CENTRO UNIVERSITÁRIO DO PARÁ - CESUPA
ESCOLA DE NEGÓCIOS, TECNOLOGIA E INOVAÇÃO - ARGO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOÃO TOMÉ DE FARIAS
RENATO SILVA DE AQUINO

IMPLEMENTAÇÃO DE AMBIENTE DE BAIXA COMPLEXIDADE PARA SISTEMA DISTRIBUÍDO BASEADO EM NGINX

BELÉM
2020

JOÃO TOMÉ DE FARIAS

RENATO SILVA DE AQUINO

IMPLEMENTAÇÃO DE AMBIENTE DE BAIXA COMPLEXIDADE PARA SISTEMA DISTRIBUÍDO BASEADO EM NGINX

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Ciência da Computação na modalidade PRODUTO.

Orientador(a): MSc. Alessandra Natasha Alcantara Baganha

BELÉM

2020

JOÃO TOMÉ DE FARIAS
RENATO SILVA DE AQUINO

IMPLEMENTAÇÃO DE AMBIENTE DE BAIXA COMPLEXIDADE PARA SISTEMA DISTRIBUÍDO BASEADO EM NGINX

Trabalho de conclusão de curso apresentado à Escola de Negócios, Tecnologia e Inovação do Centro Universitário do Estado do Pará como requisito para obtenção do título de Bacharel em Ciência da Computação na modalidade PRODUTO.

Data da aprovação: / /
Nota final aluno I:_____
Nota final aluno II:_____

Banca examinadora

_____
Prof. MSc. Alessandra Natasha
Orientador(a) e Presidente da banca

_____
Prof. MSc. Eudes Danilo
Examinador interno

_____
Prof. MSc. Ricardo Casseb
Examinador interno

RESUMO

Sistemas distribuídos preveem descentralização de processos e alta escalabilidade, entretanto, nem sempre associados ao aumento de desempenho e baixa complexidade. Isto acontece devido a dificuldade de alocação de processos em determinados processadores e complicação na implementação do ambiente, além da dependência da rede - desvantagens reconhecidas da distribuição e responsáveis, desta forma, pela subutilização do sistema. O pior resultado, nesse caso, implica na negação do acesso ou limitação dos recursos para manutenção da integridade dos serviços. A busca pelo alto desempenho, simplicidade de implementação e integridade da solução são constantes e fontes de diversas publicações em Sistemas Distribuídos. Este Trabalho de Curso tem como finalidade o desenvolvimento de uma solução para este enfoque, intuindo otimização no acesso, baixa complexidade e integridade em sistemas distribuídos.

Palavras-chave: Sistemas distribuídos, baixa complexidade, integridade, tolerância a falhas, otimização de desempenho.

ABSTRACT

Distributed systems foresee process decentralization and scalability, however, not always associated with increased performance and low complexity. This happens due to the difficulty of allocating processes on certain processors and complicating the implementation of the environment, recognized disadvantages of the distribution and responsible, therefore, for the underutilization of the system. The worst result, in this case, implies denying access or limiting resources to maintain the integrity of services. The search for high performance, simplicity of implementation and integrity of the solution are constant and sources of several publications in Distributed Systems. This work aims to develop a product to propose a solution for this approach, intending to optimize access, low complexity and integrity in distributed systems.

Keywords: *Distributed systems, low complexity, scalability, fault tolerance, performance optimization.*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

ABNT - Associação Brasileira de Normas Técnicas

CPU - Central Process Unit (Unidade Central de Processamento)

PM2 - Production Process Manager (Gerenciador de Processos em Produção)

IaaS - Infrastructure as a Service (Infraestrutura como Serviço)

PaaS - Platform as a Service (Plataforma como Serviço)

SaaS - Software as a Service (Software como Serviço)

HTTP - Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)

HTTPS - Hypertext Transfer Protocol Secure (Protocolo de Transferência de Hipertexto)

XML - eXtensible Markup Language (Linguagem de Marcação Extensiva)

JSON - JavaScript Object Notation (Notação de Objeto para JavaScript)

IME - Instituto Militar de Engenharia

TCP/IP - Transmission Control Protocol (Protocolo de Controle de Transmissão)

IP - Internet Protocol Access (Protocolo de Acesso a Internet)

TB - Terabyte (Unidade de medida)

NPM - Node Package Manager (Gerenciador de Pacotes do Node)

MS - Milisegundos

SUMÁRIO

1. INTRODUÇÃO

É crescente a demanda de serviços em nuvem oferecidos pelas empresas de grande porte tais como mensagens de texto, *streamings* de vídeo, compartilhamento de nuvem. Exemplos dessas empresas podem ser encontrados no *WhatsApp®, Youtube®, Google Drive®*. O alto volume de consumo destes serviços impõe a adoção de novas arquiteturas computacionais, via de regra, associadas a um alto custo e grande complexidade de implementação.

Segundo *Tanenbaum & Steen* (2007) sistemas distribuídos podem ser definidos por "uma coleção de dispositivos autônomos conectados por uma rede de comunicação que é percebida pelos usuários como um único dispositivo provendo serviços ou resolvendo algum problema". Tal definição, feita pelo autor, remete, de forma intrínseca, ao conceito de transparência.

Dentro da arquitetura distribuída, entretanto, encontram-se alguns problemas recorrentes. Dentre eles, afirma *Steve Muir* (2004):

> "*It's hard to maintain a consistent node configuration across a distributed system like PlanetLab. The most obvious effect of this overutilization is that applications typically run much slower on PlanetLab, since they only receive a fraction of the CPU time available.*" (Muir, 2004).

Com base nas palavras de Steve Muir (2004), a preocupação recorre a questões de consistência e desempenho. Estas, apesar de inerentes ao ambiente distribuído, na prática são difíceis de obter-se, sobretudo quando se referem à transparência e tolerância a falhas.

Cotidianamente, os sistemas de grande porte oferecem um alto processamento de dados. Entretanto, eventualmente, tais sistemas apresentam uma instabilidade nos seus serviços destinados aos usuários. A alta demanda não prevista de usuários consome os recursos disponíveis que estavam previamente disponibilizados para uma operação específica, incorrendo, por vezes, em exceções inesperadas onde a solução, invariavelmente, está ligada a grandes investimentos de ambientes computacionais e complexidade de implementação.

Ainda quanto à performance na arquitetura distribuída, pode-se citar a preocupação quanto à verificação de recursos disponíveis em tempo real, o que

acarreta migração de processamento inútil e cuidado com a Tolerância a Falhas. Para Muir (2004):

> "*A further difference between developing an application on a single-user workstation and deploying it on a distributed system like PlanetLab is that one often does not have complete access to system nodes in the latter environment.*"(Muir 2004).

Assim, o aproveitamento dos serviços, a qualidade da estrutura, a simplicidade, o custo da implementação, e a adaptação do sistema tolerante à demanda operacional são fatores que estão diretamente associados ao desempenho. Segundo Coulouris (2011):

> *"Each node continually monitors the service it is getting from its parent node (and, as mentioned above, keeps this historical information for future reference). Adaptation is triggered if the detected rate drops significantly below the expected rate from the source. To avoid thrashing, a node must wait for a particular period, known as the detection time, before electing to adapt. Once a decision is made to adapt, the node will invoke the parent selection algorithm to determine a new, more optimal parent. In this way, the tree construction is constantly re-evaluated and will self-organize to optimize overall performance."(Coulouris 2011).*

Este trabalho propõe a implementação de um ambiente de baixa complexidade, contribuindo para a otimização do processamento e garantindo tolerância à falhas e consistência das operações em Sistemas Distribuídos.

1.1 JUSTIFICATIVA

Segundo David Kroenke (2017), nos primórdios da computação, todo processamento era feito apenas em um computador (*mainframe*), consistindo em um ambiente centralizado. Com o passar do tempo, surgiram novas arquiteturas de computação que buscaram distribuir e descentralizar o processamento, almejando aumento de desempenho e oferecendo tolerância a falhas.

A partir deste conceito, e do surgimento do paradigma da distribuição, serviços gerenciados por órgãos públicos e por empresas privadas conseguem suprir as milhares de requisições diárias para os servidores durante sua utilização. Porém, com

a crescente demanda por serviços digitais, conforme exemplificado na Introdução deste Trabalho, ocorre um aumento significativo no volume de requisições em servidores, causando uma taxa maior de atraso na resposta para o usuário e, em algumas situações mais específicas, a queda inesperada do servidor.

A rejeição de requisições e até, em casos raros, a inconsistência de dados do usuário no servidor, são consequências de tal situação. Com o exposto, e a necessidade constante dos serviços distribuídos, propõe-se um ambiente com balanceamento de carga e simplicidade de implementação como forma de desmistificar o acesso a um ambiente de maior poder de processamento e tolerante à falhas na camada de aplicação.

1.2 OBJETIVO GERAL

O objetivo deste trabalho é implementar um ambiente de baixa complexidade para sistemas distribuídos de alta performance, baseado em NGINX.

1.2.1 OBJETIVOS ESPECÍFICOS

Para o desenvolvimento da proposta, serão necessários cumprir como objetivos específicos:

- Implementar o Servidor de Nodes utilizando a biblioteca Express com o NodeJS
- Implementar o Servidor principal que fará o balanceamento automático de requisições para os servidores filhos
- Possibilitar a visualização em tempo real de processamento da carga através da biblioteca PM2
- Visualizar os relatórios de processamento gerados pelos Nodes através da biblioteca PM2

1.3 ESTRUTURA DO TRABALHO

A primeira parte do trabalho abrange a introdução, onde foi apresentada a proposta para o problema, além da justificativa e complexidade do assunto. Em seguida,o capítulo dois apresenta a revisão bibliográfica e a base teórica do assunto.

A metodologia utilizada é o tema do capítulo três, expondo o processo de construção do projeto. Ainda no mesmo capítulo, são expostos diagramas que constam desde o funcionamento dos servidores até a interação do usuário com o sistema aqui proposto. Integram este capítulo, também, as tecnologias utilizadas e homologação do MVP. Os resultados são apresentados no capítulo quatro, mostrando a abordagem de forma gráfica através de uma tabela. O capítulo cinco traz as discussões, abordando comentários acerca dos resultados. O capítulo seis, por fim, refere-se à conclusão do Trabalho.

2. REVISÃO BIBLIOGRÁFICA

Para um melhor entendimento sobre o projeto, serão esclarecidos alguns tópicos, iniciando-se por terminologias e conceitos utilizados.

