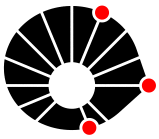

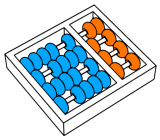

Distribuição *on demand* de replicas para softwares emergentes

I. F. Mandello *L. F. Bittencourt* *R. R. Filho*

Relatório Técnico - IC-PFG-24-02
Projeto Final de Graduação
2024 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Distribuição de replicas *on demand* para softwares emergentes

Igor Fernando Mandello * Luiz Fernando Bittencourt *
Roberto Rodrigues Filho †

15 de julho de 2024

Resumo

Este projeto visa desenvolver uma metodologia para a dinamização da distribuição remota de componentes no contexto de sistemas auto-distribuídos. A abordagem escolhida busca otimizar a escalabilidade e adaptabilidade dos sistemas, aproveitando a infraestrutura dos sistemas gerenciadores de *containers* para manter executando apenas o que seja estritamente necessário. Com isso, é possível observar como uma aplicação pode fazer o uso dos ajustes em *runtime* para melhorar seu tempo de execução, enquanto mantém os custos sob controle.

1 Introdução

Sistemas modernos possuem diversas nuances que tornam sua adaptabilidade muito mais complexa. Vemos um crescente aumento nas arquiteturas de micro-serviços [2] e *Serverless* [1] como auxílio e solução a esse problema. No entanto, essas abordagens acabam forçando a aplicação a ser escrita de uma maneira bem específica pelas equipe, criando barreiras para a transição em aplicações existentes que sejam monolíticas ou *stateful*. [11]

Além das estratégias convencionais, destaca-se o conceito de Sistemas de Software Emergentes [10], que, através de uma linguagem baseada em componentes, possibilita a troca autônoma de componentes em tempo de execução. Ao aplicar esse conceito em tecnologias como computação em nuvem, computação em névoa e computação em borda, é possível desenvolver um *framework* para a exploração do espaço de combinações disponível e subsequente uso da combinação que é mais adaptada para aquele contexto através de técnicas de *Reinforcement Learning*. [9]

Além das preocupações com performance, escalabilidade e adaptabilidade dos sistemas, também podemos acrescentar necessidades financeiras dos usuários que visam explorar essas estratégias. É notável o crescimento de setores especializados em *Cloud Economics* dentro de empresas, visando a otimização de custos dos recursos utilizados e a identificação e deleção de recursos parados. [12]

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP
†Departamento de Computação, Universidade Federal de Santa Catarina, 88900-000 Araranguá, SC

Portanto, o objetivo principal é explorar os desafios e impactos desse método de distribuição de sistemas e dinamizar a troca de configuração dos componentes, considerando a necessidade de realizar essas mudanças em tempo de execução e evitar a criação e persistência de recursos que não estão sendo utilizados. Isso desbloqueia o uso de métricas de custo ao buscar pela melhor configuração, possibilitando o emprego de ajustes de acordo com sua necessidade (melhor performance, equilibrado, melhor custo, etc.). Além disso, este estudo avalia o impacto das trocas de configuração na performance do sistema, comparando com cenários anteriores e destacando os aspectos positivos e negativos.

2 Referencial Teórico

2.1 Sistemas de Software Emergentes

Um Sistema de Software Emergente é um sistema construído com pequenos componentes reutilizáveis que permitem auto-composição e auto-otimização em tempo de execução. Segundo Filho [10], esse tipo de sistema utiliza o *framework* PAL (*Perception*, *Assembly*, *Learning*), composto por três módulos principais:

1. **Assembly**: Responsável por compor e gerenciar a estrutura do sistema, supervisionando a interação entre os componentes e permitindo modificação, adição ou remoção conforme necessário. Ele facilita a descrição e execução de composições de software através de suas funcionalidades.

2. **Perception**: Monitora o sistema e o ambiente usando o módulo *Assembly*, oferecendo funções via REST API para facilitar o acesso a processos de aprendizado e descrevendo dados de percepção coletados no ambiente de execução.

3. **Learning**: Compreende as relações entre os componentes e o desempenho do sistema, controlando o aprendizado ativo com base em objetivos pré-estabelecidos, explorando e ajustando composições para diferentes condições.

