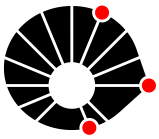
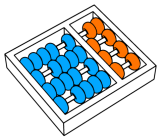




Um Estudo sobre Sistemas Auto-distributivos em Ambientes Elásticos

A. P. Oliveira *R. H. Koaro* *L. F. Bittencourt*
R. R. Filho

Relatório Técnico - IC-PFG-22-05
Projeto Final de Graduação
2022 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Um Estudo sobre Sistemas Auto-distributivos em Ambientes Elásticos

André Papoti de Oliveira* Ricardo Hayase Koaro*
Luiz Fernando Bittencourt* Roberto Rodrigues Filho†

Resumo

A capacidade de adaptar-se a diferentes situações é um processo chave em sistemas distribuídos modernos. O advento da internet em nossos cotidianos trouxe consigo um aumento considerável de heterogeneidade nos sistemas distribuídos. Uma das formas conhecidas e bastante utilizadas hoje para contornar tal problema é a computação em nuvem, que se tornou nos últimos anos um segmento fundamental para gigantes de tecnologia, conseguindo prover escalabilidade sob demanda de forma eficiente e rápida. Outras tecnologias trazem ainda mais flexibilidade para os sistemas, como, por exemplo, a capacidade de executar trocas de componentes em tempo de execução e a conteinerização de aplicações. Visando através de uma metodologia empírica explorar os desafios e impactos no uso das mais recentes tecnologias no contexto de gestão transparente de estado, este projeto apresenta um estudo do uso de tais tecnologias de acordo com diferentes modelos de consistência dentro do processo de distribuição de sistemas.

1 Introdução

A complexidade e escala de sistemas modernos demandam cada vez o uso de tecnologia que consigam prover adaptabilidade de forma eficiente devido à larga escala de operações que estes sistemas devem executar [1, 4, 5]. No contexto de sistemas distribuídos, adaptar é a habilidade de conseguir mudar seu comportamento ou própria estrutura dentre diferentes implementações disponíveis de acordo com seus objetivos de alto nível, como citado por Oswaldo (2021) [14].

Estratégias utilizadas, atualmente, em questão de infraestrutura caem normalmente em duas vertentes: *vertical scaling* em que os ajustes na infraestrutura ocorrem ao adicionar mais recursos, como poder computacional e memória e *horizontal scaling*, em que os ajustes nessa vertente ocorrem na forma de adição de nós ou réplicas na infraestrutura, mantendo os mesmos níveis de poder computacional.

Além das estratégias convencionais amplamente conhecidas, destaca-se o conceito de Sistemas de Software Emergentes [5, 9], que através da linguagem baseada em componentes, Dana[3], possibilita troca de componentes em tempo de execução de forma autônoma [3, 9].

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

Com isso, utilizando de tais conceitos combinado com tecnologias recentes, como contêineres, orquestradores de contêineres, e computação em nuvem, estudaremos o comportamento em relação à flexibilidade e escalabilidade de sistemas auto-distribuídos na nuvem com o objetivo de explorar os desafios e impactos na gestão de sistemas com estado para um ambiente distribuído e elástico, considerando que isto deve ser feito em tempo de execução, e garantindo a transparência do estado de acordo com diferentes modelos de consistência. Além disso, analisar o desempenho e impacto da mesma em diferentes configurações de distribuição do sistema, sejam eles positivos ou negativos.

2 Referencial Teórico

Esta seção apresenta conceitos necessários para o melhor entendimento do caso de estudo. É descrito também as técnicas utilizadas para a gestão do sistema abordado neste trabalho.

2.1 Computação Autonômica e Sistemas Auto-adaptativos

Computação Autonômica, definida por Kephart e Chess [1] e descrito por Oswaldo[14] como:

> Sistemas de computação que podem gerenciar a si próprios dados objetivos de alto nível pelos administradores, sem a necessidade da interação humana nos detalhes de operação e manutenção. A autogestão se dá na mudança de componentes, cargas de trabalho, demandas e condições externas, visando uma ou mais propriedades do sistema.

Portanto, pode-se dizer que um sistema autonômico tem sua infraestrutura composta de elementos autonômicos que tem a capacidade de alterar seu comportamento e relacionamento com outros elementos.

Durante o desenvolvimento deste relatório trabalhamos sobre uma aplicação que, dada a descrição, pode ser considerada um sistema autônomo e ao expô-la a um ambiente elástico buscamos explorar como melhor aproveitar as características que este tipo sistema pode oferecer e também produzir as ferramentas necessárias para que isso seja possível.

2.2 Sistemas de Software Emergentes

Da análise de Filho [5], notou-se que as abordagens convencionais utilizadas em sistemas auto-distribuídos acabam por evitar avanços dos sistemas pelo motivos numerados a seguir:

1. Dependência humana.
2. Mecanismos de adaptação não são generalistas o suficiente.
3. Impossibilidade da lógica de adaptação evoluir em frente de situações inesperadas.

Dessa forma, o conceito de sistemas de software emergentes foi proposto como uma abordagem na elaboração de sistemas auto-distribuídos visando facilitar seu desenvolvimento.

Conceito tal, que consiste em sistemas autonômicos compostos de pequenos componentes reutilizáveis a partir de métricas e eventos que ocorrem no sistema [5, 6, 9].

A implementação do conceito de Sistemas de Software Emergentes detalhada a seguir utilizou-se do *framework PAL*[9], consistindo de três módulos diferentes:

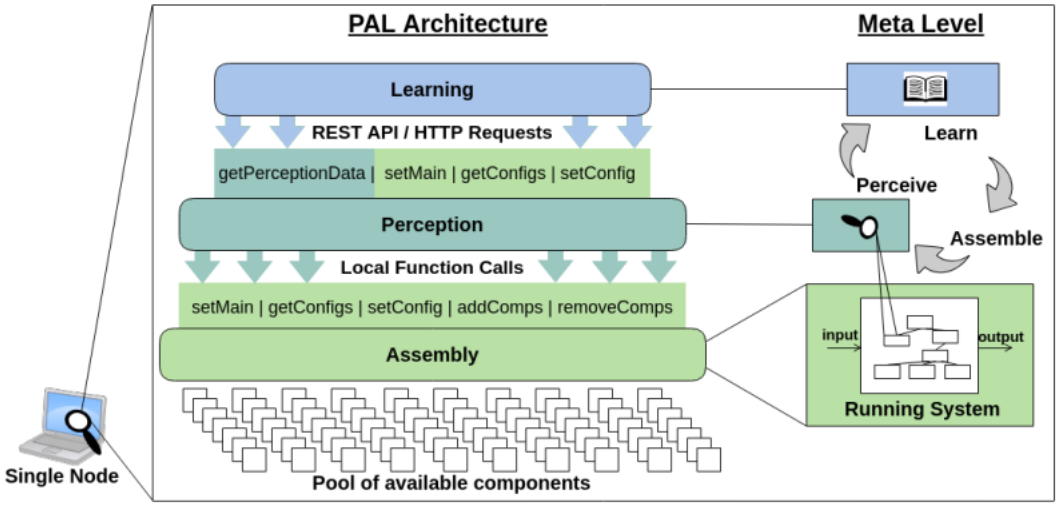


Figura 1: Arquitetura do framework PAL [9].

