Guilherme Almeida Ferreira

Desenvolvimento de um Servidor de Aplicações com Arquitetura de Microsserviços Para Laboratório de Análise de Sinais

Viçosa, MG

2023

Guilherme Almeida Ferreira

Desenvolvimento de um Servidor de Aplicações com Arquitetura de Microsserviços Para Laboratório de Análise de Sinais

Monografia apresentada ao Departamento de Engenharia Elétrica do Centro de Ciências Exatas e Tecnológicas da Universidade Federal de Viçosa, para a obtenção dos créditos da disciplina ELT 402 – Projeto de Engenharia II – e cumprimento do requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Orientador Vitor Barbosa Carlos de Souza

Viçosa, MG

2023

Guilherme Almeida Ferreira

Desenvolvimento de um Servidor de Aplicações com Arquitetura de Microsserviços Para Laboratório de Análise de Sinais

Monografia apresentada ao Departamento de Engenharia Elétrica do Centro de Ciências Exatas e Tecnológicas da Universidade Federal de Viçosa, para a obtenção dos créditos da disciplina ELT 402 – Projeto de Engenharia II – e cumprimento do requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Trabalho aprovado em 14 de Julho de 2023.

COMISSÃO EXAMINADORA

Vitor Barbosa Carlos de Souza
Orientador

Carlos de Castro Goulart
Membro Avaliador

Rodolpho Neves
Membro Avaliador

Leonardo Bonato
Membro Avaliador

Viçosa, MG
2023

Agradecimentos

O presente trabalho foi realizado com apoio da Fundação de Amparo à pesquisa do estado de Minas Gerais - FAPEMIG. Além disso, Os agradecimentos principais são direcionados ao professor Vitor Barbosa Souza, pela orientação durante a produção deste trabalho; aos Professores Rodolpho Neves e Leonardo Bonato, líderes do Núcleo Interdisciplinar de Análise de Sinais (NIAS), por darem a oportunidade de desenvolver este sistema para uma aplicação real; e da equipe NIAS-IA, a qual forneceu apoio para o aprendizado das técnicas aplicadas.

Resumo

O processamento de dados vem sendo uma das principais ferramentas dentro dos ambientes de pesquisa nos dias atuais. Para isso, muitas vezes é necessário que existam sistemas capazes de processar sinais com algoritmos avançados, o que exige uma utilização elevada de recursos computacionais. Com objetivo de demonstrar uma solução para esta questão, este trabalho apresenta a elaboração de uma prova de conceito de um sistema capaz de fornecer essa quantidade de recursos para pesquisadores do Núcleo Interdisciplinar de Análise de Sinais (NIAS), através de uma interface de usuário que da acesso a um servidor de aplicações construído utilizando arquitetura de microsserviços. Este sistema foi idealizado contendo quatro principais camadas: o *front-end*, que permite aos usuários as ações de *upload* de arquivos e recebimento de resultados das tarefas enviadas; o *back-end*, que tem o papel de realizar a comunicação entre a interface e as camadas internas do servidor; a camada de processamento, com a função de executar as tarefas enviadas pelo usuário; e, por fim, a camada de armazenamento, que tem a responsabilidade de armazenar os resultados dos projetos enviados pelos usuários. Para a implementação da arquitetura do sistema, foram usados containers como meio de virtualização, além de ferramentas de orientação a evento, como um sistema de filas de mensagem para coordenação de seus componentes.

Palavras-chaves: Servidor de aplicações; Microsserviços; Docker Containers; Processamento de dados; Sistema de filas de mensagem.

Lista de figuras

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CPS	*Cyber-Physical Systems*
DRAPS	*Dynamic and Resource-aware placement Scheme*
EBA	*Event-Based Architecture*
HPC	*High Performance Computer*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IoT	*Internet-of-Things*
NIAS	Núcleo Interdisciplinar de Análise de Sinais

Sumário

1 Introdução

Atualmente, grande parte das pesquisas relacionadas a análise de sinais tem a preocupação de como esses dados serão tratados, já que, com a evolução desses estudos, cada vez mais dados são gerados, agravando ainda mais o impacto negativo percebido no aumento de complexidade dos algoritmos de processamento de modelos atuais. Estes fatores contribuem para um aumento cada vez maior da necessidade por recursos computacionais avançados, o que induz a necessidade de elaborações de sistemas que permitam aos pesquisadores acesso a estes recursos.

Com este propósito, um dos modelos mais comuns de sistemas utilizados atualmente são os *High Performance Computers* (HPC), que, segundo M. Netto et al. (2018), são clusters que podem lidar com um alto volume de dados sendo processados por códigos de alta complexidade. Para isso, estes sistemas são administrados por orquestradores que usam filas para armazenar *Jobs* de usuários quando há um uso elevado dos recursos do servidor. Dessa forma, as dificuldades na execução de *Jobs* apresentando elevado custo computacional é amenizado.

Para desenvolver um tipo de servidor com este objetivo, a primeira dúvida que deve ser respondida é, o sistema será hospedado em nuvem (cloud) ou localmente (on-premise). Para responder a essa pergunta é necessário ter em mente quais as vantagens e desvantagens de cada tipo de implementação.

1.1 Servidores em Nuvem vs Locais

Iniciando pelos servidores em cloud, a partir de Tkachenko, Tkachenko e Tkachenko (2020) pode-se identificar que seus principais requisitos são: a escalabilidade, já que neste tipo de servidor é possível aumentar ou diminuir os recursos utilizados de acordo com a necessidade de quem demanda este serviço; a disponibilidade, pois os servidores em cloud possuem uma garantia maior de que seus serviços estarão com elevados índices de tempo em atividade; o backup e recuperação de desastres, possíveis a partir de mecanismos nativos de backup automático de dados, reduzindo o risco de perdas.

Por outro lado, estes sistemas também possuem algumas desvantagens, como: a necessidade de acesso constante e confiável à internet, levando a falhas no funcionamento quando isso não ocorre; o controle limitado, já que ao utilizar serviços terceirizados, os dados não são controlados diretamente, podendo resultar em um menor controle de privacidade e segurança; dificuldades em fixar um orçamento inicial, pois estes sistemas são pagos conforme a demanda dificultando a previsão de investimento ao iniciar um projeto

de implementação.

Os servidores on-premise também possuem alguns inconvenientes, exemplificados por: um alto investimento inicial, com gastos para adquirir tanto hardware quanto software, além de demais itens de infraestrutura; limitações na escalabilidade, uma vez que, para aumentar sua capacidade, além da aquisição de mais hardware, é necessário um esforço maior na implementação; responsabilidade de manutenção, exigindo uma equipe dedicada para o suporte das atividades do servidor, além da atualização e backup também serem de incumbência de quem implementa este tipo de sistema.

Contudo, como descrito por Ruíz et al. (2022), os servidores on-premise levam vantagem em alguns aspectos em relação à utilização de nuvem, estes são: o controle dos dados que trafegam no serviço, garantindo mais privacidade e segurança; a menor latência, melhorando o desempenho de aplicações que necessitam acesso rápido e contínuo a grandes quantidades de dados; maior facilidade em estabelecer um orçamento de projeto, permitindo um planejamento mais claro, o que viabiliza a solicitação de recursos financeiros.

1.2 Motivação para criação do Servidor

O Núcleo Interdisciplinar de Análise de Sinais (NIAS), localizado no Departamento de Engenharia Elétrica da Universidade Federal de Viçosa, tem o principal objetivo de desenvolver pesquisas sobretudo voltadas para o processamento e investigação de dados relacionados à neurociência. Por muitas vezes, seus pesquisadores utilizam algoritmos complexos, como exemplos podem ser citados: algoritmos genéticos, técnicas de descorrelação, estratégias de testes sequenciais para determinação de limiares de detecção de sinais, estratégias de estimação de componentes espectrais, redes neurais, lógica Fuzzy e outras técnicas de processamento para construção de modelos a partir dos dados obtidos.

Dessa forma, foi identificada a necessidade de prover aos seus membros uma possibilidade de usar maior poder computacional, fornecido por um servidor que fosse capaz de executar os códigos de processamento de dados e disponibilizar seus resultados, normalmente em forma de gráficos e tabelas.

Para isso, o sistema desenvolvido precisou atender a alguns critérios, definidos de acordo com as demandas dos membros do NIAS. Primeiramente, é essencial que os usuários possam se conectar de forma remota, além de possibilitar acessos simultâneos dos mesmos. Outro importante fator relacionado com o contato com o sistema é sua facilidade, em que idealmente, o usuário não necessite utilizar linhas de comando para a utilização do serviço, mas sim ter acesso a uma interface web que facilite o processo. O servidor também precisa ser capaz de processar códigos desenvolvidos em MATLAB e Python, já que estas são as ferramentas usadas pelos pesquisadores membros do núcleo.

