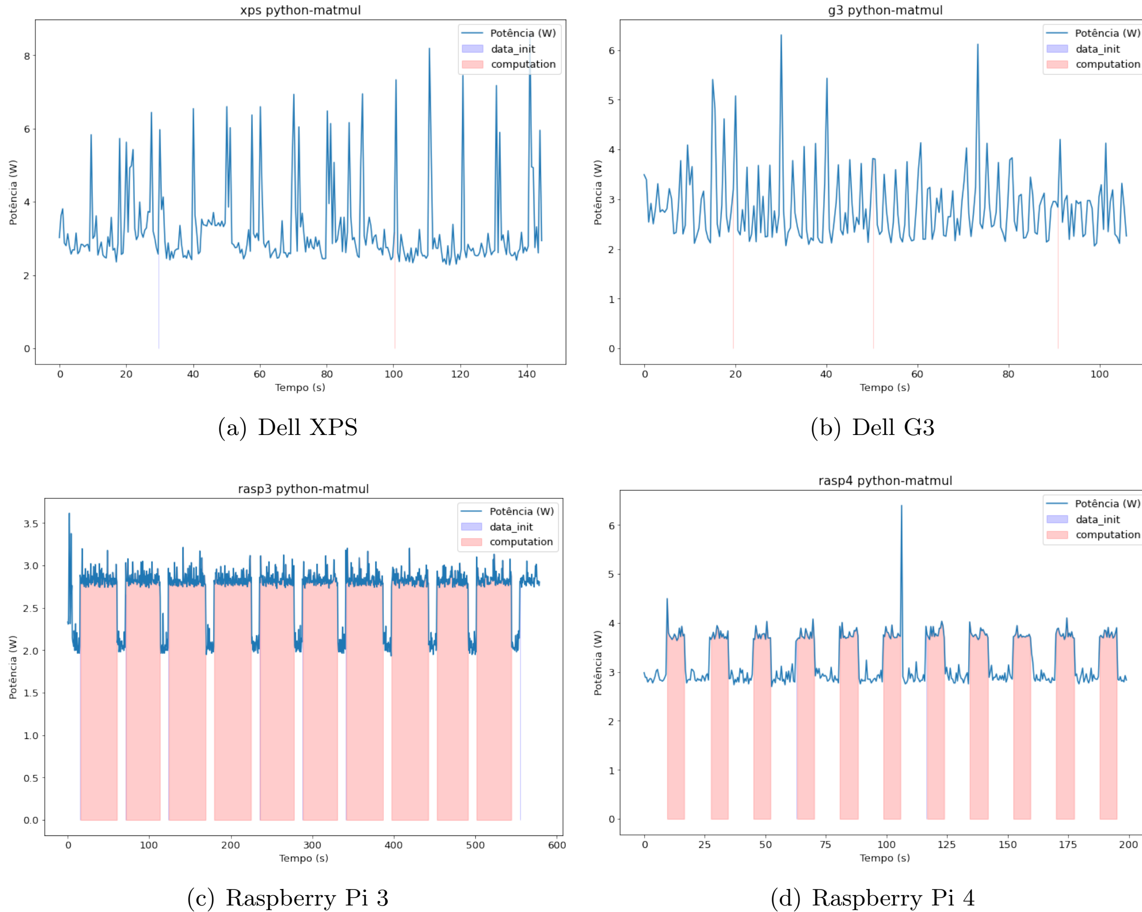
(a) Dell XPS (b) Dell G3

(c) Raspberry Pi 3 (d) Raspberry Pi 4

Figura 5: Multiplicação de Matrizes em Dana - matrizes 50 x 50.

A execução da multiplicação de matrizes usando o NumPy foi extremamente rápida nos

computadores da Dell, tanto que não é possível visualizar as áreas vermelhas em 6-(a) e 6-(b). Esse comportamento pode ser justificado pelo melhor aproveitamento de paralelismo por parte da biblioteca nesses processadores.