2.1 Sistemas Distribuídos

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em redes se comunicam e coordenam suas ações apenas passando mensagens (Coulouris,2013). Sobre demais características de um sistema distribuído, segundo Coulouris (2013):

> *"The key challenges faced by the designers of distributed systems: heterogeneity, openness, security, scalability, failure handling, concurrency, transparency and quality of service." (Coulouris, 2013).*

O autor nos remete à tópicos que são essenciais para um sistema distribuído, como: entregar um serviço de qualidade e com uma estrutura escalável e otimizada. Ainda sobre tais características, Coulouris (2013) afirma a seguir.

2.1.1 Transparência

O usuário não necessita conhecer a localização dos recursos que estão sendo fornecidos para fornecer até segurança para ambos. Replicação, é sobre o usuário não necessitar saber a quantidade X do recurso existente. Abordando

sobre a migração, que se remete aos recursos mudarem de lugar a qualquer instante.

A concorrência mencionada por Coulouris (2013), é explicada pelos recursos que podem ser disputados por usuários, porém, sem o conhecimento dos mesmos. O paralelismo se torna necessário em aplicações com sistemas distribuídos pela sua capacidade de fazer múltiplas atividades, podendo executar simultaneamente alguns recursos, sem o conhecimento do usuário.

2.1.2 Comunicação

Segundo (Coulouris, 2013) comunicação é dependente exclusivamente do esquema do sistema montado por tal empresa ou até mesmo por um órgão público. É de extrema importância que a comunicação tenha confiabilidade, portanto, o esquema escolhido deve ter um enfoque em maximizar a troca de mensagens e minimizar as taxas de perdas ou interferências. Existem atores que podem influenciar a escolha do esquema de sistema distribuído

- Atraso (Delay)
- Erro na conexão ou transmissão de dados
- Tempo máximo de espera

2.1.3 Sincronismo

O problema de sincronismo em sistemas distribuídos é um tópico deste assunto muito discutido por pesquisadores, pois, existem dois tipos de sistemas principais, no qual foram abordados: os sistemas centralizados e os sistemas distribuídos. Segundo Coulouris (2011):

> *"When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct*

> *consequence of the fact that the only communication is by sending messages through a network."* (Coulouris, 2011)

O sistema centralizado não necessita enviar mensagens para os seus servidores internos, diferentemente dos sistemas distribuídos, no qual é necessário o envio das mensagens para manter um "tempo correto" em seus servidores. No sistema centralizado, ocorre um nível de insegurança maior, devido o sistema estar presente em apenas um núcleo.

Em relação ao sistema distribuído, o tópico de segurança torna-se mais interessante devido ao seu esquema de separação da informação em diversas máquinas e discos rígidos. Outro problema existente em esquemas de sistema distribuídos é a disputa de recursos.

2.1.4 Escalabilidade

Um esquema montado de sistema distribuído pode ser expandido de duas maneiras no qual foram abordados neste Trabalho. O escalonamento horizontal, é quando uma estrutura de sistema distribuído permite a estrutura facilitada de adições de mais nós (nodes), podendo ou não aumentar o desempenho e segurança do sistema. Todavia, existe o escalonamento vertical, que propõe uma adição de recursos somente em um único nó (node), onde geralmente são melhorados os componentes dos mesmos.(Tanenbaum, 2005)

2.1.5 Tolerância à falhas

Um sistema distribuído que prevê um plano de tolerância a falhas tem grandes chances de quando ocorrer algum erro, a disponibilidade do servidor continuar, pois, a consistência e integridade do sistema garantem o fornecimento dos serviços mesmo quando há alguma falha no hardware ou software. Existem algumas maneiras de contornar o erro sem prejudicar a experiência do usuário no ambiente.

Assim sendo, um hardware pode ser replicado quando necessário para uma operação importante. Além disso, um software é projetado para recuperar o estado de dados quando uma falha é detectada. Quando um nó cair, haverá um

ciclo de segurança que optará por um desses planos, fazendo que o sistema distribuído funcione em tempo integral e com o mínimo de falhas possíveis. Segundo (Coulouris, 2013):

> "Por ser parte essencial nos sistemas distribuídos, é essencial que o serviço de arquivo distribuído continue a funcionar diante de falhas de clientes e servidores. Felizmente, um projeto moderadamente tolerante a falhas é fácil para servidores simples" (Coulouris, 2013)

2.1.6 Disponibilidade

Em relação a integridade e disponibilidade de um sistema distribuído, é necessário manter níveis de segurança adequados e funcionais. Segundo Coulouris (2011):

> *The availability of a system is a measure of the proportion of time that it is available for use. When one of the components in a distributed system fails, only the work that was using the failed component is affected. A user may move to another computer if the one that they were using fails; a server process can be started on another computer.*

O autor enfatiza que quando algum serviço for falho, o componente que o usa é afetado, por isso a disponibilidade juntamente com algum tratamento de falhas são pontos importantes para qualquer sistema distribuído.

2.2 Servidor

Um servidor é um computador equipado, geralmente, com mais de um processador que são capazes de processar uma grande demanda de dados, além de, possuir um grande estoque de memória, recursos para conectividade e uma alta gama de dispositivos para armazenamento de dados, como HD (Hard Disk) e SSD (Solid State Drive). Segundo Kurose (2013),

> "A Internet é uma rede de computadores que interconecta centenas de milhões de dispositivos de computação ao redor do mundo. Há pouco tempo, esses dispositivos eram basicamente PCs de mesa, estações de trabalho

> Linux, e os assim chamados servidores que armazenam e transmitem informações, como páginas da Web e mensagens de e-mail.” (Kurose,2013)

Há alguns servidores específicos para cada aplicação. Neste caso, será necessário abordar o servidor web. Este tipo de aplicação fornece serviços para disponibilizar o conteúdo que pode ser acessado através de navegadores via *HTTP* (*Hypertext Transfer Protocol*), principal serviço demandado pela plataforma NGINX, como será mostrado adiante, no subitem 3.1.3.3 deste Trabalho.

2.3 Nuvem

Uma nuvem em computação distribuída é formada por servidores *online* que armazenam dados do usuário, possibilitando editar, compartilhar e até mesmo excluir arquivos (Tanenbaum, 2015):

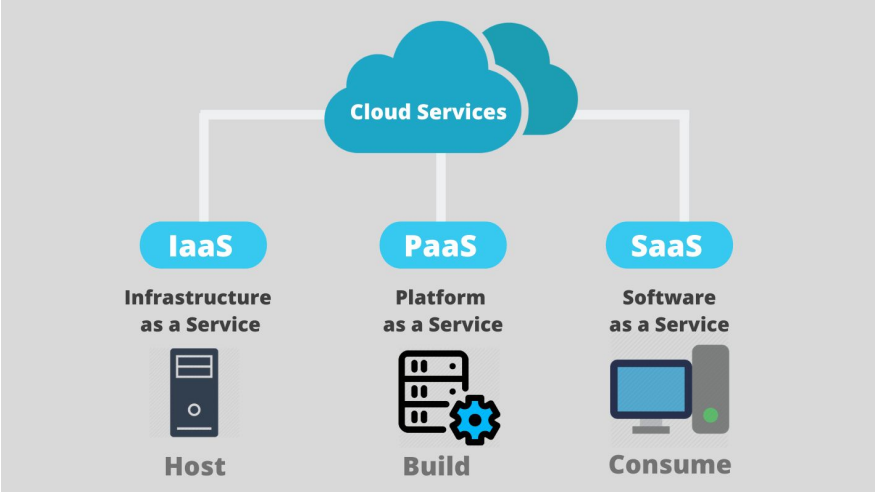
> “Como resultado do rápido progresso tecnológico, essas áreas estão convergindo rapidamente e são cada vez menores as diferenças entre coleta, transporte, armazenamento e processamento de informações. Organizações com centenas de escritórios dispersos por uma extensa área geográfica podem, com um simples apertar de um botão, examinar o status atual de suas filiais mais remotas.À medida que cresce nossa capacidade de colher, processar e distribuir informações, torna-se ainda maior a demanda por formas de processamento de informações ainda mais sofisticadas.” (Tanenbaum, 2015).

A nuvem traz possibilidades para o usuário que demanda armazenamento físico. Neste caso, o usuário necessita de uma conexão estável para acessar os dados da nuvem. Segundo Kurose (2013):

> “Diversas empresas e universidades também migraram suas aplicações da Internet (por exemplo, e-mail e hospedagem de páginas Web) para a nuvem. Empresas de nuvem não apenas oferecem ambientes de computação e armazenamento escaláveis às aplicações, mas também lhes oferecem acesso implícito às suas redes privativas de alto desempenho.” (Kurose, 2013)

Ainda sobre nuvem, outros tópicos precisam ser expostos para melhor compreensão da proposta deste Trabalho. Conforme mostra a figura 1 a seguir.

Figura 1 - Serviços da nuvem



Fonte: https://cloudmartial.com/iaas-paas-saas/

O *IaaS* são apresentadas como recursos oferecidos pelos fornecedores da nuvem, como armazenamento, networking e outros serviços associados, do ponto de vista do usuário pode-se tornar interessante, já que ao invés, de comprar algum dispositivo que substituísse o serviço prestado pela nuvem, ele poderia pagar por um serviço que não precisaria estar fisicamente no local para ser utilizado, conforme explanado por (Cloud Martial, 2020).

Ainda segundo (Cloud Martial,2020), a *PaaS* é apresentada como serviços disponíveis na nuvem, onde a empresa que a contrata com algum provedor precisa

apenas se preocupar com o ambiente de programação (software), pois, toda a infraestrutura está montada e pronta para ser utilizada pela empresa.

E por fim, o *SaaS*, são aqueles serviços que são providos de empresas que cobram pelo uso do software, licenças, certificações entre outros. Em alguns softwares, existem um limite onde o usuário pode utilizar de forma gratuita, caso ele passe, será cobrado um valor em cima do uso diário, mensal ou anual do software.

2.4 Métodos de Requisição HTTP

“As requisições HTTP consistem de protocolos com 8 (oito) definições de métodos entre eles, destacam-se as 4 (quatro) mais relevantes” (LIMA, Leandro, 2004):

- GET: A requisição *GET* é utilizada para a visualização das informações gerada através de uma resposta *HTTP*, podendo ter a representação de *HTML, XML, JSON* e outros.
- POST: O método *“POST”* é utilizado para envio de recursos através do corpo da requisição, onde podem ser passadas informações.
- PUT: O método *“PUT”* geralmente utilizado para alguma atualização, tem um corpo onde possui dados que podem ser alterados.
- DELETE: método DELETE, que através de um corpo de requisição possui alguma informação que será validada no servidor para a exclusão.

