UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Romeu Leite de Araujo Filho

APRIMORAMENTO E ANÁLISE DA EFICÁCIA DA FERRAMENTA JS-DISTRIBUTOR EM PROJETOS DE DESENVOLVIMENTO DE SOFTWARE BASEADOS EM MICROSSERVIÇOS

São Carlos, SP
2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Romeu Leite de Araujo Filho

APRIMORAMENTO E ANÁLISE DA EFICÁCIA DA FERRAMENTA JS-DISTRIBUTOR EM PROJETOS DE DESENVOLVIMENTO DE SOFTWARE BASEADOS EM MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia de Computação da Universidade Federal de São Carlos, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Daniel Lucrédio

São Carlos, SP
2025

Dedico este trabalho a todas as pessoas que, de alguma forma, contribuíram para que esta conquista fosse possível. E, acima de tudo, a Deus, por me dar forças e sabedoria para concluir esta etapa da minha vida.

Agradecimentos

Gostaria de começar expressando minha sincera gratidão a Deus, que me manteve firme e saudável durante todo o percurso deste projeto de pesquisa, me dando forças para chegar até aqui.

Agradeço primeiramente à minha família, pelo apoio incondicional e amor constante ao longo de toda a minha vida. Sem vocês, nada disso seria possível.

Sou também imensamente grato ao meu orientador, o Prof. Dr. Daniel Lucrédio, pela orientação paciente e pela generosidade em dedicar seu tempo, ao meu projeto de pesquisa. Seu incentivo foi fundamental para que eu pudesse superar os desafios e concluir este trabalho.

Deixo um agradecimento especial à Universidade Federal de São Carlos e a todos os professores do meu curso, pela elevada qualidade do ensino e pelo comprometimento em me ajudar a me formar como profissional.

A todas as pessoas incríveis que cruzaram o meu caminho, tanto dentro quanto fora da universidade, minha eterna gratidão. Cada um de vocês, com suas histórias, ensinamentos e gestos de apoio, acrescentou algo essencial à minha jornada e a tornou ainda mais significativa.

Por fim, quero agradecer de coração a todos que, de alguma forma, contribuíram para a realização deste projeto. Seja com um simples conselho, um gesto de apoio ou uma colaboração direta, cada contribuição foi valiosa e essencial para a conclusão deste trabalho. Cada um de vocês tem um espaço especial na minha trajetória, e sou profundamente grato por tudo o que vivi e aprendi até aqui.

"– Se eu vi mais longe, foi porque estava sobre os ombros de gigantes."

(Sir Isaac Newton, 1675)

Resumo

Este trabalho visa aprimorar e analisar a eficácia da ferramenta `js-distributor` em projetos de desenvolvimento de software baseados em microsserviços. A arquitetura de microsserviços é amplamente adotada devido à sua flexibilidade, permitindo que diferentes componentes sejam desenvolvidos e implantados independentemente. No entanto, a definição da arquitetura e a decomposição de sistemas monolíticos em microsserviços ainda são desafiadoras, principalmente pela falta de ferramentas automatizadas que suportem a etapa final dessa transição de forma eficiente. `js-distributor` permite transformar código monolítico em microsserviços, mantendo a lógica original da aplicação e facilitando sua implantação em ambientes de produção. Essa pesquisa apresenta uma análise da ferramenta em diferentes cenários, destacando os aprimoramentos feitos para ampliar os casos de uso possíveis e refatorar lógicas envolvendo *callbacks* (chamadas de função passadas por parâmetros), uma vez que migrar esse tipo de lógica para microsserviços pode ser desafiador. Testes realizados demonstraram que a ferramenta se mostrou eficaz na geração de microsserviços a partir das melhorias implementadas.

Palavras-chave: Geração de código, aplicações distribuídas, desenvolvimento web, serviços web, API HTTP, sistemas de mensagens.

Abstract

This work aims to enhance and analyze the effectiveness of the `js-distributor` tool in software development projects based on microservices. The microservices architecture is widely adopted due to its flexibility, allowing different components to be developed and deployed independently. However, defining the architecture and decomposing monolithic systems into microservices is still challenging, primarily due to the lack of automated tools that efficiently support the final stage of this transition. `js-distributor` allows transforming monolithic code into microservices while preserving the original application logic and facilitating its deployment in production environments. This research presents an analysis of the tool in different scenarios, highlighting improvements made to expand possible use cases and refactor logic involving *callbacks* (function calls passed as parameters), as migrating this type of logic to microservices can be challenging. Tests conducted demonstrated that the tool was effective in generating microservices from the implemented improvements.

Keywords: Code generation, distributed applications, web development, web services, HTTP API, messaging systems.

Lista de ilustrações

Figura 1 – Comportamento da Exchange 13
Figura 2 – Direct Exchange 14
Figura 3 – Fanout Exchange 14
Figura 4 – Topic Exchange 15
Figura 5 – Protótipo inicial desenvolvido como um monolito e depois migrado para um único microsserviço 16
Figura 6 – Aplicação distribuída em dois microsserviços. 17
Figura 7 – Aplicação reorganizada em quatro servidores, com um cliente React e RabbitMQ. 18
Figura 8 – Execução distribuída em dois servidores 20
Figura 9 – Principais componentes da `js-distributor` 21
Figura 10 – Detalhes internos da execução em tempo de execução 22
Figura 11 – Nós de comunicação RPC 28
Figura 12 – Comunicação entre dois nós 29
Figura 13 – Implementação com Fanout 31
Figura 14 – Implementação com Topic 32
Figura 15 – Cliente e Servidor 41
Figura 16 – Logica do Cliente e Servidor 42

Sumário

1 INTRODUÇÃO 10

2 FUNDAMENTAÇÃO TEÓRICA 12
2.1 Principais conceitos envolvidos 12
2.1.1 RabbitMQ e os modelos de comunicação assíncrona 12
2.1.1.1 Remote Procedure Call ou RPC 12
2.1.1.2 Exchanges 13
2.1.1.3 Direct Exchange 13
2.1.1.4 Fanout Exchange 13
2.1.1.5 Topic Exchange 14
2.1.2 A ferramenta `js-distributor` 15
2.1.2.1 Exemplo ilustrativo 15
2.1.2.2 Uso da `js-distributor` 18
2.1.2.3 Arquitetura interna da `js-distributor` 20
2.1.2.4 Limitações da `js-distributor` antes da realização deste trabalho 22
2.2 Trabalhos relacionados 23

3 METODOLOGIA 25

4 RESULTADOS 27
4.1 Testes utilizando a Ferramenta 27
4.1.1 Remote Procedure Call 28
4.1.2 Direct Exchange 30
4.1.3 Fanout Echange 30
4.1.4 Topic Exchange 31
4.2 Refatoração manual de funções com callbacks 32

5 CONCLUSÃO 37
5.1 Trabalhos futuros 37

REFERÊNCIAS 38

APÊNDICES 40

APÊNDICE A – REFATORAÇÃO DE FUNÇÕES PASSADAS COMO PARÂMETROS PARA EXECUÇÃO DISTRIBUÍDA 41

A.1 Server Node **42**
A.2 Client Node **44**

1 Introdução

À medida que o número de usuários em uma determinada aplicação aumenta muito, as arquiteturas tradicionais de software se mostram insuficientes para lidar com as demandas de escalabilidade, flexibilidade e resiliência. Nesse contexto, a arquitetura de microsserviços se destaca. Segundo Fowler (2014), essa é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um rodando em seu próprio processo e se comunicando com mecanismos leves. Esses serviços são construídos em torno de recursos de negócios e podem ser implantados independentemente por ferramentas de implantação totalmente automatizado. Há um mínimo de gerenciamento centralizado desses serviços, que podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de armazenamento de dados. Esse modelo permite que diferentes equipes de desenvolvimento trabalhem em paralelo, cada uma gerenciando um microsserviço distinto.

Entretanto, ao adotar uma arquitetura distribuída alguns problemas podem surgir no processo de migração do monolito. Kalske, Mäkitalo e Mikkonen (2018) afirmam que os principais desafios são a complexidade na divisão dos serviços, que pode gerar sobrecarga de desempenho se os serviços forem excessivamente detalhados, além da integração entre microsserviços, que exige uma tecnologia neutra para evitar dependência de linguagens específicas. Além disso, a implementação de uma infraestrutura robusta de monitoramento é crucial, já que o aumento no número de serviços torna mais difícil detectar falhas rapidamente, diferentemente de uma arquitetura monolítica. Por fim, é necessário garantir que os serviços sejam autônomos e possuam uma boa interface, sem comprometer a performance, o que requer cuidado na definição de suas responsabilidades e comunicação.

Apesar dessas complexidades, Kumar (2023) afirma que as arquiteturas distribuídas possuem inúmeros benefícios oferecendo experiências consistentes aos usuários em uma variedade de plataformas. Uma das principais vantagens é a capacidade de adaptação rápida e flexibilidade que proporciona aos desenvolvedores poder ajustar ou até substituir partes do sistema de forma mais ágil, sem a necessidade de uma reformulação completa. Isso é crucial para aplicações que precisam continuamente responder rapidamente às mudanças de mercado. Portanto, a capacidade de redesenhar ou reconfigurar a distribuição de um sistema, aproveitando os avanços tecnológicos e as novas necessidades dos usuários, pode ser um fator competitivo decisivo, permitindo que a empresa se mantenha relevante em um cenário dinâmico.

Diante desse contexto, a ferramenta `js-distributor`[1] foi desenvolvida para auxiliar os desenvolvedores na migração de códigos monolitos para microsserviços. A ferramenta gera automaticamente o código para configuração de servidores e comunicação via requisições HTTP ou mensagens assíncronas baseadas em filas *RabbitMQ*. As funções podem ser inicialmente implementadas e implantadas em um único código monolítico e posteriormente migradas para uma arquitetura diferente sem a necessidade de adicionar ou modificar código. No entanto, a `js-distributor` não possuía suporte a diferentes configurações do *RabbitMQ*, como múltiplos *workers* e transmissão de mensagens com auxílio de *exchanges*.

O objetivo deste trabalho foi aprimorar a ferramenta `js-distributor` para incluir suporte para comunicação entre serviços através de *exchanges* do tipo *direct*, *topic*, *fanout* e aprimorar a comunicação já implementada a partir da chamada de procedimento remoto, assim como avaliar as melhorias implementadas. Também foi elaborada uma refatoração para lidar com o caso de funções que recebem *callbacks* como parâmetros. A refatoração não chegou a ser implementada no trabalho, mas foi testada manualmente e considerada correta nos testes realizados, portanto espera-se que sua implementação na ferramenta exija um esforço mínimo.

Este trabalho está organizado em cinco capítulos. No presente Capítulo foi apresentada uma contextualização para a proposta do trabalho, além de seu objetivo. O Capítulo 2 traz uma fundamentação teórica e a descrição de trabalhos relacionados ao projeto. No Capítulo 3 é descrita a metodologia utilizada para realização dos experimentos, cujos detalhes dos resultados estão no Capítulo 4. Por fim, o Capítulo 5 traz as conclusões obtidas com o trabalho e propostas de trabalhos futuros.

