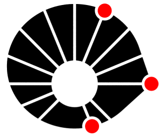

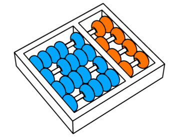

Explorando o Desempenho de Sistemas Auto-distribuídos na Borda e Nuvem

F. A. L. Guardão   *B. B. Araujo*   *L. F. Bittencourt*
R. R. Filho

Relatório Técnico - IC-PFG-22-08
Projeto Final de Graduação
2022 - Junho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

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

Explorando o Desempenho de Sistemas Auto-distribuídos na Borda e Nuvem

Bruno Berbare de Araujo[*] Felipe Augusto Leomil Guardão[*]
Luis Fernando Bittencourt[*] Roberto Rodrigues Filho[†]

Resumo

Cada vez mais os sistemas contemporâneos têm sua demanda e complexidade aumentadas. Os principais fatores que têm causado esse aumento são a heterogeneidade e a volatilidade presentes nesses sistemas. Um exemplo dessa volatilidade é o súbito aumento no número de usuários, ou serviços compartilhando a mesma infraestrutura. Com isso, algumas alternativas vêm sendo analisadas e aplicadas para atender essa alta demanda. Uma das mais disseminadas é a computação em nuvem, uma vez que a mesma possui recursos para lidar com a escalabilidade e a adaptabilidade, como exemplo dos contêineres e orquestradores de contêineres, que conseguem escalar tanto verticalmente quanto horizontalmente. Mas devido a uma baixa tolerância à alta latência e atrasos de comunicação em algumas aplicações, o desempenho desta solução pode ficar comprometido, abrindo espaço para novas soluções, como a computação na borda. Sendo ambas as tecnologias melhor utilizadas por sistemas sem estado (*stateless*), pelo fato de que sua implementação é menos complexa. Porém, sistemas com estado (*statefull*), embora possuam uma maior complexidade, também possuem um maior desempenho.

Pensando neste cenário, este projeto foi desenvolvido a fim de estudar o desempenho de um sistema híbrido — parte na nuvem, parte na borda — com uma gestão transparente do estado, de acordo com diferentes modelos de consistência, no processo de distribuição de sistemas. Através de uma metodologia empírica, são explorados diversos cenários e composições com o intuito de avaliar as melhores composições para cada cenário.

1 Introdução

Com a tendência da digitalização do mercado e o desenvolvimento de diversas tecnologias para acompanhar esse crescimento, os softwares estão ficando cada vez mais complexos, possuindo diversos módulos interconectados para provimento de serviços [1]. Dessa forma, os desenvolvedores necessitam ter o conhecimento de diversas tecnologias para fazer sua manutenção e, a medida que os sistemas se tornam mais interconectados e heterogêneos, os arquitetos são menos capazes de antecipar as interações do projeto entre os componentes, deixando tais questões para serem tratadas com o projeto já em execução [1].

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

Dessa forma, conforme os sistemas vão crescendo em complexidade e demandando uma alta escalabilidade [1, 3, 9], uma alternativa é a computação autonômica. Sistemas de computação que podem se auto-gerenciar conforme objetivos de alto nível estabelecidos pelos administradores [1, 9]. A base dos sistemas de computação autonômicos é a autogestão, garantindo a possibilidade de alterar os seus componentes, cargas de trabalho e demandas de acordo com condições externas e de maneira automática. Sobrando para os administradores apenas o direcionamento dos objetivos finais do sistema. Assim, possibilitando a simplificação do gerenciamento destes sistemas.

Outra alternativa utilizada em larga escala é a computação na nuvem [3], onde podemos citar tecnologias de contêiner e orquestradores de contêineres, como tecnologias que auxiliam o suporte à elasticidade. Com elas é possível fazer ajustes nas capacidades de recursos computacionais de um nó (*vertical scaling*) ou a adição, ou remoção de nós, ou réplicas (*horizontal scaling*) em tempo de execução. Embora seja uma ótima alternativa para atender o problema de escalabilidade, para algumas aplicações com baixa tolerância à alta latência e atrasos na comunicação, o desempenho em nuvem pode comprometer a experiência do usuário. Logo, para conseguir atender todo esse volume, é apresentado o conceito de computação em borda, baseado na movimentação de algumas dessas cargas em direção à borda da rede, melhor aproveitando os recursos que atualmente são menos explorados. Além disso, traz um desempenho maior, devido ao processamento estar mais próximo do cliente [6].

Porém, essas alternativas são mais utilizadas por sistemas sem estado (*stateless*), que melhor se aproveitam das ferramentas disponibilizadas. Aplicações *stateless* são aquelas que não registram o estado no sistema, ou seja, o sistema é responsável apenas pela manutenção e envio do estado, mas este fica armazenado em outro local. Dessa forma o estado necessita de uma manutenção manual e fixa, podendo então tornar-se um gargalo e, em algum momento, começar a afetar o desempenho [7]. Enquanto as aplicações *stateful* são aquelas que podem executar novos processamentos com base em contextos anteriores, e uma vez que uma operação *stateful* é interrompida, a mesma possui mecanismos para que o contexto não seja perdido e futuramente o processo possa continuar de onde parou. Portanto, como o contexto é armazenado na aplicação, o desempenho desta é maior, porém a complexidade de implementação desses sistemas também, principalmente devido aos mecanismos de consistência do estado.

Dito isso, para explorar os ambientes tanto na nuvem, quanto na borda, e para aproveitar as vantagens tanto dos sistemas *stateful* e *stateless*, este projeto visa estudar o desempenho em uma composição híbrida entre borda e nuvem em sistemas auto-distribuídos, que conseguem fazer a realocação de componentes com estado da borda para a nuvem em tempo de execução. O sistema em estudo armazenará o estado, podendo distribuí-lo entre borda e nuvem, utilizando-se de diferentes estratégias de manipulação do mesmo. Desta forma explorando o desempenho de diferentes composições em diferentes cenários. Para isso foi-se utilizando da abordagem de Sistemas de Software Emergentes, que facilita o desenvolvimento de sistemas auto-adaptativos através de modelos baseados em componentes, além de um *framework* capaz de construir diferentes arquiteturas de maneira autonômica [3].

Este projeto foi dividido nas seguintes seções. Na segunda seção são apresentados os conceitos que permeiam o projeto, enquanto na terceira seção são expostos os objetivos

buscados pelo projeto. A quarta seção descreve a metodologia utilizada para o desenvolvimento do projeto. Na quinta seção é descrita a abordagem adotada para a adaptação do sistema, enquanto os detalhes da aplicação implementada são detalhados na sexta seção de estudo de caso. A sétima seção expõe e avalia os resultados obtidos. Na oitava seção são destacadas as possibilidades de investigação para trabalhos futuros. E por último, na nona seção são feitas algumas considerações finais sobre o projeto.

2 Referencial Teórico

Nesta seção será apresentada a fundamentação teórica utilizada na construção do projeto. Serão introduzidos os conceitos de sistemas auto-adaptativos, computação autonômica, além de discutir a perspectiva sobre Sistemas de Software Emergentes para a implementação de sistemas auto-adaptativos, e sua conexão com sistemas baseados em componentes. Também serão apresentados conceitos sobre computação na nuvem e na borda.

2.1 Computação Autonômica e Sistemas Auto-adaptativos

Em 2001 foi publicado um manifesto pela IBM, que relatava que o principal desafio para a indústria de tecnologia da informação é a crescente complexidade dos softwares [1]. Neste manifesto é destacada a crescente complexidade no gerenciamento de sistemas computacionais, devido a integração de diversos módulos que constituem o sistema. Dessa forma, é declarado que até os melhores profissionais terão cada vez mais dificuldade em conseguir planejar, configurar e melhorar os sistemas, antevendo problemas de diversidade, escalabilidade e distribuição.

Kephart e Chess discutem que uma possível solução para esse problema é a computação autonômica [1], que pode ser definida como sistemas computacionais capazes de se auto-gerenciar a partir de objetivos de alto nível providos pelos administradores, evitando assim a necessidade da intervenção dos desenvolvedores em detalhes da manutenção. Além disso, a autogestão atua na mudança de elementos, demandas e cargas de trabalho, a partir de fatores externos, de escalabilidade ou das propriedades do sistema.

As propriedades que um sistema autonômico deve possuir são, auto-configuração, auto-cura, auto-proteção e auto-otimização. Um sistema auto-configurável é aquele capaz de se auto-configurar a partir das políticas de alto nível. Sistema auto-curável é aquele que pode detectar, analisar e consertar possíveis falhas nele mesmo, sejam estas no software ou no hardware. Sistemas com autoproteção são capazes de se defender automaticamente contra ataques maliciosos ou falhas em cascata. Sistemas auto-otimizados buscam continuamente oportunidades para melhorar seu próprio desempenho e eficiência.

Para construção de um sistema autonômico, é necessário primeiro a sua construção por elementos autonômicos, como mostra a Figura 1. Estes elementos são responsáveis por gerenciar seu próprio estado, seu comportamento e suas interações com o ambiente, que consiste em sinais, mensagens de outros elementos e o mundo externo. O comportamento interno de um elemento e sua relação com outros elementos serão guiados pelos objetivos que seu projetista nele tiver embutido, por outros elementos que possuam autoridade sobre ele, ou ainda por sub-contratos com elementos pares, tendo seu consentimento tácito ou

explícito. Esse elemento pode variar desde um recurso de software ou hardware, até sistemas legados, de forma que o administrador tenha capacidade de controlar e monitorar o sistema.

Portanto, sistemas auto-adaptativos são referenciados como aqueles capazes de mudar sua estrutura conforme necessário, mas isto traz grandes desafios de engenharia, expostos por Lemos R. *et al.* em [2]. Onde se destaca o ponto de vista da distribuição, levando-se em conta o esquema de controle de adaptação centralizado e descentralizado, ou seja, o processo de decisão a fim de resultar em adaptações feitas pelo sistema, onde o controle dessas adaptações pode ser resumido em quatro atividades: monitorar, analisar, planejar e executar (MAPE), que pode ser visto na Figura 1.

