FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”

CENTRO UNIVERSITÁRIO DE MARÍLIA – UNIVEM

SISTEMAS DE INFORMAÇÃO

ANDERSON ROGÉRIO MEIRA PEREIRA

SISTEMA COLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE

MARÍLIA

2015

ANDERSON ROGÉRIO MEIRA PEREIRA

SISTEMA COLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE

Trabalho de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador
Prof.: Me. Ricardo José Sabatine

MARÍLIA
2015

PEREIRA, Anderson Rogério Meira

SISTEMA COLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE / Anderson Rogério Meira Pereira; orientador: Prof. Me. José Ricardo Sabatine. Marília, SP: [s.n.], 2015.

96 folhas

Monografia (Bacharelado em Sistemas de Informação): Centro Universitário Eurípides de Marília.

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
MANTIDO PELA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Anderson Rogério Meira Pereira

TÍTULO: SISTEMA COLABOLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE.

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Sistemas de Informação.

Nota: 10 (Dez)

Orientador:Ricardo José Sabatine Ricardo Sabatine

1º.Examinador:Renata Aparecida de Carvalho Paschoal Renata Paschoal.

2º.Examinador:Paulo Rogerio de Mello Cardoso Paulo Rogerio Mello Cardoso

Marília, 01 de dezembro de 2015.

Dedico este trabalho aos meus pais Valdomiro e Inês, minha irmã Fernanda, minha esposa Tatiane e meu filho Felipe, pessoas que sempre acreditaram em mim, mesmo quando nem mesmo eu acreditei.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, que não mediram esforços para criar seus filhos e que me ensinaram a enxergar que no mundo a maior riqueza são as pessoas. Pai, Mãe, sou eternamente grato a tudo que vocês fizeram e ainda fazem por mim.

Agradeço muito a minha esposa Tatiane, que suportou com paciência momentos difíceis nestes quatros anos de graduação, sendo testemunha ocular de todo o esforço desprendido para que este trabalho fosse concluído com êxito. Tati, conseguimos!

Agradeço meu filho Felipe, meu companheiro de sempre e minha inspiração, por me esperar muitas vezes chegar da aula para conversar sobre o seu dia, os seu jogos, seus problemas e seus aprendizados. Chip, papai te ama!

Agradeço minha irmã Fernanda e meu cunhado Ricardo pelo suporte e incentivo sempre presentes. Fer, Ri, deu certo!

Agradeço meu grande amigo Fernando Giovanini, que não só contribuiu para eu pudesse trabalhar com tecnologia, mas também por acreditar no meu potencial.

Agradeço meu antigo colega de trabalho e parceiro Leopoldo Vettor, que me lançou o desafio de buscar a graduação, de não me acomodar e de mostrar meu valor.

Agradeço meus amigos Rafael Moraes, Fernando Mioto e Wellington Sono da Vertical3W, pela paciência diária, pelas discussões, debates e principalmente por compartilharem suas experiências e conhecimento técnico comigo.

Agradeço aos amigos que conquistei na turma de Sistemas de Informação, que compartilharam toda a jornada desses quatro anos de curso. Aos que chegaram até aqui: Vencemos molecada!

Agradeço ao Gabriel Emídio, que além de ter se tornado um grande amigo, tem sido um referencial de caráter e profissionalismo para mim. Obrigado Gazão!

Meus agradecimentos também à toda equipe da Persys, que sempre esteve de portas abertas para contribuir com meu crescimento na graduação e profissional. João, funcionou!

Agradeço também a professora Giulianna Marega por ouvir e me aconselhar sobre minha vida como universitário, minhas dúvidas sobre o mercado e pela amizade. Giu, valeu!

E por último, mas não menos importante, agradeço ao meu orientador Ricardo Sabatine, que não permitiu que eu simplesmente desistisse de produzir este trabalho e me guiou em todos os momentos para que eu atingisse êxito na entrega. Sabatine, muito obrigado!

Agradeço também a você que não está na lista acima, mas que faz parte da minha vida e que me ensina a ser uma pessoa melhor. Muito obrigado!

“Um passo a frente
E você não está mais no mesmo lugar”

Chico Science

PEREIRA, Anderson Rogério Meira. **SISTEMA COLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE**. 2015. 95 f. Trabalho de curso. (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2015.

RESUMO

Desenhar sistemas para Web que sejam escaláveis e performáticos não é somente uma tarefa difícil, mas também um grande desafio devido às altas demandas de disponibilização de dados, produtos e serviços através da internet, considerando principalmente o cenário de grandes empresas como Amazon, Facebook, Google, Twitter e Yahoo!, que possuem arquiteturas capazes de atender volumes que vão desde centenas de milhões até bilhões de usuários todos os dias. Devido à dificuldade de se escalar sistemas criados em arquiteturas mais clássicas que seguem o modelo orientado por *threads*, muitos desenvolvedores têm utilizado a arquitetura orientada a eventos em suas aplicações com o objetivo de obter alto índice de desempenho a custos menores. Assim, este trabalho visa apresentar os principais conceitos que envolvem a arquitetura de sistemas clássicos, a apresentação da arquitetura orientada a eventos, suas tecnologias, desafios, vantagens e desvantagens em relação à clássica e a validação de desempenho de um protótipo de aplicação colaborativa para marcação de pontos de acessibilidade.

Palavras-Chave: Arquiteturas Escaláveis, Node.js, API, Acessibilidade

PEREIRA, Anderson Rogério Meira. **SISTEMA COLABORATIVO ESCALÁVEL E ORIENTADO A EVENTOS PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE**. 2015. 95 f. Trabalho de curso. (Bacharelado em Sistemas de Informação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2015.

ABSTRACT

Designing systems to Web that are scalable and performers is not only a difficult task, but also a great challenge due to the high demands of availability of data, products and services over the internet, especially considering the scenario of large companies such as Amazon, Facebook, Google , Twitter and Yahoo !, which have architectures able to meet volumes ranging from hundreds of millions to billions of users every day. Because of the difficulty of climbing systems established in more traditional architectures that follow the model driven threads, many developers have used oriented architecture events in their applications in order to get high performance index at lower costs. This work aims to present the main concepts around the architecture of classical systems, the presentation of oriented architecture events, their technologies, challenges, advantages and disadvantages in relation to classical and performance validation of a collaborative application prototype for marking accessibility points.

Keywords: Scalable architectures, Node.js, API, Accessibility

LISTA DE FIGURAS

Sumário

INTRODUÇÃO

A arquitetura de software tem por objetivo tratar da forma com que são organizados logicamente e como interagem os componentes de software que constituem o sistema, por isso é crucial adotar uma arquitetura para o sucesso no desenvolvimento de grandes sistemas (Tanembaum & Steen, 2007).

Por isso, como afirmam Tanembaum & Steen (2007) a noção de um estilo arquitetônico é importante também para o desenho dos componentes de software, pois indicam como estes componentes serão conectados uns aos outros, quais dados serão trocados entre eles e principalmente como esses elementos são configurados para formar um sistema.

É importante também que estes componentes de software possam ser substituíveis, desde que respeitadas suas respectivas interfaces, utilizando do conceito de conector, que em geral é descrito como um mecanismo que serve de mediador na comunicação e cooperação entre componentes.

Desenhar sistemas para Web que sejam escaláveis e performáticos não é somente uma tarefa difícil (Silveira, et al., 2012), mas também um grande desafio devido às altas demandas de disponibilização de dados, produtos e serviços através da internet, considerando principalmente o cenário de grandes empresas como Amazon, Facebook, Google, Twitter e Yahoo!, que possuem arquiteturas capazes de atender volumes que vão desde centenas de milhões até bilhões de usuários todos os dias (Arango & Kaponig, 2009).

Estudos como o de Kegel (Liu & Deters, 2009) apontam um problema, descrito como "C10K problem", que menciona o limite conexões simultâneas suportadas pela maioria dos servidores web é de 10.000 (dez mil). Além disso, menciona que para suportar esta quantidade de conexões neste cenário, o hardware não é o único gargalo, mas que tal suporte poderia se dar através da mudança no modo em que as operações de I/O são realizadas.

Em arquiteturas clássicas, o modelo de threads é o mais utilizado para a construção de software, quando o objetivo é alcançar concorrência de I/O e outros recursos, concorrência esta que é geralmente chamada de programação serial (Dabek, Zeldvich, Kaashoek, Mazière, & Morris, 2002). Outro fator importante no uso de threads é a capacidade de utilização de multiprocessadores ou cores (Zeldovich, et al., 2003).

Welsh et al. (2001) afirmam que embora seja relativamente fácil desenvolver aplicativos baseados em threads, geralmente devido a algum recurso da própria linguagem de programação utilizada, a carga de recursos de processamento utilizadas pelas *threads* pode levar a uma degradação de desempenho à medida que o número de threads aumenta. Isso acontece

devido ao fato de que neste modelo, a cada requisição solicitada ao aplicativo é gerada uma nova *thread* separada, então quanto maior o número de conexões simultâneas, maior também será o número de threads abertas.

Levando em consideração que arquiteturas orientadas por *threads* se tornam difíceis de escalar a medida que o número de usuários cresce, vários desenvolvedores têm apostado em novas arquiteturas, e uma delas é a arquitetura orientada a eventos.

Objetivo

O objetivo geral deste trabalho foi criar um protótipo de sistema colaborativo que permitiria a uma grande massa de usuários marcar, avaliar e classificar pontos de acessibilidade através de uma interface de aplicação web. Os dados gerados pelos usuários têm como propósito auxiliar, ONGs, prefeituras e outros órgão públicos e privados no apoio à decisão de implementação de novos pontos de acessibilidade.

Para atender tal demanda o protótipo teve como requisito suportar um alto índice de requisições concorrentes simulando sua utilização em produção. Sendo assim, a arquitetura foi construída utilizando tecnologias que possuem características do paradigma orientado a eventos, visando o suporte da alta concorrência.

Como objetivo especifico, a avaliação do desempenho da arquitetura foi realizada através de uma série de testes, parâmetros e configurações com o intuito de gerar informações a respeito do seu comportamento, para que arquitetos de sistema, caso o cenário seja propício, levar em consideração este tipo de arquitetura, tanto em relação às suas vantagens quanto às suas desvantagens.

Organização do trabalho

A organização do trabalho se divide em quatro capítulos da seguinte forma: No Capítulo 1 é feita a revisão bibliográfica que trata a respeito de modelos arquiteturais clássicos de sistemas, seus tipos, características e cenários. Ainda no Capítulo 1 o trabalho aborda o tema das arquiteturas consideradas escaláveis, seus tipos, comunicação entre componentes, comunicação síncrona e assíncrona e estratégias para obtenção de desempenho.

No Capítulo 2 trata das tecnologias que utilizam o paradigma orientado a eventos onde são apresentados os conceitos que envolvem a arquitetura, tecnologias que utilizam este

paradigma, frameworks para desenvolvimento de aplicações orientadas a eventos e estudos de caso onde foram utilizadas tais tecnologia na construção de aplicações.

No Capítulo 3 é feita a descrição do protótipo criado, bem como o cenário de aplicação, suas principais características, as tecnologias utilizadas na construção, descrição das funcionalidades e os detalhes da implementação das funcionalidades.

No Capítulo 4 são apresentados os resultados de desempenho baseado nos testes e cenários nos quais o protótipo foi submetido, bem como sua contribuição.

E finalmente no Capítulo 5 foi feita a conclusão do trabalho com os pontos levantados pelo autor em relação à produção de todo o trabalho e a descrição de trabalhos futuros.

1. ARQUITETURAS DE SISTEMAS ESCALÁVEIS

Neste capítulo são abordados os conceitos que envolvem arquiteturas de sistemas escaláveis, seus principais estilos arquitetônicos, quando e como uma arquitetura é considerada escalável, os tipos de comunicação entre componentes e por último, a comparação entre *single thread* e *multithread* no cenário de aplicações web.

1.1. Definição de Arquitetura

Um fator importante na construção de software é a sua arquitetura, pois de acordo com Sommerville (2011, p. 103), o projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado e com a estrutura geral desse sistema.

Muitas pessoas tentam definir o termo "arquitetura", mas há pouca concordância entre elas. Mesmo com essa divergência quanto à definição, é possível identificar dois elementos comuns: um é a decomposição em alto nível de um sistema e suas partes e o outro é que são decisões difíceis de alterar (Fowler, Padrões de arquitetura de aplicações corporativas, 2006).

Pressman (2002, p. 358) utiliza a seguinte definição a respeito de arquitetura:

> A arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas de um sistema que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.

É importante apontar que muitas das decisões relacionadas à arquitetura dizem respeito ao desempenho. Desempenho este que pode ser afetado devido a decisões ruins de arquitetura, de modo a tornar difícil a manutenção com otimizações posteriores. Por outro lado, mesmo quando o sistema é de fácil manutenção, as pessoas envolvidas no projeto devem se preocupar desde cedo com as decisões sobre a arquitetura (Fowler, Padrões de arquitetura de aplicações corporativas, 2006).

A seguir serão apresentados tipos de arquiteturas comumente utilizadas na construção de sistemas.

1.1.1. Arquitetura Cliente Servidor

Devido a necessidade de processamento distribuído por parte dos sistemas que utilizam cada vez mais redes de computadores para tal fim, a arquitetura conhecida como Cliente Servidor, ou arquitetura cliente/servidor de duas camadas, é amplamente utilizada.

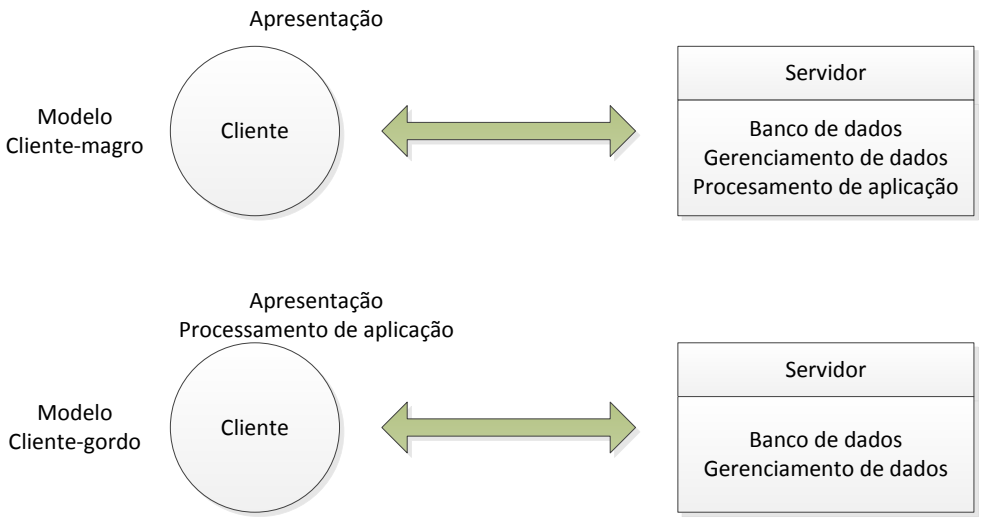
A arquitetura cliente servidor consiste na divisão do processamento entre 'clientes' e 'servidores' que fazem parte da mesma rede, onde cada um desempenha uma função específica e que está mais apto a executar (Laudon & Laudon, 2011).

Conforme contextualiza Laudon & Laudon (2011, pp. 107,108) nesta arquitetura o cliente é o ponto de entrada do usuário para função requisitada, interagindo através de inserção de dados ou requisição de dados para análise posterior. Já o servidor tem como função prover serviços ao cliente, armazenando e processando dados compartilhados, bem como executar funções como autenticação de usuário, acesso remoto, dentre outros.

Somerville (2011) apresenta duas formas deste modelo de arquitetura, o modelo cliente-magro (*thin-client*), em que a camada de apresentação é implementada no cliente e todas as outras camadas são implementadas no servidor e o modelo cliente-gordo (*fat-client*), em que todo o processamento ou parte do processamento da aplicação é executado no cliente e as funções de bancos de dados e gerenciamento são implementadas no servidor.

A Figura 1 apresenta a estrutura dos modelos de arquitetura cliente-magro e cliente-gordo.

Figura 1 - Arquitetura cliente-magro e cliente-gordo



Fonte: (Sommerville, 2011)

Vale lembrar que historicamente é a arquitetura de sistemas distribuídos mais importante e continua sendo amplamente utilizada (Coulouris, Dollimore, Kindberg, & Blair, 2013).

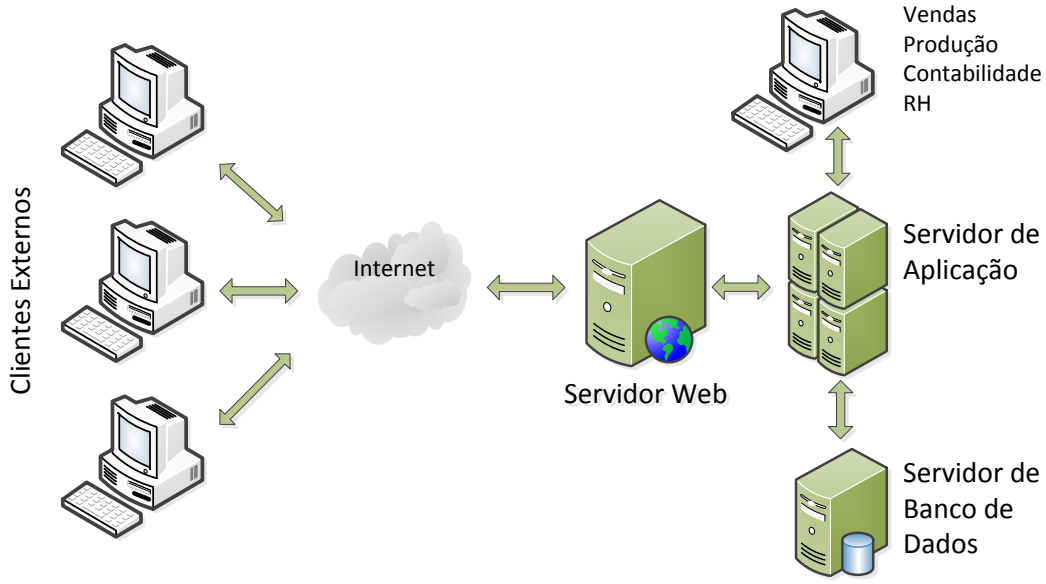
1.1.2. Arquitetura N-Camadas

Há outro cenário onde grandes corporações utilizam arquiteturas mais complexas, como as arquiteturas cliente/servidor multicamadas, conhecidas também como N-camadas, onde o trabalho de toda rede é dividido entre servidores de vários níveis, de acordo com o serviço requisitado.