[1] https://github.com/dlucredio/js-distributor

2 Fundamentação teórica

Neste capítulo são apresentados os principais conceitos necessários para o entendimento deste trabalho (Seção 2.1) e os trabalhos relacionados (Seção 2.2).

2.1 Principais conceitos envolvidos

Este projeto utiliza a ferramenta `js-distributor` para transformar um código *JavaScript* monolítico em microsserviços independentes, distribuindo funções específicas em servidores distintos. A comunicação entre os microsserviços é gerida por HTTP API ou pelo *RabbitMQ*, um *broker* de mensagens que organiza o tráfego de informações por meio de filas e *exchanges*[1]. Utilizando o protocolo *AMQP* 0-9-1[2], o *RabbitMQ* permite o roteamento flexível das mensagens, com diferentes tipos de *exchanges* adaptados às necessidades do projeto. Essa arquitetura distribuída, com o suporte do *RabbitMQ*, garante que as mensagens sejam entregues de maneira eficiente, escalável e resiliente entre os microsserviços, otimizando o desempenho do sistema.

2.1.1 RabbitMQ e os modelos de comunicação assíncrona

O *RabbitMQ* é um *broker* de mensagens e transmissão que se destacada pelo suporte a diversos protocolos, além de oferecer diversas bibliotecas de cliente compatíveis com várias linguagens de programação[3]. A flexibilidade do *RabbitMQ* permite combinar várias opções, como roteamento, filtragem e transmissão, para definir como suas mensagens transitam do produtor para um ou mais consumidores. Com a capacidade de confirmar a entrega das mensagens e replicá-las em um *cluster*, o *RabbitMQ* garante a segurança e a integridade das suas mensagens, oferecendo confiança no processo de comunicação. Esta seção aborda os diferentes tipos de comunicação que o *RabbitMQ* oferece e foram abordadas no desenvolvimento do projeto.

2.1.1.1 Remote Procedure Call ou RPC

O RPC no *RabbitMQ* permite que um cliente faça uma requisição para um servidor e aguarde uma resposta, criando uma comunicação exclusiva entre os dois. Nesse comportamento o cliente cria uma fila exclusiva para receber as respostas e envia uma mensagem

[1] *Exchange* é uma entidade que recebe mensagens e as direciona para filas com base em um tipo de roteamento
[2] https://www.amqp.org
[3] https://www.rabbitmq.com

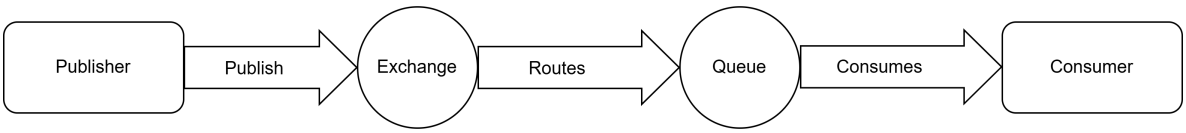
com a propriedade *reply_to*, que especifica a fila onde espera a resposta. O servidor, por sua vez, processa a solicitação e envia a resposta para a fila indicada no campo *reply_to*.

Além do *reply_to*, o padrão RPC no *RabbitMQ* também utiliza o *correlation-id*, que é um identificador único associado à mensagem enviada. Esse ID é crucial para que o cliente consiga identificar qual resposta corresponde à sua requisição original. O servidor, ao enviar a resposta, inclui o *correlation-id* correspondente, permitindo que o cliente faça a correspondência entre a resposta recebida e a solicitação que ele fez. Esse mecanismo evita confusão, especialmente em sistemas com múltiplas requisições simultâneas, garantindo que a resposta correta seja associada à requisição certa.

2.1.1.2 Exchanges

A Figura 1 ilustra o comportamento de uma *exchange*. No protocolo *AMQP*, as *exchanges* são componentes fundamentais que recebem mensagens enviadas pelos produtores (*Publisher*, na Figura) e as encaminham para as filas apropriadas, com base em regras de roteamento (*Routes*, na Figura) definidas. Exchanges não armazenam as mensagens, mas determinam para qual fila (*Queue*, na Figura) cada mensagem será direcionada, dependendo do tipo (*Direct*, *Fanout* ou *Topic*) e das condições de roteamento configuradas. Uma vez na fila, as mensagens podem ser consumidas pelo consumidor (*Consumer*, na Figura).

Figura 1 – Comportamento da Exchange



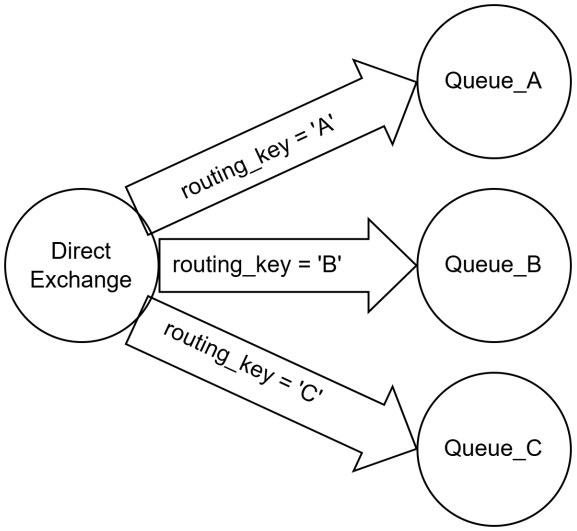
2.1.1.3 Direct Exchange

A *exchange* do tipo *Direct* encaminha mensagens para uma determinada fila a partir de uma propriedade *routing_key* na Figura 2. Esse tipo de *exchange* é ideal para cenários em que o roteamento de mensagens precisa ser muito específico, permitindo que produtores enviem mensagens diretamente para uma fila particular com base em uma correspondência exata entre a chave de roteamento e a configuração das filas aplicando o roteamento *unicast*.

2.1.1.4 Fanout Exchange

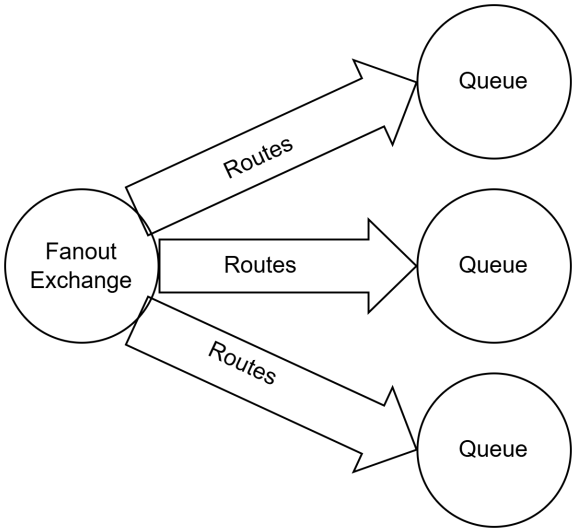
Diferente do tipo *direct*, a *exchange* do tipo *Fanout* possui o comportamento *broadcast*, que é um tipo de roteamento que encaminha a mensagem para todas as filas vinculadas a ele. Quando uma mensagem é publicada em um *fanout exchange*, todas as

Figura 2 – Direct Exchange



filas associadas recebem uma cópia da mensagem, sem a necessidade de um critério específico para o roteamento. Esse tipo de *exchange* é muito útil quando as mensagens precisam ser distribuídas para múltiplos consumidores simultaneamente como mostrado na Figura 3.

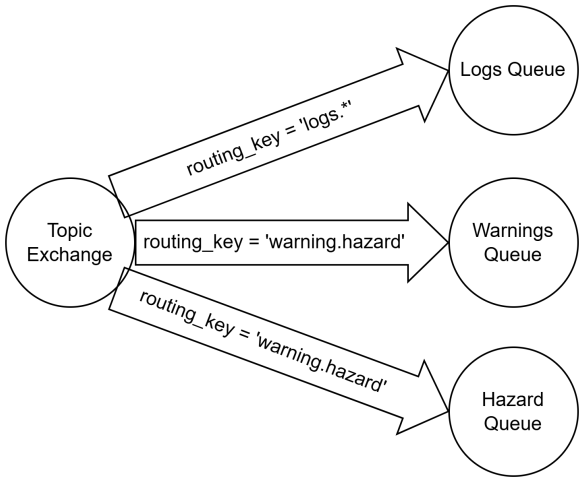
Figura 3 – Fanout Exchange



2.1.1.5 Topic Exchange

Por fim, a *exchange* do tipo *Topic* possui um comportamento *multicast*, ou seja, são feitos múltiplos *broadcasts*. As filas podem ser associadas a padrões de tópicos que utilizam caracteres especiais, como o asterisco (*) e o cerquilha (#), para corresponder a múltiplas chaves de roteamento Figura 4. Essa função possibilita a implementação de

Figura 4 – Topic Exchange



vários produtores e consumidores associados a um padrão de roteamento, sendo ideal para sistemas em que é necessário filtrar ou dividir mensagens em categorias mais dinâmicas.

2.1.2 A ferramenta `js-distributor`

Esta seção descreve a ferramenta `js-distributor`, e está dividida da seguinte maneira. A Subseção 2.1.2.1 foca em um exemplo ilustrativo do cenário onde a ferramenta pode ser útil. A Subseção 2.1.2.2 mostra um cenário de uso da ferramenta. A Subseção 2.1.2.3 descreve a arquitetura interna da ferramenta. Por fim, a Subseção 2.1.2.4 lista as limitações da ferramenta antes da realização deste trabalho.

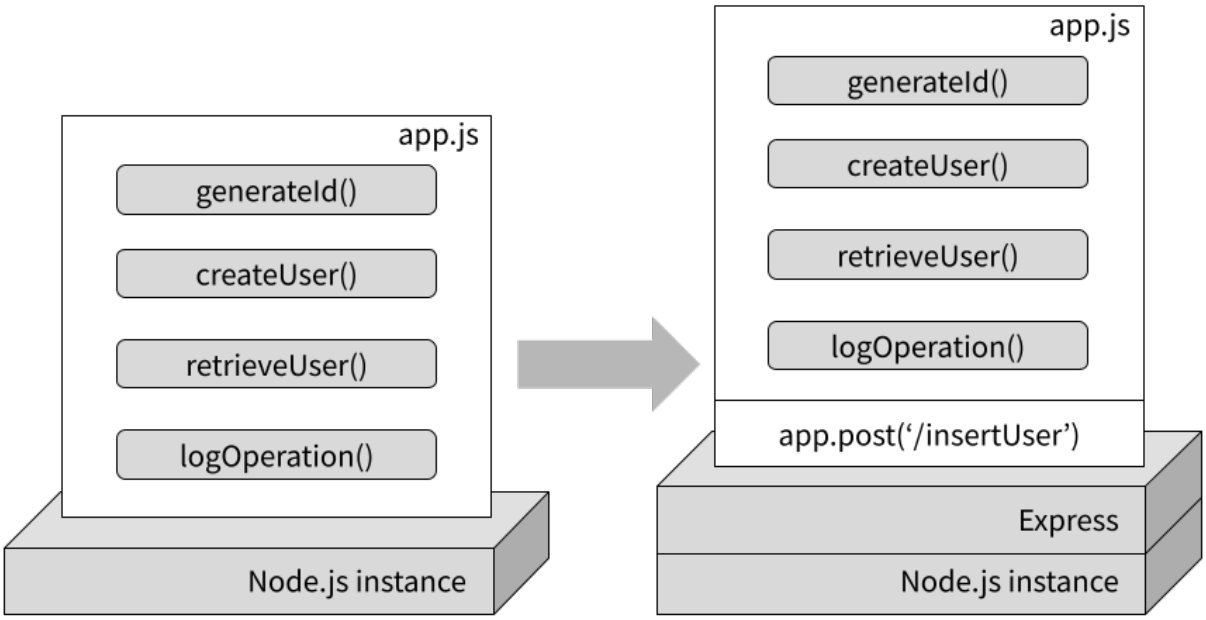
2.1.2.1 Exemplo ilustrativo

Para ilustrar um caso de uso da ferramenta podemos imaginar o seguinte cenário: um desenvolvedor precisa criar um software que execute as seguintes funções: (1) Gerar um identificador concatenando alguns dados do usuário (por exemplo, o e-mail) com um UUID compatível com RFC41227; (2) Inserir o usuário no banco de dados, usando o identificador gerado como chave; (3) Recuperar o usuário recentemente salvo no banco de dados para obter o *timestamp* exato de inserção; e (4) Registrar a operação para auditoria.

