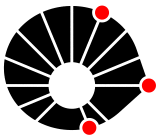

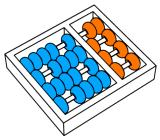

Definição de linguagem para geração de proxies para dar suporte a sistemas auto-distribuídos

L. F. E. Kina *D. C. C. D. Oliveira* *L. F. Bittencourt*
R. R. Filho

Relatório Técnico - IC-PFG-23-23
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Definição de linguagem para geração de proxies para dar suporte a sistemas auto-distribuídos

Luiz Felipe Eiske Kina* Daniel Cardoso Custodio de Oliveira*
Luiz Fernando Bittencourt* Roberto Rodrigues Filho†

Resumo

Com o crescimento do tamanho e da complexidade dos sistemas atuais, há uma crescente necessidade de otimização e aumento de velocidade de tecnologias que os suportam. Atualmente, há um grande uso de tecnologias de contêiner e computação em nuvem, no entanto, apesar delas possuírem a flexibilidade necessária dos sistemas modernos, ainda há uma grande necessidade de intervenção manual para seu funcionamento. Desse modo, sistemas com auto-distribuição surgem para melhor implementar as arquiteturas, já que esse método possibilita a gestão automática da distribuição dos sistemas e conseguem se adaptar em menor tempo. No entanto, esses sistemas ainda possuem uma dificuldade de implementação devido a sua dificuldade em ser utilizado de modo simples e *user-friendly*. Assim, surgiu a necessidade de métodos mais tradicionais para fornecer as informações necessárias para a construção desses sistemas de modo mais simplificado e rápido. Mediante uma análise de pontos-chave, foi criado um esquema de criação de sistemas com essa gestão autônoma de modo que seja acessível a futuros usos desde casos mais complexos a casos mais simplificados.

1 Introdução

Os sistemas modernos estão cada vez mais interconectados, com isso surge uma crescente complexidade e dificuldade de gerenciá-los [1]. A partir disso, Filho [2] sugere que o conceito de Computação Autonômica e Sistemas Auto-adaptativos com a intenção de melhorar o gerenciamento destes sistemas, deixando-os mais eficientes, otimizados e independentes de intervenção humana. Com a volatilidade destes sistemas modernos, esta solução Auto-adaptativa é responsável por fazer alterações imediatas conforme o ambiente em que se encontram os sistemas, baseando-se em *softwares* que possuem a autogestão como base.

A partir desta necessidade de autogestão como necessidade de adaptação em tempo de execução, a implementação desses sistemas foi feita baseada na linguagem de programação Dana [3], já que com ela, é possível a adaptação com a troca de componentes em um tempo curto.

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

Como alternativa à computação em nuvem para aplicações em larga escala [4], a computação autonômica não possui as dificuldades relacionadas às possíveis latências e atrasos em comunicação. Isso ocorre graças ao conceito de computação em borda, que se baseia na movimentação de uso em direção à borda da rede, utilizando melhor os recursos menos utilizados. Além do fato de possuir um melhor desempenho, devido à maior proximidade do usuário [5].

Em relação à infraestrutura proposta por este tipo de sistema, as estratégias mais utilizadas são a adição ou remoção de nós, sendo definido como *horizontal scaling*, ou réplicas em operação, definido como *vertical scaling*. As tecnologias de contêiner, orquestradores de contêineres e plataformas de computação em nuvem utilizam sistemas sem estado, conhecido como stateless, que não armazenam informações em si, necessitando, desta forma, da intervenção humana ou de banco de dados. Atualmente, utilizando o banco de dados como forma mais usual, a manipulação dos estados é feita manualmente e fixa [6].

Já utilizando aplicações com estado, conhecidas como *stateful*, operações identificam o estado atual e baseando-se em informações contidas em si conseguem executar novas ações. Com isso, mesmo com problemas de execuções, identificando o estado atual e ações já tomadas, as aplicações são capazes de continuar ou reiniciar o processo necessário, melhorando, assim, o desempenho [7].

Porém, a implementação, o projeto e a arquitetura dessas aplicações requerem uma maior atenção devido à maior complexidade de gerenciá-los. O objetivo deste projeto é de continuar a implementação da biblioteca para possibilitar estratégias utilizadas como remover, atualizar, clonar um nó a partir de informações obtidas de um arquivo de texto (yaml) em momentos de distribuições de listas. Foi escolhido o formato yaml porque o ele é um tipo de arquivo utilizado comumente para configurações e possui uma estrutura mais simples e intuitiva para o usuário.