Esses três módulos podem ser vistos na arquitetura presente na Figura 1.

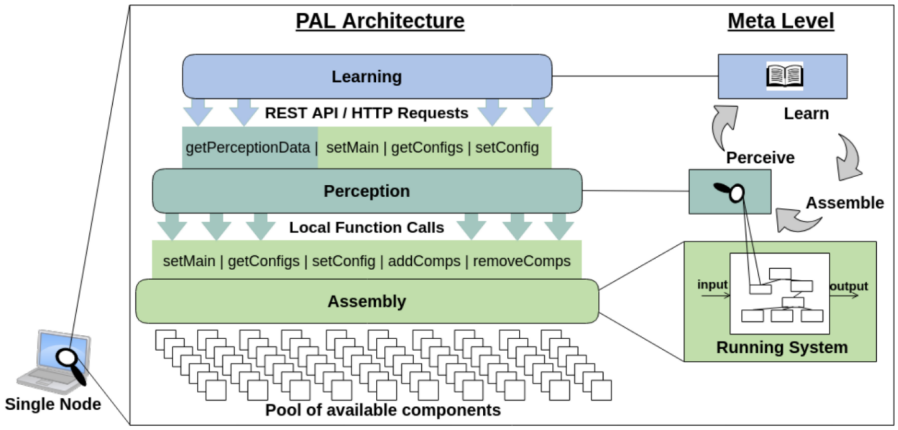

Figura 1: Arquitetura PAL

A complexidade e o dinamismo dos sistemas computacionais modernos têm impulsionado o estudo de sistemas auto-adaptativos. No entanto, abordagens tradicionais possuem limitações, como a dependência humana no projeto, incapacidade de evoluir autonomamente e mecanismos de adaptação limitados a partes específicas do sistema.

Os Sistemas de Software Emergentes foram propostos para superar essas limitações. Utilizando a linguagem Dana [8], que fornece um modelo baseado em componentes com suporte à adaptação em tempo de execução, o módulo *Assembly* controla o processo de adaptação dos sistemas em *runtime*, permitindo uma adaptação contínua e eficiente do sistema.

2.2 Kubernetes

Kubernetes [4] é uma plataforma *open-source* para automação da implantação, escalonamento e gerenciamento de aplicações em *containers*. *Kubernetes* permite que os desenvolvedores configurem, implementem e gerenciem *clusters* de *containers* de maneira eficiente. Suas funcionalidades incluem balanceamento de carga, auto-reparo, escalonamento automático, e orquestração de *containers* [13]. Utilizando um sistema declarativo baseado em YAML ou JSON, *Kubernetes* garante que os aplicativos rodem em um estado desejado. É amplamente utilizado para implementar aplicações escaláveis, resilientes e portáveis em ambientes de nuvem híbrida ou multi-nuvem. A Figura 2 contém uma ilustração reduzida da um *cluster*.

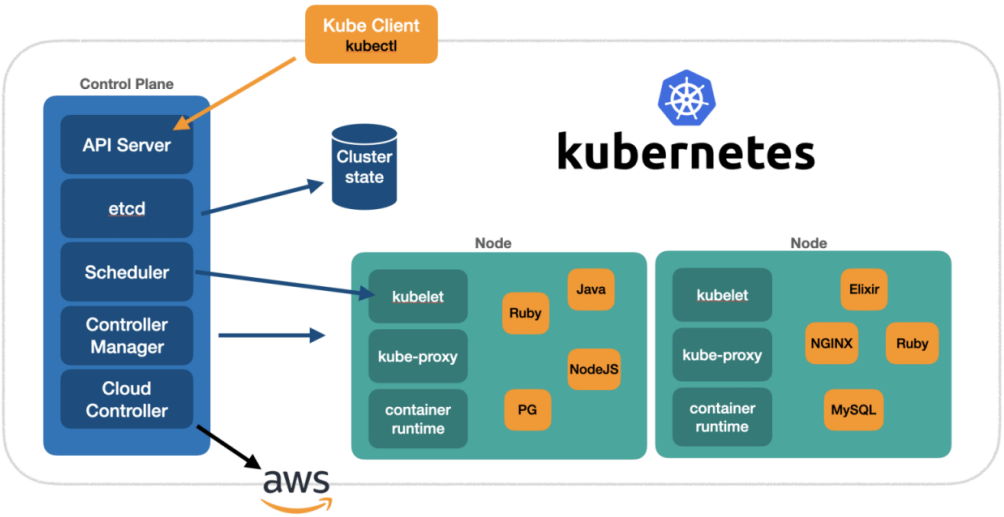

Figura 2: Ilustração simplificada de um *cluster Kubernetes*. Imagem extraída de [5].