Ao final da execução da tarefa enviada pelo usuário, é também obrigatório que seu resultado seja disponibilizado assim como idealizado pelo pesquisador. E por fim, o software deve ser construído de forma a possibilitar a futura escalabilidade, permitindo que outras máquinas fossem integradas ao sistema com fácil configuração, conforme seu crescimento se mostre necessário.

Para atender a esses requisitos, foi necessário escolher uma arquitetura que permitisse a privacidade e segurança da transação de informações dentro do servidor, já que esses serão dados confidenciais de pesquisas do laboratório. Além disso, o sistema deve ter uma previsibilidade quanto ao custo de realização, para tornar o projeto viável. Portanto, devido a estes motivos, o modelo de implementação escolhido foi o on-premise.

1.3 Objetivos

A partir das necessidades identificadas na sessão anterior, foram identificados os objetivos deste trabalho. Como objetivo geral, este trabalho busca desenvolver uma prova de conceito de um sistema que deve fornecer recursos computacionais para que os pesquisadores do NIAS possam obter os resultados de suas pesquisas através do processamento remoto de grande quantidade de dados de acordo com um algoritmo fornecidos pelo próprio pesquisador.

Mais especificamente, para alcançar este objetivo, o servidor deve:

- permitir que mais de um pesquisador entre no sistema de forma remota e simultânea, sem necessidade de utilização de terminais de comando;
- possibilitar o envio de códigos de processamento de dados desenvolvidos em MATLAB ou Python, assim como dos dados que serão processados de forma assíncrona;
- conceder acesso aos resultados dos códigos enviados pelos pesquisadores;
- ser escalável, de forma que seja possível adicionar mais máquinas ao sistema, para aumentar os recursos de processamento.

2 Revisão de Literatura

Neste capítulo será apresentado uma revisão dos conteúdos que embasaram a implementação do servidor, tanto no que se refere às ferramentas utilizadas, quanto nos projetos relacionados que influenciaram na sua realização.

2.1 Referencial Teórico

Para o desenvolvimento do sistema apresentado neste trabalho, foram empregadas diversas tecnologias utilizadas atualmente no desenvolvimento de softwares e servidores. A seguir, serão discutidas as principais ferramentas presentes no trabalho, além da justificativa do motivo de seu uso.

2.1.1 Virtualização

Um dos principais desafios enfrentados para a elaboração deste tipo de sistema é a otimização dos recursos disponíveis, permitindo que sejam utilizados de forma equilibrada e bem orquestrada, para evitar problemas como a utilização desbalanceada de recursos do servidor.

Uma das estratégias que é mais presentes atualmente para resolver este tipo de problema é a virtualização. Segundo Yadav, Pal e Yadav (2021), com esse método é possível alocar recursos de forma dinâmica simplificada, por meio do mapeamento de um recurso físico para uso como virtual, o que facilita o câmbio entre o uso destes recursos de acordo com a forma com que o sistema esta sendo utilizado. Além disso, a virtualização simplifica a escalabilidade, permitindo que novas máquinas sejam adicionadas ao cluster com pouca configuração.

Como esclarecido por Sharma et al. (2016), existem duas principais formas de virtualização, a de hardware e a de sistema operacional. A primeira, envolve virtualizar a parte física de um servidor e criar uma máquina virtual, a segunda virtualiza apenas o núcleo do sistema operacional, criando assim um container. A principal diferença entre eles é que as máquinas virtuais emulam todo o hardware da máquina Host, rodando seu próprio sistema operacional, enquanto os containers utilizam o sistema operacional do dispositivo onde estão alocados.

Ainda segundo Sharma, também é demonstrado que os containers tendem a ter um desempenho melhor quando utilizados em maior escala, já que não necessitam um sistema operacional próprio, os deixando mais leves. Além disso, os containers permitem uma maior flexibilidade na alocação de recursos, ao contraio das máquinas virtuais, que

tem definições mais estritas quanto a sua configuração. Outra vantagem dos containers é a sua implementação que, por meio de ferramentas como Docker, pode ser feita através de código, permitindo seu versionamento.

A maior desvantagem dos containers em relação às máquinas virtuais é seu isolamento de dados, já que, por utilizarem o mesmo sistema operacional da máquina host, podem ter uma menor segurança quando utilizadas em um servidor onde existam diferentes clusters executando simultaneamente, como em um contexto da nuvem. Essa inconveniência é suprimida no contexto deste trabalho, já que ele foi realizado em uma estratégia on-premise, como citado no capítulo 1.

2.1.2 Microsserviços Baseados em Eventos

A virtualização, a partir de containers, explicada na seção anterior, foi utilizada neste trabalho para estabelecer uma arquitetura de microsserviços. Este tipo de idealização de um sistema é muito utilizada nos dias atuais devido suas vantagens em relação a escalabilidade.

Segundo Bucchiarone et al. (2018), as aplicações monolíticas são aquelas que, necessariamente, compartilham os recurso de uma mesma máquina para todos os seus componentes. Já os microsserviços se diferenciam da arquitetura monolítica pois são construídos a partir de pequenas unidades escaláveis, que funcionam de forma isolada entre si, podendo utilizar recursos de diferentes máquinas, permitindo a adição de mais recursos ao sistema.

Uma forma muito utilizada para implementar os microsserviços é a arquitetura baseada em evento (EBA), que, como esclarecido por Laigner et al. (2020), é definida por um conjunto de componentes que respondem de forma assíncrona a um evento para realizar uma tarefa específica. Além disso, os autores esclarecem que esse tipo de arquitetura tem diversas vantagens em relação a sua implementação como: suporte a tecnologias desenvolvidas em diferentes linguagens, devido ao seu isolamento entre componentes; e resposta automática a falhas ou mudanças no sistema por meio dos eventos criados.

Outro fator importante é que a elaboração de uma arquitetura de microsserviços e baseada em eventos torna a manutenção mais simples, já que os seus componentes são isolados entre si, facilitando a identificação de falhas. Outra vantagem dessa estratégia é que, com uma alta coesão entre as unidades, se torna possível uma reação proativa a eventos que irão acontecer por meio de acionamentos de outros componentes.

2.1.3 Mensageria

Como já elucidado anteriormente, a coesão e resposta a eventos dentro do sistema desenvolvido neste trabalho é uma parte central. Para isso, uma das ferramentas mais

utilizadas atualmente é a fila de mensagens (*Messaging Queue*), que permite a integração coesa dos elementos isolados do sistema.

Este instrumento funciona a partir de três principais elementos, o produtor, que é a fonte do dado publicado, a fila, o local onde o dado é armazenado, e o consumidor, que recebe o dado transacionado. Outro fato importante a se destacar é que as mensagens são consumidas de forma ordenada, ou seja, a primeira mensagem publicada será, necessariamente, a primeira consumida, seguida pela segunda e assim por diante. Como retratado por Fu et al. (2020), existem quatro principais benefícios com o uso de filas:

1. Processamento assíncrono: Quando um sistema é síncrono, o produtor da mensagem precisa receber uma resposta do consumidor antes de enviar um novo dado. Já em um sistema de filas, o produtor pode publicar uma mensagem e seguir produzindo dados, sem precisar de uma resposta imediata do consumidor, evitando desperdício de recursos na espera de uma resposta;

2. Desacoplamento entre produtor e consumidor: Um sistema normalmente se comunica com diversos outros dentro de uma aplicação, o desacoplamento permite que o produtor apenas se preocupe em produzir os dados, sem precisar considerar as outras comunicações internas do sistema

3. Melhor funcionamento durante picos de produção de dados: Já que os consumidores não precisarão lidar com uma grande quantidade de processamento simultâneo, uma vez que irão consumir as mensagens apenas quando tiverem recursos suficientes para isso, eles não ficarão sobrecarregados

4. Serviço de Logs: Em sistemas em produção, a geração de logs pode ser muito grande, assim, as filas de mensagens são úteis para garantir que os logs gerados sejam armazenados e projetados na ordem correta.

Como será pormenorizado no capítulo 3, as principais vantagens exploradas neste trabalho a partir do uso das filas são, o processamento assíncrono, permitindo que um usuário não precise esperar pela disponibilidade do servidor para enviar seu arquivo, e o desacoplamento entre produtor e consumidor, já que a codificação de um não será afetada pela do outro, simplificando o desenvolvimento.

2.1.4 Application Programming Interface (API)

Com a modularização do sistema, torna-se evidente que a comunicação entre cada entidade isolada do servidor é extremamente importante para o funcionamento correto do sistema. Com o propósito de facilitar a transação de informações entre esses elementos, durante o desenvolvimento deste trabalho foi lançado mão de um recurso largamente utilizado atualmente para esse fim, as *Application Programming Interfaces* (APIs).