- *Assembly*, responsável pela gestão dos componentes disponíveis, de forma a representar todas as possíveis combinações de componentes do sistema e fornecer suporte para a troca de composição dos mesmos em tempo de execução.
- *Perception*, responsável pela criação e adição de componentes *proxies* na composição do sistema.
- *Learning*, responsável por gerir os processos de adaptação autonômicas de acordo com os objetivos do sistema, utilizando de aprendizagem de reforço.

Portanto, tanto o *framework PAL* como as demais implementações neste projeto utilizam da linguagem de programação Dana [3], já que esta nos fornece a capacidade de trocar de componentes do sistema em tempo de execução, qualidade que é essencial para alcançarmos a distribuição de estado de forma transparente.

2.3 Estratégias para Gestão do Estado

Um dos desafios enfrentados por sistemas distribuídos que armazenam estado durante sua execução é garantir disponibilidade dos dados de maneira segura e confiável.

Uma das estratégias mais populares para se atingir este objetivo é ter todos os módulos que rodam em ambientes instáveis seguirem uma arquitetura *stateless*. Nesta arquitetura módulos não armazenam qualquer dado relacionado ao estado, sua funcionalidade se assemelha a de uma função pura, e dados dessa natureza se tornam responsabilidade de módulos

que rodam em ambientes mais estáveis, geralmente bancos de dados. Assim, logo que um módulo *stateless* necessita de uma informação relacionada ao estado da aplicação ele faz uma requisição à parte responsável por armazená-lo de forma estável e, caso ocorra uma falha, não teremos o comprometimento do estado.

Porém este tipo de abordagem requer um grande esforço de implementação por parte dos desenvolvedores, além de precisar adaptar toda a arquitetura do sistema para a estratégia[4], dificultando muito a migração de sistemas para ambientes elásticos que inerentemente apresentam um maior risco de falha.

Em contrapartida, sistemas distribuídos cujos módulos apresentam uma arquitetura *stateful* ou seja, armazenam dados sobre o estado mesmo rodando em ambientes instáveis, exigem maior atenção em relação à consistência dos dados. Porém tal modelo possibilita um *tradeoff* entre a corretude dos dados e performance geral do sistema. Uma consistência fraca possibilita uma alta disponibilidade e maior *throughput* ao mesmo tempo se torna menos confiável, enquanto uma consistência forte afeta negativamente a performance geral do sistema em troca de mais segurança[7].

Uma dessas estratégias que apresenta uma ênfase maior no aumento de performance é a de *sharding*. Esta estratégia consiste em realizar o particionamento de um conjunto de dados entre diferentes réplicas de um determinado sistema, sendo bastante vantajosa quando esse conjunto de dados é independente e tem uma alta taxa de requisição, reduzindo assim a competição por recursos computacionais. Durante o relatório exploramos o comportamento de uma aplicação que implementa essa estratégia dentro de um ambiente de nuvem comparando a performance entre diferentes números de réplicas e em relação à mesma aplicação armazenando o conjunto de dados de forma centralizada.

2.4 Conteinerização de processos

Contêineres são aplicações leves que contém todas as dependências necessárias, como arquivos de configuração e bibliotecas, para executar um processo específico. São muito utilizados em aplicações que rodam em nuvem devido a sua alta flexibilidade e facilidade de escala [12].

Diferentemente de máquinas virtuais, os contêineres fazem uma virtualização apenas das camadas de software acima do sistema operacional, devido a isso diferentes contêineres em uma mesma máquina são capazes de compartilhar os seus recursos, tornando a operação muito mais escalável visto que se em algum momento um dos contêineres tem uma carga menor de trabalho para ser realizada, os outros podem utilizar os recursos que estariam reservados para o uso exclusivo deste caso houvesse a virtualização do hardware.

Utilizamos desta tecnologia durante o trabalho para executar e gerar as imagens das instâncias que vão compor os diversos módulos da nossa aplicação de forma simples, eficiente e escalonável.

2.5 Orquestração de contêineres

Na palavras de Casalicchio [13], orquestração é o conjunto de operações que a nuvem provedores e proprietários de aplicativos se comprometem (manualmente ou automaticamente)

para selecionar, implantar, monitorar e controlar dinamicamente a configuração de recursos para garantir um certo nível de qualidade de serviço.

A orquestração de contêineres, porém, não só se compromete com a gestão do estado inicial do sistema, como também permite a adaptação de sistemas auto-distribuídos em tempo de execução, como escalonamento vertical e horizontal.

Essas ferramentas foram essenciais para simplificar e automatizar o processo de levantamento ou destruição de instâncias da aplicação dentro do ambiente de nuvem.

3 Abordagem

Para realizar a gerência de estado partimos do trabalho de Oswaldo[14] que desenvolveu um sistema capaz de fazer a gestão do seu estado de maneira transparente e também de se adaptar em tempo de execução utilizando a linguagem *Dana*[3]. Este é dividido em duas camadas diferentes: A camada de aplicação e a de adaptação, também chamada de *meta-level*.

Na camada de adaptação se encontram os componentes responsáveis pela adaptação e distribuição da aplicação. O primeiro destes componentes é o *Distributor*. Este é responsável pelo início da aplicação e por escutar, interpretar e executar comandos relacionados à adaptação do sistema. Ao receber um desses comandos o *Distributor* realiza a adaptação dos componentes existentes no sistema e distribui seu estado através de um conjunto de *proxies* a fim de alcançar a configuração especificada pelo comando.