2.3 Distribuidor Remoto

Para gerenciar o estado e as aplicações dos sistemas auto-distribuídos, foi proposto um modelo de distribuidor remoto (`Remote Dist`), que consiste em um servidor ativo que aguarda por requisições do cliente para iniciar um processo de adaptação. [7]

Quando surge a necessidade de uma adaptação, o cliente é responsável por fazer a chamada requisitando a implementação desejada e o estado atual. O `Remote Dist` então faz a troca de sua implementação local para suportar a desejada pelo cliente. Após esse processo, o cliente pode fazer as chamadas através de um *proxy*, como pode ser visto na Figura 3

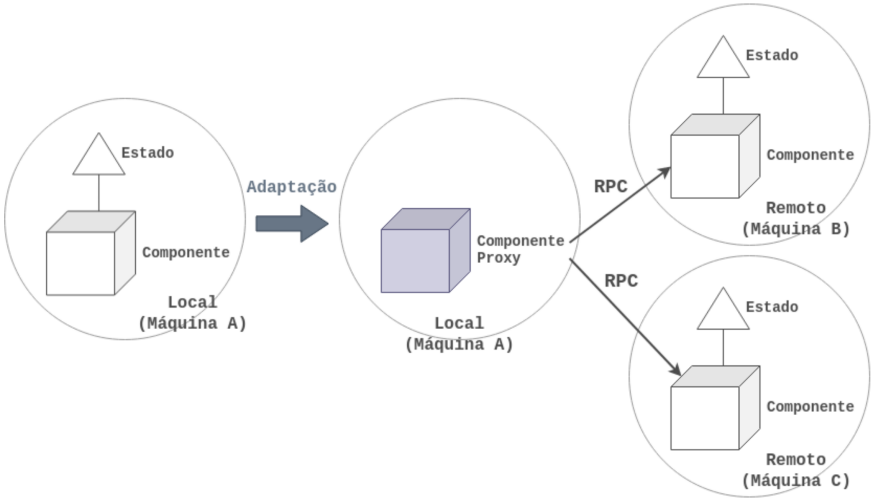

Figura 3: Processo de adaptação e distribuição com gestão de estado. O componente local é adaptado para um *proxy*, e possui seu estado extraído para os componentes remotos. O *proxy* realiza chamadas de procedimento remoto (RPCs) para o componente remoto. Imagem extraída de [7]

3 Metodologia

3.1 Simulação Local

Esse trabalho é focado na distribuição dos componentes para um ambiente remoto, portanto, foi utilizada a ferramenta **Kind** [3] para a simulação de um *cluster* **Kubernetes** localmente. Isso possibilitou uma iteração mais rápida e testes sem custos extras.

Para que a abstração seja coerente com uma *cloud* real, foi utilizado também um **Cloud Provider**, que é responsável por disponibilizar um IP externo para o *Ingress* da aplicação. Na nuvem, esse IP geralmente está associado a um *Load Balancer* e é utilizado para que clientes possam realizar chamadas para o servidor criado. Nesse caso, o *provider* simplesmente disponibiliza um IP local que aponta para o Serviço do Kubernetes vinculado ao

servidor.