Ao se estender uma arquitetura do modelo cliente-servidor para multicamadas, no qual servidores adicionais podem ser adicionados ao sistema, o processamento de aplicação é distribuído entre estes servidores, proporcionando maior escalabilidade (Sommerville, 2011).

Um exemplo da aplicação da arquitetura multicamadas, apresentado na Figura 2, seria no primeiro nível um servidor Web responsável pela apresentação de uma página Web a um cliente, em resposta a uma solicitação de serviço.

Figura 2 - Exemplo de Arquitetura N-Camadas

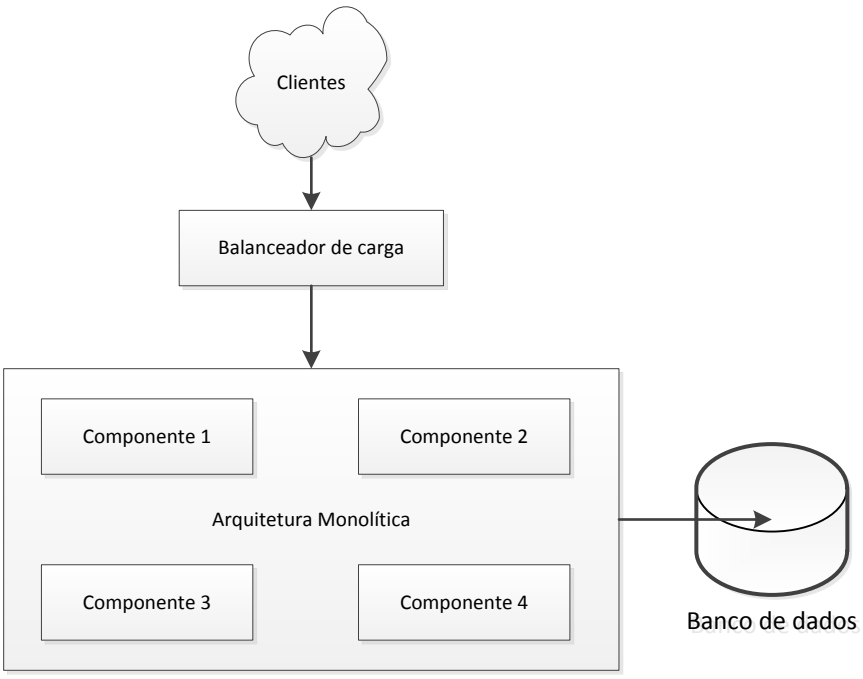


Fonte: O Autor

1.1.3. Micro Serviços

Tradicionalmente aplicações web são construídas utilizando a arquitetura monolítica onde quase todos os componentes do sistema, funcionam em um único processo. Em situações simples essa arquitetura tem vantagens pois tarefas de implementação e dimensionamento dos sistemas atrelados a um balanceador de carga são relativamente simples. Na Figura 3 é mostrado um exemplo de arquitetura monolítica e seus componentes.

Figura 3 – Exemplo de Arquitetura Monolítica



Fonte: O Autor

No entanto, conforme enfatiza Stubbs, Moreira & Dooley (2015) conforme as exigências e demandas vão aumentando, a complexidade de manutenção aumenta também, como por exemplo a alteração de um componente que pode afetar outras áreas da aplicação, tornando ineficiente a evolução do sistema.

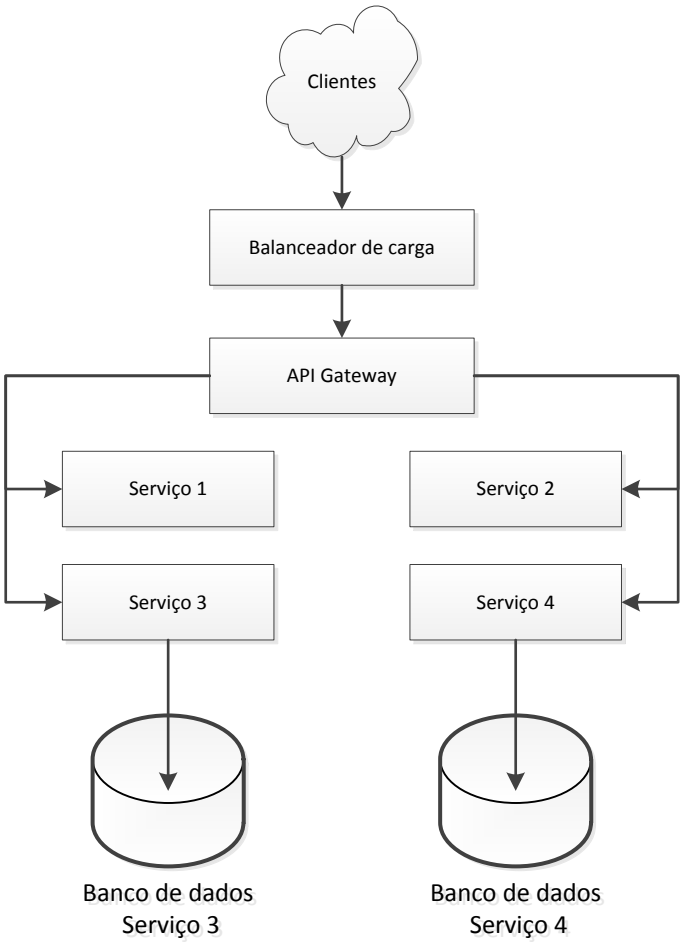
Segundo Savchenko, Radchenko & Taipale (2015), a arquitetura de micro serviços é um padrão de design para aplicativos em nuvem que implica que a aplicação seja dividida em uma série de pequenos serviços independentes, e que implementam uma determinada característica.

Newman (2015) define micro serviços como uma abordagem para sistemas distribuídos que promove serviços de baixa granulação, com seu próprio ciclo de vida e que são modelados

principalmente em torno de um domínio de negócio.

Os micros serviços podem ser considerados como meta processos em um meta-sistema operacional – eles são independentes, podem se comunicar uns com os outros utilizando mensagens, podem ser duplicados, suspensos ou movidos para qualquer recurso computacional e assim por diante conforme mostra a Figura 4.

Figura 4 - Arquitetura de micro serviços



Fonte: O autor

De acordo com Newman (2015) os principais benefícios da utilização de uma arquitetura construída com micro serviços são:

- *Tecnologia heterogênea:* Em um sistema composto por vários serviços colaborativos, é possível utilizar diferentes tecnologias, uma ao lado da outra, permitindo que seja utilizada a ferramenta, linguagem de programação e tecnologia correta para o cenário mais adequado.

- *Resiliencia:* Diferente da arquitetura monolítica, em micro serviços é possível isolar um componente do sistema em caso de falha, sem comprometer os outros serviços do sistema.
- *Escalabilidade:* Com serviços menores, é possível escalar apenas os serviços em que há necessidade, permitindo executar outras partes do sistema em um hardware menor e menos potente.
- *Facilidade de implantação:* Possibilidade de realizar uma mudança em um único serviço e implantá-lo de forma independente do resto do sistema, tornando a disponibilização de novas funcionalidades mais rapidamente.
- *Alinhamento organizacional:* Micro serviços permitem que as equipes de desenvolvimento sejam menores, produzindo bases de código menores, permitindo maior produtividade se comparado a equipes grandes com grandes bases de código.
- *Reutilização:* Seguindo o propósito de arquiteturas orientadas a serviços, como por exemplo a reutilização de funcionalidades, a arquitetura de micro serviços permite que um determinado sistema atenda a vários tipos de aplicações, seja ela web, móvel, ou desktop, atendendo às demandas de mercado.
- *Substituição otimizada:* Com serviços menores e individuais, o custo de substituição por uma melhor implementação ou até mesmo exclusão por completo é muito mais fácil pois a gestão também é mais fácil. Equipes que utilizam essa abordagem se sentem confortáveis para reescrever completamente os serviços quando necessário, ou desativar o serviço quando ele não é mais necessário.

Vários exemplos de sistemas de que possuem altas demandas como Netflix, SoundCloud e serviços da Amazon como o S3 utilizam o conceito de micro serviços em suas arquiteturas (Savchenko, Radchenko, & Taipale, 2015). No próximo ponto serão tratados conceitos que dizem respeito à escalabilidade das arquiteturas de software.

1.2. O que é uma arquitetura escalável

Segundo Coulouris, Dollimore, Kindberg, & Gordon (2013, pp. 19,20) um sistema construído sobre uma arquitetura escalável permanece eficiente quando há um aumento significativo no número de recursos e de usuários.

O termo *escalabilidade* é diferente do termo *performance* quando se trata de um sistema de software como explica Liu (2009):

- A performance mede o quão rápido e eficiente um sistema pode executar determinadas tarefas computacionais, enquanto escalabilidade mede a tendência de desempenho com o aumento da carga. Há dois tipos de tarefas de computação que são medidos através de diferentes métricas de performance. Para processamento de transações online, que é um tipo de tarefa computacional que envolve interatividade com o usuário, a métrica de tempo de resposta é usada para medir o quão rápido um sistema pode responder às solicitações dos usuários, enquanto que para processamento em lote, onde não há interação com usuários, a métrica de rendimento (*throughput*) é utilizada para medir o número de transações que um sistema pode completar durante um período de tempo. Perfomance e escalabilidade são inseparáveis um do outro. Não faz sentido falar de escalabilidade se um sistema não tem performance, no entanto, um software pode ter perfomance e não escalar.
- Para um determinado ambiente que possui um hardware de tamanho adequado, sistema operacional configurado de forma correta e *middleware* dependentes, se a perfornance de um sistema deteriora-se rapidamente com o aumento da carga (número de usuários ou volume de transações), antes de atingir o nível de carga previsto, não é escalável e eventualmente não terá boa performace.
- Se a performance de um sistema torna-se inaceitável quando atinge um determinado nível de carga em um determinado ambiente, mas não pode ser melhorado e / ou atualizado com hardware adicional, então diz-se que o sistema não é escalável.

O número de computadores e serviços tem aumentado substancialmente, principalmente na Web conforme contextualiza Laudon & Laudon (2011, p. 130):

> O E-commerce e o E-business estão impingindo novas e intensas demandas sobre a tecnologia de hardware. São necessários recursos de processamento e armazenamento muito maiores para processar e armazenar as transações digitais emergentes que fluem entre diferentes partes da empresa, e entre ela e seus clientes e fornecedores. A utilização simultânea de um site por muitas pessoas acarreta uma pressão muito grande sobre um sistema de computação. O mesmo acontece com a hospedagem de grandes quantidades de páginas da Web intensivas em recursos gráficos ou vídeos.
>
> Atualmente, administradores e especialistas de sistemas de informação precisam dar mais atenção ao planejamento da capacidade e da escalabilidade do hardware do que no passado.

A Tabela 1 mostra este significativo aumento tanto no número de computadores quanto no número de serviços web, bem como a porcentagem relativa, que cresce rapidamente, tendência esta explicada pelo aumento da computação pessoal fixa e móvel (Coulouris, Dollimore, Kindberg, & Blair, 2013).

Tabela 1 - Crescimento da Internet (Computadores e servidores web)

Data	Computadores	Servidores Web	Percentual
Julho de 1993	1.776.000	130	0,008
Julho de 1995	6.642.000	23.500	0,4
Julho de 1997	19.540.000	1.203.096	6
Julho de 1999	56.218.000	6.598.697	12
Julho de 2001	125.888.197	31.299.592	25
Julho de 2003	~200.000.000	42.298.371	21
Julho de 2005	353.284.187	67.571.581	19

Fonte: (Coulouris, Dollimore, Kindberg, & Blair, 2013)

Embora diferentes aplicações web possam não funcionar da mesma forma, há pontos comuns em uma aplicação deste tipo, onde os recursos se tornam limitados com a demanda crescente, como por exemplo o armazenamento de dados, tráfego da rede e o eventual aumento de servidores. Por isso, é essencial entender como são consumidos tais recursos pelo sistema a fim de manter o desempenho desejável.

Uma forma de entender como o sistema se comporta quando há o aumento na carga por clientes é investigar onde estes pontos estão e utilizar um planejamento de capacidade e medição sobre estes pontos, utilizando indicadores de pontos específicos para monitoramento e acompanhamento de desempenho, permitindo a expansão dinâmica da aplicação (Chieu, Mohindra, & Karve, 2011).

Em aplicações web, os indicadores mais comuns são:

- Número de usuários concorrentes.
- Número de conexões ativas.
- Número de requisições por segundo.
- Tempo médio de resposta por requisição.

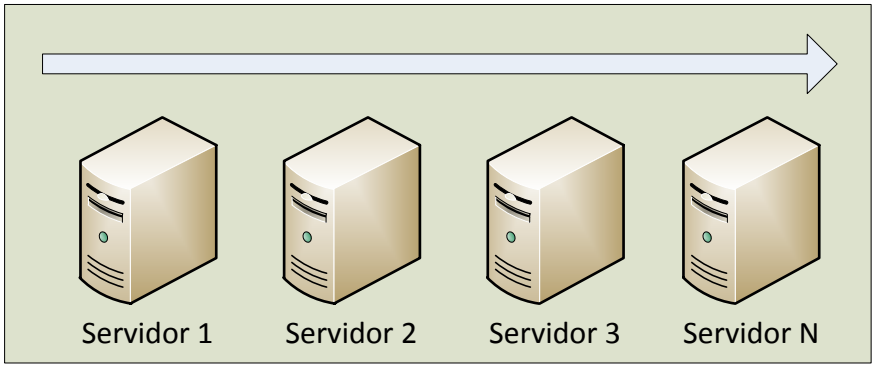
Sendo assim, os projetos baseados em arquiteturas escaláveis apresentam os seguintes desafios (Coulouris, Dollimore, Kindberg, & Blair, 2013):

- *Controlar o custo de recursos físicos*: à medida que a demanda por um recurso aumenta, o sistema deve ser ampliado para atender a esta demanda desde que o custo seja razoável;
- *Controlar a perda de desempenho*: Considerando cenários onde o gerenciamento de um conjunto de dados que possui o tamanho proporcional ao número de recursos ou usuários presentes no sistema, como por exemplo uma tabela de nomes de domínio e seus endereços ip mantidos por um servidor DNS. Neste exemplo são utilizados algoritmos que utilizam estruturas hierárquicas, conseguindo melhor índice de escalabilidade do que os que usam estruturas lineares, mas ainda assim com o aumento do consumo resulta em alguma perda de desempenho;
- *Impedir que os recursos de software se esgotem*: Como em várias situações é difícil prever com antecedência a demanda que será imposta ao sistema, o arquiteto deve sempre monitorar o funcionamento e o consumo de recursos;
- *Evitar gargalos de desempenho*: Verificando serviços e recursos compartilhados que são acessados com frequência.

1.2.1.1. Escalabilidade Horizontal

De acordo com a definição de Paudyal (2011) escalonamento horizontal é uma forma de escalar uma aplicação com a adição de mais hardware para se suportar determinada carga. Neste cenário é esperado que com a adição de máquinas adicionais o sistema possa suportar o crescimento linear de usuários simultâneos segundo mostra a Figura 5.

Figura 5 – Exemplo de solução de arquitetura para escalabilidade horizontal

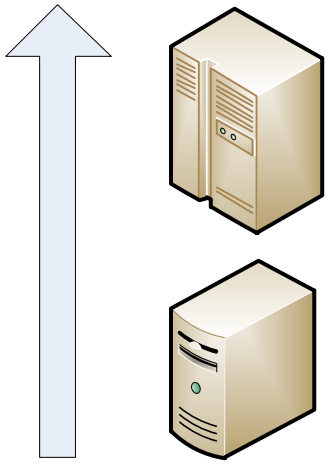


Fonte: O Autor

1.2.1.2. Escalabilidade Vertical

Escalabilidade vertical, segundo o modelo da Figura 6, significa escalar uma aplicação com a adição de mais hardware em um mesmo nó. Neste caso é feita a adição de um processador mais potente, mais memória e caso seja necessário, melhoria em outros hardwares que compõem o nó (Paudyal, 2011).

Figura 6 - Exemplo de Arquitetura de escalabilidade vertical



Fonte: O Autor

A memória vem se tornando cada vez mais barata, computadores ficando mais rápidos e a capacidade de discos rígidos cada vez maiores, mas há um limite de incremento deste hardware, e este ponto é o que limita a escalabilidade em algum momento.