No início, o desenvolvedor pode focar apenas na lógica e criar quatro funções que são executadas em uma única máquina, utilizando, por exemplo, o *Node.js*. Após os testes, ele pode encapsular toda a lógica em um *endpoint* de *API HTTP* rodando em um servidor web como o *Express*, transformando-a em um microsserviço acessível por outras partes do sistema. Isso é uma tarefa simples, que pode ser feita com algumas linhas de código. A Figura 5 a seguir ilustra esse primeiro desenvolvimento.

Agora, imagine que essa arquitetura foi implantada e o sistema começa a receber

Figura 5 – Protótipo inicial desenvolvido como um monolito e depois migrado para um único microsserviço

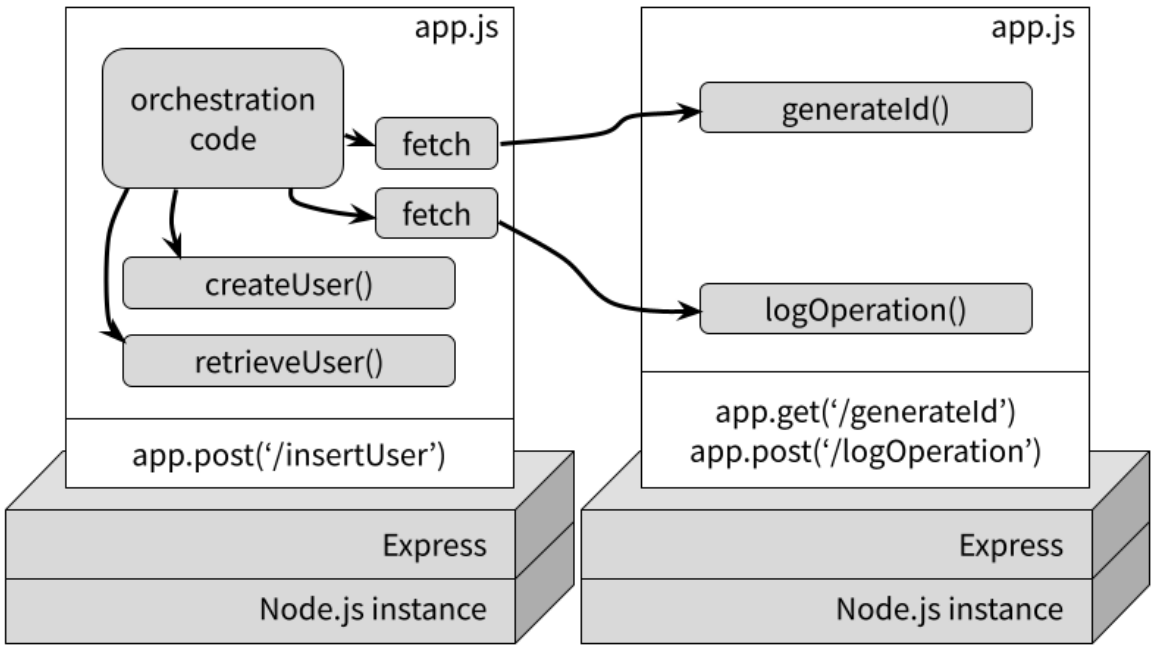


uma grande quantidade de novos usuários. Com o tempo, começam a surgir atrasos. Após uma investigação mais profunda, a equipe de desenvolvimento descobre que seria melhor separar essas funções em dois microsserviços para otimizar o uso dos recursos: (i) um para gerar os *UUIDs* e fazer o *log*. Isso faz sentido, pois essas funções são independentes, não dependem de outras e não exigem hardware potente; e (ii) outro para criar e recuperar os usuários, que é onde ocorre o processamento mais pesado, justificando o uso de um servidor dedicado, possivelmente com mais memória.

A equipe decide manter a solução baseada em *Express* para a API HTTP dos microsserviços, a fim de garantir compatibilidade com outras aplicações. A Figura 6 ilustra essa nova arquitetura. Como se pode ver, a migração para essa nova estrutura não é tão simples quanto antes. Além de criar os novos *endpoints* e configurar os servidores, os desenvolvedores precisam implementar a comunicação entre os microsserviços, o que é chamado de orquestração. Enquanto as funções locais podem ser mantidas como estavam no código original, qualquer chamada a um serviço em outro servidor precisará ser transformada em uma operação remota. Isso implica empacotar os parâmetros da requisição e da resposta conforme o formato de comunicação (como *JSON*, por exemplo), além de lidar com *callbacks* assíncronos, um aspecto comum em linguagens como o *JavaScript*, e com novos erros e exceções relacionados à comunicação.

Embora essa nova arquitetura permita um melhor aproveitamento dos recursos, ainda há oportunidades de melhoria. Por exemplo, algumas operações de leitura do banco de dados, que não exigem consistência imediata, poderiam ser movidas para um servidor diferente, acessando uma réplica do banco que suporte consistência eventual (Brewer,

Figura 6 – Aplicação distribuída em dois microsserviços.

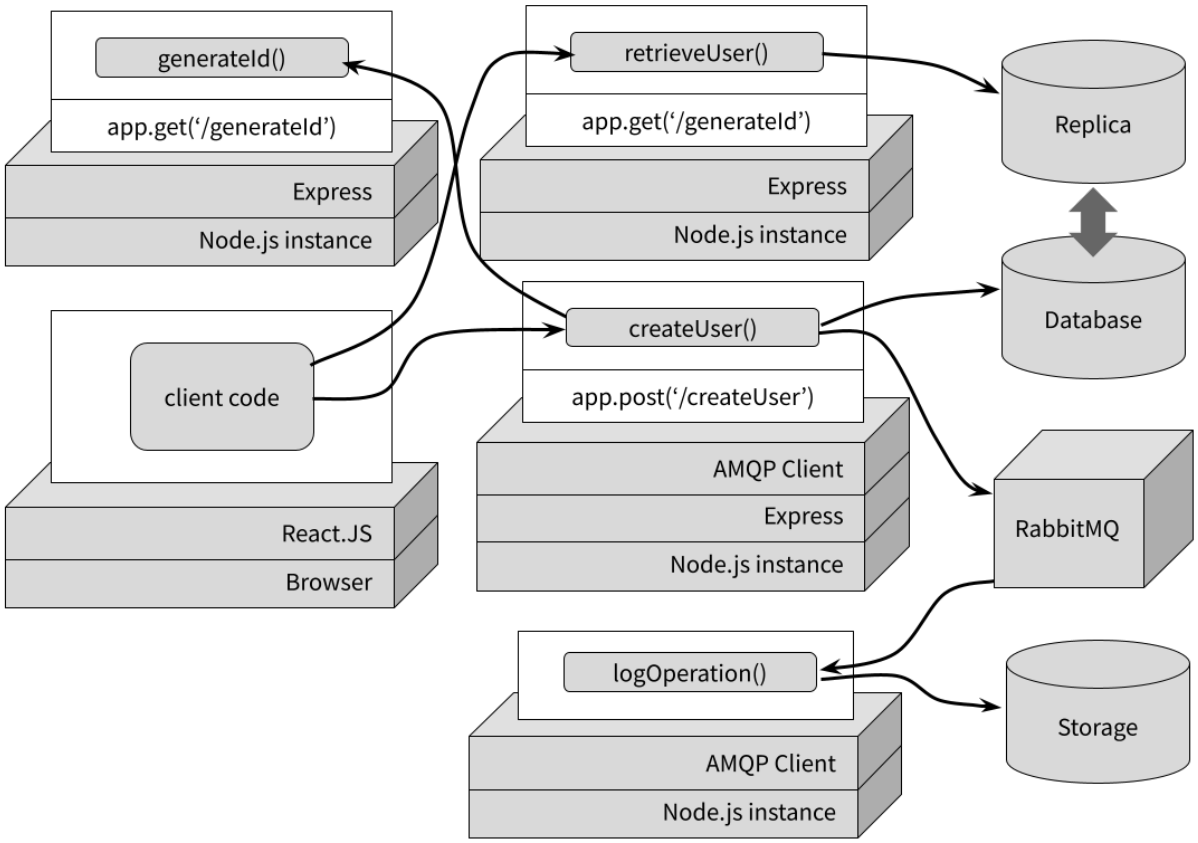


2012). Além disso, a função de *log* poderia ser integrada a um sistema de mensagens do tipo *publish-subscribe*, como o *RabbitMQ*, que executa de forma assíncrona, facilitando a orquestração e reduzindo a latência. Um cliente baseado no *React* poderia realizar chamadas para os microsserviços expostos através da API HTTP. A Figura 7 ilustra essas possibilidades em uma nova arquitetura. Note como a complexidade aumenta devido à quantidade de comunicação necessária entre os servidores.

A `js-distributor` pode gerar automaticamente o código necessário para transformar funções *JavaScript* normais em serviços distribuídos que se comunicam por meio de API HTTP ou mensagens *RabbitMQ*. Também é capaz de detectar quando uma função chamada foi movida de um servidor local para um remoto e gerar automaticamente o código necessário para garantir que continue funcionando corretamente. Além disso, ela realiza a substituição da chamada local por uma chamada remota, lidando com o empacotamento dos parâmetros da requisição e da resposta.

Esse cenário pode ser ampliado para a decomposição de aplicações monolíticas já existentes. A `js-distributor` pode ser combinada com outras metodologias (Abgaz et al., 2023) que ajudam a identificar os melhores candidatos para microsserviços dentro de aplicações monolíticas. Após identificar os candidatos, a ferramenta pode automatizar a criação dos microsserviços finais, prontos para implantação. O resultado é uma solução totalmente automatizada, que preenche a lacuna das ferramentas existentes mencionada por Abgaz et al. (2023).