Este relatório é dividido conforme: a Seção 2 introduz, brevemente, conceitos fundamentais utilizados durante o desenvolvimento do projeto. A Seção 3 identifica os objetivos, enquanto, na Seção 4, é exposto como foi o desenvolvimento da biblioteca. Na Seção 5 é explicado como foi implementado o projeto e seus resultados e, por fim, a Seção 6 apresenta a conclusão e possíveis trabalhos futuros.

2 Referencial Teórico

Os conceitos essenciais para o entendimento e desenvolvimento do projeto estão apresentados nesta seção. Sendo eles: Computação Autonômica e Sistemas Auto-adaptativos, o uso de Sistemas de Software Emergentes e sua ligação com sistemas baseados em componentes, e Estratégias para Gestão do Estado.

2.1 Computação Autonômica e Sistemas Auto-adaptativos

Em 2001, a IBM publicou um manifesto relacionado a um grande problema do progresso na indústria da tecnologia: a crescente complexidade dos softwares [1]. Devido à necessidade de integrar sistemas entre servidores e clientes, aumenta-se a dificuldade em projetar, configurar, otimizar e manter sistemas.

Assim, como proposta de solução, Kephart e Chess [1] propõem a ideia de computação autônoma, ou *autonomic computing*, na qual sistemas de computação baseiam-se em 4 pilares para o autogerenciamento, em que componentes, cargas, demandas e condições externas são alteradas. O primeiro pilar é a auto-configuração, de modo que sistemas autonômicos alteram configurações rapidamente sem necessidade de alterações manuais em detalhes de operações segundo os objetivos especificados. Segundo, a auto-otimização, em que o sistema automaticamente procura maneiras de otimizar os processos tanto em desempenho quanto em custo. Terceiro, a auto-cura, em que o sistema é capaz de identificar, diagnosticar e reparar qualquer falha que encontrar tanto por *bugs* quanto por falhas externas. Por fim, a autoproteção, que consiste no sistema identificar e criar maneiras de se proteger tanto de ataques maliciosos quanto de possíveis falhas em grande escala.

No entanto, este tipo de sistema possui seus próprios desafios, como foi destacado por Lemos R. *et al.* [8] em que este tipo de sistema requer uma rápida necessidade de se adaptar a casos de erros e problemas. Outro obstáculo é a necessidade de obter as informações externas relacionadas ao ambiente e relacionando isso ao seu estado para melhorar a especificação e a otimização do sistema. Este controle, ou seja, o responsável pelas tomadas de decisões, pode ser tanto centralizado quanto descentralizado.

No controle centralizado, é comum um componente único ser responsável por tomar todas as decisões com as informações que possui de cada componente, enquanto no controle descentralizado, as decisões e alterações são decididas a partir de interações entre as próprias componentes. Deste modo, diferentes opções foram sugeridas para lidar com este problema [8] como a abordagem em que há uma relação de hierarquia em que o nó principal é responsável pela análise e planejamento enquanto os nós secundários fornecem as informações necessárias para o monitoramento.

2.2 Sistemas de Software Emergentes

Sistemas de Software emergentes podem ser definidos como sistemas construídos com unidades de software que são capazes de auto-composição e auto-otimização baseados nos ambientes externos como as características de seu ambiente [2]. Isso foi proposto como facilitar para enfrentar as principais dificuldades resultantes do aumento da complexidade dos sistemas: necessidade de dependência humana; incapacidade de melhorar sua lógica em situações desfavoráveis; e as adaptações levam em conta apenas as partes diretamente ligadas, sem levar em conta o contexto do sistema na totalidade [4].

O desenvolvimento destes sistemas, em que os componentes possuem capacidade de serem alterados e adaptarem dependendo das condições em que o sistema se encontra levando em conta o uso em alto nível. Este uso de pequenas componentes como monitoramento de pequenas partes enquanto existe um controle central garante que o sistema se mantenha saudável e funcionando de maneira otimizada levando em conta o alto nível e não só as relações diretamente ligadas.