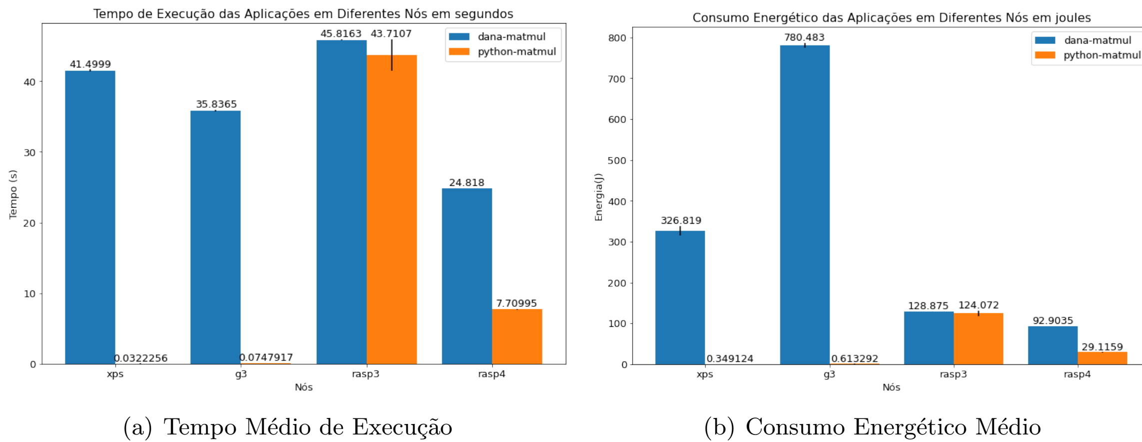
(a) Dell XPS (b) Dell G3

(c) Raspberry Pi 3 (d) Raspberry Pi 4

Figura 6: Multiplicação de Matrizes com NumPy - matrizes 1000 x 1000.

As métricas coletadas são sumarizadas nos gráficos de barras da Figura 7. Neles é possível verificar que o Dell XPS e o Dell G3 apresentaram um tempo de execução e energia consumida bem menor que os Raspberries para MatMul em Python (barras laranjas), o que já era esperado. Porém, a aplicação em Dana (barras azuis) demonstrou um resultado inesperado, o Raspberry Pi 4 obteve tanto o melhor tempo de execução quanto o menor consumo energético; o motivo desse resultado não é investigado no escopo deste projeto, mas uma análise mais aprofundada se faz necessária para possíveis otimizações no *runtime* de Dana para arquiteturas x86. Outra questão interessante, é o fato do G3 ter um tempo de execução quase 15% menor do que o XPS para MatMul em Dana, só que consome quase 2,4x mais energia, assim uma comparação de tempo de execução de uma aplicação em plataformas distintas não reflete necessariamente o consumo energético. Além disso, nota-

se que os dispositivos low power operam a um nível de consumo de potência menor que os demais, mas isso não implica que o consumo de energia será menor, já que eles geralmente são mais lentos. Assim, dependendo da aplicação podem consumir mais energia, como é o caso de MatMul em Python, onde a execução demora bem mais nos Raspberries, o que resulta em um consumo energético maior quando comparado com os computadores com processador da Intel, que são mais rápidos.



