UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS ARARANGUÁ

CENTRO DE CIÊNCIAS, TECNOLOGIA E SAÚDE (CTS)

GRADUAÇÃO EM TECNOLOGIAS DA INFORMAÇÃO E COMUNICAÇÃO

Pedro Otávio Furtado Duarte

Um Estudo sobre a Automatização da Geração de Proxy para Sistemas Autodistribuídos

Araranguá, SC

2024

Pedro Otávio Furtado Duarte

Um Estudo sobre a Automatização da Geração de Proxy para Sistemas Autodistribuídos

Trabalho de Conclusão de Curso submetido ao curso de Tecnologias da Informação e Comunicação, do Centro de Ciências, Tecnologia e Saúde (CTS) do Campus Araranguá da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Tecnologias da Informação e Comunicação.

Orientador: Prof. Roberto Vito Rodrigues Filho, Dr.

Araranguá

2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Duarte, Pedro Otávio Furtado
Um Estudo sobre a Automatização da Geração de Proxy para Sistemas Autodistribuídos / Pedro Otávio Furtado Duarte ; orientador, Roberto Vito Rodrigues Filho, 2024.
46 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Araranguá, Graduação em Tecnologias da Informação e Comunicação, Araranguá, 2024.

Inclui referências.

1. Tecnologias da Informação e Comunicação. 2. Sistemas Autodistribuídos. 3. Sistemas Distribuídos Complexos. 4. Linguagens de Definição de Interface (IDL). I. Filho, Roberto Vito Rodrigues. II. Universidade Federal de Santa Catarina. Graduação em Tecnologias da Informação e Comunicação. III. Título.

Pedro Otávio Furtado Duarte

Um Estudo sobre a Automatização da Geração de Proxy para Sistemas Autodistribuídos

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel e aprovado em sua forma final pelo Curso Tecnologias da Informação e Comunicação.

Araranguá, 10 de dezembro de 2024.

Documento assinado digitalmente
Fernando Jose Spanhol
Data: 12/12/2024 09:02:00-0300
CPF: ***.656.419-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Fernando Jose Spanhol, Dr.

Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Roberto Vito Rodrigues Filho
Data: 11/12/2024 19:24:59-0300
CPF: ***.470.901-**
Verifique as assinaturas em https://v.ufsc.br

_____

Prof. Roberto Vito Rodrigues Filho, Dr.

Orientador

Universidade Federal de Santa Catarina

Documento assinado digitalmente
Marina Carradore Sergio
Data: 11/12/2024 21:07:21-0300
CPF: ***.746.359-**
Verifique as assinaturas em https://v.ufsc.br

_____

Prof.ª Marina Carradore Sérgio, Dra.

Universidade Federal de Santa Catarina

Documento assinado digitalmente
LUIZ FERNANDO BITTENCOURT
Data: 12/12/2024 09:20:01-0300
CPF: ***.738.069-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Luiz Fernando Bittencourt, Dr.

Universidade Estadual de Campinas

Agradecimentos

Primeiramente, agradeço aos meus pais que sempre me proveram o suporte necessário e o incentivo para os estudos durante toda minha caminhada acadêmica.

Gostaria de agradecer a minha tia Cirlei Pereira Furtado, que foi de grande importância para meu ingresso na UFSC.

Ao professor Dr. Roberto Vito Rodrigues Filho, por ter aceitado orientar este trabalho, pelas correções e pelo suporte no desenvolvimento.

Agradeço também a todos os meus amigos que fizeram parte dos meus anos como discente.

A vida é uma constante oscilação entre a ânsia de ter e o tédio de possuir.

Arthur Schopenhauer

Resumo

A complexidade dos sistemas atuais vem aumentando de forma gradativa, com isso, algumas soluções como o conceito de sistemas autonômicos foram propostas. Um exemplo importante da aplicação de computação autonômica na tentativa de gerenciar a complexidade na gerência e criação de sistemas são os chamados sistemas autodistribuídos, que surgiram para permitir o desenvolvimento de sistemas locais (i.e., que executam em um único processo) capazes de distribuir seus módulos internos à medida que se vê necessária a utilização de mais recursos computacionais para atender as necessidades dos usuários. Entretanto, o desenvolvimento de sistemas autodistribuídos ainda apresenta certa complexidade, à medida que é necessário o desenvolvimento manual de *proxies* (i.e., componentes de software) que são injetados na arquitetura local do software e que permite a autodistribuição. Dessa forma, este trabalho propõe uma abordagem de automatizar o processo de geração de *proxies*. A solução proposta utiliza uma Linguagem de Definição de Interface (IDL), que descreve as informações necessárias para a criação dos *proxies* de maneira automatizada, ou seja, com o mínimo de interferência humana no processo de implementação dos *proxies*. A partir dessa linguagem, foi desenvolvido o *ProxyGenerator*, um artefato em Python que analisa a interface indicada e informações complementares fornecidas pela IDL para gerar *proxies* que atendam diferentes padrões de interação, como fragmentação *(sharding)*, e replicação *propagate* e *alternate*. O trabalho conclui que a automatização da geração de *proxies* facilita o desenvolvimento de sistemas autodistribuídos, reduzindo a complexidade no desenvolvimento de componentes fundamentais para a implementação de sistemas autodistribuídos, promovendo um maior número de possibilidades de distribuição que podem ser exploradas em tempo de execução para localizar a melhor composição do sistemas distribuídos.

Palavras-chave: Sistemas Autodistribuídos; Linguagens de Definição de Interface (IDL); Sistemas Distribuídos Complexos.

Abstract

The complexity of current systems has been gradually increasing, and as a result, solutions such as autonomic systems have been proposed. An important example of applying autonomic computing to manage complexity in the design and management of systems is the so-called self-distributing systems. These systems emerged to enable the development of local systems (i.e., software executing in a single process) capable of distributing their internal modules as more computational resources are demanded to meet user needs. However, the development of self-distributing systems still presents some complexity, as it requires a human-led development of proxies (i.e., software components) that are injected into the software's local architecture and enable self-distribution. Therefore, this work proposes an approach to automate the proxy generation process. The proposed solution uses an Interface Definition Language (IDL) to describe the necessary information for automatically creating proxies, minimizing human intervention in the proxy implementation process. Based on this language, ProxyGenerator was developed, a Python artifact that analyzes the specified interface and additional information provided by the IDL to generate proxies that support different interaction patterns, such as sharding and the propagate and alternate replication strategies. The study concludes that automating proxy generation facilitates the development of self-distributing systems by reducing the complexity involved in creating essential components. This promotes a broader range of distribution possibilities that can be explored at runtime to identify the best composition for the distributed system.

Keywords: Self-distributed systems; Interface Definition Language (IDL); Complex Distributed Systems.

Lista de ilustrações

Figura 1 – Comparação de custos de implementação. 7
Figura 2 – Exemplo de implementação de RPC em Dana. 9
Figura 3 – Processo de autodistribuição utilizando *proxy*. 11
Figura 4 – Fluxograma de geração de *proxy*. 18
Figura 5 – Definição da linguagem utilizando notação EBNF. 20
Figura 6 – Exemplo de arquivo IDL para uma interface de lista. 21
Figura 7 – Interações entre funções. 22
Figura 8 – Cabeçalho do *proxy*. 24
Figura 9 – Corpo do *proxy* (implementações dos métodos). 25
Figura 10 – Rodapé do *proxy*. 26
Figura 11 – Adicionando novo método de interação (variável *interactionList*). . . . 26
Figura 12 – Experimento de Divergência de Desempenho - Carga A 29
Figura 13 – Experimento de Divergência de Desempenho - Carga B 30
Figura 14 – Experimento de Carga 31

Sumário

1 INTRODUÇÃO 1
1.1 Justificativa 2
1.2 Objetivos 3
1.2.1 OBJETIVO GERAL 3
1.2.2 OBJETIVOS ESPECÍFICOS 3
1.3 Metodologia 3
1.4 Organização do trabalho 5

2 REFERENCIAL TEÓRICO 6
2.1 Linguagem Específica de Domínio (DSL) 6
2.2 Chamada de Procedimento Remoto (RPC) 8
2.3 Proxy 9
2.4 Sistemas Autodistribuídos 10

3 TRABALHOS RELACIONADOS 13
3.1 Linguagem Específica de Domínio (DSL) 13
3.2 Chamadas de Procedimento Remoto (RPC) 14
3.3 Sistemas Autodistribuídos 15

4 ARQUITETURA DA SOLUÇÃO 17
4.1 Visão geral da arquitetura 18
4.2 Definição da Linguagem Específica de Domínio 19
4.3 *ProxyGenerator*: Gerador automatizado de *proxy* 20
4.4 Estrutura de um *proxy* 23
4.5 Expansão de métodos de interação 24

5 RESULTADOS E DISCUSSÕES 27
5.1 Cenários 27
5.2 Resultados 28

6 CONCLUSÃO 32

REFERÊNCIAS 34

1 INTRODUÇÃO

Sistemas contemporâneos possuem dezenas de milhões de linhas de código e precisam de profissionais habilidosos para instalar, configurar e manter, o que leva ao aumento na complexidade na criação e gestão de sistemas contemporâneos (BLAIR, 2018). Visando uma forma de gerir e reduzir essa complexidade, a solução proposta é a utilização de sistemas autonômicos, que operaram autonomicamente a partir de políticas e objetivos definidos em alto nível pelos administradores. Com isso, um dos benefícios da utilização desse tipo de sistema, é a redução da necessidade de interação humana observando a operação do sistema a todo momento, visto que o sistema poderá se adaptar de acordo com a carga de trabalho, falhas de hardware ou software e até mesmo lidar com erros durante sua execução. Os sistemas autonômicos baseiam-se em 4 propriedades: autoproteção, auto-otimização, autocura e a autoconfiguração (KEPHART; CHESS, 2003).

A partir do conceito de sistemas autonômicos, o trabalho de Rodrigues-Filho e Porter (2022), entitulado *"Hatch: Self-distributing systems for data centers"* introduz o conceito de sistemas autodistribuídos, utilizando tecnologias que implementam um modelo baseado em componentes (como OpenCOM de Coulson et al. (2004) e Dana de Porter e Rodrigues-Filho (2021)). Sistemas autodistribuídos são sistemas que distribuem componentes, que originalmente foram projetados para serem executados localmente, para outros processos em máquinas externas em tempo de execução e de forma ininterrupta, à medida que o sistema demanda mais recursos computacionais para operar.

Na abordagem descrita por Rodrigues-Filho e Porter (2020), os autores exploram, de forma conceitual, a utilização de *proxy* para realizar a distribuição de componentes em tempo de execução. Essa abordagem funciona da seguinte maneira: em tempo de execução o framework de sistemas autodistribuídos substitui o componente local por um *proxy*, ou seja, um componente que implementa a mesma interface do componente que ele substitui, mas que sua única função é encaminhar requisições que chegam a ele para réplicas do componente, que agora executam em processos remotos (em máquinas externas). Além disso, o *proxy* também é responsável pela consistência do estado do componente, utilizando um modelo de consistência de dados apropriado para o dado em questão.

A respeito da auto-otimização, o *framework* SDS de Rodrigues-Filho et al. (2023) combina o modelo baseado em componentes citado anteriormente com um agente inteligente que toma decisões adaptativas em tempo de execução escolhendo a melhor composição considerando métricas como tempo de resposta e consumo de energia.

Entretanto, a geração desses *proxies*, até o presente momento, é feita de forma manual pelo desenvolvedor, adicionando uma camada a mais de complexidade na construção do

sistema. Portanto, este trabalho visa estudar e propor uma forma de automatizar a geração desses *proxies* a fim de tornar mais viável a implementação de sistemas autodistribuídos.

Este capítulo está organizado em quatro seções. A primeira seção apresenta a justificativa para a elaboração deste trabalho. A segunda seção contém os objetivos do trabalho, segmentados entre objetivo geral e objetivos específicos. Na terceira seção encontra-se a metodologia utilizada para o desenvolvimento deste trabalho. A quarta e última seção deste capítulo contém a estrutura do trabalho como um todo.

1.1 Justificativa

A geração manual de *proxy* que atualmente é demandada pelo atual *framework* SDS é parte importante do processo de desenvolvimento de sistemas autodistribuídos. Dessa forma, visando aumentar a adoção deste *framework* e reduzir a dificuldade de seu uso por desenvolvedores, é importante a automatização deste processo.