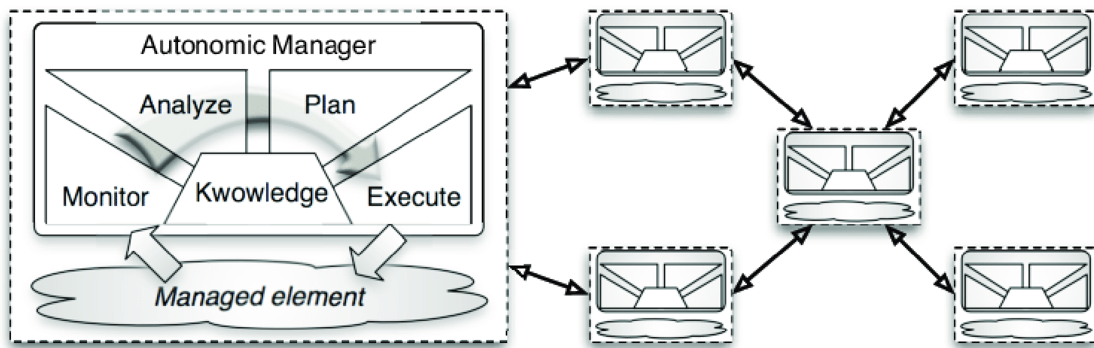

Figura 1: Estrutura de um elemento autonômico. Adaptado de [1].

Em particular, com a descentralização, refere-se a um tipo particular de controle em um sistema de software auto adaptável. Em um sistema descentralizado não existe um único componente que possui as informações completas do estado do sistema, e os processos tomam decisões de adaptação com base apenas em informações locais. Em um sistema auto adaptativo centralizado, por outro lado, as decisões sobre as adaptações são feitas por um único componente. Nessa perspectiva, o controle em um software autoadaptativo pode ser centralizado ou descentralizado, independente se o software gerenciado é distribuído. Na prática, no entanto, quando o software é implantado em um único processador, o controle de adaptação é geralmente centralizado. Do mesmo jeito, o controle descentralizado do sistema é esperado em cenários onde o sistema é distribuído. Com isso podem-se explorar diferentes abordagens para o controle da adaptação [2], como por exemplo o *master-slave*, observável na Figura 2. Dessa forma cria-se uma hierarquia entre os nós do seu sistema, onde o nó *master* é responsável pela análise e planejamento da adaptação, enquanto os nós *slaves* são encarregados de monitorar e fornecer as informações ao nó *master*, além de executar as adaptações passadas por ele.

2.2 Sistemas de Software Emergentes

Como já destacado, devido à crescente complexidade na elaboração de sistemas, existe uma necessidade de avanços nos estudos de sistemas auto-adaptativos. Contudo, a partir de

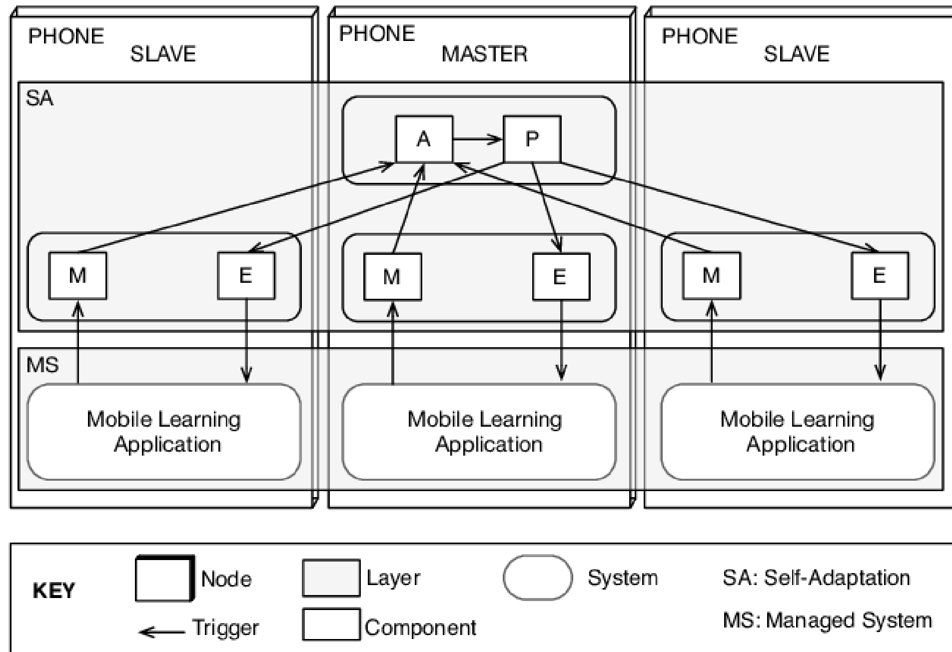

Figura 2: Controle de adaptação *Master-slave* [2].

uma análise feita em [3], pode-se concluir que ainda existem particularidades que impedem o avanço desses sistemas, dentre elas podem ser citadas:

- Alta dependência humana no processo de definição de regras de adaptação;
- Adaptação reduzida a apenas algumas partes do sistema;
- Incapacidade de evoluir a adaptação de forma autonômica, considerando-se fatores externos não mapeados.

Portanto, o conceito de Sistemas de Software Emergentes surge em razão de facilitar a criação desses sistemas auto-adaptativos. Ele consiste em construir sistemas a partir de pequenos componentes reutilizáveis, levando-se em conta os objetivos de alto-nível, eventos ou parâmetros fornecidos pelo sistema, ou usuário [3]. Desta forma o sistema pode moldar sua estrutura para melhor se adaptar aos fatores externos. Assim, o uso de componentes pequenos permite uma maior variação do sistema, facilitando a adaptação do mesmo, de forma que ele possa se reorganizar sem a necessidade de um *reboot*.

Com isso, o conceito de Sistemas de Softwares Emergentes foi posto em prática a partir do *framework* PAL, descrito em [3]. A sigla PAL vem de seus três módulos *Perception*, *Assembly* e *Learning*, como pode ser observado na Figura 3.

O módulo *Perception* é responsável por gerar e adicionar componentes *proxies* ao sistema a partir das composições do *Assembly*, de modo que eles monitoram a integridade do sistema

e o ambiente operacional. O módulo *Assembly* vasculha componentes do repositório, cria uma representação em memória de todas as composições arquitetônicas possíveis do sistema e fornece suporte para que, em tempo de execução, possam ser feitas mudanças na composição do mesmo. Finalmente o módulo *Learning* conduz todo o processo de composição do software autonômico por meio de abordagens de aprendizado de máquina por reforço.

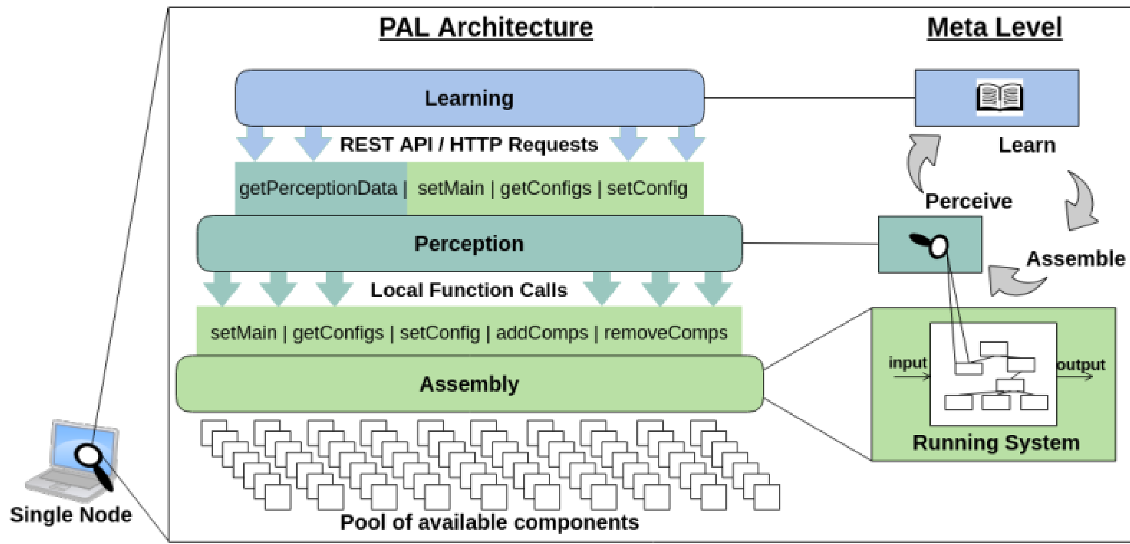

Figura 3: Arquitetura do *framework* PAL [3].

Conforme mencionado, um dos pilares do conceito de Sistemas de Software Emergentes são os modelos baseados em componentes. Por isto, o *framework* PAL, assim como as implementações deste projeto, utilizam a linguagem Dana [1]. Dana é uma linguagem de programação adaptativa, que nativamente fornece um modelo baseado em componentes através da definição dos componentes e suas interfaces [11]. Além de uma API com suporte a adaptação de componentes em tempo de execução. Utilizando então a linguagem Dana , o módulo *Assembly* provê seus próprios métodos para controlar o processo de adaptação de sistemas em tempo de execução, como os exemplos abaixo, retirados de [3].

- `String[] getConfigs()`: retorna uma lista de composições arquiteturais disponíveis em formato de *strings*;
- `char[] getConfig()`: retorna a composição arquitetural atual do sistema no formato de *string*;
- `bool setConfig(char configDesc[])`: altera a composição arquitetural do sistema em tempo execução, segundo a descrição passada por parâmetro;

[1] `http://www.projectdana.com/`

- `bool addComp(char compName[])`: adiciona um novo componente passado por parâmetro a lista de componentes disponíveis, aumentando então o número de composições arquitetônicas possíveis.

Por meio desses métodos o *Assembly* consegue detectar as mudanças de componentes que implementam uma mesma interface, assim como suas dependências de outras interfaces e seus componentes, e assim recursivamente até que seja feito o mapeamento de todas as composições possíveis da arquitetura do sistema, além de prover a adaptação entre elas em tempo de execução. Pode-se ver na Figura 4 uma ilustração do processo descrito acima.