1.3. Comunicação entre os Componentes

Um item importante de uma arquitetura de software que seja escalável é definir como os componentes, que são entidades de comunicação do sistema, do modelo devem interagir.

Quanto aos tipos de comunicação entre componentes podemos considerar os seguintes tipos de paradigma (Coulouris, Dollimore, Kindberg, & Blair, 2013):

- *Comunicação entre processos*, que se refere à comunicação de baixo nível entre processos dos sistemas distribuídos, incluindo primitivas de passagem de mensagens, a programação de soquetes devido acesso direto à API oferecida pelos protocolos Internet, e comunicação em grupo, conhecida como multicast.

- *Invocação remota*, que abrange técnicas baseadas em troca bilateral entre as entidades que se comunicam, resultando na chamada de procedimento ou método remoto.
- *Protocolos de requisição-resposta*, que são um padrão imposto em um serviço de mensagens para suportar a computação do tipo cliente-servidor e normalmente envolvem troca de mensagens por pares entre o cliente para o servidor ou vice-versa, com a primeira mensagem contendo a codificação da operação a ser executada no servidor e um vetor de bytes contendo argumentos associados, já a segunda mensagem contém os resultados da operação que são novamente codificados como um vetor de bytes. Apesar de primitivo, é um paradigma bastante utilizado devido ao seu desempenho, sendo esta a estratégia utilizada pelo protocolo HTTP.
- *Chamada de procedimento remoto*, ou RPC, onde os procedimentos nos processos podem ser chamados como se fossem procedimentos no espaço de endereçamento local, ocultando a codificação e decodificação de parâmetros e resultados, a passagem de mensagens e a preservação semântica que s é exigida em uma chamada de procedimento. Esta abordagem é bastante usada na arquitetura cliente-servidor por ser direta e elegante, com servidores fornecendo operações por meio de serviços e os clientes chamando estas operações diretamente, como se estivessem disponíveis de forma local, oferecendo transparência de acesso e localização.

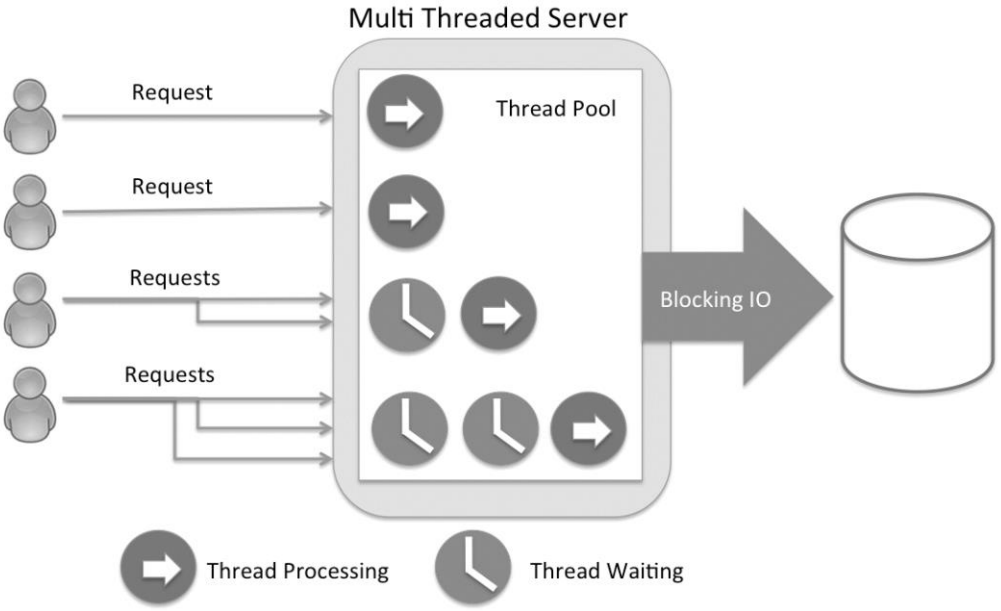
Por isso, conforme explicam Coulouris, Dollimore, Kindberg, & Blair (2013),este grupo de paradigmas de comunicação possuem algo em comum, que é a comunicação representada pela relação bilateral entre um remetente e um destinatário, com remetentes direcionando de forma explicita suas mensagens e invocações para os destinatários associados. Por sua vez, na maioria dos casos os destinatários conhecem a identidade dos remetentes, o que significa também que as duas entidades, remetente e destinatário, devem existir ao mesmo tempo.

Outro fator importante na comunicação entre os componentes do sistema diz respeito aos protocolos de *middleware*. Segundo Tanembaum & Steen (2007) um middleware é uma aplicação que geralmente reside logicamente na camada de aplicação, que contém muitos protocolos para uso geral e que suportam serviços de comunicação de alto nível.

1.3.1. Comunicação de I/O Síncrono Bloqueante

Segundo Teixeira (2013), na programação tradicional, o I/O é realizado da mesma forma que uma chamada de função local, onde o processamento é bloqueado até que a execução desta função termine. Este modelo bloqueante é utilizado desde os primeiros sistemas operacionais, onde há compartilhamento de tempo de processamento pelos processos, que corresponde a um usuário humano. O propósito, de acordo com o modelo proposto na Figura 7, era isolar os usuários entre si e esperar que eles terminem uma operação antes de decidir qual será a próxima operação.

Figura 7 - Funcionamento de um servidor *multithread*



Fonte: (Strongloop, 2015)

Neste modelo é gasto grande parte do tempo mantendo uma fila ociosa enquanto, é executado um I/O. Tarefas como por exemplo enviar e-mail, consultar o banco de dados ou fazer uma determinada leitura em disco gastam grande parte desse tempo, bloqueando o sistema inteiro enquanto não são finalizadas.

Teixeira (2013) aponta que para este modelo, a princípio, a alternativa é a utilização de *multi-threads*, onde um segmento é um tipo de processo leve que compartilha a memória com cada outro segmento dentro do mesmo processo. Threads são criadas como uma extensão ad hoc do modelo anterior para acomodar e executar várias *threads* simultaneamente. No modelo *multi-thread*, quando um segmento está aguardando uma operação de I/O, outro

segmento pode assumir a CPU. Quando a operação de I/O termina, este segmento acorda, o que significa que a thread que estava em execução é interrompida e, eventualmente é retomada mais tarde. Além disso, alguns sistemas permitem que estes segmentos sejam executados paralelamente em diferentes núcleos da CPU.

No cenário onde é utilizado *multithreading*, o desenvolvedor deve tomar bastante cuidado com o acesso simultâneo à memória compartilhada pois ele não sabe exatamente qual conjunto de threads está em execução em um determinado momento. Se o aplicativo depende fortemente de um estado compartilhado entre threads, este modelo de programação pode levar facilmente a erros estranhos e normalmente difíceis de se encontrar (Teixeira, 2013).

Outra alternativa apresentada por Teixeira (2013), seria utilizar o modelo de *multi-thread* cooperativa, onde o desenvolvedor é responsável por agendar algumas *threads* para execução. No entanto, esta abordagem pode se tornar complexa e sujeita a erros, pelas mesmas razões do *multithreading* padrão.

Ainda sobre o modelo bloqueante, Pereira (Pereira, 2013) enfatiza que com o aumento de acessos ao sistema, os gargalos de *performance* tendem a ser mais frequentes, aumentando a necessidade de se fazer um upgrade nos hardwares dos servidores e mesmo que eles sejam feitos o ideal seria buscar novas tecnologias que façam um bom uso do hardware existente, que utilizem o máximo do poder do processador atual, não o mantendo ocioso quando o mesmo realizar tarefas do tipo bloqueante.

Teixeira (2013) alerta também que com o uso amplo de redes de computadores e da internet, este modelo de “um usuário, um processo”, não escala bem, pois o gerenciamento de um grande número de processos feitos pelo sistema operacional e presentes neste tipo de aplicação tem um grande custo tanto de hardware quanto de software, fazendo com que o desempenho dessas tarefas começa a cair em decadência depois de que um certo número é alcançado.

1.3.2. Comunicação de I/O Assíncrono Não-Bloqueante

Para entendermos os conceitos aplicados em arquiteturas assíncronas não-bloqueantes é necessário saber como funciona o modelo de programação orientada a eventos.

Como vimos anteriormente em arquiteturas síncronas bloqueantes o estilo de programação é ilustrado pela Figura 8:

Figura 8 - Código de execução síncrona

```
result = query('SELECT * FROM posts WHERE id=1');
do_something_with(result);
```

Fonte: O autor

Neste exemplo de pesquisa a uma base de dados é necessário aguardar todo o processo de acesso ao banco de dados terminar para que a próxima função seja executada.

Em um sistema orientado a eventos tal trecho de código seria escrito da forma apresentada na Figura 9:

Figura 9 - Exemplo de execução assíncrona

```
query_finished = function(result) {
    do_something_with(result);
}

query('SELECT * FROM posts WHERE id=', query_finished);
```

Fonte: O autor

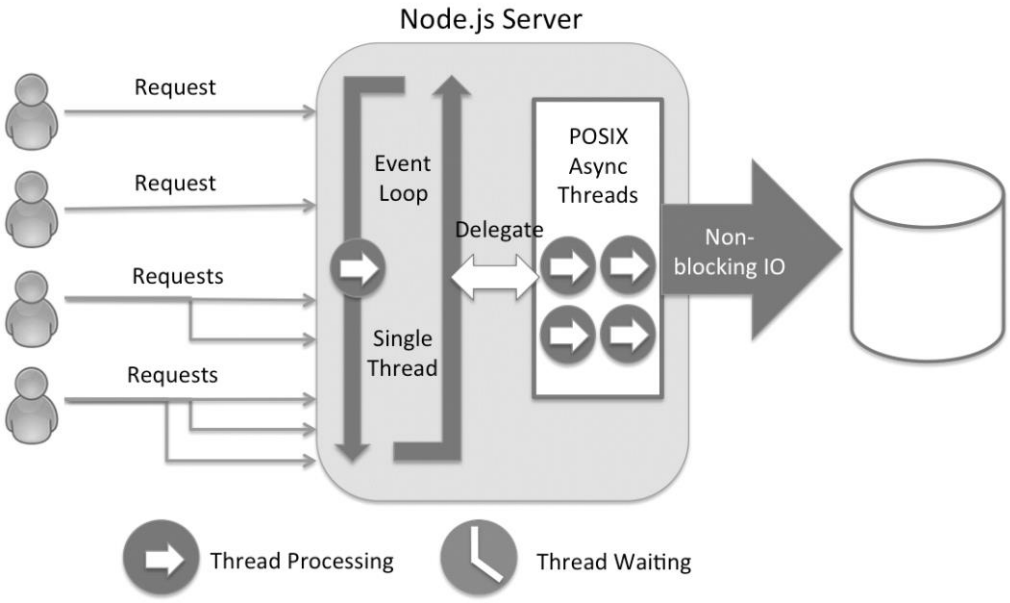
Neste segundo exemplo primeiro é definido que quando a consulta ao banco for concluída, armazenar o valor na função chamada “query_finished”. Depois essa função é passada como um argumento para a função “query”. Quando esta consulta for concluída, a função “query_finished” é invocada simplesmente retornando o resultado.

Conforme conceitua Teixeira (Teixeira, 2013), programação orientada a eventos é um estilo de programação onde o fluxo de execução é determinado por eventos. Os eventos são manipulados pelos manipuladores de eventos chamados de handlers ou retornos de chamada de envio, chamados de *callbacks*. Um *callback* é uma função que é invocada quando algo significante acontece, por exemplo quando o resultado de uma base de dados está disponível ou quando o usuário clica em um botão.

Este estilo de programação, onde ao invés de usar um valor de retorno o desenvolvedor define quais funções serão chamadas pelo sistema quando determinados eventos ocorrerem é chamado de programação orientada a eventos ou programação assíncrona. Neste estilo, conforme o exemplo apresentado na Figura 10, o processo atual não é bloqueado quando for realizado uma operação de I/O, na verdade eles ocorrerão de forma paralela e cada callback

será chamado quando a operação terminar (Teixeira, 2013).

Figura 10 - Exemplo de servidor assíncrono com Node.JS



Fonte: (Strongloop, 2015)

Em uma abordagem assíncrona a troca de mensagens é feita de tal maneira que o sistema que envia a mensagem não precisa esperar o processamento da resposta, ficando livre para continuar sua rotina, sendo uma invocação não bloqueante e é uma ótima opção para sistemas que necessitem executar rapidamente algum tipo de rotina ao ser disparado um evento (Silveira, et al., 2012).

1.4. Single thread vs Multithread

Thread é definida como uma abstração do sistema operacional de uma atividade em um ambiente de execução. Várias *threads* em um ambiente de execução formam um processo do sistema operacional. O sistema operacional cria ou destrói threads dinamicamente sempre que for necessário, para que se obtenha a maximização da concorrência entre os processos do sistema (Coulouris, Dollimore, Kindberg, & Blair, 2013). Um processo costuma ser definido como um programa execução.

Alguns trabalhos não fazem mais distinção entre arquiteturas multi-processadas e

multi-thread (Cordeiro, 2006), pois consideram que threads estão disponíveis em todos os sistemas operacionais modernos, diferenciando apenas entre processos leves no caso das threads e processos convencionais presentes nos sistemas operacionais antigos.

Segundo Cordeiro (2006) há argumentos que defendem threads devem ser utilizadas apenas quando paralelismo real é de fato necessário.

A utilização de threads como única forma para se obter escalabilidade é muito difícil pois são encontrados os seguintes problemas (Cordeiro, 2006, p. 6):

- O programador é obrigado a coordenar o acesso a regiões de memória compartilhada; tarefa complicada mesmo para programadores experientes;

- Torna a depuração mais complicada, pois threads introduzem dependências temporais de modo que a ordem em que as *threads* foram escalonadas se torna decisiva para a detecção de bugs;

- Podem impedir que o sistema seja divido em módulos independentes ou que utilize métodos de call-back, pois estes podem introduzir *deadlocks*;

- Podem reduzir o desempenho do sistema. Granularidade fina na utilização de *locks*, em geral, reduz o desempenho e aumenta a complexidade do programa. Além disso, a sobrecarga utilizada por um grande número de threads, pode comprometer o desempenho.

Por outro lado Cordeiro (2006) aponta que em outros trabalhos há o argumento de que o uso de threads é a melhor forma de abstração para a criação de programas onde há concorrência entre as operações, ressaltando a falta de uma biblioteca de threads em nível de usuário que permita um maior controle sobre itens como escalonamento interno das threads, mecanismos de sincronização mais leves e compiladores que realizem melhores otimizações.

Na arquitetura que utiliza *single thread*, ou *single-process* orientada a eventos, há somente um único processo servidor que despacha eventos. Nesta abordagem, que trabalha de forma assíncrona, o servidor é capaz de intercalar o processamento de eventos de diversas conexões, permitindo o processamento de múltiplas requisições sem deixar a CPU ociosa. (Cordeiro, 2006).

Cordeiro (2006, pp. 8-9) afirma que de modo geral, a utilização de arquiteturas baseadas em eventos mostra-se mais eficiente do que o uso de um grande número de *threads* para atingir o nível de concorrência necessária para o processamento de um grande número de requisições simultâneas.

Considerando o cenário de aplicações Web, a escolha e utilização de um determinado

servidor web por exemplo, está intimamente ligada à sua estratégia de I/O que pode ser tanto *multi-thread* como *single thread*, de alta performance, pois essa estratégia é que permitirá uma boa qualidade de serviço em situações onde servidores recebem milhares de requisições de clientes simultaneamente (Beltran, Carrera, Torres, & Ayguade, 2004).

1.5. Considerações finais

A escalabilidade é fundamental para o sucesso de muitas organizações que cada vez mais necessitam de agilidade e rapidez no fornecimento de informações e que fazem seus negócios na web. Nestes casos, um sistema escalável deve ser capaz de funcionar normalmente à medida em que cresce a sua demanda e deve permitir que mais recursos sejam adicionados a qualquer momento para o atendimento dessas novas demandas (Chieu, Mohindra, & Karve, 2011).

Há um debate sendo travado a décadas, que é a utilização de uma arquitetura baseada em threads ou a arquitetura baseada em eventos, principalmente no que diz respeito à sua adequação, a questão central é saber se threads ou eventos são melhores no cenário onde há concorrência de I/O (Dabek, Zeldvich, Kaashoek, Mazière, & Morris, 2002).

Sistemas que não são escaláveis podem apresentar problemas de desempenho e disponibilidade, causando grande impacto negativo às organizações.

Devido aos limites de escalabilidade impostos pelo modelo baseado em threads, muitos desenvolvedores têm optado pelo modelo baseado em eventos para gerenciar concorrência.

Nenhum dos métodos e arquiteturas apresentados neste capitulo resolve todos os problemas de escalabilidade em qualquer cenário. É responsabilidade do arquiteto de *software* junto com sua equipe definir a ferramenta mais adequada para o cenário mais adequado.

O próximo capítulo tratará especificamente de arquiteturas que utilizam o paradigma orientado a eventos.

2. TECNOLOGIAS QUE UTILIZAM O PARADIGMA ORIENTADO A EVENTOS

Há várias tecnologias que utilizam o paradigma orientado a eventos como por exemplo o servidor web Nginx que em certos cenários de alto tráfego consegue atender a mais de 500 milhões de requisições HTTP por dia (Nedelcu, 2013).

Outra tecnologia que utiliza este paradigma é o Node.js. Criado em 2009 por Ryan Dahl e mais 14 colaboradores, o Node.js é uma tecnologia que possui arquitetura totalmente não bloqueante, apresentando boa performance com consumo de memória e utilizando ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento (Pereira, 2013).