(a) Tempo Médio de Execução

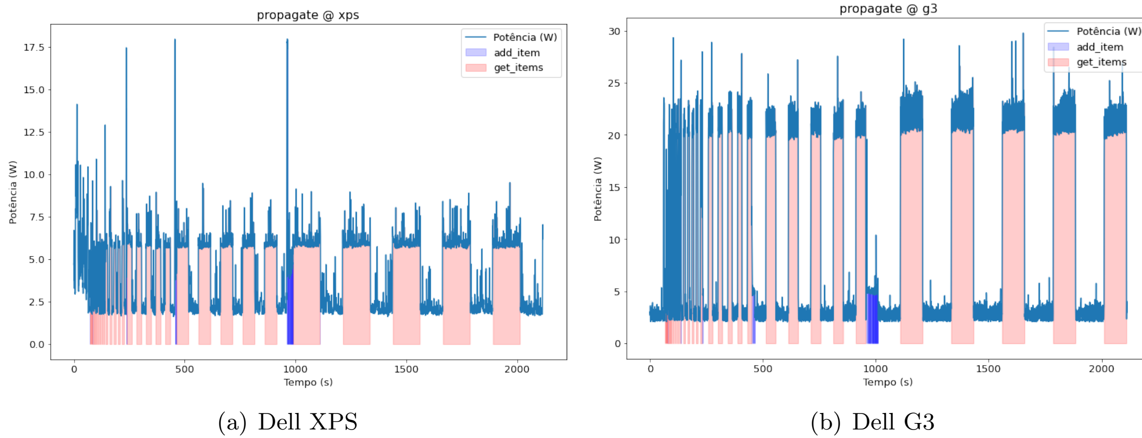
(b) Consumo Energético Médio

Figura 7: Multiplicação de Matrizes em Nós Diferentes.

Partindo dos resultados da Figura 7, os melhores nós para execução da aplicação seriam o Raspberry Pi 4 para a implementação em Dana e o Dell XPS para a que usa NumPy. Assim, com esse *placement* o sistema estaria em uma configuração tanto com o melhor tempo de execução quanto com o menor consumo de energia.

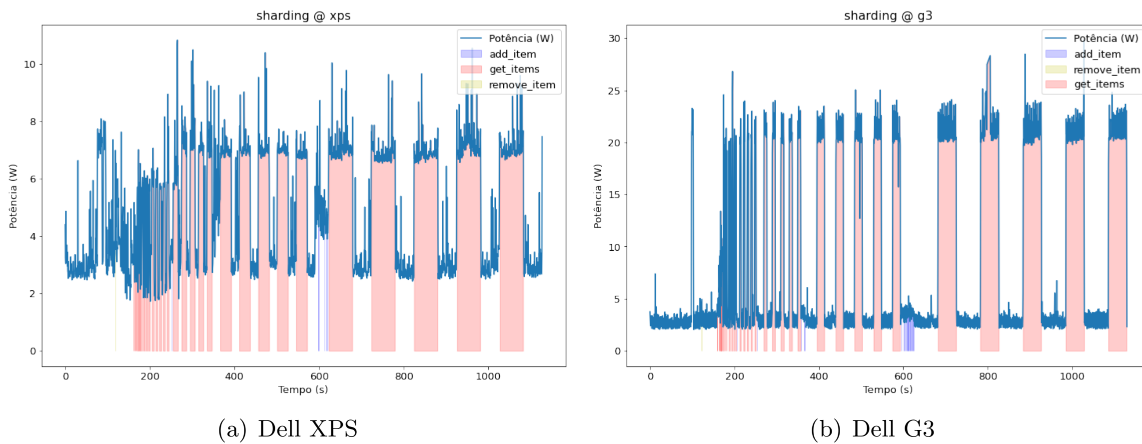
5.1.2 Servidor Web

O servidor web só foi testado no Dell XPS e Dell G3, devido ao fato de que o *runtime* Dana para Raspberry não suporta a execução da aplicação em questão, apresentando falhas ao executar o código. Assim como nos gráficos de Potência x Tempo apresentados anteriormente, os gráficos das Figuras 8 e 9 tem a linha azul como a potência em Watts, as áreas em azul neste caso são as regiões onde uma elemento é inserido na lista, em vermelho é a operação de leitura e amarelo, para 9 marca a remoção de itens da lista.