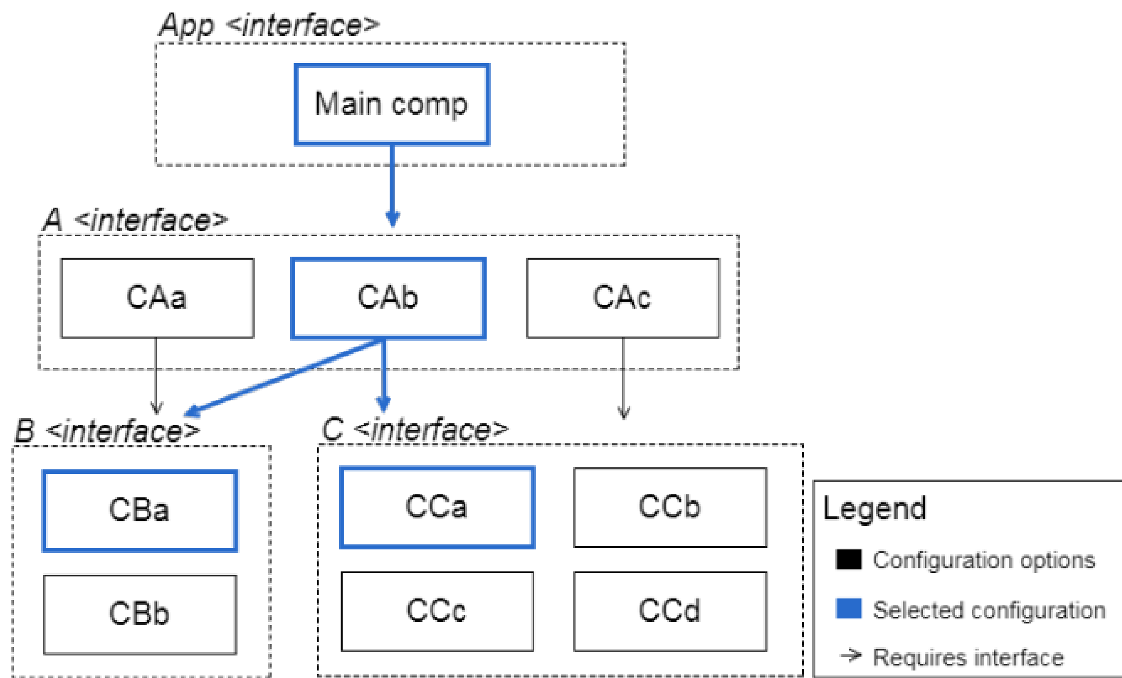

Figura 4: Exemplo de uma arquitetura genérica representada pelo módulo *Assembly*, resultante da execução da função `setMain()` do módulo *Assembly* [3]. As setas representam as dependências entre componentes e interfaces, além das possíveis variações de componentes para a composição arquitetural, e em azul está a composição definida pelo *Assembly*

2.3 Sistemas Auto-Distribuídos e Gerenciamento do Estado

Sistemas auto-distribuídos, classificados como uma subcategoria de sistemas emergentes, são construídos utilizando modelos baseados em componentes e também podem ser postos em prática a partir do framework PAL. Mas diferentemente dos sistemas de softwares emergentes, onde é feita apenas a troca de um componente pelo *Assembly*, nos sistemas auto-distribuídos o novo componente acaba sendo um *proxy* que implementa de forma transparente chamadas de procedimento remoto [8].

As plataformas de nuvem e borda, que proveem mecanismos de adaptação à nível de infraestrutura, são muito utilizadas por aplicações que demandam adaptação em tempo de execução. Entretanto, esses mecanismos de adaptação não oferecem nenhum suporte à gestão de estado a nível de aplicação[7]. Entende-se por estado o resultado de uma sequência de operações de escrita e leitura sobre os dados da aplicação. Com isso os desenvolvedores precisam retirar o estado da aplicação, e armazená-lo em um outro sistema, por exemplo um banco de dados. Um dos motivos dessa abordagem é a complexidade em gerir este estado quando ele se encontra distribuído e também a complexidade de manter a consistência dos dados, uma vez que é necessário mantê-los não somente em um, mas em múltiplos locais ao mesmo tempo. Com isso, a maneira de lidar com o estado permanece estática, dificultando a exploração de alternativas para aumentar o desempenho em diferentes condições de implementação [7].

Ao observar estes problemas, duas abordagens podem ser tomadas. Primeiramente, pode-se se priorizar a corretude dos dados, tornando o sistema menos performático. Em contrapartida, um sistema mais performático teria uma consistência de dados mais baixa, tendo que ceder em alguns aspectos como ordenação de escrita, entre outros [12].

Nos sistemas de software emergentes, é possível expor o estado durante o tempo de execução. Assim, durante a adaptação do sistema, pode-se extraí-lo de um componente e injetá-lo em um componente substituto. Para isto, existe a estratégia de *sharding*, que consiste em particionar o estado em diferentes subconjuntos do conjunto total, utilizando-se de alguma estratégia como por exemplo um *hash*. Esta abordagem é interessante quando o modelo de dados tem partes distintas que são mais independentes entre si, como um banco de dados.

A gestão de estado através de *sharding* é apenas uma das formas de lidar com a auto-distribuição de componentes com estado de forma transparente e consistente. Outras formas são abordadas por Oswaldo, *et al.* [9] que explora diferentes cenários de auto-distribuição, e mostra quando é favorável usar uma ou outra abordagem de consistência de estado

2.4 Computação na Nuvem

A computação na nuvem, semelhante ao comércio eletrônico, é uma das terminologias técnicas mais vagas da história, como destacado em [3]. Uma razão é que a computação na nuvem pode ser usada em diversos cenários e sistemas, outra é que a computação em nuvem é promovida por diversas empresas para alavancar seus negócios. Mas a computação na nuvem pode ser definida como um conjunto de serviços habilitados para rede, fornecendo infraestruturas de computação escaláveis, com garantia de qualidade de serviços, normalmente personalizadas, baratas e sob demanda, podendo ser acessadas de maneira simples e abrangente [5].

A computação na nuvem tem cinco características principais, sendo: recursos de computação em grande escala; alta escalabilidade e elasticidade; *pool* de recursos compartilhados (recursos virtualizados e físicos); escalonamento dinâmico de recursos; e propósito geral [3]. Além de todos os provedores possuírem e disponibilizarem os recursos em uma mesma plataforma, facilitando a conexão entre diversos serviços, bastando apenas o administrador escolher o provedor que melhor atenderá suas necessidades.

Um desses provedores é o *Google Cloud Plataform (GCP)*, criado em 2008 pela empresa *Google LLC* com o intuito de facilitar o desenvolvimento de aplicações com alta escalabilidade. Dentre os diversos serviços disponibilizados tem-se o *Google Kubernetes Engine (GKE)*, sendo uma forma simples de implantar, escalonar e gerenciar o *Kubernetes* de maneira automática. *Kubernetes* é um serviço de orquestração de contêineres, sendo contêineres imagens executáveis leves e portáveis que contém software e todas as suas dependências[2]. Para ser feita essa orquestração é necessário primeiro a construção de um *cluster*, definido como um conjunto de servidores de processamento, chamados nós, que executam aplicações conteinerizadas[2]. Para a criação desses nós é necessário primeiro a criação de *Deployments*, que podem ser descritos como objetos de API que gerenciam um ou mais aplicativos replicados, normalmente executando *Pods* sem estado local[2]. Um *Pod* como a própria documentação do *Kubernetes* descreve, é o menor e mais simples objeto *Kubernetes*. Um *Pod* representa um conjunto de contêineres em execução no seu *cluster*. Para que um *Pod* possa se comunicar com outro, ou com clientes fora do *cluster* é necessário fazer a configuração dos *Services*, que nada mais são que uma abstração para expor uma aplicação que está executando em um conjunto de *Pods* como um serviço de rede.

2.5 Computação na Borda

Um segundo conceito que vem crescendo nos últimos anos é a computação na borda, que vem como uma alternativa para a computação na nuvem, pois muitos aplicativos baseados na nuvem operam com um servidor central para processar os dados e as requisições geradas pelos dispositivos, como *smartphones*, *tablets* e *wearables*. Cada vez mais este modelo exige uma comunicação e infraestrutura computacional mais robusta, para conseguir atender todo esse volume. Então o conceito de computação na borda é baseada na movimentação de algumas dessas cargas em direção à borda da rede, dessa forma melhor aproveitando os recursos que atualmente são menos explorados, como estações-base, roteadores e *switches* [6]. Além de trazer uma latência de comunicação menor, devido ao processamento estar mais próximo do cliente.

Como já destacado, uma das grandes motivações e benefícios da computação na borda é a baixa latência para comunicação entre o cliente e o servidor, pois uma vez que o processamento é feito mais próximo do cliente, o tempo de resposta diminui consideravelmente [6], além disso o processamento que antes precisava ser feito no cliente, agora pode ser feito em outro dispositivo, dessa forma diminuindo o consumo de recursos do cliente [6]. Outros benefícios são a diminuição do consumo de energia, uma vez que os data center são alguns dos grandes consumidores de energia, e uma melhor maneira de lidar com o grande volume de dados, que vem crescendo diariamente, uma vez que antes era necessário aumentar fisicamente o espaço dos servidores, agora é possível utilizar os diversos aparelhos que já existem para distribuir essa carga [6].

[2] https://kubernetes.io/docs/reference/glossary/

3 Objetivos

Este projeto procura estudar os benefícios e os desafios de se adaptar sistemas auto-distribuídos com estado, de forma híbrida entre a borda e a nuvem, a fim de analisar o desempenho do sistema em diferentes cenários com determinados recursos e demandas, de tal forma que o estado seja gerido de maneira transparente em tempo de execução.

Dessa forma, busca-se discutir e fornecer respostas às seguintes questões:

1. É possível afirmar que o desempenho na borda, em comparação com a nuvem, é maior independente do cenário analisado?

2. Quais são os cenários em que há ganhos de desempenho do sistema na borda? E na nuvem?

Dada estas questões, visa-se explorar os cenários disponíveis pelo modelo híbrido na construção do sistema, a fim de analisar os resultados mais evidentes, independente se eles são positivos ou negativos.

