Dedico este trabalho a minha família

a minha futura esposa e a Deus

AGRADECIMENTOS

Primeiramente a Deus, por me capacitar e me dar condições de conquistar mais uma etapa na minha vida, toda honra e glória é dada a ele. “Tudo isso é para o bem de vocês, para que a graça, que está alcançando um número cada vez maior de pessoas, faça que transbordem as ações de graças para a glória de Deus. ”

Gostaria também de dedicar esse trabalho a minha família, uma vez que sem o apoio deles nada seria possível. Todo o esforço feito durante os anos de faculdade para que eu me mantivesse matriculado e provendo-me todas as ferramentas necessárias para trilhar um grande caminho. A família é o alicerce principal em nossas vidas, neles nos mantemos firmes, encontramos forças para prosseguir e suporte nos momentos difíceis. Ao meu irmão Kayke Moreno que sempre me suportou em tudo sem medir esforços e fez tudo se tornar possível, a minha mãe Roseangela Maria que sempre foi uma guerreira, ao meu padrasto Jorge Jóse pelas horas de estudos e ensinamentos matemáticos, a minha irmã Jully Rodrigues pelo incentivo e influência por fazer Ciência da Computação, ao meu pai Júlio Cesar que com todo os elogios e alegria me ajudaram a me manter firme e convicto de que esse era o caminho e a minha avó Lurdes que me ajudou a ter outras conquistas na minha vida o meu muito obrigado por tudo, este trabalho tem um pouco de vocês em cada linha.

Gostaria de separar um parágrafo de agradecimento a minha futura esposa, Ana Carolina, mulher que Deus colocou em minha vida e que eu sou grato por tudo, sem você, muitos momentos eu não teria sabedoria para superar, você mesmo saber foi um espelho e referência de dedicação e determinação, sempre vi em você a pessoa que eu deveria me tornar para obter conquistas significativas em minha vida, a você o meu muito obrigado também.

Consagre ao Senhor tudo o que você faz, e os seus planos serão bem-sucedidos. Provérbios 16:3

Esforcem-se para ter uma vida tranquila, cuidar dos seus próprios negócios e trabalhar com as próprias mãos, como nós os instruímos; a fim de que andem decentemente aos olhos dos que são de fora e não dependam de ninguém. 1 Tessalonicenses 4:11-12

RESUMO

Conforme o crescimento da internet, mais pessoas podem estar conectadas, compartilhando, visualizando dados online. Nos dias atuais, o desenvolvimento de sistemas requer muito mais responsabilidade e planejamento, pois não capitaliza apenas alguns mil usuários por mês ou por trimestre. Atualmente são milhões de acessos simultâneos, de compartilhamentos, visualizações, um verdadeiro caos nos tráfegos de dados.

Por conseguinte é necessário analisar como lidar com um sistema que suporta tanta mudança sem afetar sua estrutura de arquitetura, mantendo a manutenção simples quando trata-se milhares de dados. O microserviço tem surgido comO uma solução que envolve não somente performance, todavia, manutenção, simplicidade do projeto, o famosos “as a service” tem sido uma solução para melhorar o atendimento a milhares de usuários simultâneos, o que traz escalabilidade e consequentemente performance.

Neste trabalho, abordou-se temas principais com relação à arquitetura Microserviço e Monolítica, avaliou-se a arquitetura de microserviço e como as pequenas entregas, tornam-se eficientes para construir grandes sistemas. Estes, são compostos por diversos sistemas com tecnologias independentes que se comunicam, visando construir um único sistema. Este estudo visa apresentar as características dosmicroserviços, além dos seus benefícios, avaliar sua estrutura arquitural fragmentada e como impactapositivimante na manuntenção e criação dos sistemas de grande e pequeno porte atualmente.

A arquitetura como meio provedor no aumento de performance e resiliência na construção de uma aplicação será questionada, assim como o que é de fato determinante para isso (arquitetura, sistema operacional ou linguagem de programação), e se pequenas decisões impactam tanto na construção de um sistema de alto desempenho.

Este trabalho tem como objetivo explorar os pontos de performance server-side, ou seja, buscar fatores que ajudam a maximizar a capacidade de desempenho de uma aplicação e buscar resultados que possam especificar o que de fato e em que ponto uma decisão pode ser primordial para performance de uma aplicação.

Palavras-chave: microserviço, arquitetura, sistemas, monolítico, performance, desempenho, modularização, resiliência, linguagem de programação, aplicações

ABSTRACT

As long as internet was getting bigger more people became able to connect themselves with each other, sharing and consuming data. Based on that growth, software development has required much more responsability and planning, moreover, it‘s not about build software to cover thousand of people at month but millions of concurrent accesses, sharing and data visualization.

Therefore all internet growth it’s important to manage system which are able to change without affecting in its architecture structure been able to keep simple maintenance. Microservice architecture has come up with a solution that not only involves performance, however, maintenance, simplicity of projection, the famous “as a service” has been a solution to improve service to thousands of concurrent users.

In this work, the main themes related to the Microservice and Monolithic architecture were analyzed, the microservice architecture was evaluated and as the small deliveries, they became efficient to build large systems. These are composed of several systems with independent technologies that communicate, aiming to build a single system. This work aims to present the characteristics of the microservices, in addition to their benefits, to evaluate its fragmented architural structure and how it positively impacts on the maintenance and creation of large and small systems currently.

The primordial definition of the architecture for the increase of performance and resilience of its application will be questioned, as well as what is in fact determinant to obtain performance (architecture, operating system or programming language), and if small decisions impact so much in the construction of a high performance system.

This work aims to explore the server-side performance points and search for results that can specify what actually and at what point a decision may be paramount to an application‘s performance.

Keywords: microservice, architecture, systems, monolithic, performance, performance, modularization, resilience, programming language, applications

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Na era da globalização, em pleno século XXI, o crescimento populacional é enorme e o crescimento tecnológico também cresce exponencialmente. A tecnologia da informação e a telecomunicações caminham juntos há anos, provendo serviços e meios, físicos ou lógicos, para estabelecer uma comunicação rápida, prática e eficaz.

Bem como o crescimento do uso tecnológico as ferramentas que a compõem também apresentam um aumento com passar dos anos.Hoje, a internet se equipara ao que foi a eletricidade na Era Industrial, o que é percebido pela sua enorme capacidade de distribuição da informação e transferência de dados em todo o domínio de atividade humana.

Em 1974, foi publicada uma proposta da arquitetura TCP por Cerf e Kahn, no projeto, “A Protocol for Packet Network Intercommunication”, em tradução livre, “Um protocolo para intercomunicação de pacote de redes” e completado em 1978 pelo protocolo ip, forneceu padrões compatíveis para diferentes sistemas de interconexão de computadores.(Jorga Zahar Editor, , 27, capítulo 1) esse padrão foi responsável por permitir de maneira simples a comunicação entre diferentes servidores pelo o mundo inteiro, por consequência, o desenvolvimento de aplicações tornou-se possível e fácil.Hoje, 38 anos depois a internet já vislumbra novas maneiras de comunicação; em fases finais de testes, o protocolo HTTP2 ou (HTTP) vem para revolucionar a internet como um todo com seu conceito de multiplexação, ou seja, permissãode múltiplas conexões para baixar múltiplos arquivos sem congestionar a requisição feita a um servidor, o que hoje acontece no protocolo HTTP1.1.

O número de usuários da Internet cresceu fortemente durante os últimos 10 anos, possibilitando o compartilhamento de milhões de dados em tempo hábil de resposta. Esta evolução só foi possível devido ao crescente e contínuo aperfeiçoamento das tecnologias nas áreas de redes e de processamento de informações. Segundo o Comitê Gestor da Internet Brasileira, um estudo feito em 2015 mostra que 58% da população brasileira usam internet, o que representa 102 milhões de internautas. (CGI.BR,) Diante deste cenário, é necessário analisar como suportar o volume de acessos, bem como prover velocidade de resposta, com resiliência e de maneira eficiente. Além disso, as aplicações precisam manter-se organizadas, de forma a colaborar para que o desenvolvimento possa ser fácil e ágil, identificando pequenas decisões podem impactar negativamente os sistemas de forma relevante.

Todos esses desafios têm sido debatidos por grandes empresas, a fim de que, ao atingirem seu público, estejam preparadas para suportar situações ao extremo

acesso. O google recentemente lançou a tecnologia AMP(Accelerated Mobile Pages Project), em tradução livre, “Páginas móveis aceleradas” que visa otimizar o carregamento de páginas nas versões móveis, afim de consumir menos banda larga e com maior velocidade. Já usado por grandes empresas, bem como a Globo.com segundo acessos feitos durante a escrita deste trabalho. Empresas como Amazon e Netflix aderem ao microserviço como uma solução em busca de escalabilidade, independência e consequentemente performance, segundo Newman em seu livro *Building Microservices.* (O’Reilly Media, Inc., 2015)

Grandes instituições estão sempre em busca de performance, estudam novas teorias, tecnologias, arquiteturas, investem em novas soluções, aproveitam o que já existe, colaboram, sempre em busca de alguns milésimos de segundos a menos. Essas buscas constantes envolvem diversos setores de melhorias dentro da arquitetura de um sistema, desde a mais detalhista mudança, bem como, otimização de processos e threads em nível de sistema operacional até infraestrutura, com soluções de balanceamento de carga, virtualização de máquinas dedicadas e cacheamento na busca de soluções frontend e backend, estas buscas geralmente estão voltadas na obtenção desempenho e ao mesmo tempo na construção de sistemas eficientes com manutenibilidade conforme o crescimento, ademais mantendo-se simples.

> Mais especificamente, o valor real do microserviço é realizado quando nos concentramos em dois aspectos principais – velocidade e segurança.(IRAKLI NADAREISHVILI,)

A partir dos pontos levantados anteriormente, cabe o questionamento se toda essa busca de performance tem como fator principal apenas a arquitetura escolhida para a composição de um sistema, ou se apenas um modelo de arquitetura componentizado é o suficiente para criar aplicações resilientes e com um bom desempenho.

A motivação acerca desse trabalho se dá não só pelo entusiasmopor tecnologia ou pela imersão vivida nesse “submundo” o qual todos estão vivendo. O estímuloé proveniente da busca incansável por soluções de alta escalabilidade, novas tecnologias, entendê-las, consumi-las e tirar proveito de cada benefício a cerca dessas soluções.

Este trabalho tem como objetivo comparar a performance de aplicações desenhadas com diferentes arquiteturas e analisar o fator preponderante para criação de uma aplicação escalável e resiliente. Também se propõe a apresentaros benefícios e os pontos de atenção de ambas as arquiteturas dentro de cenários de aplicações da realidade atual, traçando um comparativo entre a arquitetura monolítica e microserviço e acompanhando a evolução da busca por uma solução mais eficaz.

> Um microserviço é um componente implementável independentemente de um escopo limitado que ofereça suporte à interoperabilidade através

> da comunicação baseada em mensagens. (IRAKLI NADAREISHVILI, , page 20)

Para a realização dos testes computacionais, foi desenvolvida um estudo de caso para validar o que de fato ajuda na criação de aplicações escaláveis, se são as tecnologias acerca de uma arquitetura, ou a definição de uma arquitetura apenas traz esse benefício. Foram traçados comparativos de performance e de estrutura das aplicações. Analisou-se o que de fato é suficiente para criar aplicações resilientes, como pequenas decisões podem afetar a performance do todo, bem como, a linguagem e técnica de programação podem ser suficiente para obter um sistema de alto desempenho.

Este trabalho está organizado da seguinte forma: no capítulo 2, serão discutidos os fundamentos básicos para comunicação na web; no capítulo 3, será apresentada as características da arquitetura de uma aplicação monolítica e os pontos mais abordados com relação às facilidades e aos problemas ocasionados pelo o crescimento de uma aplicação; o capítulo 4 conceitua a arquitetura microserviço, ressaltando a modularização implementada por diversas empresas que buscam otimização e reuso; no capítulo 5, será mostrado o estudo de caso objeto de estudo deste trabalho; no capítulo 6, serão apresentados e analisados os resultados dos experimentos realizados; por fim, no capítulo 7, serão apresentadas as conclusões deste trabalho.

2. A FUNDAMENTAÇÃO DA COMUNICAÇÃO NA WEB

A arquitetura montada na construção do protocolo HTTP[1] demonstra grande semelhança acerca do microserviço, trata-se de um protocolo de camada de aplicação segundo o modelo OSI. O seu desenvolvimento teve como objetivo flexibilizar todo seu processo de comunicação para comportar diversas necessidades distintas, tais quais: comunicação entre diferentes computadores, padrões na informação do destinatário e remetente da informação para evitar perdas de pacote, definição de regras para facilitar o entendimento de cada segmentação dos dados, além de evitar alto custo na comunicação entre ambas máquinas usando informações que mostram se a conexão já está estabelecida ou não.

Toda requisição feita pelo browser é executado pelo método GET, ou seja, ao acessar uma página web ou uma aplicação mobile é feito o método GET para obter-se as informações. Todavia, o protocolo é extensível a outros métodos, tais quais: GET, POST, PUT,DELETE, OPTIONS,HEAD,TRACE,CONNECT e PATCH. Cada método possui a sua particularidade de uso e depende das necessidades de cada caso.

O protocolo HTTP permite que a comunicação seja feita de diversas maneiras, constantemente é necessário a passagem de parâmetros para URL, que podem conter a descrição de um determinado recurso. Por exemplo, havendo a necessidade de recuperar todas as blusas de uma loja do tamanho G, podemos utilizar uma URL como a seguinte:

/loja-virtual/blusas?tamanho=G

Os métodos HTTP suportam parâmetros, sejam eles: *query parameters* ou *body parameters*. O termo denominado *query parameters* são passados na própria URL, tal qual o exemplo anterior, todavia, o termo denominado *body parameters* é inseridos no corpo da requisição, encapusalado junto a outras informações, usados no método POST.

O protocolo HTTP fornece informações sobre toda a requisição; host - mostra qual foi o DNS utilizado para chegar a este servidor; User-Agent - o meio utilizado para acessar o endereço; Accept - realiza negociação com o servidor a respeito do conteúdo aceito(se é text/html, application/json, application/xml); Accept-language - negocia com o servidor a respeito do idioma usado na resposta; Accept-Encoding - negocia com o servidor o tipo de codificação a ser usada na resposta(GZIP - usado para compactar arquivos html,css, js); Connection - ajusta o tipo de conexão com o servidor (recomendado o uso do keep alive - que mantém a conexão persistente, sem

[1] HyperText Transfer Protocol, em tradução livre, Protocolo de Transferência de HiperTexto

interrupções e reaberturas).

O protocolo HTTP serve como base para uso da tecnologia REST que por sua vez é a base da arquitetura de microserviço. Esta conceituação base será importante para o entendimento de como REST se beneficia dos princípios deste protocolo de forma a tirar o máximo de proveito.

No capítulo a seguir, serviço REST na implementação de comunicação web. Buscou-se explicar a base do serviço REST e expor a importância que tem para o entendimento de como tornou-se prático a integração entre aplicações e serviços distintos.

2.1. A ARQUITETURA REST NA IMPLEMENTAÇÃO DE SERVIÇOS WEB

O REST (Representational State Transfer ou **Transferência de Estado Representativo**, em tradução livre) é uma técnica de desenvolvimento de web services que foca em entrega de serviços web, dado a passagem ou reconhecimento de parâmetros para identificar o recurso buscado. Essa técnica é baseada nos conceitos do protocolo HTTP que é a base para comunicação web, aprópria navegação do usuários em sites é entendida como uma utilização de REST.(Casa do Código,)

O serviço REST é baseado em recursos, que possuem identificadores e endereços próprios. Estes identificadores são referências de um recurso dentro do sistema, muito conhecido como semântica de recursos. Um dos pontos chave é o uso de *media types* para alterar as representações de um mesmo conteúdo, sob perspectivas distintas, pois é possível buscar dados mais genéricos ou detalhados de um recursos com tipos diferentes, como por exemplo: JSON ou XML.

Como toda técnica, o REST tem seus princípios chave para comunicação “as a service” no mundo web, estes princípios serão importantes para o entendimento de como o REST e seus princípios moldam e dão vida a arquitetura de microserviços. Nesta técnica, entende-se que tudo que deve ser identificado necessita ter um ID, que no mundo web, entendemos como URI (Universal Resource Identifier). Uma URI pode ser usado para identificação de qualquer recurso - dar um caminho para um determinado (path) conteúdo. A facilidade de identificar um objeto no sistema, trazer especificações de um recurso, visando ter um esquema único de nomes tanto para a web quanto à comunicação entre máquinas.Outro grande princípio REST é a vinculação entre recursos, conhecido como HATEOAS (Hypermedia As the Engine Of Application State), que possibilita relacionamentos dentro da modelagem de um sistema, permitindo que o consumidor das informações troque de estado através destes links. Já muito difundido no mundo web o conceito de hipermídia é usado em todas as requisições a um website. Ao abrir uma requisição, o browser (chrome, firefox, explorer) lê todo

o arquivo html, ao encontrar hiperlinks para css e js externos, acessa essas rotas, consome os seus conteúdos e volta para renderização da página. Percebe-se que o conceito é o mesmo, numa única requisição a uma rota definida pela sua aplicação, o sistema permite que ao cliente que ao consumir uma informação ele possa obter informações de outros recursos relacionados àquele recurso buscado por ele, como por exemplo:

<cliente>

<enderecos>

<link href="/cliente/1/endereco/1" title="Endereço comercial" />

<link href="/cliente/1/endereco/2" title="Endereço residencial" />

<link href="/cliente/1/endereco/3" title="Endereço alternativo" />

</enderecos>

</cliente>

(Casa do Código, , 41, capítulo 3)

Observa-se que neste exemplo é possível que em uma única requisição para cliente/1 seja feita a leitura e a identificação de todos os seus endereços através dos links utilizados, denominados, HATEOAS.

Outro princípio importante dentro do REST é a comunicação sem estado, pois o REST exige que o estado seja transformado no estado do próprio recurso(objeto), ou mantido por quem solicita, no caso ocliente. Isto significa, que a aplicação não deve guardar o estado da comunicação além de tudo aquilo que está dentro de uma única requisição, a razão pela qual isso é importante, é a escalabilidade, muito ligado a performance. Pensando no número de clientesque podem interagir com uma API REST, seria um impacto grande caso fosse necessário manter o estado de cada cliente. (RICHARDSON, 2007)

Após a explicação fundamental acerca do serviço REST é importante ressaltar que a arquitetura de microserviço se aplica como uma comunicação entre diferentes servidores, em espaço físicos diferente e com troca de dados contínuas. Além disso, tendo em vista que nesse processo é vital que haja integridade e segurança na troca de dados, tão somente é preciso entender o motivo pelo qual os mecanismos de proteção são necessários para que então os ataques sejam evitados. Uma requisição geralmente permeia diversos equipamentos dentro da rede, tais quais: firewalls, switches, proxies etc. Nota-se que nos equipamentos não há a garantia de que são confiáveis, uma vez que nesse caminho pode haver alteração em qualquer um deles. Dois ataques bem conhecidos são: *eavesdrop* e *man-in-the-middle.* No *Eavesdrop,* o atacante intercepta a requisição mas não a altera. Este tipo de interceptação é apenas

para saber o tipo de conteúdo que está sendo transitado ou para ter condições detalhadas para repetir(reproduzir) a requisição - conhecido como replay attack. (Casa do Código, , 191, Capítulo 8)

No *Man-in-the-middle,* o atacante desvia o trajeto da mensagem, com o propósito de manipular o conteúdo para então reenviá-la ao servidor de destino, aparentando ter sido enviado normalmente pelo remetente. (Casa do Código, , 181, Capítulo 8)

Uma maneira de evitar esse tipo de ataque, é o uso de criptografia. A aplicação de uma camada de segurança usando SSL(Secure Sockets Layer) sobre o protocolo HTTP, conhecido também como HTTPS (Hypertext Transfer Protocol over Secure Sockets Layer). O SSL utiliza certificado digital, este certificado é usado para que o cliente confirme a autenticidade do servidor e criptografe as informações usando este certificado. Ao usar os certificados digitais, o servidor fornece um sistema de criptografia baseadas em dois tipos de chaves: pública e privada.

Enquanto a chave pública é conhecida por todos os clientes de um serviço específico sendo usada para encriptar, a privada é conhecida apenas pelo provedor do serviço (o próprio servidor), sendo usada para decriptar.

Dentro desta concepção de comunicação entre diferentes aplicações, alguns conceitos se fazem necessários para que se possa construirum sistema baseado em microserviços. É importante salientar que sem segurança, esse processo de construção de APIS, logo, microserviços se torna inviável. A integridade dos dados é tão importante quanto a performance e modularidade do sistema.

Neste tipo de comunicação, a aplicação que está fornecendo os dados precisa identificar seus clientes, além de saber se o cliente é realmente um cliente válido, a aplicação ainda precisa identificar as permissões deste cliente, as famosas roles. Para isso, o protocolo HTTP permite dois tipos de autenticação: *basic* e *digest*.

A *basic* utiliza a chave authorization no header da requisição usando o algoritmo Base64 para encriptar o seguinte formato: {usuário:senha}.A seguir, pode ser observado um exemplo:

usuário Kaueh e a senha kaueh moreno, o header com a chave authorization teria o seguinte valor abaixo:

Authorization: Basic a2F1ZWg6a2F1ZWggbW9yZW5v==

Base64. . . (, Site usado para simular encode de dados utilizando o algoritmo Base64)

O algoritmo Base64 trata-se de uma conversão de dados binários para alfanuméricos.

O *digest* é outro mecanismo de segurança que de tão robusto não se faz

necessário uso de SSL para proteção de usuário e senha(embora se faz necessário para proteger o body da requisição). Este mecanismo tem como negativo, o fato de forçar que haja duas requisições para que a aplicação cliente possa utilizá-la. Por isso muitas vezes ocorre a não preferência em usar este mecanismo.

Ao efetuar uma requisição,o cliente recebe um cabeçalho conhecido como desafio, que contém diversos parâmetros ,tais quais: realm, qop, nonce e opaque que nada mais são o contexto de autenticação. Uma vez que tem esses dados em mãos, o cliente calcula a resposta levando em conta todos os dados; identificação do usuário, senha, número de mensagens já enviadas, o método a ser utilizado(GET, POST, FETCH, DELETE), a URL e uma string aleatória gerada pelo próprio cliente.

Este mecanismo se apodera do uso do algoritmo MD5, segundo Alexandre Saudate escritor do livro REST Construa API'S inteligentes de maneira simples, o MD5 não é um algoritmo de criptografia, e sim, de hash e uma vez calculado, não podeser revertido.(Casa do Código, , 206, capítulo 8.7)

Um grande fator de importância na garantia de uma comunicação segura entre serviços é a utilização de técnicas de segurança para intercomunicação, uma delas é o uso do OAuth, em princípio é uma técnica que permite o usuário reaproveitar seu login e senha de uma outra aplicação sem ter que prover estas informações para aplicação a qual o cliente está se comunicando, como por exemplo: ao cadastrar um cliente numa aplicação qualquer, utilizar-se do login e senha e informações básicas daquele usuário de aplicações como o facebook, que por sua vez pode restringir os dados passados a aplicação solicitante. Estarestrição de dados também é uma característica desta técnica de autenticação, que é o acesso restrito.

Há duas versões do OAuth, 1.0 e 2.0. Na primeira versão citada, a aplicação que irá solicitar as informações de um usuário a outra aplicação, necessita ter um registro na aplicação que irá servir, muito conhecido como *Access Key*, as mais famosas são: as do google, para utilização de suas APIS, tais quais: Google Maps, Youtube, Google Engine etc. Etambém as do Facebook, para criação do client secret na construção de bots do messenger.

Essas chaves são usadas para o estabelecimento de confiabilidade entre aplicações, para então gerar um accessKey temporário, usado para prover a identificação de endpoints e callbacks que serão necessários ser feitos. O usuário será redirecionado para a aceitação da passagem de informações e logo após será dado um callback para a aplicação solicitante, utilizando o acesso temporário de autenticação as informações básicas desse usuário, bem como, login e senha.

Durante esse processo de *Back and forwards* entre aplicações diversos parâmetros estão envolvidos:

- **oauth_nonce**: string aleatória, usada na assinatura da mensagem;
- **oauth_callback**: a URL da aplicação solicitante;
- **oauth_timestamp**: tempo da requisição, esperado pelo servidor;
- **oauth_consumer_key**: a consumer key do solicitante;
- **oauth_version**: versão do mecanismo OAuth usado;
- **oauth_signature**: um hash para representar a assinatura da mensagem;
- **oauth_signature_method**: algoritmo usando na assinatura;
- **oauth_token**: token temporário ou o token definitivo - neste último caso, conhecido como *Access Token*.
- **oauth_token_secret**: a senha do token usado;
- **oauth_verifier**: verificador retornado a aplicação solicitante.(Casa do Código, , 210, Capítulo 8)

A vesão 2.0 pode ser considerada mais simples, uma vez que, o redirecionamento é feito para aplicação que fornecerá os dados, e após as autorizações o *callback* será efetuado com o *Access Token.* A razão pela qual a versão 2.0 é mais simples, se dá pelo fato de atuar sobre o protocolo HTTPS.

No capítulo a seguir a priori aborda-se temas generalistas sobre aplicações monolíticas. Buscou-se explicar o desenvolvimento padrão das aplicações denominadas monolíticas e apontar os pontos positivos e negativos desta arquitetura.

3. CARACTERÍSTICAS DE UMA APLICAÇÃO MONOLÍTICA

Tradicionalmente, aplicações monolíticas formam um grande pacote, essas aplicações geralmente são compostas por três partes consideradas principais, tais quais:

- frontend, que é a camada de interface para o cliente (páginas HTML e Javascript).
- backend que é conhecido como server-side, responsável por manipular as requisições HTTP(agora HTTPS), executar as regras de negócios do sistemas, fornecer dados as bases de dados e renderizar resultados ao frontend para que os usuários possam interagir, por fim o banco de dados (base de dados em um mesmo lugar, geralmente um sistema de banco de dados relacional) que tem como objetivo a persistências dos dados gerados pelo sistemas

Figura 1 a seguir descreve um pouco a composição de uma aplicação monolítica

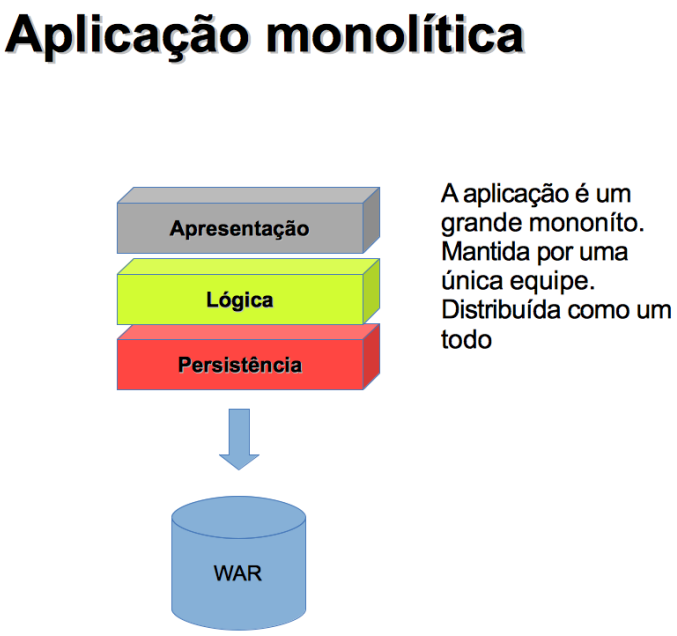

Figura 1 – Arquitetura de uma aplicação monolítica

Fonte: O bom programador - http://www.obomprogramador.com/2015/03/micro-servicos-o-que-sa o-e-para-que.html

Estas aplicações denominadas server-side são consideradas monolíticas por sua arquitetura em blocos definidos de responsabilidade na composição do sistema.