Um exemplo de como essas duas tecnologias se integram pode ser visto na Figura 4. Note que cada serviço possui um *External IP* configurado, da mesma forma que estaria caso a aplicação estivesse rodando em nuvem e esse IP fosse público.

```
NAME                        TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
remote-dist-0-dbe58fbda464  LoadBalancer   10.96.243.211   172.18.0.6    2011:30821/TCP,2010:30611/TCP
remote-dist-1-dbe58fbda464  LoadBalancer   10.96.136.60    172.18.0.4    2011:31560/TCP,2010:31469/TCP
remote-dist-2-dbe58fbda464  LoadBalancer   10.96.233.207   172.18.0.5    2011:30294/TCP,2010:32255/TCP
remote-dist-3-dbe58fbda464  LoadBalancer   10.96.132.82    172.18.0.7    2011:30979/TCP,2010:32504/TCP
remote-dist-4-dbe58fbda464  LoadBalancer   10.96.5.180     172.18.0.8    2011:32446/TCP,2010:32354/TCP
remote-dist-5-dbe58fbda464  LoadBalancer   10.96.220.42    172.18.0.9    2011:30188/TCP,2010:30985/TCP
remote-dist-6-dbe58fbda464  LoadBalancer   10.96.193.123   172.18.0.10   2011:31363/TCP,2010:31944/TCP
remote-dist-7-dbe58fbda464  LoadBalancer   10.96.251.142   172.18.0.11   2011:30952/TCP,2010:30980/TCP
```

Figura 4: Serviços na aplicação Kubernetes rodando localmente através de Kind com um Cloud Provider.

3.2 Dinamização do Remote Distributor

No formato atual, o cliente necessita saber em tempo de compilação todos os *IP*s disponíveis de `Remote Dists`. Isso impossibilita que o sistema seja transparentemente escalável e dificulta mudanças caso a aplicação esteja rodando na máquina de um usuário final. Nessa mesma linha, podemos imaginar possíveis conflitos num cenário multi-usuário, onde diversos sistemas rodando localmente fazem a requisição para um mesmo `Remote Dist`, causando um problema de concorrência durante o tempo de adaptação.

Outro problema dessa abordagem é a necessidade de manter o servidor rodando e esperando por chamadas a todo momento, que invalida a adição de uma métrica de custo no algoritmo de *Reinforcement Learning* que decide o momento de *triggar* uma adaptação, já que independente da configuração escolhida, o custo de `compute` (isto é, apenas máquinas, sem considerar *networking* ou usos de sistemas em *cloud*, como *File Systems* ou Banco de Dados) seria próximo ou igual.

A solução escolhida para esse problema foi implementar um serviço intermediário - chamado `Implementation Provider` - que é responsável pela criação e gerenciamento dos `Remote Dists`. Mais detalhes sobre a implementação do serviço estão na seção 3.3.

Para que seja possível a criação dinâmica do Distribuidores, foi necessário também desenvolver um método para que não seja necessário realizar a chamada que define qual implementação será utilizada pelo Distribuidor. Isso foi feito através de duas variáveis de ambiente - `IMPLEMENTATION` e `STATE` - que representam as mesmas variáveis que são enviadas no pedido de distribuição, e são lidas no mesmo que o Distribuidor inicia.

Também foi implementado a adição de um arquivo em `/tmp/ready` ao final da distribuição (e a sua deleção no início), isso será útil para saber o Distribuidor está pronto para servir a implementação desejada.

3.3 Implementation Provider

Como dito anteriormente, foi criado um servidor para intermediar a interação com o *cluster* Kubernetes. Esse servidor deve ter um DNS fixo e deve ser a única informação que o cliente deverá saber no tempo de compilação.

Foram implementadas duas rotas: `POST /distribute` e `POST /destroy`, que recebem os seguintes objetos:

```
# Distribute
{
    "implementation": "implementation-name",
    "replicas": 123,
    "metadata": [
        {
            "key": "value"
        }
    ]
}

# Destroy
{
    "implementation": "implementation-name"
}
```

A requisição de `distribute` utiliza um *template* de YAML com a ferramenta `Jinja` para a criação do YAML final, que é enviado para a *master* do Kubernetes. Com isso, é criado um `Pod` para cada replica requisitada pelo cliente, assim como um serviço que aponta para o `Pod` criado. O *template* da seção de *metadata* do `Pod` pode ser visto a seguir:

```
apiVersion: v1
kind: Pod
metadata:
  name: remote-dist-{{ replica }}-{{ uuid }}
  labels:
    app: remote-dist
    replica: "{{ replica }}"
    implementation: "{{ implementation }}"
    id: "{{ uuid }}"
    app.kubernetes.io/managed-by: implementation-provider
```

Nas *labels* do `Pod` podem ser vistas duas outras propriedades: `UUID` e a label de `managed-by`. A *label* de `UUID` tem apenas o objetivo de deduplicar o nome do `Pod`, evitando erros que podem ocorrer caso uma implementação esteja sendo distribuída enquanto outra versão dela ainda não foi destruída. No caso da `managed-by`, temos apenas uma indicação a título de governança de recursos, indicando para usuários que o serviço `implementation-provider` controla aquele `Pod`, e sendo útil caso o serviço necessite fazer um *wipe* - onde só é necessário executar um *delete* com um *label selector*.