Para isto, o framework PAL, demonstrado na figura 1 possibilitou a sua arquitetura [2]. Este framework utiliza três módulos principais para organizar o sistema. O *Perception* fica responsável por fornecer as métricas dos componentes para verificar sua função dentro do sistema, através da coleta de métricas por meio de proxies nas conexões entre componentes.

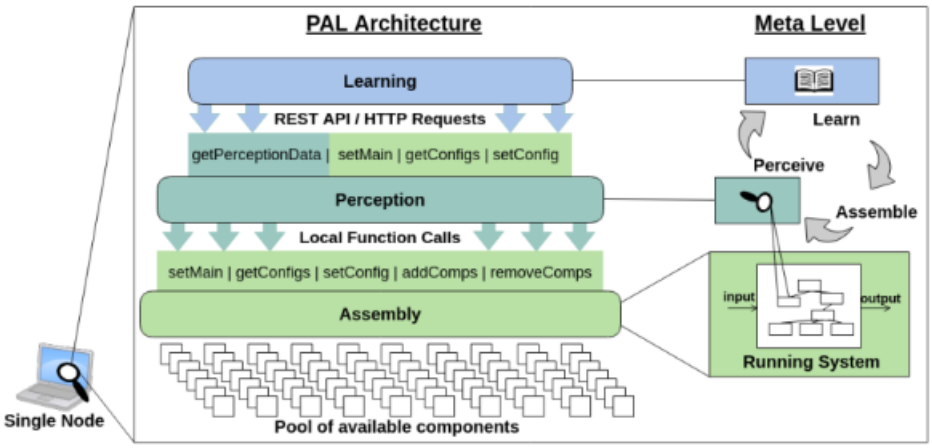

Figura 1: Arquitetura do framework PAL

Já o *Assembly* possui a função de receber as informações fornecidas pelo *Perception*, analisar as possíveis composições do sistema e, a partir disso, decidir qual a configuração ótima da arquitetura, permitindo sua reconfiguração em tempo de execução. Por fim, o *Learning* utiliza aprendizado por esforço para avaliar e otimizar o sistema consoante as necessidades previamente estabelecidas.

Como um dos pilares deste conceito é o modelo baseado em componentes, a linguagem Dana [3] foi proposta para possibilitar a reorganização dos componentes em tempo de execução. Deste modo, o módulo *Assembly* realiza alterações e adaptações em tempo de execução baseado em todas as composições possíveis, como exemplificação dos métodos para controlar o processo de alterações segue os exemplos abaixo.

- **string[ ] getConfigs()** : um método que retorna composiçoes de arquiteturas possiveis em formato de lista de strings;
- **char[ ] getConfig()** : um método que retorna qual a arquitetura atual em que o sistema se encontra, em formato de string;
- **bool setConfig(char configDesc[ ])** : um método que recebe um parâmetro de arquitetura do sistema e, em tempo de execução, altera a arquitetura do sistema;
- **bool addComp(char compName[ ])** : um método que recebe uma lista de componentes possíveis e adiciona um novo componente aumentando o número de composições de arquiteturas possíveis.

Por meio de métodos como esses, o *Assembly* consegue captar as alterações dos componentes que implementam uma interface, suas dependências de outras interfaces e seus componentes, e assim, recursivamente, consegue mapear todas as possíveis composições de arquitetura do sistema e, em tempo de execução, consegue otimizar as adaptações necessárias. A figura 2 ilustra esse processo descrito acima.