Neste capítulo vamos tratar dos conceitos que compõem a arquitetura baseada em eventos, tecnologias, suas limitações e vantagens, bem como estudos de caso onde há aplicação destas tecnologias.

2.1. Arquitetura baseada em eventos

Por definição, um evento é uma ocorrência dentro de um sistema ou domínio em particular. A palavra *evento* é também utilizada para definir uma entidade de programação que representa uma ocorrência em um sistema computacional (Etzion & Niblett, 2011).

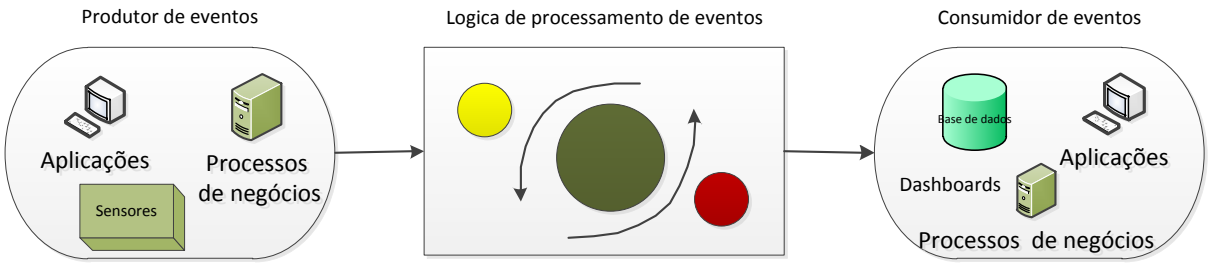
Este paradigma não é novo conforme explica Etzon & Niblett (2011, p. 7, tradução nossa):

> O uso de eventos em sistemas de informação não é novo. Nos primórdios da computação, eventos eram utilizados em forma de exceções cujo o papel era interromper o fluxo normal de execução e criando um processamento alternativo.
>
> Por exemplo, se um programa tentou dividir por zero, um evento de exceção seria lançado, permitindo que o programador interrompesse a execução do programa com uma mensagem de erro, ou permitir que seja executada uma ação corretiva e, em seguida, continuar com o processo de cálculo. Os eventos são utilizados em interfaces gráficas, onde os componentes de interface (por exemplo, botões ou menus) são projetados para reagir a eventos, como cliques do mouse ou teclas pressionadas.

Segundo a definição de (Candy & Schulte, 2009) a arquitetura orientada a eventos é um estilo arquitetural onde um ou mais componentes de um sistema de software executa uma

instrução em resposta ao recebimento de uma ou mais notificações de eventos. O componente de software que detecta um evento de negócios e envia a notificação é chamado de produtor de evento, já o componente que recebe a notificação e reage é o consumidor do evento. Este conceito é ilustrado na Figura 11:

Figura 11 - Exemplo de estrutura que mostra produtores de eventos e consumidores de eventos



Fonte: (Etzion & Niblett, 2011)

Candy & Schulte (2009) aponta cinco princípios que um sistema implementado utilizando a arquitetura orientada a eventos deve seguir:

- Individualidade: Cada evento é transmitido individualmente, onde o produtor do evento não permite que um lote de eventos se acumule antes de enviar o primeiro.
- *Push*: As notificações são "empurradas" pelo produtor do evento, não "puxadas" pelo consumidor do evento. Isso significa que o produtor do evento decide quando ele irá enviar uma notificação, minimizando o intervalo de tempo entre o evento do mundo real e o envio da mensagem de notificação.
- Imediatismo: O componente de software do consumidor faz algo em resposta imediatamente após receber a notificação.
- Mão-única: O produtor emite a notificação e passa a fazer outro trabalho. Ele não recebe resposta ou qualquer outra mensagem de retorno do consumidor do evento.
- Livre de comandos: A notificação, é somente um aviso, não um pedido de execução de um comando especifico. Ela não prescreve a ação que o consumidor do evento deverá executar, pois no consumidor contém a lógica que determina como ele vai reagir.

Etzon e Niblett (2011) classificam as aplicações orientadas a eventos nas categorias a seguir:

- Observação: O processamento baseado em eventos é utilizado para monitorar um sistema ou processo que verifica algum comportamento excepcional e gera alertas quando ocorre este comportamento. Neste caso, a reação, se houver, é deixada para os consumidores dos alertas, sendo o trabalho do aplicativo somente produzir o alerta, como por exemplo um sistema de monitoramento de pacientes, ou um sistema de monitoramento de bagagens.
- Disseminação da informação: Outra razão para se utilizar o processamento baseado em eventos é entregar a informação certa para o consumidor certo na granularidade certa, na hora certa, ou seja, a entrega de informações personalizadas. Um exemplo deste tipo são os sistemas de emergência que enviam alertas para socorristas.
- Comportamento operacional dinâmico: O processamento baseado em eventos é frequentemente usado para conduzir as ações executadas por um sistema de forma dinâmica, de modo a reagir a eventos de entrada. Neste caso o resultado da aplicação é diretamente afetado pelos eventos de entrada.
- Diagnósticos ativos: O objetivo da aplicação é diagnosticar um problema com base nos sintomas observados, por exemplo em caso de falha mecânica de um veículo ou um sistema de help-desk.
- Processamento preditivo: Nesta categoria o objetivo é identificar antecipadamente eventos que possam ocorrer, de modo que eles possam ser eliminados ou ter seus efeitos atenuados, como por exemplo um sistema de detecção de fraude.

Vale lembrar que estas categorias de utilização não são exclusivas, a utilização de uma ou mais categorias depende do cenário em que a aplicação funcionará.

2.2. NGINX

O servidor Web Nginx é atualmente o segundo servidor web open source mais utilizado de acordo com uma pesquisa da Netcraft. Nginx é utilizado por 15% dos principais sites da internet (P, R, & Kamath, 2015).

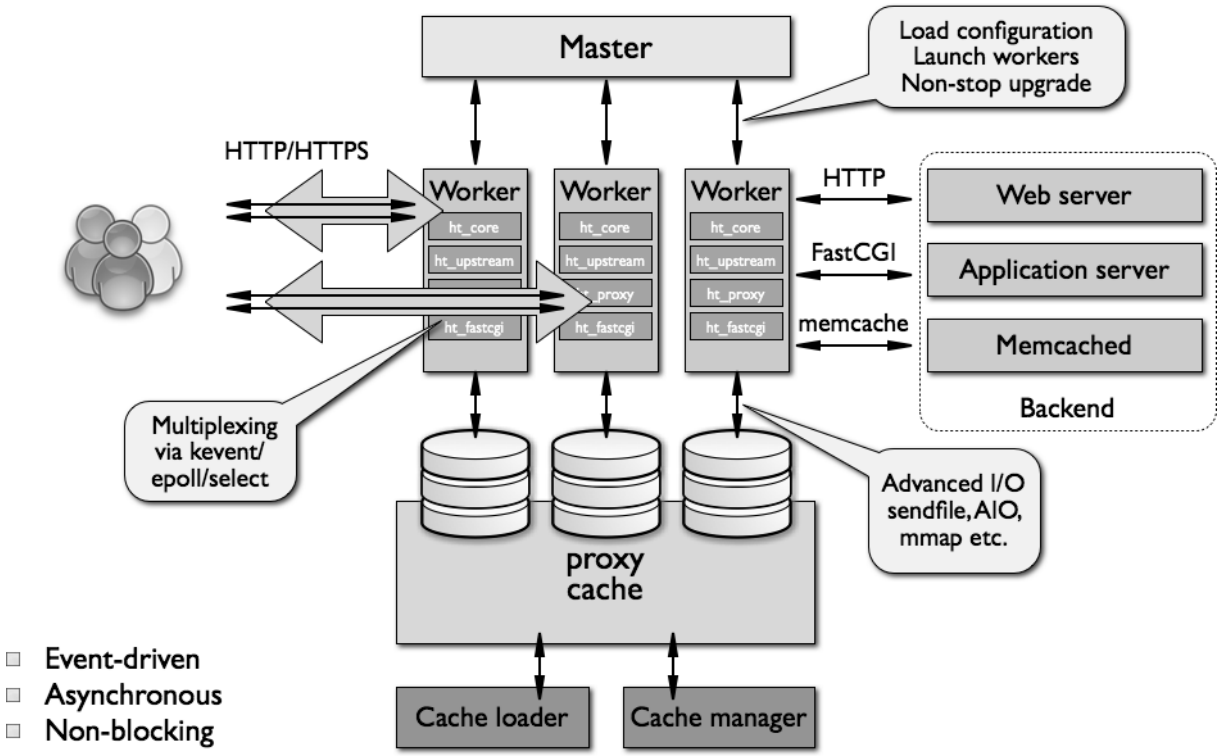
Sua criação se iniciou em 2002, por um desenvolvedor independente chamado Igor Sysoev, para atender as necessidades de alto tráfego de um site russo, o Rambler, que em 2008 recebeu mais de 500 milhões de requisições por dia. Atualmente o Nginx é utilizado em grandes sites como o Facebook, Netlflix, Wordpress, SourceForge, dentre outros, provando ser um servidor web muito eficiente, leve e poderoso (Nedelcu, 2013).

Segundo explica P, R, & Kamath (2015) o Nginx possui uma arquitetura modular, não-

bloqueante e orientada a eventos, conhecida como arquitetura de assimétrica de multi processamento. O principal processo orientado a evento manipula todos os processos associados às requisições HTTP. Quando uma operação no disco é necessária, o processo principal do servidor envia instruções a um auxiliar, que retorna notificação ao processo principal.

No *worker* do Nginx são carregados os módulos funcionais e o seu núcleo. O núcleo do Nginx é responsável por manter um laço de execução responsável por executar os módulos de código em cada estágio do processamento da requisição. A visão geral da arquitetura do servidor Nginx é ilustrada na Figura 12.

Figura 12 - Arquitetura do servidor Nginx



Fonte: (P, R, & Kamath, 2015)

2.3. Node.js

Devido à necessidade de revolver os problemas apresentados no capítulo anterior sobre a dificuldade em se escalar sistemas construídos sobre arquiteturas bloqueantes clássicas, em 2009, Ryan Dahl com a ajuda inicial de 14 colaboradores criou o Node.js (Pereira, 2013).

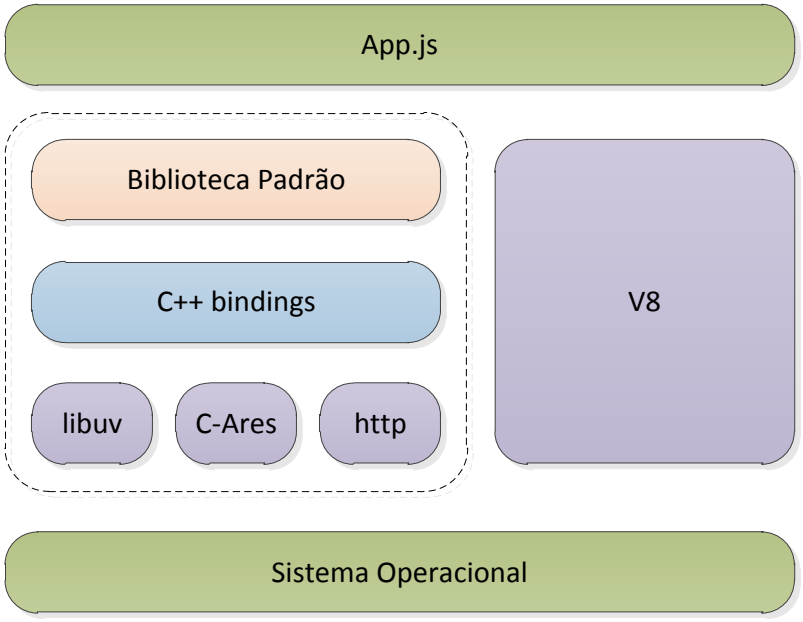
Node.js, é uma plataforma construída sobre a máquina virtual JavaScript V8, utilizada

no Google Chrome que possibilita o desenvolvimento de aplicações de rede facilmente escaláveis e rápidas utilizando a linguagem JavaScript. A utilização do V8 no lado servidor, permite que o código seja compilado para código de máquina nativo ao invés de utilizar somente código interpretado na forma de *bytecode* (Cantelon, Harter, Holowaychuck, & Rajlich, 2014).

A biblioteca padrão do Node é dividida em duas partes: Bibliotecas que fornecem o loop de eventos de I/O não bloqueante rápido para redes e sistemas de arquivos, como a libuv e a segunda parte sendo a sua biblioteca HTTP que provê os métodos de manipulação do protocolo (Young & Harter, 2015).

A Figura 13 apresenta uma visão geral e de alto nível de como estes componentes internos se encaixam.

Figura 13 - Visão geral das partes que compões o Node.js



Fonte: (Young & Harter, 2015)

A princípio Ryan Dahl optou por utilizar a linguagem C para construir projetos utilizando Node, mas a manutenção se tornou bastante complexa o que fez com que ele optasse pela linguagem Lua, mas a mistura de bibliotecas bloqueantes e não bloqueantes poderia confundir os desenvolvedores, o que fez com que ele finalmente optasse por utilizar JavaScript (Teixeira, 2013).

Segundo também aponta Teixeira, (2013) JavaScript possui recursos importantes como *closures* e funções de primeira classe, tornando-se uma ferramenta poderosa na

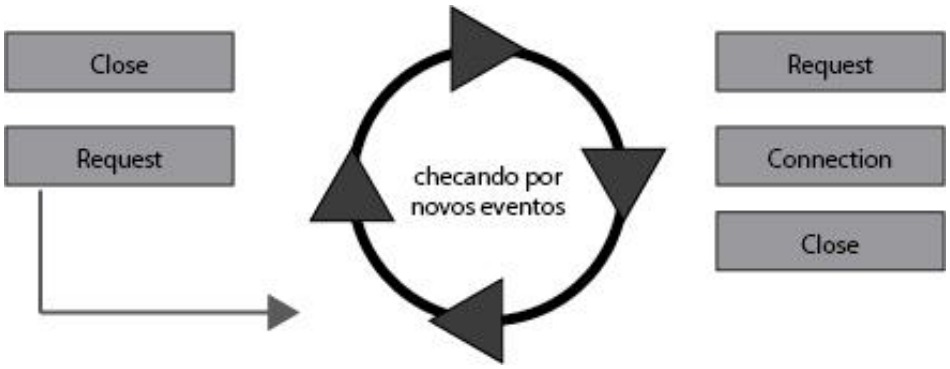
programação orientada a eventos. Esta capacidade contribuiu muito para que o Node.js se tornasse tão popular.

Conforme explicam Pasquali (2013) e Pereira (2013) o Node.js processa as instruções JavaScript em uma única *thread*, ou seja cada aplicação terá instância de um único processo, atendendo as chamadas através do *Event-Loop*.

O *Event-Loop*, visualizado na Figura 14, de acordo com Pereira (2013), é um mecanismo interno, dependente das bibliotecas *libev* e *libeio* escritas em C, sendo o agente responsável por escutar e emitir eventos no sistema, como um loop infinito que a cada iteração verifica em sua fila de eventos se um determinado evento foi emitido. O EventEmitter é o módulo responsável por emitir eventos na maior parte das bibliotecas do Node.js.

Quando um determinado código emite um evento, o mesmo é enviado para a fila de eventos onde o Event-Loop irá executá-lo e em retorne seu call-back.

Figura 14 - Event-Loop no Node.js



Fonte: (Pereira, 2013)

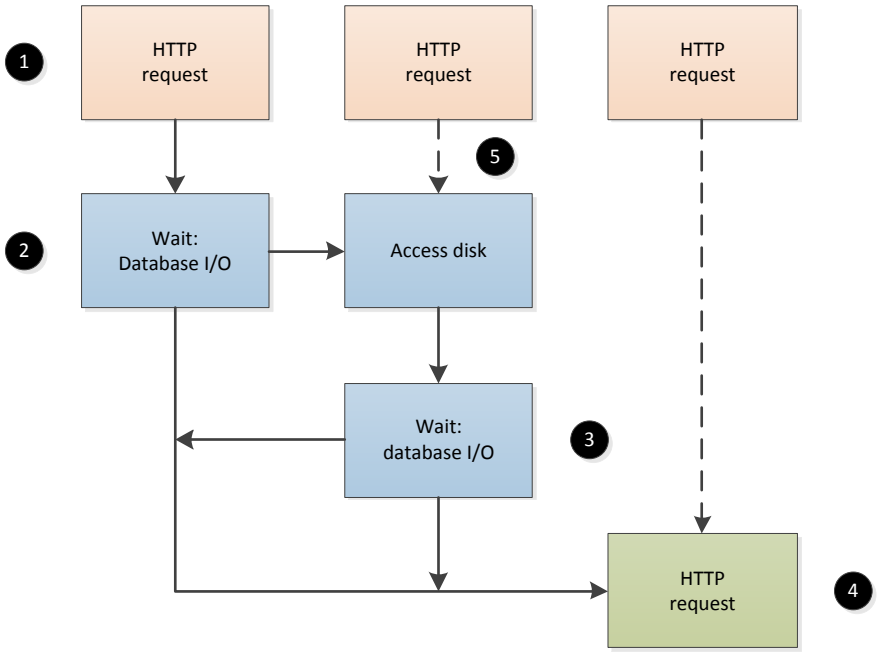
No exemplo ilustrado pela Figura 15, Young & Harter (2015) apresentam o cenário um servidor de propagandas, onde o Node.js utiliza sua API de processamento e manipulação de arquivos e rede assíncrona da seguinte forma:

(1) Uma requisição HTTP é recebida e analisada pelo módulo HTTP do Node. A aplicação então realiza uma consulta na base de dados utilizando uma API assíncrona que executa um call-back para a função de leitura de dados (2).