A complexidade da criação manual de *proxies* se origina nos diversos aspectos da distribuição do componente que os desenvolvedores precisam considerar em tempo de projeto, tais como: qual algoritmo será utilizado para manter a consistência do estado do componente durante a sua distribuição, necessária para permitir que o sistema opere de maneira confiável e eficiente, evitando estados inconsistentes entre réplicas, que poderiam causar falhas durante a execução ou mesmo leituras inconsistentes. Além da definição dos detalhes da comunicação entre o *proxy* e o componente original (discutidos no Cap. 2), tais como: a lista de IPs dos servidores remotos, a maneira como as respostas serão lidas e analisadas, entre outros.

Visando a automatização da geração de *proxies*, a utilização de Linguagens Específicas de Domínio (DSLs) e, em particular, a criação de uma Linguagem de Definição de Interface (IDL) é uma abordagem promissora, devido ao o conceito de IDL auxiliar a preencher a lacuna existente entre a criação de um algoritmo para a geração automática de *proxy* e a manutenção da consistência do estado da aplicação, visto que permite que o usuário possa descrever com precisão a interface do componente criado, permitindo assim, que o algoritmo de geração autonômica tenha acesso às informações necessárias para sua execução adequada, tais como a informação de se o método, em algum momento, altera o estado de algum atributo da classe (impacta na classe).

Portanto, o trabalho proposto visa explorar a utilização de Linguagens Específicas de Domínio (DSL), mais especificamente linguagem de definição de interface (IDL), como parte crucial no processo de automatização da geração de *proxy*, reduzindo assim, o processo de desenvolvimento de sistemas autodistribuídos, e com isso, facilitando a adoção do *framework* de SDS.

1.2 Objetivos

1.2.1 OBJETIVO GERAL

O objetivo geral deste trabalho consiste em facilitar o desenvolvimento de sistemas autodistribuídos utilizando o *framework* SDS, por meio de um artefato de software que automatiza a geração de *proxy* utilizando Linguagens de Definição de Interface. Este objetivo principal se desdobra nos seguintes objetivos específicos.

1.2.2 OBJETIVOS ESPECÍFICOS

Para a execução do projeto, pode-se citar a necessidade da realização das seguintes atividades:

- Definir uma gramática para a Linguagem de Definição de Interface Desenvolver uma gramática para a Linguagem de Definição de Interface (IDL), detalhando os elementos necessários para descrever interfaces de componentes distribuíveis;
- Projetar e implementar um protótipo funcional de um gerador de *proxies* automatizado utilizando a IDL como base;
- Validar o gerador de *proxies* em cenários reais, avaliando o desempenho dos *proxies* gerados com base em métricas como tempo de resposta e consistência do estado.

Dessa forma, o objetivo deste estudo é estabelecer uma IDL para conter informações da interface dos componentes que serão distribuídos, e assim, criar *proxies* que estabelecerão padrões de interação entre os componentes locais e os componentes que serão distribuídos, preservando a integridade e consistência do estado dos componentes distribuídos durante esse procedimento. Além disso, o objetivo é desenvolver um protótipo funcional de um gerador de *proxies* e aplicá-lo em uma situação real, com uma interface de um sistema real, para produzir *proxies* aptos a distribuir componentes durante a execução. Finalmente, a avaliação do trabalho será realizada medindo a eficiência de cada *proxy* gerado, utilizando métricas de desempenho (como o tempo de resposta) para validar o protótipo desenvolvido.

1.3 Metodologia

O estudo realizado é de natureza exploratória, realizado de forma empírica e cíclica, cujo objetivo é reduzir a complexidade do desenvolvimento de sistemas autodistribuídos. A metodologia empírica visa a legitimidade de teorias e hipóteses através de experimentos práticos (OLIVEIRA, 2011). Desta forma, este trabalho foi conduzido com o objetivo de validar, de forma empírica, o artefato de software que gera *proxies* de forma automatizada.

A pesquisa de trabalhos relacionados utilizou *strings* de busca baseadas em palavras-chave do domínio estudado, tais como Linguagem de Definição de Interface, Linguagem Específica de Domínio e Sistemas Autodistribuídos, em bases confiáveis como *Scopus* e *IEEE Xplore*. A seleção considerou títulos, resumos e número de citações, garantindo a relevância e confiabilidade das obras, com foco em qualidade e aplicabilidade ao estudo.

Para a parte prática, referente a construção do protótipo, este estudo utiliza a metodologia de Pesquisa em Ciência de Design (Design Science Research). Essa metodologia é comumente utilizada em disciplinas como Ciência da Computação, Sistemas de Informação e Engenharia de Software, tendo como principal objetivo solucionar problemas práticos, geralmente relacionados à informação e inovação.

A abordagem DSR segue um ciclo iterativo que compreende as seguintes etapas:

- Identificação do Problema;
- Definição dos Requisitos;
- Design da Solução;
- Desenvolvimento e Implementação;
- Avaliação e Validação;
- Comunicação dos Resultados;
- Refinamento Iterativo (se necessário);

As etapas presentes no Quadro 1 compõem uma estrutura sólida para a condução da pesquisa, visto que encaminham o pesquisador desde a identificação do problema até a comunicação dos resultados.

A avaliação do trabalho foi realizada em um cenário real, utilizando sistemas auto-distribuídos com os *proxies* gerados automaticamente. Os experimentos foram realizados em laboratório e envolveram duas máquinas conectadas via uma rede local Wi-Fi 2.4GHz, representando um ambiente distribuído mais próximo de condições reais, principalmente considerando atrasos e latências de rede. Nesses experimentos, foram analisados diferentes padrões de interação, avaliando-se o desempenho e a funcionalidade dos *proxies* na prática. A abordagem empírica permitiu validar a eficácia do artefato *ProxyGenerator* e demonstrou como os *proxies* gerados podem ser aplicados diretamente em sistemas reais, atendendo às demandas de distribuição dinâmica e consistência do estado.

Para realizar os experimentos, utilizou-se de uma versão adaptada do código de Araujo et al. (2021), que está disponível em um repositório público[1].

[1] Repositório do GitHub: https://github.com/k1vz/agp_sds

Quadro 1 – Etapas da metodologia preenchidas.

Etapa	**Descrição**
Identificação do Problema	Investigar uma maneira de facilitar o desenvolvimento de sistemas autodistribuídos utilizando o *framework* SDS
Definição dos Requisitos	Definir os requisitos e padrões de acordo com o problema encontrado
Design da Solução	Reunir hipóteses encontradas e identificar a viabilidade de cada
Desenvolvimento e Implementação	Elaborar um protótipo para a geração automatizada de *proxy* utilizando uma IDL
Avaliação e Validação	Realizar experimentos indicando a usabilidade do artefato gerado
Comunicação dos Resultados	Apresentação dos resultados em forma de gráficos
Refinamento Iterativo	Reciclagem do processo de pesquisa, se necessário, utilizando das informações adquiridas a fim de encontrar melhoras

Fonte: O autor, 2024.

1.4 Organização do trabalho

Este trabalho está organizado em cinco capítulos a fim de auxiliar o entendimento do leitor a respeito do que foi desenvolvido, sendo eles:

- O segundo capítulo, composto pelo referencial teórico, o qual embasa os tópicos abordados a diante, explorando conceitos de Linguagens Específicas de Domínio (DSL), Linguagem de Definição de Interface (IDL), e Sistemas Autodistribuídos (SDS);
- No terceiro capítulo, são analisados trabalhos relacionados que embasam a proposta, como os desafios de gerar *proxies* manuais em sistemas autodistribuídos;
- O quarto capítulo descreve a arquitetura da solução proposta, detalhando o *Proxy-Generator* e a definição da linguagem utilizada;
- O Quinto capítulo apresenta os resultados, segmentados em cenários e resultados, discutindo as informações obtidas;
- O Sexto capítulo é composto pela conclusão e sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados os conceitos fundamentais que sustentam o trabalho, tais como: Linguagens Específicas de Domínio (DSL), Chamada de Procedimento Remoto (RPC) e Sistemas Autodistribuídos (SDS). Esses conceitos são cruciais para compreender a proposta de automatização da geração de *proxies*, visto que definem as bases teóricas do assunto abordado. A abordagem inicia com uma explicação sobre DSLs, destacando sua aplicabilidade no desenvolvimento de linguagens adaptadas a contextos específicos, seguida pela explicação sobre RPCs, que fornecem o mecanismo de comunicação remota essencial utilizado nos *proxies*, por fim, são embasados os conceitos de sistemas autodistribuídos e do *framework* SDS. Ao compreender os conceitos citados, é possível assimilar o que foi desenvolvido no presente trabalho.

2.1 Linguagem Específica de Domínio (DSL)

Ao oposto de uma Linguagem de Propósito Geral (GPL)[1] como Java ou C++ que podem ser aplicadas em diferentes tipos de problemas, uma Linguagem Específica de Domínio (DSL)[2] é uma linguagem de programação adaptada a um contexto em particular (RAJA; LAKSHMANAN, 2010). Geralmente são de um nível mais elevado do que as demais linguagens de programação, o que indica que a sua sintaxe é formada de maneira próxima à linguagem humana[3], porém são construídas em uma linguagem de nível menor (FENG et al., 2024). Stojanović, Plehati e Kovačević (2024) citam alguns exemplos de DSL como: linguagens de marcação (HTML), linguagens de troca de dados (JSON) e linguagens de consultas a bancos de dados (SQL).

De acordo com Krasanakis e Symeonidis (2024), uma DSL provém a utilização de operações e conceitos de um domínio específico, facilitando a utilização do sistema pelos usuários letrados no contexto em que foi projetado. Os componentes de um DSL podem-ser combinados através de interfaces de usuário textuais ou gráficas, representando um caso de uso complexo no campo abordado. Segundo Stojanović, Plehati e Kovačević (2024) as maneiras mais comuns de implementação de uma DSL são: interpretada, compilada, pré-processada, incorporada.

Para Raja e Lakshmanan (2010), há algumas características que toda DSL deve conter: ser simples de modo a reduzir o tempo de aprendizado; ser construída sobre o vocabulário de domínio do usuário; a sintaxe fornecida pela DSL deve esconder o aspecto

[1] General-Purpose Language
[2] Domain-Specific Language
[3] Quanto maior o nível da linguagem, mais próximo da linguagem humana ela está

de programação inerente à aplicação do cliente; ter o custo para sua implementação menor comparado a uma GPL. Na Figura 1 é exibido um gráfico comparando os custos de implementação de um software utilizando uma metodologia com DSL e uma metodologia tradicional, a partir disso, pode-se perceber um custo inicial mais elevado em comparação com a metodologia tradicional, porém a situação se inverte ao decorrer do tempo.

Fonte: (DEURSEN; KLINT; VISSER, 2000 apud KLEIN, 2015).

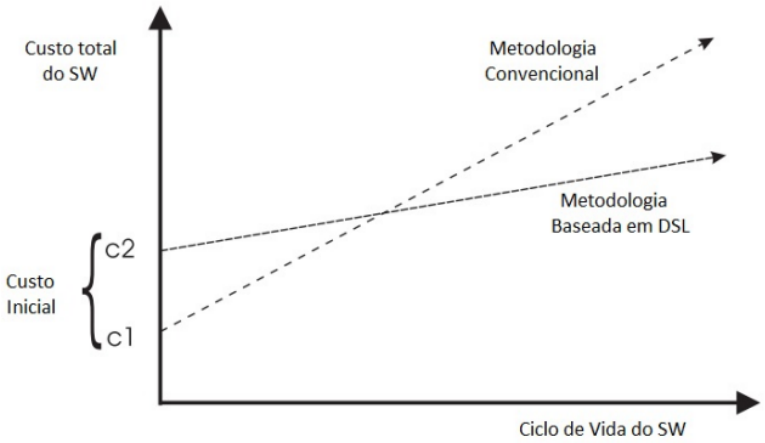

Figura 1 – Comparação de custos de implementação.

Feng et al. (2024) explica que um dos principais benefícios da adoção de uma DSL é um aumento na produtividade dos desenvolvedores e o auxílio na comunicação com os usuários, visto que para um usuário lecionado na área consiga realizar a leitura e compreensão do código, é necessária pouca ou nenhuma documentação.