Figura 7 – Aplicação reorganizada em quatro servidores, com um cliente React e RabbitMQ.



2.1.2.2 Uso da `js-distributor`

O processo começa com o código fonte regular e não distribuído, escrito em *JavaScript*. Isso pode ser uma aplicação monolítica legada existente ou uma nova aplicação desenvolvida sem considerar a distribuição. O único requisito é que o código deve ser composto por funções, que é a maneira normal de se utilizar o *JavaScript*. Considere as funções a seguir: (1) generateId(): gera um ID com base em um prefixo fornecido; (2) createUser(): insere um usuário no banco de dados(PostgreSQL), com um e-mail e nome fornecidos. A função primeiro chama generateId(), passando o e-mail como prefixo para o ID do usuário gerado. Depois, instancia o cliente PostgreSQL e executa a consulta "INSERT", passando os parâmetros necessários; (3) main(): executa a aplicação simplesmente chamando createUser() com um e-mail e nome. A Listagem 2.1 ilustra esse exemplo.

Listing 2.1 – Sample JavaScript code

```
1 import { v4 as uuidv4 } from 'uuid'; // UUID package
2 import pkg from 'pg'; // PostgreSQL module
3 const { Client } = pkg; // PostgreSQL client
4 function generateId(prefix) {
5  console.log(`Generating ID with prefix: ${prefix}`);
6  return `[${prefix}:${uuidv4()}]`;
```

```
}
async function createUser(email, name) {
 console.log(`Create: email=${email}, name=${name}`);
 const id = generateId(email);
 const client = new Client(
   { connectionString: 'postgresql://db:db@localhost:5432/db' });
 try {
  await client.connect();
  const query = 'INSERT INTO Users (id, email, name) VALUES ($1, $2, $3);';
  await client.query(query, [id, email, name]);
 } catch (e) { console.error('Error:', e);
 } finally { await client.end(); }
}
async function main() {
 console.log(`Running application...`);
 await createUser("user@email.com", "John Doe");
 console.log(`Done!`);
}
main();
```

Para utilizar a abordagem proposta para distribuir esse código, o desenvolvedor deve fornecer um arquivo *config.yml* (Listagem 2.2). Este arquivo de configuração descreve completamente como as funções existentes devem ser distribuídas em servidores diferentes, para cada função, o arquivo de configuração deve especificar os parâmetros esperados para cada função. Neste exemplo, foram usados apenas *endpoints* do tipo API HTTP.

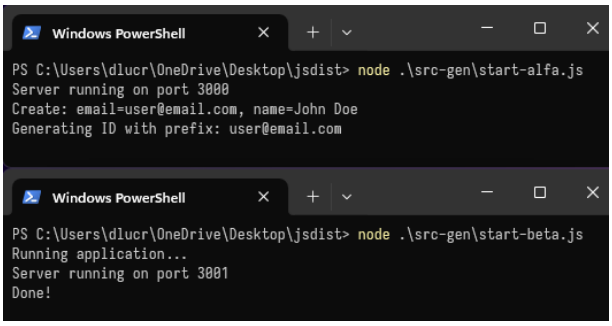
Listing 2.2 – Distribution configuration file

```
servers:
  - id: alpha
    port: 3000
    url: localhost
  - id: beta
    port: 3001
    url: localhost
functions:
  - name: generateId
    parameters:
      - name: prefix
      - type: string
    method: get
    server: alpha
  - name: createUser
    parameters:
      - name: email
        type: string
      - name: name
        type: string
```

```
21      method: post
22      server: alpha
23    - name: main
24      parameters: []
25      method: get
26      server: beta
```

Além do código do servidor, a abordagem também gera código do lado cliente, que pode ser usado para se conectar com as funções neste servidor. Após executar ambos os servidores, a saída mostrada na Figura 8 é obtida. Observe como cada servidor gera apenas as saídas para as funções que ele contém. Além disso, como desejado, a lógica global geral é mantida, com cada microsserviço individual contribuindo para ela.

Figura 8 – Execução distribuída em dois servidores



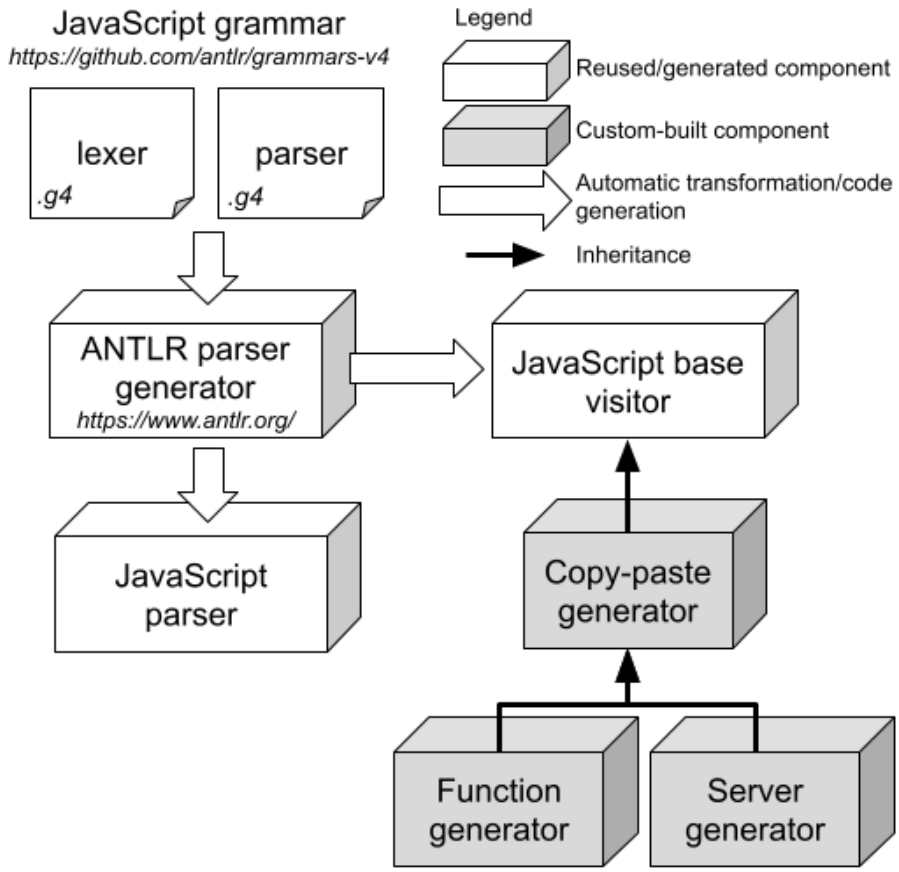
Este exemplo mostra um cenário simples, com dois servidores executando localmente, e apenas com servidores de API HTTP. Mas a abordagem também suporta comunicação via *RabbitMQ*. Por exemplo, o *config.yml* pode ser modificado para incluir um terceiro servidor para hospedar a função generateId(), utilizando uma fila *RabbitMQ* para comunicação.

2.1.2.3 Arquitetura interna da `js-distributor`

A Figura 9 ilustra a arquitetura da `js-distributor`.

A arquitetura é baseada em um analisador *JavaScript* construído com o *ANTLR*, um gerador de analisadores baseado em LL que utiliza uma técnica avançada conhecida como ALL(*) (Parr; Harwell; Fisher, 2014). Esse gerador é amplamente utilizado e oferece suporte para várias linguagens, incluindo *JavaScript*. Como ocorre com a maioria dos geradores de analisadores, ele gera automaticamente um *lexer* e um *parser* com base em uma gramática escrita em *Extended Backus-Naur Form (EBNF)*. A partir de uma gramática para a linguagem *JavaScript14* no formato *ANTLR (.g4)* composta por dois arquivos (lexer e parser), o *ANTLR* gera dois componentes: um *parser* e um visitante base.

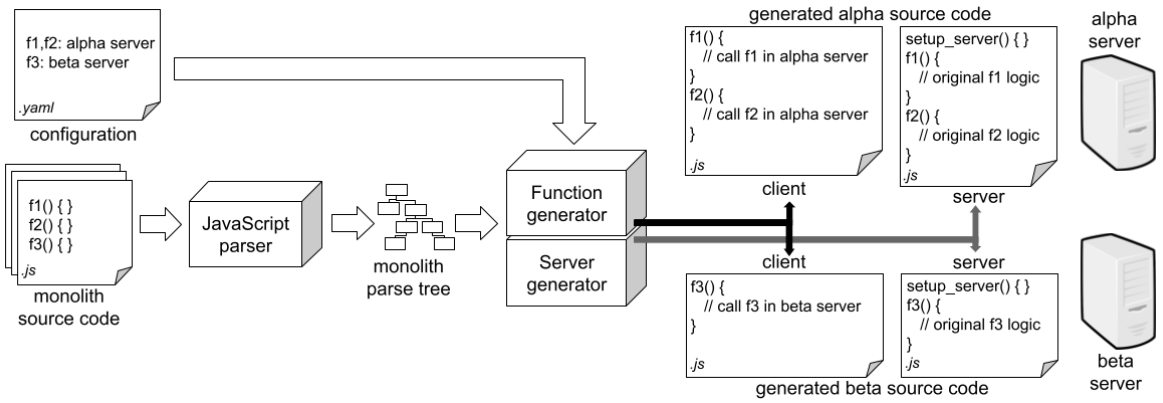
Figura 9 – Principais componentes da `js-distributor`



O *parser* é o componente responsável por ler o arquivo fonte (neste caso, arquivos *JavaScript*) e produzir uma árvore de análise (também conhecida como árvore de sintaxe concreta). O visitante base é simplesmente um conjunto de funções que pode ser reutilizado, por herança, para realizar ações semânticas ao visitar os nós da árvore de análise. Foram desenvolvidos três visitantes:

- Gerador de copiar-colar: este visitante gera código equivalente à árvore de análise. Em outras palavras, ele copia a lógica do código original e a reproduz como está (daí seu nome). Ele é utilizado como base para os outros visitantes, sempre que estes precisem produzir código que deve ser executado exatamente da mesma forma que na aplicação original;
- Gerador de funções: este visitante gera, para cada função que compõe um microsserviço, o código do lado cliente para invocá-la. O código realiza requisições HTTP ou envia mensagens assíncronas para o servidor, conforme a configuração desejada;
- Gerador de servidor: este visitante gera, para cada função que compõe um microsserviço, o código do lado servidor para receber as chamadas. O código cria pontos finais HTTP ou configura uma fila de mensagens para receber mensagens assíncronas dos clientes, conforme a configuração desejada.