Enquanto o Node aguarda esta função terminar a execução, o servidor está pronto para ler um arquivo estático no disco que representa uma página web (3). Uma vez que a requisição ao banco de dados é finalizada, os resultados são usados no processamento da resposta (4).

Enquanto este processamento acontece, outras requisições são recebidas pelo servidor e são executadas de acordo com os recursos disponíveis.

Figura 15 - Servidor de propagandas construído com Node.js



Fonte: (Young & Harter, 2015)

Neste cenário, sem se preocupar com manipulação de threads, o Node usa recursos de I/O de forma bastante eficiente, somente usando técnicas de programação em JavaScript (Young & Harter, 2015).

Com Node.js é possível trabalhar com outros protocolos além do HTTP, como o HTTPS, FTP, SSH, DNS, TCP, UDP, Websockets e outros protocolos que são disponíveis através de módulos desenvolvidos pela comunidade (Pereira, 2013).

O modelo de execução assíncrona não bloqueante do Node.js segundo Ihrig (2013) fornece soluções extremamente escaláveis para servidores com sobrecarga mínima, sendo utilizado por empresas de grande porte incluindo Microsoft, LinkedIn, Yahoo! e Walmart.

2.3.1. Frameworks

A comunidade Node.js é bastante ativa. Seu repositório de pacotes oficial e gerenciador de pacotes chamado NPM (NPM, 2015), conta com quase 200.000 pacotes desenvolvidos pela comunidade.

Segundo Pereira (2013) criar sistemas de grande porte somente utilizando a API nativa do Node.js pode ser uma tarefa trabalhosa, pois conforme surge a necessidade de se implementar novas funcionalidades, várias linhas de código são escritas, contribuindo para o aumento da complexidade do projeto e dificultando futuras manutenções.

Devido a este revés, foram criados frameworks para desenvolvimento utilizando a plataforma Node.js, dos quais trataremos a seguir.

2.3.1.1. Express

O Express é um framework baseado no módulo HTTP do Node.js e componentes Connect que são chamados de middleware e que acrescentam funcionalidades ao framework (Mardan, 2014).

Sua estrutura é bastante simples e não opinativa, permitindo que o desenvolvedor tenha flexibilidade na organização da estrutura da aplicação. Fornece também recursos como roteamento baseado em caminhos de URL, gerenciamento de sessão, análise de requisições, todos eles utilizado no desenvolvimento web (Vladutu, 2014).

Pereira (2013) também lista as outras características do framework:

- Permite trabalhar também com MVR (Model-View-Routes) ou MVC (Model-View-Controller);
- Roteamento de urls via callbacks;
- Middleware;
- Interface RESTful;
- Suporte a File Uploads;
- Configuração baseada em variáveis de ambiente;
- Suporte a helpers dinâmicos;
- Integração com Template Engines;
- Integração com SQL e NoSQl.

Na Figura 16 o exemplo de uma aplicação simples utilizando o Express onde é iniciado um servidor que escuta conexões na porta 3000 e responde com uma mensagem “Hello World” na URL raiz (“/”) do servidor.

Figura 16 - Exemplo de criação de um servidor utilizando o Express

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host,
      port);
});
```

2.3.1.2. Restify

O Restify (2015) é baseado no Express e é utilizado na criação de APIs RESTful. Enquanto o Express atende a aplicações que utilizam funcionalidades do próprio browser, o Restify é mais enxuto, tornando a criação de API mais performática. Na Figura 17 vemos um pequeno exemplo de criação de uma aplicação utilizando o framework Restify:

Figura 17 - Exemplo de aplicação utilizando o Restify

```
var restify = require('restify');

function respond(req, res, next) {
  res.send('hello ' + req.params.name);
  next();
}

var server = restify.createServer();
server.get('/hello/:name', respond);
server.head('/hello/:name', respond);

server.listen(8080, function() {
  console.log('%s listening at %s', server.name, server.url);
});
```

Fonte: (Restify, 2015)

A Figura 18 mostra o tipo de cabeçalho de resposta capturado pelo comando *curl* ao acessar a URL da aplicação escrita utilizando o Framework Restify:

Figura 18 - resposta da aplicação feita em Restify

```
$ curl -is http://localhost:8080/hello/mark -H 'accept: text/plain'
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 10
Date: Mon, 31 Dec 2012 01:32:44 GMT
Connection: keep-alive

hello mark


$ curl -is http://localhost:8080/hello/mark
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 12
Date: Mon, 31 Dec 2012 01:33:33 GMT
Connection: keep-alive

"hello mark"


$ curl -is http://localhost:8080/hello/mark -X HEAD -H 'connection: close'
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 12
Date: Mon, 31 Dec 2012 01:42:07 GMT
Connection: close
```

Fonte: (Restify, 2015)

2.3.1.3. Sails.js

O Sails.js (2015) é um framework também baseado no Express que utiliza uma estrutura MVC semelhante ao do Ruby on Rails, com ferramentas de geração de APIs REST e uma ORM chamada Waterline que provê acesso a vários tipos de bancos de dados. O Sails também possui uma implementação de websockets para a utilização em aplicações em tempo real e jogos multiplayer.

2.3.1.4. Krajenjs

O Krakenjs (2015) é um framework criado pela equipe de desenvolvimento do PayPal baseado no Express e que estende algumas funcionalidades através dos seguintes módulos que podem ser utilizados de forma independente:

- Lusca: módulo de implementa a camada de segurança da aplicação;
- Kappa: módulo de proxy;
- Makara: módulo de internacionalização de templates da página;

- Adaro: módulo de templates que utiliza o DustJS para renderização das páginas.

2.3.1.5. Hapi

Desenvolvido pela equipe do Walmart Labs que depois abriu o código para a comunidade, o Hapi (2015) é utilizado na criação de API e outros tipos de aplicações web. Diferente dos frameworks apresentados anteriormente, o Hapi foi criado em torno da filosofia de que a configuração é melhor do que o código, que a lógica de negócios deve ficar isolado da camada de transporte, fornecendo um ambiente moderno e abrangente em que o esforço gasto deve ser utilizado para a entrega de software de valor para o negócio. O modelo de configuração das rotas no framework e criação de uma aplicação simples é ilustrada pela Figura 19.

Figura 19 - Exemplo de aplicação utilizando Hapi

```
var Hapi = require('hapi');

var server = new Hapi.Server();
server.connection({ port: 3000 });

server.route({
    method: 'GET',
    path: '/',
    handler: function (request, reply) {
        reply('Hello, world!');
    }
});

server.route({
    method: 'GET',
    path: '/{name}',
    handler: function (request, reply) {
        reply('Hello, ' + encodeURIComponent(request.params.name) + '!');
    }
});

server.start(function () {
    console.log('Server running at:', server.info.uri);
});
```

2.3.2. Limitações e princípios de desenvolvimento

Segundo Pereira (2013) é fundamental que o código escrito em Node.js invoque o mínimo possível de funções bloqueantes, pois cada função síncrona impedirá, naquele instante, que o Node.js continue executando demais códigos até que aquela função seja finalizada. Por isso é recomendado sempre utilizar funções assíncronas para aproveitar a característica

principal do Node.js.

Pereira (2013) apresenta nas figuras 20, 21, 22 e 23 as diferenças entre a execução de uma chamada síncrona e assíncrona em relação a uma determinada linha do tempo:

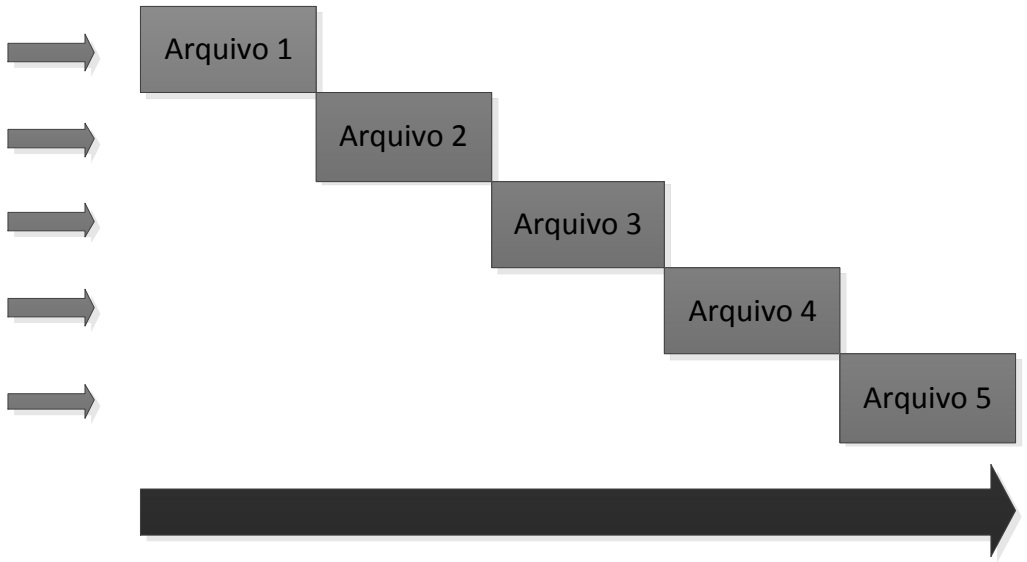
Figura 20 – Exemplo de código de execução síncrona

```
var fs = require('fs');

for(var i = 1; i <= 5; i++) {
    var file = "sync-txt" + i + ".txt";
    var out = fs.writeFileSync(file, "Hello Node.js!");
    console.log(out);
}
```

Fonte: (Pereira, 2013)

Figura 21 - Linha do Tempo de uma execução síncrona



Fonte: (Pereira, 2013) adaptado pelo autor

Figura 22 – Exemplo de código de execução assíncrona

```
var fs = require('fs');

for(var i = 1; i <= 5; i++) {
    var file = "async-txt" + i + ".txt";
    fs.writeFile(file, "Hello Node.js!", function(err, out) {
        console.log(out);
    });
}
```

Fonte: (Pereira, 2013)

Figura 23 - Linha do Tempo de uma execução assíncrona

Fonte: (Pereira, 2013) adaptado pelo autor

Outro exemplo citado por Cantelon, Harter, Holowaychuck, & Rajlich (2014) fala sobre situações onde a aplicação realiza cálculos matemáticos que envolvem o uso intensivo da CPU, como o cálculo de uma sequência Fibonacci apresentado na Figura 24.

Figura 24 - Implementação não otimizada de um Servidor HTTP que realiza o cálculo Fibonacci em Node.js

```
var http = require('http');

function fib(n) {
    if(n < 2) {
        return 1;
    } else {
        return fib(n - 2) + fib(n - 1);
    }
}


var server = http.createServer(function(req, res) {
    var num = parseInt(req.url.substring(1), 10);
    res.writeHead(200);
    res.end(fib(num) + "\n");
});

server.listen(8000);
```

Fonte: (Cantelon, Harter, Holowaychuck, & Rajlich, 2014)

Neste exemplo, enquanto a *thread* do servidor Node está calculando o resultado, não é possível atender nenhuma requisição HTTP. A melhor solução para este caso é utilizar processos filhos para calcular a sequência e enviar depois o resultado utilizando o modulo *child_process* para tal situação.

A implementação envolve dois arquivos, fibonacci-server.js ilustrado na Figura 25, que será o servidor e fibonacci-calc.js que realizará o cálculo.

Figura 25 - fibonacci-server.js

```
var http = require('http');
var cp = require('child_process');

var server = http.createServer(function(req, res) {
    var child = cp.fork(__filename, [req.url.substring(1)
    child.on('message', function(m) {
        res.end(m.result + '\n');
    });
});

server.listen(8000);
```

Fonte: (Cantelon, Harter, Holowaychuck, & Rajlich, 2014)

O servidor usa o método *cp.fork()* para adicionar a lógica de cálculo em um processo

separado do Node, que irá informar o processo pai usando process.send(), conforme o exemplo de código visto na Figura 26:

Figura 26 - fibonacci-calc.js

```
function fib(n) {
    if(n < 2) {
        return 1;
    } else {
        return fib(n - 2) + fib(n - 1);
    }
}

var input = parseInt(proccess.argv[2], 10);
proccess.send({ result: fib(input) });
```

Fonte: (Cantelon, Harter, Holowaychuck, & Rajlich, 2014)

Neste caso o método *proccess.send()* envia o resultado da função *fib()* que é executada em um processo filho para o processo pai sem interferir na execução do mesmo.

Pereira (2013) aponta que mesmo sendo vantajoso e performático trabalhar de forma assíncrona é preciso tomar cuidado para que divesas funções assíncronas encadeadas se tornem complexas a ponto de dificultar a manutenção de código. Esta situação mostrada na Figura 27 é chamada pelos desenvolvedores Node.js de *callback hell*.

Figura 27 - Exemplo de *callback hell*