(a) Dell XPS (b) Dell G3

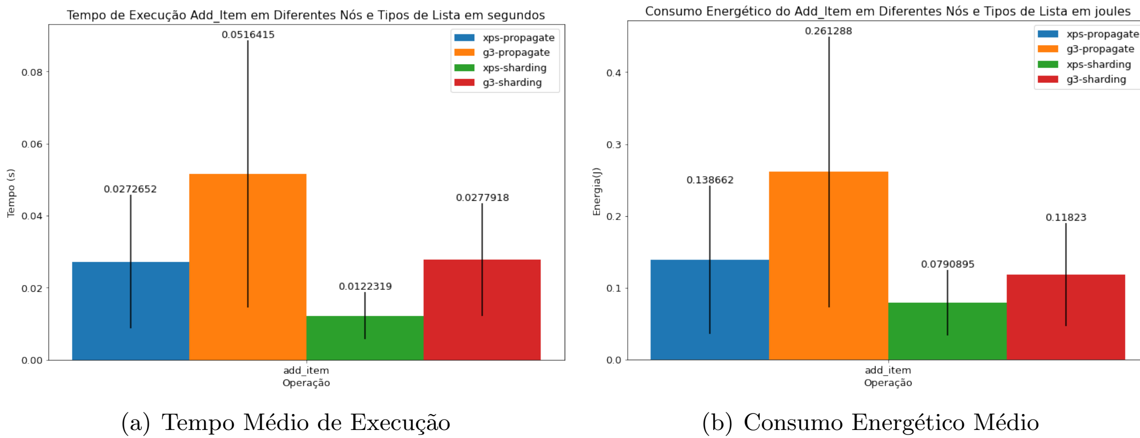
Figura 8: *Remote Distributor* com componente *propagate List*.



(a) Dell XPS (b) Dell G3

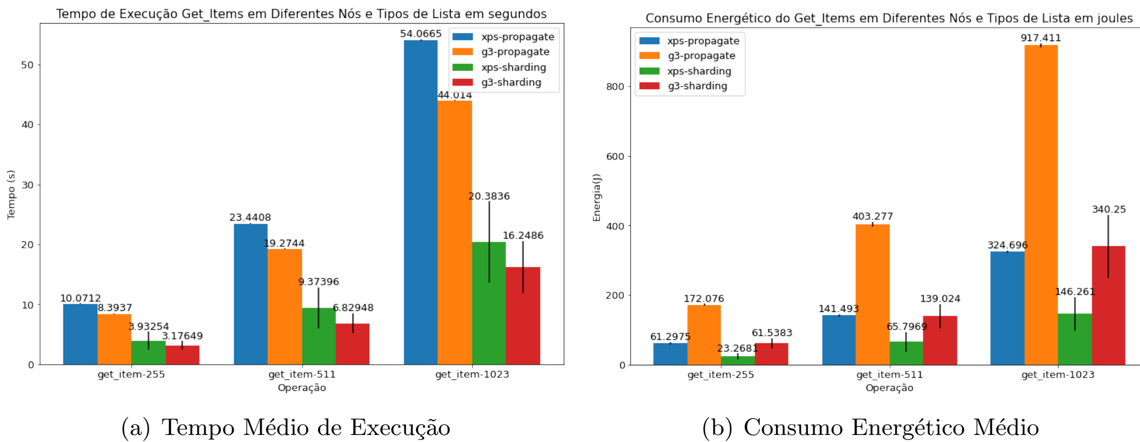
Figura 9: *Remote Distributor* com componente *sharding List*.

Os gráficos de barras das Figuras 10 e 11 apresentam as métricas (tempo de execução e consumo energético) para as operações de escrita (*add_item*) e leitura (*get_items* com 255, 511 e 1023 elementos na lista), respectivamente, da execução do *Remote Distributor* com as configurações de lista *propagate* e *sharding*. O tempo médio de execução é menor para a configuração de *sharding* em todas os cenários quando comparado com *propagate*, o que era esperado. Comparando os resultados de cada computador (XPS - azul e verde, G3 - laranja e vermelho), é possível verificar que para a operação de inserção o XPS apresenta melhor tempo de execução e consumo de energia que o G3; entretanto, para leitura, por possui a ordenação, apresenta um comportamento semelhante a multiplicação de matrizes, onde o G3 realiza a operação mais rápido, mas o consumo energético é bem maior que o XPS.



(a) Tempo Médio de Execução (b) Consumo Energético Médio

Figura 10: Métricas da Operação de Escrita da Lista em diferentes configurações.



(a) Tempo Médio de Execução (b) Consumo Energético Médio

Figura 11: Métricas da Operação de Leitura da Lista em diferentes configurações..