Segundo Kelly e Tolvanen (2007), toda DSL deve conter alguns elementos de de estrutura, tais como: a sintaxe (abstrata e concreta) e a semântica. A sintaxe abstrata é um metamodelo que define as ideias e as diretrizes da modelagem da DSL, já a sintaxe concreta pode-ser definida como a notação gráfica das classes da sintaxe abstrata, contudo, é a semântica quem provê o significado de cada elemento da DSL.

Para Stern (2000), Linguagem de Definição de Interface (IDL) descreve os tipos, métodos e formatos de dados que os programas ou sistemas devem usar ao se comunicarem entre si, utilizadas para descrever e criar uma DSL.

Para Stern (2000), uma IDL é bastante interativa, pois em um ambiente de uma IDL (ou um interpretador), é possível escrever um comando e ver o resultado imediatamente, o que não acontece em linguagens compiladas como C, por exemplo. Onde é necessário que o desenvolvedor escreva todo o código para somente depois de compilado, que ele possa ver o resultado da sua execução. Wain e Ashworth (2005) explicam que IDLs são geralmente utilizadas para implementar sistemas que utilizam de Chamadas de Procedimento Remoto

(RPC), um exemplo de IDL de grande relevância é o CORBA, que foi desenvolvido para simplificar a troca de dados entre sistemas distribuídos heterogêneos.

O conceito de Linguagem Específica de Domínio possui grande importância dentro do contexto do presente trabalho, especialmente considerando o conceito de Linguagem de Definição de Interface, visto que foi utilizada para descrever informações essenciais para o processo de geração de *proxy*, porém que não estão presentes na interface original, e com isso, foi necessário elaborar uma IDL para obter essas informações.

2.2 Chamada de Procedimento Remoto (RPC)

De acordo com Geng et al. (2024), o RPC é uma abstração que permite esconder os detalhes da comunicação tornando-as semelhante às chamadas locais. É uma tecnologia de desenvolvimento de sistemas distribuídos que permite a comunicação entre componentes remotos utilizando alto nível, tomando do desenvolvedor a responsabilidade de realizar a integração entre as diferentes de arquitetura, protocolos utilizados, dentre outros.

Segundo RH et al. (2024), quando uma chamada RPC é executada e chega ao servidor, é realizada a ação solicitada pelo cliente e automaticamente iniciada uma rotina de despacho, devolvendo o conteúdo ao cliente. Como explica Cruz (2013), "a estrutura RPC define um esquema de encapsulamento de todas as funções associadas à conexão remota num pedaço de código chamado de *stub*". Durante todo o processo, ambos os lados acreditam que tudo está acontecendo localmente.

Chen et al. (2023) explicam que a utilização de RPC permite a criação de aplicações em rede utilizando modelos de programação simples e possuem familiaridade, visto que é necessário que sejam definida a interface do programa em um arquivo, o qual será transformado em um *stub*, o qual é responsável por organizar os parâmetros da conexão e interagir com a camada de transporte (os sockets TCP/IP). Kumar et al. (2024) aponta que alguns dos *frameworks* mais populares de RPC são o Apache Thrift e o gRPC.

Os *stubs* são gerados utilizando uma linguagem de definição de interface (IDL), tais como: SunRPC, CORBA e Apache Thrift. IDL nada mais é que uma DSL específica para RPC, utilizada para anotar as informações necessárias na geração de *stubs*. Na Figura 2 há um exemplo de implementação de RPC em uma função de soma utilizando a linguagem Dana: a função *somaOriginal*, como o próprio nome expressa, é a versão primitiva, que seria executada de forma local; já a função *somaRPC* é a implementação da mesma função em um sistema distribuído, que seria executada em outro dispositivo.

No *framework* SDS os *stubs* são chamados de *proxies*, que são utilizados para encaminhar as requisições aos nós remotos, com isso, parte do processo de geração de *proxy* envolve a compreensão do funcionamento de um RPC e como ele é construído.

Fonte: O autor, 2024.

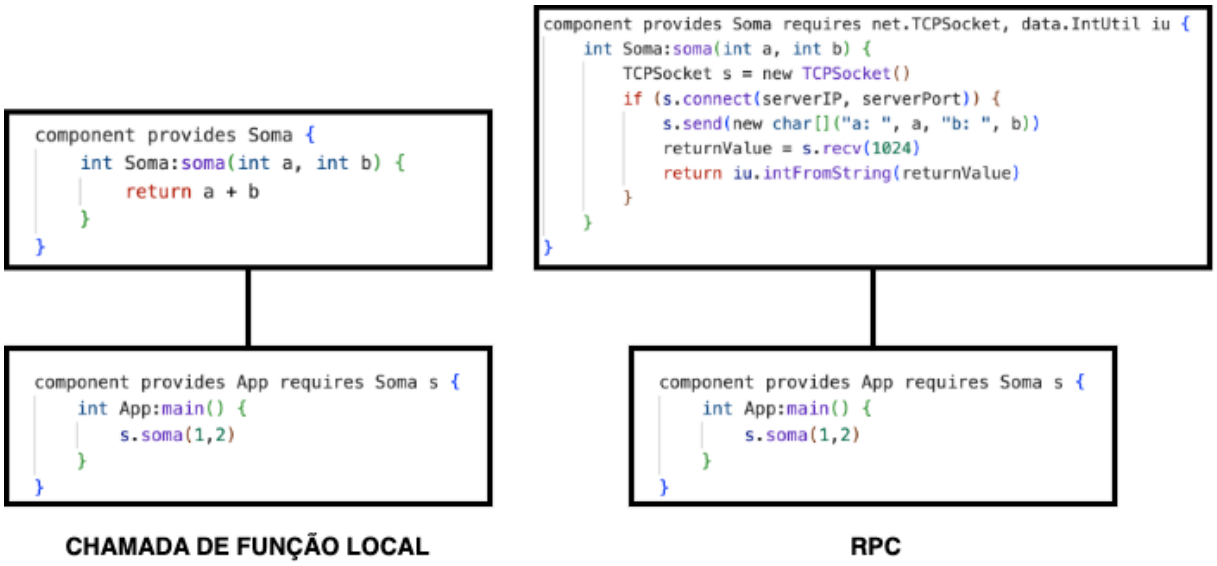

Figura 2 – Exemplo de implementação de RPC em Dana.

2.3 Proxy

Dentro do conceito de Sistemas Autodistribuídos, um *proxy* é um componente intermediário que atua como uma representação de outro componente em sistemas distribuídos. Ele simula a interface do componente original (provendo e esperando os métodos com as mesmas assinaturas), permitindo que chamadas e requisições sejam encaminhadas para destinos remotos sem que o sistema precise conhecer detalhes da comunicação ou a estratégia de consistência de estado utilizada entre as réplicas. Em sistemas autodistribuídos, o *proxy* é essencial para garantir que a distribuição de componentes entre diferentes nós ocorra de forma transparente, preservando a consistência do estado e a continuidade das operações. Além disso, ele encapsula a lógica de comunicação, utilizando técnicas a tecnologia de RPC para facilitar a interação entre componentes localizados em diferentes máquinas, tornando-se uma ferramenta crucial para a escalabilidade e eficiência desses sistemas (RODRIGUES-FILHO et al., 2022).

Com isso, é responsabilidade do *proxy* o comportamento de divisão e paralelização de um sistema autodistribuído, como representado na Figura 3. O termo *proxy* no âmbito dos sistemas autodistribuídos é utilizado como uma analogia ao *proxy* de redes de computadores, visto que atua de maneira semelhante, sendo responsável por encaminhar as requisições para um nó remoto, de acordo com a composição utilizada. Entretanto, a principal diferença em relação ao conceito de *proxy* de redes de computadores é não estar do lado de fora do sistema, e sim no lugar de um componente, dentro do sistema (ROSS, 2008).

O *proxy* é responsável por implementar o *stub* para a comunicação utilizando RPC, para isso ocorrer, o *proxy* precisa simular a interface do componente original, fazendo com que o *proxy* continue aparentando ser o componente implementado local, porém encaminhando as requisições a nós remotos, permitindo assim, a migração para uma arquitetura de sistemas distribuídos de forma transparente (RODRIGUES-FILHO et al., 2021). Em sistemas autodistribuídos, essa abordagem possibilita que os componentes sejam redistribuídos dinamicamente, aproveitando melhor a infraestrutura disponível.

Além disso, o *proxy* é responsável por gerenciar a consistência do estado do componente. Para isso, ele utiliza modelos de consistência de dados que asseguram que as alterações realizadas em um nó remoto sejam refletidas corretamente nos demais. Este processo inclui a coordenação entre *proxies* e réplicas remotas, garantindo que o sistema mantenha sua integridade mesmo em cenários de alta complexidade.

2.4 Sistemas Autodistribuídos

Segundo Woodfin (2002), no modelo de sistemas distribuídos a responsabilidade para a distribuição e paralelização da aplicação é dividida entre o programador e o sistema em tempo de execução. Em teoria, o programador irá descrever como o programa irá se dividir e paralelizar, além da maneira com que irá se recombinar.

Para concluir o processo de realocação do componente, no nó de destino, é necessário que haja um *RemoteDist* em execução, visto que ele é o responsável por criar um *proxy* local (no nó de destino) e conectá-lo ao *proxy* original, permitindo assim, receber e manusear os dados no nó remoto (DIAS et al., 2022).

A Figura 3 representa a realocação de um componente que utiliza o estado dos atributos *(stateful)* por um *proxy*, na primeira etapa o componente original (Comp B) ainda está em execução, com a comunicação entre eles sendo regida pela **Interface I**; no segundo estágio, o algoritmo de adaptação pausa o recebimento de requisições nesse componente a fim de evitar que o estado do componente seja alterado durante o processo de adaptação; na terceira etapa o *proxy* (Comp C) já está carregado em memória e recebe uma cópia do estado do componente original (Comp B); por fim, na última etapa a comunicação entre o Comp A e o Comp C é estabelecida e o algoritmo de adaptação retoma o recebimento das requisições (RODRIGUES-FILHO et al., 2022).

De acordo com Rodrigues-Filho e Porter (2022), há diferentes métodos de interação que podem ser utilizados para definir a forma que as requisições serão encaminhadas ao nó remoto, sendo elas `relocação` (*Alternate*): Substitui um componente real por um *proxy*, que encaminhará a um nó remoto, alternando o nó de destino, `replicação` (*Propagate*): Realocação, porém para mais de um nó remoto e `fragmentação`(*Sharding*): Divide o componente original por tipos de dados e o aloca em diferentes nós remotos. Os quais

Fonte: Rodrigues-Filho et al. (2022).

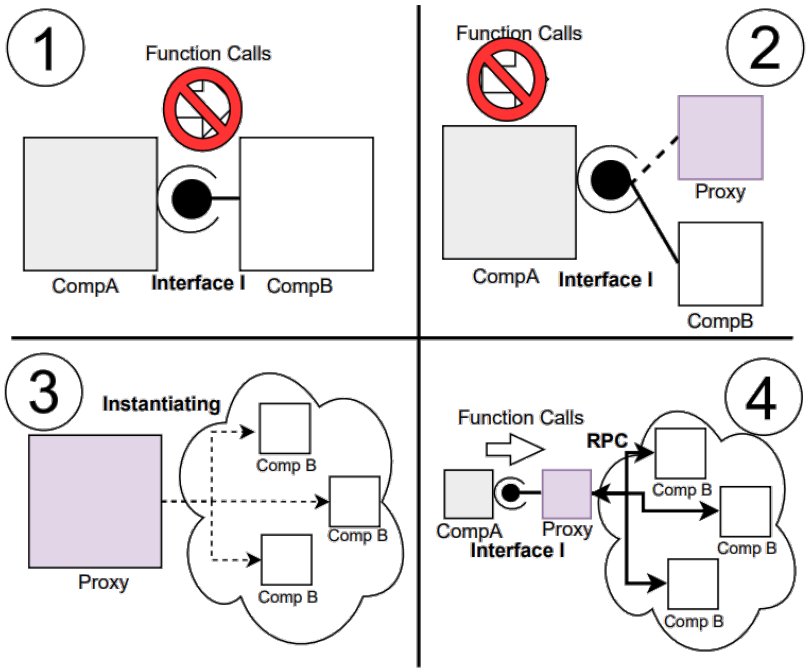

Figura 3 – Processo de autodistribuição utilizando *proxy*.

serão abordados com mais profundidade a seguir.