Figura 10 – Detalhes internos da execução em tempo de execução



A Figura 10 mostra como esses componentes funcionam durante a execução. No exemplo da figura, o monolito tem três funções, f1(), f2() e f3(), conforme mostrado no lado esquerdo. Existem dois servidores configurados: alpha (contendo f1() e f2()) e beta (contendo f3()), conforme mostrado na configuração no canto superior esquerdo da figura. O analisador *JavaScript* gerado é responsável por analisar o arquivo fonte do monolito e produzir uma árvore de análise sintática. Os visitantes (Gerador de Função e Gerador de Servidor) leem a árvore de análise sintática e, com base no arquivo de configuração, produzem o código do cliente e do servidor para cada função, para que cada uma seja executada em seu respectivo servidor: f1() e f2() em alpha, e f3() em beta, conforme configurado. O código fonte gerado contém versões tanto do cliente quanto do servidor para cada função. A versão cliente é destinada a ser utilizada pelas partes da aplicação que invocarão as funções e é gerada de acordo com a tecnologia de comunicação escolhida (HTTP GET/POST ou *RabbitMQ*). A versão servidor contém a lógica original, além do código de configuração do servidor. Durante o processo de geração de código, os visitantes também substituem as chamadas de funções originais pelas geradas nos clientes, de modo que toda a arquitetura reflita a lógica do monolito. Finalmente, cada partição é implantada em seu respectivo servidor.

2.1.2.4 Limitações da `js-distributor` antes da realização deste trabalho

Apesar de possuir um funcionamento bastante eficiente, a versão original da ferramenta `js-distributor` apresentava algumas limitações importantes em relação aos modelos de comunicação suportados. Ela não era capaz de lidar com diferentes modelos de comunicação (*unicast*, *multicast* e *broadcast*) e não tinha suporte para múltiplos *workers*. O sistema estava limitado a um modelo mais simples de RPC, mas com limitações importantes. Não havia garantia de unicidade das filas ou das mensagens enviadas, o que poderia gerar problemas em cenários com múltiplas chamadas simultâneas entre os mesmos nós. Na prática, utilizava-se sempre o mesmo nome de fila, e o *correlation-id* não era empregado para garantir a verificação da mensagem de retorno, o que não atendia a

todas as necessidades de arquitetura distribuída de aplicações mais complexas.

Além disso, o arquivo de configuração, que era responsável por estruturar a distribuição das funções entre os servidores, continha parâmetros desnecessários. Esses parâmetros extras tornavam o processo de configuração mais complexo de manter, sem adicionar valor real ao funcionamento do sistema. Outro ponto crítico era que a abordagem utilizada não suportava casos em que funções precisavam receber outras funções como parâmetros, como *callbacks*. Isso limitava bastante a flexibilidade do sistema em diferentes tipos de interações entre as funções distribuídas.

2.2 Trabalhos relacionados

Em um estudo que revisou sistematicamente a literatura, Abgaz et al. (2023) apresentaram um *framework* de decomposição de monolito em microsserviços M2MDF que identifica as principais fases e elementos-chave da decomposição. Os resultados da análise sugerem que a etapa de implantação de microsserviços, crucial para validar a eficácia da sua aplicação, carece de estudos significativos, além de uma ausência de uma descrição do processo de decomposição de ponta a ponta.

Em uma pesquisa semelhante, Abdullah, Iqbal e Erradi (2019) exploraram a implantação de diferentes configurações de microsserviços baseada em uma abordagem de caixa-preta com auxílio de um método de aprendizado de máquina não supervisionado. O estudo utilizou métricas de desempenho como tempo de resposta e vazão para verificar a escalabilidade da aplicação. No entanto, a abordagem deles replica o monolito inteiro, redirecionando as chamadas, em vez de particionar o código.

Esperança e Lucrédio (2017) propõem um algoritmo de particionamento para protótipos baseados em Java utilizando *Remote Method Invocation*. Nos testes realizados, o particionamento foi bem sucedido, com a versão distribuída reproduzindo fielmente a funcionalidade da versão monolítica. Além disso, foram observadas melhorias em termos de consumo de memória, ainda que a latência tenha sofrido uma queda de desempenho considerável. O algoritmo também apresentou limitações por se basear em uma linguagem orientada a objetos fortemente tipada, o que trouxe desafios ao processo de transformação automática. Uma linguagem mais flexível, como *JavaScript*, apresenta facilidades nesse processo. Além disso, o trabalho de Esperança e Lucrédio (2017) não explora comunicação assíncrona, algo que a ferramenta `js-distributor` incorpora.

Outro estudo relevante é o de Kaplunovich (2019), que investiga uma arquitetura *serverless* para decompor sistemas monolíticos, transformando métodos *Java* em funções *AWS Lambda* em *Node.js*. Ele aborda os desafios enfrentados enfatizando a análise e transformação, destacando as vantagens da arquitetura *serverless* em comparação com as demais. Apesar das contribuições, sua abordagem não permite que o desenvolvedor

escolha quais funções devem ser distribuídas.

De maneira similar, Carvalho e Araújo (2019) discutem a computação em nuvem realizando a conversão automática de aplicações *Node.js* em funções *serverless* a partir de um framework chamado *Node2FaaS*. Apesar dos ganhos de desempenho em aplicações de grande uso de entrada/saída de arquivos, eles destacam que nem todos os cenários se beneficiam da decomposição, corroborando as observações de Esperança e Lucrédio (2017), e ressaltam a importância da inspeção automática da aplicação para guiar as decisões de distribuição.

Aplicações como o *Service Weaver* [4] e *frameworks* como o *Temporal* também são uma alternativa à distribuição automática de serviços. Essas ferramentas simplificam a divisão da lógica de negócios em componentes menores e gerenciam os *workflows* de maneira consistente e tolerante a falhas, mas exigem que o desenvolvedor especifique os serviços previamente.

Soluções baseadas em IA, como o *ChatGPT*, têm sido cada vez mais adotadas para auxiliar na decomposição de código monolítico em microsserviços. Contudo, essas ferramentas são caracterizadas por um comportamento não determinístico, o que pode gerar resultados inconsistentes ou inesperados. Isso ocorre porque a IA pode produzir diferentes respostas para o mesmo problema, tornando o processo menos previsível e com a necessidade de ajustes adicionais.

[4] https://serviceweaver.dev/

3 Metodologia

Neste trabalho, o objetivo foi realizar melhorias na ferramenta `js-distributor`, uma solução capaz de distribuir código de um projeto *JavaScript* monolítico entre múltiplos servidores, utilizando *RabbitMQ* para comunicação entre as funções distribuídas. A metodologia para a implementação das melhorias envolveu as etapas de instalação, configuração, adaptação do protótipo existente e análise das alterações realizadas.

O primeiro passo foi garantir que o ambiente estivesse corretamente configurado para executar a ferramenta. Para isso, o repositório do `js-distributor` foi clonado diretamente do *GitHub* para a máquina local. Em seguida, as dependências necessárias foram instaladas bem como todas as bibliotecas necessárias. Como a ferramenta depende do *RabbitMQ* para a comunicação entre os servidores, também foi necessário instalar a biblioteca *AMQP*, além de configurar o *RabbitMQ* na máquina local. Para completar, foi necessário instalar o *ANTLR*, que é utilizado para processar a linguagem de configuração e gerar o código de distribuição.

Após a instalação, o próximo passo foi configurar o ambiente de desenvolvimento. Foi criada uma estrutura de diretórios no projeto, incluindo a pasta *src* onde foram definidas funções de exemplo. Em seguida, um arquivo de configuração no formato *YAML* foi criado, o qual define os servidores e as funções que seriam distribuídas. O arquivo *config.yml* especifica a alocação de funções para diferentes servidores e a forma como esses servidores se comunicam entre si, separando os papéis de produtores e consumidores de mensagens. O protótipo original da ferramenta foi executado com o comando *npm run generate-single*, gerando o código necessário para testar a distribuição das funções entre os servidores. Durante a execução, foram gerados os arquivos, cada um desses arquivos representava um servidor ou uma função. Os servidores foram testados localmente em terminais separados.

A partir da análise do funcionamento original do `js-distributor`, foram implementadas diversas melhorias. Primeiramente, foi otimizada a configuração das filas da comunicação RPC já existente na ferramenta, posteriormente foram adicionadas as implementações referentes às *exchanges*. Além dessas melhorias, o arquivo de configuração *YAML* também foi otimizado para se adequar a maiores possibilidades de testes. Por fim, foi realizado um estudo para possibilitar que funções passadas como parâmetros possam ser automaticamente distribuídas, utilizando o *RabbitMQ* como meio de comunicação entre os nós. A partir desse estudo, foi criada uma especificação de refatoração, a qual descreve o passo a passo necessário para transformar qualquer função que receba uma função como parâmetro em um microsserviço distribuído, de forma que a lógica original seja

preservada. Essa refatoração foi testada em um sistema no qual essa situação já ocorre. No futuro, essa refatoração será implementada na `js-distributor`.

Com as melhorias implementadas, o próximo passo foi realizar testes *ad-hoc* para garantir que o sistema de distribuição de funções estivesse operando conforme o esperado. Esses testes foram realizados de forma exploratória, com a execução do código gerado em múltiplos servidores para verificar a comunicação entre as filas do *RabbitMQ* e o processamento correto das funções distribuídas. O objetivo foi identificar comportamentos inesperados ou falhas no sistema, avaliando seu desempenho sob diferentes condições. Durante o processo, a análise foi feita de forma flexível com foco nas melhorias implementadas visando os resultados desejados e assegurando que o comportamento inicial das funções do monolito se mantivesse o mesmo.

4 Resultados

A fim de aferir o comportamento das modificações propostas na ferramenta, foram realizadas algumas demonstrações levando em consideração diferentes cenários de teste, visando uma maior abrangência e comparando o comportamento das possibilidades de distribuição do monolito.

Nos testes feitos, 100% dos casos foram bem sucedidos, ou seja, dado um sistema monolítico, foram geradas distribuições em diferentes combinações, para valiar os aprimoramentos implementados, e que em todos os casos o comportamento original do monolito foi mantido. Isso foi verificado por meio de testes *ad hoc*.

4.1 Testes utilizando a Ferramenta

Para os exemplos a seguir foi utilizado um monolito *JavaScript* que calcula se um número é divisor comum de dois outros números. A função *calculateRemainder* retorna o resto da divisão de dois números, a função *isDivisor* avalia e determina se é um divisor caso o resto seja igual a 0, a função *isDivisorOfBoth* retorna verdadeiro caso um número seja divisor de ambos e por fim a *checkCommonDivisor* avalia o retorno da função anterior e define se um número é divisor comum de dois outros números.