O *placement* da lista e qual configuração utilizar está fortemente relacionado aos requisitos do sistema. O estudo realizado por Oswaldo [22] que para uma sistema que possui restrições quanto a consistência dos dados, *sharding* não é uma opção ideal, mesmo que apresente um tempo de resposta menor. A escolha em que nó executar as listas remotas depende do padrão de acesso, caso operações de inserção de itens predominem, o XPS seria a melhor opção dado que possui tempo de execução e consumo de energia menor que o G3; em cenários com mais leituras, o G3 pode ser escolhido se tempo de resposta baixo for um requisito, mas priorizando a eficiência energética, o XPS seria o nó preferido.

5.2 Utilização do KubeEdge

O KubeEdge se mostrou boa plataforma para execução de experimentos na borda de maneira automatizada, dado que estende as funcionalidades de gerenciamento do Kubernetes

para os nós da borda. Assim, scripts que utilizam um cliente do Kubernetes podem realizar a implantação de serviços, ou alterar alguma configuração de um serviço já existente para testar diferentes configurações. A integração do KubeEdge com aplicações como o *ServerCTL* [23] pode ser interessante, especialmente para cenários na qual a utilização da borda é necessária, sendo assim um possível trabalho futuro.

6 Conclusões e Trabalhos Futuros

Neste projeto foi apresentado um serviço para coleta de métricas de desempenho e consumo energético a nível do sistema (computador), com o qual é possível obter um perfil energético de uma aplicação executando na mesma máquina. A aplicação realiza requisições HTTP para marcar o início e fim de regiões de interesse no código, sendo que para obtenção dos dados de energia, modelos lineares e contadores de performance do processador são utilizados. Assim, ao final de uma região obtém-se o tempo e energia gastos nela. Esse tipo estratégia para coleta de métricas se mostrou simples e fácil, de modo que só é necessário adicionar requisições nos pontos de interesse do código, não precisando *linkar* o código da aplicação com a biblioteca de perfilamento, basta executar um contêiner Docker utilizando uma imagem já construída, disponibilizada em um repositório do **Apêndice B**. Um possível trabalho futuro está na melhora do serviço de coleta de métricas, para ter suporte para uma gama maior de plataformas, além de coletar outras métricas que podem prover um perfil melhor da aplicação e.g. uso de rede.

Com os resultados obtidos de duas aplicações (MatMul e Servidor Web) através do serviço de consumo energético, foi possível realizar uma análise de como cada aplicação se comporta nos nós de um *cluster* heterogêneo. Verificou-se que o tempo de execução não é o suficiente para determinar onde a aplicação deve ser executada para chegar em uma configuração eficiente, dado o tempo nem sempre reflete o consumo de energia em hardwares distintos. Porém, um modelo sofisticado para *placement* de serviços utilizando as métricas do serviço de consumo energético não foi proposto, o que é uma linha na qual esse projeto pode prosseguir. Ainda nessa linha, como resultados inesperados podem ocorrer (e.g. MatMul em Dana), abordagens que realizam experimentações em tempo de execução antes da decisão do *placement* são preferidas. Dentre elas, a utilização de algoritmos de aprendizado de máquina por reforço que testam *placements* antes de convergir para a uma configuração podem ser interessante, já que várias variáveis podem interferir no gasto energético e desempenho, podendo ser difícil de antecipá-las, principalmente em ambientes operacionais voláteis.

Outra questão explorada foi o uso do KubeEdge, que facilita a execução de experimentos na borda, e uma futura integração com o *ServerCTL* [23], desenvolvido por Oliveira e Koaro, pode ser uma boa adição para o ambiente de sistemas auto-distribuídos.

Os repositórios de código e imagens Docker utilizados nos experimentos estão listados nos **Apêndices A** e **B**, respectivamente.

Referências

[1] Jeffrey O Kephart and David M Chess. *The vision of autonomic computing. Computer*, 36(1):41–50, 2003.