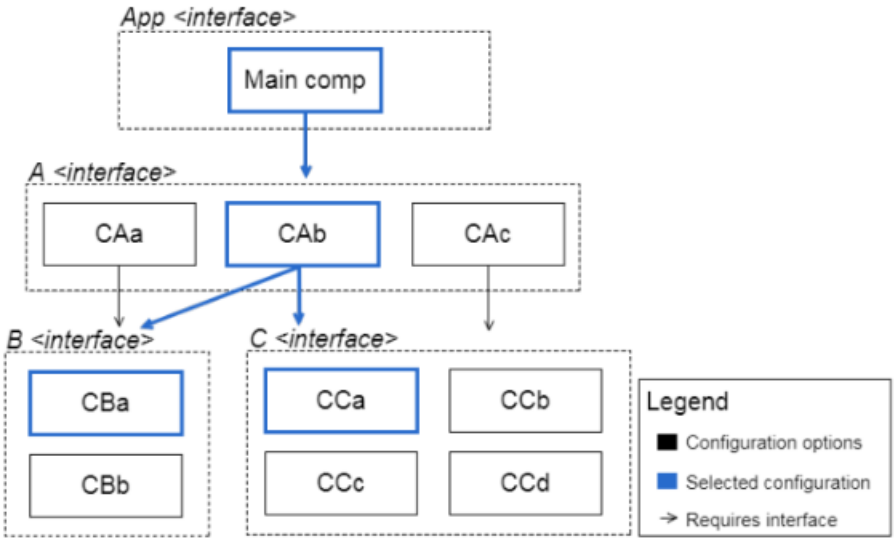

Figura 2: Exemplo de uma arquitetura genérica representada pelo módulo *Assembly* [4]. Pode-se observar as dependências entre componentes e interfaces através das setas e as variações possíveis de componentes para a arquitetura do sistema. A composição em azul representa o Assembly.

2.3 Gestão de Estado

Em sistemas auto-distribuídos, não somente a disponibilidade é importante como também a consistência do estado de cada atributo. Em sistemas tradicionais, como plataformas de nuvem e borda, os mecanismos de adaptação não oferecem suporte a gestão de estado - a sequência resultante de leitura e escrita sobre seus dados - a nível de aplicação [6]. Deste modo, é necessária uma intervenção manual de modo a encontrar o estado da aplicação e armazená-la em um banco de dados. Dessa forma, fica complexa a tarefa de manter a consistência dos dados quando o estado se encontra distribuído.

Esse problema pode ser abordado de duas formas. Primeiro, a consistência e corretude dos dados podem ser priorizadas em relação à *performance* do sistema. A outra abordagem consistirá em um sistema com maior *performance*, mas com uma consistência de dados menor, tendo que abdicar da corretude em algumas tarefas de escrita.

Nos sistemas de software emergentes, como o estado pode ser exposto em tempo de execução, durante a adaptação do sistema, pode-se extrair o estado de um componente e inseri-lo em um componente substituto. Sendo assim, um exemplo para lidar com isso em casos em que o modelo de dados possui partes distintas e independentes, seria a estratégia de sharding, já que ela consiste em particionar o estado em diferentes subconjuntos do conjunto total.

3 Objetivos

Este projeto procura facilitar a criação de *proxies* e, a partir disso, facilitar a criação de estruturas como listas em sistemas auto-distribuídos. Para isso, por meio de uma estrutura mais comumente utilizada para passar configurações, o yaml, o objetivo é de conseguir criar um proxy automaticamente e a partir deste proxy, criar um sistema em que listas seriam criadas com as configurações requeridas pelo desenvolvedor. Além disso, foi realizada uma comparação de velocidade entre a propagação local e o *sharding*, analisando qual cenário seria mais favorável para o uso de cada um.

4 Metodologia

Para conseguir atingir os objetivos propostos na pesquisa, um conjunto de operações foram implementadas com variações para avaliar como se comportam as adaptações em cenários distintos. Isso, aliado ao framework PAL de auto-adaptação garante que o sistema irá realizar as alterações necessárias e previstas.

Utilizando o trabalho de Oswaldo [9] como base, que possibilita a gestão do estado de maneira transparente de um sistema e de se adaptar em tempo de execução, continuamos a implementação do *Distributor*. Sendo este o responsável por iniciar a aplicação e depois disso, receber, interpretar e executar os comandos para gerenciar o sistema como um todo.

Em sistemas com configuração em que o sistema está distribuído entre vários *Remote Distributors*, os proxies existem para que instâncias sem estado ou com estados parciais armazenados localmente possam ser acessados em outra instância. Neste trabalho a forma de distribuição de Sharding vai ser a mais abordada, nesta distribuição, um hash multiplicativo possibilita que o *Distributor* identifique o *Remote Distributor* para determinar qual será a solução para qual estado e o local em que deve ficar. O módulo Assembly possibilita isso já que segue o comportamento do componente de mesmo nome do framework PAL e a linguagem Dana possibilita essa fácil e veloz troca de informações. Os *Remote Distributors* são as componentes que esperam as instruções do *Distributor* para serem alterados conforme a melhor configuração e a melhor arquitetura. Além disso, eles retornam seu estado local e geram proxies para melhor encaixar entre os outros *Remote Distributors* segundo o instruído pelo *Distributor*.