Rodrigues-Filho et al. (2022) indica que de acordo com os experimentos realizados, o conceito de sistemas autodistribuídos com o controle de estado transparente permite flexibilidade o bastante para aplicações componentizadas explorarem todo o potencial do conceito borda-nuvem *(Edge-Cloud Continuum)*, que consiste em integrar de forma fluida e adaptativa os recursos computacionais localizados em infraestruturas de borda *(edge)* e em nuvens *(cloud)*, facilitando a utilização de sistemas distribuídos, e com isso a escalabilidade horizontal.

Chaurasia, Verma e Verma (2024) explicam que a arquitetura de sistemas distribuídos oferece diversas vantagens, tais como: alta performance, confiabilidade, modularidade, escalabilidade e tolerância à falhas, permitindo diversas topologias para conexão.

Como aponta Kina et al. (2023), devido à utilização de computação de borda (recursos mais próximos do usuário) a computação autonômica não possui as mesmas dificuldades relacionadas à latência e atrasos na comunicação que sistemas em nuvem para aplicações em larga escala. Araujo et al. (2021) apontam que em SDS a implementação dos métodos é realizada pelo *proxy*, com isso, todas as chamadas são encaminhadas aos nós remotos, mantendo a consistência dos dados. Em decorrência disso, o consumo de recursos na borda é menor, pois todo o processamento é feito na nuvem.

Como citado anteriormente na seção 2.4, no *framework* SDS a implementação dos métodos é realizada através de um *proxy* possuindo a mesma interface do componente real, porém apenas encaminhando as requisições para um nó remoto (através da utilização de RPC), fazendo com que o processamento seja realizado pelo componente remoto. Para a escolha de qual será o nó remoto de destino da requisição, o sistema pode utilizar diferentes métodos de interação, tais como: fragmentação (*sharding*), replicação (*propagate*), relocação (*alternate*).

Segundo Araujo et al. (2021) na fragmentação (*sharding*) o conjunto de dados é particionado em diferentes réplicas, de forma horizontal, frequentemente utilizada em bancos de dados. Tem como principal vantagem reduzir o uso de recursos computacionais na borda e permitir o balanceamento de cargas entre as réplicas, principalmente vantajoso quando o sistema possui uma alta taxa de requisições e também quando o componente em questão não necessita de uma consistência forte de estado, conforme descrito por Oliveira et al. (2022). No caso do SDS, a estratégia de particionamento de estado implementa um hash multiplicativo, que utiliza como chave algo relacionado ao item transferido, como o seu valor (KINA et al., 2023).

Araujo et al. (2021) também explicam que a replicação (*propagate*) também permite a escala horizontal, dividindo o processamento do componente original em diferentes hosts remotos, porém todos os nós remotos possuem o conteúdo completo. Esse método de interação realiza *broadcasts*[4] para transmitir as informações, com isso, como apontado por Preguiça et al. (2010), um dos possíveis problemas que podem acontecer referente à conservação dos dados é um dos nós não receber uma requisição que altere seu conteúdo, resultando em um estado diferente dos demais, a fim de evitar problemas como esse, são utilizadas estratégias de consistência de estado, como por exemplo o *Total Order Broadcast*, que garante a ordenação das mensagens recebidas por todos os nós, portanto caso uma mensagem seja entregue fora da ordem, ele se encarrega de corrigir sua posição (KLEPPMANN, 2018).

O método de interação de relocação (*alternate*) funciona de maneira semelhante à um *load balancer*, distribuindo as requisições recebidas utilizando uma determinada estratégia de balanceamento, como explica Zhang et al. (2018). No caso do SDS, a técnica utilizada é o *round robin*, que é um dos métodos mais simples e de fácil implementação, onde as requisições são distribuídas em ordem aos nós remotos (a primeira vai ao primeiro nó, a segunda ao segundo nó, e assim por diante). Assim como no conceito de *load balancer* pode ser implementado, o *framework* SDS utiliza de uma técnica que envolve um cálculo matemático, chamado hash, para definir qual será o nó escolhido para destino do envio da requisição.

[4] Envio simultâneo de dados para diversos clientes

3 TRABALHOS RELACIONADOS

Neste capítulo serão abordados trabalhos nas áreas de DSL, IDL e Sistemas Autodistribuídos (SDS). Panayiotou et al. (2024) apresentam a SmAuto, uma DSL que facilita o desenvolvimento de sistemas inteligentes via *low-code*. Em IDL, destaca-se o Apache Thrift, *framework* que permite comunicação eficiente entre linguagens via RPC. Nos SDS, Filho e Porter (2020) discutem o uso de *proxies* para distribuir componentes, enquanto Rodrigues-Filho e Porter (2022) apresentam o Hatch, sistema que realoca, replica e fragmenta componentes. Woodfin (2002) propõe o modelo Self-Distributing Computation (SDC), que divide responsabilidades entre programador e sistema para otimizar a distribuição de tarefas.

Este estudo seguiu critérios de seleção rigorosos para garantir a qualidade dos estudos identificados. A pesquisa foi realizada com base em diferentes *strings* de busca, formuladas a partir das principais palavras-chave relacionadas ao domínio abordado: Linguagem de Definição de Interface, Linguagem Específica de Domínio e Sistemas Autodistribuídos. As fontes consultadas incluem bases confiáveis de artigos acadêmicos e publicações técnicas como por exemplo *Scopus* e *IEEE Xplore*. O método de seleção das obras incluiu a análise do título e do resumo de cada publicação. Além disso, o número de citações foi utilizado como critério complementar, servindo como um indicador da confiabilidade e relevância de cada obra dentro da comunidade acadêmica e técnica. Essa abordagem permitiu identificar referências alinhadas aos objetivos do estudo, priorizando qualidade e aplicabilidade prática.

3.1 Linguagem Específica de Domínio (DSL)

No geral, as DSLs são amplamente exploradas como uma ferramenta direcionada para simplificar o trabalho em contextos específicos. Pode-se citar como exemplo o trabalho de Panayiotou et al. (2024) que propõe uma implementação de DSL (SmAuto) a fim de integrar as tecnologias de internet das coisas (IoT) e sistemas ciber físicos (CPS), permitindo a utilização de uma abordagem *low-code*[1], a qual facilita o processo de democratização dessa tecnologia, propiciando o emprego dessas tecnologias na criação de casas e indústrias inteligentes.

Além disso, Diniz (2016) propõe uma DSL para a Abstração de Solução de Processamento de Eventos Complexos, realizando a criação de uma interface gráfica para a construção de regras de Processamento de Eventos Complexos (CEP) utilizando a aborda-

[1] Abordagem de desenvolvimento de software que envolve o mínimo de codificação.

gem Model-Driven Development (MDD). O objetivo principal é reduzir a complexidade do desenvolvimento de soluções CEP, permitindo que usuários não especialistas possam criar e configurar regras através de uma interface intuitiva, contribuindo para iniciativas como Cidades Inteligentes, onde a tomada de decisões em tempo real é essencial.

Os trabalhos apresentados destacam a flexibilidade e a adaptabilidade das DSLs para resolver problemas de domínios variados. No entanto, esses trabalhos focam em contextos amplamente distintos e não abordam a automatização da geração de *proxies* para sistemas autodistribuídos. Em contraste, este trabalho utiliza uma DSL voltada exclusivamente para a definição de interfaces (IDL), introduzindo um formato estruturado que permite automatizar a criação de *proxies*, reduzindo significativamente a complexidade associada ao desenvolvimento de sistemas autodistribuídos.

3.2 Chamadas de Procedimento Remoto (RPC)

O artigo de Rocha (2017) explora como a combinação das tecnologias de Java (linguagem de programação) e CORBA (Common Object Request Broker Architecture, *middleware*) permite a integração e a comunicação entre componentes desenvolvidos em diferentes linguagens de programação e hospedados em ambientes heterogêneos. A utilização do CORBA proporciona um *framework* para a interoperabilidade entre sistemas distribuídos, facilitando a comunicação entre objetos remotos por meio de um protocolo padronizado. O que se torna particularmente relevante em contextos industriais e corporativos, onde há necessidade de coordenação entre diversas plataformas tecnológicas. A abordagem discutida no artigo destaca as vantagens da utilização de IDLs para abstrair a complexidade dos serviços distribuídos e facilitar a manutenção e evolução dos sistemas.

Em meio ao mercado, algumas implementações de IDL que utilizam RPC podem ser encontradas, tais como: o Apache Thrift, desenvolvido pelo Facebook em 2020, atualmente é um projeto de código aberto mantido pela Apache Software Foundation. Segundo Chadokar (2023), ele foi projetado para permitir uma forma eficiente e escalável de comunicação entre diferentes linguagens de programação.

Diferentemente das abordagens citadas que focam na interoperabilidade e no desempenho de sistemas distribuídos genéricos, o presente trabalho utiliza o conceito de RPC dentro de sistemas autodistribuídos, utilizando *proxies* gerados automaticamente para permitir a autodistribuição e manter a consistência do estado dos componentes. A inovação está no uso de uma IDL personalizada para capturar informações específicas do domínio de sistemas autodistribuídos, necessárias para automatizar esse processo.

3.3 Sistemas Autodistribuídos

No domínio de sistemas autodistribuídos é necessário o destaque para alguns trabalhos:

Rodrigues-Filho e Porter (2020) explicam que sistemas modernos precisam cada vez mais de capacidade de adaptação a ambientes mutáveis, para isso, tecnologias como o Reflective Middleware e outros metamodelos, permitem uma boa flexibilidade, porém não se preocupam com o manuseio do estado de maneira automatizada. Com isso, demonstram uma possível solução utilizando um modelo baseado em componentes que permite realocar, replicar e dividir componentes que necessitam do manuseio do estado. Isso é feito através da substituição do componente real por um *proxy*, que encaminha as chamadas feitas ao original para um componente remoto, através de RPC *(Remote Procedure Call)*. Ao utilizar esse conceito, podem ser reutilizados métodos e balanceado o uso de componentes frequentemente importantes, permitindo um desempenho melhor do sistema de modo geral.

Rodrigues-Filho e Porter (2022) comentam sobre alguns dos desafios no ramo de sistemas distribuídos: o projeto, a implantação e a manutenção, o que pode ser perceptível na infraestrutura do projeto. Com isso, surge a oportunidade do desenvolvimento de uma solução para a adversidade citada anteriormente, o Hatch, que utiliza a ideia de abstrair um sistema em subcomponentes, os quais podem distribuir os módulos do sistema em diferentes nós da estrutura e distribuí-los em diferentes máquinas durante o tempo de execução, sem pausas. Utilizando técnicas como `relocação` (*alternate*)[2], `replicação` (*Propagate*)[3] e `fragmentação` (*Sharding*) [4]. Por fim, é comentado a respeito de algumas limitações, onde, por exemplo, neste artigo foi utilizada somente a métrica de tempo de resposta; outras métricas podem apresentar resultados diferentes. Além disso, neste artigo foi utilizada somente uma instância do Hatch; em uma operação real, provavelmente seriam utilizadas mais instâncias dividindo os mesmos recursos.

Rodrigues-Filho et al. (2022) mencionam a respeito da dificuldade para a manutenção dos sistemas distribuídos modernos decorrente da constante volatilidade que cada sistema enfrenta, visto que pode ocorrer em cada uma das camadas (infraestrutura, plataforma e aplicação/serviços). Isso faz com que, caso seja feita a escolha de uma tecnologia inadequada para determinado fim, haja perda de desempenho. Com isso, desenvolveu-se a arquitetura de programação em pequenos métodos ou funções, permitindo a alteração e mudança de determinadas partes do código sem afetar o restante do sistema. O seguinte artigo explora a noção de sistemas autodistribuídos com gerenciamento de estado, uti-

[2] *Alternate:* Substitui um componente real por um *proxy*, que encaminhará a um nó remoto, alternando o nodo de destino.

[3] *Propagate:* Realocação, porém para mais de um nó remoto

[4] *Sharding:* Divide o componente original por tipos de dados e o aloca em diferentes nós remotos.

lizando a escala horizontal na nuvem caso haja a necessidade para tal. É utilizado um algoritmo que compreende o padrão de solicitações para que o sistema se adapte durante sua execução, alterando seus módulos internos. Em suma, relata-se a necessidade da criação de *proxies* de distribuição manualmente, acarretando um esforço maior do que o desejado pelos autores. Visando solucionar isso, foi idealizado um repositório para o armazenamento de *proxies* de componentes de distribuição, para facilitar a produção de aplicações futuras.