Em configurações onde o estado está distribuído entre múltiplos *Remote Distributors* os *proxies* são utilizados para que instâncias que não tenham o estado, ou parte dele, armazenado localmente possam acessá-lo em outra instância que o contenha através de uma mesma interface. No caso do nosso trabalho, estaremos distribuindo a estado utilizando configuração de *Sharding* onde, através de um *hash multiplicativo*, o *Distributor* é capaz de saber em qual dos *Remote Distributors* disponíveis no momento se encontra determinada parte do estado ou em qual deles deve armazenar uma nova parte dele. Este tipo de operação é possível por conta do módulo *Assembly* que segue o comportamento do componente de mesmo nome do framework PAL e a capacidade de aplicações escritas em Dana de adaptar seus componentes em tempo de execução.

O outro componente dessa camada, previamente já citado, são os *Remote distributors* que aguardam comandos do *Distributor* para eventos de adaptação, fornecem seu estado local e gerem os *proxies* usados para acessar partes do estado presente em outros *Remote distributors* e fornecê-las para sua aplicação local.

Já na camada de aplicação temos a estrutura que armazena o estado local e os métodos que interagem com ele. No caso do sistema usado o estado é uma lista de números inteiros.

Durante a fase de implementação do projeto, nosso objetivo foi obter a capacidade de rodar a aplicação de forma flexível dentro de um ambiente de nuvem, podendo criar e destruir instâncias sem que o estado fosse perdido ou alterado a fim de aproveitar todas as vantagens que o meio nos provém.

Para concretizar este objetivo foi necessário permitir que o componente *Distributor* fosse capaz de se comunicar com o motor do Kubernetes[17] porém, como a linguagem

Dana[3] não possui uma API nativa para este fim, partirmos do trabalho de Dias[1] [15] que desenvolveu um servidor em Python[15], denominado de *Server CTL*, que faz chamadas à API do Kubernetes[17] e um novo componente no *meta-level* de mesmo nome que se comunicam através de requisições HTTP e assim disponibilizando as informações que o *Distributor* necessita para realizar a adaptação. Porém, tanto o componente quanto o servidor ainda não eram capazes de fazer a adaptação da aplicação livremente, se limitando a apenas rodar esta localmente ou com *sharding* do estado para duas instâncias de forma rígida.

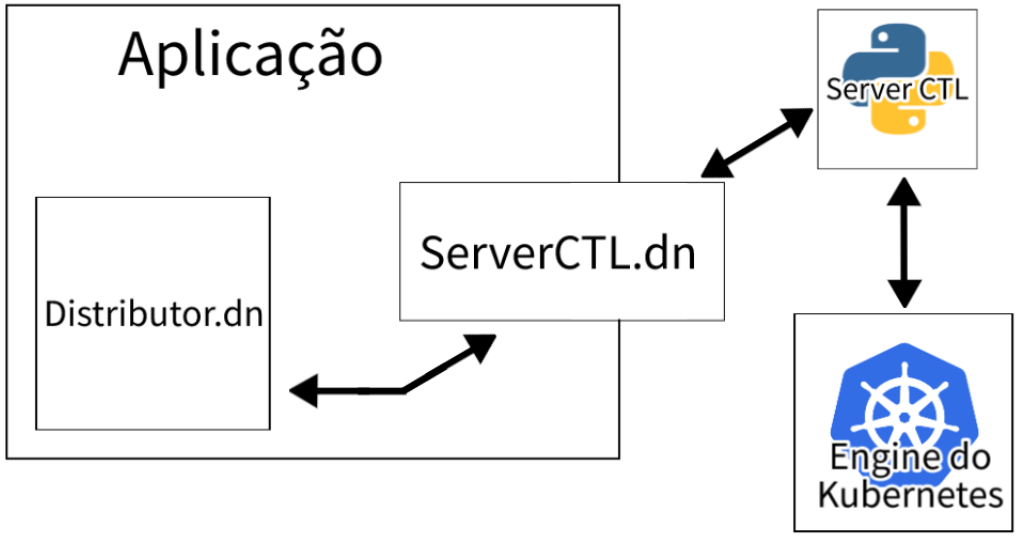


Figura 2: Resultado do experimento utilizando requisições assíncronas: Os vales na curva são momentos onde temos uma operação de escrita que aumenta o tamanho da lista, em seguida temos operações de leitura que vão progressivamente aumentando seu tempo de resposta devido ao gargalo gerado pela falta de aptidão do servidor para atender requisições simultâneas

Nosso trabalho consistiu em refatorar tanto o módulo quanto o servidor para que pudesse fazer as requisições e disponibilizar os dados necessários para que a aplicação fosse capaz de usar a elasticidade do ambiente na nuvem por completo além de configurações adicionais para a criação das imagens. Dentre os dados extraídos estão os endereços IP dos *Remote distributors* dentro do *cluster* essenciais para a configuração dos *proxies* existentes no *meta-level*.

Além disso, foi desenvolvido um cliente utilizando NodeJS[16] capaz de fazer requisições HTTP síncronas e assíncronas à aplicação para a realização dos experimentos. As requisições disparadas por esse cliente são processadas pelo *Distributor* que, através da comunicação com o *Server CTL*, consegue realizar a gestão de estado e processar a requisição do cliente.

[1] Repositório com o trabalho de R. Dias: https://github.com/RSilvaDias/SCTL-PFG

O ambiente escolhido para o projeto foi o *Google Cloud*[18] utilizando *Docker* para a criação das imagens e *Kubernetes*[17] para fazer o orquestramento dos contêineres. A escolha de *stack* foi feita principalmente pela grande quantidade de discussões acerca das tecnologias presente na internet e familiaridade dos orientadores com as mesmas, visto que nenhum dos alunos tinha experiência prévia com ferramentas semelhantes. Dito isso, os conceitos desenvolvidos durante essa atividade podem ser executados em qualquer *stack* de *cloud computing*.

4 Estudo de caso

A aplicação desenvolvida por Oswaldo[14] implementa uma lista com a qual podemos interagir realizando leituras e adições sendo ambas ordenadas ou não dependendo da topologia da aplicação no momento. A questão da ordenação em si não é o foco, mas sim o fato de que esta representa um processamento arbitrário sobre o estado e nos permite visualizar os ganhos que o ambiente elástico traz à aplicação. Os testes realizados durante o seu trabalho[14] foram feitos em um *cluster* local utilizando apenas duas instâncias executando *Remote Distributors* simultâneas.