O objetivo é expressar como o estado tem que ser lidado utilizando essa linguagem. Não só determinando os atributos da interface, como também como lidar com o estado atual. E a partir das informações, o proxy é gerado de forma automática.

Utilizando a lista encadeada como interface básica e concreta, ela possui duas informações básicas, os elementos da lista e o ponteiro *iterator*. A partir desta interface como ponto de partida, podemos pensar no que é preciso expressar para gerar os proxies de forma automática a partir dessas informações. Várias proxies podem ser geradas a partir da lista, como o *Sharding*, o *Propagator* e o *Distributor*.

Ao observar, especificamente, o funcionamento do *Sharding*, como a lista está sendo definida e quais informações são necessárias, é necessário que essas informações sejam definidas pelo desenvolvedor para ser possível a criação automática do *proxy*. Como, por exemplo,

a consistência dos dados (forte, em todo momento do sistema os estados replicados estão consistentes, mas com redução de performance; ou, eventual, em um momento específico os estados podem não estar consistentes, mas em um passo futuro estarão constantes). Isso pode ser exemplificado por uma lista ordenada, em que ao ser fragmentada e depois ao ser juntada localmente é difícil identificar qual a ordem correta em que os elementos foram adicionados, a menos que tenha essa informação explícita na interface adquirida anteriormente para que o proxy consiga implementar.

Além disso, a assinatura de um método é outro ponto importante durante a criação dessa estrutura. Por exemplo, ao introduzir uma interface ADD, responsável pela inserção de um elemento, é necessário expressar a funcionalidade de forma que seja gerado um proxy a partir disso. Na inserção, é necessário identificar o que é esse elemento que será inserido e como encaixar ele dentro das regras da interface em que está sendo inserido. No caso de *Sharding*, algo associado ao item, como seu valor, é utilizado e a partir disso é gerado uma chave, a qual é inserida numa função hash e o resultado será o identificador da réplica obtida. Assim, os itens serão bem espalhados de acordo com essa função hash.

Desta forma, o projeto visa identificar e explicitar o vocabulário da linguagem necessário para as listas e para o *Sharding*. Desse modo, o programador saberá o que é fundamental de informações para criar um proxy de forma automática.

5 Resultados

Utilizando como base o trabalho de Oswaldo [9], procuramos fazer uma análise de pontos principais que cada modelo em específico precisaria. Como o replication possui uma implementação relativamente simples, o *Sharding* possui alguns pontos que requerem uma maior atenção. Como observado, o *Sharding* pode levar a inconsistência de dados, mas caso seja utilizado esse modelo de gestão de dados e uma necessidade de manter uma consciência de dados, isto pode ser implementado com uma queda de performance.

Primeiro, foi observado quando que faz sentido ser implementado o *Sharding* em relação a distribuição local. Para fazer essa comparação, é necessário explicitar como ela será feita. A distribuição local, mantém todos os dados na mesma máquina, desta forma o tempo total (T_L) poderá ser representado pelo tempo de rede (T_R) para o desenvolvedor enviar o método somado ao tempo de processamento dos dados (T_{PD})

$$T_R + T_PD = T_L$$

Já para o *Sharding*, é necessário levar em conta o tempo de rede(T_R), o tempo que o *Distributor* demora para processar(T_D), o tempo de rede para distribuir as informações entre o *Distributor* e os *Remote Distributors*(T_{RD}) e, finalmente, o tempo de processar os dados nos nós remotos(T_{DNR}). A soma de tudo isso leva ao tempo total (T_L)

$$T_R + T_D + T_RD + T_DNR = T_L$$

Com isso, encontramos os dados tanto para a adição quanto para obter os dados em uma lista. Assim, fizemos a adição de um número elevado de elementos à lista para garantir que o