Nos itens acima, são idealizadas diferentes propostas para diferentes problemas, contudo, todos necessitam do desenvolvimento manual dos *proxies* para permitir a distribuição do sistema. Visando solucionar isso, esse trabalho propõe a utilização de uma DSL para automatizar esse processo.

No âmbito de propostas para a automatização da geração de *proxies* em sistemas autodistribuídos, o trabalho de Kina et al. (2023) ganha destaque, nele foi utilizada uma Linguagem Específica de Domínio baseada em YAML. Da mesma maneira que o presente trabalho, o objetivo é simplificar a implementação de sistemas autodistribuídos, reduzindo a necessidade de intervenção manual. Entretanto, a abordagem foca no padrão de interação *sharding* e nos métodos *cross-sharding* (métodos que interagem com mais de um nó em cada requisição), sem definir padrões de interação. Em contrapartida, o presente trabalho teve como foco a geração de *proxy* de uma forma completa e genérica.

Existem outras formas de pensar em sistemas autodistribuídos, como Woodfin (2002) no trabalho Self-Distributing Computation (SDC), onde visa otimizar a distribuição de tarefas computacionais em sistemas distribuídos, dividindo a responsabilidade entre o programador e o sistema. A implementação em Java, chamada Mimoid, mostrou como a interação entre o programador e o sistema pode aumentar a flexibilidade e eficiência na distribuição e consolidação de tarefas, minimizando a complexidade de programação. Com a distribuição de responsabilidades, o modelo SDC oferece um maior nível de adaptabilidade.

Após o estudo dos trabalhos relacionados no campo de sistemas autodistribuídos, verifica-se que o estado da arte dessas tecnologias apresenta lacunas que tornam a geração de *proxies* uma tarefa manual e custosa. Esses trabalhos mostraram que, apesar dos avanços em arquiteturas distribuídas e linguagens específicas, a criação de *proxies* para sistemas autodistribuídos ainda demanda um esforço significativo por parte dos desenvolvedores. Dessa forma, como indicado na seção 1.1, identificou-se a necessidade de automatizar a geração de *proxies* utilizando uma linguagem específica de domínio (DSL), que permita descrever a interface dos componentes distribuídos e possibilitar o funcionamento adequado da geração automatizada dos *proxies*. Este trabalho, portanto, busca preencher essa lacuna ao propor uma DSL que simplifica e automatiza o processo, aumentando a eficiência e auxiliando na redução da complexidade no desenvolvimento de sistemas autodistribuídos.

4 ARQUITETURA DA SOLUÇÃO

Neste capítulo, será apresentada a arquitetura da solução desenvolvida para a geração automatizada de *proxies* para sistemas autodistribuídos como descrito na seção 1.1, com foco na interação entre dois artefatos principais: a linguagem de definição de interface (IDL), especificada no arquivo $idl.json$, e o programa gerador de *proxy* *ProxyGenerator*.

Como indicado na seção 2.4, o *proxy* é o responsável por substituir o componente durante o processo de autodistribuição por um componente virtual, o qual apenas encaminha as requisições à um nó remoto, semelhante ao conceito de *stubs* em tecnologias de RPC, como o Apache Thrift (discutido na Seção 2.2), o qual se propõe a facilitar criação de serviços de comunicação entre componentes distribuídos, permitindo a substituição automatizada.

Durante o processo de geração automatizada de *proxies*, são utilizados como base tanto o conteúdo contido na interface fornecida pelo usuário quanto as informações adicionais definidas pela linguagem de definição de interface (IDL) que não estão presentes na interface original, tais como a possibilidade do método impactar algum atributo, ou seja, se ele altera o estado de um atributo da classe.

Por exemplo, em uma interface de lista, o atributo *content* pode armazenar todos os elementos da lista. Operações como adicionar ou remover elementos modificam o estado interno da lista, o que caracteriza um impacto sobre o atributo. Essa informação é essencial para determinar o tipo de *proxy* gerado, já que diferentes padrões de interação (como *sharding* ou *propagate*, abordadas na seção 2.4) podem ser aplicadas com base no impacto causado pelos métodos no estado do componente.

A arquitetura proposta utiliza três padrões de interação para definir como os métodos se comportam em um ambiente distribuído: *sharding*, *propagate* e *alternate*. No padrão de *sharding*, os dados ou operações são divididos e distribuídos entre diferentes nós remotos, otimizando a carga de trabalho ao fragmentar as responsabilidades entre múltiplos locais. No padrão de *propagate*, uma mesma informação é replicada e enviada para todos os nós relevantes, garantindo que todos compartilhem o mesmo estado atualizado. Já o padrão *alternate* é utilizado em casos onde os métodos não impactam diretamente o estado alternam entre diferentes nós a cada chamada, distribuindo as requisições de forma balanceada entre os nós disponíveis, de forma semelhante à um balanceador de cargas. Além disso, durante o desenvolvimento do presente trabalho, foi desenvolvido um novo padrão de interação, mesclando a abordagem do *propagate* para métodos que impactam no estado do componente e *alternate* para métodos que não impactam. Esses padrões são

fundamentais para assegurar que cada *proxy* gerado se comporte corretamente, atendendo às necessidades específicas de cada contexto e garantindo a consistência e a eficiência do sistema autodistribuído.

Primeiramente, será fornecida uma visão geral da arquitetura e da interação entre seus componentes. Posteriormente, cada componente será descrito em detalhes, começando pela linguagem de definição de interface, seguida pela explicação do funcionamento do gerador de *proxy*.

4.1 Visão geral da arquitetura

A solução proposta é composta de um programa em Python denominado *ProxyGenerator*, que tem como entrada o arquivo da interface a partir da qual o usuário deseja automatizar e um arquivo de utilizando arquivo de IDL no formato de JavaScript Object Notation (JSON) contendo as informações adicionais necessárias para a geração do *proxy*, chamado *idl.json*.

Fonte: O autor, 2024.

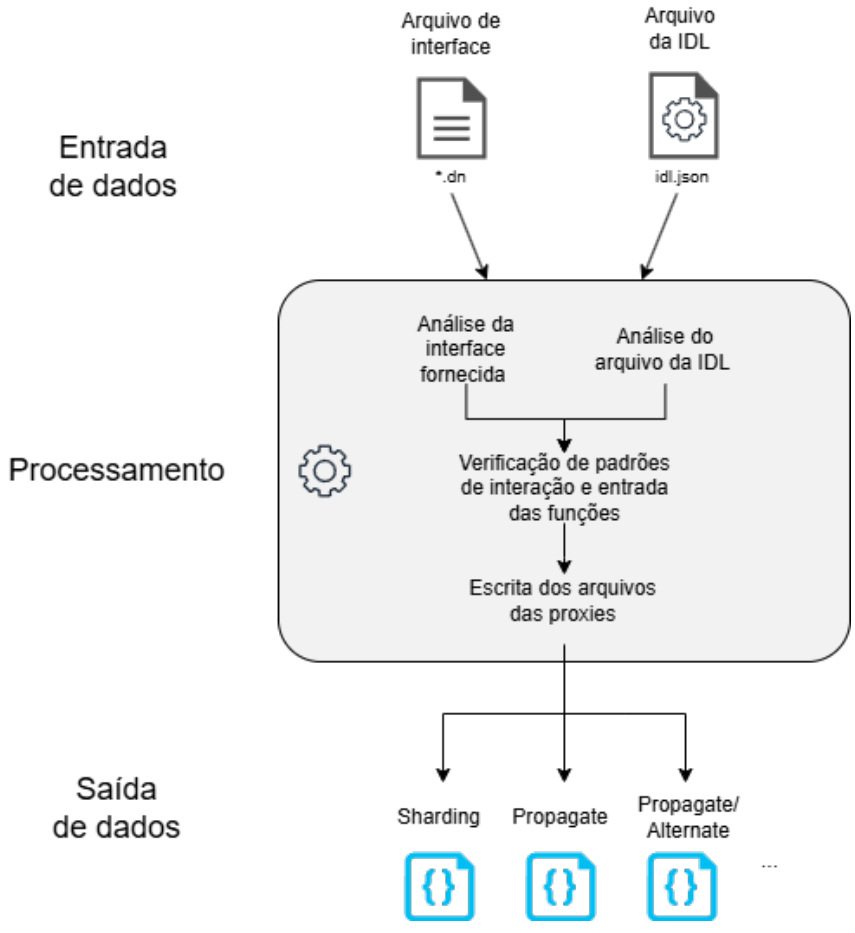

Figura 4 – Fluxograma de geração de *proxy*.

De forma geral, como ilustrado na Figura 4, para utilizar a geração automatizada de *proxy* é esperada a utilização de um arquivo de interface utilizando a linguagem de programação Dana como entrada para o programa, em conjunto com um arquivo da IDL contendo as informações adicionais necessárias para a execução do código, em formato JSON.

O programa inicia examinando os dados de cada método individualmente, tais como o tipo de dado de retorno, tipo de dados de entrada, número de parâmetros de entrada, entre outras. Essas informações são cruciais para a execução pois no *proxy* haverá a implementação dos métodos de diferentes formas. Em seguida, o *ProxyGenerator* examina o arquivo da IDL buscando identificar se o método em questão tem impacto nos atributos da classe, caso tenha, em qual atributo impacta e qual o tipo dele. Com essas informações sintetizadas, inicia a verificação de padrões de interação e a escrita dos arquivos das *proxies* de acordo com a necessidade identificada.

4.2 Definição da Linguagem Específica de Domínio

Como indicado na seção 4.1, durante o processo de geração de *proxy*, não é possível realizá-la sem o auxílio de informações adicionais que não estão presentes na interface, principalmente considerando que o método pode impactar em algum atributo da classe, o que afeta na decisão dos padrões de interação adotados para implementar o método. A fim de expressar essas informações adicionais necessárias, foi desenvolvida uma linguagem específica de domínio (DSL), especificamente para descrição de interfaces, com isso optou-se pela definição da gramática de uma linguagem de definição de interface utilizando a sintaxe com notação JavaScript Object Notation (JSON), a escolha do JSON como notação decorre de sua simplicidade e legibilidade.

A linguagem inicia-se por três principais atributos: $interface\_path$, $output\_path$ e *methods*. Os dois primeiros atributos contém o caminho (*path* – estrutura de pastas no sistema de arquvios) para o arquivo de interface da qual os *proxies* serão geradas. Já o segundo atributo, indica o caminho (*path)* para a pasta onde serão gerados os *proxies*.

O atributo *methods* é um objeto que contém um conjunto de objetos nomeados de acordo com os métodos presentes no arquivo de interface. Para cada método, o usuário deve informar se ele impacta algum atributo da classe através do atributo booleano *impact*. Caso impacte, é necessário especificar o atributo afetado e o seu tipo. A estrutura dessa linguagem específica de domínio é descrita a seguir utilizando a notação EBNF (Extended Backus-Naur Form), como exemplificado na Figura 5. A notação EBNF foi escolhida por permitir uma maior precisão e entendimento a respeito da gramática de linguagens específicas de domínio, conforme indicado por Feynman (2016).

A Figura 6 exemplifica a descrição da interface para a automatização de uma

Fonte: O autor, 2024.

```
IDL ::= '{' InterfacePath ',' OutputPath ',' Methods '}'

InterfacePath ::= '"interface_path"' ':' STRING

OutputPath ::= '"output_path"' ':' STRING

Methods ::= '"methods"' ':' '{' MethodList '}'

MethodList ::= (MethodEntry (',' MethodEntry)*)?

MethodEntry ::= MethodName ':' MethodData

MethodName ::= STRING

MethodData ::= '{' '"impact"' ':' BOOLEAN (',' Attributes)? (',' AttributesType)? '}'

Attributes ::= '"attrs"' ':' StringArray

AttributesType ::= '"attrs_type"' ':' StringArray

StringArray ::= '[' StringList ']'

StringList ::= (STRING (',' STRING)*)?

STRING ::= '"' CHARACTER* '"'

BOOLEAN ::= 'true' | 'false'

CHARACTER ::= qualquer caractere exceto aspas duplas (")
```

Figura 5 – Definição da linguagem utilizando notação EBNF.

interface *List*, a qual possui seis métodos, onde três deles impactam o atributo *content*, do tipo *collection*. Neste exemplo o método *add*, altera o estado do atributo ao inserir dados em uma lista, o método *remove* também impacta pois retira dados do atributo, já o método *update*, altera o conteúdo ao atualizar as informações presentes nele. Já os métodos que não impactam nos atributos, o *getFirst* retorna o primeiro elemento da lista não alterando o atributo e consequentemente não impactando na Lista, o *getLength* retorna o tamanho da lista e o *getContents* retorna todo o seu conteúdo.