Agora, utilizando as ferramentas que desenvolvemos e a versão adaptada do sistema para ambientes de nuvem, queremos explorar como este se comporta dentro de um ambiente elástico. Para isso, realizamos uma série de testes que exploram as possíveis configurações do sistema com relação à sua topologia e ao número de instâncias compartilhando um estado e observamos principalmente a variação do tempo de resposta das requisições enviadas a este entre diferentes configurações e diferentes tamanhos da lista.

Para as possíveis topologias a serem testadas temos três implementações:

- Adição ordenada e leitura simples: Substituímos o componente que faz a implementação do método de adição de um elemento na lista por outro que faz a inserção de forma ordenada. Internamente a lista é implementada usando a estrutura de dados de *heap* e por conta disso a complexidade desta operação é *O(log(n))*.

- Leitura ordenada e adição simples: Substituímos o componente que faz a implementação do método de leitura da lista por um que retorna um array ordenado dos elementos da lista. O algoritmo utilizado para realizar a ordenação é o *Merge Sort* e, portanto, a complexidade da operação de leitura é *O(n.log(n))*.

- Leitura e adição ordenadas: Substituímos os componentes de leitura e adição pelos citados nos itens anteriores. Apesar desta configuração não trazer nada de novo, ela é interessante para realizarmos testes que busquem emular cenários mais realistas no sistema dado que ambas operações geralmente são acopladas a algum tipo de processamento nestes casos.

E em relação ao número de instâncias rodando a aplicação variamos entre rodar a aplicação em configuração "local", onde temos uma única instância do *Distributor* rodando na nuvem e responsável por todos elementos da lista, ou realizando a distribuição dos elementos através da estratégia de *sharding* para um número qualquer de instâncias do

Remote Distributor. Em nossos experimentos realizamos testes realizando o *sharding* entre duas, quatro e oito instâncias.

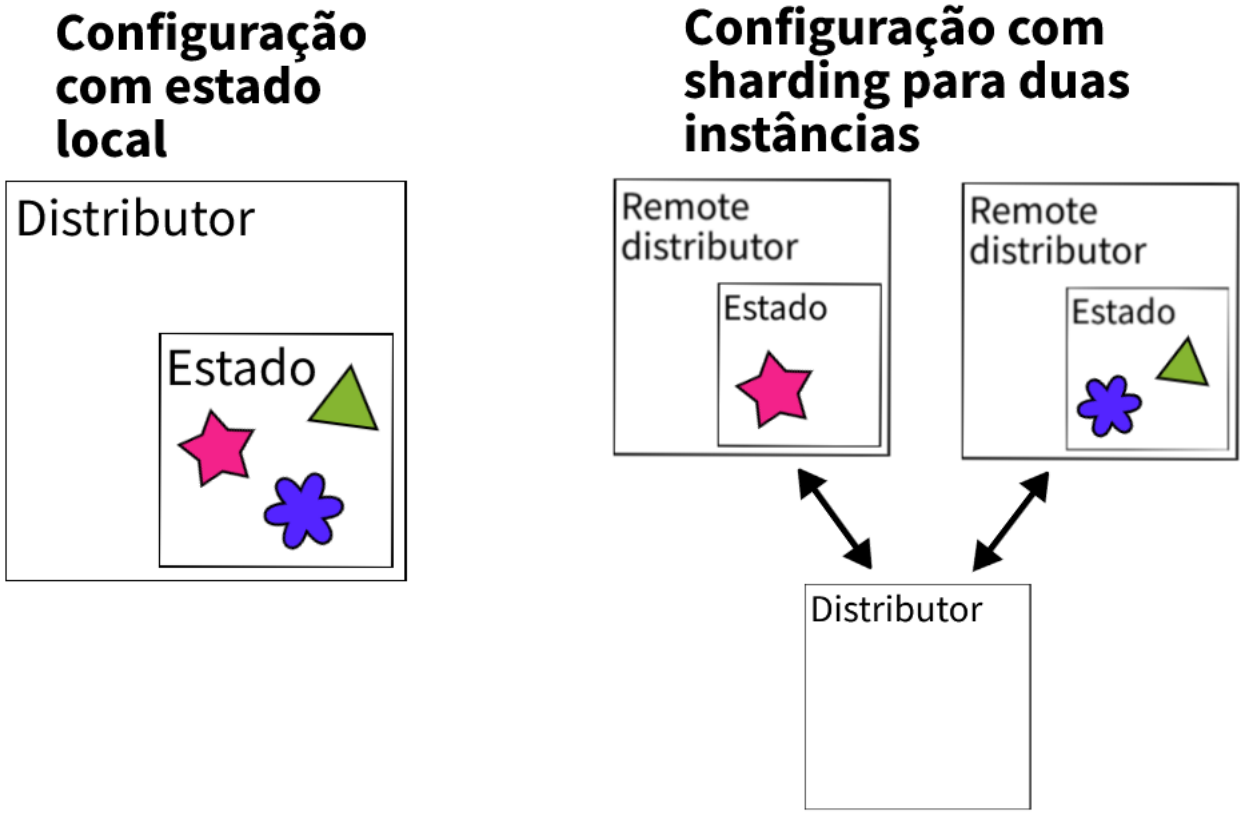


Figura 3: Resultado do experimento utilizando requisições assíncronas: Os vales na curva são momentos onde temos uma operação de escrita que aumenta o tamanho da lista, em seguida temos operações de leitura que vão progressivamente aumentando seu tempo de resposta devido ao gargalo gerado pela falta de aptidão do servidor para atender requisições simultâneas

Para as *requests* que enviamos ao sistema durante os testes também variamos a proporção entre requisições de leitura e escrita e a cadência com que essas são disparadas e observamos como o sistema se comporta diante diferentes cargas. Em relação à proporção, realizamos testes com mais operações de leitura e menos de leitura para simularmos cargas que mais comuns as de um sistema distribuído que esteja em produção. Também realizamos testes com só um dos tipos de operação para estressar o módulo corresponde ao máximo e observar seu comportamento.

Já em relação à cadência realizamos testes com requisições assíncronas, onde requisições são disparadas com um intervalo fixo de tempo entre si durante a duração do teste, e requisições síncronas, onde é preciso esperar uma resposta ou timeout da request atual para enviarmos a seguinte.