Listing 4.1 – Common Divisor Monolith

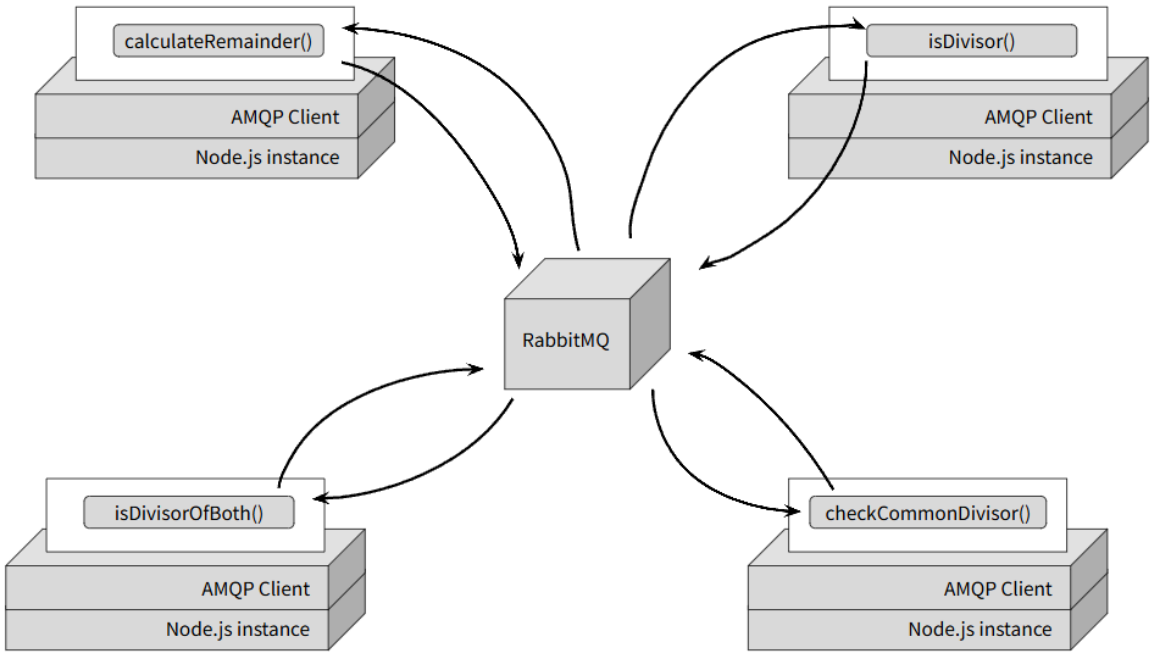
```
1 function calculateRemainder(a, b){
2     return b % a;
3 }
4
5 function isDivisor(a, b){
6     const remainder = calculateRemainder(a, b);
7
8     if(remainder === 0) return true
9     else return false
10 }
11
12 function isDivisorOfBoth(divisor, a, b){
13     return isDivisor(divisor, a) && isDivisor(divisor, b);
14 }
15
16 function checkCommonDivisor(divisor, a, b){
17     if(isDivisorOfBoth(divisor, a, b)){
18         return `${divisor} is a common divisor of ${a} and ${b}`;
19     }else{
20         return `${divisor} is not a common divisor of ${a} and ${b}`;
```

```
21      }
22 }
```

Esse monolito foi utilizado para os testes com *Remote Procedure Call(RPC)*, *Direct Exchange*, *Fanout Echange* e *Topic Exchange*. Como um resultado adicional, além de manter o comportamento, cada alternativa de comunicação traz características próprias e/ou benefícios adicionais.

4.1.1 Remote Procedure Call

Figura 11 – Nós de comunicação RPC



Neste sistema, cada função reside em um nó diferente e a comunicação entre elas ocorre via um modelo estilo RPC (Remote Procedure Call) utilizando *RabbitMQ* representada pela Figura 11. A comunicação começa com o cliente criando uma fila de retorno exclusiva determinada no arquivo *config.yml* 4.2 pelo parâmetro *callback_queue* ao iniciar. Para uma solicitação RPC, o cliente envia uma mensagem com duas propriedades: *reply_to*, que é definida como a fila de retorno, e *correlation_id*, que é definida como um valor único para cada solicitação.

A solicitação é enviada para uma fila chamada *rpc_queue*, onde o servidor RPC aguarda solicitações. Quando uma solicitação aparece, o servidor faz o trabalho e envia uma mensagem com o resultado de volta para o cliente, usando a fila do campo *reply_to*. Isso garante que a resposta seja enviada para o cliente correto.

O cliente, por sua vez, aguarda os dados na fila de retorno. Quando uma mensagem aparece, ele verifica a propriedade *correlation_id*. Se coincidir com o valor da solicitação,

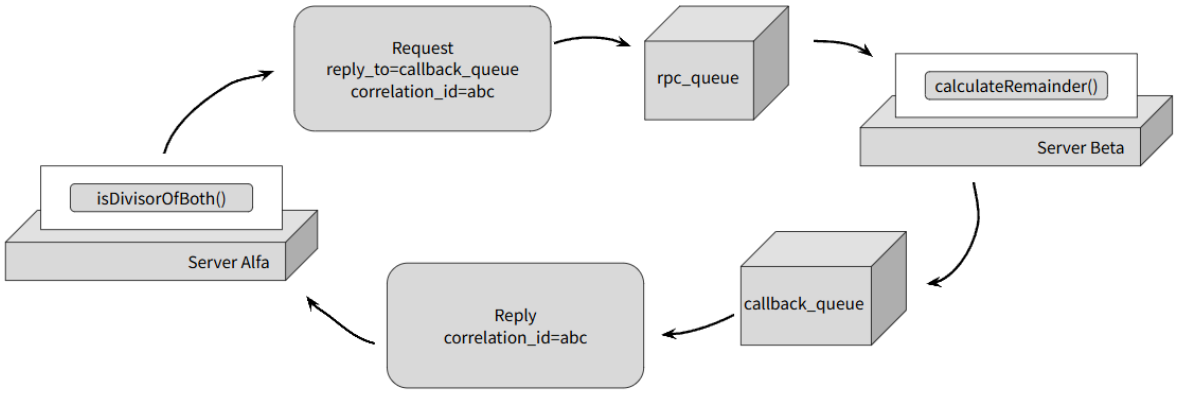
ele retorna a resposta para a aplicação. Se não houver correspondência, a mensagem pode ser descartada com segurança, evitando processamentos indevidos.

Uma das grandes vantagens de usar o padrão *Remote Procedure Call (RPC)* é a possibilidade de executar funções em um computador remoto e esperar pela resposta, como se a execução fosse local. Esse modelo simplifica a comunicação entre sistemas distribuídos, porém pode gerar confusão se o programador não tiver clareza sobre quando uma chamada de função é local ou quando é uma chamada remota, que pode ser mais lenta.

Listing 4.2 – Exemplo config.yml para RPC

```
functions:
  - name: calculateRemainder
    server: alfa
    method: rabbit
    queue: calculateRemainder_call_queue
    callback_queue: calculateRemainder_callback_queue
  - name: isDivisor
    server: beta
    method: rabbit
    queue: isDivisor_call_queue
    callback_queue: isDivisor_callback_queue
  - name: isDivisorOfBoth
    server: gama
    method: rabbit
    queue: isDivisorOfBoth_call_queue
    callback_queue: isDivisorOfBoth_callback_queue
  - name: checkCommonDivisor
    server: delta
    method: rabbit
    queue: checkCommonDivisor_call_queue
    callback_queue: checkCommonDivisor_callback_queue
```

Figura 12 – Comunicação entre dois nós



4.1.2 Direct Exchange

Essa configuração permite enviar mensagens para filas específicas com base na chave de roteamento, o que proporciona um controle preciso sobre onde as mensagens serão entregues. *Direct Exchange* é ideal para o roteamento *unicast* de mensagens, no entanto, essa abordagem pode se tornar limitada em cenários mais complexos, onde um roteamento mais flexível é necessário, pois a dependência da chave de roteamento exige uma configuração mais rígida entre produtor e consumidor.

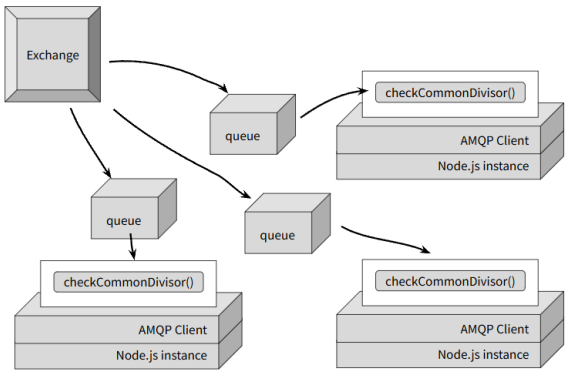
Listing 4.3 – Exemplo config.yml para Direct Exchange

```
functions:
  - name: calculateRemainder
    server: alfa
    method: rabbit
    exchange_name: alfa_exchange
    exchange_type: direct
    routing_key: calculateRemainder_key
  - name: isDivisor
    server: beta
    method: rabbit
    exchange_name: beta_exchange
    exchange_type: direct
    routing_key: isDivisor_key
  - name: isDivisorOfBoth
    server: gama
    method: rabbit
    exchange_name: gama_exchange
    exchange_type: direct
    routing_key: isDivisorOfBoth_key
  - name: checkCommonDivisor
    server: delta
    method: rabbit
    exchange_name: delta_exchange
    exchange_type: direct
    routing_key: checkCommonDivisor_key
```

4.1.3 Fanout Echange

O padrão *Fanout Exchange* permite enviar uma cópia da mensagem para todas as filas vinculadas a ele, sendo ideal para o roteamento de mensagens em modo *broadcast*, onde a mesma mensagem precisa ser distribuída para múltiplos consumidores simultaneamente. Esse modelo é bastante útil em cenários como atualizações em tempo real, como em sites de notícias esportivas ou sistemas distribuídos que precisam de atualizações de estado e configuração. No entanto, a principal desvantagem é a ineficiência quando nem to-

Figura 13 – Implementação com Fanout



das as filas precisam receber a mesma mensagem, já que essa *exchange* envia a mensagem para todas as filas sem discriminação.

Na Figura 13a *Fanout Exchange* foi utilizada para estabelecer a comunicação dos nós da função *checkCommonDivisor* replicados para simular um cenário onde vários consumidores esperam pela mensagem de resposta. Nessa situação uma chamada de qualquer uma dessas funções retornaria uma resposta para todas se comportando como um *log*.

Listing 4.4 – Exemplo config.yml para Fanout Exchange

```
1 functions:
2   - name: checkCommonDivisor
3     server: delta
4     method: rabbit
5     exchange_name: fanout_exchange
```

4.1.4 Topic Exchange

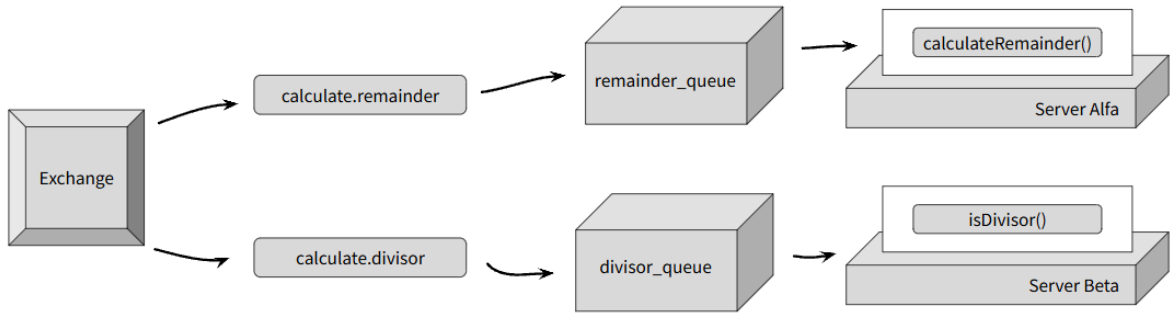
Essa implementação proporciona uma maior flexibilidade no roteamento de mensagens, permitindo que elas sejam enviadas para uma ou mais filas com base em uma chave de roteamento que combina com um padrão específico. Nesse modelo, diferentes consumidores podem selecionar de forma seletiva os tipos de mensagens que desejam receber, o que o torna perfeito para implementar um processamento de tarefas em segundo plano por múltiplos trabalhadores, ou atualização de preços de ações. No entanto, essa flexibilidade pode gerar complexidade na configuração e no gerenciamento dos padrões de roteamento, principalmente em sistemas grandes, com muitos tópicos e consumidores.