4.3 *ProxyGenerator*: Gerador automatizado de *proxy*

Nesta seção será descrito o componente principal da geração automatizada de *proxy*. Para este trabalho, optou-se pela implementação utilizando a linguagem de programação Python devido à sua sintaxe concisa, facilitando o desenvolvimento e a manutenção do código, além de sua extensa documentação, facilitando na resolução de problemas. Nota-se que, a geração de *proxy* pode ser implementada com quaisquer outra linguagem, visto que não foi utilizada nenhuma API/biblioteca exclusiva da linguagem Python durante a sua construção.

A implementação foi segmentada em seis funções principais a fim de facilitar a legibilidade, compreensão, manutenção e extensão do código, sendo elas: *cleanLine*,

Fonte: O autor, 2024.

```
{
    "interface_path": "./interface/List.dn",
    "output_path": "./result/",
    "methods": {
        "add": {
            "impact": true,
            "attrs": ["content"],
            "attrs_type": ["collection"]
        },
        "remove": {
            "impact": true,
            "attrs": ["content"],
            "attrs_type": ["collection"]
        },
        "getFirst": {
            "impact": false
        },
        "getLength": {
            "impact": false
        },
        "getContents": {
            "impact": false
        },
        "update": {
            "impact": true,
            "attrs": ["content"],
            "attrs_type": ["collection"]
        }
    }
}
```

Figura 6 – Exemplo de arquivo IDL para uma interface de lista.

$readInterfaceFile$, $writeHeader$, $writeFooter$, $writeFunction$, $generateProxyFiles$. A interação entre as funções está ilustrada na Figura 7. O gerador de *proxy* inicia na função $readInterfaceFile$ que recebe como parâmetro o arquivo de interface, o qual lê o arquivo linha a linha, chamando a função $cleanLine$ para remover os comentários e a identação dela a fim de permitir uma extração precisa dos dados presentes na interface, em seguida, é executada a função principal, denominada $generateProxyFiles$, que recebe como parâmetro um objeto contendo as informações sintetizadas pela função $readInterfaceFile$, a partir disso, itera sobre os modos de interação, criando um arquivo de *proxy* para cada modo de interação e também chama as funções de escrita $writeheader$, $writeFunction$ e $writeFooter$.

Vale ressaltar que o conteúdo do código utilizado para a implementação dos métodos provém de trechos de códigos já existentes (*snippets* de código), com isso, não foi utilizado

nenhum modelo de linguagem grande (LLM) ou outro tipo de algoritmo de inteligência artificial.

Fonte: O autor, 2024.

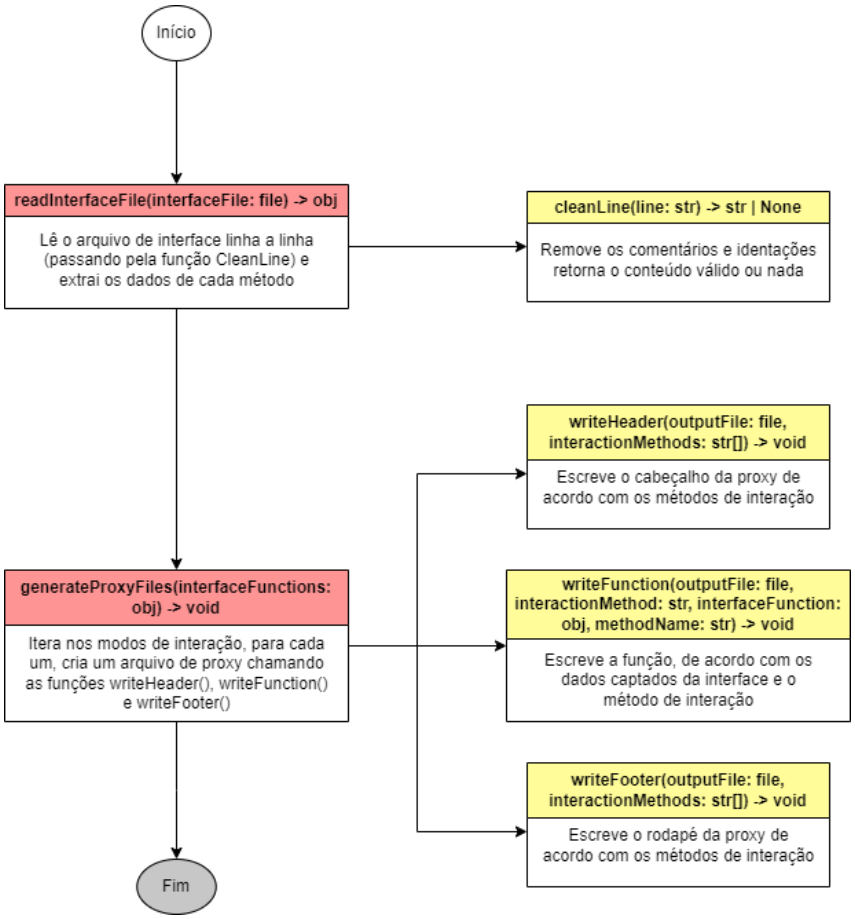

Figura 7 – Interações entre funções.

Em seguida será apresentado a implementação de cada função:

- $cleanLine(line : str) -> str|None$: tem como parâmetro a linha não processada em formato *string*[1] e retorna uma *string* ou um valor nulo (caso a linha não tenha nenhum conteúdo válido). Nessa função foram utilizadas expressões regulares (Regex) para identificar e desconsiderar comentários no arquivo de interface, além do método *strip* que remove os espaços em branco.
- $readInterfaceFile(interfaceFile : file) -> void$: recebe como parâmetro o arquivo de interface, acessado através do $interface\_path$ presente no arquivo de IDL ($idl.json$). Itera a interface linha a linha utilizando a função *cleanLine* para desconsiderar comentários e identações. retorna um dicionário contendo os nomes

[1] Tipo de dado textual.

dos métodos presentes no arquivo de interface. Em cada um deles: o tipo do seu retorno ($returnType$), o nome da interface que pertence ($interfaceName$), a lista de parâmetros ($parameterList$) e o número de parâmetros (que depende se algum dos parâmetros é opcional - $numParam$).

- $writeHeader(outputFile : file, interactionMethods : str[]) -> void$: tem dois parâmetros de entrada: o arquivo de saída (output_file) do tipo TextIOWrapper e a lista de métodos de interação presentes naquele arquivo, e escreve o conteúdo de acordo com a necessidade de cada método de interação.

- $writeFooter(outputFile : file, interactionMethods : str[]) -> void$: segue a mesma lógica da função $writeHeader$, com os mesmos parâmetros.

- $writeFunction(outputFile : file, interactionMethod : str, interfaceFunctionData : obj, methodName : str) -> void$: possui quatro parâmetros de entrada: o arquivo de saída, o método de interação utilizado, um dicionário do conteúdo daquele método na interface ($interfaceFunctionData$) e o nome do método ($methodName$). A partir dos parâmetros citados, escreve o conteúdo do método analisando se o método impacta ou não nos atributos da classe (atributo *impact* do arquivo de IDL).

- $generateProxyFiles(interfaceFunctions : obj) -> void$: recebe como parâmetro um dicionário contendo as informações sintetizadas da interface através do retorno da função $readInterfaceFile$, itera em uma lista de tipos de interação ($sharding$, $propagate$, $mixed\_sharding$, $mixed\_propagate$), para cada tipo, cria um arquivo na pasta informada pelo usuário e chama as funções $writeHeader$, $writeFunction$ e $writeFooter$ para cada arquivo individualmente.

4.4 Estrutura de um *proxy*

A fim de facilitar a compreensão de como é estruturado um *proxy*, serão apresentadas as Figuras 8, 9 e 10 para demonstrar de forma exemplificada como é a implementação de um *proxy* para a interface *List*.

A Figura 8 ilustra o cabeçalho de um *proxy*, ele é composto por informações a respeito do componente e também a respeito da tecnologia RPC (da parte de comunicação com as réplicas remotas). Dentre as informações do componente, estão as declarações das operações disponíveis (quais métodos ele possui, seguido pelas informações do que o componente provê (qual interface ele está implementando) e o que ele requer para sua plena execução (quais bibliotecas e ferramentas externas, ex: *parser* e *socket* de rede). A respeito do RPC é no *proxy* onde serão declarados métodos para definir como as requisições serão enviadas e como as respostas recebidas serão tratadas, dentre outras informações.

Fonte: O autor, 2024.

```
/* Available list operations */
const char ADD[]          = "add"
const char GET_LENGTH[]   = "getLength"
const char GET_CONTENTS[] = "getContents"
const char CLEAR_LIST[]   = "clearList"

/* IPs */
const char LOCALHOST[] = "localhost"

component provides List:heap(Destructor, AdaptEvents) requires data.json.JSONEncoder parser,
    net.TCPSocket, data.StringUtil strUtil, io.Output out, data.IntUtil iu, hash.Multiplicative hash
{
    IPAddr remoteDistsIps[] = null
    IPAddr remoteListsIps[] = null
    int toggleContents = 1

    void setupRemoteDistsIPs() {
        if (remoteDistsIps == null) {
            remoteDistsIps = new IPAddr[2]
            remoteDistsIps[0] = new IPAddr()
            remoteDistsIps[0].ip = new char[](LOCALHOST)
            remoteDistsIps[0].port = 8081
            remoteDistsIps[1] = new IPAddr()
            remoteDistsIps[1].ip = new char[](LOCALHOST)
            remoteDistsIps[1].port = 8082
        }
    }

    void setupRemoteListsIPs() {
        if (remoteListsIps == null) {
            remoteListsIps = new IPAddr[2]
            remoteListsIps[0] = new IPAddr()
            remoteListsIps[0].ip = new char[](LOCALHOST)
            remoteListsIps[0].port = 2010
            remoteListsIps[1] = new IPAddr()
            remoteListsIps[1].ip = new char[](LOCALHOST)
            remoteListsIps[1].port = 2011
        }
    }
```

Figura 8 – Cabeçalho do *proxy*.

A Figura 9 ilustra o corpo do *proxy*, o foco desse trabalho, onde estão presentes as implementações de cada método de acordo com a composição utilizada. No caso da ilustração, os métodos *add* e *getContents()* estão implementando a composição *propagate*, a qual encaminha a mesma informação para todas as réplicas remotas da mesma maneira.

Já a Figura 10 demonstra o rodapé do *proxy*, nele estão presentes alguns métodos comuns para todas as composições, que são utilizados no âmbito da adaptabilidade a eventos, por este motivo, foi uma seção pouco explorada neste estudo.

4.5 Expansão de métodos de interação

Neste trabalho, utilizou-se como exemplo na criação e definição da linguagem de definição de interface uma interface Lista, como indicado na Figura 6, entretanto, trabalhos futuros podem estender a linguagem proposta para outras interfaces, as quais podem possuir outros métodos de interação entre o *proxy* e os componentes remotos. Nesse sentido, de forma que a linguagem proposta continue vigente, conseguindo se adaptar e evoluir de acordo com as demandas.

Fonte: O autor, 2024.

```
void List:add (store Data item) {
    Request request = new Request()
    request.functionName = "add"
    request.numParams = 1

    char requestStr[] = parser.jsonFromData(request, null)
    char param[] = parser.jsonFromData(item, null)
    char content2[] = new char[](requestStr, "!", param, "\r\r\r\r")

    makeGroupRequest(content2)
}


Data[] List:getContents (opt Type t) {
    Request request = new Request()
    request.functionName = "getContents"
    request.numParams = 0

    char requestStr[] = parser.jsonFromData(request, null)
    char content2[] = new char[](requestStr, "!", " ", "\r\r\r\r")
    Response response = makeRequest(content2)
    Int nums[] = parser.jsonToArray(response.value, typeof(Int[]), null)
    return nums
}
```

Figura 9 – Corpo do *proxy* (implementações dos métodos).