Ao realizarmos testes compostos de requisições assíncronas nos deparamos com uma limitação para a realização dos experimentos. A quantidade de recursos disponíveis para o trabalho na plataforma do *Google Cloud*[18] somados a baixa eficiência do servidor escrito em Dana[3], uma vez que a linguagem ainda está em um estágio de prova de conceito e não tem a performance necessária para ambientes de produção, fizeram com que o servidor tivesse uma capacidade de *throughput* muito baixa. Assim, ao executarmos a sequência de requisições assíncronas, observamos que os resultados obtidos sofriam com a interferência de um gargalo na capacidade do servidor de processar as requisições. Este é um desses resultados:

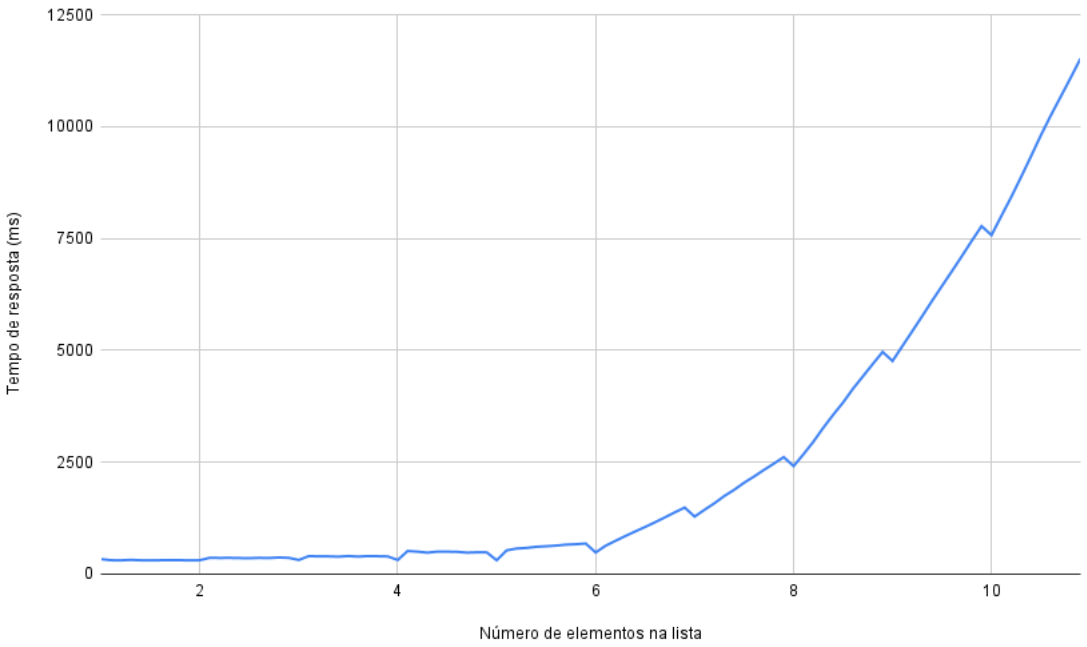


Figura 4: Resultado do experimento utilizando requisições assíncronas: Os vales na curva são momentos onde temos uma operação de escrita que aumenta o tamanho da lista, em seguida temos operações de leitura que vão progressivamente aumentando seu tempo de resposta devido ao gargalo gerado pela falta de aptidão do servidor para atender requisições simultâneas

Nele fixamos uma taxa de 5 requisições por segundo, sendo que a cada 10 requisições a primeira era uma operação de escrita e as seguintes eram operações de leitura, e o sistema foi configurado para a usar a composição onde a inserção é simples, a leitura é ordenada.

Pelo resultado podemos observar que, para um mesmo tamanho de lista, requisições de leitura foram progressivamente aumentando o tempo de resposta apesar do processamento necessário para atender cada *request* ser sempre o mesmo visto que a lista não foi alterada. Todas as *requests* são processadas pelo *Distributor* que sempre é executado em uma única instância independentemente da configuração que estamos utilizando para realizar a distri-

buição do estado e, ao enviarmos uma taxa de requisições maior que o *Distributor* suporta, temos um incremento inconstante do tempo de resposta das requisições que não são relacionados a configuração atual do sistema, fugindo assim do escopo que queremos explorar durante esse trabalho.

Por conta desse motivo, decidimos realizar o resto do estudo utilizando apenas requisições síncronas para que seja possível obter de forma mais clara a relação entre o tempo de resposta com o tamanho da lista e o número de instâncias armazenando o estado sem a interferência de outros fatores.

5 Avaliação

Esta seção apresenta os resultados dos experimentos, além de uma análise do desempenho levando em conta o tempo de resposta do sistema aos diferentes cenários de composição do mesmo.

Com nossos testes tentamos responder se existe e como se comporta, caso exista, a relação entre os componentes distribuídos e o tempo de resposta do sistema. Como mencionado anteriormente na seção de Abordagem, para realizá-los utilizamos o método de *sharding* para gerir estado e as operações de escrita e de leitura mencionadas na seção de Estudo de Caso.

Para analisar o desempenho do sistema em relação ao tempo de resposta é necessário entender as variáveis que compõem o tempo total, que será diferente dependendo da configuração do sistema.

Na composição local da aplicação, o componente *Distributor*, onde são armazenados os *proxies* do sistema, não realizou o *sharding* dos dados, ou seja, todos os dados estão todos localizados na mesma máquina, dessa forma, o tempo total (T_L) pode ser descrito pela seguinte fórmula:

$$T_L = T_R + T_{LP} \tag{1}$$

Onde (T_R) representa o tempo de rede para o usuário enviar a requisição e (T_{LP}) representa o tempo de processamento dos dados na aplicação.

Já para uma composição distribuída, onde é feita a distribuição do estado da aplicação através do mecanismo de *sharding*, o tempo total (T_D) é representado pela fórmula:

$$TR_D = T_R + T_{DP} + T_R^{'} + T_{DP}^{'} \tag{2}$$

Onde, T_R ainda representa o tempo de rede para o usuário enviar a requisição, T_{DP} o tempo que o *Distributor* leva para processar a requisição, enviar para os nós remotos e retornar uma resposta para o usuário, $T_R^{'}$ representa o tempo de rede para transmitir da requisição entre o *Distributor* e as instâncias de *Remote Distributor* e, por fim, $T_{DP}^{'}$ representa o tempo de processamento em cima da coleção de dados nos nós remotos.