Qualquer alteração no sistema consiste obrigatoriamente em uma nova versão da aplicação server-side, isto é, uma modificação em um consumidor, classe responsável por montar um fila de notificação (consumo) para outro serviço por exemplo, causará a uma atualização forçada em todo o sistema, pois estão todos dentro do mesmo container.

As aplicações monolíticas são muito bem-sucedidas no propósito geral de desenvolvimento e atende muitas instituições em diversos cenários, todavia a partir do momento em que as aplicações passam a crescer, seus ciclos de mudança começam a ficar amarrados e extensos. Uma pequena alteração feita em uma parte pequena do software faz com que toda a aplicação monolítica necessite ser refeita (rebuild). A estrutura modular vai ficando cada vez mais difícil de se manter, sendo difícil separar as mudanças que deveriam afetar somente um módulo, visando a escalabilidade da aplicação, é necessário escalar toda a aplicação, ao invés de escalar somente as partes que necessitem de maiores recursos, de modo que que a manutenção seja satisfatória e eficaz mesmo com o crescimento exponencial do sistema, mantendo lógicas simples conforme o crescimento de responsabilidades e serviços ao redor de um sistema e que seja fácil uma mudança dentro de ecossistema sem afetar o todo, bem como, uma alteração de linguagem ou adoção de uma outra tecnologia. O surgimento dessas necessidades enfrentadas para manter uma grande aplicação monolítica começou a desenhar-se com a componentização ou modularização, que será abordado no seção 4.2 . Um framework conhecido por ter o conceito de componentização, DRY(Don't repeat Yourself) e independências entre os módulos para que sejam genéricos e possam ser reutilizados por diversos projetos é o Django.

> Em tradução livre, Todo conceito distinto e / ou peça de dados deve viver em um, e apenas um, lugar. A redundância é ruim. A normalização é boa. (Django Software Foundation,)

Uma sintetização desse assunto pode ser representada por um cenário hipotético de uso, tal qual, ao usar-se um e-commerce como parte de um módulo dentro de uma aplicação, entende-se que esse módulo pode ser potencialmente genérico o suficiente para adequar-se a outros projetos que queiram utilizar um e-commerce, esse é conceito de componentes, apps(aplicações ou parte delas) independentes que o framework Django traz consigo, pode-se considerar um conceito muito parecido com o que o microserviço também tem.

> As várias camadas do quadro não devem "conhecer" umas das outras, a menos que absolutamente necessário. Embora o Django venha com uma pilha cheia por conveniência, as peças da pilha são independentes de outra sempre que possível. (Django Software Foundation,)

Na construção de um software é necessário entender o tamanho da aplicação e mover-se em busca de soluções mais adequadas a cada ciclo vivido pelo sistema. Potencialmente, soluções de microserviço podem se adequar de maneira mais eficaz nos casos de grandes sistemas com enorme complexidade de regra de negócio que tenha a necessidade de fazer alterações se correr grandes riscos de maneiras isoladas, todavia, o monolítico pode atender boa parte desses requisitos se bem arquiteturado e evoluído conforme as necessidades.

> Faz parte do papel do arquiteto de software tomar decisões baseadas em trade-offs. Nunca teremos uma solução perfeita (ADRIANO ALMEIDA - CONSULTOR, DESENVOLVEDOR E INSTRUTOR DA CAELUM, 2015)

Em princípio, pode-se dizer que a arquitetura monolítica atende muito bem as necessidades da maioria dos softwares desenvolvidos, diversas ferramentas auxiliam na rápida identificação de métricas e cenários para um melhor ajuste da aplicação para suportar diversos acessos simultâneos, bem como NewRelic [1] e Sentry [2] permitem que as aplicações sejam bem monitoradas, tenham rápido ponto de ajuste e um controle enxuto dos serviços providos e gerados. A manutenibilidade, acaba se beneficiando deste modelo, pois exige que os desenvolvedores conheçam bem a aplicação e não tenham que mexer em diversos serviços separados. Como já vimos no capítulo anterior, as aplicações monolíticas já trazem o conceito de modularização/componentização, como o framework Django citados no capítulo 3.1. Potencialmente, grandes resultados de performance com uma aplicação monolíticaenxuta e bem definida podem ser alcançados, definitivamente grande parte das aplicações não utilizam totalmente o conceito de microserviço, pois o mesmo, é difícil e custoso para desenvolver, requer uma maturidade grande na parte tecnológica da empresa, e nem sempre compensa, dado a audiência da aplicação (NADAREISHVILI et al., 2016). Claramente, sempre é necessário que haja a noção do que realmente está sendo feito, para que o crescimento da aplicação não venha a ser um fator prejudicial ao longo do tempo, devido ao crescimento de complexidade e dificuldade de manutenção.

A seguir, no próximo capítulo, abordou-se sobre a arquitetura microserviço e todos os pontos principais a cerca desta arquitetura, eventualmente os pilares, os benefícios e também as precauções que devem ser tomadas ao adotar-se o microserviço como parte do ecossistemas de um software, com a finalidade de vincular a transição entre modularização e o conceito de microserviço um capítulo foi dedicado à modularização no processo evolutivo dos softwares.

---

[1] Serviço de monitoração e otimização de uma aplicação. Monitora todos os requests, rotas, alerta sobre erros na aplicação e mostra consumo de CPU, memória e I/O (NEW RELIC, INC,)

[2] Sistema de monitoramente de erros em tempo real de aplicações de software. Comunica via sms, email e até mesmo por chat. (FUNCTIONAL SOFTWARE, INC.,)

4. CARACTERÍSTICAS DE UMA ARQUITETURA MICROSERVIÇO

Com o intuito de sintetizar os fundamentos e os principais pontos do microserviço, este capítulo tem como objetivo traçar de maneira clara alguns conceitos. O microserviço pode ser definido como componentes independentes , fáceis de versionar o código para os servidores em produção e com memória persistente dedicada (ex. banco de dados). Isto significa que o microserviço tem toda a característica de um aplicação dentro de uma aplicação. Todo o conceito de modularização que já se arrasta no meio de desenvolvimento de software, se torna de fato um microserviço a partir do momento em que dedica-se uma máquina(servidor) para responder por esse módulo de maneira independente da aplicação como um todo. Segundo Sam Newman em seu livro Building Microservice, microserviços são pequenos serviços autônomos que trabalham juntos.

O grande benefício de ter essa total separação não só de projeto mas também de máquinas, é que torna-se possível adequar cada parte do projeto a uma linguagem ou tecnologia diferente. A arquitetura torna-se modular, sendo possível escolher a melhor linguagem de programação, seguida das melhores ferramentas que a compõem para que seja construído um serviço que traga benefício a aplicação geral. Ou seja, além de desacoplar o projeto, de acordo com cada serviço é possível escolher novas linguagens, visando não só a melhoria do sistema, mas também agregando a multi-linguagem aos desenvolvedores envolvidos no projeto. Um crescimento não só tecnológico mas também de mão de obra.

O microserviço é um componente implementável de forma independente que suporta interoperabilidade através de comunicação baseada em mensagem(NADAREISHVILI et al., 2016, Página 20, Capítulo 1).

Alguns entusiastas da arquitetura e especialistas no desenvolvimento de software acreditam que o microserviço trata-se muito mais que uma simples arquitetura proveniente de uma evolução modular na construção de um software, o microserviço é um conceito de visão, cultura e inovação. Ao mergulhar-se mais a fundo nesses pilares culturais, chega-se à seguinte conclusão: Microserviço se resume em focar na comunicação, time e na inovação.

> “Os meios de design de modos de comunicação podem ser tão valiosos quanto o design dos módulos de código. A Lei de Conway afirma que as organizações são constrangidas para produzir desenhos de aplicativos que são cópias de suas estruturas de comunicação. Isso geralmente leva a pontos de fricção não intencionais. A ”Manobra Converse Converse“ recomenda a evolução de sua equipe e estrutura organizacional para promover a arquitetura desejada. ” (LAW,)

O comportamento da empresa em termos de equipe, comunicação entre equipes, a maneira como são incentivados o surgimento de novas ideias, a eliminação de dependências em termos de idealização e componentização de um projeto, além de incentivar a autonomia entre times, fazem parte de uma mentalidade encorajadora disposta a utilizar o microserviço. Microserviço é muito mais que source code, acredita-se que o comportamento empresarial facilita para aquisição desta estratégia, isso ocorre devido a complexidade em estrutura uma arquitetura baseadas em microserviços. Muito mais que ter microserviço é saber desenhar a arquitetura, e para isso é necessário que a empresa tenha todos os pilares alinhados, pois a utilização e migração não ocorre apenas por necessidade, mas por entendimento de que a inovação é parte do crescimento do sistema, a comunicação faz com que haja novas ideias e haja uma maior integração na construção do sistema e o time será responsável por manter e crescer o sistema.

4.1. OS BENEFÍCIOS DA ARQUITETURA

É importante frisar que o microserviço não é a solução para todo o desenvolvimento de software, há casos os quais esta arquitetura será uma tragédia, pois haverá tanto detalhe para construir que será destinado mais tempo resolvendo tudo que o microserviço traz consigo do que efetivamente desenvolvendo um software simples e prático.

Grandes empresas estão adotando o microserviço como já citado anteriormente neste trabalho no capítulo 1 tal qual Amazon e Netxflix, Werner Vogels diretor de tecnologia da Amazon descreve a adoção com a afirmação “equilíbrio na velocidade e segurança para escalar”, por consequência dessa adoção, ele descreve o porquê da migração de seus serviços para utilizar a arquitetura microserviço e qual foi o grande ganho dessa mudança.

> “Podemos escalar nossa operação de forma independente, manter a disponibilidade do sistema sem paralelo e introduzir novos serviços rapidamente sem a necessidade de reconfiguração maciça” - (Werner Vogels, Chief Technology Officer, Amazon Web Services)

Tendo uma visão em componentes independentes e em escalabilidade a Amazon conseguiu o aumento da velocidade em sua aplicação além do aumento da segurança do mesmo, sempre coexistindo escalabilidade e disponibilidade. O microserviço permite que haja menos dependência, resultando em um aumento da produção do código, além de permitir que haja o uso de múltiplas linguagens e frameworks em um único projeto. Os benefícios vão além da visão de software pronto, com entrega resiliente e rápida, todavia, ajudam no processo de desenvolvimento dentro da organização da empresa, independência de deploy, a facilidade de alteração em uma parte do

código sem ter que afetar todo o sistema. Uma mudança não interfere na reestruturação de todo o software mas apenas com a parte que está servindo.

O tempo de subir uma modificação à produção(deploy) é reduzido, além de minimizar o custo e número de subidas que não precisam ser feitas, pois um container de código não está atrelado a outra parte do sistema. Como dito antes, alguns benefícios acerca do microserviço estão além do resultado final, eles contemplam o processo de desenvolvimento, manutenção, suporte etc. A criação de serviços independentes, junto com o aumento de escalabilidade(performance), resiliência, maior eficiência, gerenciamento independente, facilitação da recomponentização das partes e a facilidade de testar são pontos que engrandecem a aceitação e absorção desta arquitetura.

De fato a arquitetura de microserviços traz uma maior autonomia para os sistemas que sofrem, devido ao seu tamanho, sua necessidade de ser rápido, a manutenção necessária ao mesmo. Ao adotar este modelo de desenvolvimento, as empresas buscam estar em um patamar acima dos demais, gerando satisfação aos clientes que utilizam a ferramenta, e também aos desenvolvedores que a mantém.

> “A base de código monolítico que tínhamos era tão maciça e tão ampla que ninguém sabia disso. As pessoas desenvolveram suas próprias áreas de especialização e custódia em torno dos submódulos da aplicação.” - Phil Calçado, former Director of Engineering, SoundCloud(NADAREISHVILI et al., 2016, Página 28)

Um grande benefícios que as arquiteturas acabam explorando de maneira massiva, e certamente o responsável por boa parte de melhora na performance da aplicação, são os serviços assíncronos. Este tipo de técnica é claramente difundida na tecnologia como um todo, obviamente os cuidados em torno deste serviço é grande, ainda mais quando o mesmo é utilizado dentro de uma aplicação monolítica, a qual toda a funcionalidade do sistema está debaixo de uma única aplicação.

Este serviço assíncrono não é algo específico de aplicações REST, todavia concentra boa parte de seu uso, isso ocorre devido a liberação do cliente para que o mesmo não fique obstruído esperando o processamento de uma comunicação. O conceito de “future”, em tradução livre, futuro, define que o processamento será retornado assim que ficar pronto, a aplicação será notificada assim que o retorno estiver disponível, isso evita que a aplicação pare e aguarde cada processo ficar pronto para, aí sim, dar continuidade a outras funcionalidades e deveres.

A arquitetura de microserviços permite que sejam escolhidas diversas estratégias para aumentar a resiliência e performance da aplicação, e isto vai desde a escolha

da melhor linguagem para aquele tipo de serviço, até drivers de bancos, tal qual o drive motor do mongodb para queries e eventos assíncronos com o base de dados.

> “Usamos o Motor em ambientes de alto débito, processando dezenas de milhares de pedidos por segundo. Isso nos permite tirar o máximo proveito do hardware moderno, garantindo que utilizamos toda a capacidade de nossas CPUs compradas. Isso nos ajuda a ser mais eficientes com o poder de computação, calcular o gasto e minimizar o impacto ambiental de nossa infra-estrutura como resultado.”
> —David Mytton, Server Density (MOTOR,)

4.2. MODULARIZAÇÃO, A INDEPENDÊNCIA ENTRE MÓDULOS REDUZ RETRABALHO

No visão mais básica do microserviço, entende-se como uma quebra na aplicação em menores partes. A modularização facilita a automação e possibilita um aumento de abstração.

A modularidade permite que haja uma interface de abstração que permite teste na aplicação de maneiras mais granulares, o que aumenta a credibilidade de serviços individuais dentro do sistema. Definitivamente, a modularização ajuda na velocidade de entrega dos sistemas, permite também uma abordagem de desenvolvimento poliglota.

Ter um sistema coeso, que comunica-se de maneira coesa, necessariamente necessita da modularização, pois ao ajustar a intercomunicação deixa-se para trás todo o conceito de dependências desnecessárias que a modularização muitas vezes traz, e passa a dar lugar a construção de serviços que permitem mudanças de acordo com as regras de negócio.

Assim, a criação de um sistema considerado bom com capacidade de crescimento vai muito além de entender o comportamento de suas propriedades, e sim, como será a comunicação entre os seus módulos (NADAREISHVILI et al., 2016, Página 33, Capítulo 2).

Ser modular é muito mais que quebrar o sistemas em partes, é analisar como será a comunicação entre as partes, tratar que o sistema seja coeso e como será o relacionamento entre os serviços; é nesse momento que toda a complexidade da construção de um sistema utilizando microserviço precisa estar. Lidar com sistemas complexos requer uma atenção entre o controle sobre o que está sendo feito e servido, versus a influência em que cada componente tem no sistema(em termos performáticos e de responsabilidade).

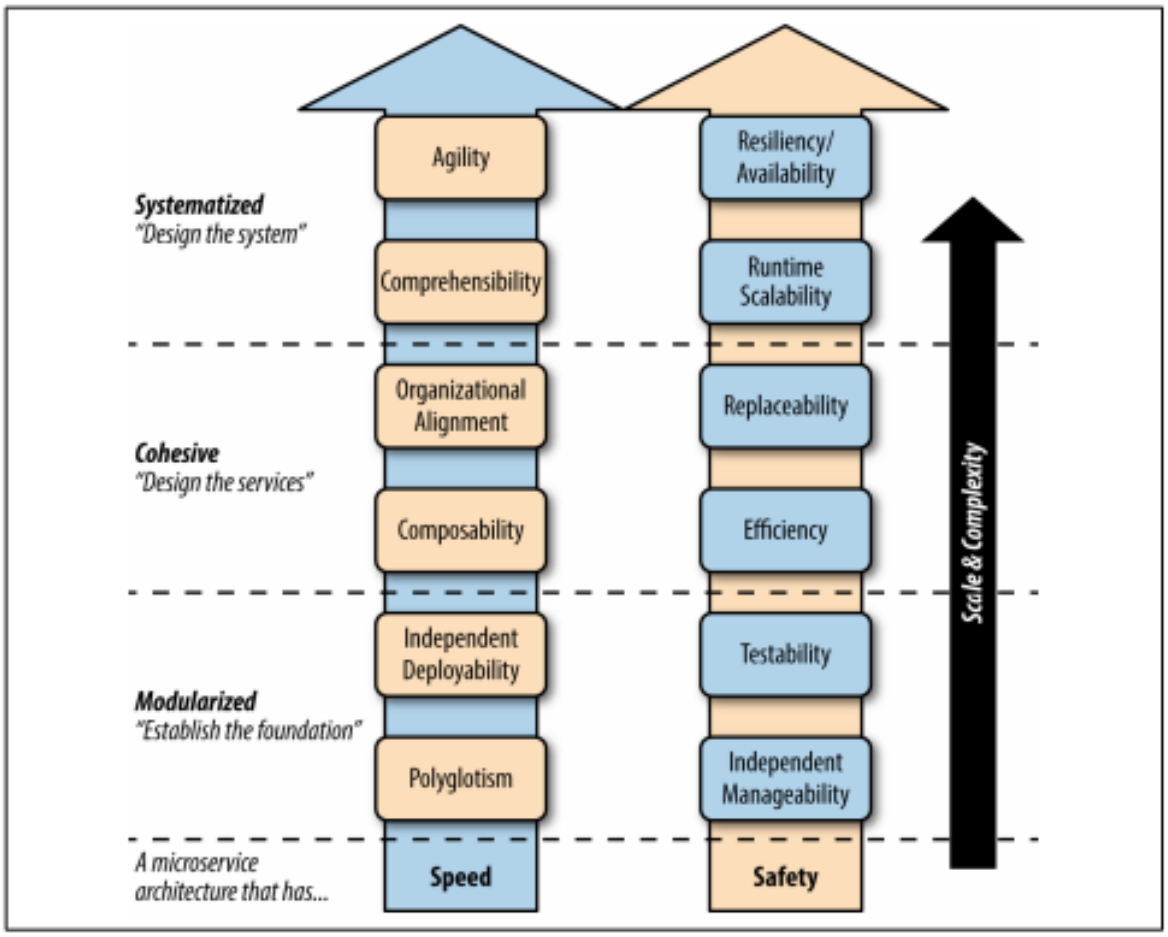

Figure 2-1. A maturity model for microservice architecture goals and benefits

Figura 2 – Modelo de maturidade dos objetivos e benefícios da arquitetura de microserviços
Fonte:O'reilly Microservice Architecture - página 34

4.3 PRECAUÇÕES E ÔNUS GERADOS PELO MICROSERVIÇO

Escolher o uso de uma arquitetura de microserviços não trás apenas pontos positivos a sua aplicação, como tudo no mundo da tecnologia, há desafios, questões e pontos de precauções para desenvolver aplicações deste tipo.

Alguns desafios podem ser encontrados ao adotar-se o microserviço, tais quais:

- Problemas com evolução de serviços, ou seja, incompatibilidade entre cliente e serviço.
- Validações de conteúdos duplicados .
- Falta de domínio sobre as técnicas REST, como por exemplo, HATEOAS.
- Problemas de segurança, evitar expor rotas internas, tratar rotas externas para que tenham desempenho .

O microserviço requer uma arquitetura complexa, e necessita de muita maturidade da parte de quem implementa, é necessário saber os pontos de falhas e orquestrar todos pequenos serviços para que o funcionamento seja perfeito. Há uma grande dificuldade em arquitetar e implementar. Todavia a recompensa pode ser muito benéfica, principalmente ao cliente.

A expressão: “Nem tudo são flores”, se encaixa bem no cenário da arquitetura de microserviço, com o crescimento do projeto a tendência é que haja mais microserviços para representar cada modularização do seu código. Este crescimento faz com que haja necessidade de interação entre os diversos microserviços existentes, eis que surge o problema. O aumento de interação, produz o aumento de requests(requisições) feitas para obter informações em diferentes pontos, deixando as aplicações sujeitas ao round trip time(RTT)[1] . Este delay é um possível ponto de oneração da aplicação, causando um efeito significativo na latência[2] da aplicação. Há diversas técnicas utilizadas para mitigar este cenário, o objetivo deste trabalho não é aprofundar nas técnicas utilizadas na mitigação de latência e round-trip time de uma aplicação com arquitetura de microserviço, todavia, as técnicas serão pontuadas e abordadas de maneira superficial.

Um grande pattern usado para solucionar este problema, é o BFF, conhecido também como Backends for Frontends. *O backends for Frontends será descrito pela sigla BFF ao decorrer deste capítulo.*

Simplificando, a técnica BFF é usada como um agregador de APIS, com a visão de simplificar ao cliente que seja feito um único request com especificações unitárias ou agregadoras, e cabe a API centralizadora (BFF) internamente requisitar o conjuntos de pedidos feito pelo cliente. Diversos benefícios podem ser visto neste pattern, um deles é a diminuição do round-trip dependendo de como foi arquitetado a aplicação, pois um único request é feito de maneira externa, enquanto os outros serviços podem trafegar em redes internas de maneira privada. Assim como os benefícios, os malefícios também podem ser percebidos, tal qual: Centralização de regras em um único ponto, faz com que haja um ponto de falha vital da aplicação, sendo assim, é necessário cuidados para não haver sobrecarga de um único serviço. Uma indisponibilidade deste serviço, pode representar uma indisponibilidade de toda a aplicação, mas isto está atrelado a um outro grande assunto do momento: “Resiliência” - que é a capacidade de absorção de grandes cargas sem gerar indisponibilidade de uma aplicação. Contudo, este assunto apesar de estar atrelado à performance de uma certa maneira, não será abordado profundamente neste trabalho.

[1] Tempo necessário para um sinal ou pacote de dados trafegar de um endpoint ao outro.
[2] Tempo entre um request(requisição) de um recurso e o retorno completo do dado

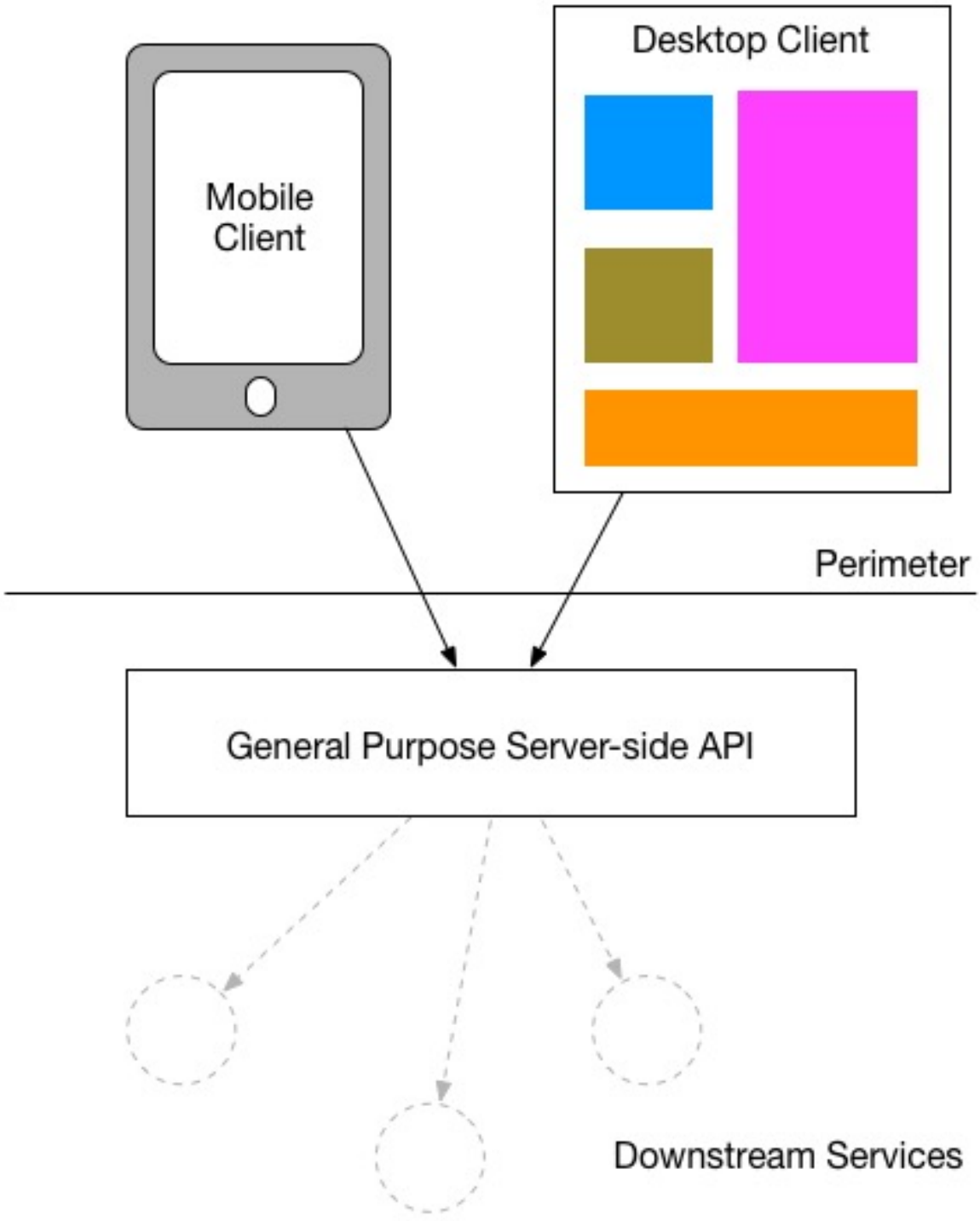

Figura 3 – Api server side de propósito Geral

Fonte: Sam Newman patterns Bff

Outra grande técnica usada por grandes aplicações objetivando o aumento de performance de suas aplicações, é uso de programação reativa, com actor models. Actor models é um modelo de troca de mensagem assíncrona entre processo e entre

aplicações, o que gera um estímulo do concorrência nas tasks de operação dentro dos ecossistemas, gerando um aumento significativo de performance das aplicações.