Seguindo para o *container* do Distribuidor Remoto, foi criado um `readinessProbe` utilizando o arquivo `/tmp/ready` mencionado anteriormente. Também foi criado um mapa de `env` dinâmico, que, além de adicionar a variável de implementação, também repassa todos os campos enviados no objeto de `metadata` daquela replica. Em especial, é por meio desse objeto que passamos a variável de estado.

```
spec:
  containers:
  - #...
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/ready
      initialDelaySeconds: 5
      periodSeconds: 5
    env:
{{ env | to_yaml | indent(6, true) }}
```

A título de exemplo, se estivermos distribuindo uma lista remota e queremos que um determinado `shard` seja iniciado com os items 1, 9 e 17, ao fazer a requisição de distribuição para o `Implementation Provider`, as variáveis de ambientes resolvidas pelo servidor seriam:

```
env:
    IMPLEMENTATION: "../distributor/RemoteList.o"
    STATE: [{"i": 1}, {"i": 9}, {"i": 17}]
```

Por fim, também foi adicionado *requests* e *limits* [6] aos `Pods`. Além de ser uma boa prática em sistemas distribuídos - evitando *resource starvation* - adicionar esses parâmetros garante que ao rodar localmente, as réplicas não tomem todo o espaço disponível no computador, e que a latência seja compatível com a latência que teria rodando na *cloud*.

```
spec:
  containers:
  - # ...
    resources:
      requests:
        cpu: 200m
        memory: 256Mi
      limits:
        cpu: 200m
        memory: 256Mi
```

Após a criação de todo os `Pod` requisitados e a vinculação dos respectivos serviços, o servidor utiliza a informação do *readiness probe* para verificar se todos os Distribuidores estão prontos e com *IP*s associados. Só então a lista dos *IP*s dos novos distribuidores é finalmente retornada ao cliente para uso.

3.4 *Sharding* Dinâmico

Foi criada uma implementação nova para a interface de Lista presente no código do `Distributor`. Essa implementação tem como inspiração o código presente em `ListCPSharding.dn` e foi chamada de `ListCPDynamicSharding.dn`.

O código consiste em uma variação do *Sharding* convencional, mas trocando as referências aos `Remote Dists` fixos por uma chamada ao `Implementation Provider`. Foi também criada uma função (1) que define o número de *shards* desejado em relação ao tamanho da lista no momento da ativação daquela implementação.

$$\text{shards} = f(items) = \begin{cases} 1, & \text{se } x < 2000 \\ 2, & \text{se } x < 6000 \\ 4, & \text{se } x < 9000 \\ 8, & \text{senão} \end{cases} \tag{1}$$

Após feita a requisição ao `Implementation Provider` com a quantidade de réplicas desejadas, a implementação a ser utilizada e o estado de cada uma das réplicas, a lista de *IP*s retornados é uada na construção do vetor de `Remote Lists` e pode então ser utilizado nas requisições.

3.5 Testes de latência

Para testar a latência do serviço, foi criado um *script* em Python para fazer as requisições. Além do suporte já existente da requisição de `POST` - que cria um item - foi também adicionado suporte ao `DELETE`, que remove um item da lista.

Com isso, é possível testar mais eficientemente o impacto do *sharding* nas operações, já que apenas fazer o *retrieval* da lista completa não tira 100% de proveito das vantagens do *shard*.

O *script* consiste em um ciclo de adições de item, *reset* dos dados coletados pelo *Perception*, remoção de itens (inexistentes, para estarmos sempre atuando no pior cenário e não termos alteração do tamanho da lista) e por fim coleta dos dados de latência novamente. Essa última coleta por fim é usada - juntamente com a quantidade de itens e a quantidade de *shards* atuais - para consruir um `DataFrame Pandas`, que é então salvo em CSV.