Um ponto importante a ressaltar é o fato de que no *cluster* disponível para o trabalho ficamos limitados a usar no máximo três nós, portanto houve casos em que, devido a distribuição dos contêineres entre os nós não ser determinística neste caso, existiam contêineres

rodando no mesmo nó do *Distributor*, tornando o tempo de rede $T_R^{'}$ praticamente nulo para estes nós. Porém, devido ao processamento nos nós remotos serem paralelos, desde que todos os nós não estivessem na mesma máquina que o *Distributor* a fórmula se mantém, pois $T_R^{'}$ vai ser igual ao maior tempo de transmissão entre o *Distributor* e um nó remoto.

Na análise das equações 1 e 2, percebemos que a distribuição do sistema não é vantajosa para todo cenário. O uso do mecanismo de *sharding* só é mais rápido no quesito de tempo de resposta quando: $T_{LP} > T_{DP} + T_R^{'} + T_{DP}^{'}$ ou, em outras palavras, quando o tempo de processamento local for superior ao tempo do *proxy* local, somado ao tempo de rede para transmitir ao nó remoto e ao seu respectivo tempo de processamento.

Dessa forma esperamos que para coleção menores de dados realizar a distribuição não seja beneficial para o tempo de resposta visto que o tempo adicional de rede entre nós $T_R^{'}$ não compensa a diferença entre realizar o processamento local e realizar o processamento de forma distribuída em nós remotos. Porém, conforme o crescimento da coleção de dados, dividir este processamento em múltiplas instâncias compensa o tempo adicional de rede entre nós e esta configuração passa a se tornar a mais vantajosa.

Toda a dinâmica de chamadas de requisições pode ser vista na figura a abaixo:

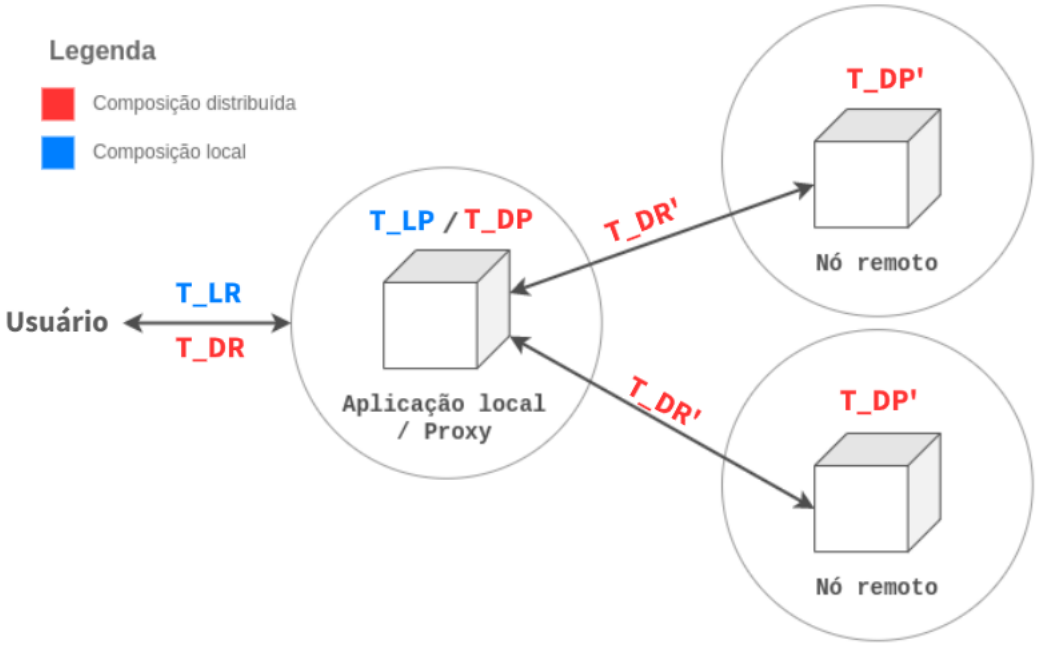


Figura 5: : Ilustração das variáveis envolvidas na análise do tempo de resposta total para as composições local e distribuída. [14]

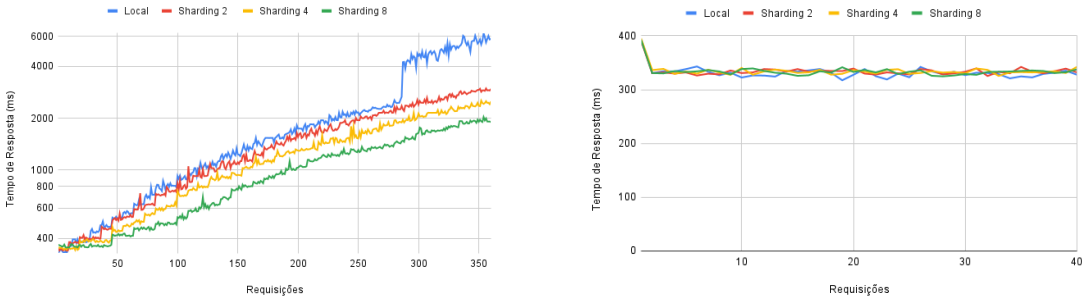
Com isso, utilizamos o cliente desenvolvido em Node.js[16] para realização de cargas de trabalho com diferentes proporções de operações escrita e leitura, totalizando 400 requisições síncronas por teste com as diferentes implementações do componente *List* mencionado na seção Estudo de caso. Desta forma foram gerados os seguintes cenários:

Cenário	Operação com processamento	Carga de trabalho (leitura:escrita)
A	leitura e escrita	9 : 1
B	leitura	9 : 1
C	leitura e escrita	1 : 9
D	escrita	0 : 1

Tabela 1: Cenários de teste para avaliar a análise sobre a composição de *s*harding.

5.1 Cenário A

Para validar nossas hipóteses, o sistema foi executado primeiramente na configuração local, em seguida o sistema foi executado com configurações onde a distribuição foi feita para dois, quatro e oito *pods*. Obtivemos os seguintes tempos de resposta:



(a) Requisições GET

(b) Requisições POST

Figura 6: Média dos tempos de respostas para o cenário A da Tabela 1

Com isso, para as requisições de leitura, verificamos que a composição distribuída se mostrou mais vantajosa, as configurações de *sharding* com dois e quatro *pods* mostraram uma performance superior à da configuração local, conforme o esperado. Porém a hipótese de que as configurações distribuídas apresentariam um desempenho pior que a configuração local para uma lista de menor tamanho não é verdade nesse caso. Como podemos ver no gráfico os valores de tempo de resposta no começo do experimento ficaram todos bem próximos, logo não há diferença razoável de performance nessa faixa, independente da configuração do sistema e, logo em seguida, configurações que envolvem o *sharding* já se mostram mais atrativas. Por conta disso, concluímos que o tempo de requisição para os nós T'_{DR}, pelo fato de estarem todos no mesmo *cluster* que o *proxy*, é irrelevante.