A solução proposta permite que o usuário expanda os métodos de interação suportados, basta inserir na variável *interactionList* o nome do novo método, na função *writeFunction* inserir as particularidades de cada tipo de método (se impacta ou não em algum atributo da classe). Se necessário inserir algo no cabeçalho da *proxy*, inserir na função *writeHeader*, da mesma forma para o rodapé na função *writeFooter*. Como apresentado na Figura 11.

Fonte: O autor, 2024.

```
void AdaptEvents:inactive() {
    if (content != null) {
        content = getContents()
        char msg[] = new char[]("clearList!\r\r\r\r")

        setupRemoteDistsIPs()
        for (int i = 0; i < remoteDistsIps.arrayLength; i++) {
            makeRequestSharding(remoteDistsIps[i], msg, true)
        }
    }
}

void AdaptEvents:active() {
    if (content != null) {
        setupRemoteDistsIPs()
        ShardState shardState[] = new ShardState[remoteDistsIps.arrayLength]
        Thread thread[] = new Thread[remoteDistsIps.arrayLength]
        for (int i = 0; i < content.arrayLength; i++) {
            Int num = content[i]
            int remoteIdx = hash.h(num.i, remoteDistsIps.arrayLength)
            if (shardState[remoteIdx] == null) {
                shardState[remoteIdx] = new ShardState()
            }
            shardState[remoteIdx].state = new Int[](shardState[remoteIdx].state, num)
        }
        for (int i = 0; i < remoteDistsIps.arrayLength; i++) {
            char state[] = parser.jsonFromArray(shardState[i].state, null)
            char msg[] = new char[]("../distributor/RemoteList.o!", state, "\r\r\r\r")
            thread[i] = asynch::makeRequestSharding(remoteDistsIps[i], msg, true)
        }
        for (int i = 0; i < remoteDistsIps.arrayLength; i++) {
            thread[i].join()
        }

    }
}
```

Figura 10 – Rodapé do *proxy*.

Fonte: O autor, 2024.

```
SHARDING = "sharding"
PROPAGATE = "propagate"
ALTERNATE = "alternate"
MIXED_SHARDING = "mixed_sharding"
MIXED_PROPAGATE = "mixed_propagate"
NOVO_METODO = "novo_metodo"
interactionList = [SHARDING, PROPAGATE, MIXED_SHARDING,
MIXED_PROPAGATE, NOVO_METODO]
```

Figura 11 – Adicionando novo método de interação (variável *interactionList*).

5 RESULTADOS E DISCUSSÕES

Neste capítulo, serão apresentados os resultados dos experimentos em conjunto com uma discussão a respeito dos principais resultados. Este capítulo está dividido entre as seguintes subseções: cenários, descrição e apresentação dos experimentos, resultados e discussões a respeito das conclusões obtidas.

O conjunto de experimentos, explorados neste capítulo, visa mostrar que quanto maior o número de *proxies*, maior a flexibilidade para compor o sistema, permitindo uma maior adaptabilidade e uma eficiência do sistema em geral. Dessa forma, é possível compreender a utilidade do *ProxyGenerator* como uma importante ferramenta para o desenvolvimento de aplicações utilizando o *framework* SDS.

5.1 Cenários

Na realização dos experimentos, utilizou-se uma versão customizada[1] do código de Araujo et al. (2021) (como mencionado no Cap. 1) que consiste em um servidor web que implementa o protocolo HTTP/1.0. O servidor web possui em sua composição a interface *List*, que é implementada por um componente que executa localmente e é responsável por manter na memória números inteiros passados por requisições HTTP.

O servidor web implementa um serviço com dois métodos: o método *ADD* e o método *GET_CONTENTS*. O método *ADD* é responsável por adicionar números inteiros à lista, enquanto a função *GET_CONTENTS* retorna para o cliente o conteúdo da lista. Note que a operação *ADD* é uma operação de escrita do estado da lista, ou seja, ela altera o seu valor (adicionando novos números). Enquanto a operação *GET_CONTENTS* é uma operação de leitura, o que significa que não altera os elementos da lista.

Devido ao código atual ter um baixo custo computacional originalmente e apenas manipular itens em uma lista, para melhor ilustração de um cenário de uso real, em que o sistema executa tarefas com alto custo de processamento, optou-se por utilizar um algoritmo para checar se cada item multiplicado por um fator é um número primo. Este algoritmo aumenta, artificialmente, a demanda por CPU, por requisição ao serviço.

O algoritmo segue a Equação 5.1, em que o π é uma função que calcula números primos, e recebe como parâmetros a variável i que representa o número de itens que consta na lista, e o $FATOR$ que é um fator multiplicativo que à medida que seu valor aumenta, maior é a complexidade de tempo da função π, forçando assim, um maior uso de CPU.

[1] Repositório do GitHub: https://github.com/k1vz/agp_sds

$$\pi(i * FATOR) \tag{5.1}$$

Nos experimentos em que o maior número de requisições é de escrita, esse código foi adicionado ao método *ADD*, já quando a maioria das requisições são de leitura, o código foi adicionado ao método *GET_CONTENTS*.

Foram realizados um total de três experimentos. Os dois primeiros experimentos, denominados de experimentos de divergência de desempenho, os quais foram projetados para demonstrar que as composições do servidor web possuem desempenhos distintos quando submetidos a ambientes operacionais distintos. Enquanto que o terceiro experimento, nomeado experimento de carga, mostra o servidor web operando à medida que aumenta-se o número de itens em sua lista.

Nos experimentos de divergência de desempenho foi alterada a característica da carga de trabalho processada pelo servidor e o fator multiplicativo do algoritmo de números primos, utilizado para intensificar o uso de CPU. No primeiro experimento, a carga de trabalho A constitui-se majoritariamente com requisições do tipo **leitura**, isto é, a maior parte das requisições não altera o estado da lista, e o fator multiplicativo é definido como 1. Enquanto que no segundo experimento, a carga de trabalho B constitui-se majoritariamente por requisições do tipo **escrita**, isto é, requisições que alteram o estado da lista e o fator multiplicativo foi alterado para 1000.

No experimento de carga, utiliza-se uma carga de 10 requisições sequenciais de leitura, enquanto é acrescentado de forma gradativa o número de itens na lista. Este experimento, visa demostrar como cada composição do servidor web se comporta à medida que aumenta-se o número de itens na lista, e consequentemente, a demanda de CPU.

Os experimentos foram realizados em duas máquinas com Ubuntu 22.04, conectadas a uma rede local Wi-Fi 2.4 GHz. Uma máquina hospeda o servidor web local, e à medida que o servidor web se distribui, ele se distribui para a segunda máquina, simulando de forma mais fidedigna as interações entre os *proxies* e os componentes distribuídos.

5.2 Resultados

Os resultados dos experimentos de divergência de desempenho são apresentados nas Figuras 12 (carga de trabalho A) e 13 (carga de trabalho B). Já os resultados do experimento de carga é mostrado na Figura 14. Nos experimentos de divergência de desempenho, é demonstrado que a composição do servidor web que obtém o melhor desempenho varia de acordo com as características da carga de trabalho, sendo utilizada a expressão ciclos de observação, variando de 1 a 4, indicando os conjuntos de requisições: 10, 30, 50 e 100, respectivamente. No experimento de carga, mostra-se que os *proxies* gerados

são todos funcionais, tendo uma composição que se destaca à medida que o servidor é estressado.

No primeiro experimento (Figura 12), o servidor web é submetido à carga de trabalho A (majoritariamente de leitura e fator multiplicativo 1 – ou seja, demanda pouca CPU por requisição) e mostra que a composição local tem o melhor desempenho. Isso é devido ao fato de que as operações não demandam muito tempo de CPU e são atendidas rapidamente pelo servidor local. Como não há um aumento da demanda de CPU, verifica-se no experimento, que não há benefício no encaminhamento da requisição para que seja processada de forma distribuída, e portanto, as composições distribuídas (que usam *proxy*) possuem a desvantagem de ter somado ao seu tempo de resposta a latência extra da rede, resultado do encaminhado da requisição da máquina local (onde o servidor web está executando) para a segunda máquina (onde os componentes distribuídos estão executando).

Fonte: O autor, 2024.

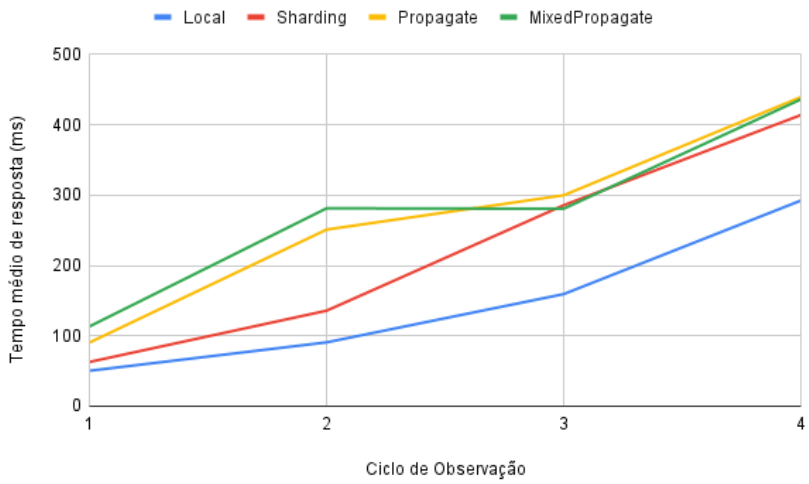

Figura 12 – Experimento de Divergência de Desempenho - Carga A

Em contra partida, como mostra a Figura 13, a composição do servidor web que possui melhor desempenho é a composição de *sharding*. Nesse caso, isso acontece pois o fator multiplicativo da carga de trabalho B é definida para 1000, o que intensifica o uso de CPU por requisição. Assim, a composição de *sharding* divide o tamanho da lista em dois, e o processamento é feito em somente metade da lista, ou seja, mesmo adicionando o tempo de latência para encaminhar a requisição, o tempo de resposta total acaba sendo menor, porque o processamento é feito em uma lista com metade dos itens da lista local.

Os experimentos de divergência de desempenho demonstram, com sucesso, que dependendo do ambiente operacional em que o servidor web é executado, uma composição distinta terá uma maior vantagem em relação a outra. No primeiro caso, a composição local

Fonte: O autor, 2024.

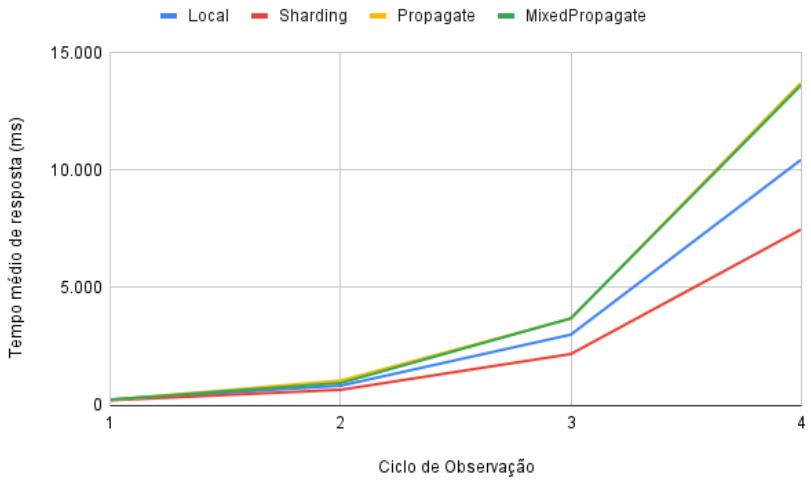

Figura 13 – Experimento de Divergência de Desempenho - Carga *B*

tem vantagem em relação as demais composições. No segundo caso, em contra partida, a composição *sharding* demonstra uma vantagem em relação as demais composições.

Por fim, apresenta-se o resultado do experimento de carga. Este experimento visa mostrar o desempenho de cada uma das composições à medida que o número de itens na lista aumenta. Esse experimento mostra que a composição *sharding* tem um melhor desempenho que as demais composições à medida que o número de itens na lista aumenta. Essa correlação é valida pois o aumento de itens na lista resulta em um maior tempo de processamento. Levando em conta que o *sharding* divide a lista em dois fragmentos, o número de itens por fragmentos é sempre a metade do número de itens total, o que resulta em um menor tempo de processamento e um menor tempo de resposta.