O ciclo descrito anteriormente se repete com adições de 10 itens, até que o objetivo de 10000 itens seja atingido, onde o algoritmo para de rodar.

A função (1) - que descreve a quantidade de *shards* - foi também implementada nesse *script*. Quando o valor de $f(items)$ se altera, é feita uma requisição para a troca do modo de distribuição de dinâmico para local e em seguida para dinâmico novamente. Isso é feito para forçarmos a troca da quantidade de *shards* atuais, mais detalhes sobre isso na seção 5.

4 Resultados

Como esperado, a latência do servidor aumentou rapidamente no cenário com apenas um *shard*, saindo do *baseline* de latência de 10ms para 110ms após 10 mil itens terem sidos

inseridos na lista. A Figura 5 retrata a discrepância entre os dois cenários extremos que foram analisados (1 e 8 *shards*), é possível perceber que para o tamanho de lista estudado, 8 *shards* foram suficientes para manter a latência inicial, sem aumentos perceptíveis.

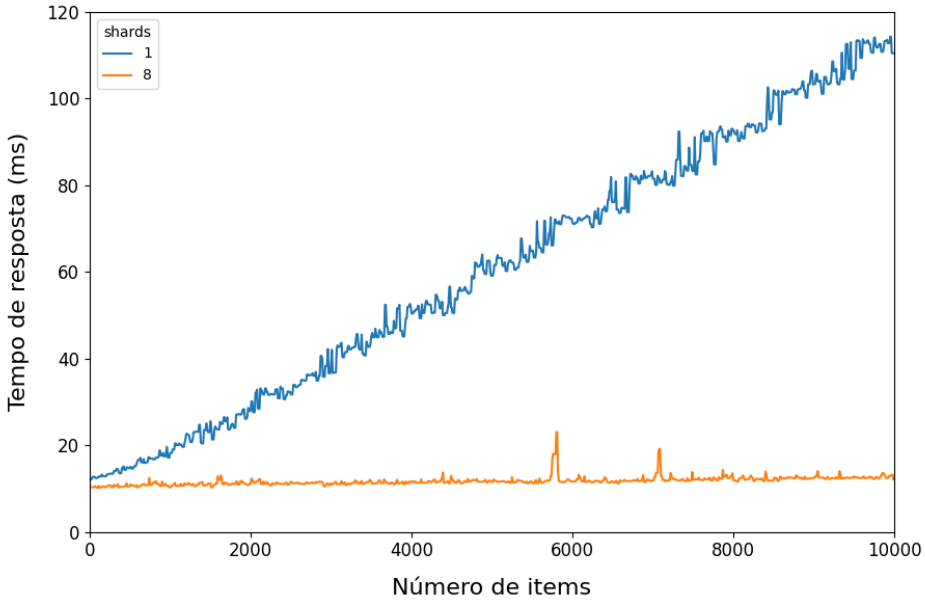

Figura 5: Latência com 1 e 8 *shards*.

Conforme aumentamos o número de *shards* disponíveis, menor é o coeficiente angular de aumento da latência em relação à quantidade de itens. Algo a se notar é que o coeficiente de latência diminui mais rápido do que o aumento do número de *shards*, como pode ser visto na Equação 2, que estima os parâmetros da função de latência $\mathcal{L}$ através de uma regressão linear dos valores de latência obtidos para cada número de *shards*. Essa função recebe o número de *shards* e a quantidade de milhares de itens (x), e retorna a latência estimada para aquele cenário.

$$\mathcal{L}(x, shards) = \begin{cases} 10.724x + 9.49, & \text{se } shards = 1 \\ 3.139x + 11.06, & \text{se } shards = 2 \\ 0.782x + 12.91, & \text{se } shards = 4 \\ 0.237x + 11.38, & \text{se } shards = 8 \end{cases} \tag{2}$$

Na Figura 6, podemos visualizar essas diferentes taxas de crescimento para cada configuração de *sharding*. Dessa vez a escala da imagem está limitada em 60ms para facilitar a visualização da diferença entre os *shards* com menor latência.

Por fim, podemos aplicar a troca dinâmica do número de *shards* de acordo com o número de itens. A Figura 7 explicita a queda de latência ao longo das trocas de configuração.