```
var fs = require('fs');
fs.readdir(__dirname, function(erro, contents) {
    if (erro) { throw erro; }
    contents.forEach(function(content) {
        var path = './' + content;
        fs.stat(path, function(erro, stat) {
            if (erro) { throw erro; }
            if (stat.isFile()) {
                console.log('%s %d bytes', content, stat.size);
            }
        });
    });
});
```

Fonte: (Pereira, 2013)

2.4. Estudos de casos

A seguir serão apresentados dois estudos de caso onde foi utilizado Node.js em conjunto com outras tecnologias para criação de uma aplicação com interação em tempo real com seus usuários e na criação de um serviço de troca de mensagens mais robusto.

2.4.1. Virtual IBM Whiteboard for mobile (Jaramillo, Duy, & Newhook, 2014)

Devido a necessidade de comunicação entre colaboradores da IBM localizados em várias partes do mundo, o time de inovação chamado CIO percebeu que seu modelo de seu sistema de vídeo conferencia era ineficiente, pois em suas reuniões, cada um os participantes da conferencia transmitia a imagem de um quadro branco ao invés de mostrar o próprio rosto. Mesmo com ótimas ferramentas de exibição de apresentações, os membros gostariam de poder compartilhar a tela como se ela fosse um quadro branco, então surgiu software *CIO Whiteboard*.

O CIO Whiteboard foi criado em 2012 sendo um aplicativo colaborativo para dispositivos móveis, multiusuário que permite desenhar e compartilhar telas em tempo real entre duas ou mais pessoas de forma remota. Para desenhar ou escrever nestas telas o usuário pode utilizar uma caneta ou o dedo numa interface bastante simples e fácil.

Este aplicativo foi implementado como um aplicativo hibrido desenvolvido com Apache Cordova 3.1 que se comunica com um servidor implementado em JavaScript com Node.js. O Servidor é responsável por estabelecer e gerenciar as sessões entre os clientes, e controlar os eventos de desenho das telas em tempo real e enviar a todos os clientes da sessão. Na persistência dos dados da sessão é utilizado o CouchDB.

Toda a comunicação cliente/servidor é controlada por *Websocket* utilizando o IBM Mobile Connect, permitindo a comunicação bidirecional e em tempo real entre o servidor e o cliente.

A utilização de tecnologias recentes como Node.js e CouchDB é parte da missão do laboratório de inovação do CIO e quem permitem a comunicação via *Websocket* por exemplo, permite que todos os usuários vejam o desenho realizado por um outro usuário em tempo real, trazendo dinamismo às iterações entre os participantes da conferencia, bem como recursos outros recursos básicos de desenho como escolher cor, apagar traços, desfazer/refazer, etc. O

O CIO Whiteboard possui versões para iPad e para o navegador Firefox (Jaramillo, Duy, & Newhook, 2014).

2.4.2. MIT iLab Service Broker

O *iLab Shared Architecture*, ou ISA, criado pelo Instituto de Tecnologia de Massachusetts é uma infraestrutura de serviços web que permite o acesso online aos seus laboratórios e equipamentos. O ISABM, composto por um *Service Broker*, software cliente e servidor permite um suporte a experimentos onde a especificação é definida antes da execução. Escrito inicialmente em C# e utilizando o Microsoft .NET Framework, este sistema apresentava a limitação de ser executado somente em ambientes com o ecossistema Windows (Colbran & Schulz, 2015).

Segundo Colbran & Schulz (2015), Len Payne reescreveu o ISABM entre 2011 2012 em Java para remover a dependência da plataforma Microsoft Windows, mantendo a implementação do protocolo de serviços SOAP devido à compatibilidade com servidores dos laboratórios existentes, mas a integração com outros ambientes educacionais era difícil devido a problemas arquiteturais e uma base de código bastante complexa para adição de novas funcionalidades.

A substituição do *iLab Service Broker*, por outro serviço desenvolvido pela equipe do MIT, chamado de Agente, escrito em Node.js possibilitou a diminuição da sobrecarga dos servidores, permitiu a escrita e modificação rápida de código, sem a necessidade de compilação. Este novo Agente, ao invés de fazer o repasse das funcionalidades para outros serviços como era feito anteriormente pelo *Service Broker*, como por exemplo autenticação de usuário, passou a atender a todas estas requisições com maior flexibilidade já que permitiu a utilização de *OAuth* e interfaces únicas para diferentes grupos de usuários (Colbran & Schulz, 2015).

2.5. Considerações finais

A arquitetura orientada a eventos, apesar de não ser um paradigma novo, tem se mostrado bastante eficaz quando o cenário de implementação exige alta concorrência de recursos. Este tipo de arquitetura também fornece alta eficiência, pois utiliza de forma mais otimizada o hardware existente, diminuindo a escalabilidade horizontal e vertical e por consequência diminuindo os custos de hardware.

O Node.js é uma ferramenta bastante viável na construção de sistemas orientados a eventos, e altamente escaláveis, sendo adotada por grandes *players* do mercado. Se utilizado de forma padronizada e utilizando as melhores práticas, o Node.js pode ser uma grande opção, frente às oferecidas atualmente, como o Tornado ou Event Machine.

Vale salientar que decisões de arquitetura como esta devem ser aplicadas de acordo com o cenário, pois em aplicações que exigem alta demanda de processamento da CPU, Node.js pode não ser a melhor opção, apesar de que já é possível a utilização do mesmo em *clusters*. Já em sistemas Web como Web API's e sistemas em Tempo Real, o Node.js mostra todo o seu potencial.

3. SISTEMA COLABORATIVO PARA MAPEAMENTO E AVALIAÇÃO DE PONTOS DE ACESSIBILIDADE

Este trabalho teve como objetivo desenvolver sistema colaborativo que realiza o mapeamento de pontos de acessibilidade via Web, com o propósito de facilitar a mobilidade de pessoas com necessidades especiais, permitindo não só localizar os pontos, mas também verificar suas condições através dos dados inseridos pelos próprios usuários.

Neste capítulo será apresentado o cenário onde o sistema pode ser utilizado, características do sistema, suas funcionalidades e a apresentação da arquitetura proposta, bem como os componentes envolvidos.

3.1. Cenário

A forma com que vivemos, a quantidade, velocidade da informação que é recebida através da internet e a qualidade de vida motivada pelos avanços da medicina dentre outras se torna possível através não só do desenvolvimento tecnológico, mas também do impacto que a tecnologia causa na vida das pessoas.

Mesmo com todos estes avanços, o descaso da sociedade com as pessoas portadoras de necessidades especiais, salvo exceções e baseado no dia-a-dia dessas pessoas, ainda é grande.

Aos poucos, devido principalmente à lei de número 10.098, de 19 de dezembro de 2000 (Palácio do Planalto, 2000), que estabelece normas gerais e critérios para a promoção da acessibilidade de pessoas portadoras de deficiência ou com mobilidade reduzida, os estabelecimentos públicos e privados junto com o Governo Federal tem realizado ações para atender a tal exigência.

É fato que ao utilizarmos o transporte coletivo, andar pelas ruas, ir ao shopping, passear pelo parque, somos surpreendidos pela dificuldade que pessoas portadoras de deficiência tem em pleno século XXI de se locomover dignamente devido a pontos de acessibilidade ineficientes ou até inexistentes. É preciso analisar se após quase 15 anos da sanção desta lei as reinvindicações destas pessoas foram ouvidas e se de fato elas usufruem dos benefícios desta lei.

Ainda em 2015 não se tem notícia de uma ferramenta que proporcione a pesquisa de forma fácil e rápida de pontos de acessibilidade e sua categorização em determinadas áreas geográficas e estabelecimentos públicos e privados pelo país.

Há também a necessidade de que os pontos existentes possam ser sinalizados e

avaliados pelos próprios usuários, pois mesmo que existam os pontos de acessibilidade conforme diz a lei, não há um registro da eficácia do ponto e satisfação por parte do usuário.

Outra necessidade importante é de um canal de fácil acesso via internet onde os portadores de necessidades especiais possam sugerir pontos em locais onde os portadores de necessidades especiais frequentam e ainda não existem.

Para que uma ferramenta desse tipo atinja o maior número possível de pessoas é imprescindível que o canal mais acessível seja a internet, pois segundo a pesquisa realizada em 2013 pelo Centro de Estudos das Tecnologias de Informação e Comunicação no Brasil (Barbosa, 2014), estimou-se, em números absolutos, que 30,6 milhões de domicílios brasileiros possuem um computador, seja ele de mesa, portátil ou tablet. Este número correspondia a 49% dos domicílios, sendo que em 2008 este número era de apenas 25%.