A Figura 2 a seguir possui uma ilustração das sequências de código para os métodos de requisições:

Figura 2 - Script de status HTTP

Fonte: http://leandromtr.com/codigos-de-status-http/

Segundo Molinari (2016), os cabeçalhos são definidos como:

> "Os cabeçalhos são metadados que podem ser utilizados para enviar informações relevantes para o servidor e para quem está no meio caminho da conexão. As informações presentes nos cabeçalhos podem ser usadas no meio do caminho por servidores de cache, servidores web e outros softwares." (Molinari,2016)

2.5 Algoritmos utilizados no Trabalho de Curso

Alguns algoritmos foram recorrentemente citados neste Trabalho de Curso, seja no âmbito dos Sistemas Operacionais, para escalonamento dos processos; seja para a ordenação no atendimento da demanda das cargas de trabalho. Neste item serão descritos tais algoritmos.

2.5.1 Algoritmos de escalonamento

Em relação aos algoritmos de escalonamento tem-se que:

> "Os algoritmos de balanceamento de carga envolvem quatro políticas: transferência, seleção, localização e informação. A política de transferência determina se uma máquina pode participar numa transferência de tarefas, como servidor ou receptor de processos. A política de seleção define os processos que devem ser transferidos da máquina mais ocupada

> para a menos ocupada. A política de localização é responsável pela procura de um parceiro de transferência adequado (servidor ou receptor) para uma máquina, uma vez que a política de transferência decidiu sobre seu estado. Um servidor oferece processos, quando está sobrecarregado; um receptor requisita processos, quando não está ocupado. A política de informação define quando e como a informação sobre o estado dos computadores é atualizada no sistema”**(**NERY, Bruno Rodrigues; DE MELLO, Rodrigo Fernandes; DE CARVALHO, 2006**).**

A seguir serão apresentados os algoritmos utilizados pela plataforma NGINX, adotada neste Trabalho de curso, conforme será melhor exposto nos subitens 2.5.1.1 e 2.5.1.2**.**

2.5.1.1 Round Robin

O algoritmo de Round Robin dispõem de frações de tempo para cada processo em partes iguais desta forma os processos são executados sem prioridades. segundo o site do ime acessado em 2 de dezembro de 2020

> **“**Inspirado na história de Robin Hood onde, na procura de justiça, Robin roubava dos ricos para entregar aos pobres, fazendo assim com que todos no seu reino tivessem o mesmo tanto de bens. Uma das mais simples e robustas entre as atuais técnicas utilizadas para problemas de distribuição de carga, nesse escalonamento o sistema operacional possui um timer, chamado de quantum, onde todos os processos ganham o mesmo valor de quantum para rodarem na CPU, depois que o quantum acaba e o processo não terminou, ocorre uma preempção e o processo é inserido no fim da fila. Se o processo termina antes de um *quantum*, a CPU é liberada para a execução de novos processos.” (IME, acessado em 2 de dezembro 2020)

2.5.1.2 Least Connection

O algoritmo “Least Connection” tem como conceito o menor número de conexões ativas nos servidores, é possível levar em conta a quantidade de carga em que cada servidor está processando no momento, segundo o site Kemp Technologies (acessado em 2 de dezembro de 2020):

> “Least Connection load balancing is a dynamic load balancing algorithm where client requests are distributed to the application server with the least

> number of active connections at the time the client request is received. In cases where application servers have similar specifications, an application server may be overloaded due to longer lived connections; this algorithm takes the active connection load into consideration.”
>
> (Kemp Technologies, acessado em 2 dezembro de 2020)

2.5.2 Algoritmo de ordenação Bubble Sort

“O *Bubble Sort*, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vector diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.”(Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 2001).

3 METODOLOGIA

Neste capítulo explicam-se e descrevem-se as ferramentas e tecnologias utilizadas para o desenvolvimento do produto apresentado.

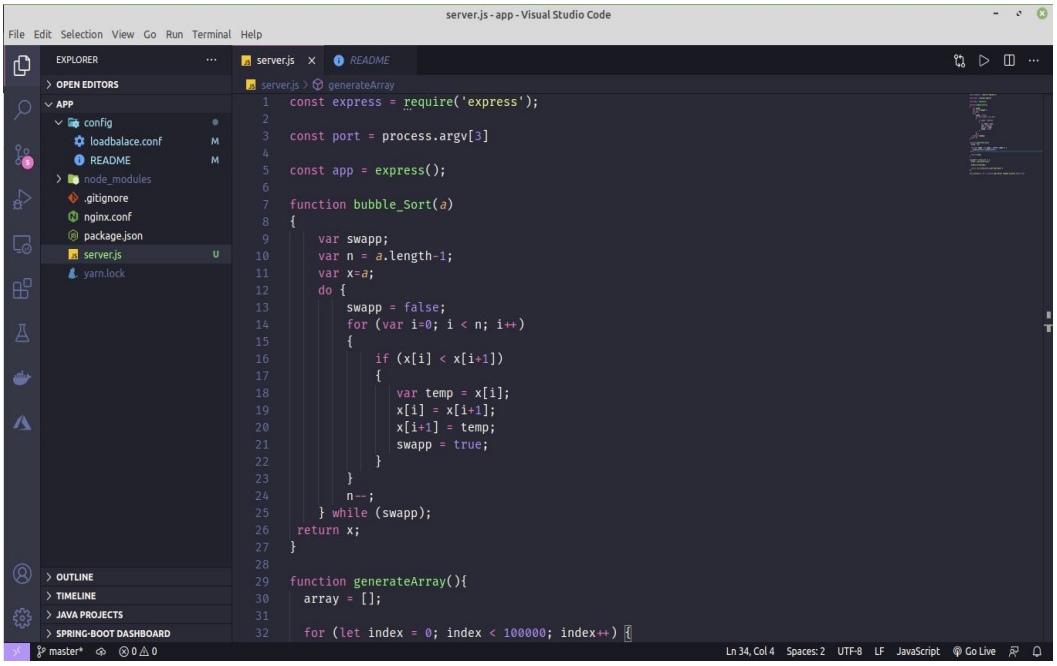
3.1 Desenvolvimento do produto

O sistema operacional utilizado para o desenvolvimento do ambiente foi o Linux Mint ®, versão 20 Ulyana - Cinnamon devido a vasta documentação encontrada. Entretanto, a construção em ambiente Windows pode igualmente ser feita sem complicações.

Inicialmente, precisa-se de um editor de texto instalado. Neste projeto foi utilizado o Visual Studio Code, devido a facilidade de utilização, mas qualquer um poderia ser utilizado.

É possível instalá-lo através deste link: https://code.visualstudio.com/, como mostra a Figura 3 a seguir.

Figura 3: Uma amostra do editor de texto Visual Studio Code.



Fonte: Autor

Será necessário instalar o runtime do *JavaScript,* o Node JS. Através deste link, é possível acessar o *site* do framework e instalá-lo com o respectivo sistema operacional.

Comandos para instalação:

- “curl -sL https://deb.nodesource.com/setup_15.x | sudo -E bash -”
- “sudo apt-get install -y nodejs”

  Link para download: https://nodejs.org/en/

Tem-se também que instalar o responsável por gerenciar e fazer o balanceamento automático da carga para os servidores, funcionalidade *built in* do servidor *NGINX*. A instalação pode ser feita através do link para download: http://nginx.org/en/download.html

No Linux, será criada uma pasta chamada “/etc/nginx” onde será o caminho para a ferramenta. Em seguida, será necessário colocar o *script* de inicialização dos servidores que processam a carga. No Linux deverá ser acessado a pasta de configuração do *NGINX*, a partir do comando “*cd conf.d*”, e criado um *script* chamado “loadbalance.conf”, com o comando *“sudo nano loadbalance.conf”*.

A partir deste ponto o arquivo será criado e poderá ser editado. Logo após, deverá ser alterada a configuração padrão do arquivo para o *script* em seguida, conforme mostrado na Figura 4:

Figura 4: *Script* de configuração NGINX

```
upstream loadbalance {
    server 127.0.0.1:8000;
    server 127.0.0.1:8001;
    server 127.0.0.1:8002;
    server 127.0.0.1:8003;
}

server {
    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_set_header X-NginX-Proxy true;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_pass http://loadbalance;
    }
}
```

Fonte: Autor

Depois de acrescentar o *script* no arquivo, será necessário a reinicialização do servidor NGINX, pois o arquivo será carregado pela ferramenta. Deverá ser feito o reinício através do comando: *"sudo systemctl restart nginx"*.

A configuração do servidor de gerenciamento está concluída, agora será necessário a instalação das bibliotecas *PM2 e Express*. A instalação das bibliotecas podem ser realizadas de várias formas. Citam-se duas, a seguir.

Caso utilize o *NPM, o instalador de pacotes padrão do Node JS, é possível baixá-lo através deste link:* https://www.npmjs.com/. Após baixado o gerenciador de pacotes, é possível a instalação através do comando: "npm install pm2@latest -g" e *"sudo npm install -g express"*. Já caso seja utilizado o *Yarn, outro gerenciador de pacotes*, que estará disponível pelo link: (https://www.yarnpkg.com/), sendo necessário utilizar os comandos: *"yarn global add pm2" e "sudo yarn add global express".*

Após as instalações finalizadas, é preciso criar uma pasta fora dos diretórios administrativos, portanto, é recomendado que crie na área de trabalho. Logo em

seguida, o arquivo do servidor deverá ser criado através do comando: *“sudo nano server.js”*, e por fim, é preciso que seja adicionado ao arquivo criado, as seguinte linhas de código:

Figura 5 - Código para criação de servidor

```
const express = require('express');

const port = process.argv[3]

const app = express();

function bubble_Sort(a)
{
    var swapp;
    var n = a.length-1;
    var x=a;
    do {
        swapp = false;
        for (var i=0; i < n; i++)
        {
            if (x[i] < x[i+1])
            {
                var temp = x[i];
                x[i] = x[i+1];
                x[i+1] = temp;
                swapp = true;
            }
        }
        n--;
    } while (swapp);
 return x;
}

function generateArray(){
  array = [];

  for (let index = 0; index < 100000; index++) {
    array.push(Math.random(1000));
  }

  return array;
}

app.get('/',(req,res) => {
  array = generateArray();

  bubble_Sort(array);

  return res.status(200).send("Task done!")
})
```

Fonte: Autor

Com o servidor que processa as cargas e o que redireciona configurados, deve-se colocar em produção com os seguintes comandos:

- “pm2 start server.js -f -- --port 8000”
- “pm2 start server.js -f -- --port 8001”
- “pm2 start server.js -f -- --port 8002”
- “pm2 start server.js -f -- --port 8003”

Com os comandos executados, os servidores ficaram nas portas 8000, 8001, 8002, 8003, podendo ser acessada diretamente pelo endereço “http://localhost:8000” ou “http://127.0.0.1:8000”.

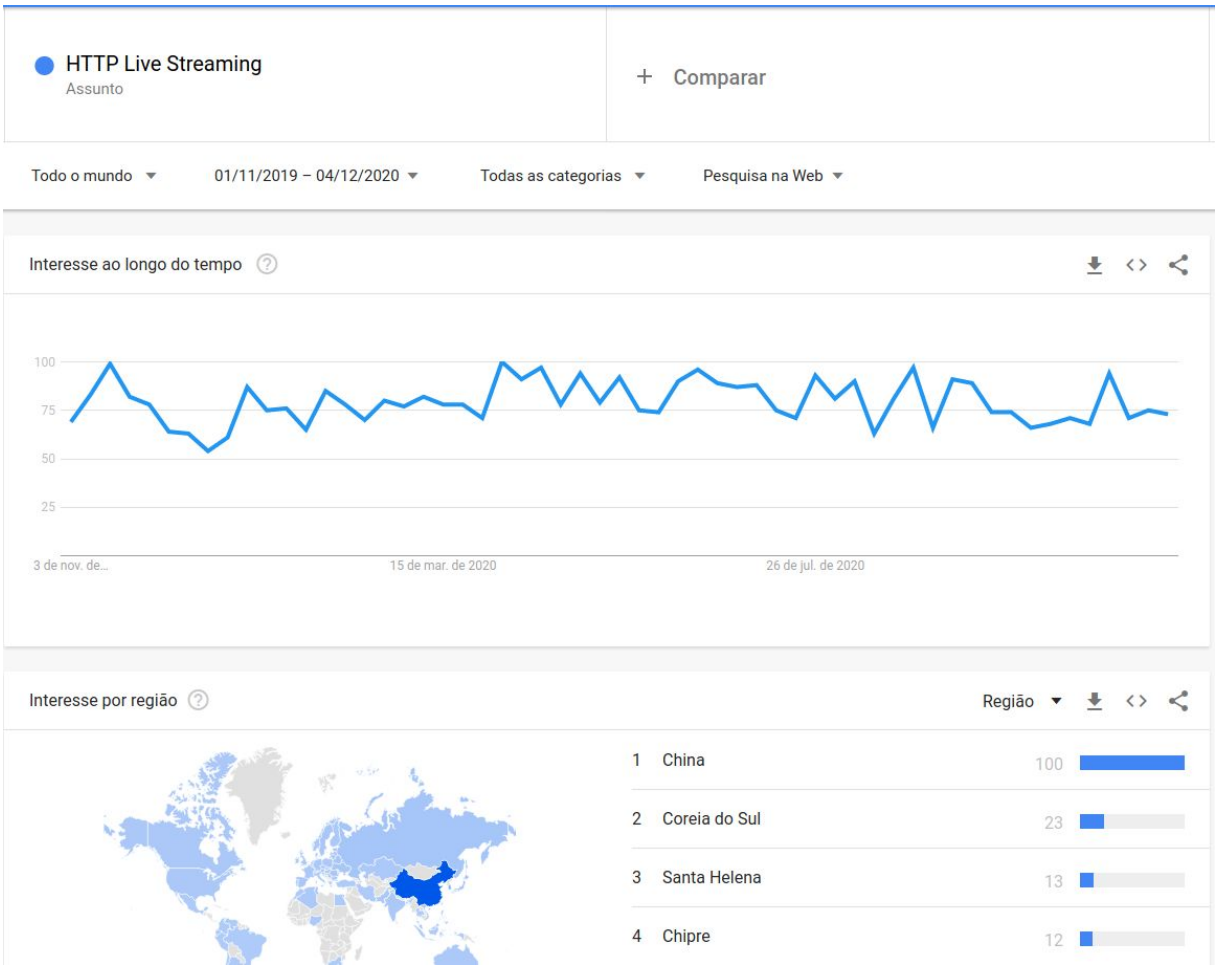
Para visualização mais prática e intuitiva, use o comando: “pm2 imonit” no terminal. Assim pode-se ver os servidores, o processamento e a quantidade de memória utilizada no momento em que for enviado alguma requisição para o servidor NGINX.

Para melhor visualização sobre as requisições, o software utilizado para teste foi o Insomnia, que por sua vez nos permite enviar requisições *HTTP*. Porém, para simples teste de qualquer servidor, é necessário saber o *IP* privado da máquina em uso, pois o servidor de balanceio automático é hospedado neste *IP*.

3.1.1 Mercado e público-alvo

Para exposição do mercado diretamente beneficiado pela solução proposta, vale destacar a adoção da plataforma NGINX como *streaming* de vídeo largamente utilizado a partir de janeiro de 2020 (NGINX, 2020). Este momento marca o crescente da pandemia provocada pelo COVID 19, iniciada em Wuhan, na China, conforme pode ser observado em pesquisa realizada pelo Google Trends exposta a seguir na Figura 6

Figura 6: Gráfico do *Google Trends* sobre *HTTP Live Streaming*



Fonte: https://trends.google.com.br/trends/explore?q=%2Fm%2F06w2h79

Este mercado alvo do software NGINX que aponta para *streaming* de vídeo pode ser validado nas palavras da própria companhia:

> “With schools around the world shutting their doors and rapidly implementing distance learning, we expect the use of streaming video to increase exponentially over the coming weeks and months.” (NGINX, 2020)

Deste fato surge a ideia apresentada na presente proposta de homologar tal solução em ambientes de baixa complexidade, visando, sobretudo, os aspectos de tolerância à falhas, otimização de desempenho e integridade do ambiente, conforme explorado no capítulo introdutório deste Trabalho de Curso.

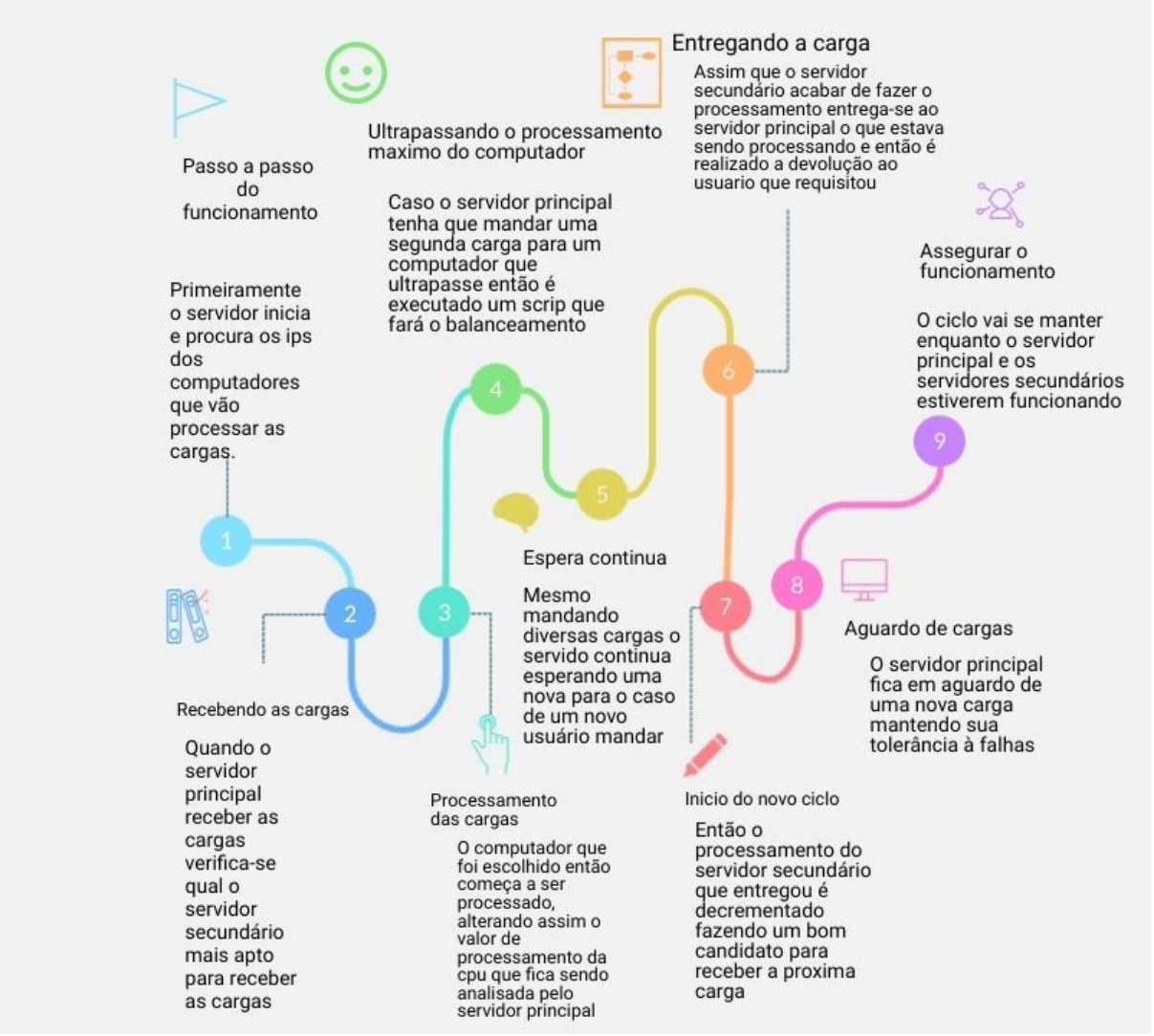
Nosso principal público-alvo são empresas ou instituições que demandam da necessidade de um sistema estável para execução das aplicações de larga escala de processamento distribuído. Este cenário foi, sobretudo, agravado pelas circunstâncias pandêmicas que aceleraram a busca por distribuição, suplantando todos os desafios inerentes e dependem, no momento atual, de tecnologia para continuidade de serviços.

3.1.2 Engenharia de software

Para melhor entendimento deste Trabalho, os diagramas a seguir abordam detalhes mais específicos sobre as funcionalidades, casos de uso, dados, e estados.

3.1.2.1 Infográfico

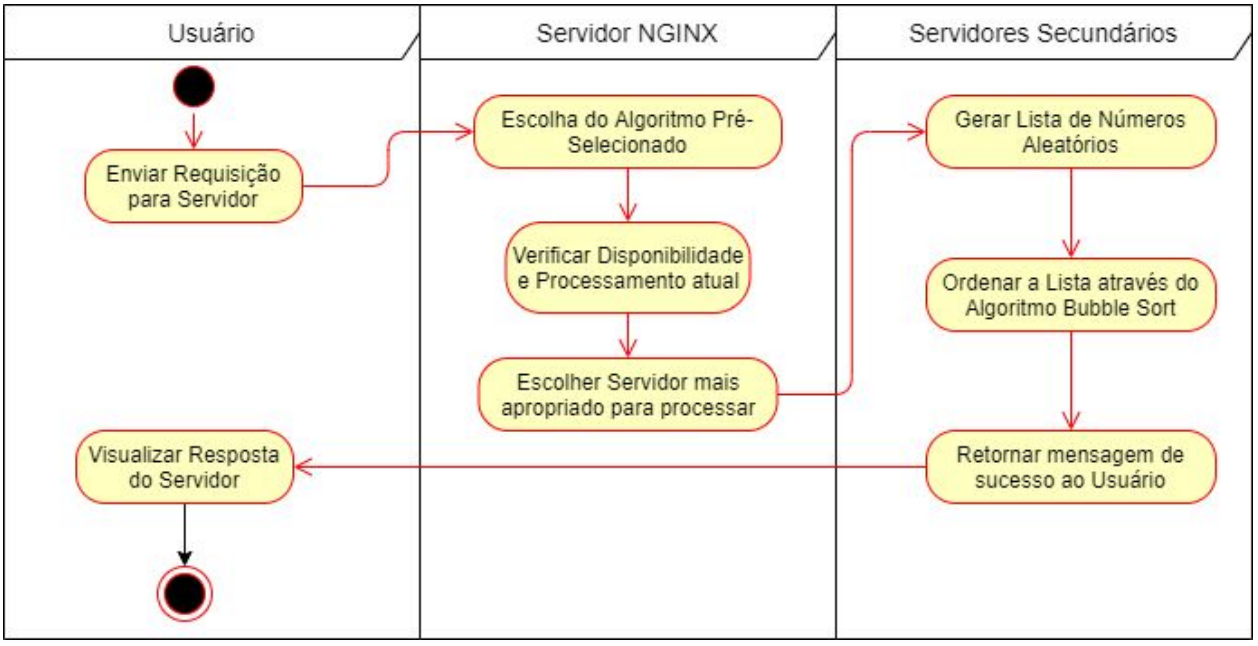
Infográfico 1 - Explicação detalhada sobre o ambiente



Fonte: Autor

3.1.2.2 Diagrama de atividades

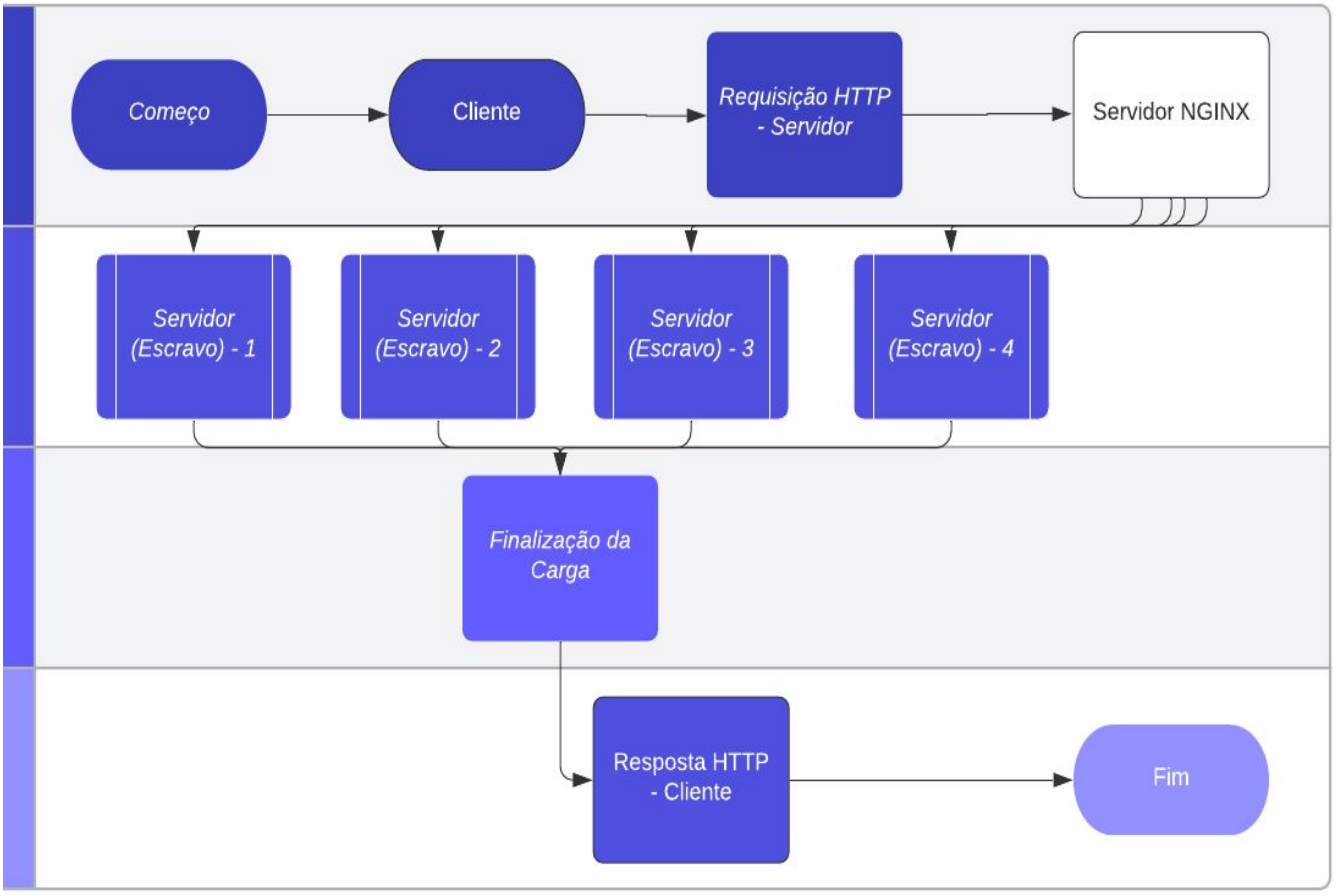
Diagrama 1 - Fluxo de atividades da aplicação



Fonte: Autor

3.1.2.3 Diagrama de estados

Diagrama 2 - Funcionamento da Requisição até o Servidor



Fonte: Autor

3.1.3 Tecnologias utilizadas

Neste trabalho de curso, foram utilizadas quatro principais tecnologias destacadas a partir do enfoque de desenvolvimento do ambiente. São estas: a linguagem de programação *JavaScript*, devido a facilidade de criação de servidores; o

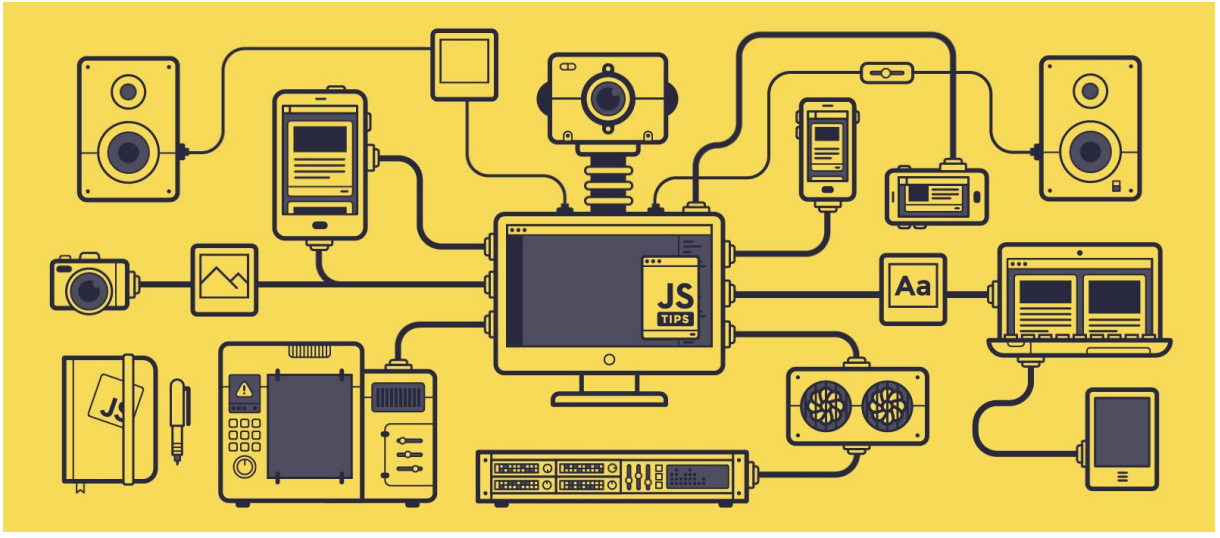
runtime Node.Js, em decorrência de sua independência multiplataforma; a plataforma *NGINX*, devido à robustez e simplicidade de utilização; e a biblioteca *PM2*, para maior integração e monitoramento dos servidores.

3.1.3.1 *JavaScript*

A linguagem *JavaScript*, foi criada na década de 90 com o propósito de permitir a codificação de comportamento dinâmico e interativo no *front-end* das páginas Web. No entanto, desde a criação o JavaScript tem sido cada vez mais utilizado em contextos diversos. Com esta ampliação do contexto de uso da linguagem, diferentes tipos e estilos de código-fonte surgiram e devido à flexibilidade de desenvolvimento da linguagem, padrões e boas práticas de desenvolvimento foram sendo esculpidos pela experiência própria de cada desenvolvedor, sem uma norma declarada oficialmente (ROCHA; FELIPE, 2017).

A figura 7 (sete) mostra uma visão geral sobre a linguagem, destacando pontos onde são mais utilizados no desenvolvimento em multiplataformas.

Figura 7: Visão dos dispositivos em que a linguagem pode ser implementada.



Fonte: https://guildadocodigo.atelie.software/onde-estamos-no-javascript-e91e620ae25a

3.1.2.2 Node.js

Uma definição para o *Node.js* é ser um ambiente de execução *JavaScript server-side*. Desta forma o *Node.js* é feito para criar aplicações *JavaScript* para rodar como uma aplicação *standalone*, ou seja, não dependendo de uma aplicação *front-end*.

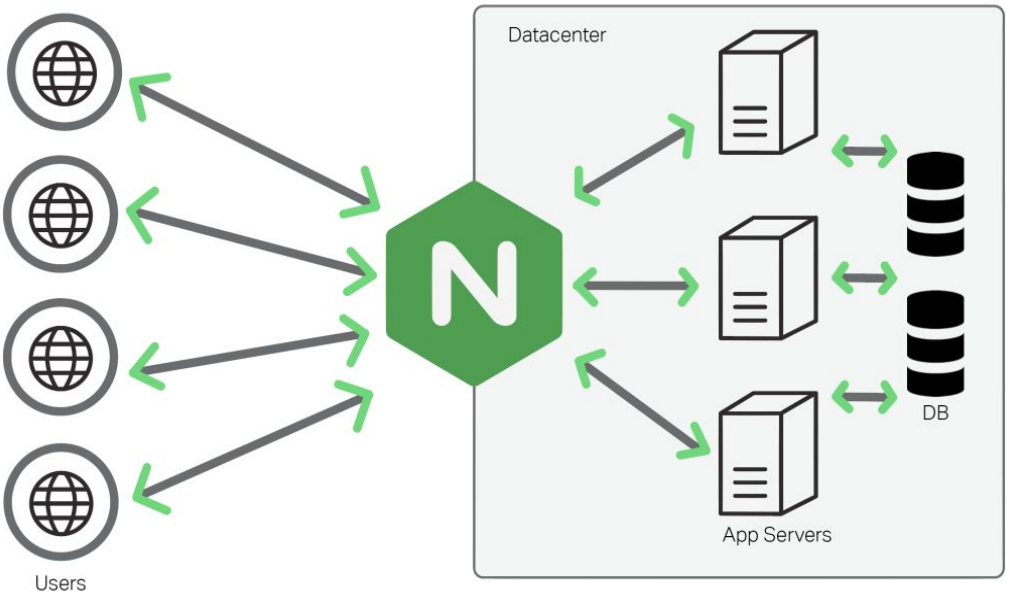
3.1.3.3 *NGINX*

NGINX é a tecnologia utilizada como servidor, responsável pela administração e balanceio automático das cargas em todos os servidores subsequentes. Tal adoção já foi explorada no subitem 3.1.1 referente a Mercado e Público Alvo, acima neste Trabalho. Sobre a história do NGINX, segundo Molinari (2016):

> "O NGINX, que se pronuncia engine X, é um servidor web criado em 2004 para competir com o Apache. A motivação para a sua criação foi otimizar a quantidade de memória e CPU gastas para servir páginas web." Molinari (2016)

Seguindo a linha de pensamento de Molinari (2016), a estrutura formada pelo NGINX no produto segue a mesma ideia da figura a seguir:

Figura 8 - Arquitetura do NGINX



Fonte:https://medium.com/@samual.r.moot/nginx-web-server-and-reverse-proxy-server-c25f68a61d92

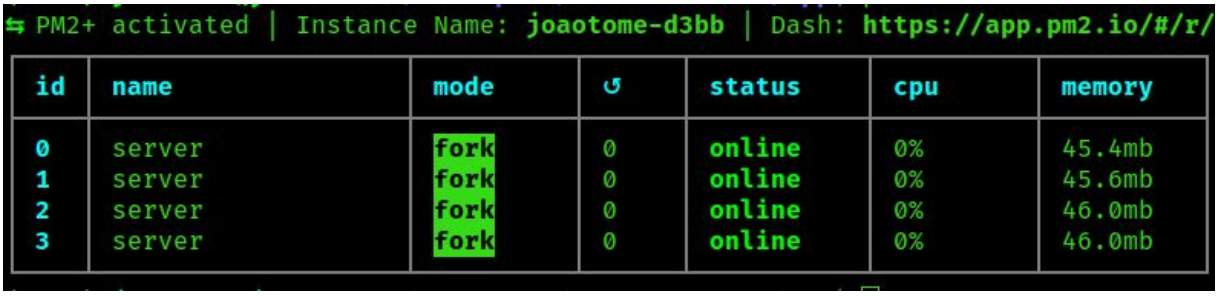
Na Figura 8 (oito) foi abordado um sistema web, onde o NGINX é responsável pelo gerenciamento das requisições de chegada e encaminhá-las para os servidores,

ressaltando a escolha do algoritmo na configuração do *script* de inicialização do NGINX.

3.1.3.4 *PM2*

O PM2 é o gerenciador de processos que ajuda a manter as aplicações online e possibilita a observação dos recursos gastos. Durante o processo de implementação do produto, a biblioteca tornou-se essencial na visualização de informações sobre os servidores de forma facilitada e através de gráficos, como mostrado na Figura 9.

Figura 9 - Representação visual da biblioteca



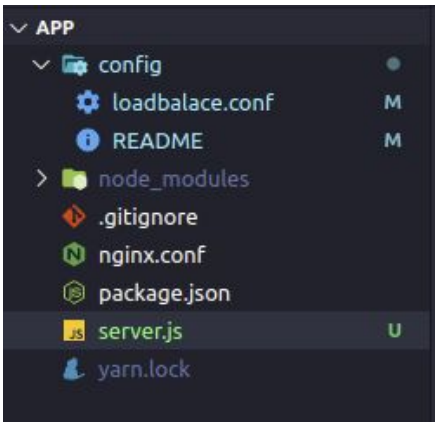
Fonte: Autor

A utilização da biblioteca *PM2 possibilitou* o uso de forma simplificada no *start-up* dos servidores que processam as cargas, nas quais foram enviadas do servidor principal da aplicação.

3.1.4 Funcionalidades do produto

Nesta parte do trabalho, serão abordadas as principais funcionalidades do produto, iniciando com a estrutura de pastas, como expõe a Figura 10.

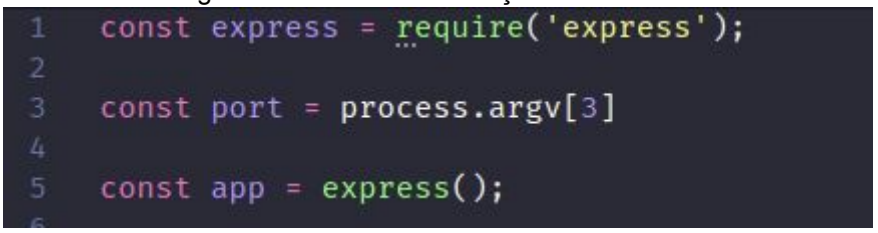
Figura 10: Estrutura de Pastas



Fonte: Autor

Em seguida, será exposto o *script* principal da aplicação. Neste ponto, a abordagem de algumas linhas de código serão mostradas como componentes importantes para a execução e testes do produto. A partir deste trecho, será apresentado a implementação do Trabalho. Visto que, na figura 11, é feita a criação do servidor através da biblioteca Express. Como mostrado a seguir na Figura 11:

Figura 11: Chamada de função do servidor

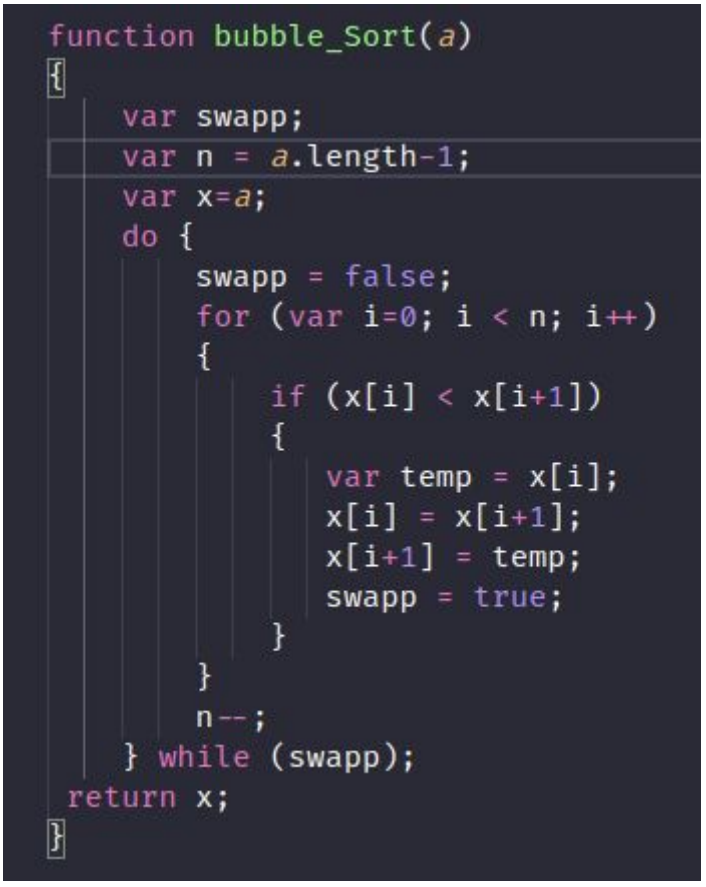


Fonte: Autor

A primeira linha, é chamada a biblioteca do Node.Js, para criação de servidores, a segunda linha é utilizada somente para captação da porta em que o servidor será hospedado. E por fim, a criação do servidor é realizada.

Após a implementação do servidor for concluída, será necessário criar a carga que possibilitará os testes do servidor, neste caso, foi utilizado o algoritmo de Bubble Sort, como mostrado na Figura 12.

Figura 12: Criação do Algoritmo Bubble Sort



```
function bubble_Sort(a)
{
    var swapp;
    var n = a.length-1;
    var x=a;
    do {
        swapp = false;
        for (var i=0; i < n; i++)
        {
            if (x[i] < x[i+1])
            {
                var temp = x[i];
                x[i] = x[i+1];
                x[i+1] = temp;
                swapp = true;
            }
        }
        n--;
    } while (swapp);
 return x;
}
```

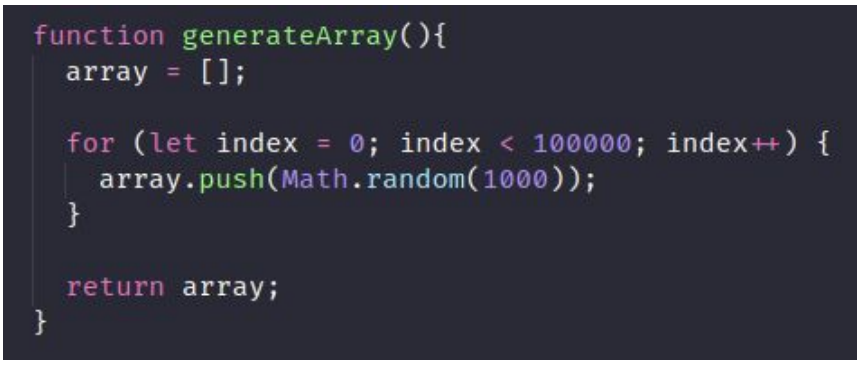
Fonte: Autor

Este algoritmo descreve o uso da carga principal da solução proposta, invocando o comportamento do ambiente e a forma de manipulação da capacidade máxima de processamento do servidor. Pode-se destacar a identificação de conceitos inerentes à disciplina de Análise e Complexidade de Algoritmos, Segundo (Martins, 2014):

> "Vários critérios podem ser utilizados para escolher o algoritmo, mas vai depender das pretensões de utilização do algoritmo. Pode-se selecionar o algoritmo somente para um experimento, ou será um programa de grande utilização, ou será utilizado poucas vezes e será descartado, ou ainda, terá aplicações futuras que podem demandar alterações no código. Para cada uma das respostas anteriores, pode-se pensar em uma solução diferente. Calcular o tempo de execução e o espaço exigido por um algoritmo para uma determinada entrada de dados é um estudo da complexidade de algoritmos. " (Martins, 2014)

Sobre o algoritmo Bubble Sort, é imprescindível uma lista de entrada, como é mostrado na Figura 13:

Figura 13 - Geração da Lista de Forma Aleatória



```
function generateArray(){
  array = [];

  for (let index = 0; index < 100000; index++) {
    array.push(Math.random(1000));
  }

  return array;
}
```

Fonte: Script gerado do Trabalho (2020)

Observa-se a utilização do algoritmo Bubble Sort. Como o algoritmo pode ser percebida a criação de uma lista vazia e, logo em seguida, a lista é usada para armazenamento de números (cem mil números expostos de forma aleatória de zero a mil). A saída acontece na terceira linha, onde a lista é totalmente ordenada pelo algoritmo Bubble Sort, conforme expõe a Figura 14.

Figura 14 - Geração da Rota do Servidor

```
app.get('/',(req,res) ⇒ {
  array = generateArray();

  bubble_Sort(array);

  return res.status(200).send("Task done!")
})

app.listen(port, () ⇒ console.log(`Server rodando na porta ${port}`));
```

Fonte: Script gerado pelo produto (2020)

Inicialmente, o servidor não possui um serviço de rotas geradas automaticamente. Assim, será necessária a criação de uma rota com endereço estático. Caso a rota seja requisitada, será gerada uma lista não ordenada e será executado o algoritmo *Bubble Sort* sobre a lista. Em seguida, ocorrerá o retorno da mensagem de sucesso.

Na última linha, será executada a instância do servidor, juntamente com a porta que foi definida no começo do *script*.

3.1.5 Homologação do MVP

Para a checagem da solução apresentada foram utilizados diversos computadores para a conferência do desempenho. É importante observar que todos dispositivos testados encontravam-se na mesma rede local. Esta ressalva pode ser validada através das métricas de desempenho como velocidade de transmissão, além da confiabilidade e disponibilidade dos servidores. Segundo KRIŽANIĆ (2010): *“to measure the performance of a computer system, you need at least two tools—a tool to load the system (load generator) and a tool to measure the results (monitor)*”.

3.1.6 Comercialização do produto

Atualmente a solução encontra-se finalizada podendo ser utilizada por qualquer empresa que demande maior poder de processamento. Em um primeiro momento, entretanto, pretende-se disponibilizar o produto em ambientes acadêmicos de forma a servirem de incentivo a criação de ambientes distribuídos *open source*, além da possibilidade de auxiliar na abordagem de disciplinas específicas da área de Ciência da Computação.

A implementação da solução para aplicações de *streaming* de vídeo, inspiração inicial da ferramenta NGINX citada (conforme descrito anteriormente no subitem 3.1.1 (Mercado e Público Alvo), também é uma realidade para a qual o ambiente está pronto. Assim, podendo ser utilizado para o apoio de ambientes educacionais ou outros, dadas as circunstâncias atuais e a dependência conhecida de tecnologia estável, simples e de alto poder de desempenho.

3.1.7 Produtos correlatos

Existem alguns produtos que se assemelham sobre o servidor de balanceamento automático, como:

- *HA Proxy*
    - Este software, foi criado em 2001, e apresenta-se de maneira livre, totalmente open-source, seu funcionamento é similarmente igual ao do *NGINX*, ambos são balanceadores de carga, e abrangem a maior parte das requisições via *TCP/IP* ou *HTTPS*.
- *http-proxy*
    - É uma biblioteca criada principalmente para ajudar na manipulação do *proxy*, e pode ser configurado como balanceador de carga e como *proxy* reverso, além de suportar *Web Sockets*.

Entretanto, todos os sistemas pesquisados mostraram-se obsoletas, além de dependentes de plataformas proprietárias, seja de Sistemas Operacionais ou de Aplicações.

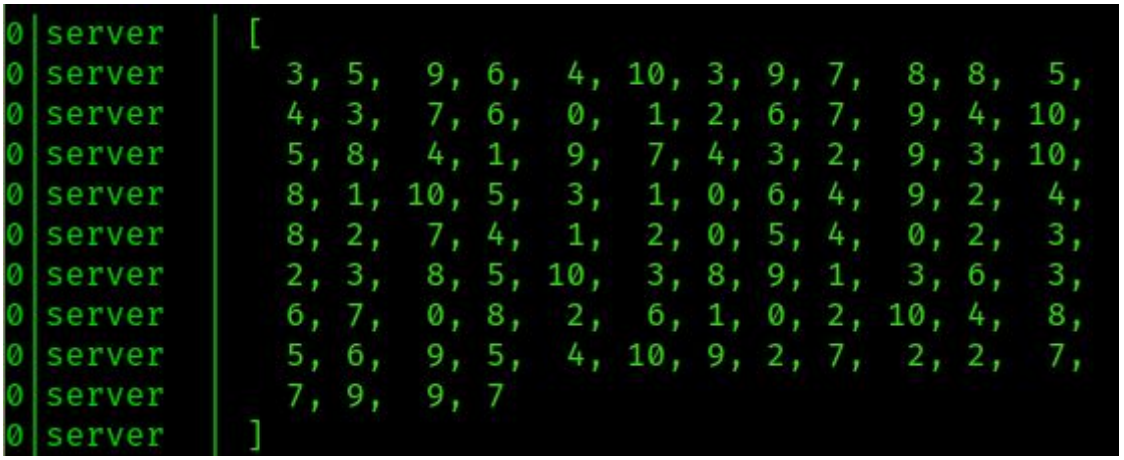
3.2 Área de trabalho/abrangência

A solução pode ser amplamente utilizada, estando no escopo ambientes baseados em *WEB* e arquitetura *TCP/IP*. Não há problemas com restrições geográficas.

3.3 Análise de dados/Uso de arquivos de entrada e saída

O MVP foi rodado a partir de uma carga (arquivo de entrada) gerada pela ordenação de listas promovida pelo algoritmo Bubble Sort, conforme mostrado a seguir, na Figura 15.

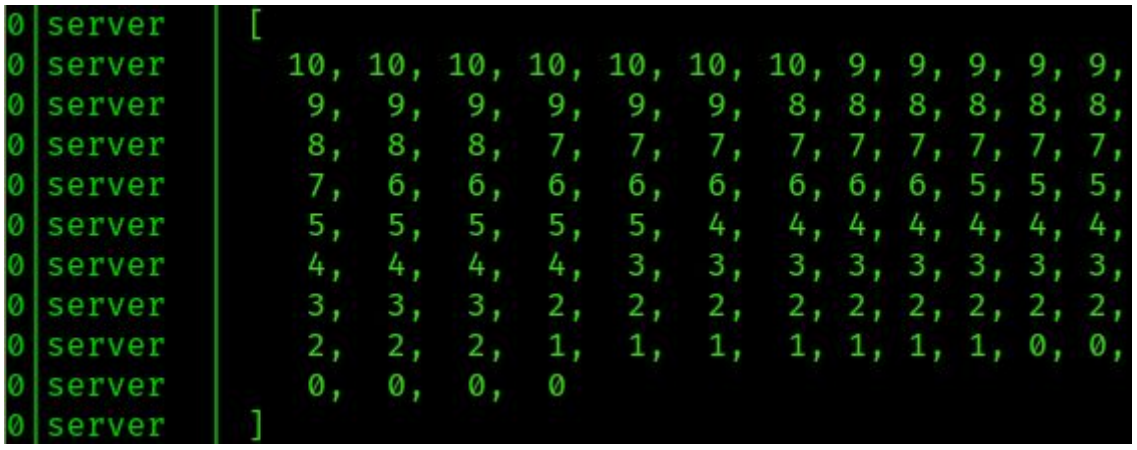
Figura 15: Os números randômicos foram gerados



Fonte: Autor

Quanto à geração de dados na saída, a resposta do sistema é apresentada como um texto de finalização da tarefa, conforme apresentado na Figura 16

Figura 16: Demonstrando a saída com a lista totalmente ordenados



Fonte: Autor

4 RESULTADOS

Nesta seção serão apresentados os resultados obtidos pelo produto deste Trabalho de Curso.

A requisição a partir do método *GET* sobre as listas utilizadas como Carga de Trabalho expõe a forma de acesso e recebimento das interações realizadas entre o servidor principal e os servidores clientes. Vale destacar a adoção do algoritmo *Round Robin* para atendimento das solicitações, conforme exposto no subitem 2.5 Algoritmos utilizados.

Os resultados de tempo de resposta a partir do algoritmo *Round Robin* comparados ao uso do algoritmo *Least Connection* podem ser analisados a partir da Tabela 2. Relevante perceber a pequena alteração numérica, a partir da primeira casa decimal (em milissegundos) nos resultados decorrente da alteração do algoritmo.

A coluna "número de cargas" representa o número de *REQUESTs* realizados para obtenção dos resultados, explicitando que o tempo de resposta apresentado é decorrente de uma média de 5 repetições para cada interação. Este apuro aproxima o resultado final do que seria obtido para um maior fluxo de carga.

A configuração do computador utilizado para a realização dos testes e prototipação do ambiente foi:

- *Processador Intel® Core™ i5-7400*
- *2x Pente de Memória 4GB DDR4 2400MHz Crucial*

- *HD Toshiba 1TB*

Em seguida, estão expostas as tabelas com os resultados obtidos e citados durante o processo de testes, Tabela 1 e Tabela 2.

- Tabela 1 - Algoritmo Round Robin

Nº Cargas	**Servidores (Número)**	**Requisição (Tipo)**	**Tamanho da Lista**	**Número Máximo (Aleatório)**	**Média Tempo de Resposta de Execução**
1	4	*GET*	100000	1000	24.1 ms
2	4	*GET*	100000	1000	23.7 ms
3	4	*GET*	100000	1000	24,5 ms
4	4	*GET*	100000	1000	28.9 ms
5	4	*GET*	100000	1000	30.22 ms

- Tabela 2 - Algoritmo *Least Connection*

Nº Cargas	**Servidores (Número)**	**Requisição (Tipo)**	**Tamanho da Lista**	**Número Máximo (Aleatório)**	**Média Tempo de Resposta de Execução**
1	4	*GET*	100000	1000	24.2 ms
2	4	*GET*	100000	1000	23.8 ms
3	4	*GET*	100000	1000	27.3 ms
4	4	*GET*	100000	1000	28.45 ms
5	4	*GET*	100000	1000	31.4 ms

5 DISCUSSÃO

A literatura especializada na área de Sistemas Distribuídos, conforme Tanenbaum (2015), Coulouris (2011) aponta, como características inerentes ao conceito, a transparência, a tolerância a falhas e a escalabilidade. Entretanto, das soluções, em geral, encontradas no mercado, percebe-se a dificuldade de manter-se vigentes tais características. Mesmo as mais conceituadas empresas do ramo, como as citadas na introdução deste trabalho, já apresentaram falhas na disponibilidade de seus serviços, como em 2018, quando o Google® ficou fora por 1 turno, ou mesmo o Youtube® em novembro do corrente ano.

Buscou-se abordar neste Trabalho de Curso a implementação de dada solução Distribuída que, de forma simples e com controle de desempenho, fosse capaz de prover as características mínimas que assegurem a confirmação dos estudos realizados na área. Muir (2004) aponta que estes desafios estão ligados ao balanceamento da carga e tolerância à falhas.

A NGINX, conforme pode-se observar nos resultados apresentados, mostrou-se alinhada com tais perspectivas. O balanceamento de carga e a simulação da queda de servidores foram realizadas com sucesso nos testes apresentados.

Outra prerrogativa diretamente ligada ao desempenho do ambiente está relacionada à política de escalonamento para atendimento das requisições recebidas pelo ambiente distribuídos, conforme preconiza Tanenbaum (2015). A NGINX confirma a variabilidade dos tempos de resposta obtidos quando são alternados ambos algoritmos citados.

É válido citar, entretanto, que tais resultados são negligenciáveis, no que compete especificamente aos algoritmos escalonáveis, exceto quando se imputam pesos para cada requisição. Em última análise nota-se que o algoritmo “*Round Robin”* tem um desempenho ligeiramente superior quando comparado com *“Least Connection*”, conforme esperado na literatura sobre o assunto. Segundo (Coulouris,2013):

> "Escalabilidade: um sistema distribuído é considerado escalável se o custo da adição de um usuário for um valor constante, em termos dos recursos que devem ser adicionados. Os algoritmos usados para acessar dados compartilhados devem evitar gargalos de desempenho e os dados devem ser estruturados hierarquicamente para se obter melhores tempos de acessos. Os dados acessados frequentemente podem ser replicados." (Coulouris, 2013).

Ainda sobre a capacidade da aplicação ser escalável, o algoritmo de *Least Connection* possui um ponto de vantagem sobre o *Round Robin.* Tal vantagem reside no fato de que ao receber uma grande demanda de requisições, o *Least Connection* tende a escolher o que apresenta um menor volume de conexões, o que representa maior expectativa de velocidade.

Neste ponto, vale também explicitar que tal escolha realizada pelo *"Least Connection*" implica, entretanto, em maior tempo dedicado à verificação da disponibilidade das conexões. Este fato acaba contabilizando e justificando tempo médio ligeiramente maior observado, conforme mostrado nas Tabelas 1 e 2 acima expostas.

Confrontando os resultados apresentados, pode-se concluir que os tempos decorridos da escolha de um ou outro algoritmo fica apenas à cargo da preferência do usuário ou imposição da aplicação. Caso o sistema esteja prezando por mais velocidade de resposta, o algoritmo "*Round Robin"* é o mais indicado; em contrapartida, uma solução mais escalável aponta para o algoritmo "*Least Connection"* como o mais indicado.

6 CONCLUSÃO

Este Trabalho de Curso apresentou o desenvolvimento de um ambiente de baixa complexidade para sistemas distribuídos de alta performance, baseado em NGINX. Para cumprimenrto dos requisitos necessários à implemtação foi levantado um Servidor de Nodes utilizando a biblioteca *Express* com o *NodeJS*, um Servidor principal para o balanceamento automático de requisições para os servidores filhos, a visualização em tempo real de processamento da carga através da biblioteca *PM2* e um relatório de processamento gerado pelos Nodes através da referida biblioteca *PM2.*

Outras etapas envolveram os testes realizados para o balanceamento de cargas com foco não somente na velocidade de entrega e na escalabilidade do sistema, mas também, na simplicidade, consistência e tolerância à falhas na camada de aplicação.

O projeto apresentou certas dificuldades para seu término. Em alguns ambientes, como o institucional, o teste não foi possível devido a instalação de recursos necessários estar indisponível, bem como a política de segurança não permitir. No ambiente residencial, dispôs da dificuldade de realizar os testes devido a carência de recursos computacionais. Vale ressaltar, também, que durante o cenário de pandemia, as visitas ficaram mais complicadas de serem executadas. Entretanto, os testes foram realizados com diversos computadores em rede local residencial.

Durante a finalização deste trabalho acadêmico surgiram possibilidades de novos projetos, como por exemplo, o aplicativo educacional da matéria de sistemas distribuídos ministrada para o Curso de Ciência da Computação da Instituição de Ensino para a qual esta proposta foi apresentada. A criação do ambiente mostrou-se simples a ponto de elucidar a teoria abordada, além de não dispor de recursos de grande porte.

Outra proposta de trabalho futuro será realizar o MVP a partir de *streaming* de vídeo, conforme tendência observada através de análise no Google Trends já apresentado no subitem 3.1.1 referente ao Mercado foco da ferramenta. Este Trabalho abordou a implementação de tolerância a falhas na camada de aplicação. Em nuvem muitas implementações são baseadas em ambientes de três camadas. Como Trabalho futuro serão definidas mais camadas na homologação do *failover.*

7 REFERÊNCIAS

BRIN, Sergey; PAGE, Lawrence. The anatomy of a large-scale hypertextual web search engine. 1998.

Buyya, R., 1999. High Performance Cluster Computing: Architecture and Systems, vol 1. pag 45, New Jersey: Prentice Hall.

CASAVANT, T. L.; KUHL, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Transactions on Software Engineering, 1988. CASE, R. P., PADGES A. Architecture of the IBM System/370. Communications of ACM, v.21, n.1, p. 73-96, jan 1978.

CHANG, F. et al. Bigtable: a distributed storage system for structured data. 2006. Disponível em: http://static. googleusercontent.com/media/research.google.com/pt-BR// archive/bigtable-osdi06.pdf.

Coulouris, G., Dollimore, J. & Kindberg, T., 2007. Sistemas Distribuídos: Conceitos e Projeto. 4ª ed. Bookman, Porto Alegre.

Escalonamento Round-Robin. Disponível em: <http://www.ime.usp.br./>

Jájá, J., 1992. An Introduction to Parallel Algorithms. Redwood City, USA: Addison Wesley Longman Publishing Co.

JAIN, Raj. The art of computer systems performance analysis. john wiley & sons, 2008.

KRIŽANIĆ, J. et al. Load testing and performance monitoring tools in use with AJAX based web applications. In: The 33rd International Convention MIPRO. IEEE, 2010. p. 428-434.

KROENKE, DAVID. Sistemas de informação gerenciais. Saraiva Educação SA, 2017.

LIMA, Leandro; PETRICA, Eder. Protocolo HTTP.

MACHADO, F.B.; MAIA, L.P. Arquitetura de sistemas operacionais. São Paulo: LTC, 2007.

MARTINS, Walteno. Apostila de Análise de Algoritmos, 2014 e Minas Gerais, Universidade do Estado de Minas Gerais Campus de Ituiutaba, 2014 ,p 2.

MCGUIRE, Jacob M. Highly scalable least connections load balancing. U.S. Patent n. 6,996,615, 7 fev. 2006.

MOLINARI, Willian. Desconstruindo a Web: As tecnologias por trás de uma requisição. 1. ed. atual. São Paulo: Casa do Código, 2016. 255 p. v. 1. ISBN 978-85-5519-210-4. Disponível em: https://www.casadocodigo.com.br/products/livro-desconstruindo-web. Acesso em: 2 dez. 2020.

MUIR, Steve. The Seven Deadly Sins of Distributed Systems. In: WORLDS. 2004.

NERY, Bruno Rodrigues; DE MELLO, Rodrigo Fernandes; DE CARVALHO, André CPLF. Escalonamento de processos utilizando técnicas de ACO. XXV Concurso de Trabalhos de Iniciação Científica, p. 1-15, 2006.

MAURO,Tony; Enabling Video Streaming for Remote Learning with NGINX and NGINX Plus; Disponível em: <https://www.nginx.com/blog/video-streaming-for-remote-learning-with-nginx/>

Parallella | Supercomputing for Everyone. Disponível em: < http://www.parallella.org/>.

PRISCO, R.; LAMPSON, B.; LYNCH, N. Theoretical computer science. Revisiting the Paxos algorithm, 2000.

RIBEIRO, J.A.S.; MUNIZ, J.D. Implantação do curso de Linux em escolas públicas: projeto conhecendo o Linux. 2012. Monografia (Bacharelado em Sistemas de Informação) - Faculdade Anhanguera de Belo Horizonte, Belo Horizonte, 2012.

Ribeiro, U., 2005. Sistemas Distribuídos: Desenvolvendo Aplicações de Alta Performance no Linux. Editora Axcel Books do Brasil.

SHUSTEK, L. J. Analysis and Performance of Computer Instruction Sets. PhD. Thesis, Stanford University, mai 1978. Disponível em: < http://www.slac.stanford.edu/cgiwrap/getdoc/slac-r-205.pdf>.

SILVA, D.; SOBRAL, L.; DE OLIVEIRA BARROS, Márcio. Um Estudo em Larga Escala sobre a Estrutura do Código-fonte de Pacotes JavaScript. Graduate final Report, 2017, Applied Computer School of Center of Exact Sciences and Technology, Unirio. Rio de Janeiro, 2017

SOUZA, A. T. Sistemas Distribuídos – Sistemas distribuídos e Paralelos. FASUL, 2011. Disponível em: <http://www.slideshare.net/adrianots/2-sd-conceitossistemas-distribuidos-eparalelos>.

STERLING, Thomas. An Introduction to PC Clusters for High Performance Computing. International Journal of High Performance Computing Applications. v.15, n.2, p. 92-101, mai 2001.

Tanenbaum, A. S., 2003. Sistemas Operacionais Modernos. 2ª ed., Prentice Hall, São Paulo.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, 2ed. MIT Press e McGraw-Hill, 2001. ISBN 0-262-03293-7. Problem 2-2, pg.40.