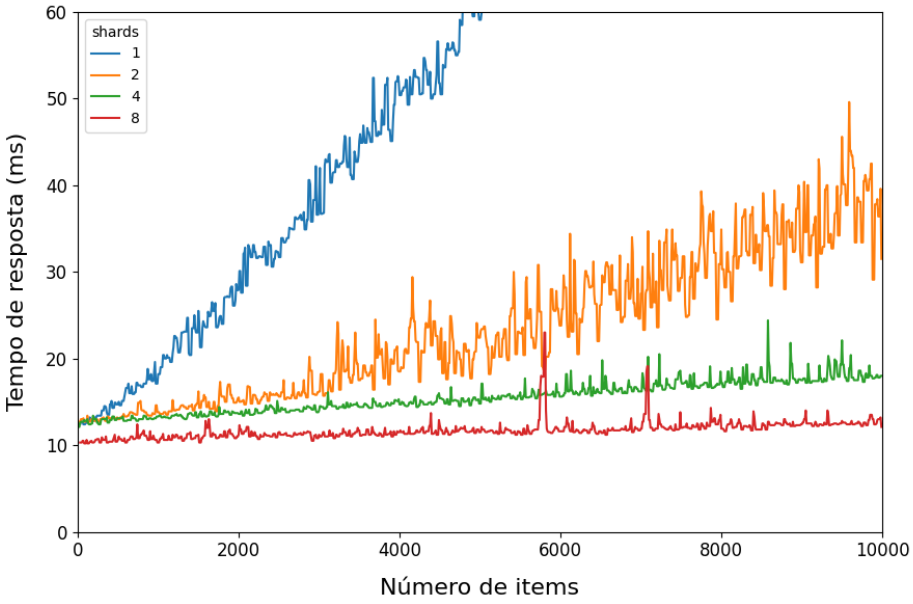

Figura 6: Latência para cada configuração de *sharding*.

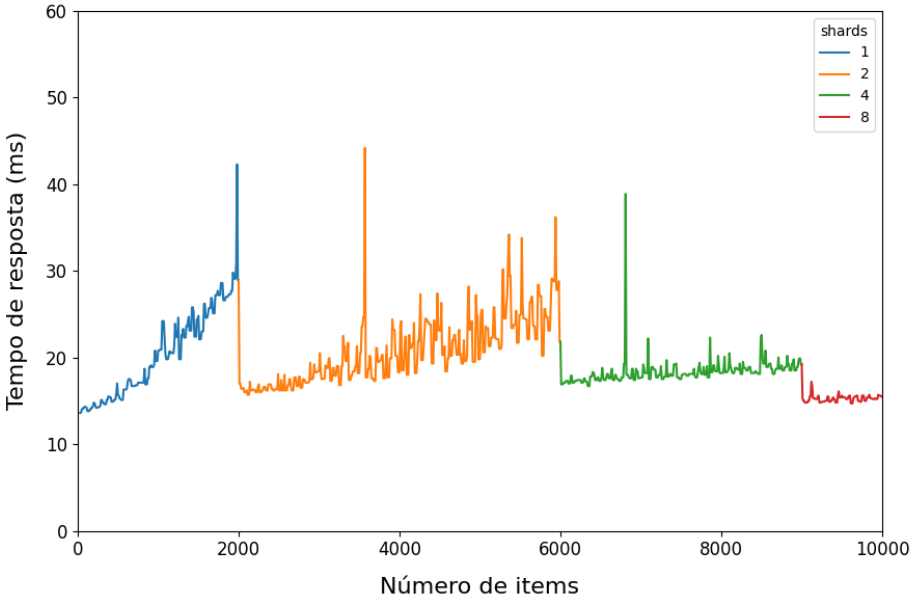

Figura 7: Configuração dinâmica de *shards*.

5 Trabalhos Futuros

Esse trabalho possibilita um aprofundamento maior no segmento de custos do algoritmo de *Reinforcement Learning* responsável por decidir qual configuração será utilizada pelo Sistema. É possível adicionar uma métrica de custo que também será ponderada por ele, levando em consideração os custos públicos das principais provedoras de *cloud*. Sem uma limitação de custo, é natural que o sistema deseje escalar para o maior número de *shards* possível, pois como vimos, isso sempre trará a melhor latência.