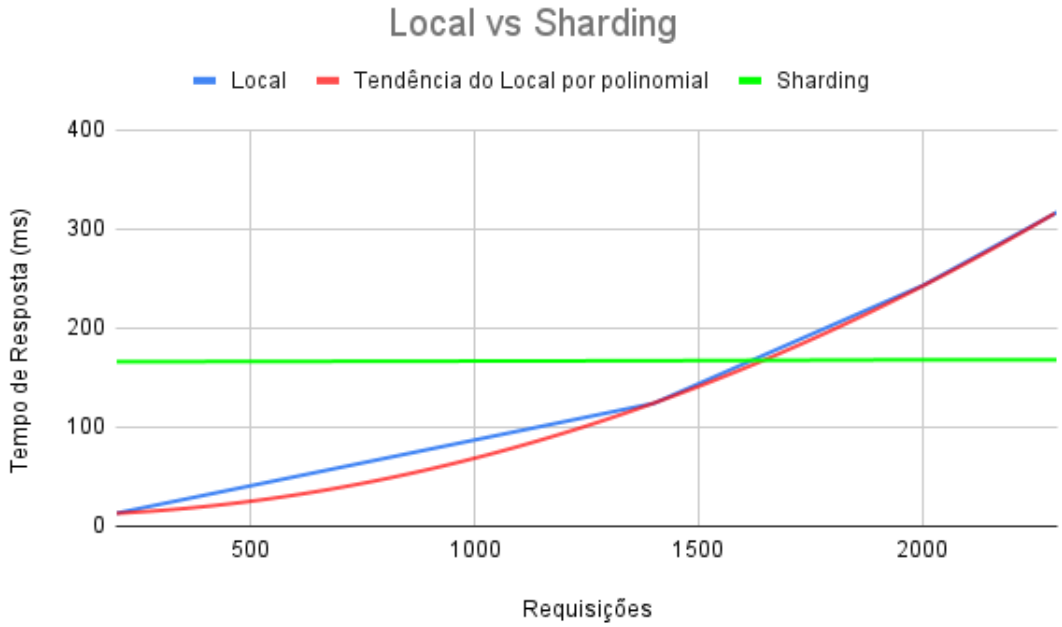

Figura 3: Tempo de resposta da adição de elementos na lista em relação ao número de requisições

processamento em nós remotos possa ser comparado com a distribuição local, já que com um número pequeno de elementos, a distribuição local terá uma vantagem já que não precisará lidar com um grande número de dados e não sofrerá com problemas de processamento local.

A figura 3 demonstra que com o aumento de requisições, o sharding possui um tempo de resposta menor em relação ao Local. O tempo de resposta do local possui uma tendência polinomial de crescimento com o aumento de requisições, enquanto o do Sharding mantém-se próximo ao de uma constante.

Já com o tempo de resposta de requisições de leitura dos dados, o Local possui um desempenho melhor em relação ao Sharding, como pode ser observado na figura 4. Os tempos ficam praticamente constantes independentemente do número de requisiçoes que são chamadas. Isso pode ser explicado porque a função getContents(), que já estava implementada, precisava fazer o processamento em todos os *Remote Distributors* que estavam na nuvem, aumentando significamente o tempo de resposta.

Porém observando todos os processos envolvidos, com o aumento de requisições de adições de elementos, rapidamente o *Sharding* pode ser uma melhor escolha.

A partir disso, foi feita uma análise para buscar quais são as chaves essenciais que precisam ser passadas pelos desenvolvedores do sistema que querem criar uma lista com um tipo específico de distribuição dependendo de sua necessidade. Esses dados seriam passados por meio de um arquivo de texto yaml com o preenchimento dos dados necessários. A figura 5 demonstra como seria o processo da criação automática das listas em nós remotos. Com