De acordo com Lamothe et al. (2021), as APIs são interfaces que podem ser usadas por diferentes clientes, oferecendo uma funcionalidade específica. Seu uso não depende do ambiente em que foi desenvolvida, já que é empacotado em as bibliotecas, *frameworks* e serviços Web que a utilizam para seu funcionamento. Seu uso favorece a elaboração de uma arquitetura de microsserviços, já que permite o isolamento e comunicação coesa dos componentes de um sistema.

Além disso, segundo Ofoeda et al. (2019), a utilidade das APIs é destacada pelo fato de que elas estabelecem um terreno comum para que aplicações e softwares possam estabelecer uma comunicação, independente da linguagem em que são desenvolvidas. Através dessa comunicação de serviços, externa ou internamente nas organizações, é possível criar valor.

2.2 Trabalhos Relacionados

A arquitetura de microsserviços, assim como a orquestração de containers para a elaboração de servidores de aplicação, tem sido temas largamente discutidos na conjuntura atual, para o desenvolvimento de formas mais eficientes de administrar e disponibilizar recursos para os mais diversos fins. Nesta seção serão apresentados trabalhos que apresentam soluções e debates em relação a estes cenários.

2.2.1 Desenvolvimento de Edge Servers

Atualmente a maior parte das aplicações na arquitetura de microsserviços é desenvolvida em sistema cloud, essas funcionalidades costumam utilizar containers como sua ferramenta de virtualização, e para a orquestração se faz uso de Softwares como Kubernetes, que é instrumento centralizado.

Um paradigma apresentado por Jimenez e Schelen (2020) é o de *Edge Servers*, que são servidores hospedados em máquinas na borda da rede, próximos ao usuário final, com a função de permitir que a transferência de dados ocorra de forma mais próxima de sua fonte. Essa ação, diminui a latência com que essa transferência ocorre, minimizando a necessidade de uma internet extremamente veloz, além de permitir uma maior segurança e privacidade dos dados e diminuir os custos com a massiva transferência de dados que pode ocorrer através de sistemas cloud.

No contexto do trabalho supracitado, a principal aplicação para a qual o servidor foi desenvolvido é em IoT (*Internet-of-Things*) e CPS (*cyber-physical systems*) que são sistemas de dispositivos capazes de conectar indústrias, cidades, produção de energia, administração de saúde, casas, além de inúmeros outros exemplos. Segundo os autores, esses cenários geram uma quantidade massiva de dados, podendo chegar a 25 exabytes

(2^{60} bytes) por mês nos dias atuais, o que justifica o desenvolvimento de sistemas locais e não só em cloud.

É proposto então, pelo artigo, uma ferramenta chamada HYDRA, que utiliza uma lógica de descentralização, em que cada nó de um cluster funciona, ao mesmo tempo, como um orquestrador e próprio recurso. Isso permite que aplicações sejam implementadas de acordo com os seus requisitos de forma flexível, integrando serviços em cloud e locais. Dessa forma, resolvendo assim alguns obstáculos dos serviços de orquestração centralizados em cloud, como a escalabilidade, disponibilidade e eficiência no uso de recursos.

Os resultados deste trabalho demonstram que uma orquestração feita não só em sistemas cloud pode ser mais eficiente para alguns contextos que demonstram um grande crescimento em uso atualmente, como os já citados IoT e CPS. Contudo, o artigo demonstrou a utilização deste servidor para a implementação de uma quantidade massiva de aplicações em um cluster de grande escala, não apresentado uma usabilidade para servidores com limitações de recursos e poucos nós.

2.2.2 Orquestração de Containers

Para a implementação correta de uma arquitetura de microsserviços utilizando containers, um dos maiores obstáculos a serem superados a é orquestração dos componentes. Neste tipo de infraestrutura, a comunicação entre cada elemento de um servidor, assim como a melhor forma alocar estes componentes para a otimização do uso de recursos, é de fundamental importância.

Neste âmbito, existem algumas ferramentas para a implementação de de lógicas de orquestração para o melhor funcionamento de um servidor como o proposto neste trabalho, algumas das mais utilizadas atualmente são, Kubernetes, Docker Swarm e Apache Mesos, que utilizam diversas estratégias para a orquestração de containers. Como será destacado a seguir, Acharya e Suthar (2022) discorrem sobre as diferentes facetas dessa orquestração.

Segundo o artigo, existem cinco principais classificações da orquestração de containers. Inicialmente, o controle de recursos tem o objetivo de encontrar o local com melhores características, em relação aos recursos, dentro de um cluster para instanciar um novo container. É proposto o esquema DRAPS (*Dynamic and Resource-aware placement Scheme*) para distribuir containers dinamicamente em um cluster, considerando o melhor uso dos diferentes recursos.

Além disso, o *Scheduling* tem o propósito de implementar serviços em nó de containers de forma a utilizar os recursos da melhor forma possível, sem subutilização ou superutilização em nenhuma parte da máquina. Para este fim, foi proposta a Teoria *Stable Matching* para encontrar a melhor forma de mapear containers para os *hosts* (máquinas hospedeiras). Isso é realizado de forma que o algoritmo de escalonamento envia a melhor

configuração de containers para as máquinas físicas de forma a reduzir o desperdício de recursos.

Muitas vezes classificado como um tipo de *Scheduling*, o *Load Balancing* é realizado com o intuito de alocar corretamente os serviços dentro dos containers existentes dentro de um cluster. Os softwares Kubernetes e Docker Swarm possuem diferentes algoritmos que alocam aplicações dentro de containers pré estabelecidos, baseados em critérios como a ocupação desses containers ou, até mesmo, fazendo a escolha de forma aleatória em relação a quais containters implementar essas aplicações.

Outras duas classificações das estratégias de orquestração são: a verificação de saúde, que é utilizado para garantir que as réplicas dos containers estejam rodando de forma adequada, além de redirecionar as requisições caso ocorra algum erro em um container específico; e o *Auto Scaling*, que serve para escalar automaticamente os recursos de um cluster em nuvem conforme a necessidade.

Ainda segundo o trabalho de Acharya e Suthar, adversidades que precisam ser tratadas na literatura desse campo são: a alta disponibilidade a qual os servidores precisam prover, a alocação mais efetiva de containers dentro dos nós de um cluster; a observabilidade a partir dos logs desses sistemas; e a segurança e privacidade dos dados transferidos no servidor. Além disso, a maior parte dos trabalhos nesse contexto são focados no desenvolvimento usando ferramentas de cloud, existindo uma lacuna em relação a sistemas on-premise em menores escalas, que dispõem de limitações de recurso, área na qual o trabalho atual pretende avançar.

3 Métodos de Implementação

Este servidor foi construído para atender aos requisitos dados a seguir: 1. Permitir que mais de um pesquisador entre no sistema de forma remota e simultânea; 2. Possibilitar o envio de códigos de processamento de dados desenvolvidos em MATLAB ou Python, assim como dos dados que serão processados; 3. Conceder acesso aos resultados dos códigos enviados pelos pesquisadores; 4. Ser escalável, de forma que seja possível adicionar mais máquinas ao sistema, para aumentar os recursos de processamento. A partir destes objetivos, foram idealizadas as soluções que serão esclarecidas neste capítulo.

3.1 Interface do usuário - Front-End

Para que os usuários possam entrar em contato com o servidor, é necessária a criação de uma interface que seja de fácil interação, e que, ao mesmo tempo, atenda aos requisitos dados ao servidor. Para atender ao requisito 1, o pesquisador deve ter acesso a uma página que permita que ele realize o envio de arquivos de sua máquina. Esta página deverá ser acessível através da internet a partir de qualquer sistema operacional, possibilitando o acesso remoto ao sistema.

É importante destacar que, dado o modelo Web Server cliente-servidor, o servidor é implementado a partir de um Web Server de modo que a interface desenvolvida é a única forma de interação entre o cliente e o servidor, já que é a parte do sistema que estabelece a comunicação com o usuário (*Client-Side*). No entanto, todo o processamento realizado a partir das informações coletadas por essa interface emana dos componentes confinados no lado do servidor.

Dessa forma, foi utilizada a linguagem HTML, juntamente com o protocolo de comunicação HTTP, para a criação de um site na web, permitindo tanto a exibição de uma interface amigável quanto o envio de arquivos através da rede de forma otimizada, sendo necessário ao usuário apenas o acesso a um navegador. O esquema para a criação da interface é ilustrado pela Figura 1.

Também é possível observar através do diagrama, a delimitação de um objeto importante para o sistema, denominado *Job*. Este item deverá ser um arquivo compactado no formato ".zip" que contém o código em MATLAB ou Python desenvolvido pelo pesquisador usuário do servidor, assim como os dados em que este código irá atuar. Logo, este elemento representa a mensagem que o usuário deseja enviar ao servidor, ou seja, a entrada para o sistema.

A criação de um endereço web para o servidor também permitiu atender ao requisito

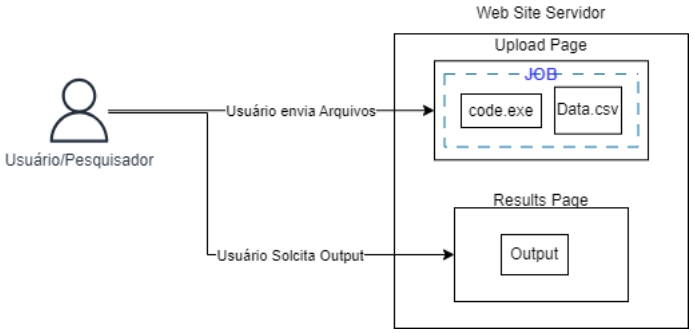


Figura 1 – Esquema para interface de usuário.

número 2, por meio da criação de uma página de resultados, onde os usuários podem entrar e requisitar ao servidor os *outputs* de seus códigos. Após feita a requisição, o sistema verifica se existem resultados disponíveis e os envia para a interface, permitindo que o usuário realize o download do arquivo. Esta segunda página também é mostrada no esquema representado pela Figura 1

Para possibilitar que o arquivo enviado chegue ao servidor da forma desejada, foi desenvolvido um back-end baseado no protocolo HTTP para a comunicação entre a interface e o servidor, permitindo a transação de informações entre essas duas partes do sistema. Os detalhes do back-end são abordados na seção a seguir.

3.2 Comunicação entre interface e servidor - Back-End

A primeira função que deve ser atribuída ao back-end é estabelecer a comunicação entre a interface do cliente e o servidor, permitindo que as mensagens enviadas pelo usuário sejam recebidas pelas camadas pertinentes do sistema. Portanto, mostrou-se necessário desenvolver uma maneira pela qual o *Job* possa ser enviado pelo usuário ao *Server-Side*, assim como o *Output* desse *Job* possa ser recebido na pelo pesquisador, através da interface, após seu processamento por parte do servidor.

Para isso, foi utilizado o framework Flask (2023) para a linguagem Python. Essa ferramenta possibilita criar rotas para permitir acesso dos usuários às páginas corretas do site do servidor, além de permitir o trafego de informações entre estes dois ambientes através do protocolo HTTP.

A partir do esquema ilustrado pela Figura 2, fica clara a função do back-end para o servidor. No diagrama, o *Job* enviado pelo usuário é encaminhado para o Back-end através do método HTTP POST, que permite a inserção de dados no *Server-Side*. Dessa forma, utilizando o Flask, é possível direcionar os arquivos de *Input* para as unidades de processamento do sistema (descritas na seção 3.4).

Para que os pesquisadores possam ter acesso aos *Outputs* de seus códigos, também

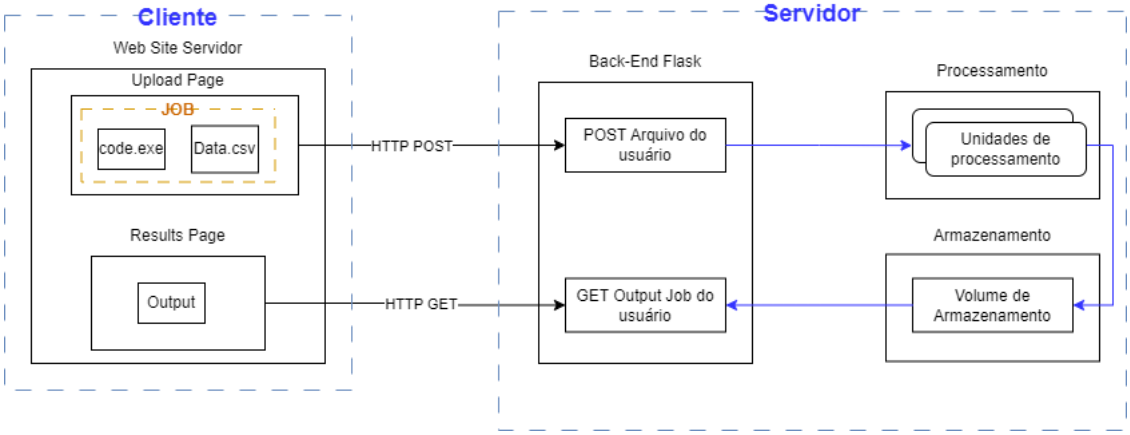


Figura 2 – Esquema para Back-end do servidor.

é necessária a utilização do mesmo tipo de comunicação, mas agora através do método HTTP GET, que permite que o cliente realize uma requisição ao servidor para consultar uma informação. O back-end, por sua vez, encaminha esta solicitação ao volume de armazenamento (apresentado em detalhes na seção 3.5), que responde com os resultados gerados pelas unidades de processamento a partir da execução dos códigos contidos nos *Jobs*.

Contudo, ainda é necessário solucionar um dos obstáculos que podem ser gerados pelo requisito 1 do servidor, o qual exige que mais de um usuário possa acessar o sistema de forma simultânea. Este problema ocorre quando toda as unidades de processamento estão ocupadas por outros *Jobs*, no momento em que um pesquisador envia um novo arquivo para ser executado pelo servidor. Quando o cenário descrito ocorre, se não tratado, um erro é retornado para o cliente da requisição. Para solucionar este empecilho, foi implementado o sistema de filas, descrito na seção 3.3.

3.3 Sistema de filas

As filas são uma ferramenta intermediária de comunicação, que tem a função de realizar a transação de mensagens entre o produtor, que envia a mensagem, e o consumidor, que a recebe. Outra característica importante deste instrumento é a possibilidade de desacoplamento temporal entre o produtor e o consumidor da mensagem, ou seja, o produtor poderá enviar conteúdo em um maior ritmo do que o consumidor consegue processá-lo. Neste caso, as mensagens são acumuladas dentro da fila, até que sua leitura se torne possível.

Para a criação deste sistema será utilizado o RabbitMQ (2023), que é um software open source que permite a criação de filas de forma simples e fácil de ser integrada com o resto dos componentes, já que é nativamente integrável com o Python, linguagem utilizada para o back-end. A denominação correta para este recurso é *Message Broker*, que nada

mais é que um instrumento para a gestão de mensagens, através da criação de filas e tópicos. No contexto do servidor, a mensagem transacionada é o *Job*, descrito na seção 3.1.

Com o RabbitMQ, o gerenciamento das mensagens ocorre através de um *Exchange*, que recebe o conteúdo do produtor e a direciona para a fila correta, como ilustrado na Figura 3. Para o sistema, inicialmente será criada apenas uma *Exchange* e uma fila, para permitir o desacoplamento entre cliente e servidor. Além disso, será necessário instanciar um produtor e um consumidor, para o envio e recebimento de mensagens, algo que será realizado utilizando o back-end.

Como já descrito na seção 3.2, o framework Flask permite a configuração do roteamento da interface de usuário. Além disso, esta ferramenta também possibilita a criação de APIs, explicadas na seção 2.1.4. Por meio deste recurso, foram criados o produtor e consumidor das filas, como ilustrado na Figura 3. O produtor será uma API que recebe o *Job* através do *back-end* do servidor, o transforma em uma mensagem e publica na *exchage* do *Message Broker*, que a designa para a fila correta. Ao final deste processo, o produtor retornará um código 200 como resultado, indicando uma requisição HTTP bem sucedida. O consumidor, por sua vez, assim que identificar a disponibilidade do sistema, receberá esta mensagem e enviará para a camada de processamento.

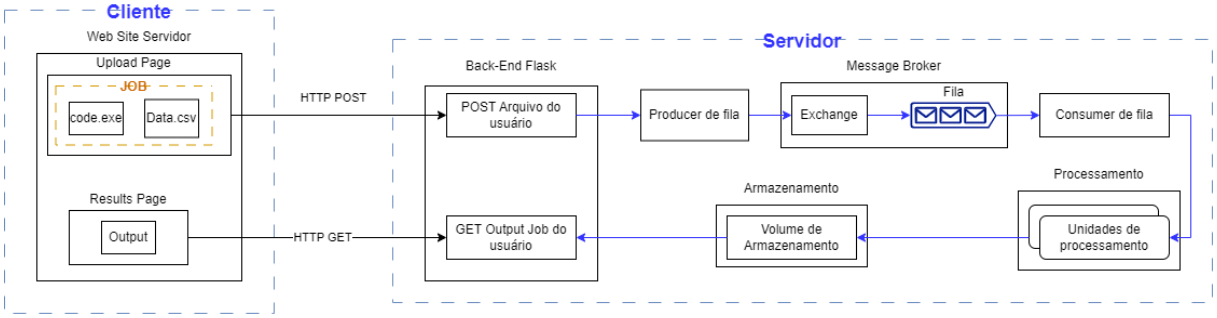


Figura 3 – Representação do sistema de filas na arquitetura do servidor.

Agora, com o problema de envio dos códigos e dados dos pesquisadores resolvido, é necessário realizar seu processamento. Para isso, deve haver uma forma de executar o script desenvolvido para o software MATLAB, aplicá-lo nos dados pertinentes e disponibilizar o *Output* para o usuário. Com o objetivo de superar esses obstáculos, foram criadas a unidades de processamento descritas na seção a seguir.

3.4 Unidades de processamento

As Unidades de processamento foram criadas com a premissa de que receberão um código de MATLAB ou Python e os dados de entrada para o mesmo, como ilustrado na Figura 3. Dessa forma, é necessário que o componente seja capaz de processar scripts desenvolvidos para estes Softwares, da maneira mais rápida e eficiente possível.

Outra dificuldade que essas unidades buscam solucionar é o monitoramento dos recursos disponíveis no servidor, já que é muito custoso realizar esta tarefa de forma eficiente utilizando apenas as ferramentas nativas do sistema operacional Linux.

Sendo assim, viu-se necessário confinar parte dos recursos computacionais disponíveis em containers. Com o uso dessa ferramenta, tornou-se possível destinar os recursos necessários para o processamento dos *Jobs*, além de facilitar o gerenciamento destes recursos de forma mais otimizada, o que é melhor descrito na seção 2.1.1.

Para isso será utilizado o Software Docker (2021), que permite a criação de containers altamente configuráveis. Esses containers devem então ser criados contendo uma instalação funcional do MATLAB ou Python. Além disso, a unidade deverá ser capaz de executar o código de forma automática assim que recebe-lo.

Com este propósito, as unidades de processamento foram desenvolvidas como APIs, capazes de receber arquivos via protocolo HTTP, o que acionará uma determinada ação. No caso desta camada, assim que o Job for enviado pelo consumidor, a unidade de processamento irá descompactar o arquivo do usuário, executar o script desenvolvido, obter o resultado, compacta-lo novamente para um arquivo ".zip" e enviar para um diretório específico dentro do próprio container. Após este processo ser realizado de forma correta, as unidades irão enviar o código 200 de resultado para os consumidores, e apenas após isso o consumidor se tornará disponível para receber uma nova mensagem. Caso algum problema aconteça, será retornado um código 500 (HTTP server error) por parte da unidade de processamento, engatilhando a ação do consumidor de retornar o Job para a fila.

Outra vantagem da utilização de containers, é que cada unidade é independente, possibilitando que mais de um *Job* seja executado ao mesmo tempo de forma paralela, aumentando a velocidade com que o servidor é capaz de processar requisições dos usuários. Dentro deste trabalho, a prova de conceito desenvolvida dispõe de duas unidades de processamento para demonstrar este paralelismo, como ilustrado pela Figura 4. Além disso, cada unidade de processamento será acoplada a um único consumidor, para que este acoplamento permita que *Jobs* sejam enviados para serem executados apenas quando existirem containers de processamento disponíveis.

Até aqui foram mostradas as vantagens da utilização de containers, incluindo o isolamento de seus processos em relação ao resto do sistema. Contudo, isso origina um problema, já que os resultados gerados pelos códigos dos pesquisadores, também ficam confinados nessas unidades após seu processamento. Para acessar esses dados, será então utilizada a ferramenta de volumes, descrita na seção a seguir.

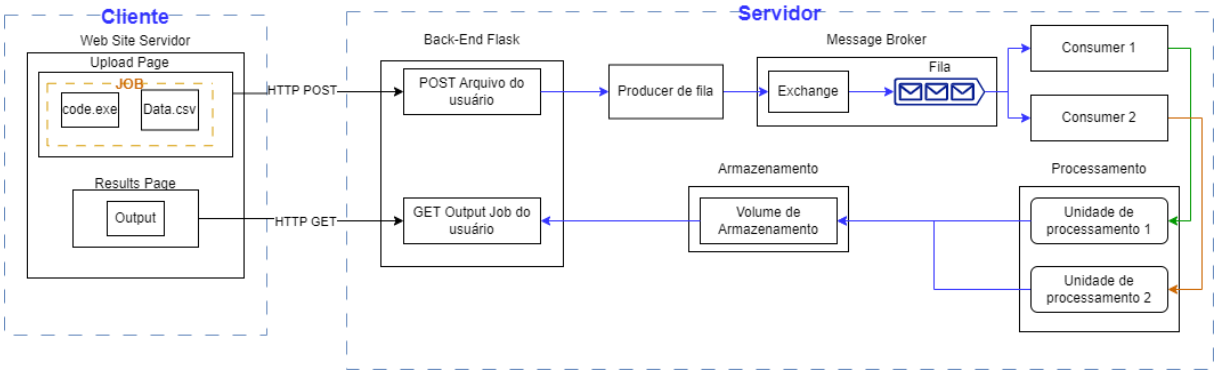


Figura 4 – Arquitetura completa do servidor.

3.5 Volume de Armazenamento

A informação do que é processado dentro de um container não pode ser acessada por sistemas externos, mesmo que estejam hospedados no próprio *host*. Contudo, para as demandas do servidor, demonstrou-se necessário o acesso a essas informações, mais especificamente, aos resultados dos códigos enviados pelos pesquisadores, ou seja, aos *Outputs*.

Para isso, foi feito o uso da ferramenta de Volumes, nativa do Docker. Esta *feature* permite conectar uma pasta interna de um container com a memória física da máquina *host*, de forma que, quando algo é criado nesse diretório do container, a informação poderá ser acessada de forma externa.

Ainda tendo como base o esquema ilustrado na Figura 4, fica claro a necessidade de estabelecer uma comunicação entre as unidades de processamento e armazenamento do servidor. Lançando mão dos volumes, se torna então possível esta comunicação, em que, ao final da execução do *Job*, seu *Output* é salvo em um diretório do container que aponta para o armazenamento físico do *host*, representado por um diretório de mesmo nome, mas externo ao container.

Uma vez finalizado este processo, o resultado fica disponível para acesso de outras partes do sistema. Ainda acompanhando o esquema da Figura 4, o próximo passo é enviar a solução para a interface do cliente, e para isso foi utilizado o sistema de Back-End (seção 3.2), onde foi desenvolvida uma requisição HTTP que acesse o volume físico do *host*, e retorne o *Output*, o que permitirá que o usuário acesse este conteúdo através do site, como ilustrado na Figura 1.

3.6 Implementação da Arquitetura

O último passo para a implementação do sistema descrito neste capítulo é a sua arquitetura. Foi necessário, portanto, definir como cada componente deve ser colocado

em funcionamento, além de configurar a comunicação interna para a transmissão das informações, como os Jobs dos usuários. Para isso, foi utilizada a ferramenta docker-compose, que facilita a definição de containers, assim como sua comunicação e armazenamento.

O software de virtualização Docker, já explicado na seção 2.1.1, isola partes do sistema contendo códigos que aplicam determinadas regras de processamento. Dessa forma, caso haja necessidade de expandir a capacidade da aplicação, devido a um eventual aumento de demanda, o processo de ampliar a capacidade de cada container, ou até mesmo de criar réplicas para os containers existentes, é facilitado.

Por este motivo, todas as partes do servidor serão implementadas utilizando esta ferramenta. Foi delimitado então um container que confinará os sistemas de back-end e front-end, outro que conterá o Message Broker, o produtor e os consumidores de mensagens também estarão isolados em diferentes containers e, por fim, mais dois que corresponderão às unidades de processamento.

Finalmente, para configurar toda a comunicação entre as camadas do sistema, foi criada uma *network* customizada, para identificar cada container por seu nome, que é definido de forma a funcionar como um identificador único dentro do sistema. Além disso, foi possível configurar quais as portas em que os containers esperam receber sua comunicação. Foi também possível, através do docker-compose, criar a unidade de armazenando descrita na seção anterior, além de identificar para quais diretórios internos dos containers o volume de armazenamento físico está apontado.

Com a utilização do docker-compose, a implementação de todo o sistema se tornou muito mais simples, sendo necessário apenas um comando para que todas as imagens docker sejam montadas, os containers sejam criados e a comunicação entre eles seja estabelecida da forma como foi idealizada durante este trabalho.

4 Resultados e Discussão

Este capítulo será dedicado à apresentação do funcionamento do servidor proposto neste trabalho. Além disso, aqui também será apresentada a discussão sobre os resultados obtidos, assim como as limitações do sistema desenvolvido.

4.1 Preparação de Job do usuário

Para validar o funcionamento do servidor, foram utilizados arquivos compactados contendo scripts Python que geram uma determinada saída. Os arquivos compactados são enviados através da interface de usuário e, posteriormente, seus resultados são disponibilizados através da pagina *Results*, da mesma interface.

A seguir esta descrita a estrutura em que o arquivo do usuário deverá ser montado para que o servidor consiga executá-lo, e gerar seus resultados, de forma correta.

Diretório Comprimido em formato .zip

```
nome do projeto
├── code.py
├── requirements.txt
└── output
    └── Resultados retornados pelo servidor
```

Nessa estrutura é possível notar quatro principais elementos: o *nome do projeto*, que será usado como identificador do Job enviado pelo usuário; *code.py*, que deverá ser o executável deste Job, sendo o único arquivo que deverá ter este nome; *requirements.txt*, que deverá conter todas as bibliotecas utilizadas no projeto, sendo um arquivo necessário apenas para projetos em Python; e a pasta *output*, para onde deverão ser direcionados todos os resultados que o usuário pretende ter acesso, podendo ter qualquer estrutura interna que o pesquisador deseje. É importante notar que estes elementos devem ter exatamente esses nomes ao serem enviados ao servidor, com exceção do *nome do projeto*. Além disso, este diretório compactado pode ter outros arquivos, como dados que serão processados dentro do script *code.py* ou outras classes de suporte.

Para os testes deste capítulo, foram utilizados 5 *Jobs* semelhantes, desenvolvidos na linguagem Python. Cada um realizou uma requisição a uma página da internet para fazer o download de uma imagem no formato .png de seu respectivo número e enviá-la à pasta output. Por exemplo, o Job "python-file_1" realiza o download de uma imagem contendo o Número 1, e assim por diante. Ao final do envio da imagem para a pasta output, o

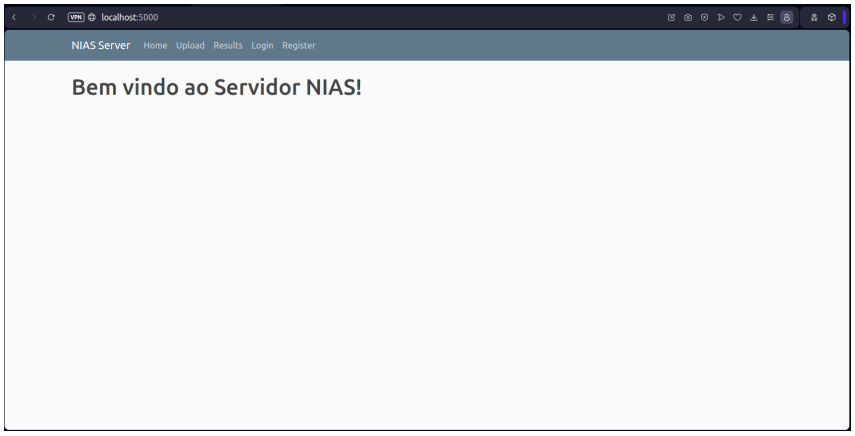
código realiza uma pausa de 30 segundos, para simular o processamento de scripts mais trabalhosos, o que será um cenário mais provável durante o funcionamento do servidor.

O Objetivo dos testes, portanto, é demonstrar que o pesquisador, ao acessar o servidor, será capaz de enviar um diretório compactado com o projeto que ele deseja que seja executado e obter seu resultado esperado, neste caso, a imagem a qual seu script foi programado para realizar o download.

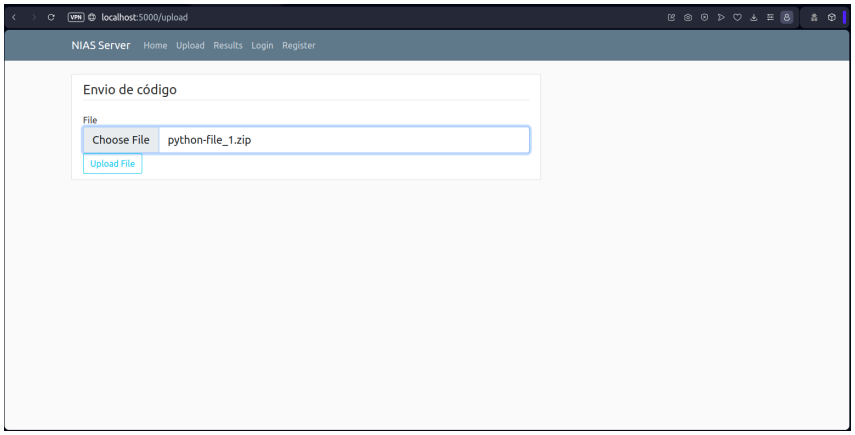
4.2 Validação de funcionamento do Servidor

Inicialmente são mostradas as páginas que poderão ser acessadas pelo usuário, de acordo com a Figura 1, assim como suas funções. Lembrando que, para atender aos requisitos do sistema, essa interface deve permitir acesso simultâneo de diversos clientes, assim como permitir uma interação simples dos clientes com o servidor. A seguir estão ilustradas as páginas Home, Upload e Results, respectivamente, Figuras 5a, 5b e 5c

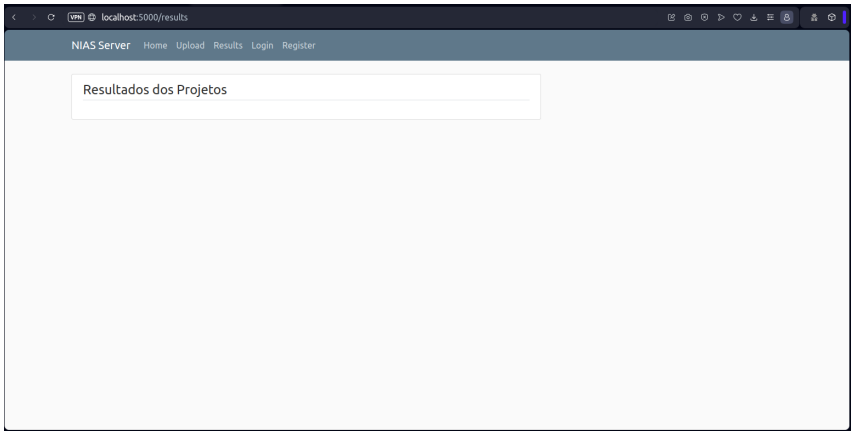
Como pode ser observado, a página Home é o primeiro local para onde o pesquisador é direcionado. A partir dela, é possível acessar Upload, que permite ao usuário enviar um diretório compactado no formato .zip, nos mesmos moldes do apresentado na seção 4.1. Por fim, a página Results é onde o usuário tem acesso aos outputs dos seus Jobs.



(a) Pagina Home



(b) Pagina para Upload de arquivo do usuário



(c) Pagina para Acesso a resultados de Job

Figura 5 – Páginas da interface de usuário.

Para demonstrar o funcionamento do sistema, são apresentados os logs de cada um de seus componentes. Inicialmente, a Figura 6 apresenta o comportamento da interface do usuário. Na imagem, é possível perceber, através do destaque número "1" (em azul), que foram realizadas 5 acessos simultâneos ao servidor (códigos de resultado 304), demonstrando a capacidade de paralelismo da página de usuário. Observa-se também, através do destaque "2", em vermelho, uma sucessão de comandos executados que representam o envio do arquivo do usuário pela interface e a resposta do produtor de mensagens indicando o sucesso da publicação da mensagem na fila.

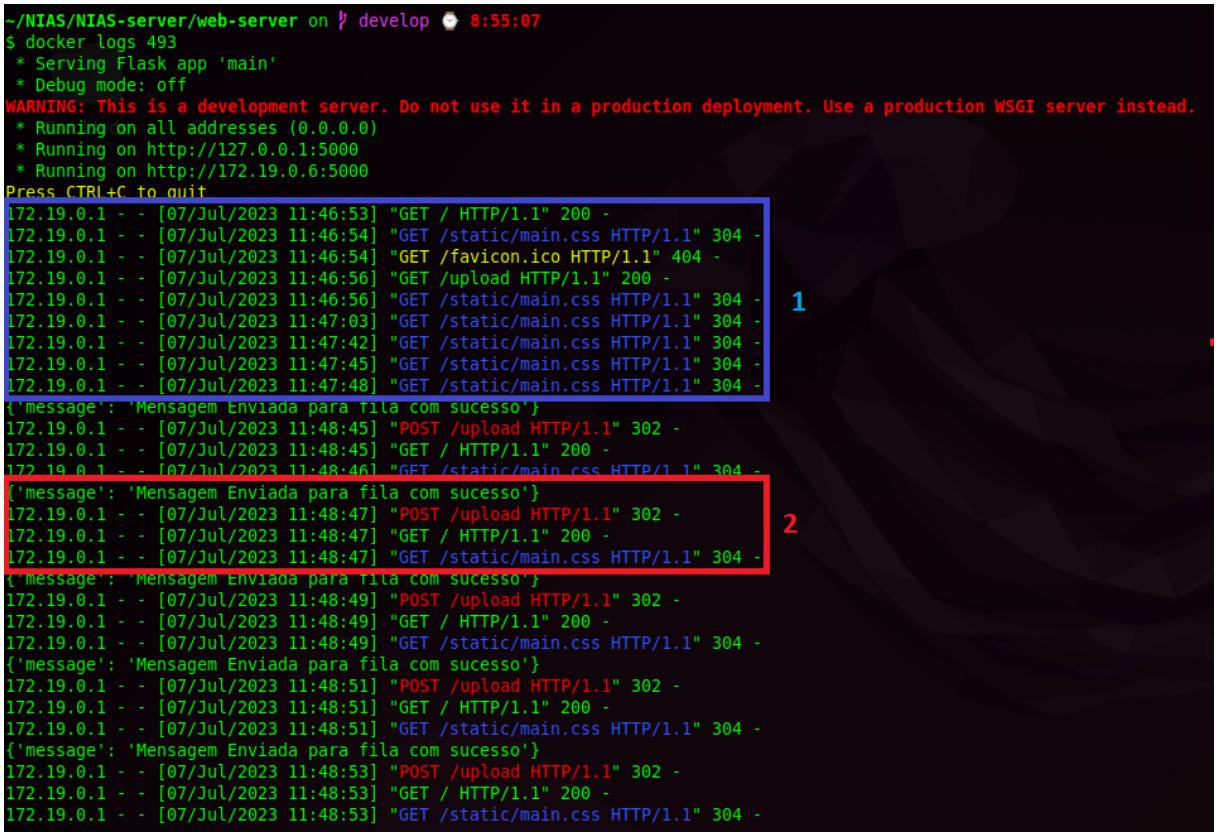


Figura 6 – Logs da página de usuário.

A seguir, a Figura 7 ilustra que os 5 arquivos enviados pela interface do servidor foram corretamente publicados na fila do *Message Broker*. Fato que é comprovado pela sequência de 5 resultados de sucesso retornados pela API do produtor (código 200).

```
~/NIAS/NIAS-server/web-server on  develop  8:57:13
$ docker logs 20f
 * Serving Flask app 'MessageProducer'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://producer:60800
Press CTRL+C to quit
172.19.0.6 - - [07/Jul/2023 11:48:45] "POST /product_message HTTP/1.1" 200 -
172.19.0.6 - - [07/Jul/2023 11:48:47] "POST /product_message HTTP/1.1" 200 -
172.19.0.6 - - [07/Jul/2023 11:48:49] "POST /product_message HTTP/1.1" 200 -
172.19.0.6 - - [07/Jul/2023 11:48:51] "POST /product_message HTTP/1.1" 200 -
172.19.0.6 - - [07/Jul/2023 11:48:53] "POST /product_message HTTP/1.1" 200 -
```

Figura 7 – Logs do produtor de mensagens para a fila.

Uma vez publicadas na fila, o *Message Broker* tem a função de armazenar as mensagens até que existam consumidores aptos a recebê-las. A Figura 8 apresenta um gráfico fornecido pelo próprio software Rabbit MQ, que possui três diferentes linhas. Em vermelho, é representado o total de mensagens armazenadas na fila em questão; em azul, são mostradas as mensagens que já foram consumidas, mas o consumidor ainda não confirmou o sucesso de seu processamento, ou seja, esse estágio representa o momento em que o *Job* já chegou à unidade de processamento mas ainda não foi totalmente executado; por fim, em amarelo, são indicadas as mensagens que estão prontas para serem consumidas mas ainda não existem consumidores disponíveis para recebe-las.

A partir do gráfico, é possível observar que, inicialmente, todas as 5 mensagens enviadas pelos usuários estão armazenadas na fila, duas delas já foram recebidas pelos consumidores e estão sendo processadas pelas unidades de processamento e três estão aguardando por disponibilidade. Após isso, conforme os *Jobs* são processados, os consumidores se tornam novamente disponíveis e apanham novas mensagens, o que é indicado pela gradativa descida de todas as linhas do gráfico, em uma progressão de 30 em 30 segundos, devido ao modo com o qual os códigos dos usuários foram construídos.

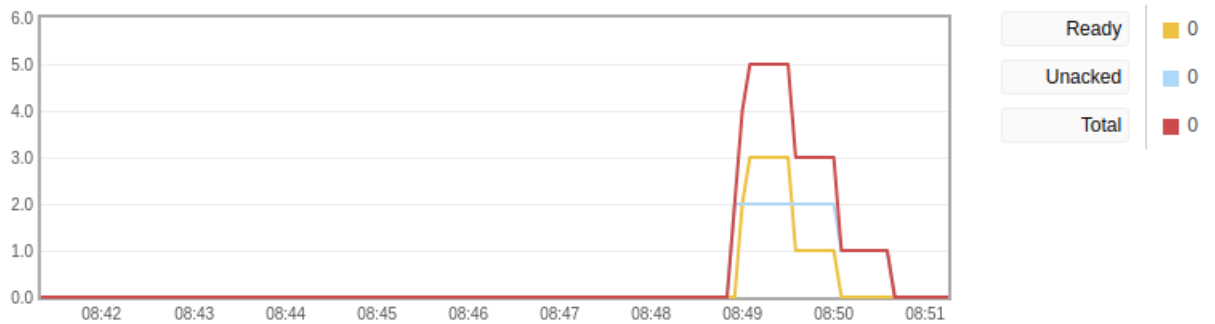


Figura 8 – Gráfico de armazenamento de mensagens no Message Broker.

Na arquitetura deste trabalho, foram utilizadas duas unidades de processamento para a execução dos *Jobs* de usuário, e como demonstrado no capítulo 3, cada unidade

é acompanhada de um consumidor. Este fato pode ser observado através do gráfico da Figura 8, já que, das cinco mensagens que estão na fila no início do processamento, é ilustrado pela linha azul que duas delas estão simultaneamente sendo processadas, o que demonstra o funcionamento paralelo das duas unidades de processamento implementadas neste trabalho.

Por fim, para ilustrar o último passo do fluxo do sistema, temos a Figura 9, que demonstra como a página Results é apresentada quando há resultados disponíveis. Nesse caso, são mostrados os outputs de todos os 5 projetos, denominados por "output_" + "Nome do projeto". Além disso, a figura 10 comprova que o que foi apresentado pelo servidor ao usuário é a saída esperada de seu código.

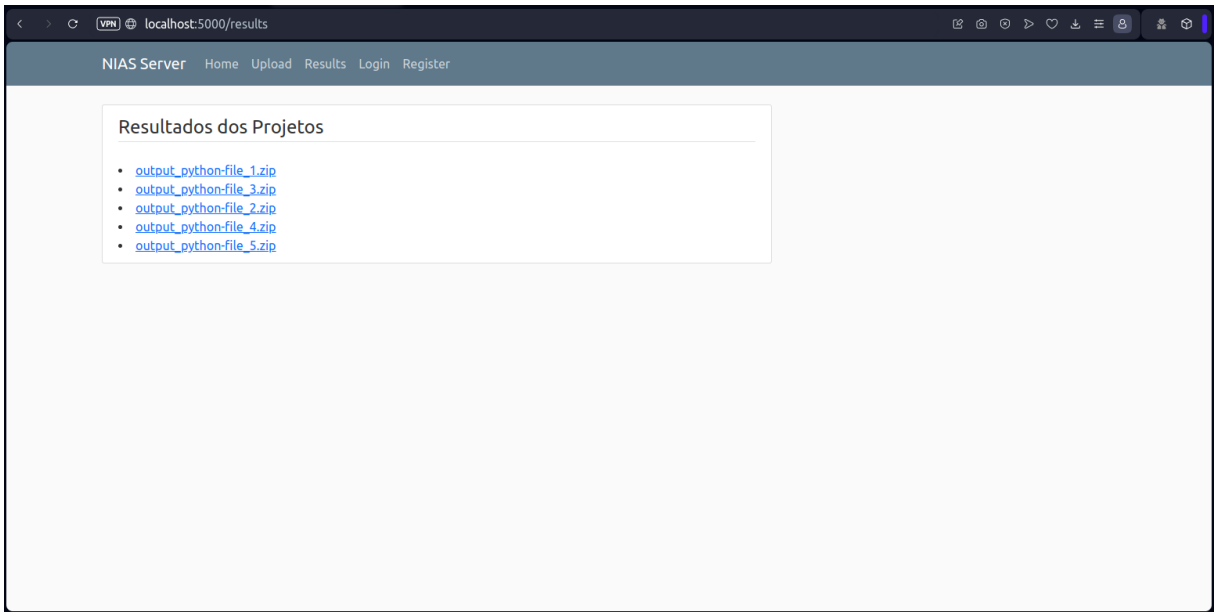


Figura 9 – Página de resultados com os *Outputs* dos *Jobs*.

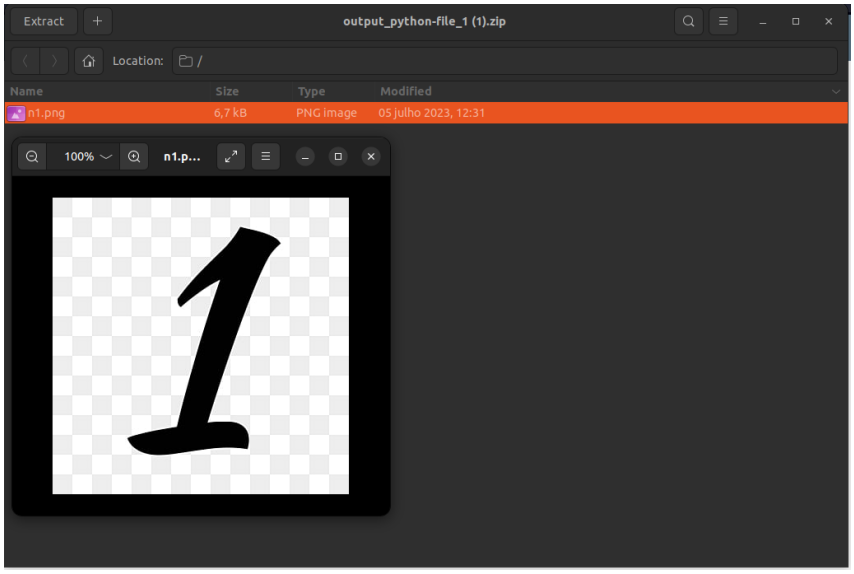


Figura 10 – Exemplo de resultado de *Job* de Usuário.

4.3 Discussão dos Resultados

A partir dos resultados apresentados, foi possível comprovar o funcionamento do sistema proposto, assim como foi idealizado no capítulo 3. Nessa implementação, foi necessário desenvolver todas as camadas do servidor de aplicações, desde a interface de usuário, passando pela comunicação entre diferentes blocos dentro do sistema e processamento dos dados trafegados, até o armazenamento dos resultados em volumes.

Ao retomar os requisitos iniciais deste trabalho, verifica-se que: o servidor permite acesso simultâneo de diferentes clientes, assim como suporta o envio de Jobs mesmo sem capacidade de processá-los instantaneamente, devido a implementação do sistema de filas; é capaz de processar scripts desenvolvidos na linguagem Python, contudo, necessita de futuras adaptações para executar códigos desenvolvidos no MATLAB; concede acesso aos resultados dos projetos enviados pelo usuário, através da página Resultados, como demonstrado pela Figura 9; e permite a escalabilidade de seus componentes, já que foi desenvolvido em uma arquitetura de microsserviços, usando containers como ferramenta de virtualização, porém, será necessária a implementação de um orquestrador mais sofisticado para a concretizar esta escala.

5 Conclusão e Trabalhos Futuros

Neste trabalho foi apresentado uma prova de conceito de um servidor de aplicações baseado em microsserviços, capaz de manter uma coerência entre seus componentes através de uma implementação de arquitetura que responde, de forma coordenada, a eventos como o upload de arquivos por parte do usuário. Também é importante salientar a utilização do recurso de mensageria que permite o desacoplamento temporal entre o cliente da aplicação e o processamento do servidor, além de permitir uma simplificação do código durante o desenvolvimento.

Apesar de cumprir com boa parte dos requisitos iniciais do projeto, o sistema desenvolvido tem algumas limitações, abrindo oportunidade para futuros trabalhos com objetivo de seu aprimoramento. Primeiramente, é necessário que haja algumas adaptações para que seja possível o envio de códigos desenvolvidos para o software MATLAB, como: a adição de uma nova fila, para armazenar Jobs desenvolvidos com esta plataforma; configuração da interface de usuário, para que o pesquisador possa selecionar a identificação de qual a linguagem em que seu projeto foi desenvolvido; e implementação de uma unidade de processamento capaz de executar arquivos através do software MATLAB.

Um passo significativo para a validação do sistema apresentado que não foi realizado neste trabalho, são os testes de carga, fundamentais para averiguar qual o trafego que este sistema suporta em seu estado atual. Este processo deverá ser feito subentendo o servidor a um alto volume de requisições e envios de arquivos em curtos períodos de tempo, para que seja possível a correta previsão da quantidade de informações que podem ser transacionadas dentro sistema. Além disso, também é fundamental validar qual a capacidade das unidades de processamento, para garantir que as tarefas enviadas pelos usuário poderão ser corretamente executadas.

Outra importante melhoria é referente à escalabilidade. Em relação a isso, para que seja possível a utilização de mais máquinas para aumentar a quantidade de recursos disponíveis para o sistema, é necessária a implementação de um orquestrador de containers para executar tarefas como o escalonamento de *Jobs* com objetivo de balanceamento de carga dentro do servidor, além de permitir a comunicação entre containers hospedados em diferentes máquinas. Alguns softwares que podem ser utilizados para este fim são o Kubernetes e o Docker Swarm.

Além dessas melhorias, o sistema também precisa de avanços em relação a segurança e monitoramento antes de ser colocado em um ambiente produtivo. Alguns desses aprimoramentos são: a implementação de um gerenciamento de usuários e permissões, para que apenas pesquisadores cadastrados possam ter acesso ao servidor; trocar procolo de

comunicação HTTP por HTTPS, já que o segundo permite que as informações comunicadas pelo servidor sejam encriptadas, aumentando assim a segurança do sistema; elaboração de uma estratégia de gestão e tratamento de logs, a fim de monitorar e dar manutenção em possíveis falhas durante o funcionamento.

Por fim, também é possível conceber uma arquitetura mais otimizada para lidar com a transmissão de arquivos dentro do servidor, já que não é a prática mais adequada fazer este tipo de dado passar diretamente por uma fila, sendo uma melhor elaboração utilizar sistemas de gerenciamento de documentos, como servidores NFS, para armazenar os arquivos de *Jobs* e enviar apenas sinais de comando através das filas. Com isso também se tornaria possível a criação de containers de processamento sob demanda, otimizando assim a utilização de recursos do servidor.

Referências

ACHARYA, J. N.; SUTHAR, A. C. Docker Container Orchestration Management: A Review. In: SHARMA, H. et al. (Ed.). *Proceedings of the International Conference on Intelligent Vision and Computing (ICIVC 2021)*. Cham: Springer International Publishing, 2022. p. 140–153. ISBN 978-3-030-97196-0. Citado na página 15.

BUCCHIARONE, A. et al. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*, v. 35, n. 3, p. 50–55, maio 2018. Citado na página 12.

DOCKER. *Docker Documentation*. 2021. Disponível em: <https://docs.docker.com/reference/>. Citado na página 21.

FLASK. *Flask Documentation*. 2023. Disponível em: <https://flask.palletsprojects.com/en/2.3.x/>. Citado na página 18.

FU, G. et al. A Fair Comparison of Message Queuing Systems. *IEEE Access*, v. 9, p. 421–432, 2020. Citado na página 13.

JIMENEZ, L. L.; SCHELEN, O. HYDRA: Decentralized Location-aware Orchestration of Containerized Applications. *IEEE Transactions on Cloud Computing*, p. 1–1, 2020. Citado na página 14.

LAIGNER, R. et al. From a Monolithic Big Data System to a Microservices Event-Driven Architecture. p. 213–220, 2020. Citado na página 12.

LAMOTHE, M. et al. A Systematic Review of API Evolution Literature. *ACM Computing Surveys*, out. 2021. Citado na página 14.

M. Netto et al. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Computing Surveys*, 2018. Citado na página 8.

OFOEDA, J. et al. Application Programming Interface (API) Research: A Review of the Past to Inform the Future. *International Journal of Enterprise Information Systems*, v. 15, n. 3, p. 76–95, jul. 2019. Citado na página 14.

RABBITMQ. *RabbitMQ Documentation*. 2023. Disponível em: <https://www.rabbitmq.com/documentation.html>. Citado na página 19.

RUíZ, L. M. et al. Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware. *IEEE Access*, v. 10, p. 33083–33094, 2022. Citado na página 9.

SHARMA, P. et al. Containers and Virtual Machines at Scale: A Comparative Study. *International Middleware Conference*, p. 1, nov. 2016. Citado na página 11.

TKACHENKO, O.; TKACHENKO, O.; TKACHENKO, K. Actual Trends of Cloud Computing and Technologies in Optimization of Data Storage. *Digital Platform Information Technologies in Sociocultural Sphere*, v. 3, n. 2, p. 192–208, 2020. Citado na página 8.

YADAV, M. P.; PAL, N.; YADAV, D. K. A formal approach for Docker container deployment. *Concurrency and Computation: Practice and Experience*, v. 33, n. 20, maio 2021. Citado na página 11.