Na Figura 14 a *Topic Exchange* foi utilizada para configurar as filas das funções *calculateRemainder()* e *isDivisor()* a partir do padrão de roteamento *calculate.remainder* ou *calculate.divisor*, isso permite a chamada apenas alterando a chave padrão e até enviar para as duas filas simultaneamente a partir da chave `calculate.*`, que usa um caractere coringa para essa finalidade.

Listing 4.5 – Exemplo config.yml para Topic Exchange

```
functions:
  - name: calculateRemainder
    server: alfa
    method: rabbit
    exchange_name: topic_exchange
    exchange_type: topic
    routing_key: calculate.remainder
  - name: isDivisor
    server: beta
    method: rabbit
    exchange_name: topic_exchange
    exchange_type: topic
    routing_key: calculate.divisor
```

Figura 14 – Implementação com Topic



4.2 Refatoração manual de funções com callbacks

Considerando um cenário no qual se deseja distribuir o código 4.6 a seguir em dois microsserviços, cada um responsável por uma função. Esse exemplo mostra a função *greet*, que recebe dois parâmetros: *name* e *callback*. Quando chamada, a função imprime uma saudação no console e, em seguida, executa o *callback*, que é a função *sayGoodbye*.

Listing 4.6 – Exemplo de função passada como parâmetro

```
function greet(name, callback) {
  console.log(`Hello, ${name}!`);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}
```

Embora o uso de *callbacks* funcione bem em sistemas pequenos, ele pode gerar problemas quando o sistema é distribuído. O principal desafio está no fato de que, ao transformar uma função em um microsserviço, o *callback* não pode ser facilmente enviado pela rede. Isso ocorre porque o *callback* depende do contexto de execução da função original, e esse contexto (que inclui variáveis e o estado do sistema no momento da execução) não é simples de transmitir por meio da rede. Para que o *callback* funcione corretamente em um serviço remoto, seria necessário transmitir todo o ambiente de execução, o que pode resultar em grande complexidade e, em muitos casos, pode ser impossível garantir a consistência entre o contexto original e o transmitido.

Para resolver esse problema, a abordagem usa um sistema de filas de mensagem para desacoplar os processos. Em vez de passar o *callback* diretamente entre os serviços, podemos enviar mensagens entre os nós, o que permite que os serviços se comuniquem de forma desacoplada. Isso elimina a necessidade de compartilhar o contexto de execução entre os serviços e facilita a flexibilidade do sistema. Dessa forma, diferentes microsserviços podem operar em nós distintos sem depender diretamente uns dos outros, tornando o sistema mais fácil de manter e escalar.

Essa refatoração manual está descrita no apêndice A e possui 2 etapas, cada uma com 4 passos, a primeira consiste em determinar o nó do servidor bem como seu comportamento e a segunda tem como foco o nó do cliente. Nos testes feitos, verificou-se que é possível distribuir código com *callback* em diferentes nós, usando mensagens para manter o mesmo comportamento do monolito. A função utilizada para os testes é um fragmento selecionado do projeto usado nos estudos por Abdullah, Iqbal e Erradi (2019): trata-se de um sistema chamado AcmeAir (2018), que consiste em uma aplicação web de *benchmark open source* para uma companhia aérea fictícia, disponível nas implementações monolítica e de microsserviços.

A função original pode ser vista na Listagem 4.7. Nota-se que a função *cancelBooking* remove uma reserva do banco de dados usando o ID da reserva *(bookingid)* e o ID do usuário *(userid)*. Ela executa uma operação de remoção e chama uma função de *callback* que, se houver erro, imprime *"status: error"*, caso contrário, imprime *"status: success"*.

Listing 4.7 – Acmeair cancelBooking

```
1 function cancelBooking(bookingid, userid, callback /*(error)*/) {
2     dataaccess.remove(module.dbNames.bookingName,{'_id':bookingid, '
    customerId':userid}, callback)
3 }
4
5 const number = 1111;
6 const userid = 1001;
7
8 cancelBooking(number, userid, function (error) {
```

```
9     if (error) {
10        console.log('status : error');
11    }
12    else {
13        console.log('status : success');
14    }
15 });
```

A refatoração desenvolvida foi aplicada, e o resultado pode ser visto nas Listagens 4.8 e 4.9. Nessa implementação, a lógica da função principal e da função de *callback* foi separada em dois nós distintos, dessa forma o *callback* não é mais executado diretamente. Em vez disso, a função principal se comunica com o nó que contém a lógica do *callback* através de um sistema de filas. A comunicação começa no nó do cliente 4.9 que possui a lógica do *callback* e a chamada da função principal com seus respectivos parâmetros, que é enviada para o nó do servidor 4.8 através da fila *cancelBooking_call_queue*. No nó do servidor 4.8 está contida a lógica da função principal que recebe mensagens pela fila *cancelBookingQueue*. Os parâmetros recebidos são passados para a função *cancelBooking* que é executada e retorna uma resposta para a fila contida em *callbackQueueName*. O nó do cliente 4.9 por sua vez consome essa resposta na fila *callbackQueueName* e chama a função de *callback* executando sua respectiva lógica.

Listing 4.8 – Server Node

```
1 import amqp from 'amqplib';
2
3 export async function cancelBooking_server() {
4
5     const connection = await amqp.connect('amqp://localhost');
6     const channel = await connection.createChannel();
7
8     const cancelBookingQueue = 'cancelBooking_call_queue';
9     await channel.assertQueue(cancelBookingQueue, { durable: false });
10
11    channel.consume(cancelBookingQueue, async (msg) => {
12        const params = JSON.parse(msg.content.toString());
13        const callbackQueueName = params.callbackQueueName
14
15        const bookingid = params.bookingid
16        const userid = params.userid
17
18        cancelBooking(bookingid, userid, (error) => {
19            const msgContent = {
20                error: error
21            }
22
23            channel.sendToQueue(
```

```
24                 callbackQueueName,
25                 Buffer.from(JSON.stringify(msgContent))
26             );
27         })
28     }, { noAck: true });
29 }
30
31 function cancelBooking(bookingid, userid, callback /*(error)*/) {
32     dataaccess.remove(module.dbNames.bookingName,{'_id':bookingid, '
       customerId':userid}, callback)
33 }
34
35 cancelBooking_server();
```

Listing 4.9 – Client Node

```
1 import amqp from 'amqplib';
2
3 export async function cancelBooking_client(number, userid, callback) {
4
5     const connection = await amqp.connect('amqp://localhost');
6     const channel = await connection.createChannel();
7
8     const callbackQueue = 'cancelBooking_callback_queue';
9     await channel.assertQueue(callbackQueue, { durable: false });
10
11    const msgContent = {
12        number: number,
13        userid: userid,
14        callbackQueueName: callbackQueue
15    }
16
17    channel.consume(callbackQueue, async (msg) => {
18        const params = JSON.parse(msg.content.toString());
19        const error = params.error
20
21        callback(error)
22
23    }, { noAck: true });
24
25    channel.sendToQueue(
26        'cancelBooking_call_queue',
27        Buffer.from(JSON.stringify(msgContent))
28    );
29
30 }
31
32 function main() {
```

```
33      const number = 1;
34      const userid = 123;
35
36      cancelBooking_client(number, userid, function (error) {
37          if (error) {
38              console.log('status : error');
39          }
40          else {
41              console.log('status : success');
42          }
43      });
44  }
45
46  main();
```

Conforme comentado na seção anterior, a refatoração não foi implementada na `js-distributor`, porém como a refatoração está descrita de maneira sistemática, espera-se que o esforço de implementação será mínimo.

5 Conclusão

Com base nos resultados obtidos ao longo deste estudo, foi possível alcançar com sucesso o objetivo estabelecido no desenvolvimento deste Trabalho de Conclusão de Curso. As melhorias implementadas diretamente na ferramenta facilitaram a transformação do código em sistemas distribuídos de maneira autônoma, evitando os problemas que essa migração geralmente proporciona.

O aperfeiçoamento do método RPC ao adicionar de um *correlation_id* e fila de resposta *reply_to* tornam o tratamento das mensagens mais seguro. A inclusão de *exchanges* no encaminhamento de mensagens amplia os casos de uso da ferramenta, permitindo diferentes tipos de transmissão, *unicast*, *multicast* e *broadcast* entre produtores e consumidores.

Adicionalmente, as modificações no arquivo de configurações *YAML* permitiram a utilização de parâmetros dinamicamente tipados em consonância com a linguagem *javascript*. Além disso, o estudo de refatoração de *callbacks* se mostrou muito útil para funções que não podem ser distribuídas de forma automática no estado da ferramenta atual.

Em suma, a aplicação bem-sucedida dessas melhorias permite uma maior abrangência de cenários e casos de teste, possibilitando uma melhor adoção da ferramenta `js-distributor` para os mais variados usos. Portanto, essa pesquisa não apenas atingiu os objetivos declarados como também oferece uma contribuição valiosa para a distribuição de códigos monolitos em microsserviços.

5.1 Trabalhos futuros

Embora a melhoria da ferramenta tenha alcançado resultados satisfatórios, algumas limitações foram identificadas durante os testes com funções *callbacks*. O compilador na sua forma atual não está preparado para lidar de forma automática com a distribuição de funções que recebem *callbacks* em seu parâmetro. Apesar dos esforços e estudos empregados em refatorações manuais, uma implementação direta no gerador de funções da ferramenta poderia ser abordada em pesquisas futuras acerca do assunto.

Além das presentes limitações supracitadas, outro tema poderia ser amplamente abordado em um trabalho posterior. As verificações de comportamento da ferramenta realizadas na pesquisa não englobam sistemas de grande complexidade e testes de carga que podem estressar a aplicação. Um olhar voltado para esse tipo de *benchamrk* é passível de uma proposta de implementação futura, relacionando a distribuição de monolitos em microsserviços pela ferramenta `js-distributor` com testes de carga em um sistema real.

Referências

ABDULLAH, M.; IQBAL, W.; ERRADI, A. Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software*, v. 151, p. 243–257, 2019. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121219300408. Citado 2 vezes nas páginas 23 e 33.

ABGAZ, Y. et al. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 49, n. 8, p. 4213–4242, ago. 2023. ISSN 0098-5589. Disponível em: https://doi.org/10.1109/TSE.2023.3287297. Citado 2 vezes nas páginas 17 e 23.