4 Metodologia

A fim de responder às questões levantadas nos objetivos, foi adotado para este projeto uma metodologia empírica para o seu desenvolvimento. A partir de um sistema já implementado foram utilizadas um conjunto de variações do mesmo, além de variações na arquitetura híbrida do sistema, com a ajuda do framework de auto adaptação PAL, para avaliarmos o desempenho em diferentes cenários testados. Para guiar os testes foram levantados e mapeados os parâmetros do sistema com o intuito de facilitar a construção dos cenários, além de sustentar os resultados adquiridos. Portanto foram levantados três características para avaliar os resultados:

i) *Levantamento de hipóteses:* parâmetro para ser o ponto de partida quanto a avaliação do sistema (Exemplo: O desempenho da requisição GET é mais eficaz numa composição na borda);

ii) *Requisitos do sistema:* parâmetros quanto a composição do sistema que será avaliado (Exemplo: Distribuído e na borda);

iii) *Carga de trabalho:* parâmetros quanto à carga de trabalho, a qual o sistema está exposto(Exemplo: Taxa de requisições constante).

A metodologia empírica visa testar a validade de teorias e hipóteses em um contexto de experiência [10]. A partir do levantamento de hipóteses pode-se ter uma direção para a condução da avaliação do sistema, uma vez que agora tem-se precisão sobre como o sistema deve se comportar em determinado cenário. Uma vez que foi feito o levantamento de hipóteses, agora pode-se detalhar os requisitos do sistema, ou seja, quais serão as operações realizadas com o sistema em termos de processamento, pois dessa forma pode-se observar o impacto das operações de distribuição no desempenho do sistema. Além da distribuição do

sistema, existe também a variação no tipo de operação que será feita com o sistema, uma vez que existem operações mais custosas em desempenho que outras. Por fim a carga de trabalho. Com ela é possível mapear o volume de recursos que será exigido do sistema, e conforme for aumentando esse volume, pode-se ver o desempenho variando. Além da carga pode-se variar o padrão da carga que o sistema pode estar exposto, por exemplo variando o percentual das operações de escrita e leitura do sistema.

A partir desses três pontos mencionados acima, foi possível determinar as características do sistema que guiaram as avaliações a serem feitas sobre o mesmo.

Primeiro para o levantamento de hipóteses, foi feita uma análise inicial sobre o comportamento dos diferentes tipos de requisições e composições do sistema, a fim de mapear os resultados esperados para determinados cenários, e assim tê-los como base para a realização dos testes. A partir dessas hipóteses foi feito o levantamento dos parâmetros do sistema, dessa forma pode-se mapear as diferentes composições do sistema, e construir uma variação de cenários para avaliar cada composição. Quanto à carga de trabalho, o sistema foi exposto a um volume de carga constante, dessa forma pode-se avaliar o desempenho do sistema conforme o volume crescia em uma taxa linear de crescimento. Em relação ao padrão de carga que o sistema foi testado, foi variado em termos do tipo de operação ou conjunto de operações requisitadas ao sistema.

Por fim é importante ressaltar que foram feitas simplificações na implementação,ou seja, nem todas as variações possíveis foram testadas, e o uso de uma mesma rede local como ambiente de execução. No entanto, a metodologia adotada mantém características de sistemas reais, em que todos os testes foram feitos em ambientes reais sem o uso de simulação.

5 Abordagem

Para que fosse estudado uma nova abordagem de avaliação do desempenho de um sistema auto-distribuído, foi decidido que ao em vez de implementar o sistema apenas na borda [9], ou apenas na nuvem[3], seria utilizado uma infraestrutura híbrida para avaliar os objetivos levantados, capaz de migrar totalmente de uma composição para a outra. Dessa forma foi possível analisar o desempenho do sistema tanto na borda quanto na nuvem. Na borda foi construído um sistema com a ajuda de um *framework* de auto-distribuição, capaz de adaptação em tempo de execução, como já mencionado no Referencial Teórico. Esse sistema possui duas camadas, uma de adaptação, também chamada de *meta-level* e uma de aplicação. Ambas as camadas foram implementadas utilizando a linguagem Dana [11], que fornece um modelo baseado em componentes. No *meta-level* é onde ficam os componentes responsáveis pela distribuição e adaptação do sistema.

Além disso, na nuvem foi implementado um *cluster* utilizando o serviço do *Google Kubernetes Engine (GKE)*, por meio de *scripts* para automatizar essa construção, nesses *scripts* havia o detalhamento dos *pods* com todas as suas características, além da camada de serviço para fazer a comunicação entre os *pods* e o balanceamento de carga das chamadas entre os

[3] `https://github.com/RSilvaDias/SCTL-PFG`

mesmos. Por fim foi feita a configuração dos *IP's* públicos para ser feita a comunicação entre os *pods* e a borda.

Para melhor detalhar a infraestrutura, pode-se dividi-la em duas etapas, a na borda e a na nuvem.

5.1 Infraestrutura na Borda

O ponto inicial do sistema no *meta-level* é o componente distribuidor *Distributor*, é ele que interpreta os comandos para a realizar a adaptação e inicia a aplicação. Dependendo do comando recebido o *Distributor* adapta o componente da aplicação, este possui o estado, dessa forma fazendo a ponte para a distribuição do estado, começando com ele na borda para então distribuí-los por um *proxy*, e então é feita a distribuição do estado para os componentes na nuvem. Isso só é possível graças à capacidade do módulo *Assembly* em obter todas as composições possíveis, dentre as arquiteturas disponíveis do sistema, dessa forma podendo mudar entre as composições a fim de encontrar a melhor para cada cenário, isso tudo em tempo de execução como pode ser ilustrado a Figura 5. E graças também a linguagem Dana [11] que permite a componentização de todo o sistema, dessa forma as aplicações podem adaptar seus componentes em tempo de execução, alterando seu comportamento sem ser necessária uma pausa no serviço.

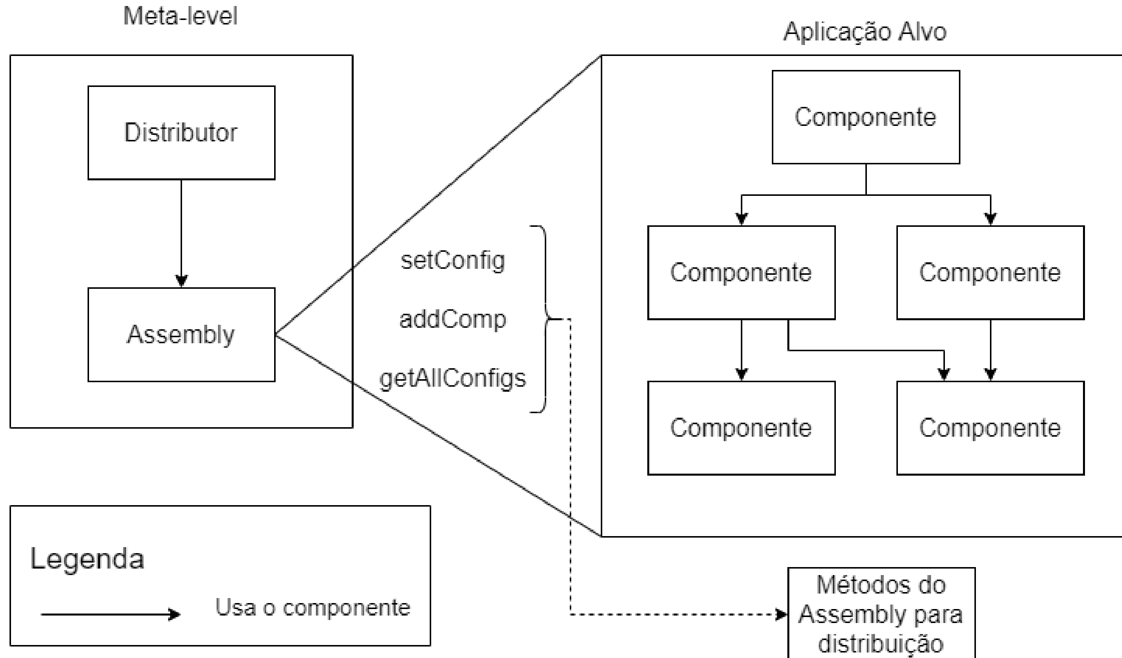

Figura 5: Os componentes do *meta-level* para sistemas auto-distribuídos. Com a utilização do *Assembly*, o *Distributor* consegue construir, adaptar e executar uma aplicação.

Durante a adaptação, o componente *proxy* tem como responsabilidade especificar as formas de lidar com o estado, mantendo sua consistência e distribuindo-o entre os nós remotos, como mencionado no Referencial Teórico. Dessa forma foi analisado um modelo de consistência de dados o *ShardingProxy*. O *ShardingProxy*, é um modelo menos restritivo, que faz um particionamento dos dados a partir de uma função *hash*. Dessa forma, não se pode afirmar nada sobre a composição do estado em cada nó, devido à intercalação das operações de escrita, que acabam sendo direcionadas para um nó ou outro dependo do valor calculado na função.

Por fim o *proxy* também é responsável pela implementação dos métodos do componente adaptado, já que ambos possuem a mesma interface. Porém esses métodos fazem as chamadas de procedimentos apenas para os nós remotos, sempre mantendo a consistência dos dados. Dessa forma, o processamento das requisições é feito remotamente na nuvem. Portanto, deixando com que esses nós remotos façam o processamento, teremos uma liberação de recursos na borda, uma vez que a borda só ficará encarregada de receber o retorno.

Todo o diagrama de distribuição e adaptação pode ser ilustrado pela Figura 6.

5.2 Infraestrutura na Nuvem

Para estudar o desempenho do sistema foi proposto a criação dos componentes de adaptação denominados *Remote Distributor*. Neles é feito o processamento na nuvem, como pode ser ilustrado na Figura 6, onde eles são os componentes dentro dos *pods*, e assim se pode analisar o desempenho na nuvem e sua eficácia, uma vez que nela não existe limite de recursos. Além de permitir o escalonamento tanto vertical, aumentando os recursos do nó, quanto horizontal, aumentando o número de nós. Porém perdendo em latência com relação a um sistema na borda.