É importante destacar que as configurações de componentes das máquinas utilizadas nos experimentos possuem influência direta nos valores observados de tempo de resposta, além disso, variáveis como o poder de processamento, a latência da rede e a demanda do sistema operacional podem impactar os resultados obtidos. o entanto, embora os valores absolutos de tempo de resposta possam variar em cenários com configurações distintas, a tendência da curva dos gráficos observada nos experimentos se manteria. Isso ocorre porque as composições analisadas apresentam comportamentos consistentes em relação às características da carga de trabalho e ao uso dos recursos computacionais, independentemente da infraestrutura utilizada. Assim, a influência das configurações impacta apenas os valores específicos dos tempos medidos, sem alterar as conclusões gerais sobre o comportamento e a eficiência das diferentes composições geradas pelo *ProxyGenerator*.

Conclui-se, dessa forma, que os *proxies* gerados pelo *ProxyGenerator* são funcionais e podem auxiliar sistemas autodistribuídos a ter uma maior flexibilidade em sua composição

e com isso ter um melhor desempenho dependendo do ambiente operacional em que o sistema se encontra.

Fonte: O autor, 2024.

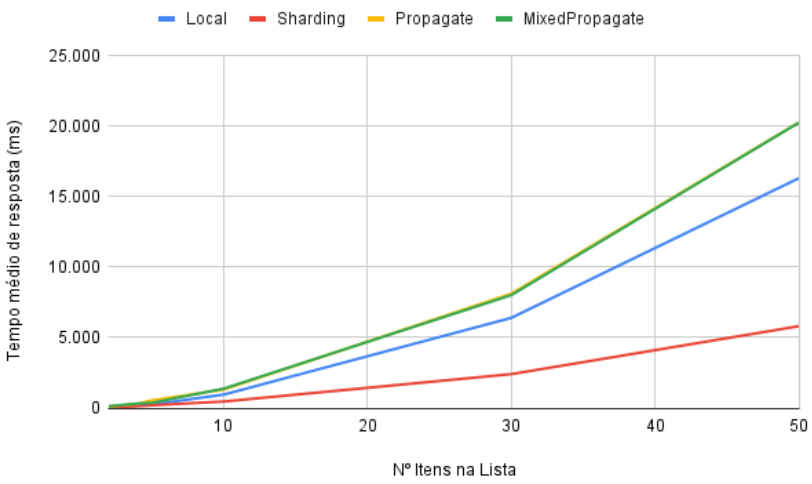

Figura 14 – Experimento de Carga

.

6 CONCLUSÃO

Devido à complexidade na criação e gestão de sistemas contemporâneos, foi idealizado o conceito de sistemas autodistribuídos, entretanto o processo de distribuição de componentes de forma autonômica necessita originalmente da geração manual de *proxy*, o que aumenta a complexidade para o desenvolvedor, visto que é necessário determinar qual algoritmo será utilizado para manter a consistência do estado do componente durante a sua distribuição, além da definição dos detalhes da comunicação entre o *proxy* e o componente original.

Visando reduzir a dificuldade do uso de sistemas autodistribuídos por desenvolvedores, esse trabalho propôs uma maneira de automatizar a geração de *proxy*. Para isso é necessário acesso às informações que não estão presentes na interface dos componentes. Com isso, optou-se pela utilização de uma Linguagem de Definição de Interface (IDL), abordada em detalhes no Capítulo 4.

Neste trabalho, foi utilizado como exemplo uma versão customizada do código de Araujo et al. (2021) que provê um servidor web que implementa a interface List. A partir disso, foi utilizada a linguagem Python para desenvolver um artefato denominado *ProxyGenerator* que junto da IDL proposta, para realizar a geração de *proxies* para diferentes composições: fragmentada *(sharding)*, replicada *(propagate)* ou replicada mista *(mixed_propagate)*.

Com base nos experimentos realizados no Capítulo 5, utilizando duas máquinas, conectadas em uma rede local via Wi-Fi, foi possível visualizar a funcionalidade e a eficácia dos *proxies* gerados pelo *ProxyGenerator*. Os resultados demonstraram que as diferentes composições oferecem vantagens específicas dependendo do cenário operacional. Por exemplo, a composição *sharding* destacou-se em cenários de alta demanda computacional, reduzindo de forma significativa o tempo de processamento ao dividir a carga entre os nós remotos. Por outro lado, a composição local mostrou superioridade em ambientes com baixo processamento, onde a latência adicional de rede nas composições distribuídas não era compensada pelo aumento no desempenho. Esses resultados evidenciam a importância de utilizar *proxies* que implementem múltiplas composições para permitir maior adaptabilidade e otimização do desempenho do sistema em tempo de execução.

Como trabalhos futuros, além da ampliação da gramática da IDL para suportar novas interfaces e estruturas de dados, é possível explorar a aplicação de algoritmos de aprendizado de máquina para identificar, em tempo real, o *proxy* mais adequado para diferentes situações operacionais. Essa abordagem pode otimizar ainda mais o desempenho do sistema autodistribuído, adaptando-se dinamicamente a variáveis como carga de trabalho

e recursos disponíveis. Outra vertente é investigar a escalabilidade do *ProxyGenerator*, avaliando sua eficiência na geração de *proxies* em ambientes de grande escala e com alta complexidade. Adicionalmente, recomenda-se a análise detalhada do custo de adaptação do sistema durante o processo de redistribuição de componentes, considerando os impactos no desempenho e no consumo de recursos computacionais. Essas extensões podem ampliar significativamente as contribuições deste trabalho, consolidando-o como base para o avanço de sistemas autodistribuídos.

Referências

ARAUJO, B. B. et al. Explorando o desempenho de sistemas auto-distribuídos na borda e nuvem. 2021.

BLAIR, G. Complex distributed systems: The need for fresh perspectives. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.: s.n.], 2018. p. 1410–1421.

CHADOKAR, S. *What is Apache Thrift? Is it the same as gRPC?* 2023. Disponível em: https://medium.com/devglossary/what-is-apache-thrift-is-it-the-same-as-grpc-2562dba125b0. Acesso em: 13 jun. 2024.

CHAURASIA, B.; VERMA, A.; VERMA, P. An in-depth and insightful exploration of failure detection in distributed systems. *Computer Networks*, Elsevier BV, v. 247, p. 110432, jun. 2024. ISSN 1389-1286.

CHEN, J. et al. Remote procedure call as a managed system service. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, 2023. p. 141–159. ISBN 978-1-939133-33-5. Disponível em: https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong.

COULSON, G. et al. A component model for building systems software. Acta Press, 2004.

CRUZ, F. W. Chamada de procedimento remoto. Acesso em: 12 jun. 2024. 2013.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, Association for Computing Machinery (ACM), v. 35, n. 6, p. 26–36, jun. 2000. ISSN 1558-1160. Disponível em: http://dx.doi.org/10.1145/352029.352035.

DIAS, R. S. et al. Runtime microservice self-distribution for fine-grain resource allocation. In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. [S.l.: s.n.], 2022. p. 234–239.

DINIZ, H. B. M. *Linguagem específica de domínio para abstração de solução de processamento de eventos complexos*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2016.

FENG, Y. et al. Domain specific language for finite element modeling and simulation. *Advances in Engineering Software*, Elsevier BV, v. 193, p. 103666, jul. 2024. ISSN 0965-9978.

FEYNMAN, R. Ebnf: A notation to describe syntax. *Cited on*, v. 10, 2016.

GENG, L. et al. Rr-compound: Rdma-fused grpc for low latency and high throughput with an easy interface. *IEEE Transactions on Parallel and Distributed Systems*, p. 1–18, 2024.

KELLY, S.; TOLVANEN, J. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2007. ISBN 9780470249260. Disponível em: http://dx.doi.org/10.1002/9780470249260.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, v. 36, n. 1, p. 41–50, 2003.

KINA, L. F. E. et al. Definição de linguagem para geração de proxies para dar suporte a sistemas auto-distribuídos. 2023.

KLEIN, M. J. *Formalização da Linguagem Guaraná DSL - Uma Abordagem Matemática para Especificação Formal da Sintaxe Abstrata Utilizando Notação Z*. Applied Computing Research Group - GCA, 2015. Disponível em: http://rgdoi.net/10.13140/RG.2.1.1731.0168.

KLEPPMANN, M. Distributed systems. *Maarten van Steen*, 2018.

KRASANAKIS, E.; SYMEONIDIS, A. Forward-oriented programming: A meta-dsl for fast development of component libraries. *Information and Software Technology*, Elsevier BV, v. 171, p. 107474, jul. 2024. ISSN 0950-5849.

KUMAR, A. et al. Pyramis: Domain specific language for developing multi-tier systems. In: *Proceedings of the 8th Asia-Pacific Workshop on Networking*. ACM, 2024. (APNet 2024, v. 33), p. 156–162. Disponível em: http://dx.doi.org/10.1145/3663408.3663431.

OLIVEIRA, A. P. d. et al. Um estudo sobre sistemas auto-distributivos em ambientes elásticos. 2022.

OLIVEIRA, M. Ferreira de. *Metodologia científica: um manual para a realização de pesquisas em Administração*. 2011.

PANAYIOTOU, K. et al. Smauto: A domain-specific-language for application development in smart environments. *Pervasive and Mobile Computing*, v. 101, p. 101931, 2024. ISSN 1574-1192.

PORTER, B.; RODRIGUES-FILHO, R. A programming language for sound self-adaptive systems. In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. [S.l.: s.n.], 2021. p. 145–150.

PREGUIçA, N. et al. Dotted version vectors: Logical clocks for optimistic replication. *arXiv preprint arXiv:1011.5808*, 2010.

RAJA, A.; LAKSHMANAN, D. Domain specific languages. *International Journal of Computer Applications*, Foundation of Computer Science, v. 1, n. 21, p. 105–111, fev. 2010. ISSN 0975-8887. Disponível em: http://dx.doi.org/10.5120/37-640.

RH, A. et al. Rpc relay protocol for naas platforms. In: *2024 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*. [S.l.]: IEEE, 2024.

ROCHA, E. N. Uso de java e corba para o desenvolvimento de sistemas distribuídos. *Pesquisa e Prática Docente no Curso de Engenharia de Controle e Automação - Universidade Do Contestado, Curitibanos/SC*, v. 89460, p. 112, 2017.

RODRIGUES-FILHO, R. et al. Exploiting the potential of the edge-cloud continuum with self-distributing systems. In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. [S.l.: s.n.], 2022. p. 255–260.

RODRIGUES-FILHO, R. et al. A self-distributing system framework for the computing continuum. In: *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*. [S.l.: s.n.], 2023. p. 1–10.

RODRIGUES-FILHO, R.; PORTER, B. Autonomous state-management support in distributed self-adaptive systems. In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. [S.l.: s.n.], 2020. p. 176–181.

RODRIGUES-FILHO, R.; PORTER, B. Hatch: Self-distributing systems for data centers. *Future Generation Computer Systems*, v. 132, p. 80–92, 2022. ISSN 0167-739X.

RODRIGUES-FILHO, R. et al. Emergent software systems: Theory and practice. *Sociedade Brasileira De Computação*, 2021.

ROSS, J. *Redes de computadores*. [S.l.]: Julio Ross, 2008.

STERN, B. A. Interactive data language. In: *Space 2000*. [S.l.: s.n.], 2000. p. 1011–1015.

STOJANOVIć, A.; PLEHATI, S.; KOVAčEVIć, Z. A domain-specific language implementation framework for c++ based on s-expressions. In: *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*. IEEE, 2024. Disponível em: http://dx.doi.org/10.1109/MIPRO60963.2024.10569883.

WAIN, R.; ASHWORTH, M. *A Java GUI and distributed CORBA client-server interface for a coastal ocean model*. [S.l.]: Citeseer, 2005.

WOODFIN, T. R. *Self-distributing computation*. Tese (Doutorado) — Massachusetts Institute of Technology, 2002.

ZHANG, J. et al. Load balancing in data center networks: A survey. *IEEE Communications Surveys & Tutorials*, v. 20, n. 3, p. 2324–2352, 2018.