[2] R. Rodrigues Filho. *Emergent Software Systems.* PhD thesis, Lancaster University, 2018.

[3] **Linguagem Dana**. https://www.projectdana.com/.

[4] Roberto Rodrigues-Filho and Barry Porter. Hatch: Self-distributing systems for data centers. *Future Generation Computer Systems*, 132:80–92, 2022.

[5] Patrick Kurp. Green computing. *Communications of the ACM*, 51(10):11–13, 2008.

[6] **PAPI**. https://bitbucket.org/icl/papi/wiki/Home.

[7] Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. *A comparative study of methods for measurement of energy of computing. Energies*, 12(11):2204, 2019.

[8] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. *Measuring energy and power with PAPI.* In *2012 41st international conference on parallel processing workshops*, pages 262–268. IEEE, 2012.

[9] Fabian Kaup, Philip Gottschling, and David Hausheer. *PowerPi: Measuring and modeling the power consumption of the Raspberry Pi.* In *39th Annual IEEE Conference on Local Computer Networks*, pages 236–243. IEEE, 2014.

[10] M. van Steen and A. Tanenbaum. *Distributed Systems.* 3rd edition, 2017.

[11] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. *An overview on edge computing research. IEEE access*, 8:85714–85728, 2020.

[12] Hany F Atlam, Robert J Walters, and Gary B Wills. *Fog computing and the internet of things: A review. big data and cognitive computing*, 2(2):10, 2018.

[13] James E Smith and Ravi Nair. *The architecture of virtual machines. Computer*, 38(5):32–38, 2005.

[14] **Docker**. https://docs.docker.com/.

[15] **Kubernetes**. https://kubernetes.io/pt-br/docs/home/.

[16] **KubeEdge**. https://kubeedge.io/en/docs/.

[17] **Linguagem Python**. https://www.python.org/.

[18] **FastAPI framework**. https://fastapi.tiangolo.com/.

[19] **Valores de Referência - Raspberry Pi**. http://www.pidramble.com/wiki/benchmarks/power-consumption.

[20] F. A. L. Guardão, B. B. Araujo, L. F. Bittencourt, and R. R. Filho. *Explorando o Desempenho de Sistemas Auto-distribuídos na Borda e Nuvem.* https://www.ic.unicamp.br/~reltech/PFG/2022/PFG-22-08.pdf, 2022.

[21] Robert van de Geijn and Kazushige Goto. *BLAS (Basic Linear Algebra Subprograms)*, pages 157–164. Springer US, Boston, MA, 2011.

[22] G. H. R. Oswaldo, L. F. Bittencourt, and R. R. Filho. *Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso.* https://www.ic.unicamp.br/~reltech/PFG/2021/PFG-21-50.pdf, 2021.

[23] A. P. Oliveira, R. H. Koaro, L. F. Bittencourt, and R. R. Filho. *Um Estudo sobre Sistemas Auto-distributivos em Ambientes Elásticos.* https://www.ic.unicamp.br/~reltech/PFG/2022/PFG-22-05.pdf, 2022.

Apêndice

A Repositórios de Código

- **Configuração de Infraestrutura** - `https://github.com/SerodioJ/pfg.git`
- **Serviço de Consumo Energético** - `https://github.com/SerodioJ/energy-consumption-service`
- **PyPAPI** - `https://github.com/SerodioJ/pypapi/tree/v6.0.0.1`
- **Execução de Experimentos** - `https://github.com/SerodioJ/energy-consumption-eval`
- **PAPI** - `https://bitbucket.org/SerodioJ/papi/src/pfg/`

B Repositórios de Imagens Docker

- **Serviço de Consumo Energético** - `https://hub.docker.com/repository/docker/serodioj/energy-consumption-service`
- **PyPAPI** - `https://hub.docker.com/repository/docker/serodioj/pypapi`
- **MatMul** - `https://hub.docker.com/repository/docker/serodioj/ecs-proxy`
- **Dana** - `https://hub.docker.com/repository/docker/serodioj/dana`
- **Servidor Web** - `https://hub.docker.com/repository/docker/serodioj/dana_app`