> Em tradução livre, Sistemas desenvolvidos como “Sistemas Reativos” são mais flexíveis, desacoplados e escaláveis. Sendo assim, são mais fáceis de manter e de mudar. São significamente mais tolerantes a falhas e quando estas ocorrem, são tratadas com elegância. Sistemas Reactivos são altamente responsivos, dando aos utilizadores um verdadeiro feedback interactivo(JONAS BONÉR,)

Segundo o manifesto reativo escrito por Jonas Bonér em 2014 que descreve as caracteristicas de um sistemas reativos, estes sistemas são responsivos, resilientes, elásticos e conduzidos por mensagens, as definições abaixo foram retiradas integralmente do manifesto reativo para melhor definição do mesmo.

Responsivos: estão focados em oferecer tempos de resposta rápidos e consistentes, estabelecendo margens de tolerância que garantam uma qualidade de serviço regular. Este comportamento consistente, simplifica o tratamento de erros, transmite confiança ao usuário final e encoraja a interação(JONAS BONÉR,).

Resilientes: O sistema continua responsivo perante a falha. Aplica-se não só apenas a sistemas críticos de alta disponibilidade - qualquer sistema que não seja resiliente não será responsivo após uma falha. Resiliência é obtida através de replicação, contenção, isolamento e delegação. As falhas são contidas em cada componente, que por sua vez estão isoladas umas das outras e assim garantem que partes do sistema possam falhar e recuperar sem comprometer o sistema como um todo. A recuperação de cada componente é delegada para outro componente externo e a alta-disponibilidade é assegurada por réplica, quando necessário. O tratamento das falhas de um componente não é da responsabilidade dos seus clientes(JONAS BONÉR,).

Elástico: O sistema mantém-se responsivo face a variações de carga. Sistemas Reativos podem reagir a mudanças no ritmo de acessos, aumentando ou diminuindo os recursos alocados para servir estes mesmos acessos. Isto implica na não existência de qualquer ponto de contenção, possibilitando a divisão ou replicação de componentes, assim como a distribuição dos acessos entre eles. Sistemas Reativos suportam algoritmos escaláveis, assim como de previsão e de reação porque geram métricas de desempenho em tempo real. Conseguem elasticidade de uma forma eficaz em hardware de baixo custo (JONAS BONÉR,).

Conduzido por Mensagens: Sistemas Reativos dependem de mensagens assíncronas para assegurar que os componentes estão desacoplados, isolados e localizados

transparentemente, com meios para oferecer delegação de erros na forma de mensagens. O roteamento de mensagens permite gerir a carga e a elasticidade enquanto que o controlo de fluxo cria e monitoriza filas de mensagens no sistema, que quando necessário são roteadas por caminhos alternativos. Transparência na localização das mensagens como um meio de comunicação, torna possível a gestão de falhas de forma idêntica, quer se trate de um “cluster” ou de apenas uma máquina. Comunicação não bloqueante permite aos destinatários consumir apenas os recursos necessários quando estão activos, traduzindo-se numa carga muito menor no sistema(JONAS BONÉR,).

Como pode-se ver, os sistemas reativos visão otimizar a comunicação entre sistemas ou processos, os sistemas passam a reagir dado a eventos e agem de maneira isolada. Muitas ferramentas são utilizadas para implementação deste modelo, são eles: Akka, Kafka, ZeroMQ etc. Estas ferramentas são usadas por grandes empresas e disponibilizam números assustadores quando trata-se de performance, o Akka por exemplo, segundo a empresa responsável pelo desenvolvimento, é capaz de suportar mais de 50 milhões msg/seg em uma única máquina (LIGHTBEND INC.,).

O próximo capítulo, trata do estudo de caso feito para medir a capacitação de performance de aplicações criadas ao decorrer deste trabalho, com o propósito de buscar o fator preponderante para obter-se desempenho nas aplicações e mensurar o quão importante cada decisão pode ser para a obtenção de resultados melhores em termos de performance e resiliência.

5. ESTUDO DE CASO

Por conseguinte do objetivo deste trabalho citado no capítulo 1 fundamentou-se um estudo de caso para testar a performance das aplicações criadas(Monolítica e Microserviço) em diversos cenários de estresse, buscou-se exaurir as aplicações criadas e analisar o que de fato corrobora para que as aplicações tenha um grande desempenho, outrossim, ao longo do desenvolvimento e dos testes realizados será abordado também algumas outras qualidades e problemas encontrados .

Ambas aplicações terão condições iguais de infraestrutura, isto é, haverá a mesma capacidade de máquina, assim também a mesma configuração de web-server(nginx) para que não hajá diferença numéricas nos testes devido a capacitação de SO (sistema operacional) .

Os critérios de performances analisados neste trabalho são :

- capacidade de requests por segundo com conexões sequenciais .
- capacidade de requests por segundos com conexões simultâneas.
- capacidade de repostas com diversos tipos de ações sendo feitas ao mesmo tempo (edição, publicação e entrega de conteúdo) .
- capacidade de entragar diversos conteúdos com um número alto de requisições .
- medição de uso de memória da máquina
- medição do uso de CPU da máquina
- medição de uso de I/O da máquina

OBS: Todos os cenários acima consideram apenas respostas obtidasda aplicação.

Assim como no processo de construção de um aplicação em empresas, os responsáveis técnicos, bem como, engenheiros e analistas, fazem as escolhas das tecnologias utilizadas na construção de um software, buscando entender o que há de melhor. Os mesmos levam em consideração alguns critérios, tais quais:

- Habilidade técnicade seus desenvolvedores
- como a linguagem ou ferramente se encaixa na arquitetura e no negócio criado

- performance
- praticidade/simplicidade

Vale ressaltar que este trabalho também levou em consideração todos esses pontos ao escolher as tecnologias inerentes na construção desse caso de uso.

Após uma análise da tecnologia e definição dos critérios de testes para o desenvolvimento das aplicações, iniciou-se então o processo de criação de cada sistema, um VPS[1] na digital ocean, uma das maiores empresas de cloud computing do mundo, o qual permite criação de servidores privados virtuais para configuração de aplicações, a digital ocean abriga gigantes mundias, como por exemplo: Jquery, .IO [2], Compose etc.

> "Reduzimos nosso aplicativo para 700 milhões de solicitações por dia em apenas 6 meses em cima da infra-estrutura DigitalOcean." (DEN GOLOTYUK - ENGINEER AT .IO,)

Com a variedade disponibilizada pela Digital Ocean em relação a capacidade das máquinas, foi escolhido o modelo que contém 1GB de memória, 1 processador core (CPU), 30GB disco SSD e 2TB de transferencia e dados pois entendeu-se que este modelo corresponde perfeitamente para os cenários de testes explorados neste trabalho, a imagem a seguir descreve as especificações de acordo com o site da digitalOcean.

---

[1] Em tradução livre, Servidores Privados Virtuais

[2] Tecnologicamente falando - é tudo sobre grandes números mesmo para pequenos projetos. Processamos 20 bilhões de eventos por dia. Os dados devem ser coletados de várias fontes e ser constantemente verificados quanto a anomalias. Um evento de cliente único pode levar a uma dúzia de outros eventos a serem armazenados e processados, e. Diferentes fatias e interseções de uma certa métrica.

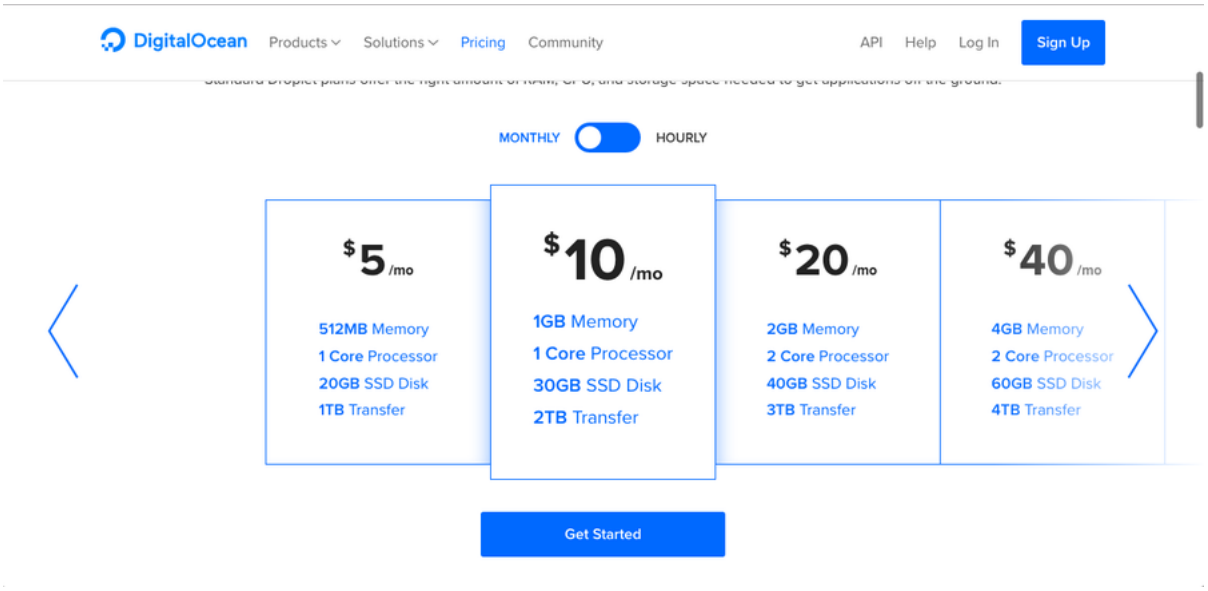

Figura 4 – Digital Ocean opções de Droplets

Fonte: DigitalOcean

O Nginx foi escolhido como webserverdado a sua alta escalabilidade e simplicidade de configuração. O Nginx é um dos maiores webserver do mundo, sendo utilizados em grandes corporações, como por exemplo, Globo.com e Amazon (WS3).

O python será usado como core de linguagem de programação em ambos os cenários, tanto na criação da aplicação monolítica, quanto na criação do microserviço. A diferença entre eles está apenas na sua arquitetura, como framework, banco de dados e serviços utilizados.

A aplicação monolítica é composta por python utilizando o framework Django versão 1.10 e banco de dados MYSQL. A escolha desse framework se dá pela facilidade de criar uma aplicação e de configurar seu ADMIN, área destinada apenas para grupos com acessos privilegiados que tem poderes de configuração na aplicação , a opção pelo MYSQL ocorre pela fácil integração entre o framework Django e a base de dados, fazendo uso de seu ORM .

A aplicação de microserviço irá usar python com o framwork Tornado, focado em AsyncIOLoop[3]

Como banco de dados, será usado o MongoDB com drive motor(Assíncrono) e outros serviços, tal qual AmozonS3 para servir estáticos.

[3] “A maioria dos aplicativos deve usar o AsyncIOMainLoop para executar o Tornado no loop de evento asyncio padrão. Aplicativos que precisam executar loops de eventos em vários tópicos podem usar o AsyncIOLoop para criar vários loops.” (THE TORNADO AUTHORS. REVISION C0F99BAC.,)

Neste capítulo já foi possível perceber que mais do que a arquitetura, a composição dos elementos podem sim contribuir para uma possível performance da aplicação, tal qual a escolha por linguagem mais adequada e serviços mais específicos.

5.1. ESTRUTURA DA APLICAÇÃO MONOLÍTICA

Este capítulo tem como objetivo explicar um pouco da arquitetura usada na criação da aplicação monolítica, rodada em cima do Django. Como objetivo princípal deste trabalho não é a modelagem e criação de um sistemas, vale ressaltar que não foi criado nenhum diagrama específicio para montagem da estrutura simplificada, e nem do fluxo da aplicação.

Num primeiro momento, foi desenhado uma estrutura básica da aplicação, muito mais para exemplificar a relação enter o webserver usado(Nginx) e a aplicação que roda em cima do gunicorn [4]

[4] Em tradução livre, Gunicorn 'Green Unicorn' é um servidor HTTP Python WSGI para UNIX. O servidor Gunicorn é amplamente compatível com várias estruturas da Web, simplesmente implementadas, aumentando os recursos do servidor e bastante rápido.(BENOIT CHESNEAU,)

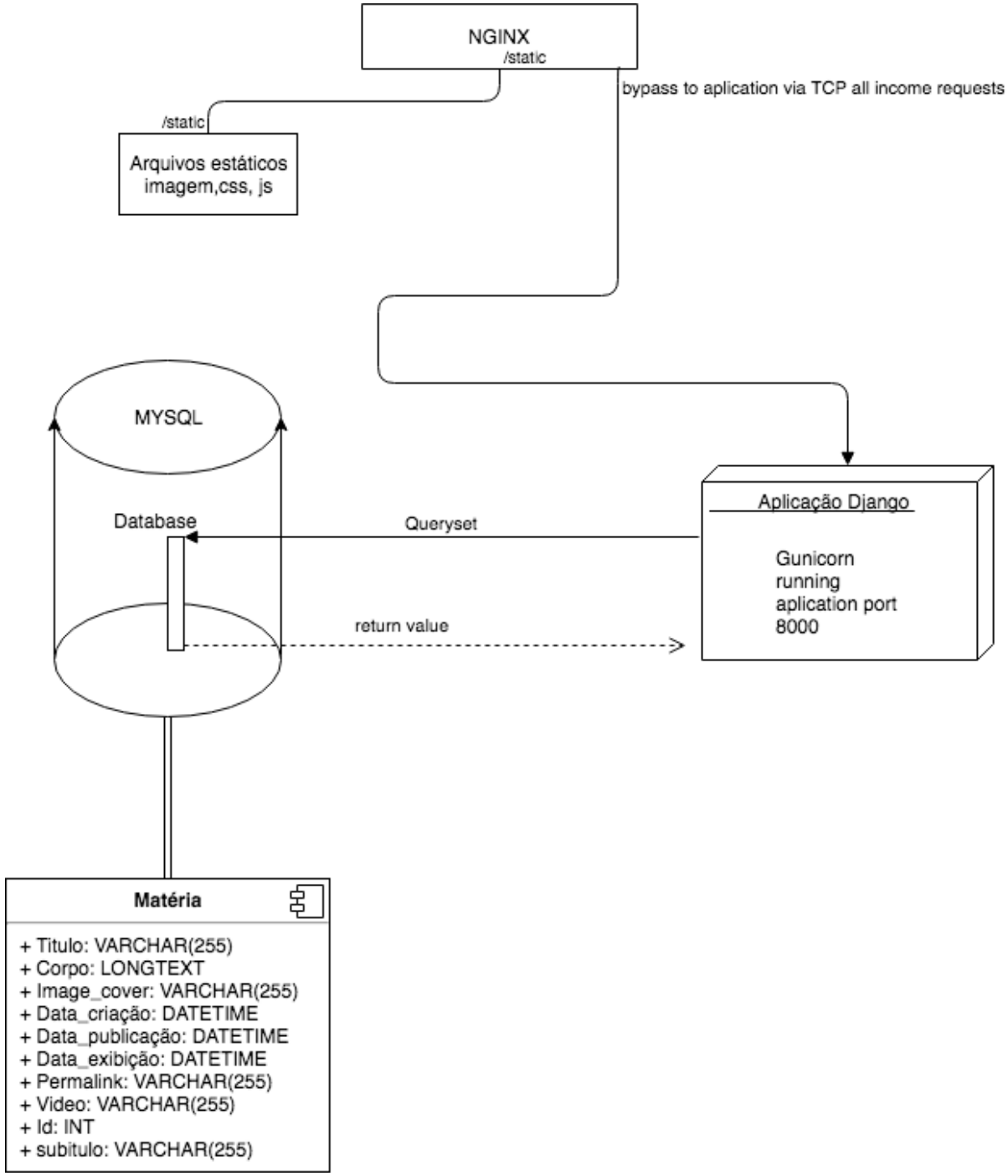

Figura 5 – Aplicação simplificada

Fonte: Próprio autor - Draw.io

O objetivo principal é ressaltar o funcionamento básico de comunicação que a aplicação criada respeita. Basicamente, todo request chega ao webserver, que está uma camada a frente da aplicação, na configuração do Nginx é determinado a porta que irá passar os request à aplicação, seja de maneira TCP na porta 80, ou via socket usando unix. No caso deste trabalho, optou-se pela comunicação padrão TCP.

Os arquivos estáticos serão servidos pelo próprio Nginx que fará uma coleta dos

arquivos estáticos e colocará num filer(path) dentro do Sistema operacional, e toda vez que qualquer coisa referenciada pelo usuário que contiver o path de estático, o nginx irá servir sem ter que a aplicação gerar essas imagens, arquivos css e até mesmo, arquivos js(javascript).

O que for desrespeito a aplicação, como rotas, geração de contexto, o Nginx enviará para aplicação, que estará rodando na porta 8000, levantado pelo Gunicorn, que utiliza o modulo python WSGI (Web Server Gateway Interface) que funciona como uma espécie de contrato entre o web server e a aplicação web, no caso aqui específico, entre o Nginx e aplicação Django.

Qualquer requisição que chega àaplicação, será interpretada e processada o prontamente devolvida ao Nginx, que por sua vez irá passar ao cliente. As configurações de Nginx foram feitas de maneiras iguais entre ambas aplicações, visando equalizar a capacidade. Números de workers connections, zip de arquivos estáticos, entre outras técnicas foram utilizadas, para um melhor entendimento um exemplo de configuração do arquivo Nginx.conf de ambas aplicações a seguir:

Código 5.1 – Ngxin.conf

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;
worker_rlimit_nofile 1035;

events {
        worker_connections 1024;
        multi_accept on;
    use epoll;
}

http {
    charset utf-8;
        sendfile on;
        tcp_nopush on;
        tcp_nodelay on;
        keepalive_timeout 30;
        types_hash_max_size 2048;
        server_tokens off;

    open_file_cache          max=1000 inactive=20s;
    open_file_cache_valid    30s;
        open_file_cache_min_uses 2;
        open_file_cache_errors   on;

    .
    .
    .
}
```

Só para ilustrar parte da configuração feita em ambas aplicações do lado webserver Nginx, deste exemplo alguns pontos são interessantes, tal qual, worker_connections

que é a capacidade de conexões simultâneas que o nginx pode fazer, está capacidade está atrelada ao número de proessos que podem ser abertos simultaneamente pelo sistema operacional, atrelada a capacidade da máquina que nesse caso 1GB de memória, 1 processador core (CPU), 30GB disco SSD e 2TB de transferencia e dados como já citado na imagem 4 . Outro ponto interessante da configuração, são as definições de multi_accpet on que habilita nginx de lidar com multiplas conexões e o keepalive_timeout que é por quanto tempo o nginx mantém uma conexões aberta para tentar reaproveitar ao máximo a mesma instância de continuar conectado sem ter que abrir outro HTTP com outro processo de handshake, 30 segundos é o tempo que uma instância de conexão se manterá aberta para em caso de acesso novamente a conexão pode ser reaproveitada.

Neste trabalho, as regras de negócio e cenários foram restringidos por motivos de concisão; um publicador de matéria, que tem basicamente duas rotas, a rotá que trás toda a lista de matérias criadas e a rota que acessa-se uma matéria específica. Sendo assim, a base de dados é composta por uma modelagem simples, sem grandes relacionamentos, bem direta e simples como descrito na imagem acima.

Para obter um entendimento maior do funcionamento interno deste aplicação foi desenhado um modelo desta arquitetura utilizando a ferramenta drwa.io[5] com o objetivo de mostrar um pouco mais do fluxo, desde a requisição feita por um cliente e todo o caminho percorrido dentro da aplicação até a renderização deste conteúdo de volta ao cliente.

[5] Ferramenta online para desenvolvimento de diagramas de softwares, gráficos, diagrama de processo, UML, ER e diagrama de conexão

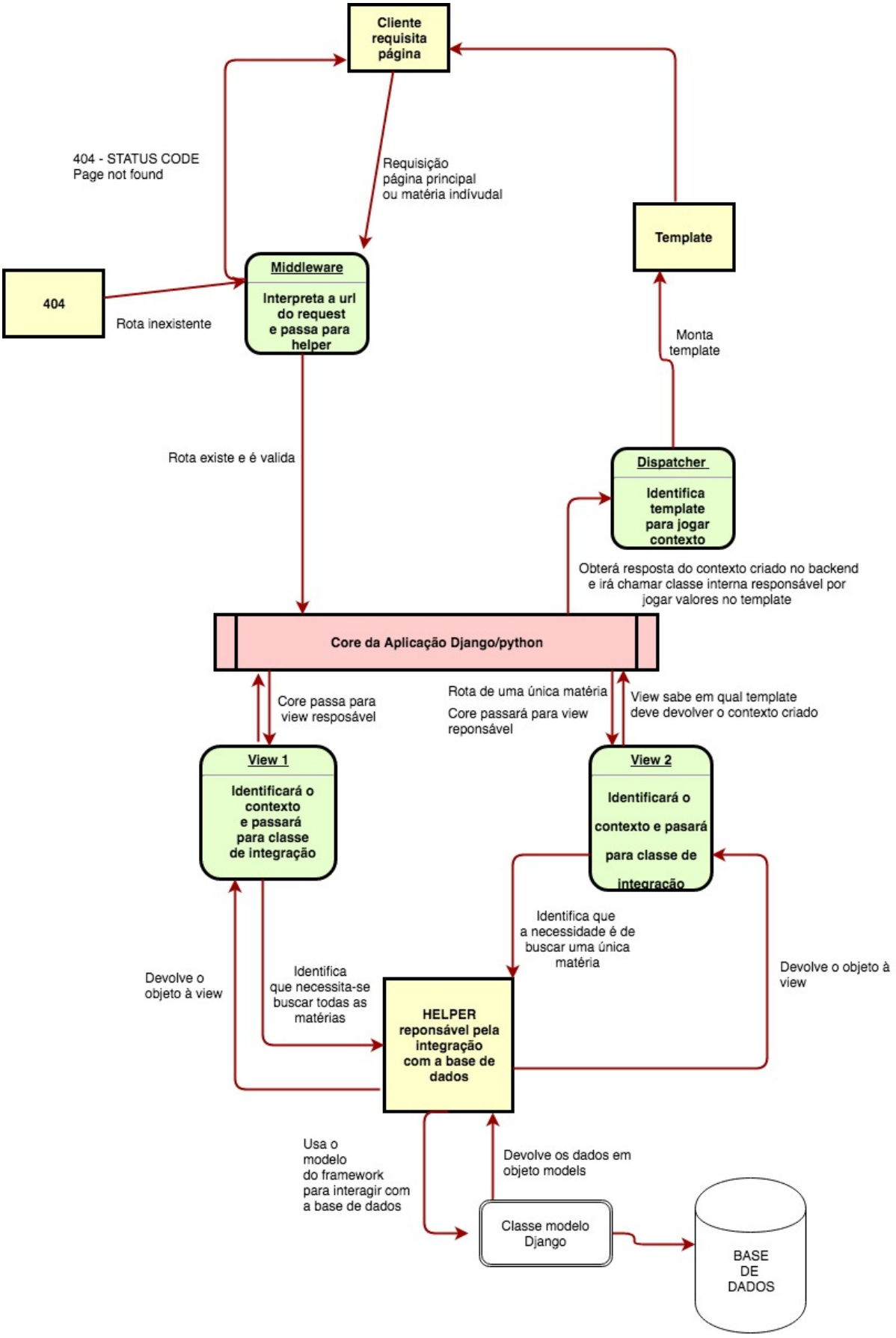

Figura 6 – Fluxo da Aplicação Monolítica

Fonte: Próprio autor - Draw.io

Inspirado no fluxo desenhado anteriormente na imagem 9 a aplicação Monolítica foi desenvolvida em cima do framework Django, que por sua vez prove, o que é denominado por eles como middleware, que são interpretador e interceptores de requests, esses caras são responsáveis por validar alguns fluxo dando um request que chegou à aplicação. Hávários middlewares pré setados no framework, um desses, valida todas as rotas disponíveis na aplicação, ou seja, rotas válidas, caso a requisição que foi transmitida, seja uma rota que não há o mesmo middleware já retorna um HttpResponsa 404 ao Nginx, que por sua vez, devolve ao cliente que solicitou.

No caso de rotas válidas, a aplicação absorverá esta request e buscará a view responsável por responder essa rota.Como exemplo, observamos que no Django, o padrão conhecido como MVC(model, view e Control) é determinado por (models, template e view). As views são classes que recebem o request, todavia, não é considera uma boa prática a obtenção de lógica nesta classe, ela simplesmente pega o request e repassa para classe que faz as regras de negócio, e devolve o contexto ao template, que por sua vez renderiza.

A aplicação irá encontrar a view responsável pela rota solicitada, e será passado o contexto do request para o mesmo. A view, processa esses dados e instância a classe que obtem as regras de negócio, e que de fato se comunica com a classe models passando as informanções necessárias e determinando o que está classe deve buscar.

Esta classe funciona como um helper, ajudando a interpretar o que a view deseja e buscando o modelo necessário. Dado que este resultado é devolvido, a view joga a resposta dentro do contexto e renderiza para o template. Esteprocesso é feito pelo Dispatcher, que pega o contexto e acha o template para que o mesmo possa, renderizar o contexto gerado em backend. Outro ponto presente na aplicação monolítica é a questão interna de administrador, recurso criado para que haja produção de conteúdo de maneira interna. Área restrita por login e senha, esta área é voltada apenas para usuários cadastrados e com permissões pré-definidas, como representados nas imagens abaixo:

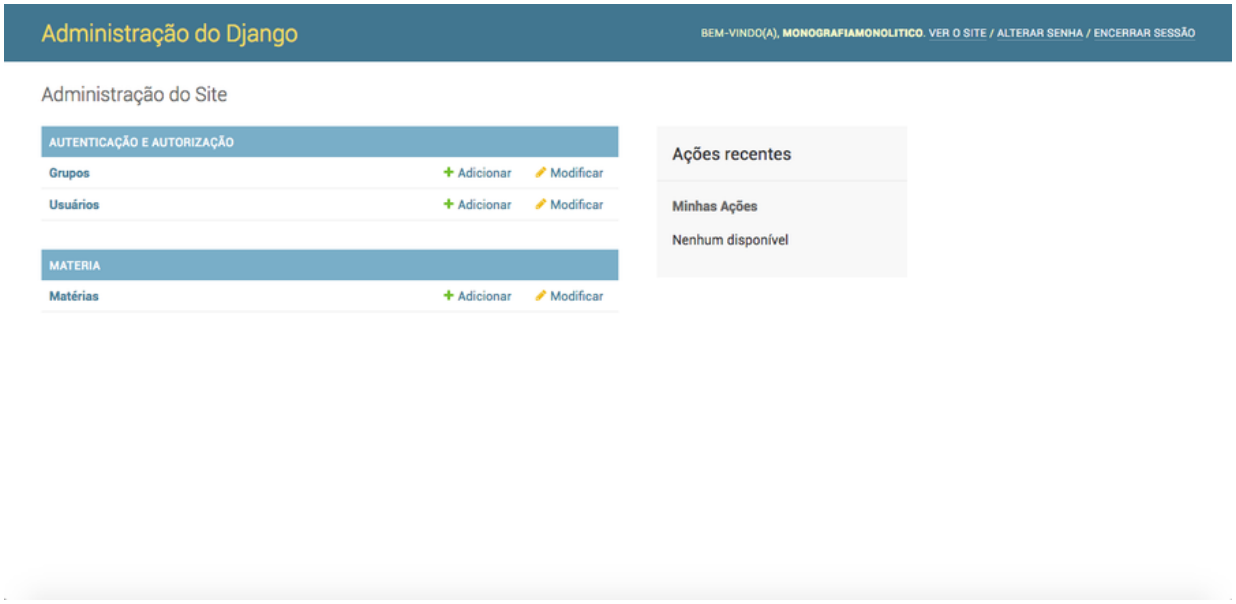

Figura 7 – Admin página principal na aplicação Monolítica

Fonte: Próprio autor

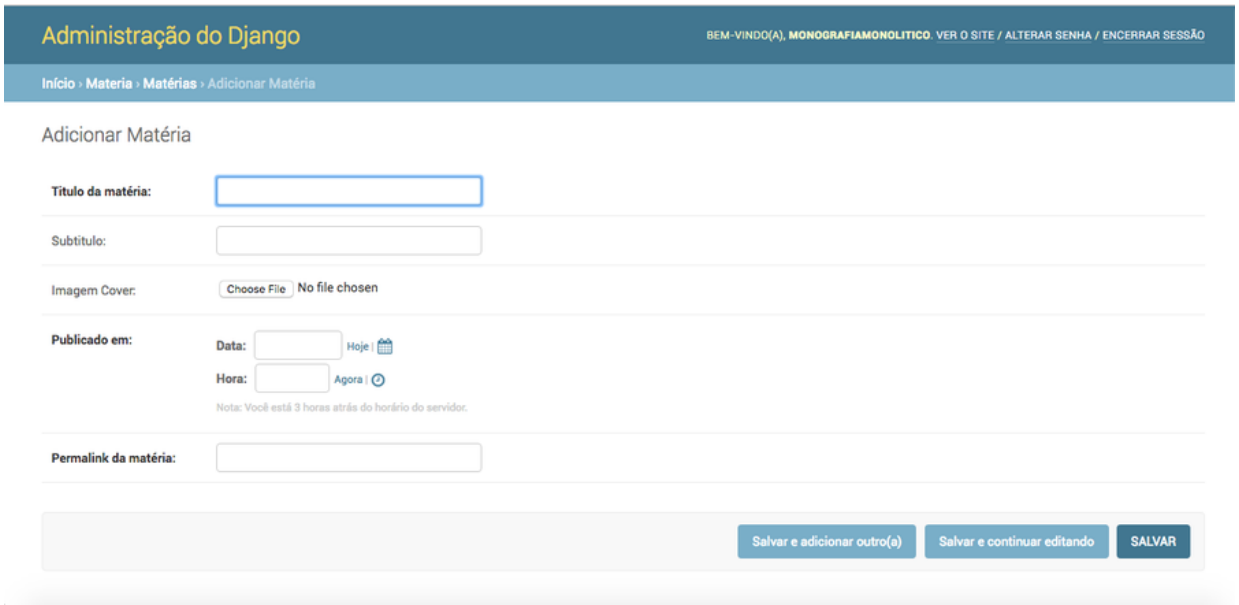

Figura 8 – Admin - Criação de matéria na aplicação Monolítica

Fonte: Próprio autor

A fim de manter a integridade entre ambas as aplicações, criou-se uma rota na aplicação Microserviço para receber posts, sendo assim, toda vez que uma matéria é criada ou deletada, um sinal é disparado. O métodoresponsável capta esse sinal e fazuma requesição a aplicação Microserviço para que a mesma possa indexar estes conteúdos. Essa foi a maneira encontrada de centralizar o ponto de criação de matéria e manter a integridade nos testes entre ambas as aplicações.

No próximo capítulo o tema abordado é sobre a arquitetura e estrura da aplicação Microserviço, bem como explicado neste capítulo 5.1 o próximo também abordara com detalha o funcionamento das classes e estrutura básica da aplicação.

5.2. ESTRUTRA DA APLICAÇÃO MICROSERVIÇO

Em virtude de uma melhor ilustração do fluxo da aplicação foi necessário explicar a estrutura simplificada criada e também os fluxos principais da aplicação Microserviço existentes. A configuração feita é parecida e bem simplificada com relação a arquitetura da aplicação, basicamente, instalou-se um web-service nginx a frente da aplicação, que é responsável por gerir rotas e receber os requests externos(clientes). O que for de entendimento do nginx, quanto a configuração de locations(rotas) ele responderá imediatamente, seja devolvendo um erro em caso de não encontrar nenhuma location especifica, ou passando à aplicação para que a mesma responda essa rota.

Conforme a imagem abaixo sugere, a aplicação foi arquitetada de maneira muito similar para que não houvesse nenhum tipo de distorção nas métricas de performance em ambas aplicações.

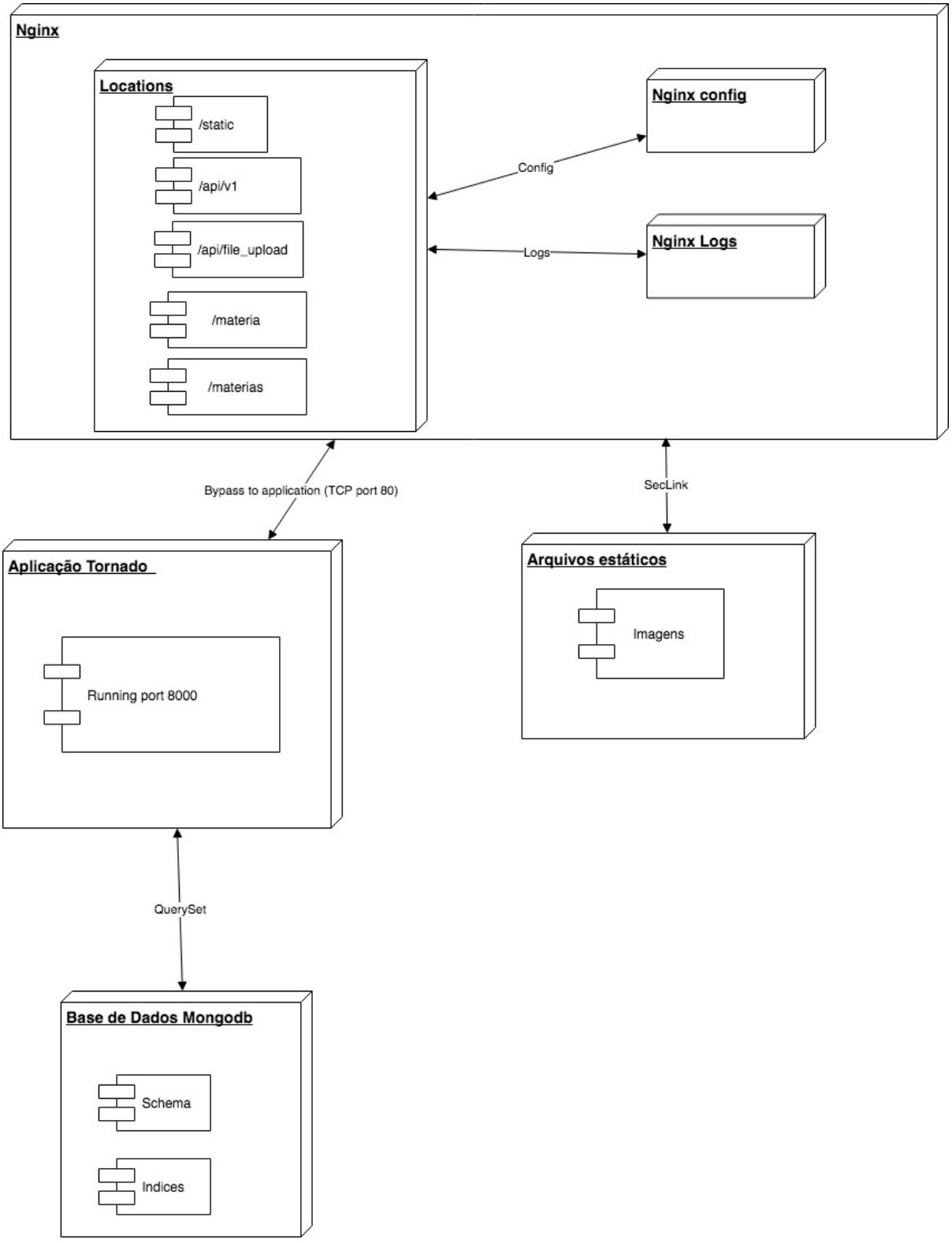

Figura 9 – Estrutura básica da Aplicação Microserviço

Fonte: Próprio autor - Draw.io

Bem como a aplicação monolítica, todo request que chega a aplicação é resolvido atráves do Nginx, que por sua vez identificará todas as locations que estão

definidas e saberá se esse request deve ser passado adiante à aplicação ou se o mesmo poderá responder com algum status code de error (404, 400, 401, 403, 200, 201, 204) dependendo da configuração e das especificações da aplicação. Caso seja uma location válida,a requisição será passada via TCP para aplicação tornado que estará em funcionamento rodando na porta 8000, todavia escutando a porta 80(porta de comunicação padrão do Nginx). Diferentemente do framework Django, o framework tornado não necessita de WSGI[6] para rodar a aplicação na porta 8000. Uma vez que este request chega à aplicação o mesmo será tratado e retornará o resultado de acordo com os processos de código executados internamente, podendo resultar um HTTP status code 200[7]. A cada rota definida a aplicação microserviço irá requisitar ao Mongodb(base de dados utilizada) o correspondente valor.

Vale ressaltar que este trabalho tem como propósito medir as aplicações sem considerar qualquer técnica de camada de cache, sejam elas: utilizandos técnicas de cacheamento dentro do Nginx, usando camadas de varnish a frente do nginx ou até mesmo usando REDIS para camadas de cache interno à aplicação. Todas estas técnias não foram aplicadas, devido a sua grande capacidade de potencialização em tempo de resposta.

Pesando na equalização dos dados entre ambas as aplicações foi criado um modelo de comunicação entre elas para que houvesse integridades nos dados gerados de um lado, pudessem refletir do outro. Sendo assim, a imagem 10 a seguir exemplifica o fluxo da aplicação Microserviço.

---

[6] WSGI (Web Server Gateway Interface) que funciona como uma espécie de contrato entre o web server e a aplicação web

[7] Status da resposta do servidor em caso de sucesso

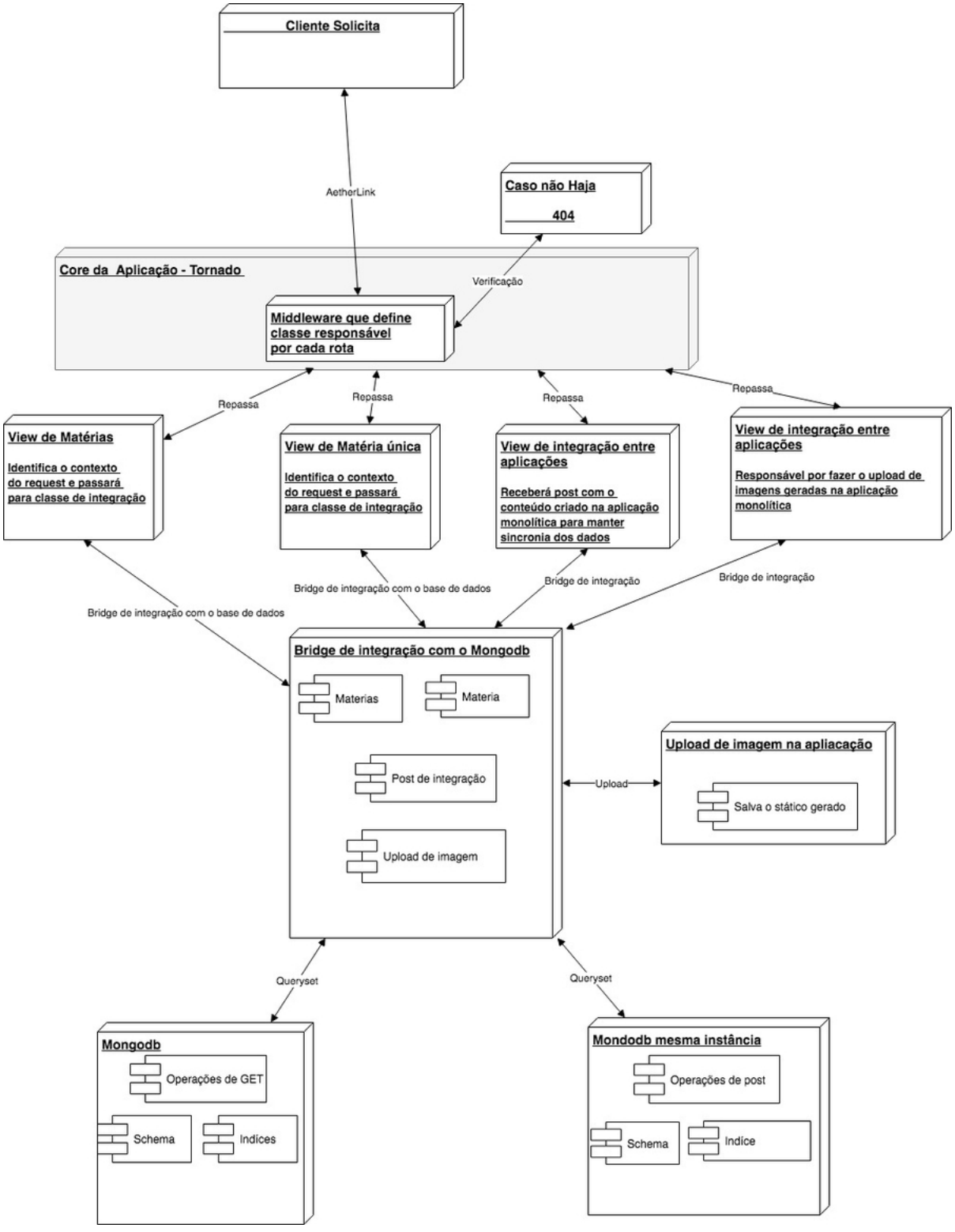

Figura 10 – Fluxo da aplicação Microserviço

Fonte: Próprio autor - Draw.io

No cenário da aplicação Microserviço, foi necessário criar um ponto centralizador de integração entre classes de Request, que no tornado denota-se por RequestHandlers e a base de dados, foi usado um conceito de patterns, conhecido como Bridge.

Basicamente, esta classe é a classe responsável por integrar as operações de query com a base de dados, tais quais: Get, Delete, Create, Update e Insert.

Como foi necessário criar um rota específica de integração entre aplicações, para que as bases fossem equalizadas, e mantivessem a integridades dos dados no momento dos testes, percebeu-seque seria importante haver um ponto de controle entre a base de dados e classes solicitantes.

Um ponto interessante, é que toda vez que uma matéria é criada na aplicação Monolítica, um sinal é disparado e é enviadoum post à aplicação Microserviço com os dados da matéria criado, neste momento, duas operações subsequentes são executadas; uma delas é o envio de arquivos, caso haja, para ser processado. A segunda operação é o envio do json referente à matéria com seus respectivos atributos. A aplicação Microserviço, por sua vez, faz o upload do estático [8]e guarda no filer da aplicação. Na base de dados é inserido apenas o path relativo do upload. O nome e o path foram mantidos iguais, pois posteriomente será mostrado que uma das etapas dos testes, foi integrar o microserviço na aplicação Monolítica, e para isso, foi necessário manter os path iguais para não haver nenhuma incompatibilidade ou quebra de estáticos e gerar um recódigo para ajustar.

Há dois pontos importantes não citados acima de maneira proposital. Um deles é a rota principal, que é vista apenas ao acessar pelo usuário, foi um maneira de aplicar um html e explicar um pouco do projeto àqueles que acessarem o host base da aplicação. O outro fluxo não desenhado, é a integração entre arquivos estáticos que foram salvos na aplicação e o Nginx, isto porque há basicamente duas manerias de ser feito isso, via script, o qual joga-se os uploaders do path interno da aplicação para um path visto pelo Nginx ou apenas apontando na location do Nginx o diretório que contém todos os estáticos que ele deve buscar e dar permissão de leitura e escrita a ele.

No capítulo 5.3 a seguir, um tema de extrema importância é abordado. Detalhes de funcionamento e curiosidade sobre a linguagem de programação usada neste trabalho de conclusão de curso, com o intuito de mostrar como talvez cada decisão no desenvolvimento de um software pode afetar siginificativamente a performance geral de um sistema entender é a linguagem a ser utilizada e suas limitações são fatores que possam tornar-se importante ao tomar uma outra decisão que venha a compensar uma possível perda de performance gerada pela linguagem. Sendo assim, o capítulo 5.3 aborda com detalhes o funcionamento da linguagem e do framework Tornado.

[8] Considera-se estático, tudo que é imagem, arquivos css, html e js,

5.3. PYTHON E O FRAMEWORK TORNADO, DETALHES DE PERFORMANCE

Esta aplicação criada, a qual foi denominada microserviço, foi criada e desenvolvida em cima do framework tornado. Há algumas curiosidades desse framework, conhecido mundialmente como um framework assíncrono. Todavia, antes de falar do framework, é importante falar da linguagem python e algumas de suas caracteristicas importantes/marcantes.

Como o objetivo deste trabalho é analisar a performance de uma aplicação, é necessário ter uma noção de como a linguagem se comporta com relação a este tópico.

O python é conhecido por sua grande facilidade e legibilidade de código, mas não é esse o foco, e sim como a linguagem lida com performance? É uma linguagem que possibilita muito processos ou multi threads? A resposta é não! Isso seria desanimador não? Já que tudo isso que foi testado e estudado pode limitar-se devido a um linguagem? Este talvez seja um fator importante na construção de um sistemas performatico.

Entender como as linguagens de programação funciona nas internas é um fator importante nessa construção; saber explorar os maiores benefícios de cada linguagem são pontos de melhoria e de um início promissor na construção de um sistema que visa performance.

O python é conhecido pelo famoso GIL(Globo Interpreter Lock), este mecanismo é usado pelo módulo interno CPython para garantir que apenas uma thread execute um python bytecode(um objeto python) por vez. Isto acontece, porque o python não é totalmente thread safe, isso significa, que se simultaneamente duas threads tentam incrementar uma referencia em um mesmo objeto python, este incremento pode potencialmente acontecer de fato apenas uma vez e não duas. O GIL vai fazer internamente todo o gerenciamento das threads bem como o sistema operacional executa com os processos, sendo assim, haverá um timeslice para cada thread ser executada, haverá troca de contexto para cada uma. Quando diz-se que apenas uma thread é executada por vez, não significa que não possa ser executada diversas threads, e sim, que toda vez que haver uma chamada interna ao CPython API functions ou uma referência acontecer à um objeto, o GIL irá garantir que apenas uma thread faça isso por vez. A própria documentação do python explica o funcionamento de uma melhor maneira, como por exemplo:

> Em tradução livre, “ O mecanismo utilizado pelo intérprete CPython para garantir que apenas uma thread executa o bytecode Python de cada vez. Isso simplifica a implementação do CPython, tornando o modelo de objeto (incluindo tipos incorporados críticos, como o dict), de forma implícita segura contra o acesso simultâneo. Bloquear todo o intérprete torna mais fácil para o intérprete ser multi-threaded, à custa de

> grande parte do paralelismo oferecido por máquinas multiprocessador.” (PYTHON SOFTWARE FOUNDATION, 2017)

Outro ponto importante com relação a performance da linguagem python. é o *generator*. *Generator* são funções que geram valores. Geralmente todo método retorna um valor e depois internamente o escopo é destruido para desalocar o espaço de memória consumido, ao ser chamado novamente, a função é executada novamente, todavia o *generator* utiliza o yield que diz basicamente ao python para retornar um valor sob demanda, isto significa que numa iteração de um array com 100 posições, uma função normal irá processar todo o dado e irá alocar em memória o resultado disso, ou seja, um array com mil objetos dentro dele. Neste cenário já podemos imaginar que isso é custoso em termos de performance, porque internamente o deslocamento de página do sistema operacional será custoso para mover tudo isso, toda vez que esse resultado transitar em tempo de execução, e isso se repetira num novo ciclo de execução(desprezando é claro, todo o conceito de cache existentes na aplicação) . O *generator* por sua vez, irá devolver apenas um objeto por vez, o que é salvo em memória é apenas um generator, uma classe generator que toda vez que é iterada ou chamada explicitamente usando o metódo next() ele irá retornar o próximo elemento do laço(loop). A citação abaixo sofreu tradução livre de contexto para um melhor entendimento do contexto.

> Geradores são iteradores, mas você só pode iterar sobre eles uma vez. É porque eles não armazenam todos os valores na memória, eles geram os valores na marcha (MUHAMMAD YASOOB ULLAH KHALID,)

Dois pontos são podem ser considerados cruciais para o grande ganho de performance da linguagem a partir do lançamento das ultimas versões, são eles, coroutine e asyncio. As coroutine são como os *generators*, todavia a coroutine consome dados e a coroutine produz dado. Ou seja, podemos criar maneiras de consumir dados sob demanda, sem colocar muito peso na memória e aumentar a performance de execução. A coroutine controla os dados que forem sendo consumidos, há o timeslice para cada dado e controle do que está sendo passado e criado para ela.

Por certo que toda as modificações relacionadas a linguagem obteve ganho de performance o python a partir da versão 3 lançou o asyncio, que por sua vez, juntou todos esses conceitos citados acima a fim de potencializar a execução de tarefas de maneira assíncrona. O python de fato resolveu entrar na brincadeira de performance. O asyncio permite que uma coroutina manipule os dados consumidos, havendo troca de contexto, gerenciamento de timeslice e garantindo que não haja conflito entre as coroutines executadas.

> An application based on asyncio requires the application code to explicitly handle context changes, and using the techniques for doing that correctly depends on understanding several inter-related concepts. (DOUG HELLMANN,)

O asyncio permite tirar vantagem do sistema operacional sem que seja custoso para ele, pois um processo já tem seu espaço de memória alocado pelo SO e seu gerenciamento já está sendo feito, porém, dentro de processos há threads, dentro de threads pode haver centenas de coroutines sendo gerenciadas e controladas para evitar deadlock[9], race condition[10] ou até mesmo starvation [11]. Sendo assim, cria-se uma espécie de multiplexador de execução assíncronas, de maneira que, os resultados que forem mais rápidos vão retornado a aplicação sem que a mesma espere(bloqueantemente) por eles. Isto significa, mais performance com um menor custo ao sistema operacional.

Em virtude da busca de performance, um grande framework é usado para obter-se performance na criação de aplicações usando o python, este framework é o Tornado, que por sua vez, é um framework assíncrono que utiliza técnicas de não ter conexão I/O bloqueantes. Com técnicas que dão a garantia que o que for rodado irá ser devolvido no futuro, tornando-o um framework poderoso para servir milhares de pessoas.

> Em tradução livre de contexto, “Tornado é uma estrutura web Python e uma biblioteca de rede assíncrona, originalmente desenvolvida no FriendFeed. Ao usar E / S de rede não bloqueadora, o Tornado pode escalar para dezenas de milhares de conexões abertas.” (THE TORNADO AUTHORS,)

A grande diferença é que uma operação bloqueante espera por algo acontecer antes do retorno, este bloqueios podem acontecer por questões de disco, conexão I/O. O que realmente acontece é que toda função mesmo que ligeiramente torna-se bloquante, e pensando ao longo de um processamento de um request(requisição feita pelo usuário) se cada função descrita no código bloquear mesmo que um pouco, claramente há uma oneração do tempo de resposta.

> Em tradução livre de contexto, “Para minimizar o custo das conexões simultâneas, o Tornado usa um ciclo de eventos de um único segmento.

[9] É quando um ou mais processos ficam bloqueados, cada um obtendo um recurso e esperando para adquirir um recurso mantido por outro processo no conjunto. (SILBERSCHATS, GALVIN E GAGNE, 2006, 7.4)

[10] Uma condição de corrida é uma falha num sistema ou processo em que o resultado do processo é inesperadamente dependente da sequência ou sincronia doutros eventos. Apesar de ser conhecido em português por ‘condição de corrida’ uma tradução melhor seria ‘condição de concorrência’ pois o problema está relacionado justamente ao gerenciamento da concorrência entre processos teoricamente simultâneos.(WIKIPÉDIA,)

[11] Processos com baixa prioridades podem nunca ser executados. (SILBERSCHATS, GALVIN E GAGNE, 2006, 5.15)

> Isso significa que todo o código da aplicação deve ser assíncrono e não bloqueável porque apenas uma operação pode estar ativa por vez.”(THE TORNADO AUTHORS.,)

Por conseguinte o capítulo 6 mostra os resultados colhidos em todas as etapas de testes feitos ao decorrer deste trabalho de conclusão de curso, a priori explica-se os critérios e limitações dos testes devido a recurso de máquina. O capítulo 6 é voltado para tangibilizar e tornar numéricos todos os processos em cenários de estresse às aplicações Monolítica e Microserviço.

6. AVALIAÇÃO DOS RESULTADOS

Em síntase, todo o escopo e resultados mostrado a cerca deste capítulo é uma representação dos experimentos feitos para medir esforço e comportamento das aplicações em cenários de estresse. Vale salientar que a realização dos experimentos ajustou-se dado as limitações encontradas, todos os resultados e números tem como base de teste um Macbook Pro, com 4gb de RAM mais SSD 240 gb. Os cenários se limitam a capacidade do computador em prover cenários de acesso as aplicações.

Os testes seguiram os seguintes critérios de avaliação; por questões de concisão, os testes foram separados em três macros avaliações denominadas NC1, NC2, NC3, NC4, NC5 e NC6, isto é, nomenclatura referentes a número de conexões que será explicada mais a frente. Há variações de concorrência e acesso sequenciais, o aumento dos valores concorrentes seguiram um critérios de base 2(binário), ou seja, as concorrências aumentam gradativamente segundo a sequencia: 2, 4, 8, 16, 32, 64, 128, 256, 512. Já os números de acessos, eles seguiram um critério de múltiplos de 10 com limite de 1 milhão por questões de capacitação de máquina usada para testar as aplicações, não foi possível criar mais do que 512 threads e nem abrir mais de 1 milhão de conexões. Para uma melhor análise de performance de cada aplicação, foram avaliados os gráficos de CPU, memória e I/O de cada aplicação.

Além dos critérios de avaliação descritos acima, os experimentos também seguiram um critério lógico de avaliação, vale relembrar que todos os testes foram executados tendo como base o que foi explicado e descrito no capítulo 1 e nos capítulos 3 e 4 o qual descreveu-se os objetivos e caracteristicas das aplicações. As aplicação são uma representação de produção de conteúdo, neste caso, matérias, que comportam titulo, imagem e que nos experimentos feitos tiveram como número máximo 10 matérias sendo processadas pelo servidor por cada requisição feita. As cinco primeiras matérias são matérias sem foto, as 5 subsequentes possuem fotos e depois disso foi criado de maneira aleatórias, diversas matérias com fotos e textos.

Os primeiros testes foram executados usando a ferramenta WRK benchmark tool [1], esta ferramenta não permite criar testes os quais os números de conexões sejam menores que o número de threads. Para um melhor entendimento de como os testes se dividiram e as nomenclaturas que foram usadas para descrição da processo a tabela 1 mostra como foi feita a divisão dos experimentos. Como é visto na tabela 1 a letra T é a representação dos números de threads, já a sigla NC é uma abreviação

[1] Wrk é uma ferramenta de benchmarking HTTP moderna, capaz de gerar carga significativa quando executada em uma única CPU multi-core. Ele combina um design multithread com sistemas de notificação de eventos escaláveis, como epoll e kqueue

para Número de Conexões, o qual seu crescimento é exponêncial, a sigla NOP é a representação de No operation, ou não, sem operação em tradução livre de contexto, a sigla NOP aperece quando não foi possível simular os cenários devido a limitação de máquina e de ferramenta, como em casos de conexões maiores que threads..

Tabela 1 – Critérios de teste de performance server-side

T	NC1	NC2	NC3	NC4	NC5	NC6
2	10	100	1000	10000	100000	1000000
4	10	100	1000	10000	100000	NOP
8	10	100	1000	10000	100000	NOP
16	NOP	100	1000	10000	100000	NOP
32	NOP	100	1000	10000	100000	NOP
64	NOP	100	1000	10000	NOP	NOP
128	NOP	NOP	1000	10000	NOP	NOP
256	NOP	NOP	1000	10000	NOP	NOP
512	NOP	NOP	1000	10000	NOP	NOP

Critérios de teste - WRK benchmark tool

Incialmente os experimentos começaram com o uso do WRK, em cada rodada de teste, avaliou-se as saídas apresentadas no wrk, representadas em tabelas para melhor entendimento do mesmo, e também como estes testes se mostraram representativo nos gráficos de cada aplicação em termos de esforço computacional. A seguir nos próximos capítulos, os gráficos apresentados apresentam nomenclaturas especificas e geradas para facilitar o entendimento dos experimentos, daqui em diante toda vez que a sigla MO aparecer, a mesma faz uma referência à aplicação Monolítica, já a sigla MI, faz referência à aplicação Microserviço, as siglas LA são referentes a latência das aplicações durantes os experimentos, essas siglas serão seguidas na rodada referente, como por exemplo: LA-NC1, representa a latência da aplicação vigente na sequência NC1.

Como critério de avaliação os teste de primeira rodada foram até o NC3 contemplando ambas aplicaçoes a segunda rodada também será composta por três fases de análise, englobando NC4, NC5 e NC6.

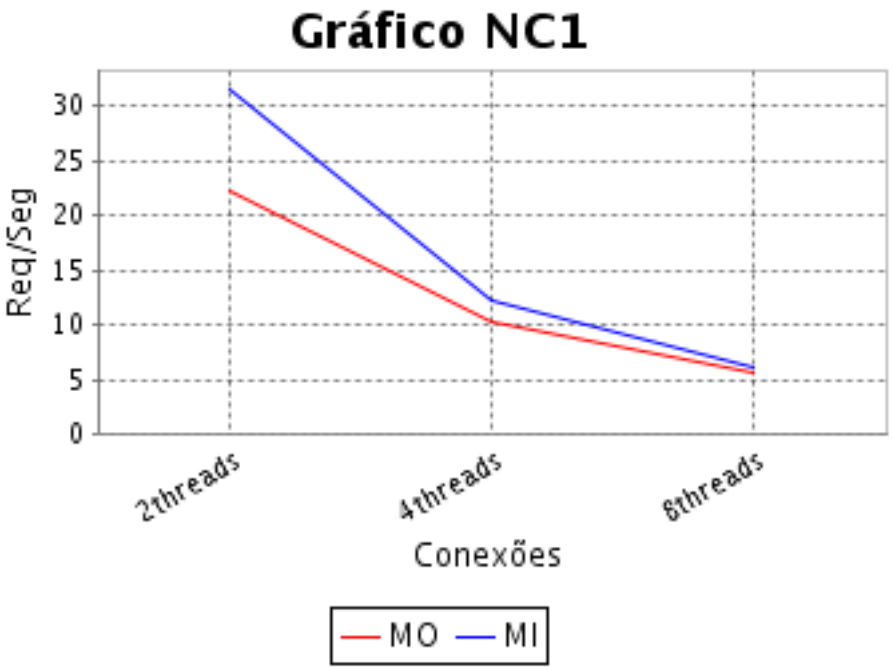

Figura 11 – Request/seg Rodada NC1

Fonte: Próprio autor - Google Chart

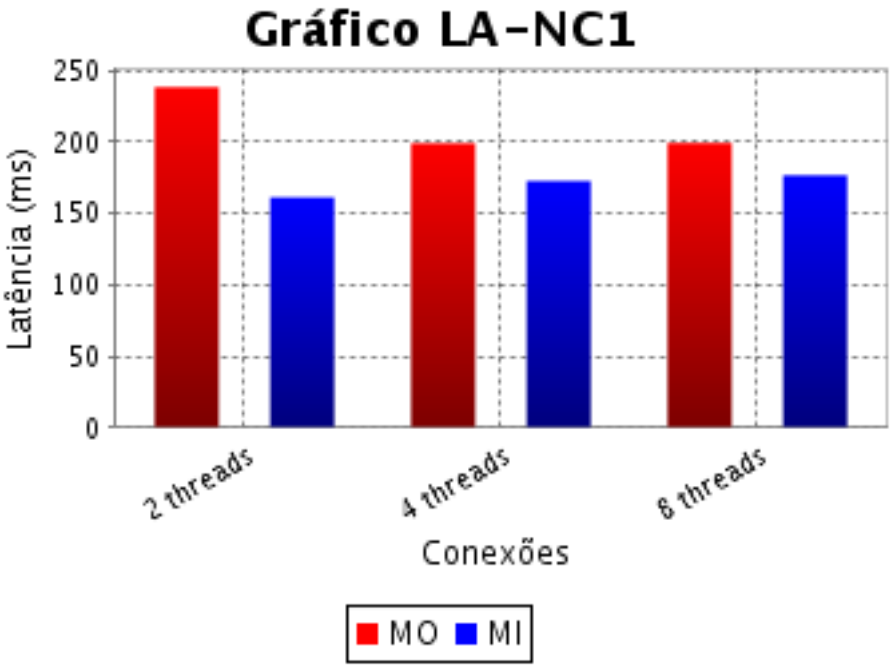

Figura 12 – Latência Rodada NC1

Fonte: Próprio autor - Google Chart

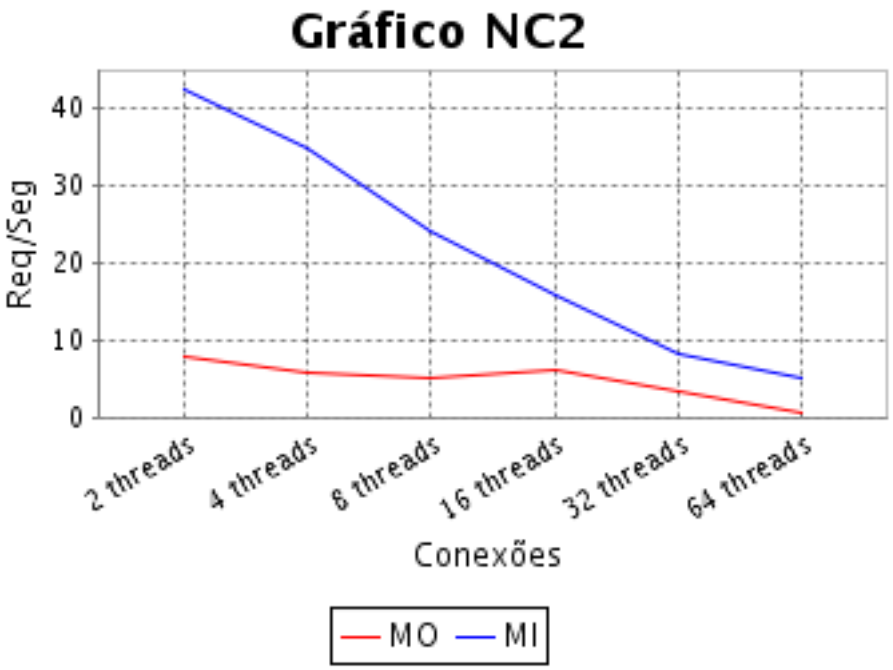

Figura 13 – Request/Seg Rodada NC2

Gráfico NC2 - WRK benchmark tool

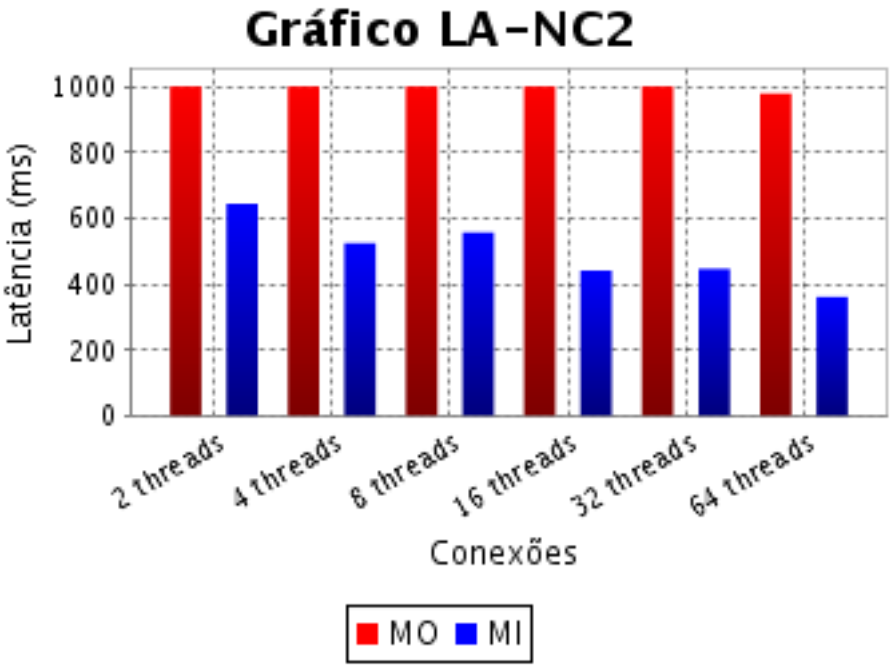

Figura 14 – Latência Rodada NC2

Fonte: Próprio autor - Google Chart

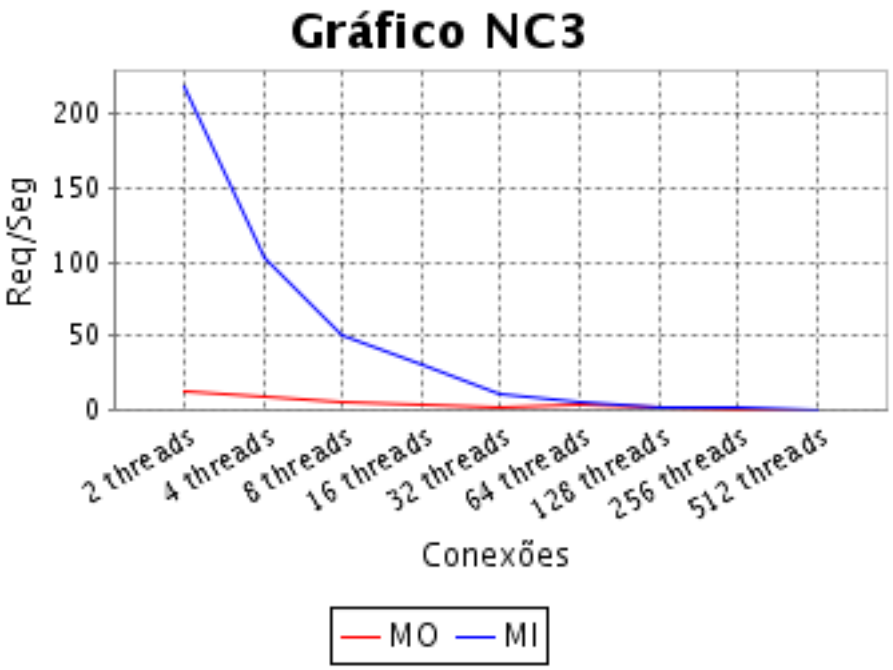

Figura 15 – NC3 Comparativo

Fonte: Próprio autor - Google Chart

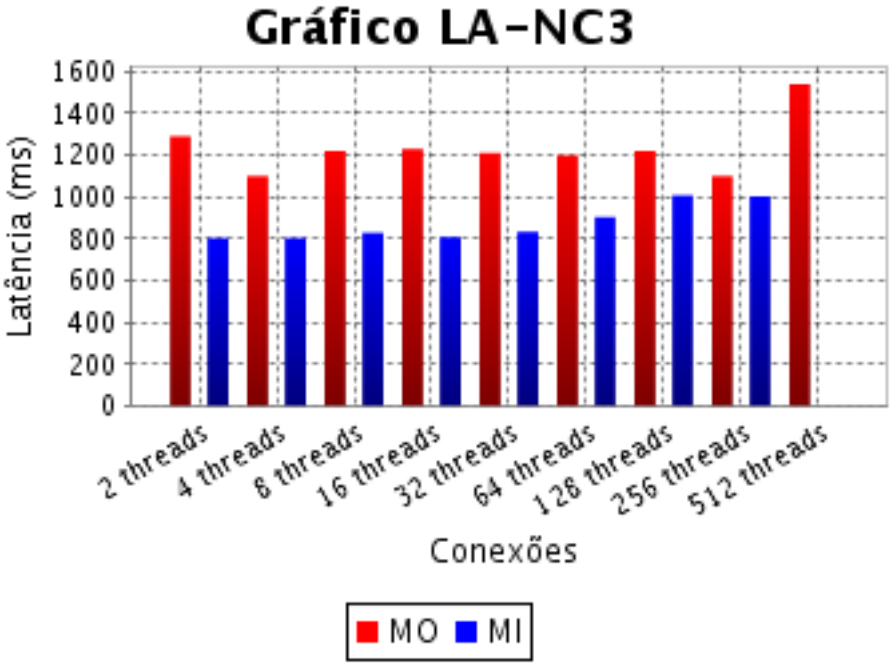

Figura 16 – Latência NC3 - Comparativo

Fonte: Próprio autor - Google Chart

Analisando os gráficos apresentados nessa primeira bateria de execução oberserva-se que a aplicação Monolítica se mostra mais lenta conforme o aumento de conexões simultâneas. Na sequência NC1 a aplicação Microserviço foi 42.18% melhor que aplicação Monolítica em capacidade de requisições por segundo, e teve uma latência de

32.38% menor em comparação com a aplicação Monolítica com 2 threads de conexão mesmo tendo a mesma localização de servidor, ambos estão localizados em Nova York, conforme o aumento de conexões simultâneas que é representada pelo número de threads vai aumentando, a diferença foi caindo, ao final do experimento NC1 com 8 threads a diferença entre as aplicações é de 10.05% em termos de capacidade de processamento de requisições por segundo, como pode-se ver a aplicação Microserviço se manteve a frente em todos os cenários comparado a aplicação Monolítica. Em alguns momento, a distância na medição de capacidade de processamento se mostrou muito grande, como também muito baixa em grandes conexões por segundo, como por exemplo 512 threads simultâneas na sequência NC3, nos picos de resultados a aplicação Microserviço chegou a ser 1641% superior à aplicação Monolítica, batendo 217.69 requisições por segundo, enquanto a Monolítica obteve 12.50 requisições por segundo, essa amostra é vista na imagem 15 citada anteriormente referentes a sequência NC3.

A figura 17 a seguir representa o esforço computacional da aplicação Monolítica durante os teste descritos acima.

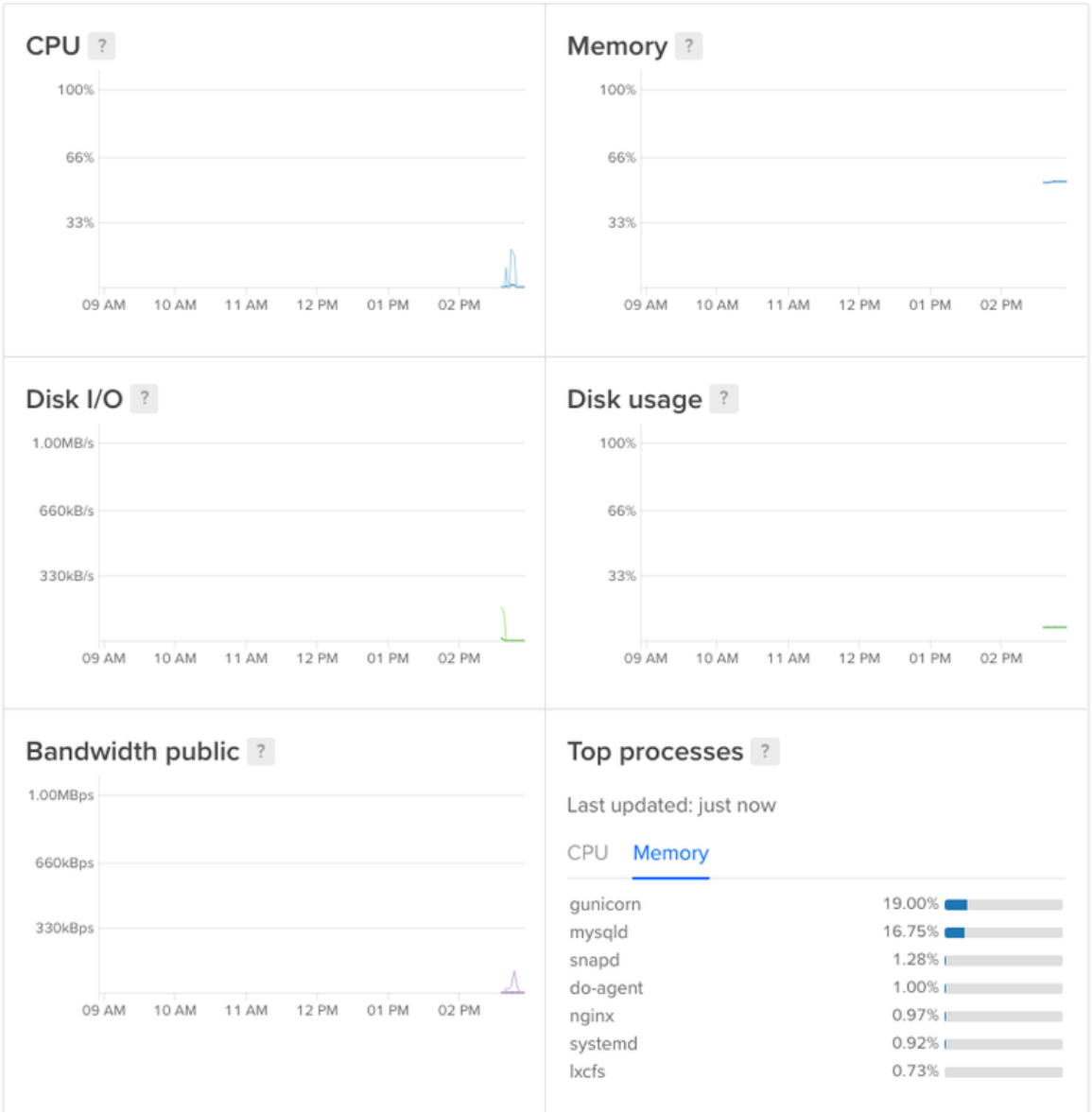

Figura 17 – Esforço computacional - Aplicação Monolítica

Fonte: Próprio autor - DigitalOcean

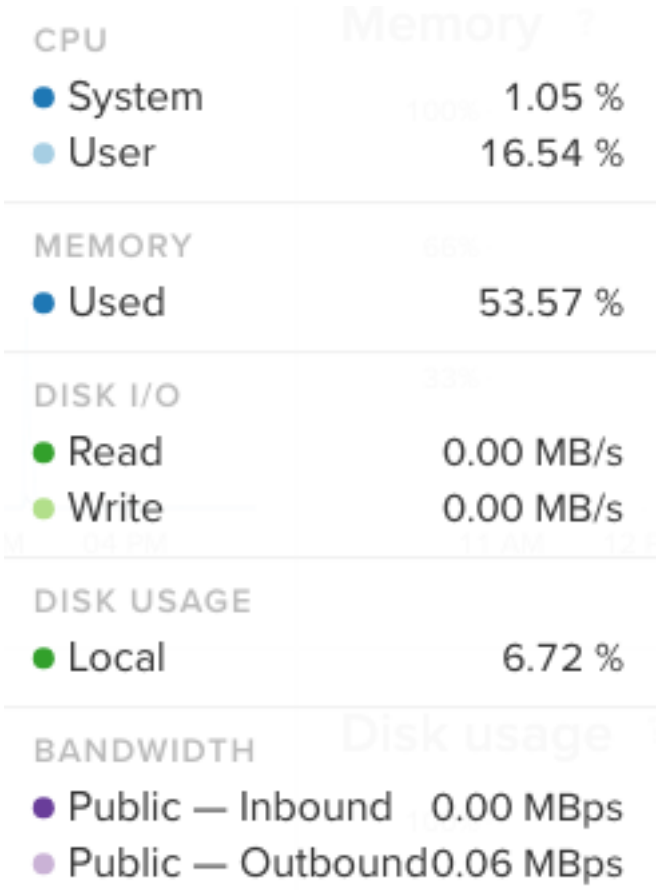

Figura 18 – Esforço computacional
Fonte: Próprio autor - DigitalOcean

Com a finalidade de analisar a performance, a imagem 17 representa o dashboard disponibilizado pelo DigitalOcean para medir o esforço dos servidores em tempo real de cada aplicação criada, dentre as opções alguns dados são importantes para análise de performance das aplicações e por motivo de concisão apenas as métricas de CPU, BandWidth e Memória foram considerados. A imagem 18 é um zoom detalhado também disponibilizado pela ferramenta, do momento de pico como pode ser visto na parte de métrica de CPU, este pico é descriminado de maneira mais detalhada na imagem 18. Apesar de computacionalmente a aplicação não ter um consumo de CPU e memória tão grande abaixo de 20% após toda a sequência, a capacidade de processamento simultâneo da aplicação cai bruscamente cerca de 4332% no caso da aplicação Microserviço conforme apresentado nos resultados dos testes descritos previamente com base no WRK. O uso de CPU chegam a mais ou menos 17% da capacidade total, ocasionando também um leve pico de leitura de disco com relação a operações de I/O, o que possivelmente pode-se considerar normal, uma vez que a cada request a necessidade de processar dados provenientes do banco de dados. No Top processos, foi possivel perceber que o mysql e gunicorn(responsável por responder a todo request que chega à aplicação) são os processos que mais consomem memória.

Assim como a aplicação Monolítica, também foi medido o esforço computacional da aplicação Microserviço e a mesma é representada pela imagem abaixo:

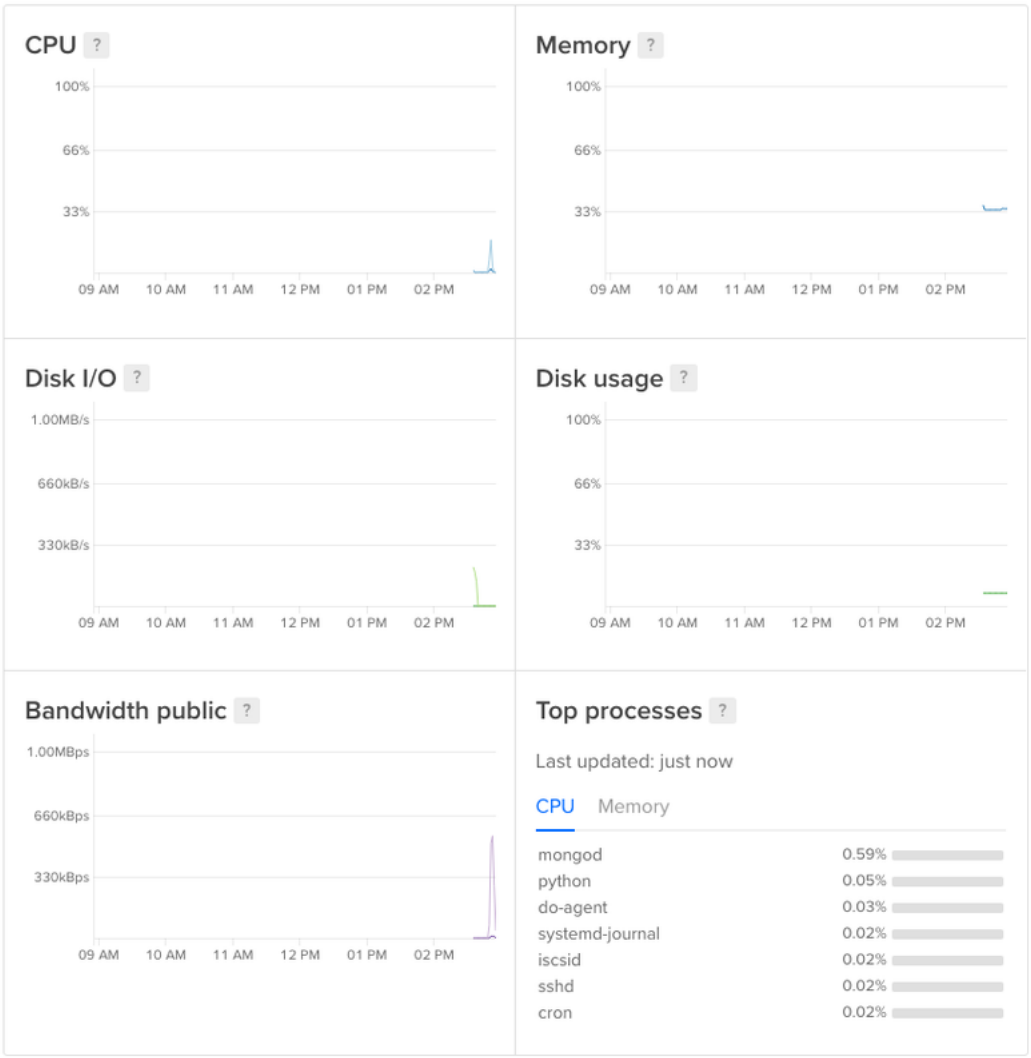

Figura 19 – Esforço computacional - Aplicação Microserviço

Fonte: Próprio autor - DigitalOcean

O esforço computacional da aplicação se mostrou bem parecido ao da aplicação Monolítica, ambas ficaram por volta de 17% do uso de CPU, com leves picos de I/O e uso de disco. Por um outro lado, o consumo de memória se mostra bem abaixo da aplicação anterior, cerca de 83.05% a menos, este número pode ser melhor descrito na figura20 a seguir na descrição python com 3.22%. Outro ponto interessante é o grande aumento de Bandwith que é o trafego de dados na banda até a aplicação, esse número pode-se considerar a representatividade de dados trafegando, ou seja, a aplicação com maiores picos significa que conseguiu trafegar mais dados pela rede, consequentemente processou mais dados, a aplicação Microserviço chegou a transferir 1900% mais que a aplicação Monolítica.

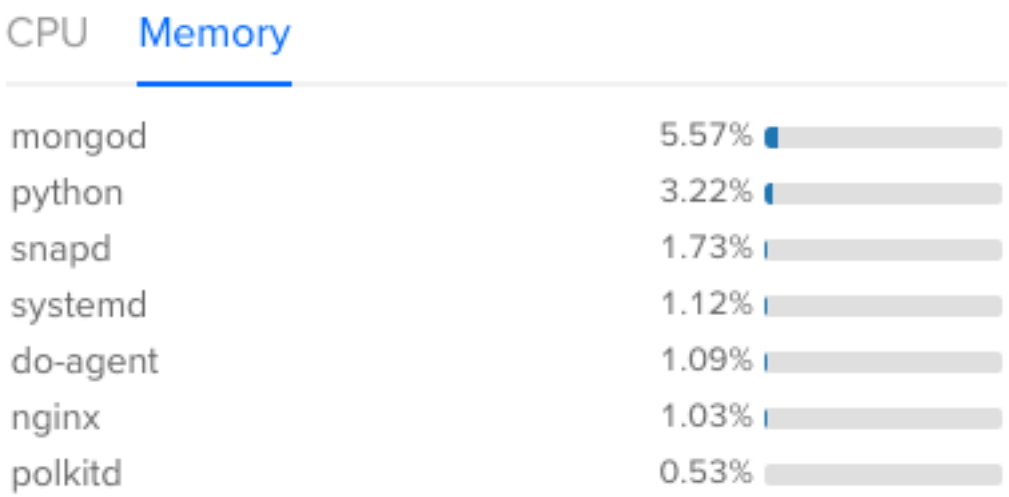

Figura 20 – Consumo de memória - Top processos

Fonte: Próprio autor - DigitalOcean

Os processos que mais consumem são de fatos iguais ao da aplicação Monolítica, porém com um número bem abaixo do que mostrado nas imagens anteriores 19 e 20 respectivamente, nessa primeira parte dos experimentos a aplicação Microserviço se mostrou bem melhor em termos de performance como já citado acima, mantendo uma latência menor, suportando mais requisições por segundo além de processar mais requests dentro de um mesmo período de teste, default de 10 segundos para cada teste. Essa primeira parte do experimento, mostrou que muito além da arquitetura, há indícios de que os componentes que estão acerca de cada aplicação são preponderante na melhora de performance; isso será falado com detalhes nos póximos capítulos.

Referente a imagem 20 anteriormente citada, alguns pontos são importantes para analisar-se, os processos que tem na definição mongod, python e nginx são os nomes que deve-se dar atenção, ademais representam respectivamente o consumo de memória do banco de dados, que utilizou-se mongo como arquitetura como citado no capítulo 4, a aplicação Tornado/python rodando na porta 8888, ou seja, o quanto de memória é consumido pela instância que mantém a aplicação de pé e por ultimo o Nginx que é a medição do webserver escolhido, quanto aos outros agentes, são inseridos pelo DigitalOcean toda vez que um servidor é criado e portante deve ser desconsiderado.

Os experimentos seguir foram apresentados de maneiras reduzidas e com mais detalhamento, com o objetivo de ter clareza e direcionamento nas análises dos resultados, para maiores detalhes, todos os testes deste trabalho estão descrito detalhadamente no apêndice.

De conformidade com os testes citados anteriormente a capacitação média da aplicação Monolítica de menos de um request por segundo, uma latência alta, com mais de um segundo de média, chegando à dois segundos de pico máximo de latência. Os experimentos que representam as sequências NC4 e NC5 demostraram a partir de um momento a aplicação Monolítica apresentou Socket errors com um número de timeout por volta de 361, assim sendo, as conexões passaram a ser cortadas devido a lentidão da aplicação em processar a requisição e devolver a resposta ao cliente mesmo com um keep alive de 30 segundos configurado no Nginx como mostrado no capítulo5.1 na configuração de Nginx, uma amostra dos socket erros pode ser visto na tabela 2 a seguir.

Tabela 2 – Amostra de Socket Errors - Aplicação Monolítica

Connect	Read	Write	Timeout
0	0	0	361

WRK benchmark tool

Conforme o aumento do número de conexões simultâneas e o aumento de conexões feitas durante os experimentos a aplicação Monolítica demonstra uma queda de performance de 99.63% quando compara-se a primeira sequência NC1 com 2 threads com a sequência NC3 com 512 threads, como também a aplicação Micrserviço que obteve uma queda de 98.41%, sendo que a aplicação Microserviço se manteve 525% melhor no pior caso desses citados aqui, NC3 com 512 threads, isto significa que o acesso concorrente onera muito mais as aplicações do que diversos acessos em sequência, esse comportamente pode-se dizer que é um comportamento esperado uma vez que acessos simultâneos exponência o esforço da máquina para lidar com chamadas em paralelo.

Assim como a aplicação Monolítica, a aplicação Microserviço também demonstrou Socket erros, numa menor proporção de 38.22% saiu de 361 para 223, o que indica uma capacitação maior de lidar com os acessos e processar os dados requisitados pelo WRK.Apesar de aplicação monolítica também sofrer com a alta concorrência alguns números chamaram a atenção, um deles, é a capacidade de requisições totais no teste, em comparação com a aplicação Monolítica, esse aumento é 3x maior, além disso, foi possível transferir muito mais dados(bytes) por segundo, mesmo sabendo que a aplicação Microserviço também teve dificuldades para processar requisições. Os testes estão melhores descriminados no apêndice.

A aplicação Microserviço melhorou de maneira significativa conforme temos

menos concorrência, passa-se a ser capaz de processar 30.71 requests/s e não mais 0.5 requests/s como mostrado anteriormente. Ambas aplicações sofrem com a concorrência, todavia esse é um motivo pelo qual as aplicações se mostram resiliêntes e performáticas, isso porque as aplicações mesmo sob estresse mantiveram entregando os conteúdos numa frequência menor mas sem cair. Não basta responder rápido, tem que suportar o número alto de acesso, e continuar respondendo de maneira rápida. Os gráficos a seguir demonstram quanto a aplicação Microserviço se manteve a frente da aplicação Monolitica. As técnicas de I/O não bloqueante e assíncronas, de criação de coroutinas fizeram com que a aplicação Microserviço se mantivesse melhor em todos os cenários de teste (NC1, NC2, NC3, NC4, NC5, NC6).

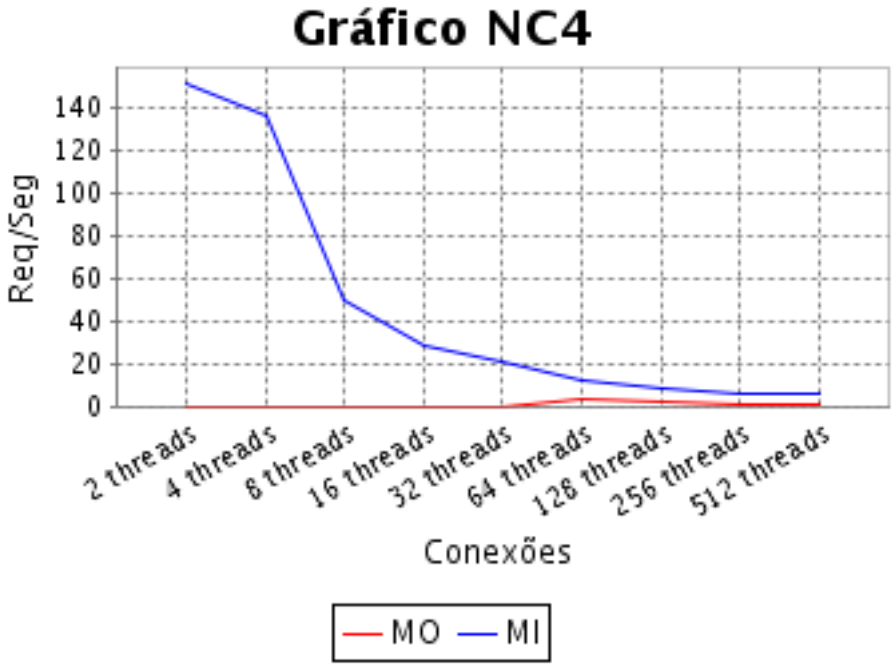

Figura 21 – Comparativo - NC4

Fonte: Próprio autor - Google Chart

Nota-se que o gráfico 21 demonstra uma diferença de 184900% na sequência NC4 com 2 threads entre as aplicações, a aplicação Monolitica sofre e tem resultados que beiram a zero, na maior parte do tempo sua média de requests por segundo se mantiveram em zero. Isto significa, que a aplicação cortou conexões e não conseguiu processar nenhum request sequer durante os 10 segundos de testes realizados. A aplicação Monolitica tem a concorrência de outros serviços, bem como o admin, área responsável por criação de matéria, gerenciamento de usuários. Criar uma matéria concorre com um processo aberto internamente para lidar com um request vindo do cliente, além disso, a aplicação Monolítica muitas vezes esta amarrada a um framework facilitador, neste caso o Django, um framework totalmente síncrono que acabar sendo um ponto de lentidão na aplicação. Outras técnicas passam a ter que ser utilizadas

para escalar a aplicação, tais quais: *load balance*, servidores de *frontend* e *backend* separados, separação de admin da aplicação, cacheamento de dados na camada de aplicação e na camadas de frontend, decerto essas necessidades de técnicas começam a desenhar para um cenário mais modular o qual é possível dividir responsabilidade entre todo o ecossistema.

Conforme o gráfico 22 a aplicação Microserviço se mostrou capaz de lidar com um número muito acima, 312400% a mais de requisições no ínicio com 2 threads e 130.06% com 32 threadhs durante o experimento NC5.

Os gráfico a seguir representa a rodada NC5.

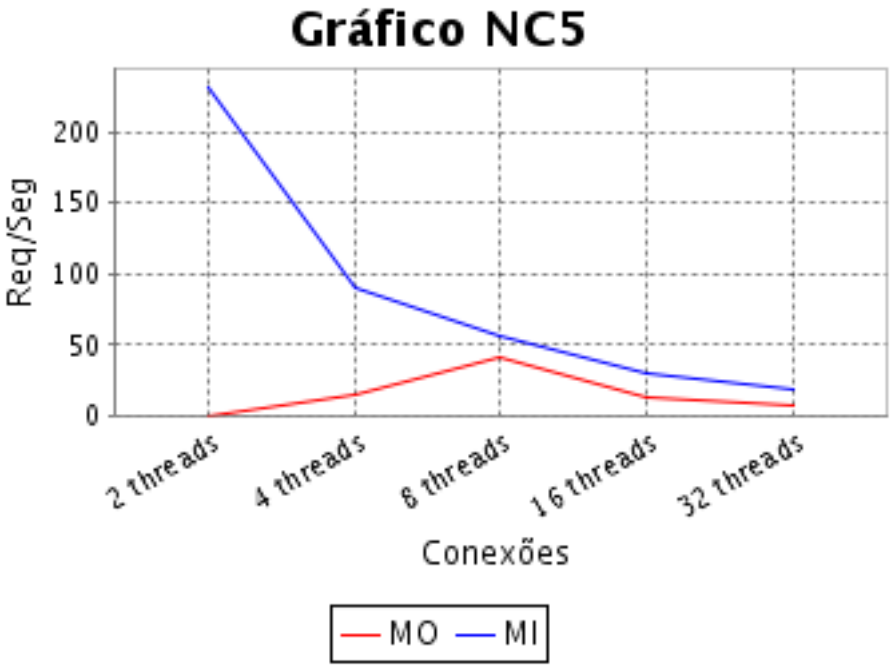

Figura 22 – Gráfico NC5 Comparativo

Fonte: Próprio autor - Google Chart

Após os testes as métricas de esforço computacional de ambas aplicações demonstrou que a aplicação Monolítica chegou a ter picos acimas de 40% de capacidade de CPU enquanto a aplicação Microserviço não chegou a 30%. Lembrando que ambas tem a mesma capacidade máquina, como citados no capítulo 5. As imagens abaixo tangibilizam melhor o esforço computacional de ambas aplicações.

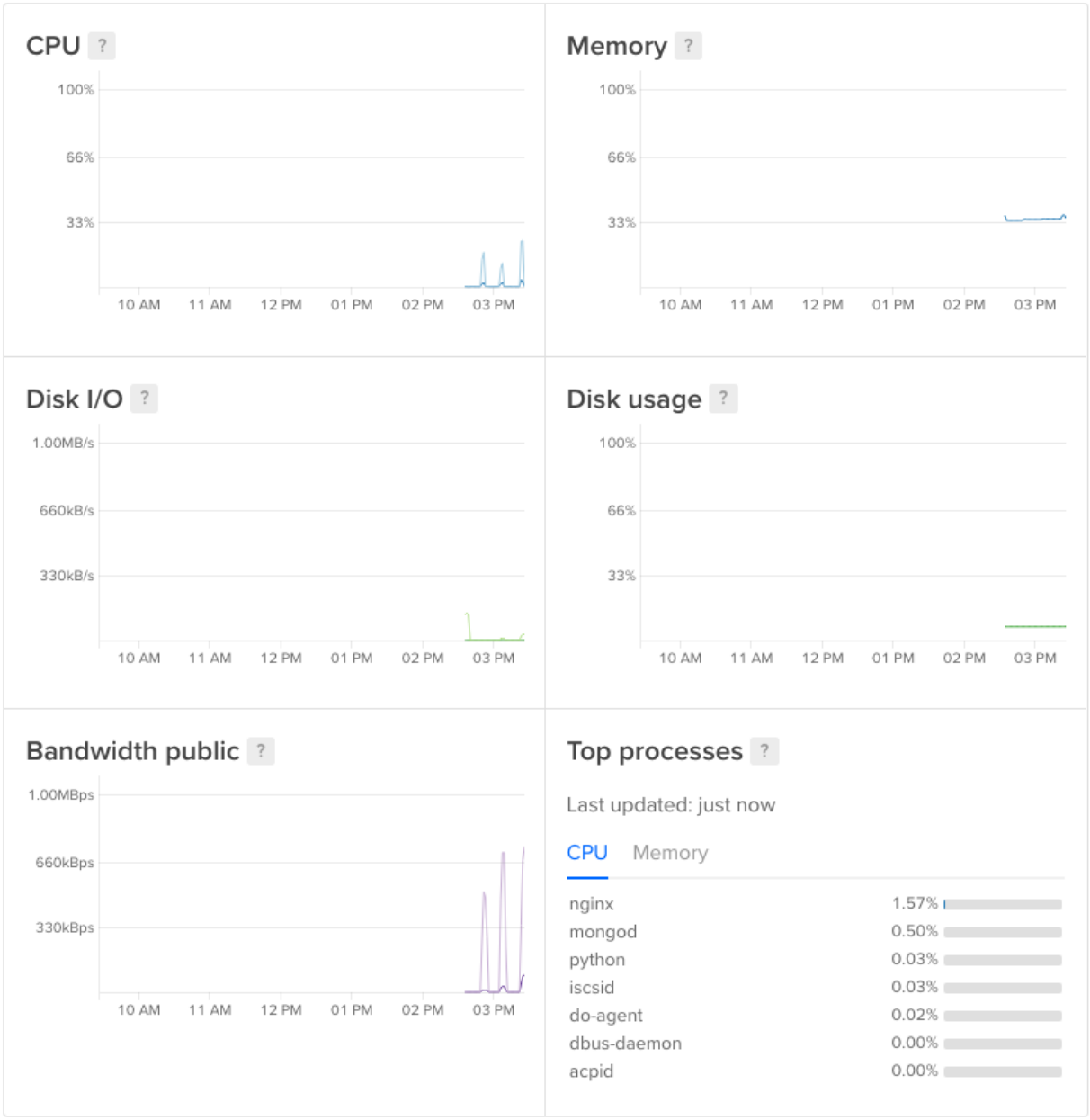

Figura 23 – Aplicação Microserviço - Esforço computacional

Fonte: Próprio autor - DigitalOcean

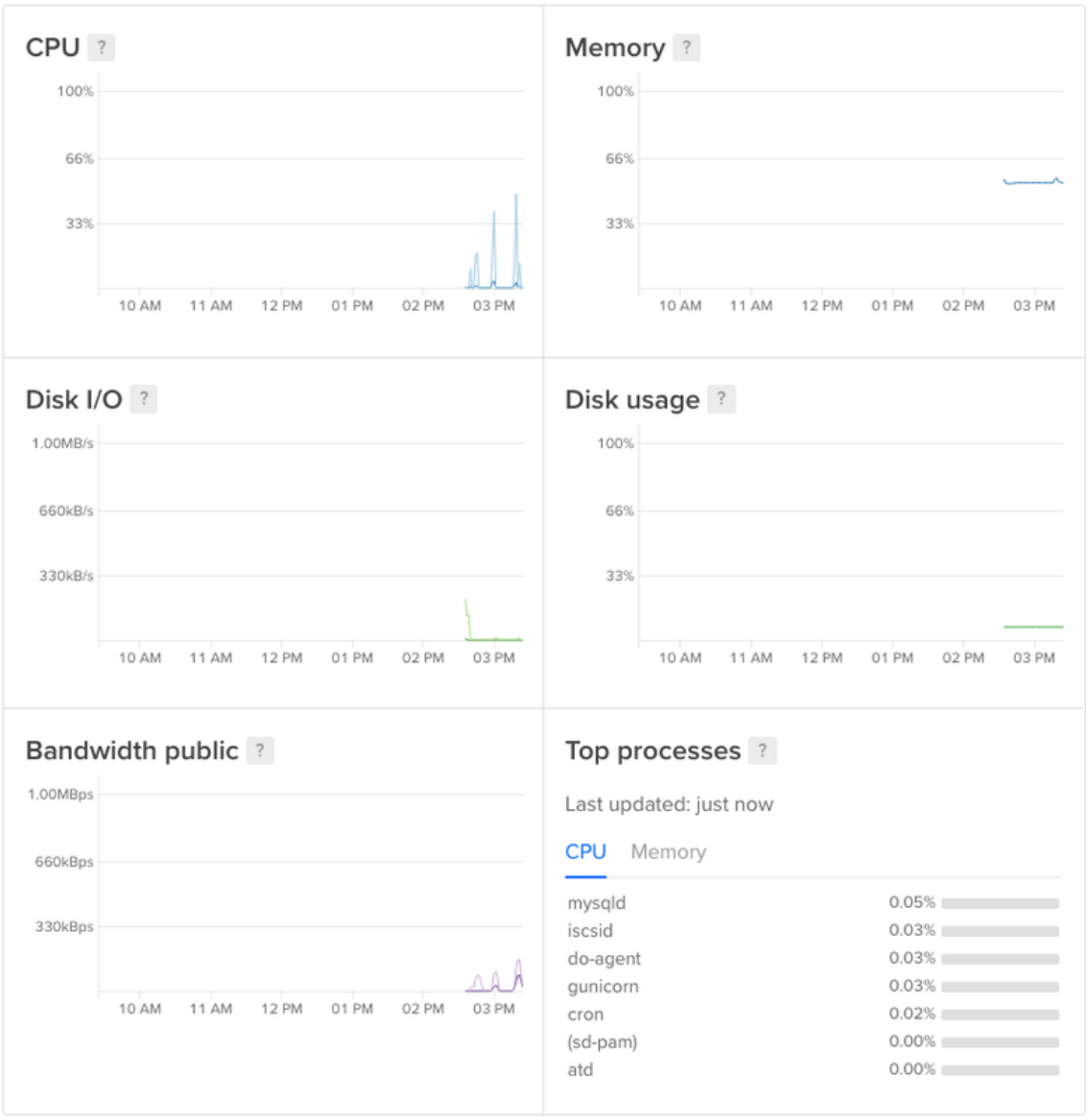

Figura 24 – Aplicação Monolítica - Esforço computacional

Fonte: Próprio autor - DigitalOcean

Em conclusão as imagens 23 e 24 pode-se percebe que algumas diferenças ente ambas aplicaões, ao analisar-se primeiramente o uso de CPU, nota-se que a aplicação Microserviço manteve em todo teste os ciclos de CPU abaixo de 33% enquanto a aplicação Monolítica chegou a mais de 50% em alguns momentos. Outro ponto de analise válido é a capacidade de processar mais dados, percebe-se que no gráfico que representa o Bandwith os picos da aplicação Microserviço são de um pouco mais de 660KBps enquanto a aplicação Monolítica não passou de 300 KBps, isso representa 54.5% a menos de eficiência em receber dados pela rede, provavelmente isso da pelo corte de conexão com muitas requisições como já citados anteriormente e denominado como timeout.

Conforme proposto, os testes tiveram o objetivo de levar as aplicações ao máximo de estresse possível, seja por máximo de capacidade dos servidores ou da máquina provedora, sendo assim, testou-se um cenário máximo permitid pela máquina(Mac 4GB de RAM) usada para executar os testes, como descrito no capítulo6, o teste executado foi NC6 com 2 threads(capacidade máxima também permitida dado o tamanho de acesso), para um melhor acompanhamento dos números, o teste encontra-se detalhado e completo no apêndice.

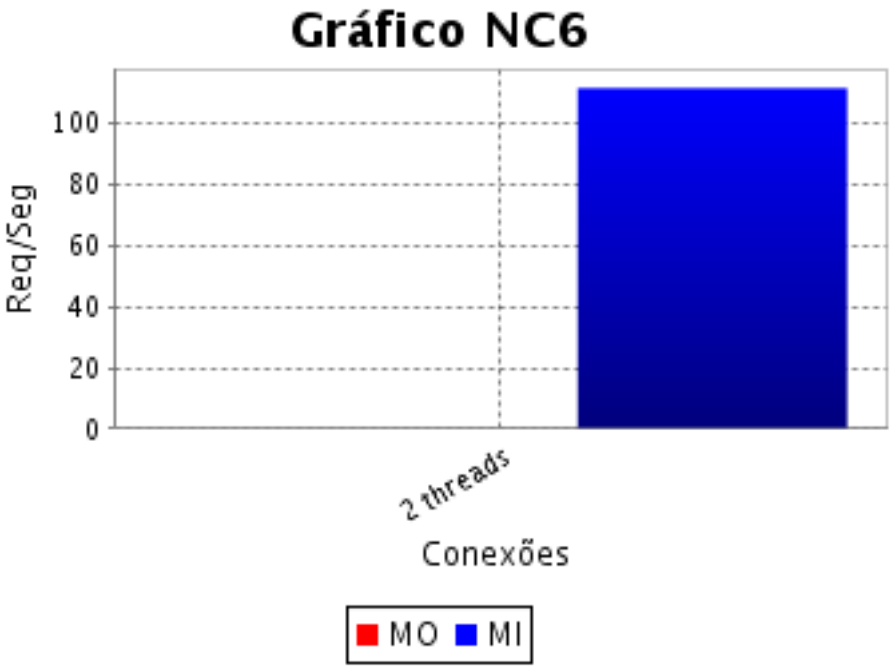

Figura 25 – Comparativo - NC6
Fonte: Próprio autor - Google Chart

De acordo com a imagem 25 percebe-se que a aplicação Monolítica simplesmente não conseguiu responder a nenhuma requisição, foi totalmente ineficiente, todavia, a aplicação Microserviço suporta mais de 100 requisições por segundo, além de um latência média abaixo de um segundo.

Com a finalidade de buscar equalização e solidificação nos resultados presente, buscou-se outra ferramente para medição de performance server side, o apache benchmark (AB) é focado em medir request/sec (requisições por segundo). Em seu uso, dados como latência são despresados e apenas informações da capacidade de aguentar requisições são catalogadas. Buscando uma simplificação dos dados filtrou-se alguns dados para uma melhor explicação da análise feita durante os testes. Foram considerados as requisições por segundo, o tempo entre uma requisição e outra e o total de requisições executadas em todo o teste.

A grande diferença entra as ferramentas é que no apache benchmark os testes são executados até atingir o número máximo de requisições solicitadas, por esse motivo

os testes demoram mais para executar, já o WRK dentro de um período que por default é de 10 segundos ele estressa ao máximo a aplicação para atingir todas as requisições e threads escolhidas na configuração de execução.

Os critérios de análise seguiram os modelos já apresentados anteriormente na tabela 1. Os gráficos mantiveram a nomenclatura apenas adicionado a sigla AB proveninente de apache benchmark que é o nome da ferramenta utilizada para teste.

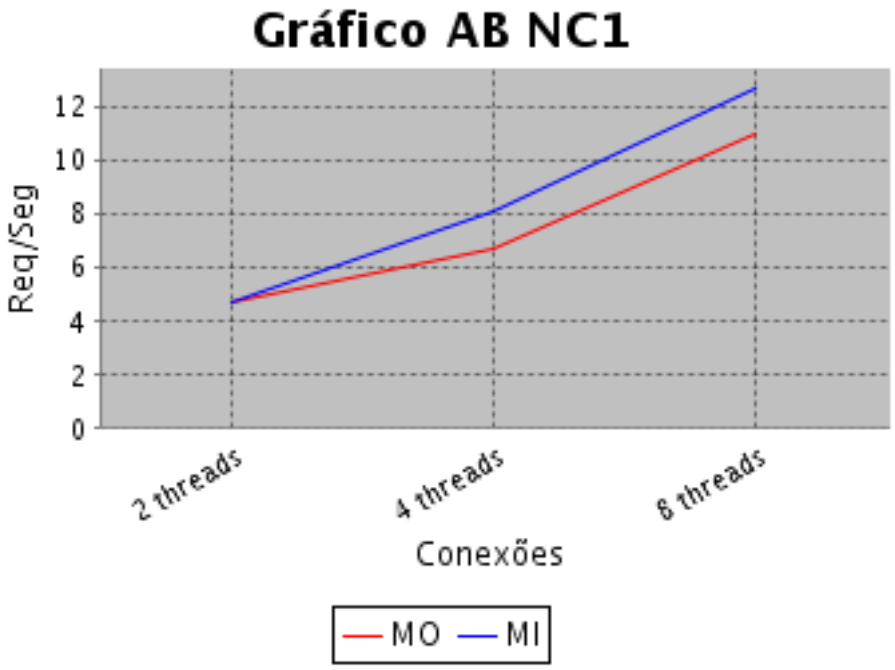

Figura 26 – Comparativo AB NC1
Fonte: Próprio autor - Google Chart

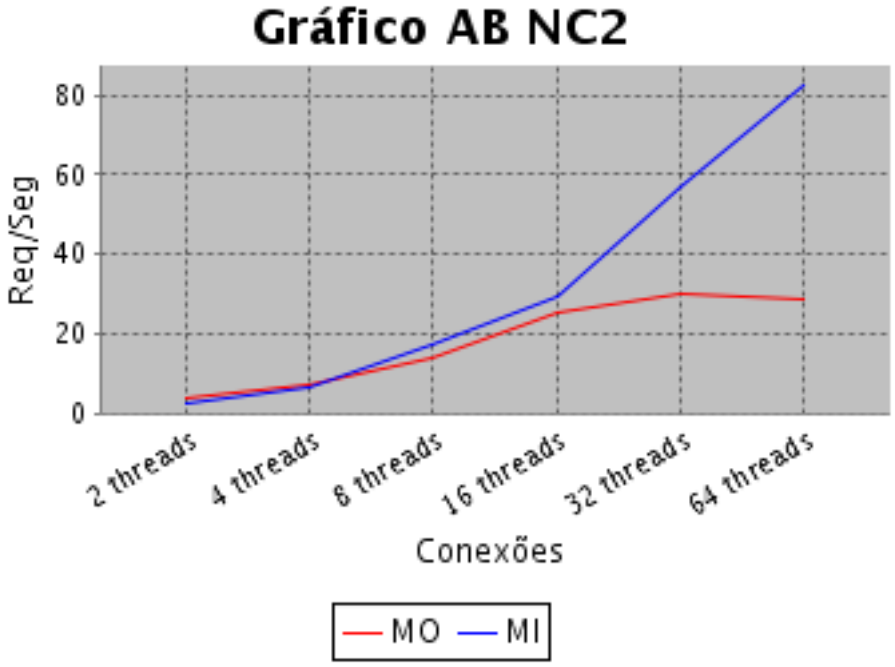

Figura 27 – Comparativ AB NC2

Fonte: Próprio autor - Google Chart

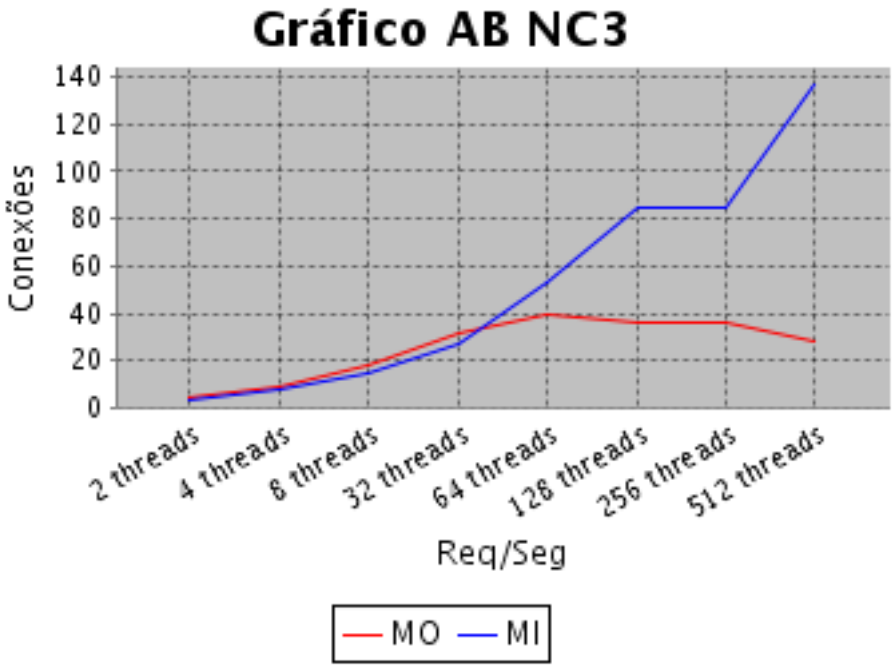

Figura 28 – Comparativo AB NC3
Fonte: Próprio autor - Google Chart

Ao analizar-se os experimentos anteriores referentes as sequências NC1, NC2 e NC3 utilizando a ferramenta AB percebeu-se um comportamente um pouco diferente com relação aos testes realizados usando a ferramenta WRK, ambas as aplicações demonstram um número de requisições por segundo muito baixo com o menor número de requisições paralelas e foram aumentando conforme esta concorrência foi sendo aumentada. Ao traçar um comparativo, na sequência NC1 da aplicação Monolítica com 2 threads a diminuição foi de 92.72% a menos de capacidade, enquanto a aplicação Microserviço teve uma diminuição de 93.36% no mesmo cenário de experimento.Em alguns momentos durante os teste a aplicação Monolítica foi melhor em mais ou menos 22% nos cenarios de poucos threads, todavia conforme o aumento de requisições paralelas a distância entre as aplicações voltou a ficar grande, e a aplicação Microserviço voltou a ter êxito no comparativo de requisições por segundo, chegando a ter 475% mais capacidade de requisição por segundo que a aplicação Monolítica.

De fato ambas aplicações não diferem muito em termos de tecnologia, ambas utilizam o mesmo tamanho de infra, possuem a mesma versão de SO, utilizam a mesma linguagem de programação.A diferença entre ambas aplicações se restringi em framework e o banco de dados, todavia, a grande diferença entre ambas aplicações é sim, o fato de uma ser muito mais voltada para um gerenciamento de todo o

sistema(Monolítico) e a outrao voltada para ser não bloqueante, ter eventos assíncronos e suportar mais o acesso em massa (Microserviço), pequenos detalhes podem potêncialmente trazer grandes benefícios ao desenvolver uma aplicação de grande porte, como pode-se ver a aplicação Microserviço chega perto dos 140 req/seg como mostrado no gráfico 28

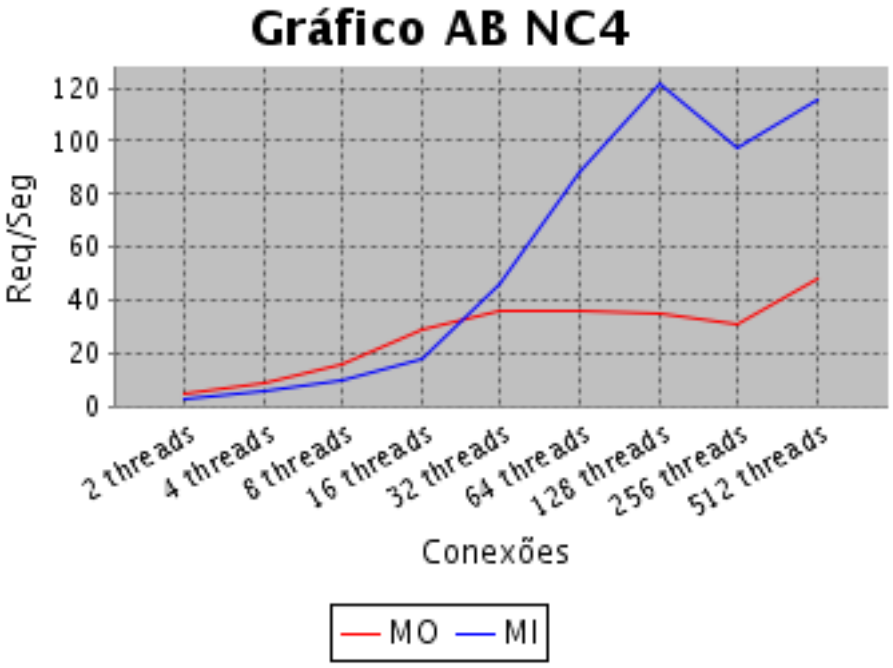

Figura 29 – Comparativo AB NC4

Fonte: Próprio autor - Google Chart

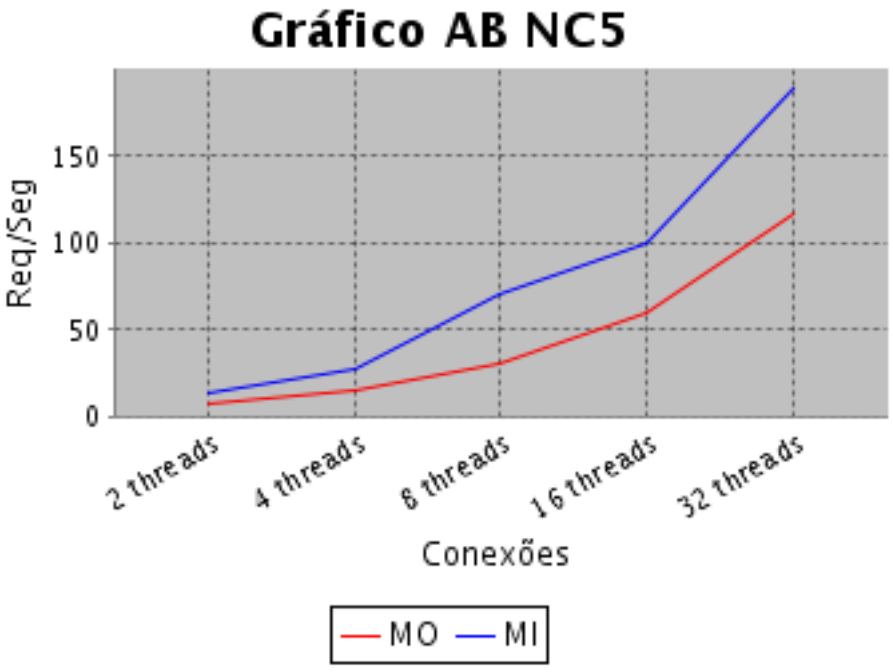

Figura 30 – Comparativo AB NC5

Fonte: Próprio autor - Google Chart

Ao analisar todas as rodadas descritas anteriormente com o Apache Benchmark nota-se que a aplicação Microserviço se mostrar abaixo em quase todas as rodadas no ínicio dos testes por volta de 4 threads até 32 threads, de fato, este tipo de comportamente não foi visto ao usar-se a ferramenta de testes WRK, ainda assim não encontrou-se razões para explicar este comportamento. O teste AB NC5 demonstrou um padrão já mostrado nos teste usando o WRK, a aplicação Microserviço se mantave a todo momento a frente, cerca de 81.81% a mais em número de requisições e capacidade de processamento.

Os testes NC6 foram descartados da amostra total, uma vez que o teste AB por ter um tempo de execução muito longo, após mais de um dia rodando o esultado não foi finalizado devido a um erro de interrupção, por conseguinte entendeu-se que já havia amostra o suficiente e optou-se então pelo descarte do mesmo na apresentação deste trabalho.

Desde o ínicio este trabalhou visou-se nortear a importância de pequenas decisões na construção de serviços performáticos e resilientes. Questionar se de fato a arquitetura definida à aplicação pode torná-la performática e qual é o fator primordial para esta decisão? Sabe-se que há diversas técnicas para otimizar uma aplicação e nenhuma delas foi considerada durante os testes, a grande informação coletada é de que decisões iniciais já desenham e credenciam uma aplicação a ser performática.

A partir dos resultados encontrados ao longo deste trabalho e pela pequena diferença entre as aplicações desenvolvidas uma última análise foi feita com a troca apenas da linguagem; criou-se então, um microserviço com os mesmos critérios do que foi feito em python, todavia, utilizando Golang.

Utiizou-se a mesmas condições de infraestrutura já citada anteriormente no Capítulo 4 figura 4 ; assim como as configurações de Nginx e o mesmo banco de dados, Mongo.

Golang é uma linguagem criado em 2009 pela google com objetivo de ser super performática e considerada uma linguagem de programação concorrente. Criada por Robi Pike[2], Ken Thompson[3] e Rober Griesermer[4] o Golang foi projeto para ser uma

[2] Robert C. Pike (1956) é um engenheiro de software e escritor. Foi responsável pelo projeto dos sistemas operacionais Plan 9 e Inferno, e da linguagem de programação Limbo, quando trabalhou na equipe que desenvolveu o sistema Unix, nos laboratórios Bell.Com Ken Thompson criou o padrão UTF-8.(CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE.,)

[3] Um dos criadores do Sistena Operacional Unix, criador da linguagem de programação B. Ganhador de vários premios técnologicos, tal qual: Prêmio Turing

[4] Pesquisador Google - Distributed Systems and Parallel Computing . Criador do UTF-8 junto com Robi Pike(CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE.,). E um dos criadores

linguagem de alta performance.

> “Go é expressivo, conciso, limpo e eficiente. Seus mecanismos de concorrência tornam mais fácil escrever programas que tirarão o máximo proveito de vários núcleos e máquinas em rede, enquanto o seu novo sistema de tipos permite a construção flexível e modular do programa. Go compila rapidamente para código de máquina ainda tem a conveniência de garbage collection e o poder de reflexão em tempo de execução. É uma linguagem compilada rápida, de tipagem estática, que parece uma linguagem interpretada digitada de forma dinâmica.” (CREATIVE COMMONS ATTRIBUTION 3.0 LICENSE,)

Uma pequena amostra foi feita para comparar diferentes tempos de resposta, por isso, apenas as rodadas NC1 e NC2 foram representadas abaixo; pequenas decisões podem influenciar no resultado final da aplicação.

A amostra foi coletada usando o ferramenta WRK apenas por questões de concisão; Entendeu-se que apenas NC1e NC2 seriam suficientes para as demonstrações abaixo, o resultado mais detalhados desses experimentos encontram-se no apêndice do trabalho. A aplicação Microserviço feita em Golang ganhou a sigla MI-GO nas descrições dos gráficos a seguir para melhor compreensão.

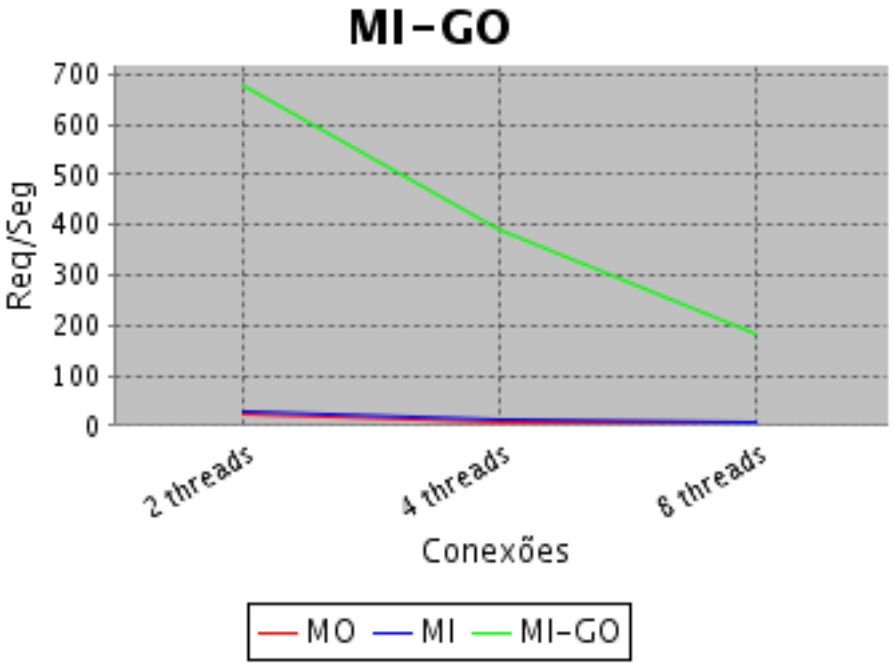

Figura 31 – Comparativo de linguagens NC1

Fonte: Próprio autor - Google Chart

da JVM (Java Virtual Machine)

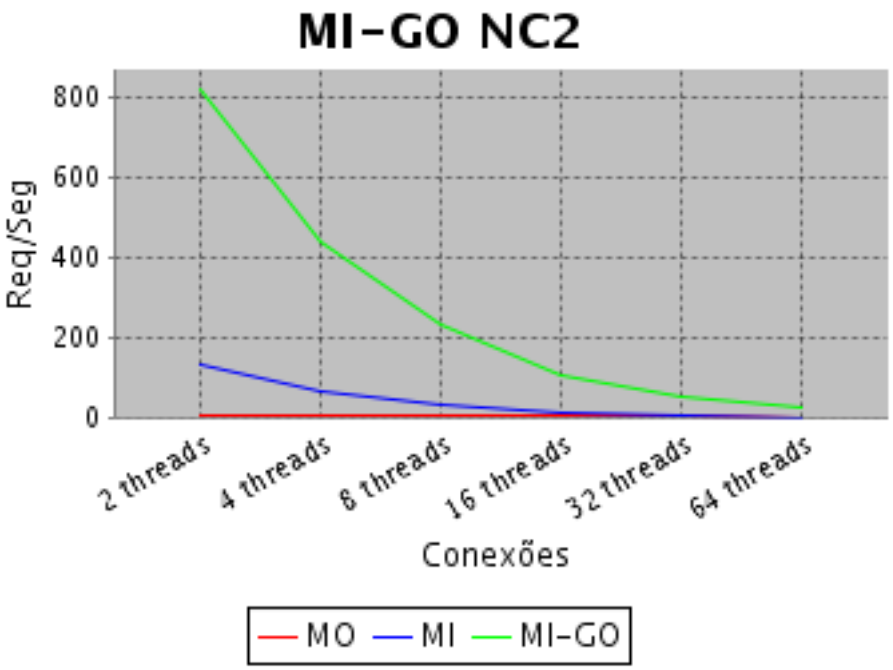

Figura 32 – Comparativo de linguagens NC2

Fonte: Próprio autor - Google Chart

Em síntase uma breve analise foi tirada das duas amostras citadas acima. Nota-se a descrepância entre os resultados, a aplicação MI-GO chegou a bater mais de 800 req/seg muito superior aos 133 req/seg atingidos pela MI em seu maior pico, isto representa um aumento de mais de 501.50%. No primeiro gráfico31 a menor capacidade da aplicação MI-GO foi de um pouco menos de 200 req/seg, para ser preciso, 183 req/seg o que de fato representa o maior pico da aplicação Microserviço. Após as amostras coletadas denote-se o quão impactante uma pequena escolha pode influenciar tanto na performance de uma aplicação.

7. CONCLUSÃO

Em conclusão do desenvolvimento presente nesse estudo, percebe-se a importância do conhecimento do desenvolvedor sob os aspectos usados para construção do sistema, dominar a linguagem de programação pode ser um fator preponderante para alcançar-se performance, ademais, entender a melhor estrutura para o banco de dados, se é adequado ou não usar um banco relacional ou simplesmente utilizar um banco não relacional, em qual cenário um será mais efetivo que o outro, todas essas técnicas potêncialmente podem ser um fator diferencial na busca de desempenho e resiliência.

Melhores técnicas de cacheamento, tratamento de erros, redundâncias e entre outras milhares técnicas são outros fatores, que mesmo não contemplados neste trabalho são importantes na evolução do sistema, obter resiliência e desempenho não é proviniente apenas na adoção de um sistema independente. O microserviço traz consigo desafios, o cuidado para evitar que o micro se torne nano-serviço e todo o ecossistema vire um caos para gerir. Da mesma maneira que o micorserviço pode diluir as aplicações, ele pode se tornar o ponto de dificuldade do sistema, o aumento de ponto de falhas e round trip time conhecido como RTT, que está relacionado a latência é um dos maiores problemas encontrados àqueles que adotam o microserviço em demasia.

Este trabalho restringiu-se em mostrar que muito além de todas as técnicas, as mais simples permitem que um sistema, mesmo que com pouco capacidade de infra, sem técnicas de cache ou load balance, pode representar números significativos para ter-se em produção. Potêncialmente nenhuma das aplicações construídas ao decorrer deste trabalho estariam preparadas para estar em produção sem técnicas de cacheamento e redundância, uma vez que pode-se considerar essas técnicas como vitáis para criação de qualquer aplicação.

Tendo em vista nenhum estudo academico encontrado voltado para performance server-side e seus componentes de construção, este trabalho visou agregar grande quantidade de assuntos e temas abordados durante o curso para executar testes que pudessem mostrar como a técnologia sempre irá se fundamentar nos pequenos detalhes para atingir-se grandes objetivos. Os resultados atingidos durante a construção deste trabalho foram satisfatório dado ao escopo proposto. Um ponto de colaboração importante que este trabalho pode trazer foi o mapeamento de cenários mais contemporâneos de um desenvolvedor web em geral, como também, detalhar funcionamentos de alguns processos na construção de um software e mostrar como um desenvolvedor deve estar atento a pequenos detalhes e conhecer profundamente as ferramentas de trabalho para atingir objetivos grandes em termos de performance e resiliência.

Em suma o fato da aplicação Microserviço lidar com requisições assíncronase não bloqueantes fez com que a performance fosse muito melhor que a aplicação Monolítica que por sua vez utiliza técnicas de processo síncrono. Os componentes que envolvem a construção de uma aplicação são eventualmente preponderantes para o alcance de desempenho e resiliência, pode-se concluir que ser um microserviço dentro de um ecossistema não faz da aplicação algo performático, ademais, ter um sistema operacional dedicado também não faz a aplicação ter alto desempenho, apesar de ter uma melhora. É provavel que pequenas decisões bem como a linguagem de programação, a maneira como utiliza-se a linguagem, framework escolhido tragam ganhos que resultam consequentemente em ganho ou perda à aplicação dependendo da decisão tomada.

O grande objetivo deste trabalho não foi seguir o óbvio e provar que uma aplicação com técnicas de assíncronismo e com um sitema operacional dedicado é possivelmente mais rápido, pois isso parece ser um pouco óbvio demais, entretanto mostrar que performance e escalabilidade não se conquista apenas com a adoção de um microserviço e nem tão pouco com as técnicas x,y ou z. Cada detalhe é preponderante no objetivo de tornar uma aplicação escalável, conhecer as limitações da linguagem de programação escolhida, entender como é possível dentro dela otimizar e maximizar a performance de uma aplicação. Durante os testes percebeu-se que a aplicação Microserviço que usa python e contém as limitações já citadas no capitulo 5.3 conseguiu atingir marcas de mais de 140 requisições por segundo sem nenhuma técnicas de cache ou load balance, esse número pode-se considerar representativo, tendo como base 140 requisições por segundo, em um minuto são 8.400 acessos, ao aumentar essa hipótese para um dia de acesso, são cerca de 12.096.000 acessos, ou seja, uma simples máquina de 1 GB de RAM com configurações simples sem nenhuma técnica de cache é capaz de aguentar mais de 12 milhões de acessos por dia.

Para trabalhos futuros, entende-se que pode ser feito uma analise mais abrangente, considerando técnicas que neste trabalho foram descartados, tais quais, load bolance, uso de cache efetivo da aplicação e do webserve. Entende-se que as técnicas de cache são vitáis para construção de sistemas resiliêntes, essas técnicas possivelmente podem melhorar significativamente os resultados aqui encontrados. Estudos que venha enriquecer de maneira mais macro e mais arquitetural a construção de sistemas que buscam desempenho e resiliência, além disso, outros trabalhos que visam trabalhar a performance não só server side porém o tempo total de resposta de uma aplicação, desde o processamento até a renderização ao cliente, a performance abrange também o tempo em que o usuário consegue perceber a resposta de uma aplicação, este trabalho tem como base possíveis estudos voltados para uma performance client side.

REFERÊNCIAS

ADRIANO ALMEIDA - CONSULTOR, DESENVOLVEDOR E INSTRUTOR DA CAELUM. *Arquitetura de microserviços ou monolítica?* 2015. Disponível em: <http://blog.caelum.com.br/arquitetura-de-microservicos-ou-monolitica/>. Acesso em: 17/03/2015. Citado na página 23.

Casa do Código (Ed.). *REST - Construa APIs inteligentes de maneira simples*. 1. ed. [S.l.]: Casa do Código. Citado 5 vezes nas páginas 16, 17, 18, 19 e 20.

BASE64 Decode and Encode. Site usado para simular exemplo. Disponível em: <https://www.base64encode.org/>. Citado na página 18.

BENOIT CHESNEAU. *Gunicorn documentation* . Disponível em: <http://gunicorn.org/>. Citado na página 36.

CGI.BR. *PESQUISA SOBRE O USO DAS TECNOLOGIAS DE INFORMAÇÃO E COMUNICAÇÃO NOS DOMIC´LIOS BRASILEIROS*. Disponível em: <http://www.cgi.br/media/docs/publicacoes/2/TIC_Dom_2015_LIVRO_ELETRONICO.pdf>. Acesso em: 03/04/2017. Citado na página 12.

CREATIVE COMMONS ATTRIBUTION 3.0 LICENSE. *Documentação da linguagem de progamação Go*. Disponível em: <http://www.golangbr.org/doc/>. Acesso em: 29/04/2017. Citado na página 72.

CREATIVE COMMONS ATTRIBUTION 4.0 INTERNATIONAL LICENSE. *Who invented UTF-8?* Acesso em: 27/04/2017. Citado na página 71.

DEN GOLOTYUK - ENGINEER AT .IO. *“We scaled our app to 700 million requests per day in just 6 months on top of DigitalOcean infrastructure.”*. Disponível em: <https://www.digitalocean.com/customers/io/>. Citado na página 34.

Django Software Foundation. *Loose coupling*. Django Software Foundation. Documentation. A fundamental goal of Django’s stack is loose coupling and tight cohesion. The various layers of the framework shouldn’t “know” about each other unless absolutely necessary. Disponível em: <https://docs.djangoproject.com/en/1.10/misc/design-philosophies/>. Citado na página 22.

DOUG HELLMANN. *Asynchronous Concurrency Concepts*. Disponível em: <https://pymotw.com/3/asyncio/concepts.html>. Citado na página 50.

FUNCTIONAL SOFTWARE, INC. *Sentry’s real-time error tracking gives you insight into production deployments and information to reproduce and fix crashes*. Disponível em: <https://sentry.io/welcome/>. Citado na página 23.

IRAKLI NADAREISHVILI. In: ____. *Microservice Architecture*. 1. ed. [S.l.]. cap. 1. ISBN 978-1-4919-5621-2. Citado 2 vezes nas páginas 13 e 14.

JONAS BONÉR. Citado 2 vezes nas páginas 31 e 32.

LAW, C. Architectures, Coordination, and Distance. Citado na página 24.

LIGHTBEND INC. *Akka IO*. Disponível em: <http://akka.io/>. Acesso em: 02/04/2017. Citado na página 32.

Jorga Zahar Editor (Ed.). *A Galáxia da Internet*. [S.l.]: Zahar. Citado na página 12.

MOTOR: Asynchronous python driver for mongodb. Disponível em: <https://motor.readthedocs.io/en/stable/>. Citado na página 27.

MUHAMMAD YASOOB ULLAH KHALID. *Generators*. Disponível em: <http://book.pythontips.com/en/latest/generators.html>. Citado na página 49.

NADAREISHVILI, I. et al. *Microservice Architecture*. First edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2016. Disponível em: <https://www.amazon.com/Microservice-Architecture-Aligning-Principles-Practices/dp/1491956259>. Citado 4 vezes nas páginas 23, 24, 26 e 27.

NEW RELIC, INC. *New Relic gives you deep performance analytics for every part of your software environment*. Disponível em: <https://newrelic.com/>. Citado na página 23.

NEWMAN, SAM. Building Microservices. In: O'Reilly Media, Inc. (Ed.). [S.l.]: O'Reilly Media, Inc., 2015. Citado na página 13.

PYTHON SOFTWARE FOUNDATION. *Global Interpreter Lock*. 2017. Disponível em: <https://docs.python.org/3/glossary.html#term-gil>. Citado na página 49.

RICHARDSON, S. R. L. Restful web services. 2007. Citado na página 17.

SILBERSCHATS, GALVIN E GAGNE. *Conceitos de Sistema Operacional com Java* . 7 edição. ed. [S.l.], 2006. Citado na página 50.

THE TORNADO AUTHORS. *Asynchronous and non-Blocking I/O*. Disponível em: <http://www.tornadoweb.org/en/stable/guide/async.html>. Acesso em: 03/02/2017. Citado na página 51.

THE TORNADO AUTHORS. *Introduction*. Disponível em: <http://www.tornadoweb.org/en/stable/guide/intro.html>. Citado na página 50.

THE TORNADO AUTHORS. REVISION C0F99BAC. *Most applications should use AsyncIOMainLoop to run Tornado on the default asyncio event loop. Applications that need to run event loops on multiple threads may use AsyncIOLoop to create multiple loops.* Disponível em: <http://www.tornadoweb.org/en/stable/asyncio.html>. Citado na página 35.

WIKIPÉDIA a enciclopédia livre. *Material de aula de Sistemas operacionais*. Disponível em: <https://pt.wikipedia.org/wiki/Condi~A\T1\textsection~A\T1\textsterlingo_de_corrida>. Citado na página 50.

Apêndices

As tabelas abaixos representam os stdout do todos os testes executados usando WRK, as siglas abaixo tem os seguintes significados:

1) AVG - Avarage, em tradução livre de contexto significa média.
2) STDEV - Estima o desvio padrão com base em uma amostra. O desvio padrão é uma medida de quão amplamente os valores estão dispersos do valor médio (a média).
3) MAX - O máximo de latência e requisições/segundo atingido no teste

- *Experimento NC1 Monolítico*

NC1 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	238.38ms	68.13ms	538.54ms	77.02%
Requisições/Segundo	22.19	11.62	50.00	53.59%

Fonte: Próprio autor

Tabela 3 – Total de transferência do teste - Aplicação Monolítica

406 requisições feitas em 10.07s	2.59MB read
Requisições/segundo	40.23
Transferência/segundo	364.99KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 4 – NC1 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	199.06ms	47.32ms	477.71ms	83.25%
Requisições/Segundo	10.34	4.26	20.00	72.83%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 5 – Total de transferência do teste - Aplicação Monolítica

394 requisições feitas em 10.01s	2.48MB read
Requisições/segundo	39.36
Transferência/segundo	356.33KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 6 – NC1 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	199.30ms	52.44ms	755.97ms	82.35%
Resquisições/Segundo	5.57	5.41	48.00	97.81%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 7 – Total de transferência do teste - Aplicação Monolítica

397 requisições em 9.32s	3.51MB read
Requisições/Segundo	42.59
Transferência/Segundo	385.57KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC2 Monolítico

Tabela 8 – NC2- 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1025.00ms	366.48ms	1091.00ms	68.75%
Resquisições/Segundo	7.88	5.92	30.00	76.39%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 9 – Total de transferência do teste - Aplicação Monolítica

99 requisições em 9.32s	2.10MB read
Requisições/Segundo	9.84
Transferência/Segundo	213.83KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 10 – NC2- 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1022.00ms	475.76ms	1099.00ms	51.06%
Resquisições/Segundo	5.90	5.11	20.00	84.31%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 11 – Total de transferência do teste - Aplicação Monolítica

130 requisições em 9.32s	2.50MB read
Requisições/Segundo	12.93
Transferência/Segundo	254.65KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 12 – NC2- 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1013.00ms	579.25ms	1097.00ms	52.63%
Resquisições/Segundo	5.13	6.29	30.00	91.43%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 13 – Total de transferência do teste - Aplicação Monolítica

125 requisições em 10.06s	2.40MB read
Requisições/Segundo	12.43
Transferência/Segundo	244.51KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 14 – NC2 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1.32s	264.01ms	1.65s	89.43%
Requisições/Segundo	6.12	5.36	30.00	92.12%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 15 – Total de transferência do teste - Aplicação Monolítica

612 requisições em 10.08s	5.94MB read
Requisições/Segundo	66.64
Transferência/Segundo	603.36KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 16 – NC2 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1.36s	305.04ms	1.88s	78.69%
Requisições	3.35	3.56	20.00	81.45%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 17 – Total de transferência do teste - Aplicação Monolítica

644 requisições em 10.10s	5.69MB read
Requisições/Segundo	63.77
Transferência/Segundo	577.31KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 18 – NC2- 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	979.02ms	293.88ms	1.89s	73.65%
Requisições/Segundo	0.69	0.49	2.00	67.16%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 19 – Total de transferência do teste - Aplicação Monolítica

612 requisições em 10.09s	5.41MB read
Requisições/Segundo	60.63
Transferência/Segundo	548.85KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- *Experimento NC3 Monolítico*

Tabela 20 – NC3 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1029.00ms	337.65ms	1087.00ms	56.25%
Resquisições/Segundo	12.50	8.00	42.00	71.43%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 21 – Total de transferência do teste - Aplicação Monolítica

202 requisições em 10.11s	3.02MB read
Requisições/Segundo	19.98
Transferência/Segundo	306.38KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 22 – NC3 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1010.00ms	493.49ms	1096.00ms	60.00%
Resquisições/Segundo	8.35	5.90	40.00	78.18%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 23 – Total de transferência do teste - Aplicação Monolítica

276 requisições em 10.31s	4.71MB read
Requisições/Segundo	26.78
Transferência/Segundo	254.65KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 24 – NC3 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1022.00ms	473.76ms	1097.00ms	61.70%
Resquisições/Segundo	5.74	5.71	40.00	88.89%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 25 – Total de transferência do teste - Aplicação Monolítica

267 requisições em 10.11s	4.07MB read
Requisições/Segundo‘	26.42
Transferência/Segundo	412.33KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 26 – NC3 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1023.00ms	471.72ms	1099.00ms	66.67%
Resquisições/Segundo	3.62	4.20	30.00	81.42%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 27 – Total de transferência do teste - Aplicação Monolítica

249 requisições em 10.09s	3.78MB
Requisições/Segundo	24.69
Transferência/Segundo	383.35KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 28 – NC3 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1021.00ms	443.85ms	1026.00ms	58.33%
Resquisições/Segundo	2.53	3.51	20.00	87.01%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 29 – Total de transferência do teste - Aplicação Monolítica

252 requisições em 10.15s	3.83MB read
Requisições/Segundo	24.82
Transferência/Segundo	386.51KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 30 – NC3 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1200.00ms	480.86ms	1970.00ms	53.85%
Resquisições/Segundo	3.41	4.75	20.00	80.72%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 31 – Total de transferência do teste - Aplicação Monolítica

270 requisições em 10.10s	4.07MB
Requisições/Segundo	26.74
Transferência/Segundo	412.41KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 32 – NC3- 128 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1220.00ms	540.66ms	2000.00ms	57.14%
Resquisições/Segundo	1.98	3.83	29.00	89.25%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 33 – Total de transferência do teste - Aplicação Monolítica

625 requisições em 10.10s	5.53MB read
Requisições/Segundo	61.89
Transferência/Segundo	560.29KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 34 – NC3 - 256 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1100.00ms	539.08ms	2000.00ms	55.96%
Resquisições/Segundo	0.86	2.19	20.00	92.74%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 35 – Total de transferência do teste - Aplicação Monolítica

565 requisições em 10.31s	4.99MB read
Requisições/Segundo	55.95
Transferência/Segundo	506.51KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 36 – NC3 - 512 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1540.00ms	522.76ms	1990.00ms	79.75%
Resquisições/Segundo	0.08	0.31	2.00	93.21%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 37 – Total de transferência do teste - Aplicação Monolítica

677 requisições em 10.31s	5.99MB read
Requisições/Segundo	67.03
Transferência/Segundo	606.82KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC4 Monolítico

Os testes da rodada NC4 foram unificados por questão de concisão, os resultados apresentados foram insignificantes, e a maior parte dos testes retornou 0 como resultado de capacidade de resposta. Por este motivos foram unificados em uma única tabela para exemplicar seus resultados.

Tabela 38 – NC4 - 2 à 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	0.00us	0.00us	0.00us	nan%
Resquisições/Segundo	0.00	0.00	0.00	nan%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 39 – Total de transferência do teste - Aplicação Monolítica

0 requisições em 10.31s	NaN
Requisições/Segundo	0.00
Transferência/Segundo	Bytes

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 40 – NC4 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1970.00ms	22.96ms	1980.00ms	100.00%
Resquisições/Segundo	3.53	5.88	19.00	73.33%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 41 – Total de transferência do teste - Aplicação Monolítica

17 requisições em 11.38s	746.39KB read
Requisições/Segundo	1.52
Transferência/Segundo	66.76KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 42 – NC4 - 128 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1570.00ms	369.26ms	1980.00ms	71.43%
Resquisições/Segundo	1.92	3.34	10.00	83.33%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 43 – Total de transferência do teste - Aplicação Monolítica

27 requisições em 10.11s	1.19MB read
Requisições/Segundo	2.67

27 requisições em 10.11s	1.19MB read
Transferência/Segundo	120.60KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 44 – NC4 - 256 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	0.00us	0.00us	0.00us	nan%
Resquisições/Segundo	1.37	2.20	10.00	88.89%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 45 – Total de transferência do teste - Aplicação Monolítica

75 requisições em 10.36s	1.96MB read
Requisições/Segundo	7.24
Transferência/Segundo	193.KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 46 – NC4 - 512 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1350.00ms	354.23ms	1879.00	75.00%
Resquisições/Segundo	1.83	2.61	10.00	90.42%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 47 – Total de transferência do teste - Aplicação Monolítica

172 requisições em 10.15s	4.25 MB read
Requisições/Segundo	16.95
Transferência/Segundo	429.20KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC5 Monolítico

Tabela 48 – NC5 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	0.00us	0.00us	0.00us	nan%
Resquisições/Segundo	0.00	0.00	0.00	nan%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 49 – Total de transferência do teste - Aplicação Monolítica

0 requisições em 13.38s	0.00B read
Requisições/Segundo	23.67
Transferência/Segundo	214.35KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 50 – NC5 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1940.00ms	106.98ms	2000.00ms	80.0
Resquisições/Segundo	14.48	13.81	57.00	83.87%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 51 – Total de transferência do teste - Aplicação Monolítica

240 requisições em 10.14s	2.12MB read
Requisições/Segundo	23.67
Transferência/Segundo	214.34KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 52 – NC5 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	0.00us	0.00us	0.00us	nan%
Resquisições/Segundo	41.19	44.73	147.00	90.00%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 53 – Total de transferência do teste - Aplicação Monolítica

101 requisições em 10.27s	0.89MB read
Requisições/Segundo	23.67
Transferência/Segundo	214.35KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 54 – NC5 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1750.00ms	140.12ms	1910.00ms	64.62%
Resquisições/Segundo	13.68	17.77	90.00	89.81%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 55 – Total de transferência do teste - Aplicação Monolítica

288 requisições em 10.13s	2.60MB read
Requisições/Segundo	28.44
Transferência/Segundo	262.52KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 56 – NC5 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1730.00ms	48.45ms	1770.00ms	77.78%
Resquisições/Segundo	7.95	13.60	62.00	95.45%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 57 – Total de transferência do teste - Aplicação Monolítica

81 requisições em 10.67s	734.87KB
Requisições/Segundo	7.59
Transferência/Segundo	68.90KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC6 Monolítico

A aplicação Monolitica apresentou resultados insignificantes nesta rodada, todos os critérios foram zero.

Resultados dos testes da aplicação Microserviço

- Experimento NC1 Microserviço

NC1 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	161.17ms	39.66ms	409.14ms	87.26%
Requisições/Segundo	31.55	11.38	50.00	66.30%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 58 – Total de transferência do teste - Aplicação Microserviço

614 requisições em 10.09s	9.65MB read
Requisições/Segundo	60.87
Transferência/Segundo	0.96MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 59 – NC1 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	172.65ms	51.45ms	457.92ms	85.62%
Requisições/Segundo	12.20	5.13	20.00	61.69%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 60 – Total de transferência do teste - Aplicação Microserviço

458 requisições em 10.10s	7.19MB read
Requisições/Segundo	45.35
Transferência/Segundo	728.61KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 61 – NC1 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	176.95ms	64.30ms	586.11ms	86.03%
Requisições/Segundo	6.13	2.88	10.00	69.64%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 62 – Total de transferência do teste - Aplicação Microserviço

448 requisições em 10.10s	7.03MB read
Requisições/Segundo	44.37
Transferência/Segundo	712.81KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC2 Microserviço

Tabela 63 – NC2 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	378.41ms	117.57ms	1920.00ms	89.58%
Requisições/Segundo	133.87	37.00	260.00	76.15%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 64 – Total de transferência do teste - Aplicação Microserviço

2596 requisições em 10.05s	81.36MB read
Requisições/Segundo	258.39
Transferência/Segundo	8.10MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 65 – NC2 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	381.48ms	74.49ms	154.84ms	78.55%
Requisições/Segundo	65.29	26.13	151.00	73.83%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 66 – Total de transferência do teste - Aplicação Microserviço

2545 requisições em 10.10s	7.03MB read
Requisições/Segundo	44.37
Transferência/Segundo	7.91MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 67 – NC2 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	369.24ms	70.08ms	691.96ms	74.37%
Requisições/Segundo	32.78	16.77	101.00	64.91%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 68 – Total de transferência do teste - Aplicação Microserviço

2528 requisições em 10.09s	79.19MB read
Requisições/Segundo	250.45
Transferência/Segundo	7.85MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 69 – NC2 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	441.52ms	248.33ms	1.81s	87.66%
Requisições/Segundo	15.91	9.58	50.00	70.79%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 70 – Total de transferência do teste - Aplicação Microserviço

2188 requisições em 10.10s	34.58MB read
Requisições/Segundo	216.59
Transferência/Segundo	3.42MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 71 – NC2 - 32 threadas

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	446.69ms	205.39ms	1.87ms	80.82%
Requisições/Segundo	8.13	4.84	20.00	74.41%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 72 – Total de transferência do teste - Aplicação Microserviço

2135 requisições em 10.07s	33.63MB read
Requisições/Segundo	212.06
Transferência/Segundo	3.34MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 73 – NC2 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	361.91ms	231.69ms	1.99s	92.77%
Requisições/Segundo	3.02ms	1.70	10.00	77.22%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 74 – Total de transferência do teste - Aplicação Microserviço

1809 requisições em 10.09s	28.40MB
Requisições/Segundo	179.24
Transferência/Segundo	2.82MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC3 Microserviço

Tabela 75 – NC3 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	802.99ms	477.41ms	2000.00ms	67.45%
Requisições/Segundo	217.69	186.65	1001.00	81.96%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 76 – Total de transferência do teste - Aplicação Microserviço

4505 requisições em 10.10s	75.20MB read
Requisições/Segundo	445.88
Transferência/Segundo	7.44MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 77 – NC3 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	803.25ms	401.71ms	1990.00ms	73.03%
Requisições/Segundo	102.91	83.99	484.00	86.56%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 78 – Total de transferência do teste - Aplicação Microserviço

4132 requisições em 10.05s	81.35MB
Requisições/Segundo	411.34
Transferência/Segundo	8.10MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 79 – NC3 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	828.98ms	414.90ms	1990.00ms	69.26%
Requisições/Segundo	50.01	45.83	310.00	87.20%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 80 – Total de transferência do teste - Aplicação Microserviço

3956 requisições em 10.08s	75.70MB
Requisições/Segundo	392.58
Transferência/Segundo	7.51MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 81 – NC3 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	807.44ms	377.32ms	2000.00ms	74.34%
Requisições/Segundo	30.60	30.10	181.00	86.67%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 82 – Total de transferência do teste - Aplicação Microserviço

4303 requisições em 10.07s	65.66MB
Requisições/Segundo	427.14
Transferência/Segundo	6.52MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 83 – NC3 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	810.03ms	414.30ms	1990.00	75.24%
Requisições/Segundo	19.25	14.67	176.12	87.43%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 84 – Total de transferência do teste - Aplicação Microserviço

4112 requisições em 10.07s	72.03MB read
Requisições/Segundo	265.33
Transferência/Segundo	7.81MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 85 – NC3 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	838.24ms	466.92ms	2000.00ms	69.01%
Requisições/Segundo	9.30	8.51	70.00	83.02%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 86 – Total de transferência do teste - Aplicação Microserviço

3985 requisições em 10.10s	77.13MB read
Requisições/Segundo	394.59
Transferência/Segundo	7.64MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 87 – NC3 - 128 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	903.55ms	554.70ms	2000.00ms	55.97%
Requisições/Segundo	6.31	6.60	60.00	89.49%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 88 – Total de transferência do teste - Aplicação Microserviço

3913 requisições em 10.09s	40.38MB
Requisições/Segundo	387.63
Transferência/Segundo	4.00MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 89 – NC3 - 256 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1007.00ms	460.34ms	2000.00ms	66.73%
Requisições/Segundo	2.14	3.16	20.00	88.85%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 90 – Total de transferência do teste - Aplicação Microserviço

1975 requisições em 10.11s	33.54MB read
Requisições/Segundo	195.42
Transferência/Segundo	3.32MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 91 – NC3 - 512 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	1003.00ms	508.84ms	2000.00ms	66.24%
Requisições/Segundo	0.50	0.50	1.00	50.48%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 92 – Total de transferência do teste - Aplicação Microserviço

2605 requisições em 10.10s	7.03MB read
Requisições/Segundo	44.37
Transferência/Segundo	4.20MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC4 Microserviçø

Tabela 93 – NC4 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	176.95ms	64.30ms	586.11ms	86.03%
Requisições/Segundo	243.43	216.19	1250.00	83.63%%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 94 – Total de transferência do teste - Aplicação Microserviço

4513 requisições em 10.05s	63.53MB read
Requisições/Segundo	449.00
Transferência/Segundo	6.32MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 95 – NC4 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	831.46ms	531.85ms	2000.00ms	58.00%
Requisições/Segundo	132.90	120.41	670.00	85.37%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 96 – Total de transferência do teste - Aplicação Microserviço

4668 requisições em 10.04s	61.01MB read
Requisições/Segundo	464.84
Transferência/Segundo	6.08MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 97 – NC4 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	773.21ms	483.88ms	2000.00ms	66.17%
Requisições/Segundo	49.61	56.43	525.00	89.31%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 98 – Total de transferência do teste - Aplicação Microserviço

3454 requisições em 10.10s	62.34MB read
Requisições/Segundo	341.94
Transferência/Segundo	6.17MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 99 – NC4 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	717.50ms	445.54ms	2000.00ms	66.25%
Requisições/Segundo	6.13	2.88	10.00	69.64%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 100 – Total de transferência do teste - Aplicação Microserviço

3664 requisições em 10.10s	59.07MB read
Requisições/Segundo	362.94
Transferência/Segundo	5.85MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 101 – NC4 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	802.90ms	523.56ms	2000.00ms	62.52%
Requisições/Segundo	21.41	33.20	530.00	92.95%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 102 – Total de transferência do teste - Aplicação Microserviço

4563 requisições em 10.09s	65.61MB
Requisições/Segundo	452.07
Transferência/Segundo	6.50MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 103 – NC4 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	863.13ms	498.00ms	2000.00ms	65.27%
Requisições/Segundo	12.30	13.66	150.00	88.87%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 104 – Total de transferência do teste - Aplicação Microserviço

4416 requisições em 10.10s	58.34MB
Requisições/Segundo	437.23
Transferência/Segundo	5.78MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 105 – NC4 - 128 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	822.78ms	532.16ms	2000.00ms	63.71%
Requisições/Segundo	8.44	9.56	90.00	84.65%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 106 – Total de transferência do teste - Aplicação Microserviço

3890 requisições em 10.10s	52.19MB read
Requisições/Segundo	385.10
Transferência/Segundo	5.17MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 107 – NC4 - 256 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	847.28ms	509.81	2000.00ms	63.32%
Requisições/Segundo	5.64	9.41	131.00	92.33%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 108 – Total de transferência do teste - Aplicação Microserviço

4187 requisições em 10.10s	65.45MB read
Requisições/Segundo	414.56
Transferência/Segundo	6.48MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 109 – NC4 - 512 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	915.37ms	501.81ms	2000.00ms	60.90%
Requisições/Segundo	5.85	7.53	67.00	89.63%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 110 – Total de transferência do teste - Aplicação Microserviço

5237 requisições em 10.15s	88.49MB read
Requisições/Segundo	516.18
Transferência/Segundo	8.72MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC5 Microserviço

Tabela 111 – NC5 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	792.33ms	435.35ms	2000.00ms	68.03%
Requisições/Segundo	232.01	182.26	950.00	70.86%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 112 – Total de transferência do teste - Aplicação Microserviço

4405 requisições em 10.03s	61.94MB read
Requisições/Segundo	438.99
Transferência/Segundo	6.17MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 113 – NC5 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	885.19ms	543.75ms	2000.00ms	71.33%
Requisições/Segundo	91.17	79.98	362.00	75.78%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 114 – Total de transferência do teste - Aplicação Microserviço

2777 requisições em 10.10s	21.64MB
Requisições/Segundo	274.85
Transferência/Segundo	2.14MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 115 – NC5 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	826.70ms	492.85ms	2000.00ms	63.94%
Requisições/Segundo	56.82	45.38	330.00	72.29%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 116 – Total de transferência do teste - Aplicação Microserviço

3176 requisições em 10.11s	27.59MB
Requisições/Segundo	313.09
Transferência/Segundo	2.73MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 117 – NC5 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	940.48ms	497.27ms	2000.00ms	61.54%
Requisições/Segundo	30.71	26.70	268.00	81.01%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 118 – Total de transferência do teste - Aplicação Microserviço

3128 requisições em 10.13s	25.84MB read
Requisições/Segundo	308.77
Transferência/Segundo	2.55MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 119 – NC5 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	847.71ms	508.65ms	2000.00ms	64.92%
Requisições/Segundo	18.29	19.45	143.00	87.88%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 120 – Total de transferência do teste - Aplicação Microserviço

3285 requisições em 10.16s	26.26MB
Requisições/Segundo	323.37
Transferência/Segundo	2.58MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC6 Microserviçø

Tabela 121 – NC6 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	741.86ms	416.16ms	2000.00ms	78.61%
Requisições/Segundo	111.17	129.34	426.00	80.77%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 122 – Total de transferência do teste - Aplicação Microserviço

909 requisições em 10.39s	4.01MB
Requisições/Segundo	87.45
Transferência/Segundo	395.10KB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Abaixo estão descritos detalhadamente todas as rodadas de teste da aplicação MI-GO, o microserviço feito em Golang; apenas as rodadas NC1 e NC2 foram feitas.

- Experimento NC1 Microserviço Golang

Tabela 123 – NC1 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	700.83ms	400.97ms	620.12ms	82.31%
Requisições/Segundo	677.97	109.43	890.00	67.50%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 124 – Total de transferência do teste - Aplicação Microserviço

13521 requisições em 10.02s	369.00MB read
Requisições/Segundo	1348.77
Transferência/Segundo	36.81MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 125 – NC1 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	500.26ms	300.04ms	470.87ms	82.37%
Requisições/Segundo	392.40	58.40	535.00	65.00%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 126 – Total de transferência do teste - Aplicação Microserviço

15657 requisições em 10.02s	427.34MB read
Requisições/Segundo	1562.04
Transferência/Segundo	42.63MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 127 – NC1 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	500.75ms	300.95ms	610.80ms	87.57%
Requisições/Segundo	183.21	34.07	260.00	68.12%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 128 – Total de transferência do teste - Aplicação Microserviço

14638 requisições em 10.03s	399.48MB read
Requisições/Segundo	1458.76
Transferência/Segundo	39.81MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

- Experimento NC2 Microserviço Golang

Tabela 129 – NC2 - 2 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	610.28ms	260.14ms	1200.10ms	72.70%
Requisições/Segundo	822.33	133.64	1130.00	64.00%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 130 – Total de transferência do teste - Aplicação Microserviço

16401 requisições em 10.04s	447.59MB read
Requisições/Segundo	1633.27
Transferência/Segundo	44.57MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 131 – NC2 - 4 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	570.01ms	230.09ms	1200.39ms	72.42%
Requisições/Segundo	441.20	60.01	570.00	66.50%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 132 – Total de transferência do teste - Aplicação Microserviço

17618 requisições em 10.05s	480.91MB read
Requisições/Segundo	1753.77
Transferência/Segundo	48.04MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 133 – NC2 - 8 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	510.24	190.10ms	1056.88ms	70.00%
Requisições/Segundo	234.98	26.89	343.00	72.88%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 134 – Total de transferência do teste - Aplicação Microserviço

18784requisições em 10.10s	512.63MB read
Requisições/Segundo	1868.47
Transferência/Segundo	50.99MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 135 – NC2 - 16 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	550.61	220.92ms	1219.28ms	71.84%
Requisições/Segundo	108.50	21.13	180.00	66.31%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 136 – Total de transferência do teste - Aplicação Microserviço

17412 requisições em 10.08s	475.21MB
Requisições/Segundo	1727.00
Transferência/Segundo	47.13MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 137 – NC2 - 32 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	580.97ms	260.06ms	1224.76ms	72.47%
Requisições/Segundo	51.12	13.47	110.00	75.09%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 138 – Total de transferência do teste - Aplicação Microserviço

16493 requisições em 10.10s	450.11MB read
Requisições/Segundo	1632.73
Transferência/Segundo	44.56MB

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 139 – NC2 - 64 threads

Thread Stats	AVG	STDEV	MAX	+/- STDEV
Latência	360.40ms	140.43ms	1100.90ms	70.66%
Requisições/Segundo	27.27	8.10	101.00	62.83%

Tabela gerada pelo próprio autor - WRK benchmarking tool

Tabela 140 – Total de transferência do teste - Aplicação Microserviço

17644 requisições em 10.10s	481.53MB read
Requisições/Segundo	1747.10
Transferência/Segundo	48.15MB

Tabela gerada pelo próprio autor - WRK benchmarking tool