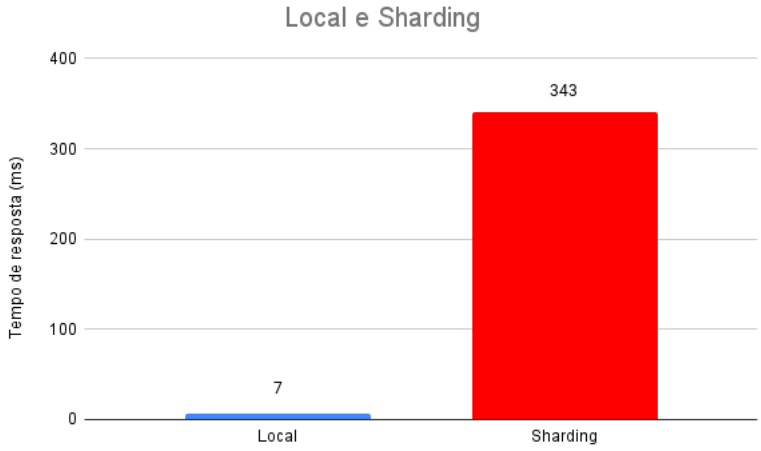

Figura 4: Tempo de resposta da execução de getContents() para Local e Sharding

o *input* do usuário via um yaml, é criado um *proxy* automaticamente e através dele, com as chaves dadas, são criadas as listas. A partir desses dados, foi criado um interpretador deste arquivo de texto utilizando a linguagem *Python* para receber as informações dadas, interpretar qual tipo de distribuição será realizado e a partir disso criar um arquivo, em tempo de compilação, do tipo ListCP que irá gerar uma lista com a distribuição que foi fornecida e com as informações contidas no yaml. As chaves essenciais para que o arquivo gere as informações necessárias para uma lista são:

Type : Consiste no tipo de distribuição que será utilizado, o compilador em *Python* que verificará esta chave e a partir dessa informação criar o arquivo de *ListCP* de acordo, os dados esperados são: *Sharding*, *Propagating* ou *Alternate*.

Key_generator : Esta chave determina qual atributo da estrutura de dados que deve ser gerado o hash que vai ser utilizado no sharding, como é uma particularidade do Sharding, ela não seria utilizada em configurações de Local e Propagate. Essa chave serviria para possibilitar o aumento de fragmentos no *hashing*.

Methods : Esta chave consiste na lista dos métodos que serão implementados para a estrutura que será gerada pelo proxy. Deste modo, a lista terá os métodos que foram requeridos pelo desenvolvedor de modo otimizado. Além disso, uma das variáveis necessárias para cada método que for dado na lista é o *cross sharding* que indicaria se o método precisa ser realizado em *cross sharding* ou não, essa flag facilitaria o modo em que o método é aplicado, ao saber se será necessário o *cross sharding* em algum dos métodos indicados, a criação da lista e de seus métodos fica simplificado para o uso de determinado método.

Type_list : Esta chave determina qual o tipo de estrutura de dados que a lista conterá, ou seja, ela espera um entrada como uma string ou um int. Isso possibilitar a criação da lista e o que ela espera que esteja dentro dela.

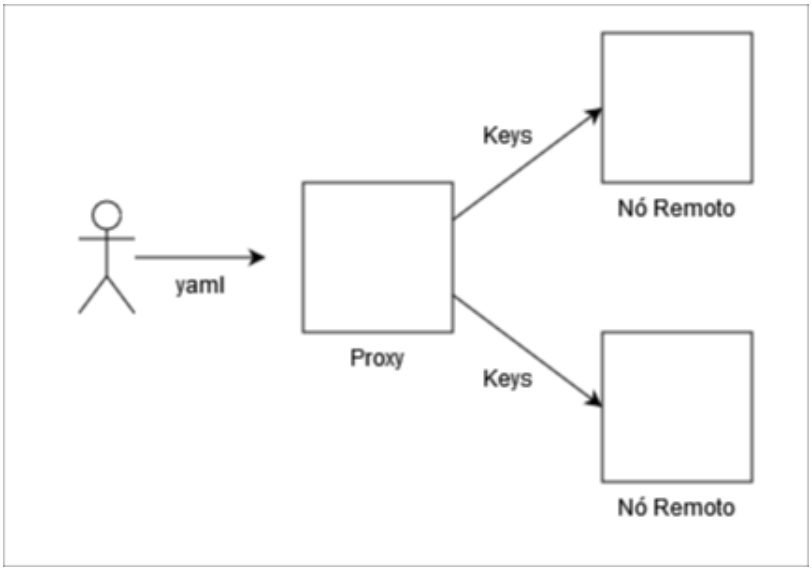

Figura 5: Ilustração do fornecimento de dados pelo yaml, que gera um proxy e a partir disso, o proxy consegue gerar os nós remotos com suas respectivas chaves.

Consistency_model : Esta chave determina qual a necessidade de consistência esperada durante a distribuição da lista. Os casos esperados são: *strong*, no qual o sistema mantém os estados distribuídos iguais a todo momento, o problema dessa abordam é a queda brusca do desempenho da máquina; ou, o *eventual*, em que em alguns momentos os estados que foram distribuídos não estão constantes com o restante das distribuições, no entanto em um momento futuro será estabilizado e eventualmente terão uma constância nos dados, o lado positivo dessa abordagem é a melhora significativa da performance.