É interessante notar que a hipótese de que a diferença de tempo de resposta entre as configurações distribuídas e local aumentariam conforme a lista aumentasse, o que pode ser confirmado com a subida abrupta no tempo de resposta para a distribuição local no momento que a lista chega em aproximadamente 29 itens.

Outro ponto importante é o fato do processo de escrita além de inserir o dado enviado no *heap* inclui também a tarefa de balanceá-lo, de forma que a complexidade de tempo do

processamento de leitura se torna O(n).

Para as requisições de escrita, dada a complexidade da operação de escrita ser de *O(log(n))* a quantidade de itens na lista não foi suficiente para gerar uma carga de trabalho suficiente que tornasse a distribuição do sistema vantajosa para esse caso. Para termos um aumento significativo no tempo de resposta das requisições de escrita precisaríamos atingir um tamanho de lista que faria com que o tempo de resposta de leitura fosse irrazoável.

5.2 Cenário B

Idêntico aos testes do cenário A, exceto pela topologia da aplicação que foi modificada para usar o módulo de escrita simples:

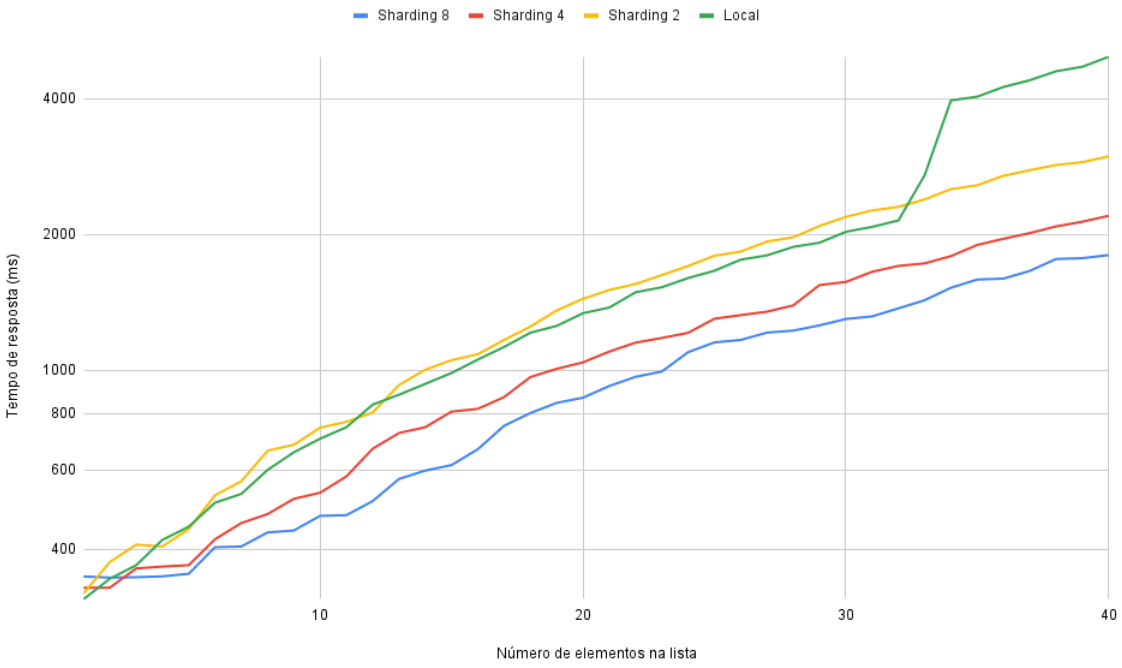


Figura 7: Média dos tempos de respostas para o cenário B da Tabela 1

Tivemos um resultado muito similar ao cenário A, visto que usamos o mesmo módulo para a leitura. Até 32 itens na lista há uma paridade entre rodar o sistema localmente ou com *sharding* para duas instâncias e é possível observar que conforme o número de instâncias aumenta, menor nosso tempo de resposta fica dado que a carga de processamento é distribuída entre mais instâncias.

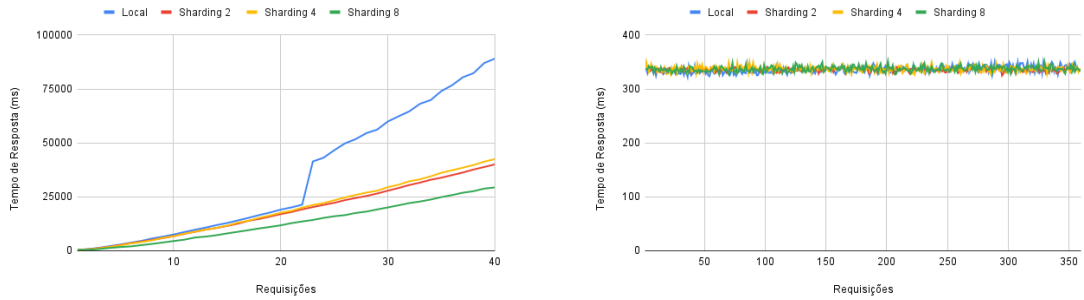
Devido ao módulo de escrita ser simples, as requisições de escrita deste experimento tiveram um tempo de resposta constante próximo de 300ms, comportamento este que já era esperado.

Algo que devemos destacar é que, idealmente, o cenário B teria um tempo de resposta maior, para um mesmo número de itens na lista dado que não há ordenação nenhuma sendo

feita durante a escrita enquanto no cenário A ao inserirmos um item em um nó o fazemos de forma com que este seja inserido e processado no *heap*, mudando sua complexidade de tempo. Porém, como o algoritmo utilizado para a ordenação durante a leitura é o *Merge Sort* e este possui uma complexidade de *O(n.log(n))* não temos nenhum ganho real de performance por realizar a ordenação durante a inserção, pelo numero de itens da lista não ser expressivo o bastante para que a diferença seja notada.