ACMEAIR. *Acme Air*. 2018. Https://github.com/acmeair/acmeair-nodejs/tree/master. Acessado em: 8 fev. 2025. Citado na página 33.

BREWER, E. Cap twelve years later: How the "rules"have changed. *Computer*, v. 45, n. 2, p. 23–29, 2012. Citado na página 17.

CARVALHO, L. R. d.; ARAÚJO, A. P. F. d. Framework node2faas: Automatic nodejs application converter for function as a service. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2019. (CLOSER 2019), p. 271–278. ISBN 9789897583650. Disponível em: https://doi.org/10.5220/0007677902710278. Citado na página 24.

ESPERANÇA, V. N.; LUCRÉDIO, D. Late decomposition of applications into services through model-driven engineering. In: *Proceedings of the XXXI Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (SBES '17), p. 164–173. ISBN 9781450353267. Disponível em: https://doi.org/10.1145/3131151.3131165. Citado 2 vezes nas páginas 23 e 24.

FOWLER, M. Microservices. 2014. Acessado em: 8 fev. 2025. Disponível em: https://martinfowler.com/articles/microservices.html. Citado na página 10.

KALSKE, M.; MÄKITALO, N.; MIKKONEN, T. Challenges when moving from monolith to microservice architecture. Springer International Publishing, Cham, p. 32–47, 2018. Citado na página 10.

KAPLUNOVICH, A. Tolambda: automatic path to serverless architectures. In: *Proceedings of the 3rd International Workshop on Refactoring*. IEEE Press, 2019. (IWOR '19), p. 1–8. Disponível em: https://doi.org/10.1109/IWoR.2019.00008. Citado na página 23.

KUMAR, A. R. S. Rest vs. messaging for microservices. *Informatics - An eGovernance Publication from national Informatics Centre*, National Informatics Centre, Ministry of Electronics & IT, Government of India, 379, A4B4, Floor-3, NIC, A-Block, CGO Complex, Lodhi Road, New Delhi-110003, India, 2023. ISSN 0971-3409. Disponível em: https://informatics.nic.in/uploads/pdfs/19c88a8a_informatics_july_2023.pdf. Citado na página 10.

PARR, T.; HARWELL, S.; FISHER, K. Adaptive ll(*) parsing: the power of dynamic analysis. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 49, n. 10, p. 579–598, oct 2014. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2714064.2660202. Citado na página 20.

Apêndices

APÊNDICE A – Refatoração de funções passadas como parâmetros para execução distribuída

Para a refatoração a seguir será utilizado o seguinte exemplo A.1:

Listing A.1 – Exemplo com callback

```
1 function mainFunction(param1, callbackFunction) {
2     console.log('Performing operation...');
3
4     callbackFunction(`Operation complete with ${param1}`);
5 }
6
7 let param1 = 'param_example';
8 let user = 'Client'
9
10 function callback(result){
11     console.log(`${user} says: result = ` + result)
12 }
13
14 mainFunction(param1, callback);
```

Para começar a refatoração é necessário criar dois nós. Um *Client* e um *Server* como na Figura 15.

Um *Client* ficará responsável por iniciar a chamada e incluir a lógica da função de *callback* e um *Server* por conter a lógica e a definição da função principal Figura 16.

Figura 15 – Cliente e Servidor

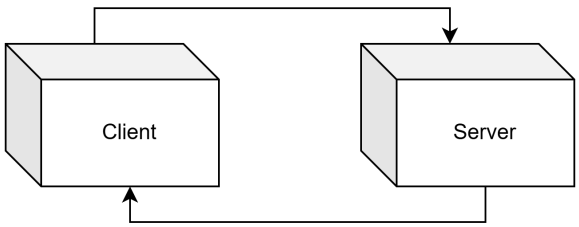
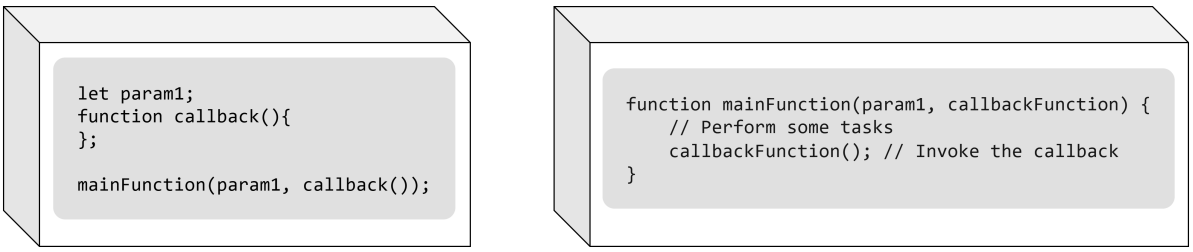


Figura 16 – Logica do Cliente e Servidor



A.1 Server Node

No nó *server* primeiro adicionamos a função principal exatamente como ela foi definida A.2.

Listing A.2 – Passo 1

```
1 function mainFunction(param1, callbackFunction) {
2     console.log('Performing operation...');
3
4     callbackFunction(`Operation complete with ${param1}`);
5 }
```

Agora é necessário definir a comunicação do *server* através da função *mainFunction_server()*. Para isso é necessário importar o protocolo *AMQP* e criar a *connection* e *channel* A.3.

Listing A.3 – Passo 2

```
1 import amqp from 'amqplib';
2
3 export async function mainFunction_server() {
4
5     const connection = await amqp.connect('amqp://localhost');
6     const channel = await connection.createChannel();
7 }
8
9 mainFunction_server();
```

Dentro da função *mainFunction_server()*, definimos a fila do *Server* e consumimos a mensagem passada pelo cliente e seus respectivos parâmetros A.4.

Listing A.4 – Passo 3

```
1 ...
2
3     const mainFunctionQueue = 'mainFunction_call_queue';
4     await channel.assertQueue(mainFunctionQueue, { durable: false });
5
6     channel.consume(mainFunctionQueue, async (msg) => {
```

```
7         const params = JSON.parse(msg.content.toString());
8         const callbackQueueName = params.callbackQueueName;
9         const param1 = params.param1;
10
11    }, { noAck: true });
12
13 ...
```

Por fim dentro do método *channel.consume*, utilizamos os parâmetros recebidos pelo *server* e passamos para a chamada da função *mainFunction()* definida previamente para que seja executada a lógica da função principal. Posteriormente enviamos a resposta para o nó do cliente que está esperando na fila contida em *callbackQueueName* A.5.

Listing A.5 – Passo 4

```
1 ...
2
3         mainFunction(param1, (result) => {
4             const msgContent = {
5                 result: result
6             }
7
8             channel.sendToQueue(
9                 callbackQueueName,
10                Buffer.from(JSON.stringify(msgContent))
11            );
12        })
13 ...
```

O código completo do nó do *server* é descrito a seguir A.6:

Listing A.6 – Código completo do servidor

```
1 import amqp from 'amqplib';
2
3 function mainFunction(param1, callbackFunction) {
4     console.log('Performing operation...');
5
6     callbackFunction(`Operation complete with ${param1}`);
7 }
8
9 export async function mainFunction_server() {
10
11    const connection = await amqp.connect('amqp://localhost');
12    const channel = await connection.createChannel();
13
14    const mainFunctionQueue = 'mainFunction_call_queue';
15    await channel.assertQueue(mainFunctionQueue, { durable: false });
16
```

```
17      channel.consume(mainFunctionQueue, async (msg) => {
18          const params = JSON.parse(msg.content.toString());
19          const callbackQueueName = params.callbackQueueName;
20          const param1 = params.param1;
21
22          mainFunction(param1, (result) => {
23              const msgContent = {
24                  result: result
25              }
26
27              channel.sendToQueue(
28                  callbackQueueName,
29                  Buffer.from(JSON.stringify(msgContent))
30              );
31          })
32
33
34      }, { noAck: true });
35
36  }
37
38  mainFunction_server();
```

A.2 Client Node

No nó *client* começamos por definir uma função *main()* e dentro dela a chamada da função principal (nesse exemplo chamaremos de *mainFunction_client)* com seus respectivos argumentos: *param1* e a função de *callback* A.7.

Listing A.7 – Passo 1

```
1  function main(){
2      let param1 = 'param_example';
3      let user = 'Client'
4
5      function callback(result){
6          console.log(`${user} says: result = ` + result)
7      }
8
9      mainFunction_client(param1, callback);
10 }
11
12 main();
```

Nessa etapa, ainda no nó cliente vamos definir a função *mainFunction_client()* e sua comunicação com o nó servidor. Para isso utilizaremos o *RabbitMQ*, importando o protocolo

AMQP e criando uma *connection* e um *channel* A.8.

Listing A.8 – Passo 2

```
import amqp from 'amqplib';

export async function mainFunction_client(param1, callback) {

    const connection = await amqp.connect('amqp://localhost');
    const channel = await connection.createChannel();

}

...
```

Para realizar a troca de mensagens é necessário ciar uma fila de retorno *mainFunction_callback_queue* e passar essa informação junto com o parâmetro *param1* para o *Server* que vai estar esperando a informação na fila *mainFunction_call_queue*. Com esses dados, enviamos tudo através do método *sendToQueue* A.9.

Listing A.9 – Passo 3

```
...

    const callbackQueue = 'mainFunction_callback_queue';
    await channel.assertQueue(callbackQueue, { durable: false });

    const msgContent = {
        param1: param1,
        callbackQueueName: callbackQueue
    }

    channel.sendToQueue(
        'mainFunction_call_queue',
        Buffer.from(JSON.stringify(msgContent))
    );

...
```

Por fim precisamos consumir a resposta do *server* que vai ser enviada na fila de *callback mainFunction_callback_queue* e executar a lógica do *callback* A.10.

Listing A.10 – Passo 4

```
...

    channel.consume(callbackQueue, async (msg) => {
        const params = JSON.parse(msg.content.toString());
        const result = params.result
```

```
6
7           callback(result)
8
9       }, { noAck: true });
10
11  ...
```

O código completo do *client* está representado a seguir A.11:

Listing A.11 – Código completo do cliente

```
1  import amqp from 'amqplib';
2
3  export async function mainFunction_client(param1, callback) {
4
5      const connection = await amqp.connect('amqp://localhost');
6      const channel = await connection.createChannel();
7
8      const callbackQueue = 'mainFunction_callback_queue';
9      await channel.assertQueue(callbackQueue, { durable: false });
10
11     const msgContent = {
12         param1: param1,
13         callbackQueueName: callbackQueue
14     }
15
16     channel.consume(callbackQueue, async (msg) => {
17         const params = JSON.parse(msg.content.toString());
18         const result = params.result
19
20         callback(result)
21
22     }, { noAck: true });
23
24     channel.sendToQueue(
25         'mainFunction_call_queue',
26         Buffer.from(JSON.stringify(msgContent))
27     );
28
29 }
30
31
32 function main(){
33     let param1 = 'param_example';
34     let user = 'Client'
35
36     function callback(result){
37         console.log(`${user} says: result = ` + result)
```

```
38      }
39
40      mainFunction_client(param1, callback);
41  }
42
43  main();
```