Visando o estudo a partir do modelo de auto-distribuição na nuvem foi estudado a implementação[4] feita onde é feita a construção da mesma base do sistema de auto-distribuição que foi utilizado neste projeto. Nele foi feita a implementação do *Distributor* em um *cluster*, e uma nova camada chamada *ServerCTL*, que é um serviço implementado em Python que faz a interação entre o *Distributor* e o *Kubernetes* para criação e remoção de pods que irão executar o sistema. A partir dessa camada foi possível fazer o controle automatizado de criação de *pods* com *Remote Distributor* no *cluster*. Isso devido dos recursos provido pelo sistema do *GKE*, onde o *ServerCTL* funciona como uma ponte entre a aplicação e o *Kubernetes (K8s)*, onde a aplicação recebe o pedido de criação de um determinado número de *pods* e o *ServerCTL* faz a chamada para o *Kubernetes*, então foi criado *pods* no *cluster*, e para ser feita a comunicação entre os *pods*, é exposto um *IP* público. Dessa forma tendo todo o sistema implementado num ambiente elástico.

A partir desse estudo, e visando estudar os benefícios da borda, foram pensados infraestruturas híbridas. O primeiro cenário pensado foi, do *Distributor* na borda, enquanto é feita a sua inicialização do *Remote Distributor* dentro dos contêineres na nuvem. O *Remote Distributor* espera a finalização da adaptação, e então fica responsável por receber o *proxy* da borda, além do estado já devidamente adaptado. Já com o estado, o *Remote Distributor*

[4] `https://github.com/RSilvaDias/SCTL-PFG`

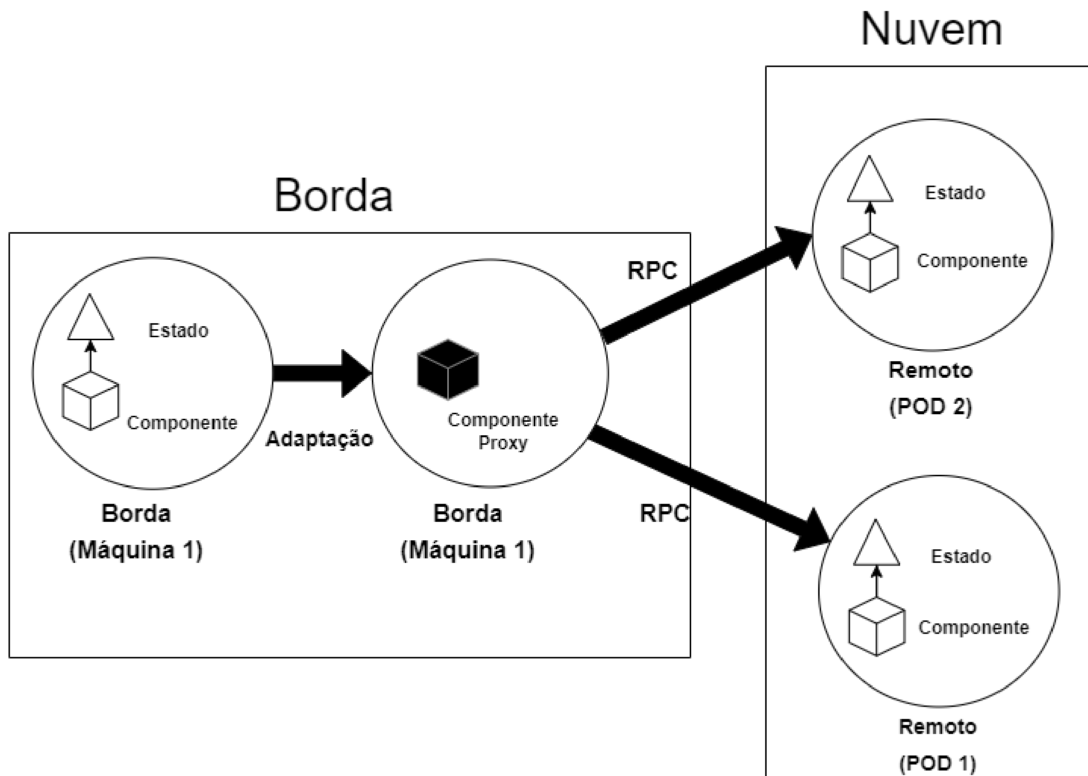

Figura 6: Diagrama dos processos de distribuição e adaptação do sistema, com estado. O componente da borda, tem seu estado compartilhado com os componentes remotos, por meio de um *proxy* e este realiza as chamadas para o componente remoto.

iniciará o *proxy* remoto, e este ficará responsável pelo processamento das requisições que o *proxy* local antes era encarregado, realizando as operações de maneira remota.

Um segundo cenário pensado foi o caminho inverso, onde o *Distributor* seria implementado na dentro de um *cluster* na nuvem, e os *Remote Distributor* na borda. Dessa forma poderia ser feita uma comparação entre as duas infraestruturas híbridas, e analisar o desempenho em tempo de resposta de requisição e consumo de recursos em todas as infraestruturas mencionadas, dessa forma com o sistema poder ser adaptado em tempo de execução, a aplicação poder ser adaptada para uma melhor composição dependendo do cenário.

6 Estudo de Caso

Como estudo de caso, foi desenvolvida uma aplicação [9] tendo como estado uma lista de números inteiros, dessa forma seria possível testar a gestão do estado e a sua distribuição. A aplicação é formada por um servidor que recebe requisições HTTP. A primeira requisição, do tipo POST, tem a finalidade de inserir um dado na lista, além dela foi analisada duas requisições do tipo GET, onde a primeira retorna todos os elementos da lista, enquanto a segunda retorna o elemento requisitado. O servidor é iniciado na borda, dentro de uma máquina local, até que seja feita alguma adaptação pelo *Distributor*, que faz então a troca do componente que contém a lista por um *proxy*. O *proxy* utilizado na distribuição do estado foi *ShardingProxy*, que faz um particionamento dos dados a partir de uma função *hash*.A estrutura da aplicação alvo pode-se ilustrar na Figura 7.

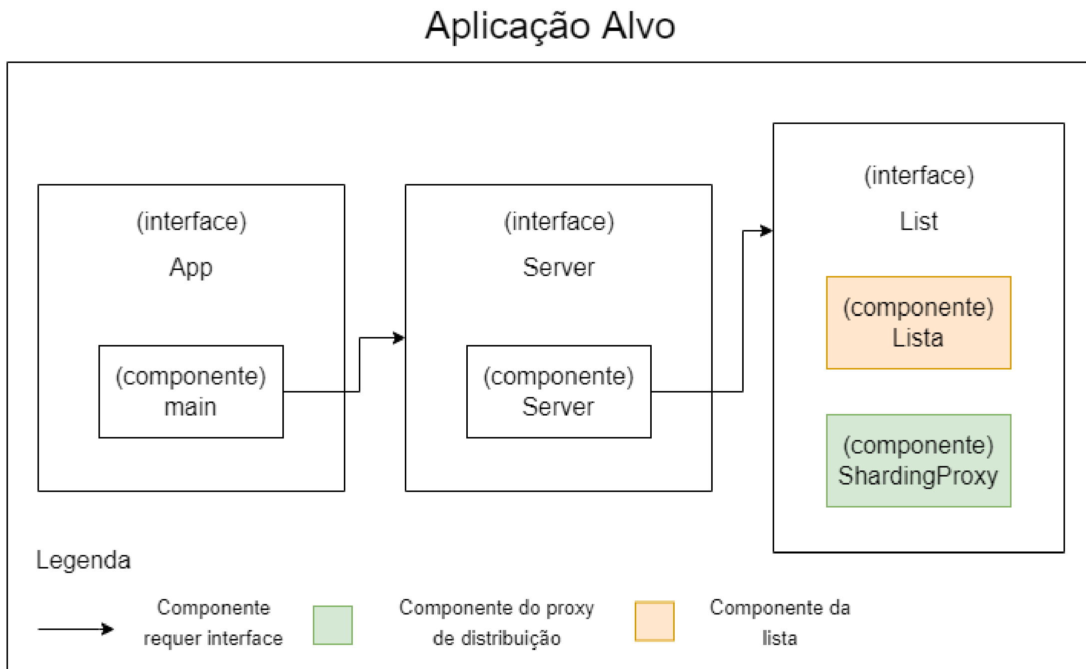

Figura 7: As interfaces com seus devidos componentes. Portanto, nas interfaces *App*, *Server* e *List*, têm-se os componentes que as implementam, respectivamente: *main*, *Server*, *List* e os *proxy* de distribuição *ShardingProxy*.

Dos dois cenários retratados na abordagem, foi analisado apenas o primeiro, onde foi feita a implementação do *Distributor* na borda, e a criação dos *Remote Distributor* na nuvem. Para isso, foi feita a configuração do *cluster* no sistema do *Google Kubernetes Engine (GKE)*, onde é preciso passar os dados de configuração dos contêineres, como número de *CPU's* e quantidade de memória que cada contêiner deve possuir. Depois com o *cluster* já criado,

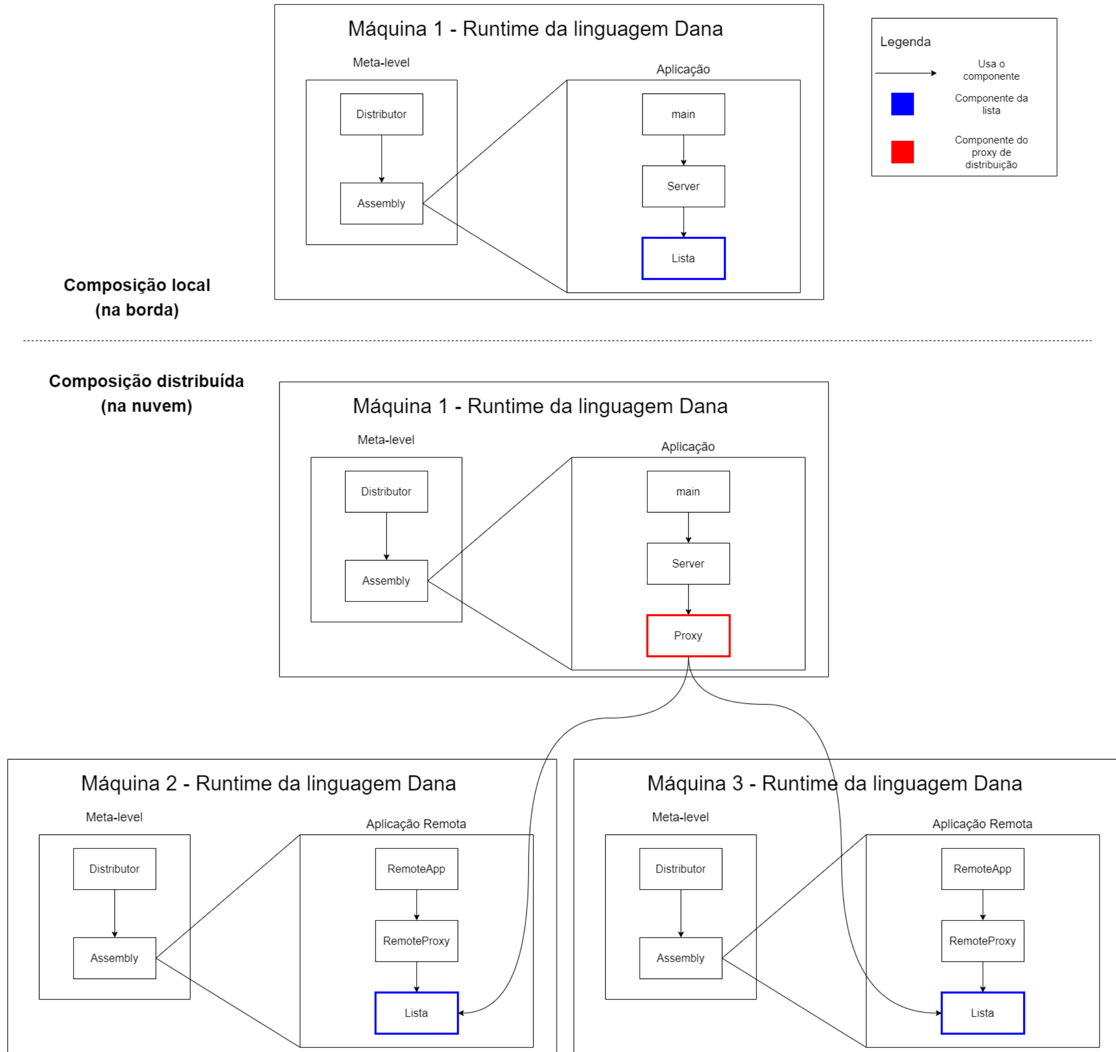

Figura 8: Arquitetura do sistema tanto na composição apenas na borda quanto distribuída, entre borda e nuvem. Na execução na borda tem-se o *meta-level* atuando na aplicação alvo, enquanto na execução distribuída o mesmo se encontra atuando na aplicação remota em duas máquinas distintas, todas elas rodando dentro da *runtime* de Dana.

foi feita a criação dos dois *Deployments* por meio de *scripts*, neles existe o detalhamento da configuração dos contêineres, com imagens do *Remote Distributor*, portanto ao final desse processo, havia um *Remote Distributor* executando dentro de cada um dos dois *Pods*. Após a criação dos *Deployments*, foi necessário criar a configuração dos *Services*, ou seja, a configuração para fazer a comunicação entre os componentes na borda e os *Pods* que estão na nuvem, dessa forma foi exposto um *IP* público de cada *Pod*, com uma configuração com

LoadBalancer. Com esses *IP's* foi possível comunicar o *Distributor* que estava na borda, com os *Remote Distributors* que estavam dentro do *Pod*.

Para a execução da aplicação, foram utilizadas quatro variações da *List*, uma interface nativa da linguagem Dana. Cada uma delas com suas devidas características em relação ao processamento dos elementos na lista. Podemos mapear as operações de cada variação em operações de leitura e escrita. A primeira variação é a *Constant*, a qual não possui um tempo de processamento atrelado às operações, a segunda é o *Readn*, onde foi atrelado um tempo de processamento na operação de leitura, esse processamento é uma ordenação na lista através do algoritmo do *Heapsort*, acrescentando um tempo de processamento de complexidade $O(n \lg n)$ no tempo total, onde n é o tamanho da lista. A terceira variação é o *Writen*, que possui um tempo de processamento acrescentado as operações de escrita, onde deve se manter uma propriedade de *Max-Heap*, de forma que ao adicionar um novo elemento na lista, a lista deve ter a propriedade mantida, acrescentando $O(\lg n)$ no tempo de processamento. Por último temos a variação do *Readn-Writen* onde em ambas é acrescentado ao tempo de processamento das implementações citadas acima, tanto a ordenação com o *Heapsort*, quanto a propriedade de *Max-Heap*.

Por fim, na Figura 8, é possível observar toda arquitetura do sistema, demonstrando a atuação do *meta-level* em relação às aplicações alvo, tanto em uma composição apenas na borda quanto em uma composição distribuída utilizando dois nós remotos, na nuvem por exemplo.

Uma vez que foi definida a arquitetura do sistema, o próximo passo foi, levantar as suposições do comportamento do sistema e do desempenho esperado na realização dos testes. Portanto com as hipóteses criadas foi possível mapear os parâmetros do sistema e os cenários de testes para pôr à prova as hipóteses levantadas. Foi proposto uma bateria de testes com cada requisição separada, realizando primeiramente os testes na borda o verificando o tempo de resposta, e após a distribuição, refazer os testes verificando o tempo de resposta agora com o sistema distribuído na nuvem. E a partir dos dados coletados validar a corretude das hipóteses levantadas. Após isso, foi proposto também a execução de alguns testes na borda com os recursos limitados, para ver o comportamento do desempenho quando há uma limitação de recursos. Por fim, foram realizados mais alguns testes agora com operações de leitura e escrita em conjunto, para verificar cenários reais de um serviço.

7 Avaliação

Esta seção apresenta os resultados obtidos nos testes de distribuição do sistema na borda e na nuvem, dando visibilidade para o desempenho do mesmo nos dois cenários e buscando responder questões levantadas na sessão de Objetivos deste projeto. Sendo essas, as questões:

1. É possível afirmar que o desempenho na borda, em comparação com a nuvem, é maior independente do cenário analisado?

2. Quais são os cenários em que há ganhos de desempenho do sistema na borda? E na nuvem?

Para isto, os casos de teste foram separados em diversas categorias, listadas aqui para facilitar a visualização:

1. Duas opções de composição do sistema Dana: *Local* e *Sharding*, o primeiro representando a computação na borda, e o segundo representando a distribuição do sistema na nuvem.

2. Duas opções de requisição: leitura (GET) e escrita (POST), utilizando as operações `getContents()` ou `getIndex()` para o primeiro caso, e a operação de `add()` para o segundo.

3. Quatro opções de implementação para a interface da classe List de Dana, contando com diferentes complexidades de tempo de processamento e uso de recursos para a leitura e escrita cada uma.

4. Possibilidade de limitar os recursos de processamento na borda para simular um poder de computação inferior da borda em relação à nuvem.

Partindo dessas premissas, foi gerado um cliente em Python capaz de realizar diversas chamadas para o servidor dentro do *Distributor*, e simultaneamente calcular o tempo de resposta de cada requisição, assim possibilitando a geração de gráficos para comparar os resultados de cada caso de teste. Além disso, também foi necessário fazer alterações na composição do sistema na borda e na nuvem para cada teste.

7.1 Avaliando a inserção na lista

No primeiro teste, buscava-se a resposta sobre qual composição, *Local* ou *Sharding*, seria melhor no cenário de escrita com complexidade $O(\lg n)$, quando é mantida a propriedade de *Max-Heap* na lista durante a inserção de elementos. Para isto, iniciou-se o processo local sem distribuição utilizando 100% da capacidade de processamento da máquina utilizada. Em seguida, no cliente, foram efetuadas um total de 3000 requisições sequenciais para o *endpoint* de adição de elementos na lista, salvando o tempo de resposta de cada requisição e guardando os resultados em um arquivo CSV para ser utilizado futuramente.

Finalizando esse teste, a lista foi apagada do processo local, ficando com 0 elementos, então foi feita a distribuição do processo para os *Remote Distributors* na nuvem através do método de *Sharding*. Em seguida, com o sistema rodando na nuvem, realizamos o mesmo teste através do cliente em Python, porém agora todos os itens que eram adicionados na lista estavam sendo distribuídos através de uma função *hash* para um dos dois *Pods* na nuvem.

Com os dados coletados, foi feita uma regressão linear em cada composição para melhor visualização da curva do tempo de resposta em relação ao número de itens na lista. Pode-se ver na Figura 9, que, quando a lista está com poucos itens, o tempo de processamento da inserção de novos itens ($O(\lg n)$) se torna irrelevante perto do tempo de latência da comunicação entre cliente-servidor quando o servidor está na nuvem. Porém conforme mais itens são adicionados, o tempo de processamento de cada inserção começa a se tornar relevante, até atingir um ponto em que o tempo de adição na composição *Local* ultrapassa o tempo

de adição em *Sharding*. Indicando que a partir de um certo tamanho da lista, o tempo de comunicação com a nuvem se torna menos relevante que o tempo de processamento na máquina local. Podemos ver esse ponto de inversão a partir de uma lista com aproximadamente 2150 itens, como pode ser observado na Figura 9 no ponto em que as duas curvas se tocam.

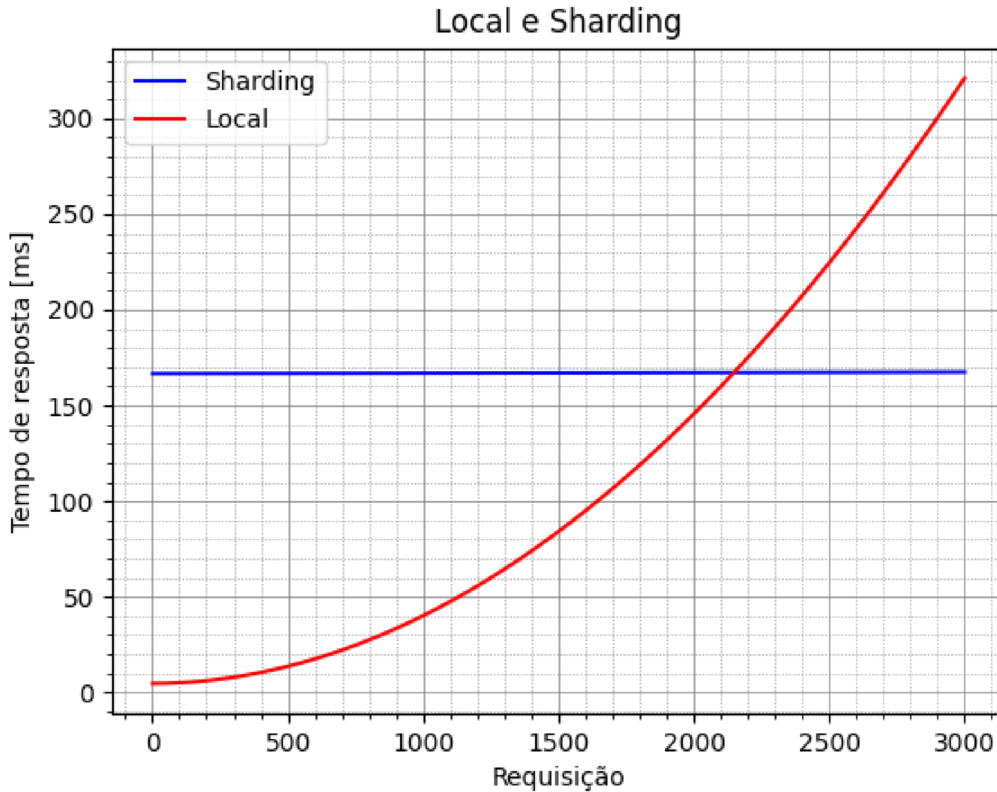

Figura 9: Tempo de resposta da adição de elementos na lista por número da requisição.

É válido mencionar também, que as máquinas distribuídas na nuvem sempre conterão um estado menor quando comparado com o processo rodando local sem distribuição, assim observa-se que o tempo de processamento será menor em cada *Remote Distributor*, já que o elemento sempre será adicionado em uma lista repartida e contém menos elementos.

7.1.1 Utilizando 50% de processamento local

Com a intenção de replicar um caso mais próximo do comum, onde o processamento na borda tem menos poder de processamento que na nuvem, foi executado o mesmo processo local anterior, porém com o comando que permite o processo utilizar somente 50% da *CPU* da máquina. O resultado obtido foi como o esperado. Com menos capacidade de

processamento, o tempo de resposta cresceu ainda mais abruptamente no ambiente local, chegando a ultrapassar o tempo de resposta na nuvem com apenas 1500 itens na lista, como pode-se observar na Figura 10.

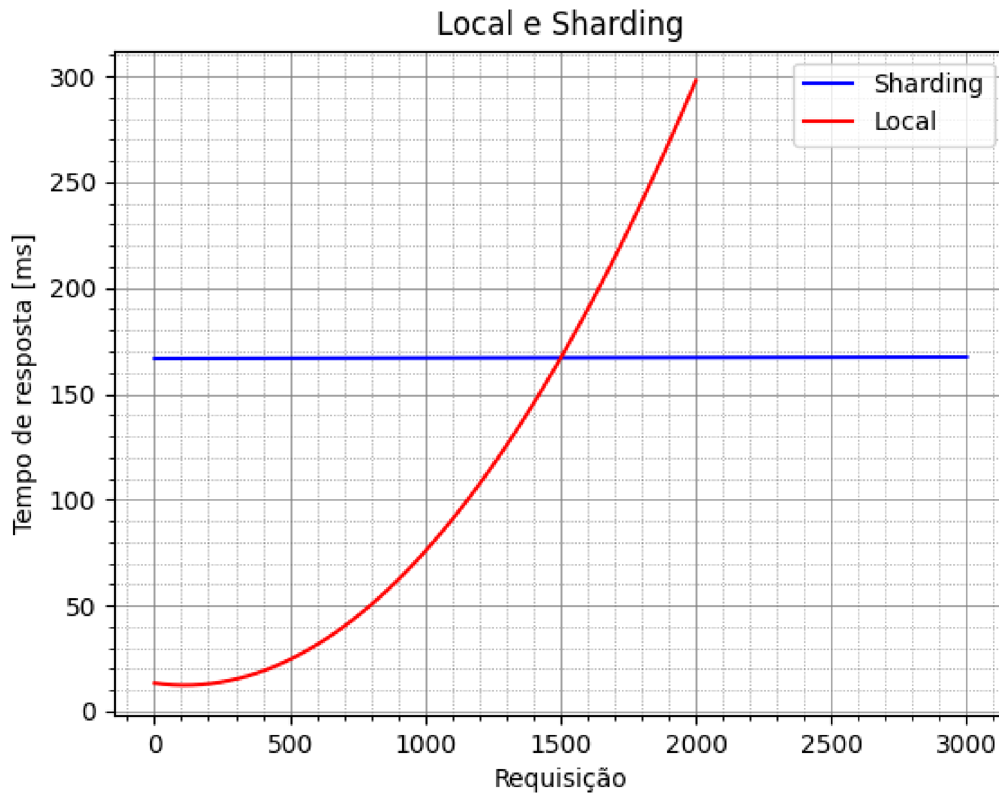

Figura 10: Tempo de resposta da adição de elementos na lista por número da requisição com processo Local rodando com 50% da CPU.

7.1.2 Avaliando inserção com outras implementações da lista

Seguindo, foram feitos testes com outras implementações da classe *List* de Dana. Para a implementação com tempo de processamento $O(n \lg n)$ na operação de leitura, foi obtido um resultado semelhante à implementação anterior. Já no teste da implementação de leitura e escrita com tempo constante, foram obtidas diferenças mais expressivas. No caso local as requisições de leitura foram extremamente rápidas em comparação com o *Sharding* na nuvem, por conta da latência de comunicação com o servidor na nuvem.

7.2 Avaliando casos de inserção e leitura

Para começar os testes com as requisições de leitura, foi primeiramente usada a chamada da função `getContents()`, que já estava implementada no projeto em Dana. Porém, observou-se que essa função não se aproveitava da particularidade de distribuição do *Sharding*, pois para coletar todos os itens, era necessário fazer o processamento nos dois *Remote Distributors* que estavam na nuvem. Logo o tempo de resposta acabava sempre sendo maior que o tempo de resposta local, independente do tamanho da lista.

Para se aproveitar do *Sharding*, foi implementada uma nova função na classe *List*, que retornava somente o item pedido na *request*, dando utilidade para o *Sharding*, uma vez que o item buscado também se submetia ao uso da função *hash* para decidir em qual *Remote Distributor* seria buscado. Assim consegue-se atingir um tempo de resposta que cresce conforme os primeiros experimentos de inserção na borda e na nuvem.

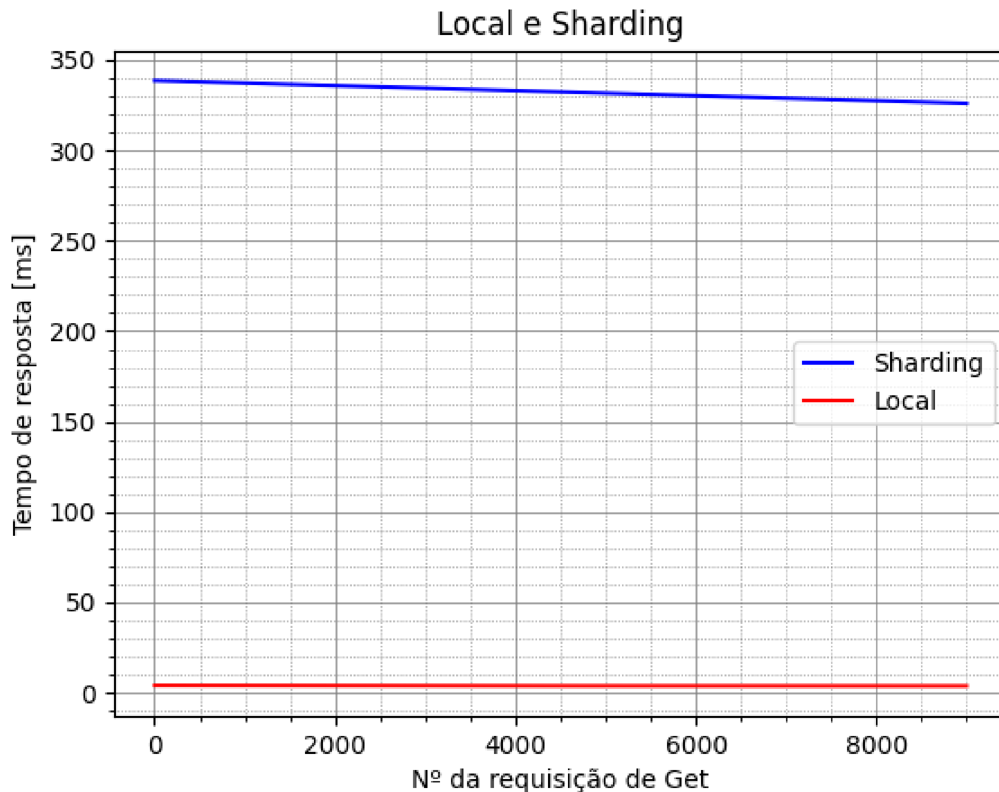

Figura 11: Tempo de resposta da leitura de elementos na lista por número da requisição.