Devemos considerar também o crescimento de linhas de telefonia móvel em operação registradas pela Agencia Brasileira de Telecomunicações (ANATEL. Agencia Brasileira de Telecomunicações, 2015), que em janeiro de 2015 atingiu a marca de 281,70 milhões de linhas, sendo que 75,75% são linhas pré-pagas.

Outro dado importante é sobre o número de portadores de necessidades especiais no Brasil. De acordo com os dados do Censo de 2010 realizado pelo Instituto Brasileiro de Geografia e Estatística (Instituto Brasileiro de Geografia e Estatística - IBGE, 2012), 45.606.048 milhões de brasileiros declaram ter pelo menos um tipo de deficiência, valor este que corresponde a 23,9% da população brasileira.

Levando em consideração todos estes dados, o fator bastante relevante a ser considerado na construção de uma ferramenta que atenda a este grande número de usuários é a escalabilidade da sua arquitetura.

3.2. Características do protótipo

O protótipo foi construído com base na arquitetura orientada a eventos e I/O não bloqueante e escalável, que permitirá a portadores de necessidades especiais, entusiastas, ONG’s, repartições públicas e estabelecimentos privados, obter informações a respeito dos pontos de acessibilidade que compõem os seus trajetos diários ou de acordo com sua necessidade, auxiliando na tomada de decisões sobre quais trajetos permitem uma locomoção menos complicada e mais acessível.

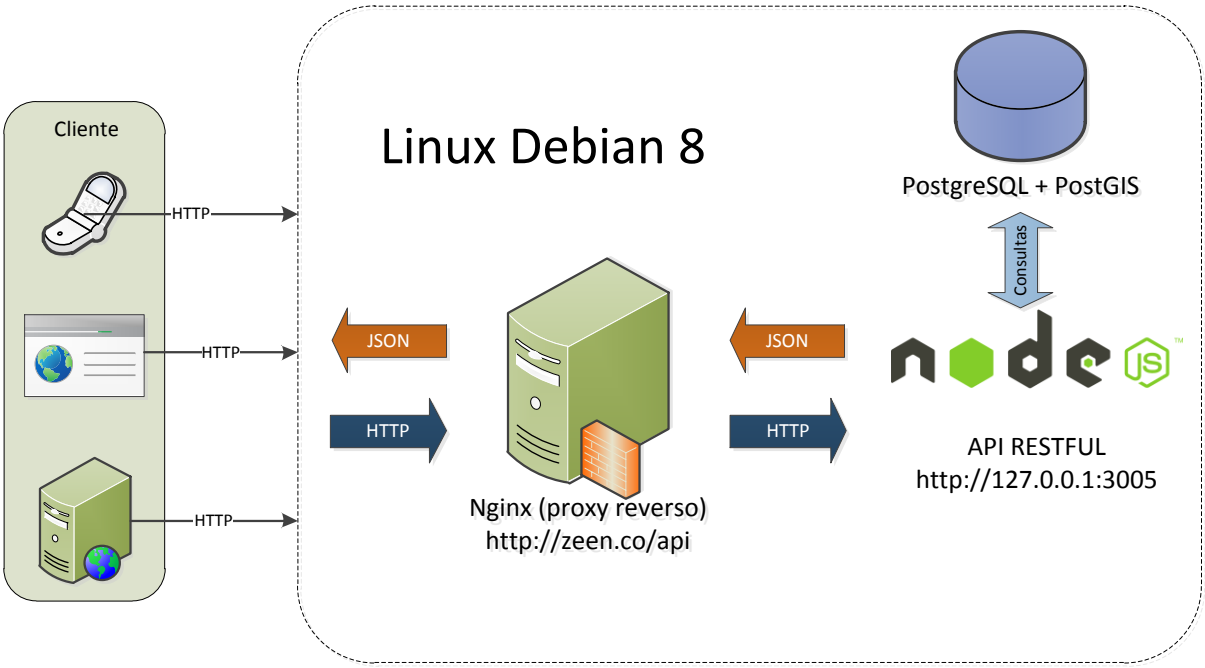
Os dados informados sobre a quantidade, qualidade e nível de satisfação no uso dos pontos de acessibilidade, permite não só que outros usuários saibam de antemão sobre suas

condições, mas também de base para que órgãos públicos e privados envolvidos tomem as devidas providencias e, se for o caso, tragam melhorias.

3.3. Arquitetura do Protótipo

A arquitetura da aplicação protótipo é composta de tecnologias de back-end e front-end, representada no diagrama da Figura 28:

Figura 28 - Diagrama da Arquitetura proposta



Fonte: o autor

- ***Cliente:*** O cliente é representado por dispositivos, aplicações e outros serviços que tenham a capacidade de consumir os recursos da API através do protocolo HTTP e manipular os dados de retorno representados pelo formato JSON.
- ***Proxy reverso:*** A função do proxy reverso nesta arquitetura é interceptar todas as requisições HTTP feitas pelo cliente e repassar estas solicitações à API RESTful.
- ***API RESTful:*** A API RESTful (Richardson & Ruby, 2007) compõe a lógica de negócios do protótipo é responsável por prover serviços para o processamento, roteamento dos recursos e consumo dos dados fornecidos pelos clientes.

O acesso aos recursos da API pública se dá através requisições ao servidor por meio de métodos, ou verbos, HTTP conforme a Tabela 2:

Tabela 2 - Recursos da API para acesso a dados dos usuários da aplicação

Recurso: Users				
URL: http://zeen.co/api/				
URI	Verbo	Parâmetros	Descrição	Resposta
/users	GET	Não se aplica	Retorna a lista de usuários do sistema	JSON
/users	POST	full_name = String email = String password = String	Cria um usuário no sistema	JSON
/users/<id>	GET	id = Inteiro	Retorna os dados do usuário conforme o id especificado	JSON
/users/<id>	PUT	id = Inteiro full_name = String email = String password = String	Edita dados do usuário conforme o id especificado	JSON
/users/<id>	DELETE	id = Inteiro	Destrói dados de cadastro do usuário conforme id especificado	JSON

Fonte: O autor

A API utiliza o formato JSON para enviar os dados de resposta conforme ilustrado pela Figura 29:

Figura 29 -Resposta da API ao consumir o recurso users

Fonte: O autor

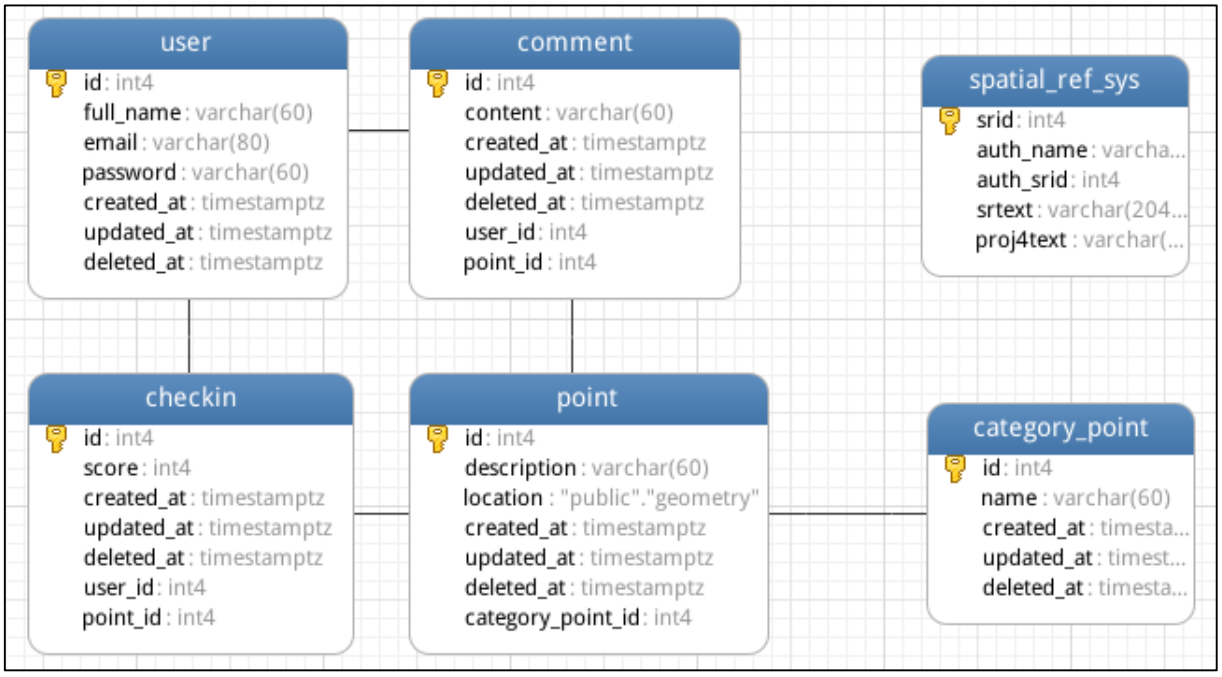
Tabela 3 - Recursos da API para acesso a dados de pontos de acessibilidade

Recurso: Points				
URL: http://zeen.co/api/				
URI	Verbo	Parâmetros	Descrição	Resposta
/points	GET	Não se aplica	Retorna a lista de pontos	JSON
/points	POST	description = String category_point_id = Inteiro location = GeoJSON	Cria um novo ponto.	JSON
/points/location/<latitude>/<longitude>	GET	latitude = Real longitude = Real	Retorna pontos próximos ao ponto com as coordenadas especificadas	JSON
/points/categories	GET	Não se aplica	Retorna a lista das categorias de pontos.	JSON
/points/categories	POST	name = String	Cria uma nova categoria de ponto.	JSON

Fonte: O autor

- ***Banco de Dados:*** O banco de dados local persistirá todos os dados da aplicação, inclusive os dados de geolocalização que serão consultados de devolvidos pela API no formato JSON. A modelagem de numa visão de alto nível é presentada pela Figura 30:

Figura 30 - Modelagem do Banco de Dados



Fonte: O autor

3.4. Tecnologias utilizadas

Para a criação do protótipo foram utilizadas diversas tecnologias que compõem o *back-end* e *front-end* da aplicação e apresentadas a seguir.

3.4.1. Back-End

- ***Sistema Operacional Debian:*** O Debian é um sistema operacional livre que utiliza o *kernel* Linux em sua composição e é conhecido pela comunidade por sua estabilidade devido a exaustivos testes realizados em seus pacotes antes do lançamento de novas versões (Debian, 2015). A versão utilizada para este projeto é a 8.
- ***Nginx:*** Nginx, ou *engine x*, é um servidor HTTP, servidor de proxy reverso, servidor de mail proxy e servidor proxy genérico para o protocolo TCP, responsável por atender 23.66% dos sites mais acessados no mundo de acordo com o Netcraft (Nginx, About Nginx, 2015). Possui o código aberto sob a licença BSD. A escolha do Nginx versão 1.6.2 para este projeto também se deve à sua arquitetura orientada a eventos, de acordo com a proposta apresentada na arquitetura.

- ***Node.js:*** Plataforma mantida pela Joyent e de código aberto, escrita sobre a Engine JavaScript V8 do Chrome, Node.js usa o modelo orientado a eventos de I/O não bloqueante, projetado para ter baixa latência e alta performance (Joyent, 2015). A versão utilizada neste projeto é a 5.0.0.
- ***Express:*** Express é um framework para Node.js, minimalista e flexível para criação de aplicações web e um conjunto robusto de recursos que permitem a criação de *APIs* para comunicação com aplicações moveis e *web services* (Project, 2015).
- ***Sequelize:*** Para a manipulação de dados do no banco de dados com Node.js e o framework Express será utilizada o ORM Sequelize. O Sequelize é um ORM baseado em promise (MDN, 2015) para Node.js e suporta os dialetos para os bancos PostgreSQL, MySQL, MariaDB, SQLite e MSSQL. Possui suporte a relacionamentos, replicação, transações, dentre outros (Sequelize, 2015).
- ***PM2:*** Em produção e no ambiente de testes da aplicação será utilizado o PM2 como gerenciador de processos de aplicações Node.js. O PM2 possui um balanceador de carga embutido e permite que seja realizado deploy da aplicação sem que seja necessário deixar a aplicação indisponível, além de gerar estáticas de uso através do Keymetrics (Keymetrics, 2015).
- ***PostgreSQL:*** O PostgreSQL é um poderoso sistema de banco de dados objeto-relacional de código aberto com mais de 15 de atividade e disponível para a grande maioria dos sistemas operacionais. É altamente escalável, tanto em número de usuários simultâneos quanto a quantidade de dados que ele pode gerenciar, de acordo com a Tabela 4 (PostgreSQL, 2015):

Tabela 4 - Limites do PostgreSQL

Limites do PostgreSQL	
Limite	Valor
Tamanho máximo da base de dados	Ilimitado
Tamanho máximo da tabela	32 terabytes
Tamanho máximo da tupla	1.6 terabytes
Tamanho máximo de campo	1 gigabyte
Quantidade máxima de tuplas por tabela	Ilimitada
Quantidade máxima de colunas por tabela	De 250 a 1600, dependendo do tipo da coluna
Quantidade máxima de índices por tabela	Ilimitado

Fonte: (PostgreSQL, 2015)

Para o projeto será utilizada a versão 9.4.5 do PostgreSQL com a utilização da extensão PostGIS.

- ***PostGIS:*** Para que o PostgreSQL possa armazenar e recuperar dados de geolocalização dos pontos de acessibilidade que serão armazenados pelo protótipo é necessário utilizar a extensão PostGIS. O PostGIS adiciona várias funcionalidades que permite um robusto gerenciamento de dados espaciais (PostGIS, 2015).
- ***JSON:*** O formato de notação de objetos JavaScript, ou JSON é utilizado por muitos serviços web para troca de dados em substituição ao formato XML (Smith, 2015). Na API a ser desenvolvida todo o conteúdo de resposta será no formato JSON para facilitar o consumo do serviço por vários tipos de clientes.
- ***HTML 5 Geolocation API:*** Para a que a localização do usuário seja capturada na execução da aplicação será utilizada a API de geolocalização do HTML 5 (W3C, 2015) através da confirmação do próprio usuário.