Com essas chaves, um arquivo de texto yaml de exemplo para o *deployment* de um proxy para a geração de uma lista de inteiros com consistência eventual e com possibilidade de operações de adicionar, remover e buscar, pode ser exemplificado:

```
1  ---
2  Type: {'Sharding', Key_generator:}
3  Methods: [{'Add', cross-sharding: False},
4            {'Remove', cross-sharding: False},
5            {'Get', cross-sharding: True}]
6  Type_list: int
7  Consistency_model: Eventual
```

Com a adição do interpretador de arquivos de textos, é possibilitada a criação e a implementação de outras métodos de gerenciamento de sistemas autonomos de modo mais simplificado e rápido. Há maneiras de melhor otimizar a maneira de receber as chaves

de modo a para reduzir a quantidade de chaves e automaticamente selecionar um modo automático de tipo de gerenciamento em trabalhos futuros. Por exemplo, em casos de necessidade de maior consistência e métodos mais simples, é possível automaticamente gerar um sistema de *Replication* de imediato, sem a necessidade do input direto do usuário.

Com a criação desta estrutura, em yaml, foi possível criar o arquivo de criação dos proxies, mas não foi possível gerar o proxy de modo automático que era um dos objetivos do projeto. Isso aconteceu por alguns motivos, como a implementação de alguns métodos não estão completamente terminados, não foi possível adicioná-los aos requerimentos.

6 Conclusões e Trabalhos Futuros

Assim, neste projeto foi abordado a possibilidade de utilizar arquivos de textos com chaves específicas para a criação de *proxies* de sistemas de gerenciamento de sistemas auto-adaptativos. Utilizando conceitos da computação autonômica, os sistemas de software emergentes e a gestão de estado, foi possível analisar e chegar a chaves essenciais para a criação automática de distribuição.

Tendo em vista o *Sharding* como principal fonte de observação devido a sua alta complexidade de implementação e alta capacidade de *performance*, foi possível analisar os problemas e benefícios desse tipo de gerenciamento. Porém, ao facilitar a criação dos proxies por meio de métodos mais *user-friendly* como um yaml, fica mais fácil a utilização desse método.

Apesar da facilidade, ainda é necessário pontos fundamentais para que o uso seja conforme as necessidades do usuário. Deste modo, encontramos as chaves essenciais: tipo de gerenciamento, métodos que devem ser implementados para a lista, gerador de chaves para o hash, o tipo de informação que a lista irá armazenar e o nível de consistência que requer o sistema. A partir dessas chaves, com a criação de um interpretador de arquivo de texto para a linguagem Dana, foi possível fazer com que proxies fossem gerados automaticamente através dessa interface mais simples.

Como o trabalho foi mais focado em *Sharding*, há uma possibilidade de trabalhos futuros explorarem os outros tipos de gerenciamento e melhorar a implementação para os outros métodos. Além disso, foi feito uma implementação inicial apenas com a interface *List*, para outras interfaces mais complexas, é essencial que mais trabalho seja feito para que esse tipo de implementação de yaml satisfaça as necessidades para essas outras interfaces.

Por fim, esse trabalho focou em casos bem específicos do sistema de auto-gestão, trabalhos futuros podem analisar os diferentes modos de adaptação de infraestrutura e, assim, otimizar os requisitos pedidos pelo usuário. Visando facilitar a criação inicial de um sistema auto-distribuído ao utilizar arquivos mais comuns, como o arquivo de texto, o trabalho foi sucedido.

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, Barry Porter, Fábio M Costa, and Iwens Sene Junior. Emergent software systems: Theory and practice. Sociedade Brasileira de Computação, 2021.

[5] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick and D. S. Nikolopoulos, "Challenges and Opportunities in Edge Computing,"2016 IEEE International Conference on Smart Cloud (SmartCloud), 2016, pp. 20-26, https://doi.org/10.1109/SmartCloud.2016.18

[6] Roberto Rodrigues Filho and Barry Porter. Autonomous state-management support in distributed self-adaptive systems. In 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pages 176–181. IEEE, 2020.

[7] 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.

[8] Rogério De Lemos, Holger Giese, Hausi A M¨uller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers, pages 1–32. Springer, 2013.

[9] G. H. R. Oswaldo Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso. Universidade de Campinas, 2021