Outro ponto importante para se notar é que a troca do número de *shards* hoje em dia é feita de forma "manual", isso é, primeiro mudamos o sistema para a distribuição local novamente e depois re-distribuímos, a fim de forçar o cálculo do número de réplicas e passar pelas funções `active` e `inactive` da interface `AdaptEvents`. Uma possibilidade seria implementar alguma forma de reconfiguração da implementação - disparada pelo próprio objeto - que segue o mesmo ciclo de vida da adaptação original.

Finalmente, é necessário melhorar os pontos de quebra do número de *shards*. Os ponto escolhidos na Equação 1 são arbitrários e resolvem suficientemente bem o problema proposto. Por meio de *Reinforcement Learning*, poderíamos aprender esses pontos, mas seria necessário explorar novamente as possibilidades conforme o número de itens na lista avança, o que pode não ser interessante para a aplicação por conta do tempo de troca exigido entra cada *resharding*.

6 Conclusão

Esse estudo explorou como podemos dinamizar os recursos utilizados por um Sistema Auto-Distribuídos, removendo a necessidade de aplicações *idle* serem mantidas de pé sem necessidade.

Demonstramos como as aplicações podem atingir um bom equilíbrio de custo-benefício de forma transparente, se aproveitando da facilidade de troca de implementações disponibilizada pelos Sistemas Auto-Distribuídos.

Verificamos que, sem uma mudança no algoritmo de adaptação para a inclusão de custo, teremos um *scaling* excessivo, a fim de minimizar latência.

Concluímos que é possível ter um sistema que requisita os recursos necessários por ele dinamicamente, diminuindo a manutenção do código através de menos trocas nas referências do recurso e melhorando a governança dos distribuidores remotos, habilitando inclusive deixar a API de gerenciamento completamente interna.

Referências

[1] Ioana Baldini et al. "Serverless Computing: Current Trends and Open Problems". Em: (2017). arXiv: `1706.03178 [cs.DC]`. URL: `https://arxiv.org/abs/1706.03178`.

[2] Namiot Dmitry e Sneps-Sneppe Manfred. "On micro-services architecture". Em: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27. URL: `https://cyberleninka.ru/article/n/on-micro-services-architecture.pdf`.

[3] *Kind*. URL: https://kind.sigs.k8s.io/.

[4] *Kubernetes*. URL: https://kubernetes.io/.

[5] *Kubernetes Architecture*. URL: https://www.linkedin.com/pulse/demystifying-kubernetes-architecture-deep-dive-code-samples/.

[6] *Kubernetes Resources*. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[7] Gabriel Henrique Rosa Oswaldo, Luiz Fernando Bittencourt e Roberto Rodrigues Filho. *Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso*. Rel. técn. IC-PFG-21-50. Institute of Computing, University of Campinas, dez. de 2021. URL: https://ic.unicamp.br/~reltech/PFG/2021/PFG-21-50.pdf.

[8] Barry Porter e Roberto Rodrigues Filho. "A programming language for sound self-adaptive systems". Em: (2021), pp. 145–150. URL: https://ieeexplore.ieee.org/abstract/document/9659507.

[9] Roberto Rodrigues Filho et al. *A self-distributing system framework for the computing continuum*. IEEE, 2023. URL: https://ieeexplore.ieee.org/abstract/document/10230110.

[10] Roberto Rodrigues Filho et al. "Emergent software systems: Theory and practice". Em: *Sociedade Brasileira De Computação* (2021). URL: https://sol.sbc.org.br/livros/index.php/sbc/catalog/download/81/351/607-1.

[11] Roberto Rodrigues Filho et al. "Exploiting the potential of the edge-cloud continuum with self-distributing systems". Em: (2022), pp. 255–260. URL: https://ieeexplore.ieee.org/abstract/document/10061783.

[12] Eli Weintraub e Yuval Cohen. "Cost optimization of cloud computing services in a networked environment". Em: *International Journal of Advanced Computer Science and Applications* 6.4 (2015). URL: https://pdfs.semanticscholar.org/5409/43cb26ccc812fba42e2698890d2476df8651.pdf.

[13] *What is Container Orchestration?* URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration.