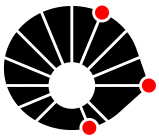

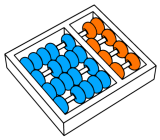

Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso

G. H. R. Oswaldo *L. F. Bittencourt* *R. R. Filho*

Relatório Técnico - IC-PFG-21-50
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Gestão transparente do estado para sistemas auto-distribuídos: primeiro estudo de caso

Gabriel Henrique Rosa Oswaldo* Luiz Fernando Bittencourt*
Roberto Rodrigues Filho†

Resumo

Sistemas contemporâneos têm cada vez mais demandado o uso de tecnologias que dão suporte à adaptação, devido ao aumento da sua heterogeneidade, e da volatilidade presentes em seus ambientes operacionais. Essa adaptação torna-se necessária ainda em tempo de execução, à medida que mudanças, cada vez mais difíceis de se antecipar, ocorrem no ambiente operacional. Tecnologias de contêiner, orquestradores de contêineres, e plataformas de computação em nuvem são exemplos de tecnologias que são vastamente utilizadas para prover essa flexibilidade e adaptabilidade demandada pelos sistemas modernos. Estas ferramentas são melhor utilizadas por sistemas sem estado (*stateless*), que por sua vez manipulam o estado antes da implantação do sistema, como por exemplo, extraindo e armazenando-o em base de dados. Porém, essa manipulação ainda acontece de maneira manual e fixa, sem beneficiar-se de diferentes alternativas. Em contrapartida, embora os sistemas com estado (*stateful*) sejam mais difíceis de adaptar, acabam tendo um melhor desempenho. De forma a aproveitar o melhor dos dois mundos, este projeto apresenta um primeiro estudo de caso na gestão transparente do estado, de acordo com diferentes modelos de consistência, dentro do processo de distribuição de sistemas. E através de uma metodologia empírica, são explorados os desafios e impactos da auto-distribuição de sistemas com estado, trazendo perspectivas futuras para a gestão autonômica do estado em sistemas auto-distribuídos.

1 Introdução

O projeto, implantação, observabilidade e manutenção de sistemas distribuídos continua sendo uma das tarefas mais desafiadoras da computação [1]. Essa complexidade leva a abordagens que ainda dependem fortemente de previsões, em tempo de projeto, das condições de operação em produção, o que torna a execução desses sistemas suscetíveis a incertezas e previsões incorretas [1, 4, 5]. Além disso, estas condições tornam-se cada vez mais difíceis de se antecipar, com o aumento da heterogeneidade dos sistemas, e da volatilidade presentes em seus ambientes operacionais [4]. Segundo o manifesto publicado pela IBM em 2001, esta é uma uma crise iminente na complexidade de software [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, para suportar operações em larga escala, sistemas modernos têm cada vez mais demandado o uso de tecnologias que dão suporte à adaptação [1, 4, 5]. A habilidade de se adaptar é muitas vezes baseada na capacidade de uma aplicação em alterar o seu comportamento, dentre diferentes implementações disponíveis, para acomodar mudanças durante a execução, de acordo com seus objetivos de alto nível [2, 4]. Estes sistemas são referenciados como auto-adaptativos, e são capazes de se adaptar em tempo de execução, tanto na camada de infraestrutura quanto na própria estrutura da aplicação, à medida que mudanças ocorrem nas condições de operação.

Em termos de infraestrutura, políticas de tolerância a falhas e algoritmos de alocação de recursos podem ser definidos para adaptação, de acordo com os objetivos de alto nível. Algumas estratégias comumente utilizadas são a adição ou remoção de nós ou réplicas (*horizontal scaling*) em operação, ou ajustes nas capacidades de recursos computacionais de um nó (*vertical scaling*). Tecnologias de contêiner, orquestradores de contêineres e plataformas de computação em nuvem, são exemplos de tecnologias que são vastamente utilizadas para prover essa elasticidade e adaptabilidade demandada pelos ambientes modernos. Entretanto, essas ferramentas são melhor utilizadas por sistemas sem estado (*stateless*), que por sua vez conseguem explorar ao máximo a flexibilidade que tais ferramentas provêem. Entende-se como o estado de uma aplicação a sequência resultante de operações de leitura e escrita sobre seus dados, em um determinado momento. Logo, os sistemas *stateless* não armazenam nenhuma referência ou informação sobre dados ou transações antigas, ou seja, o estado não é rastreado do lado do servidor. Esses sistemas dependem dos clientes ou base de dados. A extração e o armazenamento do estado em base de dados, antes da implantação do sistema, é atualmente a forma mais frequentemente usada para eliminar o estado do sistema, e torná-lo mais apto a explorar a adaptabilidade suportada pelas tecnologias mencionadas. Porém, essa manipulação do estado ainda é manual e fixa, sem beneficiar-se de diferentes alternativas [4], além de poder tornar-se um gargalo, em algum momento, afetando o desempenho.

Já as aplicações e os processos *stateful* são aqueles que podem executar novas operações com base no contexto das operações anteriores. Se uma operação *stateful* for interrompida, a mesma conseguirá ser retomada praticamente de onde parou, já que o contexto e o histórico são armazenados, tendo assim um melhor desempenho. Apesar disto, os sistemas com estado possuem uma complexidade maior em seu projeto, implementação e arquitetura, sendo mais difíceis de adaptar em tempo de execução. Esta complexidade está principalmente ligada ao nível de consistência exigido pelo estado em questão, ou seja, o nível de flexibilidade com que os nós de um sistema distribuído podem enxergar as operações sobre o estado, em um determinado ponto no tempo.

Dito isso, de forma a aproveitar o melhor dos sistemas *stateless* e *stateful*, este projeto propõe um primeiro estudo de caso para a gerência transparente de estado em sistemas auto-distribuídos. O sistema estudado em questão armazenará um estado, o qual será distribuído dentro do processo de adaptação, utilizando de distintas estratégias para a manipulação de estado, e explorando diferentes requisitos de uma aplicação e tráfego heterogêneo. A implementação utiliza da abordagem de Sistemas de Software Emergentes, a qual facilita o desenvolvimento de sistemas auto-adaptativos, através de modelos baseados em componentes e um framework, capaz de compor arquiteturas de software de forma autonômica [5].

Com isso, este é o primeiro passo para construir sistemas com estado auto-distribuídos confiáveis e em grande escala, que sejam altamente flexíveis, podendo tirar vantagem das tecnologias que dão suporte a esta elasticidade.

O trabalho foi dividido em seções. Na segunda seção são expostos os objetivos buscados com os resultados do trabalho. Na terceira seção são apresentados os conceitos que permeiam o projeto. A quarta seção descreve a metodologia utilizada para o desenvolvimento do trabalho. A abordagem adotada para a adaptação do sistema, e os detalhes da aplicação implementada como estudo de caso, são detalhados na quinta e sexta seção, respectivamente. A sétima seção apresenta e avalia os resultados obtidos. E por fim, na oitava e nona seção, são realçadas as possibilidades de investigação para trabalhos futuros dentro do tema, e as conclusões finais, respectivamente.

2 Objetivos

O objetivo deste projeto é explorar os desafios e impactos de adaptar sistemas com estado, desenvolvidos de forma local, para um cenário distribuído (auto-distribuição), considerando que isto deve ser feito em tempo de execução, e garantindo a gestão transparente do estado, de acordo com diferentes modelos de consistência. Como impactos, serão analisados tanto a gestão do estado da aplicação quanto o desempenho da mesma, em relação à percepção do usuário sobre a adaptação.

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

1. A gestão de estado funciona corretamente, de acordo com o modelo de consistência, ao adaptar o sistema? Há perdas de requisições e de disponibilidade?

2. Há ganhos de desempenho na composição adaptada do sistema, quando olhamos para tempo de resposta para o usuário?

Dada estas questões, visa-se levantar em quais cenários os impactos são mais evidentes, sejam eles negativos ou positivos.

3 Referencial Teórico

Nesta seção é apresentada a fundamentação teórica relacionada ao trabalho. São introduzidos os conceitos de computação autonômica e sistemas auto-adaptativos. Também é discutido a abordagem de Sistemas de Software Emergentes para implementação de sistemas auto-adaptativos, e sua ligação com sistemas baseados em componentes. Além destes conceitos, também é ressaltado técnicas utilizadas para lidar com consistência de dados, como estratégias para gestão do estado destes sistemas.

3.1 Computação Autonômica e Sistemas Auto-adaptativos

Em 2001 foi introduzido um manifesto pela IBM, sobre o principal obstáculo para o progresso na indústria de TI, a crise iminente na complexidade de software [1]. O manifesto ressalta a complexidade atual na gestão de sistemas computacionais, que vai muito além

de configurar a execução de um único software. Esta complexidade só aumenta juntamente com a necessidade de cada vez mais integrar sistemas corporativos com sistemas de terceiros, dentro de ambientes heterogêneos. Isto irá dificultar que até mesmo os profissionais mais qualificados sejam capazes de projetar, configurar, otimizar e manter sistemas, antecipando-se dos problemas provenientes da interconectividade e diversidade.

Como único caminho para contornar essa crise, Kephart e Chess discutem em [1] sobre a computação autonômica, definida 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. Sistemas autonômicos implementam as seguintes propriedades: auto-proteção, auto-otimização, auto-cura e auto-configuração.

Sistemas auto-configuráveis são capazes de se configurar automaticamente de acordo com políticas de alto nível, como objetivos a nível de negócios, por exemplo. Sistemas auto-otimizados buscam continuamente maneiras de otimizar sua operação, identificando parâmetros que podem ser ajustados para torná-los mais eficientes em desempenho ou custo. Sistemas auto-curáveis podem detectar, diagnosticar e reparar falhas no sistema, sejam estas bugs ou falhas. Sistemas com auto-proteção são capazes de identificar antecipadamente possíveis vulnerabilidades, ou durante a execução, tomando as ações necessárias para manterem-se seguros.

Quando olhamos para a estrutura de sistemas autonômicos, os mesmos são compostos por elementos autonômicos, que podem alterar seu comportamento interno ou relação com outros elementos. O elemento autonômico consiste de um elemento gerenciado e um único gerenciador autonômico, como mostra a Figura 1. O gerenciador autonômico é responsável por monitorar o elemento gerenciado e seu ambiente externo, alterando o seu comportamento e relacionamentos de acordo com os objetivos que foi projetado. Enquanto o elemento gerenciado pode ser desde um recurso de hardware ou de um software, até um grande sistema legado, entretanto, deve ser adaptado para que o gerenciador seja capaz de monitorá-lo e controlá-lo.

Sobre esta perspectiva, sistemas auto-adaptativos são comumente referenciados como sistemas capazes de mudar sua estrutura para acomodar mudanças, e possuem grandes desafios de Engenharia de Software, os quais são discutidos por de Lemos R. *et al.* em [2]. Em especial, sob a visão de distribuição, um desses desafios torna-se ainda mais interessante: o esquema de controle de adaptação centralizado e descentralizado. Ao nos referirmos ao esquema de controle em um sistema auto-adaptativo, estamos falando do processo de decisão que resulta em ações de adaptação, executadas pelo sistema em questão. O controle pode ser entendido como o ciclo das seguintes operações: monitorar, analisar, planejar e executar (MAPE), o qual também podemos ver na Figura 1.

No controle descentralizado, o conhecimento e decisão dos componentes para alterar o seu comportamento são provenientes de informações e interações locais. Enquanto no controle centralizado, as decisões sobre alterações do comportamento do sistema no geral são realizadas por apenas um componente. Naturalmente, espera-se que, quando o software é implantado de forma não distribuída, o controle de adaptação seja centralizado. Da mesma forma, para um cenário distribuído, o controle descentralizado do sistema torna-se

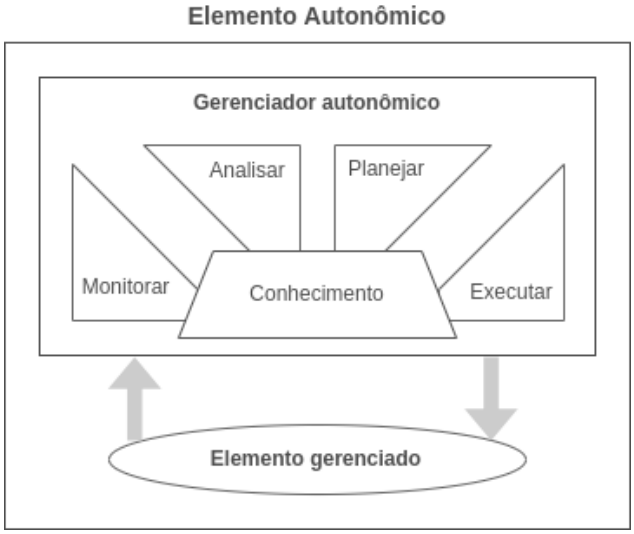

Figura 1: Estrutura de um elemento autonômico. Adaptado de [1].

mais provável. Contudo, o controle em um sistema auto-adaptativo pode ser centralizado ou descentralizado, independente de estar em um contexto distribuído ou não. A partir dessas perspectivas, surgem diferentes abordagens [2] que podem ser exploradas para lidar com o controle de adaptação, como a *Master-slave*, exemplificada pela Figura 2. Nesta abordagem há uma relação hierárquica, na qual o nó *master* é responsável pela análise e planejamento da adaptação, enquanto os nós *slaves* são responsáveis por monitorar o componente, fornecendo informações ao *master*, e por executar as ações de adaptação provenientes do mesmo.

3.2 Sistemas de Software Emergentes

Como introduzido na seção anterior, a complexidade e o dinamismo na elaboração de sistemas computacionais modernos têm sido uma grande motivação para o avanço nos estudos sobre sistemas auto-adaptativos. Entretanto, uma análise das abordagens para realizar sistemas auto-adaptativos, feita em [5], mostrou que tais abordagens possuem algumas características que impedem o avanço desses sistemas: i) dependência humana no seu projeto; ii) incapacidade de evoluir a sua lógica de adaptação de forma autonômica, frente a condições inesperadas; e iii) mecanismos de adaptação que são aplicados apenas a partes muito específicas dos sistemas.

Diante disso, a abordagem de Sistemas de Software Emergentes foi proposta como um facilitador no desenvolvimento de sistemas auto-adaptativos. Este conceito consiste em compor sistemas, de maneira autonômica, a partir de pequenos componentes de software reutilizáveis, de acordo com métricas, eventos e objetivos de alto nível fornecidos pelo usuário [5]. Isto garante que sistemas sejam capazes de mudar sua estrutura para acomodar mudanças externas, de acordo com suas metas desejadas. O uso de pequenos componentes de software permite que sejam construídas pequenas unidades de comportamento, com possíveis inter-relações entre elas e variações em suas implementações, de modo que possam

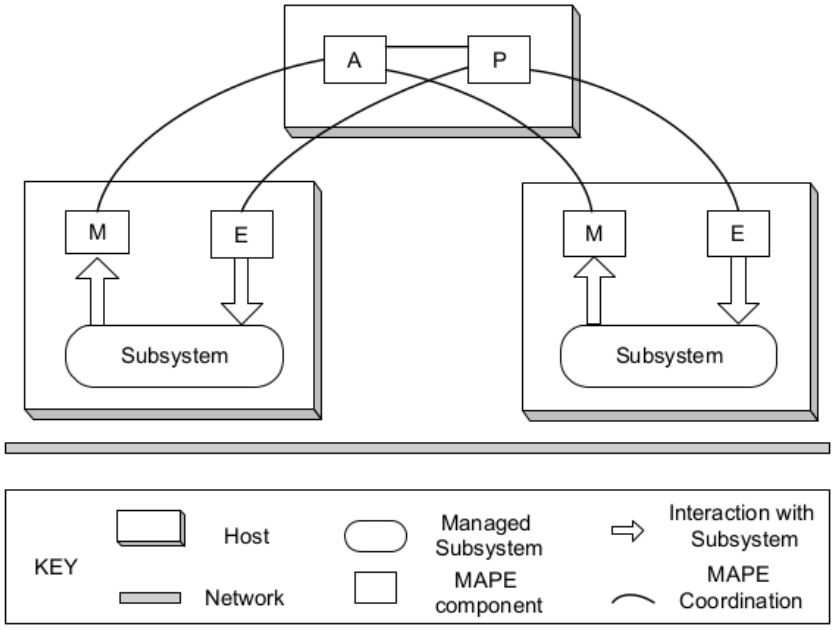

Figura 2: Abordagem de controle de adaptação *Master-slave* [2].

ser agregadas autonomicamente de diferentes formas, mantendo o sistema vivo [6].

Visto isso, a realização do conceito de Sistemas de Software Emergentes se deu através do framework PAL, introduzido em [9]. O framework possui três módulos principais: *Perception*, *Assembly* e *Learning* [5], como mostra a Figura 3. O *Assembly* é capaz de descobrir e reunir, em memória, os componentes disponíveis em um repositório, criando uma representação de todas as possíveis composições arquiteturas para o sistema, além de fornecer suporte para mudar de uma composição para outra em tempo de execução. Já o módulo *Perception* cria e adiciona componentes *proxies* entre as conexões dos demais componentes, a partir das composições do *Assembly*, de forma que estes *proxies* monitorem a saúde do sistema e seu ambiente operacional. Por fim, o módulo *Learning* utiliza dos demais módulos para conduzir o processo de adaptação autonômica de acordo com os objetivos do sistema, através de abordagens de aprendizado por reforço.

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 da linguagem Dana [3]. Dana é uma linguagem de propósito geral que fornece, nativamente, um modelo baseado em componentes, através da definição de interfaces e componentes, e também uma API com suporte a adaptação de componentes em tempo de execução. Utilizando então da 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 [5].

- `String[] getConfigs()`: retorna uma lista de *strings* em que cada uma representa

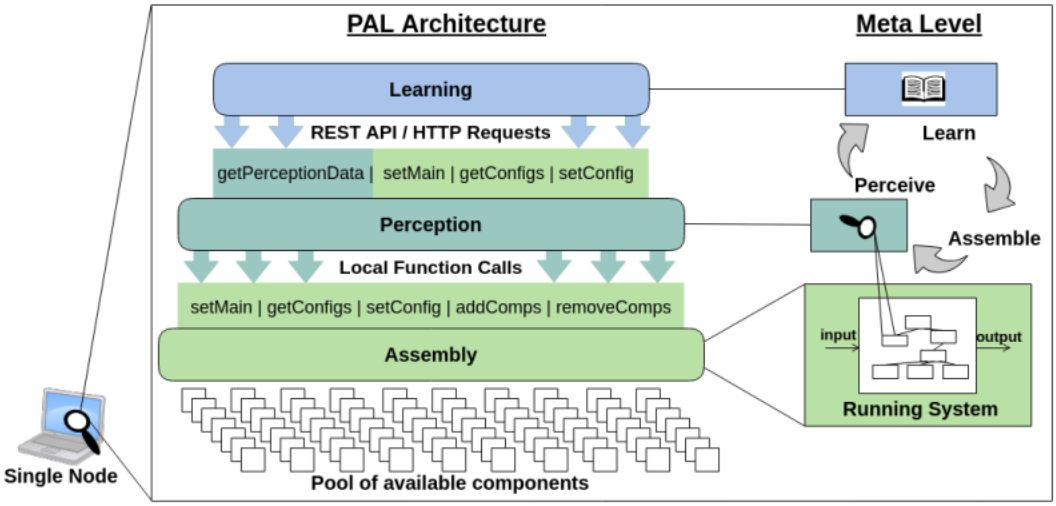

Figura 3: Arquitetura do framework PAL [9].

a descrição de uma composição arquitetural disponível.

- `bool setConfig(char configDesc[])`: altera a composição arquitetural do sistema de acordo com a descrição da nova composição passada por parâmetro, durante execução.
- `bool addComp(char compName[])`: adiciona o componente passado por parâmetro à lista de componentes disponíveis, procurando-o no repositório e carregando-o no módulo *Assembly*, aumentando então o número de composições arquiteturais possíveis.

Desta forma, o *Assembly* consegue reconhecer as variações de componentes que implementam uma mesma interface, assim como as dependências para com outras interfaces e os componentes que as implementam, e assim sucessivamente, até carregar em memória todas as composições possíveis para a arquitetura do sistema em questão, e adaptar entre elas durante execução. Uma ilustração dessa ideia pode ser vista na Figura 4.

3.3 Estratégias para Gestão do Estado

A maior parte dos estudos relacionados a computação autonômica e sistemas auto-adaptativos, dentro de ambientes distribuídos, têm sido operacionalizados através de tecnologias de computação em nuvem e na borda, os quais usam uma variedade de tecnologias de contêiner para suportar infraestruturas flexíveis que podem escalar [4]. Entretanto, essas tecnologias possuem uma lacuna no suporte à gestão de estado, tornando-se necessário que desenvolvedores extraiam o estado do sistema em questão, armazenando-o, por exemplo, em um banco de dados. A tarefa de separação ainda permanece altamente manual, e requer um projeto cuidadoso, além de deixar os sistemas com apenas uma maneira fixa de lidar com seu estado, tornando difícil explorar alternativas que poderiam aumentar o desempenho

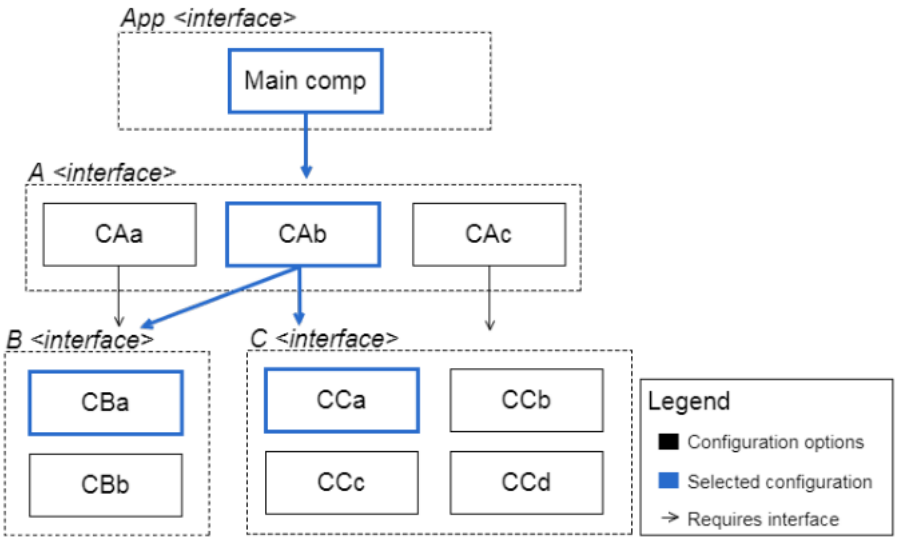

Figura 4: Exemplo de uma arquitetura genérica representada pelo módulo *Assembly* [5]. Podemos ver as dependências entre componentes e interfaces, representadas pelas setas, as possíveis composições arquiteturais dentre as variações de componentes que implementam uma mesma interface, e a composição definida pelo *Assembly* em azul.

em diferentes condições de implantação [4]. É importante ressaltar que entende-se como o estado de uma aplicação a sequência resultante de operações de leitura e escrita sobre seus dados, em um determinado momento.

Em vista disso, o cuidado exigido está principalmente relacionado à consistência do estado demandada pelos objetivos do sistema, ou seja, o nível de flexibilidade com que os nós de um sistema distribuído podem enxergar os dados, em um determinado ponto no tempo, independente do cliente que os atualizou. Uma consistência mais fraca gera alta disponibilidade, alta performance e alto *throughput*, mas em contrapartida o estado torna-se menos consistente. Já uma consistência forte possui baixa disponibilidade, baixa performance e baixo *throughput*, entretanto mantém o estado mais consistente. Dessa forma, a escolha de um modelo de consistência trata-se basicamente de um *tradeoff* entre concorrência e ordenação de operações, ou seja, entre o desempenho e a corretude do sistema [7]. E quando olhamos para a sequenciação de operações de leitura e escrita, o modelo a ser escolhido pode ser visto sob a perspectiva do estado em si (*data-centric*), ou sobre o cliente individual que está operando sobre ele (*client-centric*). Na visão *data-centric*, o sistema está ciente de múltiplos clientes atuando sobre o estado, e então pode afrouxar a consistência através de regras gerais, ou garantir que os nós cheguem a um acordo global sobre a ordenação consistente das operações [7].

Sendo assim, podemos discutir algumas estratégias para gerir o estado e manter sua consistência, durante a implementação de sistemas auto-adaptativos através da abordagem de softwares emergentes. Estas estratégias utilizam do mesmo princípio: o estado do componente é exposto em tempo de execução e, durante o processo de adaptação, o mesmo pode

ser extraído do componente antigo e injetado no componente de substituição, preservando o estado nas adaptações. Duas estratégias, que são comumente utilizadas são: a replicação e o *sharding*.

Na replicação, um componente e seu estado são replicados em vários *hosts* remotos, descarregando suas demandas de computação, permitindo assim a escala horizontal. Para isso, o componente local é substituído por um componente *proxy* que implementa a mesma interface, entretanto, os métodos da interface passam a ser chamadas de procedimento remoto (*RPCs*) para os componentes destino carregados em máquinas remotas. Com as várias réplicas, o *proxy* precisa fazer a propagação das operações de alteração de estado para manter a consistência. Este ponto é tratado na literatura como comunicação em grupo ou *broadcast*. Protocolos de *broadcast* mantém a confiabilidade garantindo que cada mensagem é eventualmente entregue a cada nó disponível, no entanto, eles diferem em termos da ordem em que as mensagens podem ser entregues a cada um [10]. Com isso, para que o *proxy* mantenha uma consistência forte, algoritmos de *broadcast* podem ser utilizados, como o de *Total order broadcast*, o qual garante que se uma mensagem *m1* for entregue antes de *m2* em um nó, então *m1* deve ser entregue antes de *m2* em todos os nós [10]. Contudo, essa operação de propagação pode levar a um *overhead* do sistema em questão, perdendo desempenho.

Já o *sharding* consiste em particionar um conjunto de dados entre diferentes réplicas. Também referenciado como particionamento horizontal, esta estratégia é frequentemente utilizada em bancos de dados. O *sharding* torna-se muito vantajoso quando o estado em questão não exige uma consistência forte, e quando partes distintas e independentes são intensamente requisitadas, reduzindo então a competição por recursos computacionais [4]. Ao implementar esta estratégia, o componente *proxy* precisa definir uma abordagem de particionamento do estado, como a utilização de um *hash* consistente, e o balanceamento de carga entre as réplicas. Sendo assim, vimos a possibilidade de estender estratégias que lidam com o estado para o cenário de adaptação, com o objetivo de suportar a gestão do estado de forma transparente, trazendo então duas novas dimensões de adaptação em qualquer sistema auto-distribuído.

4 Metodologia

Para responder às questões de pesquisa, foi adotada uma metodologia empírica para o desenvolvimento do trabalho. Foram implementadas um conjunto de variações de uma mesma aplicação, juntamente com uma versão do framework PAL de auto-adaptação, para avaliar o comportamento da adaptação em diferentes cenários.

Ao adotar uma metodologia empírica, levantam-se reflexões em torno das possíveis características que devem ser levadas em conta para sustentar os resultados buscados. Dessa forma, existem três pontos importantes para se avaliar os resultados esperados:

i) Requisitos do sistema: características quanto ao funcionamento do sistema.

ii) Carga de trabalho: características quanto ao perfil e padrão de carga de trabalho, a qual o sistema está exposto.

iii) Consistência de dados: como se dá a aceitação do sistema para diferentes tipos de modelos de consistência.

Entender quais os requisitos do sistema, ou seja, quais as operações realizadas em termos de processamento, é importante para observar o impacto das operações de distribuição no desempenho do sistema. Este impacto pode ser mais evidente dependendo da operação de processamento aplicada. De uma maneira simplificada, podemos exemplificar com operações numéricas e de ordenação. Operações realizadas com números inteiros grandes, como identificar todos os números primos até um número, tendem a gerar um processamento maior desde o início das operações. Em contrapartida, as operações de ordenação tendem a ter um impacto no processamento à medida que o tamanho da coleção de dados aumenta. Logo, elas teriam um período, enquanto o número de itens é pequeno, em que o processamento não seria muito impactante.

Em relação ao padrão de carga de trabalho, o sistema está exposto à variação, tanto em volume, do número de usuários concorrentes, quanto ao padrão de requisições (leitura e escrita). Quanto maior o volume de usuários realizando requisições de forma concorrente, maior a necessidade de distribuição. Para a variação no padrão da carga de trabalho, pode haver um percentual maior de operações de escrita e atualização, em relação às operações de leitura de dados. Logo, compreender o perfil e padrão de carga de trabalho do sistema permitirá examinar os efeitos da distribuição em cenários distintos.

Quando olhamos para a consistência de dados, diferentes aplicações vão demandar diferentes formas de lidar com o seu estado. Por exemplo, softwares de colaboração, como Google Docs[1], Trello[2], Figma[3] e muitos outros, nos quais vários usuários podem fazer alterações simultaneamente no mesmo arquivo ou dados, aceitam um modelo de consistência eventual. Por outro lado, sistemas de transações bancárias, nos quais várias operações de crédito e débito são feitas em uma única conta, aceitam somente um modelo de consistência restritivo. Portanto, é importante entender o impacto da distribuição na gestão do estado, de acordo com o modelo de consistência do sistema em questão.

Sendo assim, para sustentar a metodologia empírica do projeto, foram determinadas algumas características a serem exploradas, para cada um dos três pontos mencionados acima.

Para os requisitos do sistema, foram exploradas diferentes combinações, para um mesmo tipo de processamento, entre as operações de leitura e escrita realizadas. Dessa forma, todas as possibilidades foram avaliadas. Além disso, a operação realizada para simular um processamento foi a de ordenação, logo, o foco está em aumentar o tempo de processamento à medida que o tamanho da coleção de dados cresce.

Quanto à carga de trabalho, buscou-se avaliar o sistema sob um volume de carga constante, cujo nível de estresse fosse elevado, lidando com o pico de carga a todo momento, proveniente de múltiplos usuários concorrentes. Dessa forma, é possível observar os momentos em que o sistema satura, e quais os efeitos da auto-distribuição neste cenário. Em relação ao padrão de carga, avaliou-se variados cenários de proporção entre operações de

[1] `https://workspace.google.com/intl/pt-BR/products/docs/`
[2] `https://trello.com/`
[3] `https://www.figma.com/`

leitura e escrita, desde os níveis mais extremos, com quase 100% de operações de leitura ou escrita, até níveis mais balanceados entre ambas as operações, buscando também avaliar os efeitos da adaptação em cada cenário.

Para o modelo de consistência de dados, foi explorado tanto um modelo mais restritivo quanto outro menos restritivo. O modelo de consistência mais restrita traz um nível maior de *overhead* no sistema em operações de escrita no cenário distribuído, uma vez que precisa manter todos os nós igualmente atualizados. Enquanto no modelo com menos restrição nada é afirmado sobre a intercalação de operações de escrita, e consequentemente sobre a ordenação no armazenamento dos dados.

Por fim, é importante ressaltar que foram realizadas simplificações ao longo da implementação, como a ausência de mecanismos de tolerância à falhas, e o ambiente simples de execução em uma mesma rede local. Estes pontos serão abordados com mais detalhes nas seções adiante. Entretanto, a metodologia adotada mantém características importantes encontradas em sistemas reais, e que foram executadas em um ambiente com aspectos reais, mesmo que limitado, não sendo utilizado nenhum tipo de simulação.

5 Abordagem

A abordagem tomada, para poder avaliar os objetivos levantados, foi a construção de um sistema, utilizando-se do framework de auto-distribuição, capaz de se adaptar em tempo de execução, fazendo a gestão transparente do seu estado. O sistema foi dividido entre a camada da aplicação e a de adaptação, também chamada de *meta-level*. A aplicação será comentada na seção seguinte, enquanto nesta seção tratamos da camada de adaptação. Ambas foram implementadas utilizando a linguagem Dana, por fornecer um modelo baseado em componentes e suporte em tempo de execução para adaptação, como mencionado no Referencial Teórico.

No *meta-level* temos os componentes responsáveis pela adaptação e distribuição, que utilizam um conjunto de *proxies*, e pela mecânica de gestão do estado da aplicação. É importante salientar que a adaptação é tratada apenas em nível de aplicação e não em infraestrutura, ou seja, aquilo que é adaptado são os componentes, distribuindo-os dentro de uma infraestrutura fixa.

A camada de adaptação pode ser explicada através de dois momentos distintos, antes e depois da adaptação.

5.1 Antes da adaptação

No *meta-level* temos o ponto de início de todo o sistema, o componente distribuidor, referenciado a partir de agora como *Distributor*. O *Distributor* é de fato quem inicia a aplicação, e interpreta os comandos implementados para realizar a adaptação, a partir dos dois modelos de consistência de dados explorados.

De acordo com o comando recebido, o *Distributor* realiza a adaptação do componente da aplicação que possui o estado, ou seja, a coleção de dados, partindo da composição local para composições distribuídas, através do uso de um *proxy* para este componente. Isto é feito através da capacidade do módulo *Assembly* do framework PAL em obter todas

as composições arquiteturais disponíveis do sistema, e mudar de uma composição para outra em tempo de execução, como ilustra a Figura 5. Isso só é possível devido a toda a componentização do sistema que a linguagem Dana fornece, permitindo que aplicações genéricas tenham seus componentes adaptados durante a execução, alterando assim o seu comportamento.

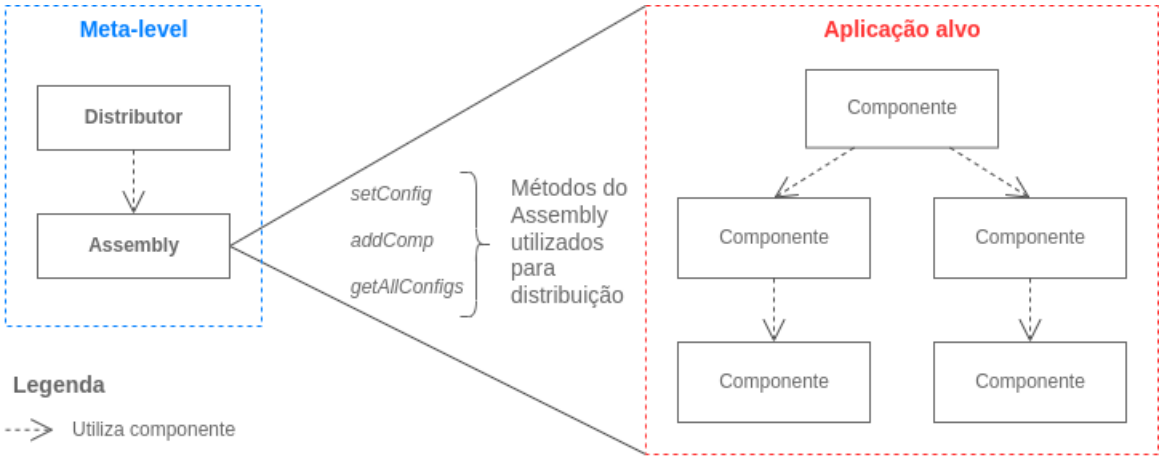

Figura 5: Os componentes do *meta-level* para sistemas auto-distribuídos. O *Distributor* utiliza do *Assembly*, que é capaz de montar/adaptar/executar a aplicação alvo.

O componente *proxy* mencionado é responsável por especificar a forma de lidar com o estado, e distribuir o mesmo para os nós remotos durante o evento de adaptação, mantendo sempre a consistência especificada. Como comentado na Metodologia, foram avaliados dois modelos de consistência de dados, um mais restritivo, e outro menos restritivo, gerando então duas implementações de *proxies*, *ReplicationProxy* e *ShardingProxy*, respectivamente. O modelo mais restritivo trabalha com os nós remotos como réplicas, ou seja, todos os nós possuem o mesmo estado, independente da intercalação de operações de leitura e escrita. Assim, todos possuem a mesma intercalação de escritas, que são propagadas para as múltiplas réplicas, e as leituras fornecem os mesmos dados, independente de qual réplica foi buscada. Já o modelo menos restritivo realiza a partição horizontal (*sharding*) dos dados, a partir de uma função de *hash* multiplicativo [8]. Logo, nada é afirmado sobre a intercalação de operações de escrita, que são direcionadas apenas para o nó específico calculado a partir da função de *hash*. Enquanto as operações de leitura possuem um *overhead* maior, de forma que precisam obter a coleção de dados de todos os nós.

Além da consistência de dados, o *proxy* implementa os mesmos métodos do componente adaptado, uma vez que provê a mesma interface. Entretanto, os métodos realizam chamadas de procedimentos remotos (*RPCs*) para os nós remotos, de acordo sempre com o requisito de consistência. Logo, são os nós remotos que realizam as operações requisitadas e o processamento de fato. Dessa forma, nada muda para a aplicação, a única diferença é que os métodos do componente adaptado agora são chamadas para procedimentos em nós

remotos, os quais realizam a operação desejada, com o devido processamento, e retornam o resultado.

Toda essa mecânica de adaptação e distribuição com gestão do estado pode ser visualizada na Figura 6.

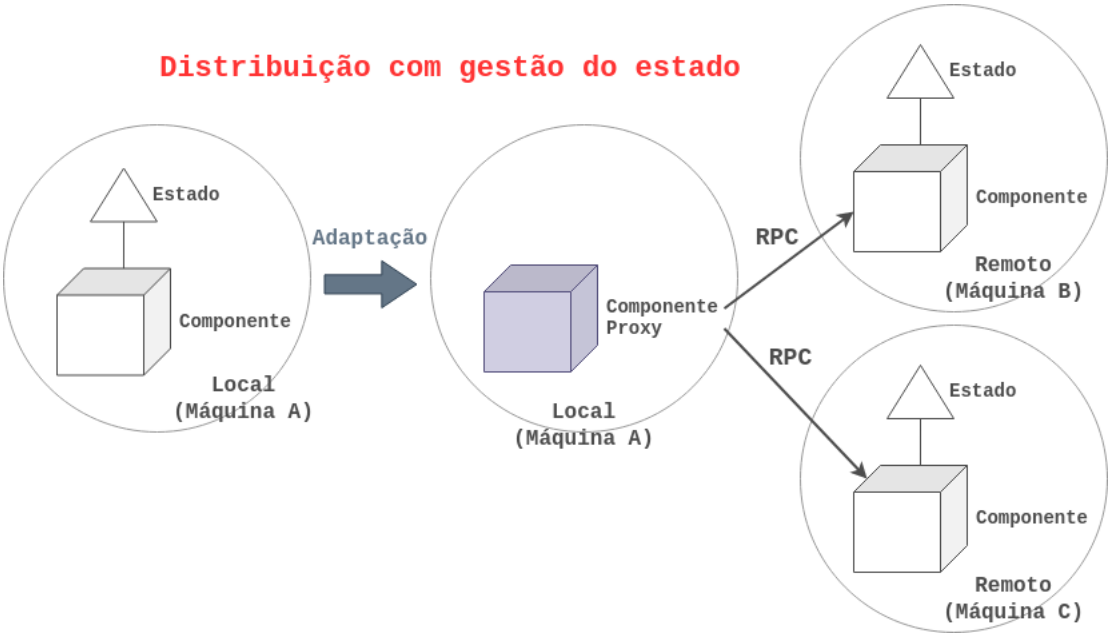

Figura 6: Ilustração do processo de adaptação e distribuição com gestão de estado. O componente local é adaptado para um *proxy*, e possui seu estado extraído para os componentes remotos, de acordo com o modelo de consistência. O *proxy* realiza chamadas de procedimento remoto (*RPCs*) para o componente remoto.

5.2 Depois da adaptação

Dentro do *meta-level* também há outro componente importante para a adaptação, chamado *Remote Distributor*, o qual é iniciado juntamente com o *Distributor*, entretanto, em nós remotos. O *Remote Distributor* aguarda o evento de adaptação, e quando o mesmo ocorre, é responsável por receber do *proxy* local, o estado do componente adaptado. Ao receber o estado, o *Remote Distributor* ativa o componente, chamado *Remote App*, que dará início ao *proxy* remoto (*Remote Proxy*). O *Remote Proxy* lidará com as requisições do *proxy* local, e realizará as operações sobre o componente adaptado, que agora executa em um nó remoto.

6 Estudo de Caso

Como estudo de caso, foi desenvolvida uma aplicação com estado, de forma que a abordagem para distribuição e gestão de estado fosse aplicada. A implementação consiste de um servidor simples, que lida com requisições HTTP dos tipos GET e POST. O servidor

utiliza como estado um componente que representa uma coleção de dados, ou seja, uma lista que armazena uma sequência de números inteiros. Nas requisições do tipo GET são retornados todos os elementos da lista, enquanto nas requisições do tipo POST é adicionado um novo número ao final da lista. Além disso, o servidor inicia com a execução local, até que o *Distributor* realize a adaptação, substituindo o componente da lista pelo *proxy*. A arquitetura da aplicação pode ser vista na Figura 7.

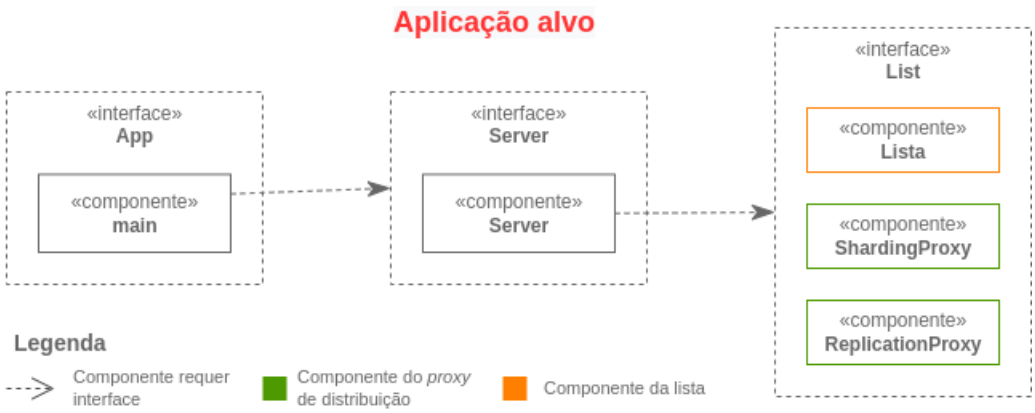

Figura 7: Os componentes da aplicação e suas interfaces. Dentro das interfaces *App*, *Server* e *List*, demarcadas com linhas pontilhadas, temos os componentes que as implementam, respectivamente: *main*, *Server*, *List* e os *proxies* de distribuição *ShardingProxy* e *ReplicationProxy*.

Para a execução local, assim como para a execução da lista nos nós remotos, foram implementadas quatro variações semânticas da interface *List*, nativa da linguagem Dana. Cada uma dessas implementações possui um requisito de processamento sobre os números. As variações são: leitura e escrita sem tempo de processamento; apenas leitura com tempo de processamento; apenas escrita com tempo de processamento; leitura e escrita com tempo de processamento. Para as operações de leitura e escrita sem processamento, os elementos da lista apenas são retornados e adicionados sem nenhuma propriedade. Para apenas a operação de leitura com processamento, os elementos são ordenados utilizando o algoritmo *Heapsort* [8], o que adiciona um tempo de processamento $O(n \lg n)$, onde n é o número de elementos na lista. Quando o processamento está apenas na escrita, a lista é mantida com a propriedade de um *Max-Heap* [8], de forma que, ao adicionar um novo elemento, a propriedade precisa ser mantida, adicionando um tempo de processamento $O(\lg n)$, onde n é, novamente, o número de elementos na lista. E ao termos um tempo de processamento tanto para operações de leitura quanto escrita, os dados são retornados ordenados pelo algoritmo *Heapsort*, e a lista é mantida com a propriedade de *Max-Heap* em inserções, adicionando um tempo de processamento $O(n \lg n)$ e $O(\lg n)$, respectivamente.

Com estas variações, mesmo que sejam simplificações, será possível observar o impacto das operações de distribuição no desempenho de sistemas com características importantes encontradas em sistemas reais. Ademais, este estudo de caso possui o foco especial em sistemas que realizam processamento sobre dados que podem crescer, em volume, conside-

ravelmente com o tempo, podendo então gerar um gargalo de desempenho.

Por fim, na Figura 8, é possível observar a arquitetura completa do sistema, exemplificando a atuação do *meta-level* em cima das aplicações alvo, tanto na composição local quanto em uma composição distribuída utilizando de dois nós remotos.

7 Avaliação

Nesta seção de avaliação são apresentados os impactos da adaptação na gestão do estado da aplicação, e quais os seus efeitos para diferentes modelos de consistência. Além disso, fazemos uma análise de desempenho do sistema, em relação ao tempo de resposta, entendendo em quais cenários composições distribuídas podem trazer benefícios. Diante disso, retomando os objetivos deste projeto, os resultados apresentados darão suporte às seguintes questões:

1. A gestão de estado manteve a consistência do mesmo, de acordo com o requisito de consistência, ao realizar a adaptação? Requisições foram perdidas e/ou o sistema ficou indisponível?

2. Houve impacto no tempo de resposta para o usuário, utilizando uma composição adaptada do sistema?

De forma a levantar em quais cenários estes impactos são mais evidentes, sejam eles negativos ou positivos, o sistema foi submetido a uma avaliação de desempenho sob variados cenários. Como mencionado na seção de Abordagem, foram implantadas duas formas de lidar com o estado da aplicação, o *Sharding* e *Replication*. Na seção de Estudo de Caso também foi comentado sobre as quatro variações em relação aos requisitos de processamento sobre as operações de leitura e escrita.

Para a carga de trabalho, o sistema foi exposto a cinco diferentes padrões de proporção entre requisições HTTP de leitura (GET) e escrita (POST), os quais são mostrados na Tabela 1. Para gerar uma carga cujo volume fosse constante, foi utilizada a ferramenta de teste de performance Locust[4], a qual permite definir o comportamento de seus usuários através de código Python[5]. Com a ferramenta foram gerados 20 usuários simultâneos, que são ativados a uma taxa de dois usuários por segundo, levando então dez segundos para produzir o pico de carga de duas requisições por segundo.

As avaliações foram realizadas dentro de uma infraestrutura fixa, com quatro máquinas, uma responsável por gerar a carga, uma com o *Distributor*, e duas com os *Remote Distributors*, todas conectadas em uma mesma rede local. Assim, o sistema era iniciado na composição local pelo *Distributor* em uma máquina, executando uma das quatro aplicações possíveis em relação ao processamento. Um comando também era disparado para o *Distributor*, de acordo com o cenário de teste buscado, para que o mesmo adaptasse o sistema para a composição distribuída, utilizando-se de um dos modelos de consistência. O código que gera uma das cargas de trabalho especificadas também era iniciado, porém em outra

[4] `https://locust.io/`

[5] `https://www.python.org/`

Proporção de operações de leitura e escrita, respectivamente, nas cargas de trabalho
9 : 1
7 : 3
1 : 1
3 : 7
1 : 9

Tabela 1: Proporções de operações de leitura e escrita nas cargas de trabalho avaliadas.

máquina, juntamente com os *Remote Distributors* nas duas máquinas restantes. Toda a implementação está disponível no Github[6].

7.1 Gestão do Estado

Nesta seção apresentamos e discutimos os resultados quanto à gestão transparente do estado da aplicação, de acordo com o modelo de consistência requisitado. Comparamos os dois modelos utilizados, *Sharding* e *Replication*, em cenários que há a necessidade de manter uma consistência restrita, e em outro não. Para obter estes resultados, o sistema foi executado com a aplicação cujas operações de leitura e escrita não possuem nenhum tipo de processamento, sob uma carga de trabalho com apenas escritas de dados, e sempre na mesma sequência e de forma síncrona, por um determinado período.

Primeiramente, a execução se deu com a composição local, e o resultado da coleção de dados ao final dessa execução foi considerado como o *estado inicial*, ou seja, o estado que possui a ordenação correta das requisições. Em seguida, o sistema foi executado com o comando para o *Distributor* realizar a adaptação da composição local para a composição de *sharding*, e depois retornar para a composição local, sob as mesmas condições. O resultado dessa execução, considerado como *estado final do sharding*, representa o estado cujo modelo de consistência nada garante sobre a ordenação das requisições. E por fim, o sistema foi executado novamente com as mesmas condições, mas agora o *Distributor* realizou a adaptação da composição local para a composição de réplicas, e depois retornou para a composição local. Isto resultou no *estado final do replication*, cuja ordenação esperada fosse a mesma do estado inicial, uma vez que o modelo é mais restritivo.

Ao final, os estados finais foram comparados com o *estado inicial*, olhando para os cenários de consistência com restrição e sem restrição, utilizando-se métricas de erro para medir as divergências obtidas.

7.1.1 Consistência sem restrição

Neste cenário, o primeiro ponto analisado foi se houve perda de requisições durante a adaptação, ou seja, se o número de itens no *estado inicial* é o mesmo para ambos os estados finais. Assim, foi observado que não houve nenhuma perda de requisição, uma vez que a

[6] `https://github.com/robertovrf/PFG_Gabriel`

quantidade de itens para o *estado final do sharding* e para o *estado final do replication* foi o mesmo que o *estado inicial*.

Como segundo ponto analisado, a métrica de erro utilizada foi a quantidade de itens do *estado inicial* que não estavam presentes nos estados finais, dividido pelo número total de itens. Ambas as composições distribuídas não obtiveram erro algum, conforme mostra a Tabela 2. Com isso, notamos que a adaptação manteve o estado correto, sem perdas, independente do modelo de consistência, quando não temos nenhuma restrição quanto a consistência.

Composição distribuída	**Erro**
sharding	0
replication	0

Tabela 2: Resultados para a gestão de estado em um cenário sem restrição de consistência, para ambas as composições distribuídas. A métrica de erro utilizada foi a quantidade de itens do estado inicial que não estavam presentes nos estados finais, dividido pelo número total de itens.

7.1.2 Consistência com restrição

Agora, quando olhamos para uma aplicação que exige uma consistência restrita sobre o estado, ou seja, a ordem dos dados, a ordenação das requisições torna-se importante. Dessa forma, cada item do *estado inicial* precisa ser igual ao item na mesma posição dos estados finais. Sendo assim, consideramos agora como nova métrica de erro a quantidade de itens que divergiram na ordenação dos estados finais para o *estado inicial*, dividido pelo tamanho total de itens. E, novamente, observou-se primeiro que não houve perdas de requisições durante a adaptação, da mesma maneira que o item anterior. Entretanto, para a questão de ordenação, houveram erros, como mostra a Tabela 3.

Composição distribuída	**Erro**
sharding	0.73
replication	0.00

Tabela 3: Resultados para a gestão de estado em um cenário com restrição de consistência, para ambas as composições distribuídas. A métrica de erro utilizada foi a quantidade de itens que divergiram na ordenação dos estados finais para o estado inicial, dividido pelo tamanho total de itens.

Para a composição *replication*, o resultado esperado era que não houvesse nenhuma divergência na ordenação dos itens, uma vez que sua implementação garante que os nós remotos recebam o mesmo conjunto de requisições na mesma ordem. E de fato foi o que aconteceu, nenhum dos itens ficou fora de ordem durante todo o processo de adaptação.

Quanto à composição com *sharding*, o resultado também foi como o esperado, não há garantia de ordenação, logo 73% dos itens estavam fora da ordem correta do *estado inicial*. Isso ocorre pois o particionamento dos dados entre os nós é feito apenas pela função de *hash* aplicada ao item (número) da lista, e nada mais é armazenado sobre o mesmo. Para manter uma maior consistência, e ainda utilizar da estratégia de *sharding*, seria necessário adicionar um *timestamp* a cada item adicionado. Dessa forma, quando fosse necessário recuperar todos os itens da lista no processo de adaptação, através de cada fragmento, o *proxy* local poderia ordená-los em ordem crescente aos *timestamps*.

Entretanto, essa abordagem possui alguns efeitos colaterais, pois adicionaria um *overhead* a mais para a operação de adaptação, além do tempo de rede para obter os fragmentos da lista. Além disso, essa abordagem tornaria a implementação do *proxy* não genérica, uma vez que adiciona uma operação específica em sua implementação. No geral, este é um problema complexo ao implementar *sharding*. Torna-se simples implementá-lo de forma específica para uma interface, mas quando tentamos mapear um comportamento básico que possa ser utilizado para qualquer interface que tenha uma coleção de dados, sendo essa uma lista ou não, a dificuldade aumenta.

Além dos fatores de consistência, quando olhamos para o processo de adaptação no geral na Figura 9, observamos que houve um período em que o sistema interrompeu o processamento das requisições até que a adaptação terminasse. Portanto, houve um *downtime*, ou seja, uma queda brusca na quantidade de requisições por segundo que o sistema pode processar. Dessa forma, quando o sistema desencadeia a adaptação, as requisições que chegam passam a aguardar em uma fila, e quando a adaptação termina, as requisições na fila são tratadas, por isso há um pico nas requisições por segundo logo em sequência. A Figura 9 mostra as requisições por segundo ao longo do tempo para um cenário específico, contudo, esse foi um comportamento observado para as diferentes aplicações e cargas de trabalho, a única variante foi o tempo que a adaptação levou para acontecer.

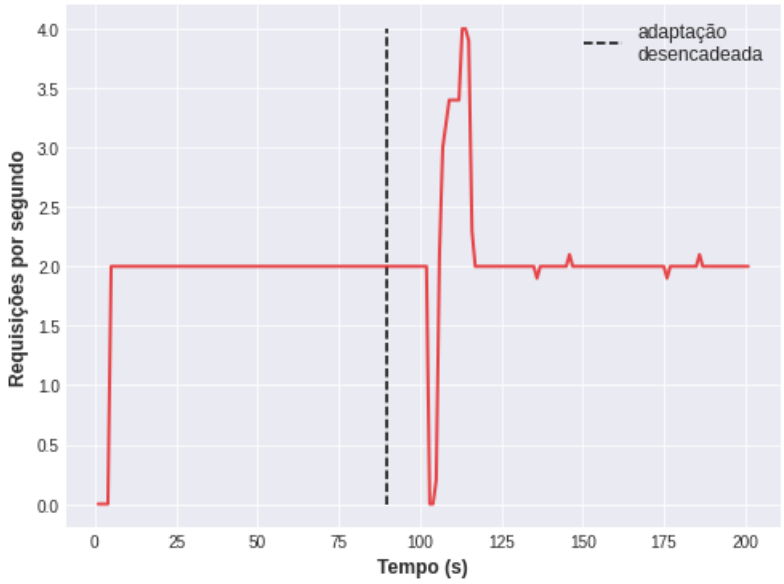

Figura 9: Comportamento das requisições por segundo que o sistema consegue lidar ao longo do tempo, para a aplicação sem processamento, adaptando da composição local para *replication*, sob a carga de trabalho de proporção 9:1 entre leitura e escrita, respectivamente.

Com isso, damos suporte à primeira questão de pesquisa levantada, mostrando que a gestão de estado transparente durante o processo de adaptação funciona corretamente, ainda que com ressalvas para uma consistência mais restrita.

7.2 Tempo de Resposta

Para esta seção, são descritos e discutidos os resultados da segunda questão de pesquisa levantada, em relação aos impactos no desempenho do sistema para uma composição distribuída, quando olhamos para o tempo de resposta. Uma análise é feita em cima das variáveis que podem impactar o tempo de resposta do sistema, comparando a composição distribuída com a local, de forma a evidenciar os cenários em que a adaptação pode trazer benefícios. A partir desta análise, experimentos são executados para validar os cenários propostos.

Para analisar o desempenho em relação ao tempo de resposta, é preciso entender quais as variáveis que irão impactar o tempo total, tanto para a composição local quanto distribuída. Quando olhamos a composição local, temos o tempo de rede para o usuário enviar a requisição, e receber a resposta da aplicação (T_{R1}), e o tempo de processamento da aplicação em cima da coleção de dados (T_{P2}), de acordo com a requisição. Dessa forma, o tempo total de resposta para a composição local (TR_L) é mostrado na equação abaixo:

$$TR_L = T_{R1} + T_{P2} \tag{1}$$

Para a composição distribuída, ainda há o mesmo tempo T_{R1}, entretanto, o tempo na

aplicação local deixa de ser T_{P2}, e passa a ser o tempo para que o *proxy* processe a requisição do usuário, encaminhe para o nó remoto, dependendo do modelo de consistência utilizado, e também receba a resposta do mesmo, retornando para o usuário. Consideramos este tempo como T_{P1}. Além disso, há o tempo de rede para transmitir a requisição ao nó remoto, e o retorno dessa requisição (T_{R2}), juntamente com o tempo de processamento no nó remoto em cima da coleção de dados (T'_{P2}). Logo, o tempo total de resposta para a composição distribuída (TR_D) é mostrado na equação abaixo:

$$TR_D = T_{R1} + T_{P1} + T_{R2} + T'_{P2} \tag{2}$$

As variáveis envolvidas para ambas as composições podem ser visualizadas na Figura 10.

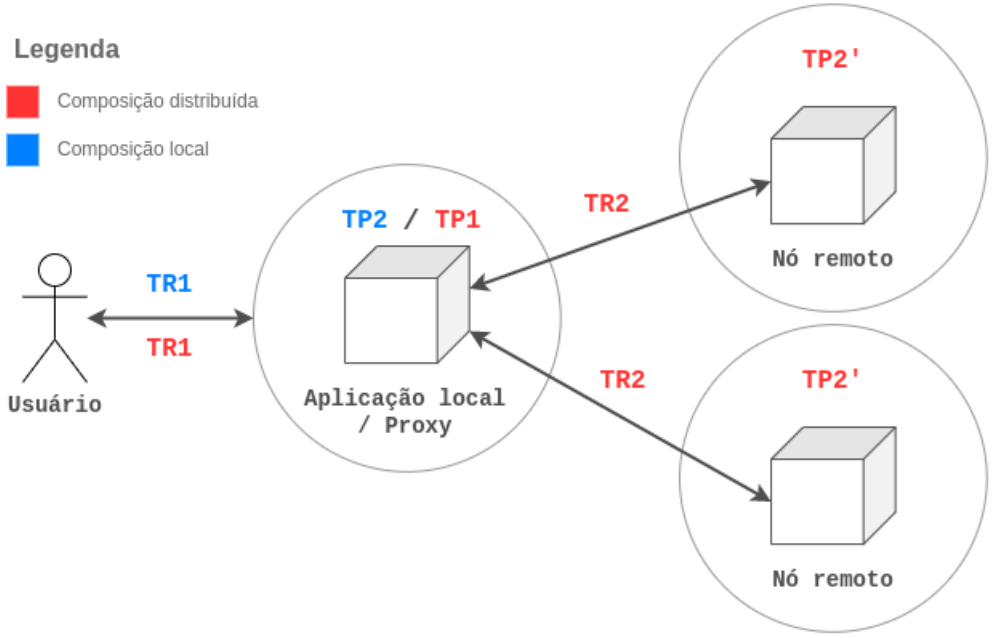

Figura 10: Ilustração das variáveis envolvidas na análise do tempo de resposta total para as composições local e distribuída.

Olhando para as equações 1 e 2, vemos que a composição distribuída só será vantajosa quando $T_{P2} > T_{P1} + T_{R2} + T'_{P2}$, ou seja, 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. Isto mostra que nem sempre a distribuição pode ser vantajosa.

Para os cenários trabalhados no projeto, podemos notar que a adaptação da aplicação para o *ReplicationProxy* pode não trazer benefícios. Isto pois T'_{P2} será igual a kT_{P2}, sendo k o número de réplicas, uma vez que todas as réplicas possuem a coleção completa de dados, e cada operação de atualização é propagada para todas as réplicas. Utilizar de replicação

só será interessante quando busca-se alta confiabilidade, uma vez que para manter a consistência é necessário abrir mão de um alto desempenho, não só para o tempo de resposta, mas também para o *throughput*, e até mesmo para recursos computacionais. O mesmo vale para o *sharding* em operações que envolvam todos os fragmentos, se considerarmos que a função de *hash* distribui uniformemente os dados em cada fragmento. Assim, o tempo de processamento irá envolver todos os fragmentos, e isto pode não ser suficientemente inferior ao tempo de processamento local.

Com esta análise, podemos identificar que o *sharding* com operações em um único fragmento demonstrou-se o melhor cenário para avaliar o impacto da distribuição no tempo de resposta, visto que o número de itens em cada fragmento é dividido pelo número total de fragmentos, logo o tempo de processamento também é reduzido. Por isso, uma alteração foi feita no *ShardingProxy*, para que operações de leitura sejam feitas em um único fragmento, e não em todos. E para validar este resultado esperado, dois cenários específicos foram executados, como mostra a Tabela 4.

É importante ressaltar a especificidade do nosso estudo de caso, a qual trata-se do fato do tempo de processamento aumentar, conforme o tamanho da coleção aumenta, e portanto o tamanho do dado trafegado pela rede também. Levando isto em conta, buscamos aumentar o processamento, porém manter o tamanho do dado trafegado pela rede constante. E para isso, uma alteração nas operações de leitura foi realizada para, ao invés de retornar todos os itens da lista, retornar apenas um item, de forma que o tempo T_{R2} não tenha grande impacto na avaliação.

Cenário	**Operação com processamento**	**Carga de trabalho**
a)	somente leitura	9 : 1
b)	somente escrita	1 : 9

Tabela 4: Cenários de teste para avaliar a análise sobre a composição de *sharding* com operações em um único fragmento, tanto de leitura e escrita com processamento.

7.2.1 Cenário a)

Para validar este cenário, o sistema foi executado com a aplicação cujo tempo de processamento está apenas na leitura, primeiramente apenas com a composição local, e depois apenas com a composição de *sharding*. Em ambos os casos, foram adicionados 100 itens antes de submetê-los à respectiva carga de trabalho, de forma que o tempo de processamento para a ordenação fosse significativo desde o início. Na Figura 11 é possível observar os resultados para a média total do tempo de resposta, assim como para a média do tempo de resposta ao longo do tempo, para ambas as composições.

Verificamos que a composição distribuída com *sharding* teve um desempenho superior a composição local, conforme o esperado. Observando o tempo médio de resposta ao longo do tempo, notamos que a composição local mal conseguiu lidar com as requisições, tendo picos de tempo de resposta, conforme o processamento de ordenação era finalizado a cada requisição. Enquanto a composição distribuída conseguiu lidar de forma muito mais ho-

mogênea com as requisições, apesar do tempo de resposta ainda ter sido alto. Para reduzi-lo ainda mais seria possível aumentar o número de fragmentos, diminuindo então o tempo de processamento em cada fragmento.

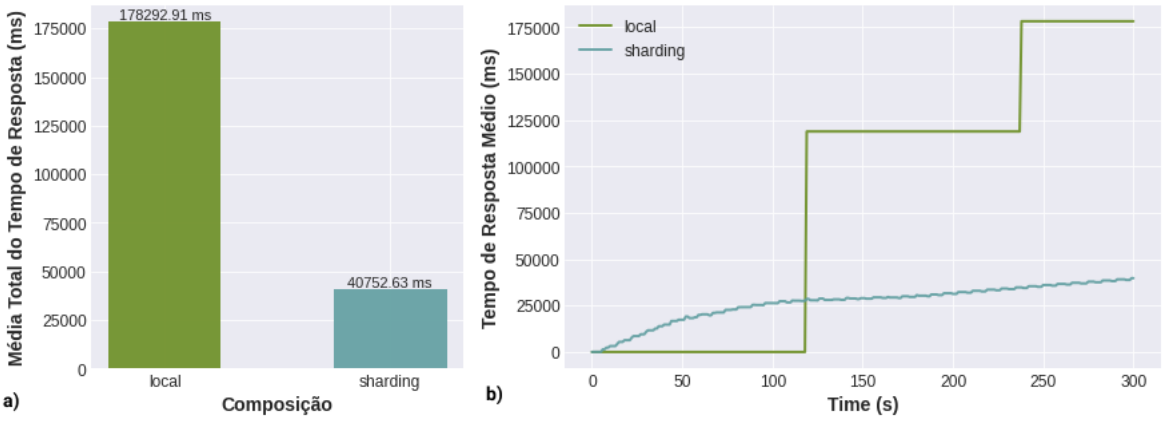

Figura 11: Resultados da execução do sistema para o cenário a) da Tabela 4. a) Média total do tempo de resposta; b) Média do tempo de resposta ao longo do tempo.

7.2.2 Cenário b)

Neste cenário, os mesmos testes foram realizados, mas com a nova carga de trabalho, e com a aplicação cujo tempo de processamento na escrita, ao manter a propriedade de *Max-Heap*. Entretanto, foram adicionados 500 itens antes de submetê-los à carga, de forma que o tempo de processamento também fosse significativo logo de início. Com os resultados mostrados na Figura 11, para a média total do tempo de resposta e a média do tempo de resposta ao longo do tempo, percebemos que a composição local ainda superou a composição distribuída no desempenho.

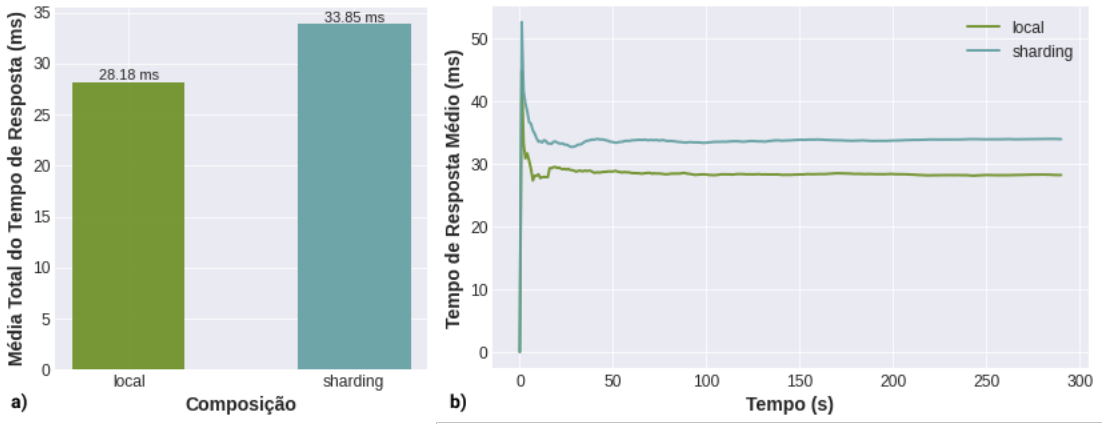

Figura 12: Resultados da execução do sistema para o cenário b) da Tabela 4. a) Média total do tempo de resposta; b) Média do tempo de resposta ao longo do tempo.

Com isso, entende-se que o tempo de processamento local ainda foi inferior à soma das variáveis do tempo total para a composição distribuída. Para validar este ponto, o processamento da aplicação de escrita foi alterado para fazer a ordenação dos itens a cada inserção, assim como na leitura, ao invés de manter a propriedade do *Max-Heap*. Dessa forma, o tempo de processamento passou a ser $O(n \lg n)$ ao invés de $O(\lg n)$, e com isso pudemos confirmar se o entendimento está correto.

O teste foi executado novamente, agora adicionando 100 itens antes de iniciar a carga. Os resultados da média total do tempo de resposta, e a média do tempo de resposta ao longo do tempo, são mostrados na Figura 13. Observamos que o desempenho da composição com *sharding* se sobressaiu em relação a composição local, utilizando de um tempo maior de processamento, conforme era esperado.

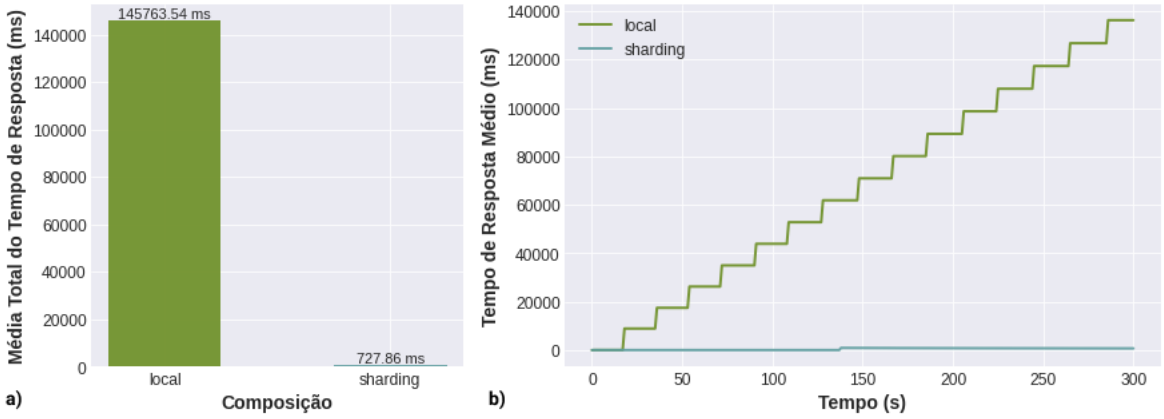

Figura 13: Resultados da execução do sistema para o cenário b) da Tabela 4, mas agora com tempo de processamento $O(n \lg n)$. a) Média total do tempo de resposta; b) Média do tempo de resposta ao longo do tempo.

Com isso, foi possível realizar uma primeira análise em cima das variáveis envolvidas no tempo de resposta quando ocorre a adaptação para um composição distribuída. Mostramos possíveis cenários em que os impactos da adaptação podem ser benéficos para o desempenho do sistema. E os experimentos realizados em cima destes cenários comprovaram a análise feita, trazendo resultados preliminares.

Entretanto, apesar das perspectivas positivas, os cenários ainda giraram em torno de casos muito específicos, e neutralizaram o efeito do tempo de rede (T_{R2}) na análise. Dessa forma, ainda há uma gama de possibilidades a serem exploradas, como situações que necessitam de operações em múltiplos fragmentos do *sharding*, aprofundar a visão da adaptação em modelos de consistência restrita, e até mesmo avaliar outras métricas de performance, como tempo de uso de CPU e uso de memória. Outra possibilidade a ser analisada seria entender o limiar, em relação ao tamanho da coleção de dados, no qual a auto-distribuição torna-se interessante, utilizando algoritmos com diferentes tempos de execução.

8 Trabalhos Futuros

Nesta seção, discutimos as limitações de nossa abordagem como oportunidades para trabalhos futuros. A adaptação de componentes para uma arquitetura distribuída mostrou resultados positivos em certos cenários, ainda que todo o ambiente de execução fosse fixo e definido antes da execução, dentro de uma mesma rede local, e com 100% do tempo disponível, sem falhas. E apesar da consistência do estado ter sido mantida durante a adaptação, mesmo que com ressalvas, a gestão do mesmo ainda necessitou do controle humano sobre o sistema.

Para tornar os resultados mais próximos a ambientes reais, há a possibilidade de explorar a implementação e execução do sistema em plataformas de computação em nuvem, de forma que as partes do sistema, como o *Distributor* e *Remote Distributor*, fossem executadas

em máquinas com diferentes redes e localidades. Também é possível explorar um tráfego distribuído, e não proveniente apenas de uma máquina.

Além do ambiente de execução estar todo em uma mesma rede local, a infraestrutura provida é fixa, utilizando-se de apenas duas máquinas para dar suporte à distribuição, as quais estão sempre disponíveis. Atualmente, plataformas de computação em nuvem oferecem um grande ferramental para trabalhar a elasticidade demandada pelos ambientes modernos. Dessa forma, há a oportunidade de trazer a adaptação para o nível de infraestrutura, e não somente em nível de aplicação. O desenvolvimento de uma nova implementação do *Distributor* seria necessário, a qual seja capaz de explorar as ferramentas que fornecem elasticidade à infraestrutura, inclusive olhando para a perspectiva de tolerância a falhas. Assim, um sistema desenvolvido para ser executado localmente, poderá alterar sua arquitetura adaptando componentes com estado para novos nós criados na infraestrutura, à medida que a carga do sistema se altera.

Ainda nesta linha, a adaptação ocorre pelo comando passado as *Distributor* diretamente via código. Porém, para que o sistema seja auto-adaptativo, existe a necessidade de explorar modelos de aprendizado de máquina com o objetivo do sistema aprender, sem a interferência humana, e a partir de métricas que podem ser coletadas diretamente no *Distributor*, qual a melhor composição distribuída do sistema. Este aprendizado pode se dar em relação ao uso de diferentes aplicações com diferentes processamentos, ou a melhor composição para uma diferente carga de trabalho, ou até qual o melhor modelo de consistência para componentes com estado em determinados momentos.

Por fim, para este último ponto sobre a consistência do estado, o trabalho foi feito em cima da interface *List*. Entretanto, há a possibilidade de estudar o quão genérica a gestão de estado pode ser, entendendo a viabilidade de ser aplicada para qualquer coleção de dados, ou até mesmo para qualquer tipo de dado abstrato. Desta maneira, poderiam ser exploradas formas de, em tempo de projeto, um determinado componente ter especificado qual modelo de consistência o mesmo aceita. Logo, o *Distributor* poderia interpretá-la e aprender em tempo de execução, garantindo assim a gestão autonômica do estado em sistemas auto-distribuídos.

9 Conclusões

Neste projeto foram estudados conceitos sobre computação autonômica e sistemas auto-adaptativos, assim como sobre Sistemas de Software Emergentes, e estratégias para lidar com consistência de estado. Em cima destes conceitos, o trabalho propôs um primeiro estudo de caso para a gestão transparente de estado em sistemas auto-distribuídos. Os experimentos realizados tiveram como objetivos analisar e mostrar os impactos da adaptação de um sistema com estado para uma composição distribuída. Os impactos observados foram em relação a gestão do estado, de acordo com diferentes modelos de consistência, e também o desempenho do sistema em termos de tempo de resposta. E com isso, foi possível obter resultados preliminares sobre os cenários em que a adaptação pode trazer benefícios.

Estes resultados mostraram que a gestão do estado durante a adaptação é capaz de manter a consistência do mesmo, sem gerar perdas ao longo do processo, especialmente

para o modelo menos restritivo. Já para o modelo mais restritivo, os resultados foram como esperados, o *sharding* não garantiu a ordenação correta das operações de escrita, enquanto o *replication* garantiu que todas as réplicas tivessem o mesmo estado da composição local. No caso do *sharding*, para poder atestar a ordenação seria necessário adicionar um overhead no *proxy* e torná-lo não genérico, de modo que *timestamps* fossem adicionados a cada item, certificando sua ordenação posterior. Outro ponto importante evidenciado de forma geral no processo de adaptação foi uma perda de disponibilidade ao longo deste processo. O sistema tornou-se indisponível durante a adaptação, armazenando as requisições que chegavam durante o período, e posteriormente gerando um pico de respostas.

Para a análise em torno do tempo de resposta, a distribuição mostrou-se vantajosa quando o tempo de processamento de uma aplicação em sua composição local for superior à soma das novas variáveis de tempo incluídas na composição remota. Estas variáveis são: o tempo para processamento da requisição no *proxy* local, o tempo de rede para o *proxy* transmitir a nova requisição ao nó remoto, e o tempo de processamento em si da aplicação no nó remoto. Isso mostrou que não é simples que a composição distribuída seja superior à local, e portanto, podem haver cenários em que ela não é vantajosa, especialmente quando o modelo de consistência é menos flexível. Dessa forma, os experimentos realizados validaram esse entendimento para o cenário de *sharding* com operações em um único fragmento. Mostramos que o tempo de resposta da composição distribuída com *sharding* foi melhor que o local para operações de leitura com tempo de processamento $O(n \lg n)$, mas foi pior para operações de escrita com processamento $O(\lg n)$. Entretanto, quando aumentamos o tempo de processamento da escrita para $O(n \lg n)$, o *sharding* foi superior, validando a análise realizada.

Pelo fato dos experimentos terem sido realizados sobre casos bem específicos, ainda há diversos cenários que podem ser explorados. Trabalhos futuros também podem ser feitos em torno de aspectos como a adaptação da infraestrutura e não só a composição do sistema, e também o estudo de modelos de aprendizado de máquina para tornar o sistema de fato auto-adaptativo. Contudo, este trabalho atingiu seu objetivo como primeiro estudo de caso para a gestão transparente de estado em sistemas auto-adaptativos. Os resultados preliminares obtidos mostraram que, assim como para sistemas distribuídos no geral, é difícil saber quando distribuir o sistema desenvolvido, mesmo sob a perspectiva de sistemas auto-distribuídos. Mas mesmo assim é possível partir de uma aplicação projetada para executar localmente, e a partir da inserção de *proxies* de distribuição, reconfigurar a aplicação local para uma aplicação distribuída, mantendo o estado e tendo ganhos (em alguns cenários) no desempenho do sistema.

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

[4] 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

[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. *RE^X: 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, Distributed Systems, 3rd ed., distributed-systems.net, 2017.

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

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

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

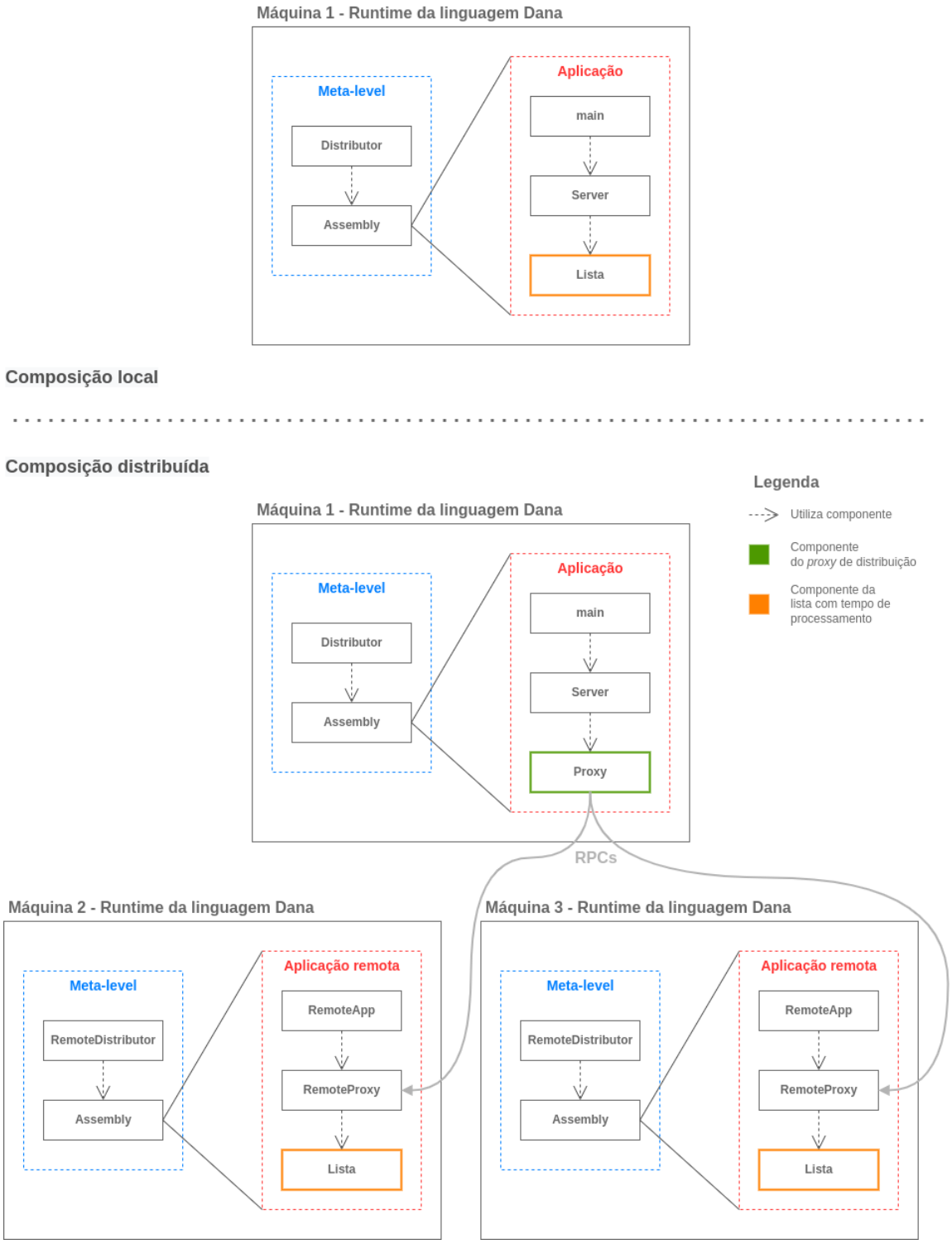

Figura 8: Arquitetura do sistema tanto na composição local quanto distribuída. Na execução local temos o *meta-level* atuando na aplicação alvo, enquanto na execução distribuída temos o *meta-level* atuando na aplicação remota em duas máquinas distintas. O *meta-level* e a aplicação alvo rodam todas dentro da *runtime* de Dana.