A partir dessa nova função de leitura, foram iniciados os próximos testes, agora fazendo requisições de inserção e leitura intercalados, dessa forma a lista crescia ao longo do teste e era possível ver os resultados da leitura com diferentes tamanhos . Foi decidido que as

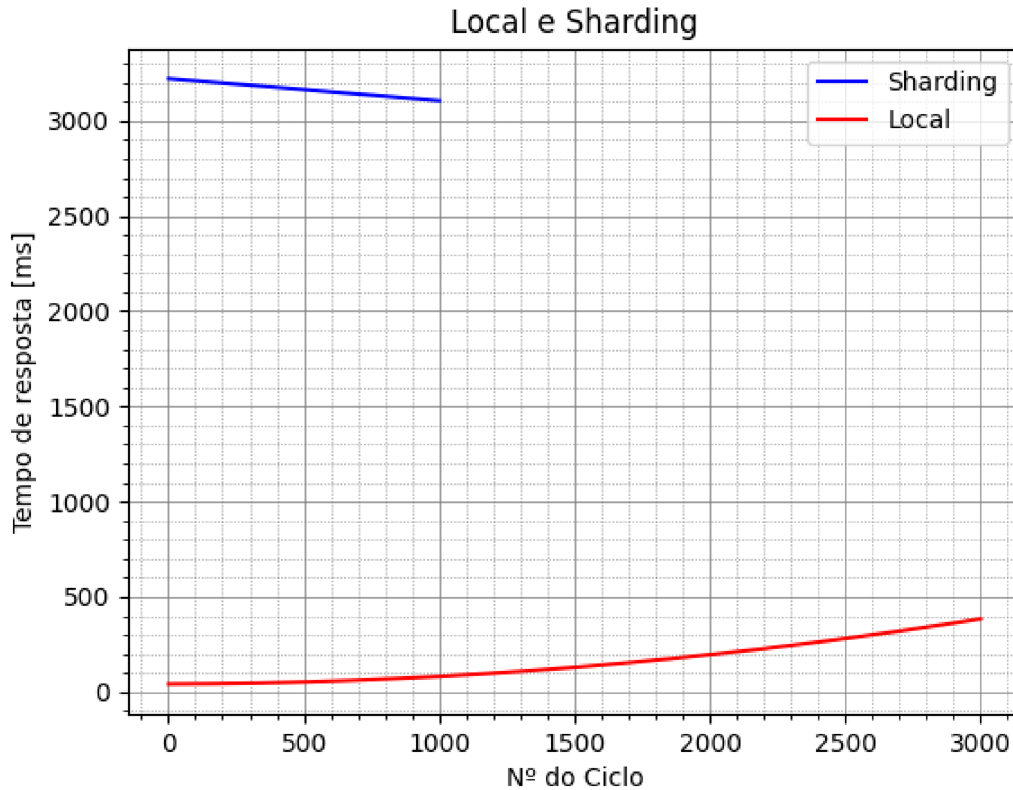

Figura 12: Tempo de resposta do ciclo (1 adição e 9 leituras) de elementos na lista por número do ciclo.

requisições seguiram uma proporção de uma inserção para nove leituras.

Como pode-se ver na Figura 11, as requisições de leitura local foram muito rápidas em comparação com o resto, na média não ultrapassando mais de 10 milissegundos e permanecendo quase constante. Porém quando observamos no todo, as requisições de inserção fizeram com que cada ciclo de 10 requisições aumentasse em uma curva polinomial como visto anteriormente no teste de inserção. Gerando o seguinte gráfico da Figura 12.

Além disso, é bom destacar que a curva da composição em *Sharding* termina antes que a curva local na Figura 12, isso se dá pelo fato de que os testes na nuvem tinham um tempo de execução elevado, por conta do alto tempo de latência de cada requisição, e como cada ciclo tem 10 requisições não foi possível coletar os dados de um total de 3000 ciclos, sendo enfim rodados somente 1000 ciclos na nuvem. Entretanto é possível notar que a curva de *Sharding* é um pouco inclinada para baixo, o que pode ter sido resultado de alguma variação na velocidade de conexão da internet durante o teste, já que o mesmo durou em torno de uma hora.

Por fim, pode-se observar que o comportamento não foi tão semelhante ao esperado neste teste, uma vez que a intenção era de ver um ponto onde a distribuição na nuvem seria mais eficiente que a distribuição local. Isso aconteceu pois o tempo de resposta médio da requisição de leitura era muito pequeno na borda, o que criava uma diferença considerável quando comparado com o tempo de resposta na nuvem, com alta latência. O que leva ao entendimento de que o tempo de latência de comunicação com a nuvem é um grande detrator quando se quer operar nela, e o cenário de uso mais conveniente seria quando existe um alto maior nível de processamento na borda, e casos onde o processamento se torna menor conforme o estado é repartido em partes menores, no caso deste projeto, a lista.

8 Trabalhos Futuros

Nesta seção, são discutidas as oportunidades que devido às limitações do nosso projeto, podem ser feitas como trabalhos futuros. Primeiro ponto é o escopo reduzido de requisições, para tornar os resultados mais próximos do ambiente real, é possível criar uma variação maior de requisições HTTP, como PUT ou DELETE, ou algumas variações das implementações de POST e GET. Além da variação de requisições, um segundo ponto que pode ser abordado em um trabalho futuro é construir a arquitetura híbrida no caminho inverso, como explicado na abordagem, onde o *Distributor* é implementado na nuvem, enquanto os *Remote Distributors* na borda, dessa forma estudando outras composições de arquitetura, e com isso encontrando novas composições com novos resultados.

Outro ponto que pode ser explorado, é a variação na quantidade de *Remote Distributors*, uma vez que foi estudado apenas os resultados com dois deles, podendo então melhorar o desempenho da distribuição já que o estado estará repartido em volumes menores. Além do escalonamento horizontal que poderia ser feito com a aplicação, outra abordagem que poderia ser implementada é um de carregamento de dados *lazy*, uma vez que todas as requisições têm um carregamento de dados *eager*, *eager* é quando todos os dados são retornados de uma só vez independente se ele será utilizado ou não, enquanto o carregamento de dados *lazy* faz o carregamento *on-demand*, em outras palavras os dados são carregados apenas quando eles são solicitados, gerando dessa forma uma otimização no processamento do estado.

Ainda nessa linha de raciocínio, é possível a criação de um serviço intermediário para a automação da criação de estruturas no ambiente escalável da nuvem, podendo então criar um número n de *Remote Distributor*, ou qualquer outro componente solicitado, de uma forma menos mecânica. Além de um estudo para melhor usufruir de uma arquitetura híbrida é o paralelismo do processamento, uma vez que até o momento esse processamento é feito sequencialmente em cada distribuição, limitando o potencial de desempenho que a composição na nuvem pode atingir.

Por fim, uma abordagem que não foi explorada dentro deste projeto, é a construção de uma Inteligência Artificial para analisar em conjunto com sistema na borda, o direcionamento do estado para locais onde ele seja mais requisitado, ou seja, poder analisar a partir da quantidade de requisições feita em uma determinada localização, trazer para borda perto desses clientes, os componentes e/ou estados mais utilizados por eles, diminuindo assim o

tempo de resposta.

9 Conclusões

Neste projeto foram estudados conceitos de sistemas auto-adaptativos, computação autonômica e Sistemas de Software Emergentes, além de conceitos sobre computação na borda e computação na nuvem. A partir desses conceitos, o projeto propôs a exploração de um sistema auto-adaptativo em uma estrutura híbrida de borda e nuvem. Dessa forma os experimentos realizados tiveram como objetivo analisar o desempenho do sistema em termos de tempo de resposta em cenários e composições variadas, a fim de encontrar uma melhor composição para cada cenário. E com isso, foi possível obter resultados preliminares onde a adaptação pode trazer benefícios.

Estes resultados mostram que em termos de tempo de resposta, o desempenho na borda é majoritariamente melhor. Portanto trazer os componentes, e em conjunto com eles, o estado para a borda tem em sua maioria um ganho de performance. Porém os resultados também mostraram que, uma vez que o processamento e os recursos na borda começam a ser disputados por outras aplicações, ou quando o estado torna-se tão grande a ponto de existir uma disputa por recurso, fazer a distribuição desses componentes e/ou estado para a nuvem traz um ganho de desempenho. Embora a latência tenha um grande peso no tempo de resposta, o mesmo torna-se inferior ao tempo de processamento devido à disputa de recursos.

Por fim, devido ao fato dos experimentos terem sido feitos sobre cenários específicos, ainda existe um conjunto de cenários que podem ser explorados. Como já mencionado, trabalhos futuros podem ser feitos em torno da arquitetura híbrida feita neste projeto, ou estudando novas composições de infraestrutura. Além de um estudo de modelos de aprendizado de máquina a fim de melhorar a adaptação do sistema. No entanto, o projeto atingiu seu objetivo de analisar os benefícios e desafios de um sistema auto-distribuído em uma infraestrutura híbrida, uma vez que foi possível determinar as melhores composições em relação ao seu desempenho em tempo de resposta, para os cenários estudados.

Referências

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

[2] de Lemos R. *et al.* (2013) *Software 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] RODRIGUES FILHO, R.; PORTER, BARRY ; COSTA, F. M. ; SENE JUNIOR, I. G. . Emergent Software Systems: Theory and Practice. In: José Ferreira de Rezende; Kleber Vieira Cardoso; Pedro Frosi Rosa; Flávio de Oliveira Silva;. (Org.). Minicursos do

XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.. 1ed.: , 2021, v. 1, p. 1-50 https://robertovrf.github.io/pdfs/sbrc2021rodrigues.pdf.

[4] Qian, L., Luo, Z., Du, Y., Guo, L. (2009). Cloud Computing: An Overview. In: Jaatun, M.G., Zhao, G., Rong, C. (eds) Cloud Computing. CloudCom 2009. Lecture Notes in Computer Science, vol 5931. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-10665-1_63

[5] Wang, L., von Laszewski, G., Younge, A. et al. Cloud Computing: a Perspective Study. New Gener. Comput. 28, 137–146 (2010). https://doi.org/10.1007/s00354-008-0081-5

[6] 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

[7] R. R. Filho and B. Porter, *Autonomous 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

[8] RODRIGUES FILHO, R.; PORTER, BARRY . Hatch: Self-Distributing Systems for Data Centers. Future Generation Computer Systems, v. 132, p. 80-92, 2022.

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

[10] M. F. de Oliveira, METODOLOGIA CIENTÍFICA: Um Manual Para a Realização De Pesquisas Em Administração, 2011. https://files.cercomp.ufg.br/weby/up/567/o/Manual_de_metodologia_cientifica_-_Prof_Maxwell.pdf

[11] Porter, B. and Rodrigues Filho, R. "A programming language for sound self-adaptive systems". In 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) (pp. 145-150). IEEE

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