5.3 Cenário C

Neste cenário utilizamos uma carga de trabalho bem mais incomum, com 9 operações de leitura para cada operação de escrita, com o intuito de entender melhor como as operações de leitura se comportam em relação à distribuição. O sistema foi executado primeiramente na configuração local, em seguida o sistema foi executado com configurações onde a distribuição foi feita para dois, quatro e oito *pods*. Esses foram os resultados:



(a) Requisições de leitura

(b) Requisições de escrita

Figura 8: Média dos tempos de respostas para o cenário C da Tabela 1

Para as requisições de leitura, vemos novamente o mesmo comportamento visto nos cenários A e B reforçando ainda mais a confirmação das hipóteses levantadas enquanto que, para o caso das requisições de escrita, apesar do aumento considerável de carga de trabalho, a quantidade de itens na lista ainda não foi o suficiente para mostrar uma melhora de performance na distribuição do sistema.

5.4 Cenário D

Por último realizamos um teste que trabalhasse o módulo de escrita ordenada o máximo possível para que pudéssemos observar seu comportamento. Para isso escolhemos a topologia local já que o estado fica todo concentrado nesta e, por consequência, o tempo de processamento sobe mais rapidamente e realizamos 2000 requisições.

Como esperado de um tempo de processamento de complexidade *O(log(n))* a curva mostra uma inclinação muito tênue, sendo necessário uma quantidade alta de requisições, e consequentemente itens na estrutura de dados para mostrar uma tendência de crescimento.

Figura 9: Média dos tempos de respostas para o cenário D da Tabela 1

6 Trabalhos Futuros

Nesta seção, discutimos as limitações de nossa abordagem como oportunidades para trabalhos futuros. Durante o trabalho ficamos limitados a utilizar apenas um *cluster* e três nós computacionais na plataforma do *Google Cloud*[18]. Nesses cenários, o tempo de comunicação entre as instâncias é desprezível, fazendo com que configurações onde ocorre a distribuição do estado em sua maior parte superiores a configurações locais. Ao rodarmos instâncias em diferentes máquinas com uma distância física considerável entre elas, teremos um tempo mais expressivo para a comunicação entre instâncias e será possível observar melhor as vantagens e desvantagens de diferentes topologias.

Além disso, a adaptação do sistema ainda é feita de forma manual através de comandos passados para o módulo *Distributor*. Para que o sistema se torne auto-adaptativo [1, 14] é necessário explorar mecanismos que, através do monitoramento da performance do sistema, sejam capazes de aprender e realizar a adaptação deste de forma autônoma.

Por fim seria interessante desenvolver mais componentes com diferentes implementações para a lista que representa o estado do sistema ou até mesmo adicionar novas estruturas de dados para aumentar o número de possíveis topologias e tornar os mecanismos de auto-adaptação mais flexíveis.

7 Conclusão

Neste projeto foram estudados conceitos sobre computação autonômica e sistemas autoadaptativos, assim como sobre Sistemas de Software Emergentes, e estratégias de gestão de dados em ambientes elásticos conteinerizados na nuvem. A partir disso, o projeto propôs a construção e análise do sistema distribuído. Os experimentos tiveram como objetivo entender o funcionamento do sistema proposto e analisar seu desempenho em termos de tempo de resposta em relação às diferentes cargas de trabalho e composição. Dessa forma, foi possível obter dados sobre situações que o sistema se beneficia da sua distribuição.

Analisando os resultados obtidos, a distribuição em um maior número de instâncias se mostrou benéfica para em relação ao tempo de resposta em todas configurações de distribuição para chamadas de leitura, principalmente em cenários que uma grande quantidade de dados era processada. Para chamadas de escrita a distribuição não se mostrou vanta-

josa, apesar de também não apresentar nenhum malefício. Pelo processamento de leitura ser de complexidade *O(n.log(n))* é fácil notar o crescimento da curva do tempo de resposta em relação ao tamanho dos dados processados, porém como as chamadas de escritas são processadas com complexidade *O(log(n))* é necessário muito mais dados para notar-se um crescimento de sua curva. Além disso, vale notar a constatação de que o tempo de requisição entre o *Distributor* e nós executando *Remote Distributors* não afetou os tempos de resposta de maneira significante.

Por fim, os resultados se mostraram satisfatórios, confirmando as hipóteses inicialmente propostas e conseguindo atingir os objetivos deste estudo de caso para a gestão transparente de estado em sistemas auto-adaptativos em ambientes elásticos.

Referências

[1] J. O. Kephart and D. M. Chess, *T*he vision of autonomic computing, in *C*omputer, vol. 36, no. 1, pp. 41-50, Jan. 2003. https://doi.org/10.1109/MC.2003.1160055

[2] de Lemos R. *e*t al. (2013) *S*oftware Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: de Lemos R., Giese H., Müller H.A., Shaw M. (eds) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol 7475. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35813-5_1

[3] Linguagem Dana: http://projectdana.com/.

[4] R. R. Filho and B. Porter, *A*utonomous State-Management Support in Distributed Self-adaptive Systems, 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), 2020, pp. 176-181. https://doi.org/10.1109/ACSOS-C51401.2020.00052

[5] Emergent Software Systems: Theory and Practice. https://robertovrf.github.io/pdfs/sbrc2021rodrigues.pdf.

[6] Barry Porter, Matthiew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. *:* A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (Savannah, Georgia, USA). USENIX, 14 pages. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter

[7] M. van Steen and A.S. Tanenbaum, *D*istributed Systems, 3rd ed., https://www.distributed-systems.net/, 2017.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (1990) *I*ntroduction to Algorithms, MIT Press, Cambridge MA and McGraw-Hill, New York.

[9] R. Rodrigues Filho. *E*mergent Software Systems. PhD thesis, Lancaster University, 2018.

[10] Kleppmann, Martin. *D*istributed Systems. Maarten van Steen (2018).

[11] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *T*he Design and Analysis of Computer Algorithms, Addison-Wesley (1901).

[12] Bentaleb, O., Belloum, A.S.Z., Sebaa, A. et al. *C*ontainerization technologies: taxonomies, applications and challenges. J Supercomput 78, 1144–1181 (2022). `https://doi.org/10.1007/s11227-021-03914-1`

[13] E. Casalicchio. *A*utonomic Orchestration of Containers: Problem Definition and Research Challenges. Blekinge Institute of Technology Karlskrona, Sweden, 2016.

[14] G. H. R. Oswaldo *G*estão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso. Universide de Campinas, 2021

[15] Linguagem Python: `https://www.python.org/`.

[16] Runtime NodeJS: `https://nodejs.org/`.

[17] Kubernetes `https://kubernetes.io/`

[18] Google Cloud Platform `https://cloud.google.com`