3.4.2. Front-End

- ***Bootstrap:*** Para a implementação de uma interface responsiva que permita uma boa experiência de uso não só no desktop, mas também em dispositivos móveis e tablets, será utilizado o framework Bootstrap, devido à sua facilidade de utilização e boa documentação (Bootstrap, 2015).
- ***jQuery:*** jQuery é uma biblioteca JavaScript rápida, pequena e com recursos que facilitam a manipulação do DOM (*Document Object Model*), eventos e implementação de Ajax. jQuery também é testada em vários navegadores, o que garante bastante versatilidade na implementação de JavaScript para a Web (jQuery, 2015).
- ***Google Maps API:*** A API do Google Maps (Google, 2015) será utilizada para a criação do mapa onde o usuário fará a interação com a aplicação, buscando pontos de acessibilidade e fazendo marcações no mesmo. Vale lembrar que este ambiente foi designado para fins de estudo e teste, pois em um cenário real é necessário utilizar um ou mais servidores para o banco de dados para que o mesmo não utilize os recursos de processamento da aplicação executada sobre a plataforma Node.js.

Toda base de código será versionada utilizando o Sistema de Controle de Versão

Distribuído GIT (GIT, 2015) devido à sua robustez e possibilidade de armazenamento grátis utilizando o Bitbucket (Atlassian, 2015).

3.5. Funcionalidades do protótipo

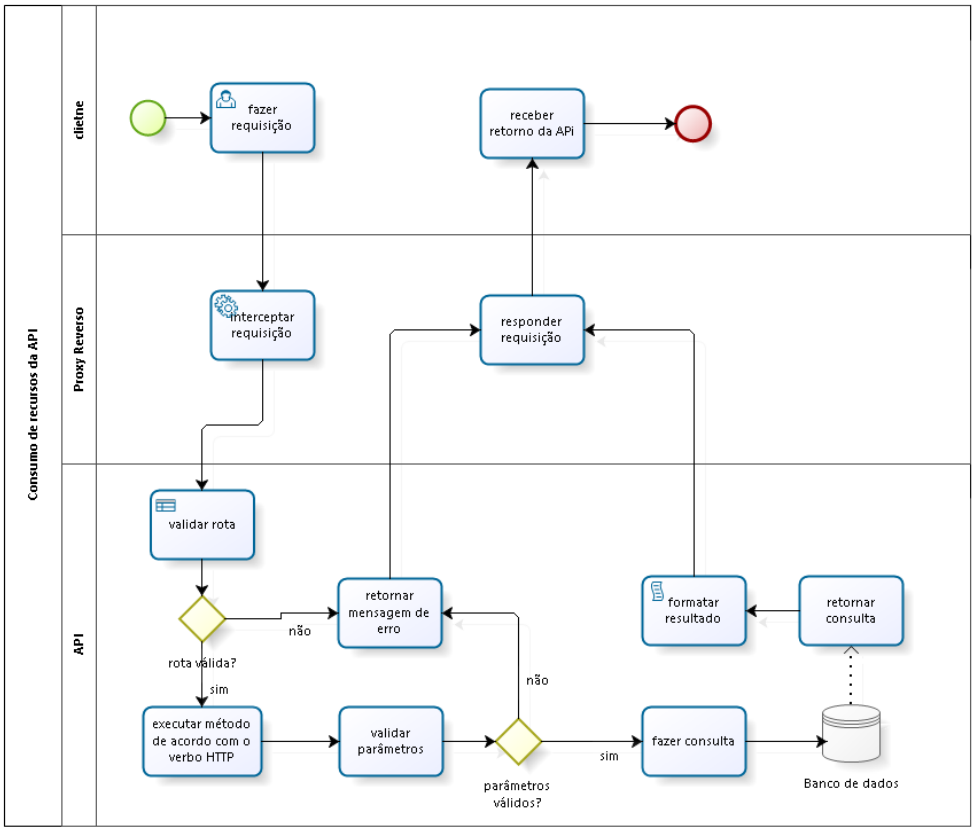
As funcionalidades serão implementadas em uma página web com design responsivo para que seja utilizada também em dispositivos móveis. :

- Criação de novos usuários através do formulário de cadastro na página inicial.
- Realização de login através do formulário de login na página inicial.
- Adição e categorização dos pontos de acessibilidade ainda não cadastrados utilizando o mapa apresentado na interface do usuário.
- Realização de *checkin* que permita a avaliação dos pontos marcados com uma escala de pontuação que vai de 1 a 5 e a escrita opcional de um comentário de texto.
- O sistema deve permitir que o usuário também sugira novos pontos e seus respectivos tipos para que os órgãos públicos e privados tenham acesso e tomem devidas providencias.
- Disponibilizar uma API pública para a coleta de dados e geração de estatísticas por parte de outros órgãos interessados.
- A API deve permitir a integração e estar pronta para integração e consumo através de dispositivos móveis.

3.6. Implementação da Aplicação

O processo de consumo dos recursos da API, independentemente do tipo de cliente, é realizado através da comunicação entre os componentes do sistema de acordo com o diagrama apresentado na Figura 31:

Figura 31 - Consumo de recursos da API



Fonte: O autor

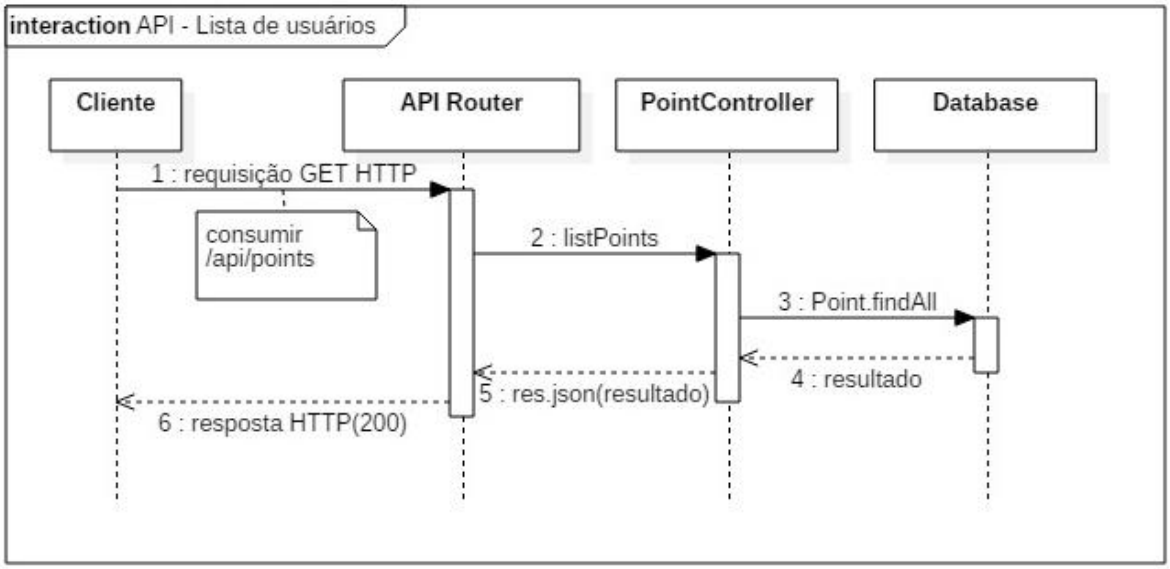
O consumo de recursos é realizado através de requisições HTTP realizadas pelo cliente à API, que possui *endpoints* específicos para cada ação e que atendem a métodos específicos de acordo com o verbo utilizado na requisição HTTP.

Caso a validação da rota, do verbo HTTP e em alguns casos os parâmetros de entrada seja positiva são invocados métodos ligados aos respectivos *endpoints* da API, caso contrário é enviado ao cliente uma mensagem de erro.

Internamente existem vários tipos de interações entre os componentes da API. Exemplos desses tipos de comunicação entre os componentes são representadas pelos diagramas de sequência nas figuras 32 e 33, que tem o propósito de exibir esses tipos de

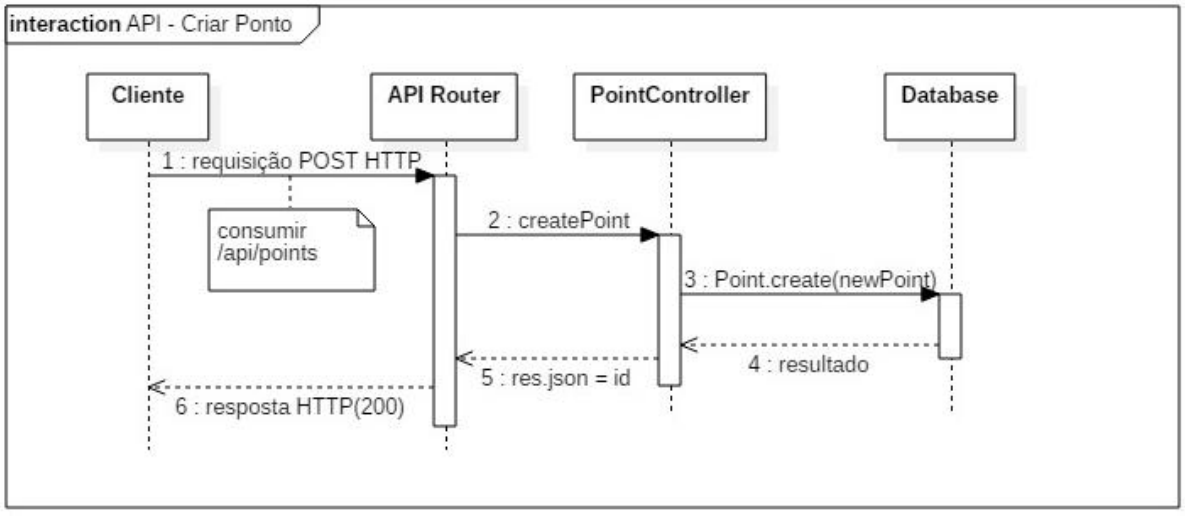
comportamento (Fowler, 2005):

Figura 32 - Diagrama de sequência que representa funcionalidade de listagem de Usuários



Fonte: O autor

Figura 33 - Diagrama de sequência da funcionalidade de criação de pontos



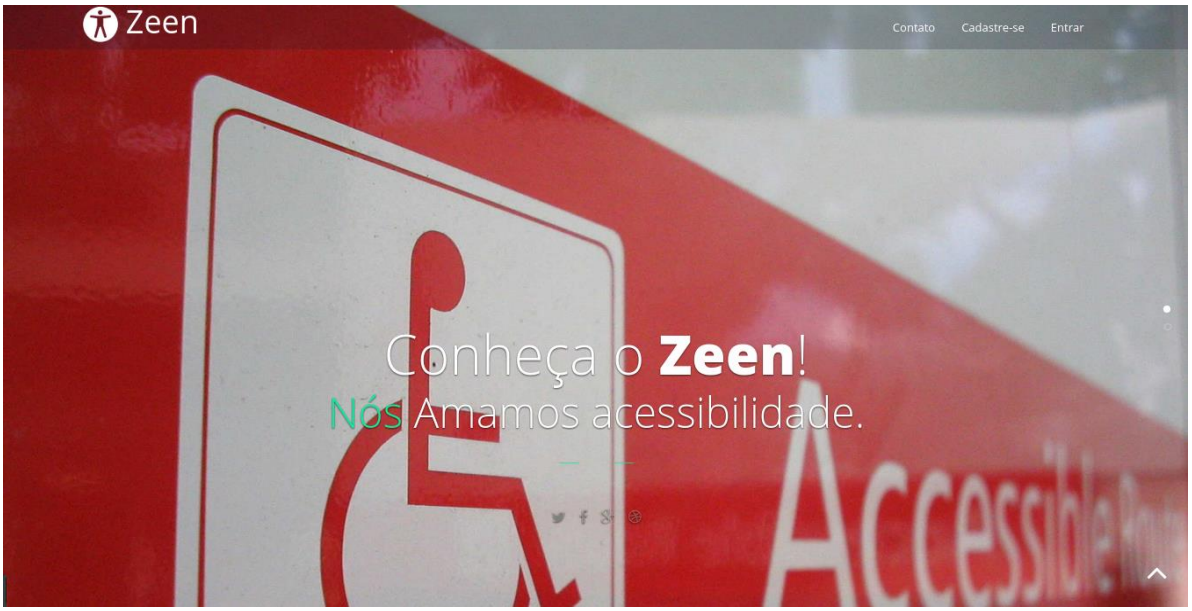
Fonte: O autor

Após a definição e apresentação do fluxo de funcionamento da API, a seguir veremos a descrição de cada uma das funcionalidades do protótipo.

3.6.1. Página principal, cadastro de usuários e login

A página principal é composta de uma página HTML com a apresentação do sistema é mostrada na Figura 34.

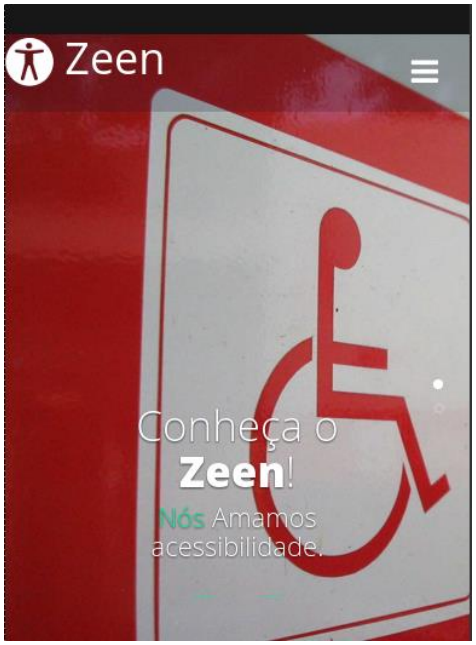
Figura 34 - Página Inicial do sistema



Fonte: O autor

Esta página também possui uma versão responsiva apresentada na Figura 35 que possibilita a navegação através de dispositivos móveis como *smartphones* e *tablets*.

Figura 35 - Versão responsiva da página inicial do sistema



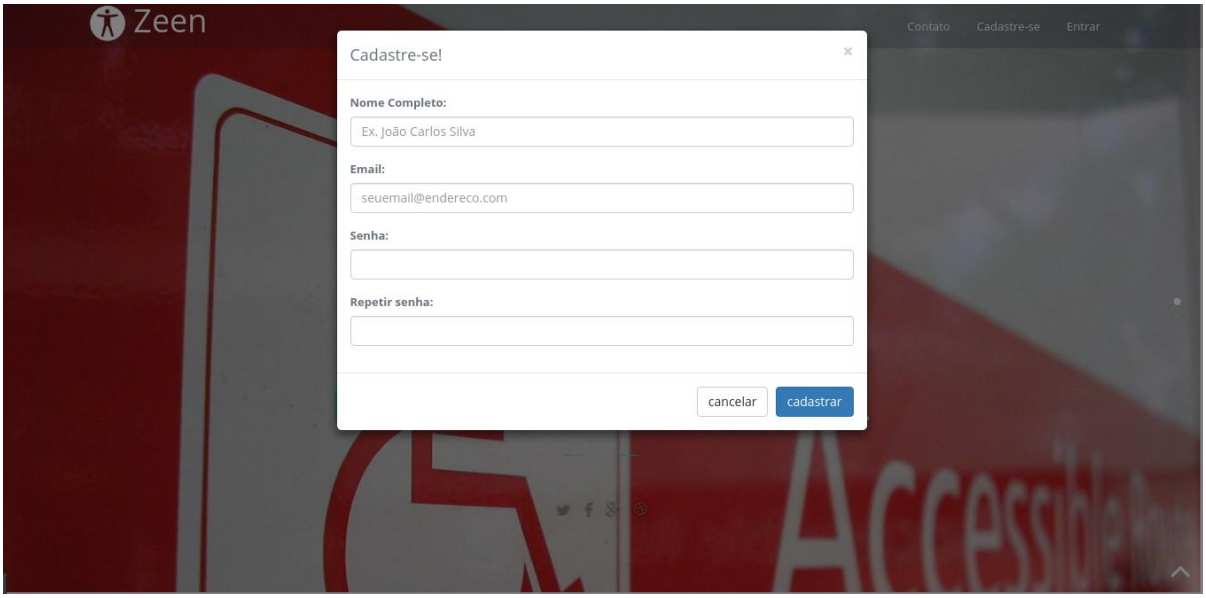
Fonte: O autor

O cadastro de um novo usuário é realizado através de um formulário HTML na página inicial visto na Figura 36. Ao clicar no botão “cadastrar”, primeiramente o sistema realiza a validação dos dados via cliente, retornando uma mensagem de erro no campo em que houver erros de validação.

Após a validação inicial o formulário envia os dados via HTTP POST utilizando AJAX, onde a API recebe os dados do usuário e faz a validação dos dados via servidor.

Uma vez que estes dados sejam validados, a API faz a inserção do novo usuário no banco e retorna uma mensagem de sucesso para a interface de cadastro. Caso haja erro na validação via servidor, a API retorna uma mensagem de erro no formato JSON que é apresentada no formulário de cadastro.

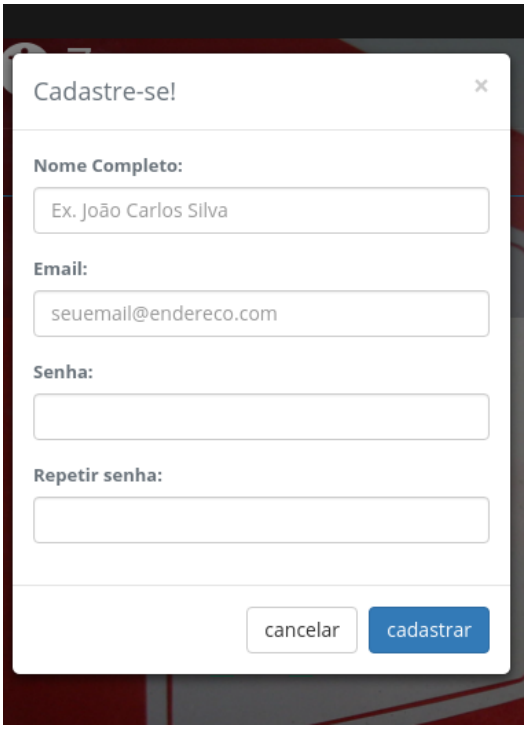
Figura 36 - Formulário de cadastro



Fonte: O autor

Para permitir o cadastro utilizando dispositivos móveis, há uma versão responsiva do cadastro de usuários visualizada na Figura 37.

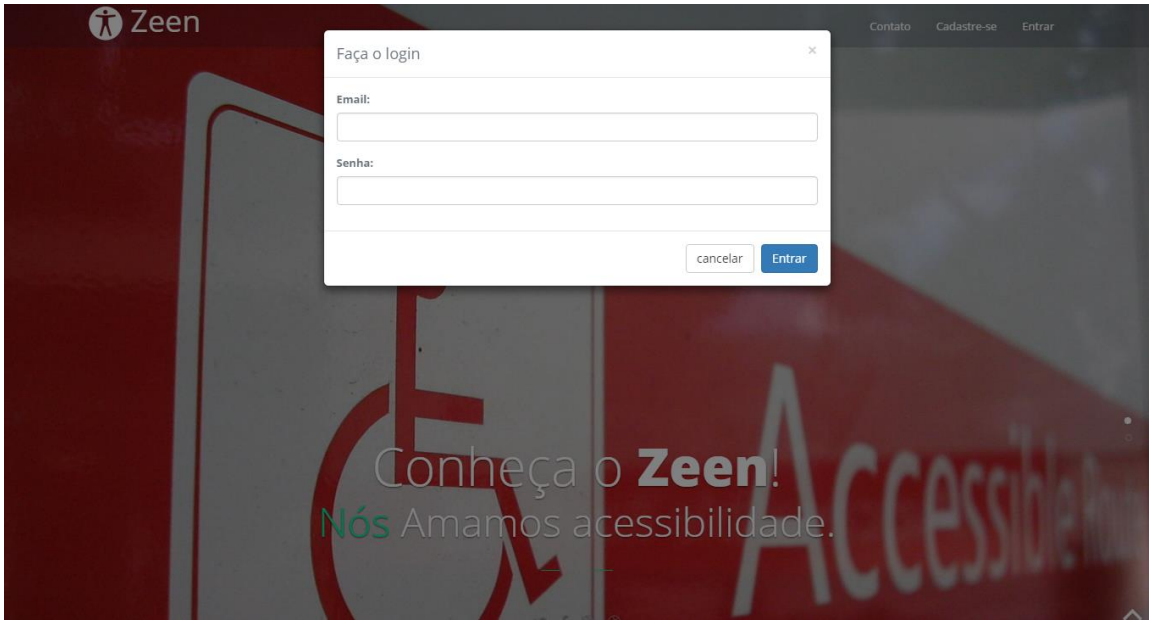
Figura 37 - Versão responsiva do formulário de cadastro



Fonte: O autor

O *login* para acesso ao sistema é feito através do formulário de *login* mostrado na página inicial através de uma janela modal apresentada na Figura 38. Ao preencher os dados do formulário e clicar no botão “entrar” o formulário dispara via AJAX uma requisição HTTP POST para a API que busca os dados do usuário armazenados no banco de dados e compara com os dados digitados no formulário. Em caso positivo o sistema armazena um id de sessão para o usuário e redireciona o mesmo para o *dashboard* da aplicação, em caso negativo é apresentada a mensagem de erro ao usuário.

Figura 38 - Formulário de login



Fonte: O autor

Assim como em outros formulários do sistema, o formulário de login também possui sua versão responsiva conforme mostra a Figura 39:

Figura 39 - Versão responsiva do formulário de login

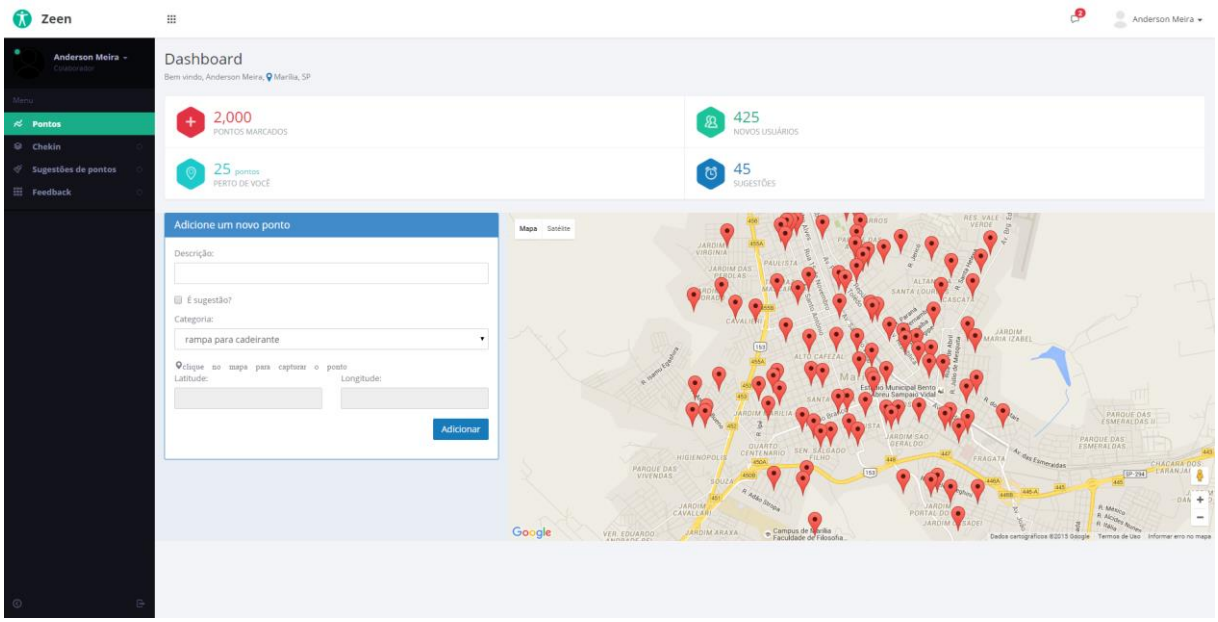
Fonte: O autor

3.6.2. Adição, categorização e sugestão de pontos

A adição de pontos permite que o usuário adicione pontos de acessibilidade existentes no local, interagindo com a interface *dashboard* do protótipo visto na Figura 40, de forma que seja possível adicionar um novo ponto no mapa para que outros usuários de forma colaborativa possam usufruir dessa informação. No formulário de adição do ponto é solicitada a descrição do ponto, uma lista de categorias de pontos, onde o usuário poderá classificar o tipo de ponto de acessibilidade que deseja cadastrar e as coordenadas de latitude e longitude são capturadas através do clique no local especificado no mapa.

A comunicação do formulário da página HTML com a API se dá por meio de requisições e respostas realizadas por AJAX.

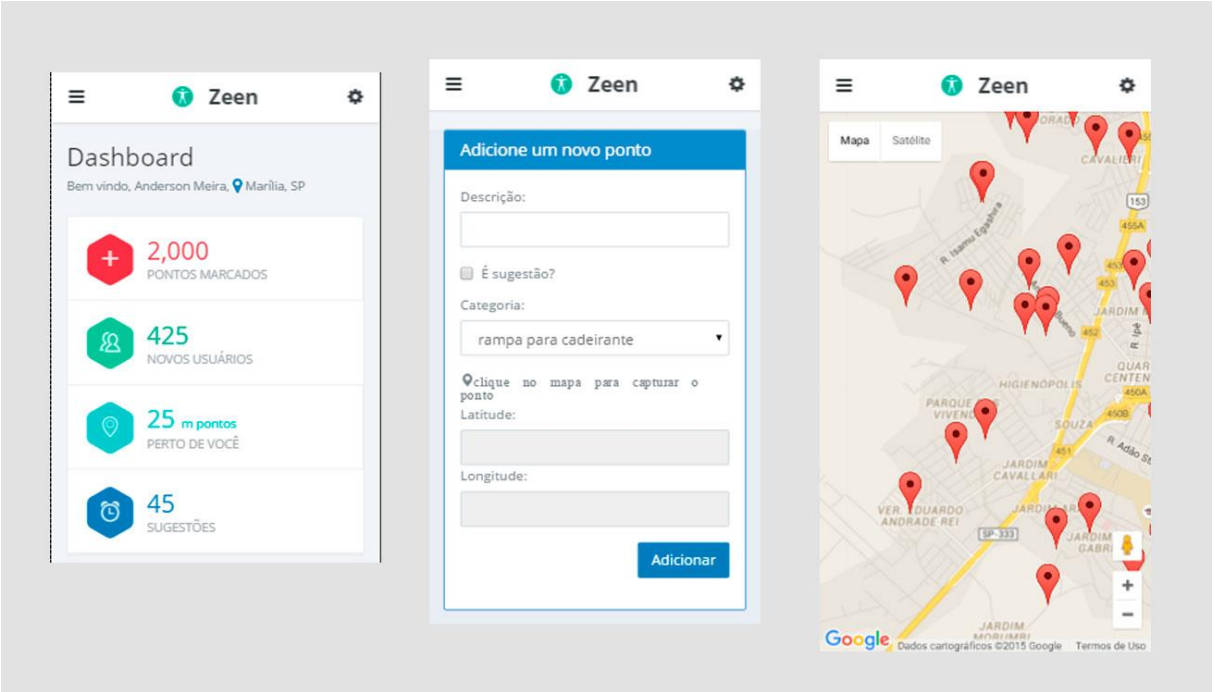
Figura 40 - Dashboard do sistema



Fonte: O autor

A versão responsiva do *dashboard*, dividido em três partes é mostrado na Figura 41:

Figura 41- Versão responsiva do *dashboard*

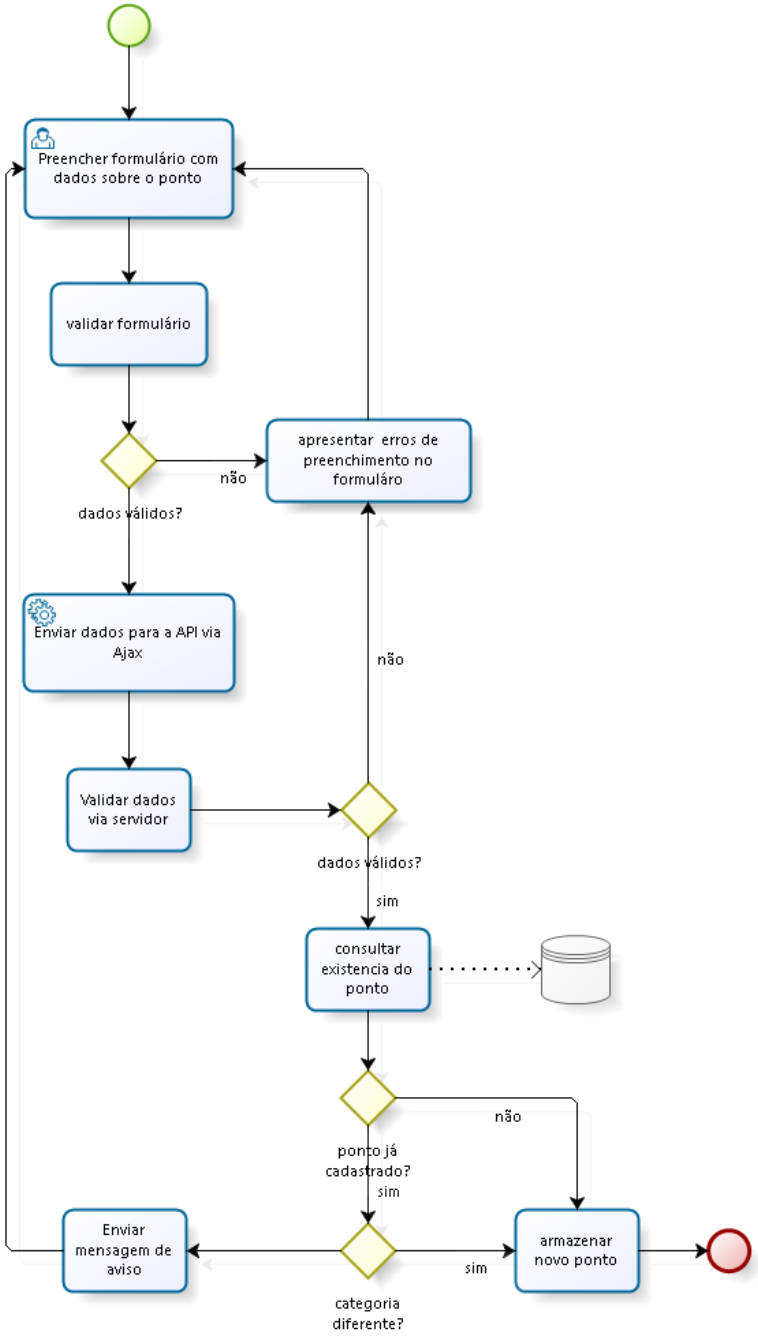


Fonte: O autor

Caso o ponto seja uma sugestão, o usuário fará a marcação da opção “sugestão” presente no formulário para que o ponto seja marcado como tal.

O diagrama de atividade (Fowler, 2005) mostrado na Figura 42 permite entender o fluxo desta funcionalidade:

Figura 42 - Diagrama de atividade da adição de pontos



Fonte: O autor

3.6.3. Fazer checkin

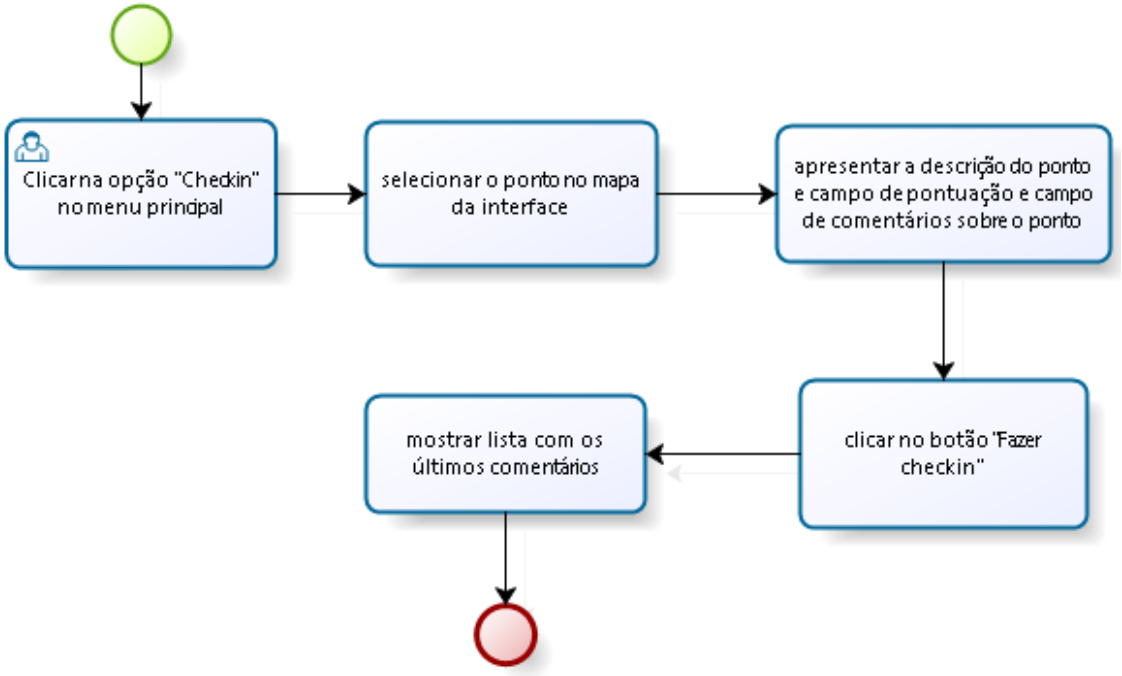
A funcionalidade de *checkin* permite que o usuário faça a marcação de um ponto onde ele está presente e possa avaliar as condições do ponto em questão, atribuindo uma nota e se caso opte, deixe um comentário.

A localização do usuário se dá pela utilização da API de geolocalização presente no HTML5 que traz de forma aproximada a localização do usuário que utiliza o sistema. Após a captura do posicionamento do ponto, é feita uma requisição à API do sistema que recupera os dados pertencentes ao ponto, como descrição, nota média de avaliação e comentários dos usuários.

No formulário presente nesta interface, o usuário informa a nota que vai de um a cinco apresentada numa lista de notas e o comentário opcional e clica no botão "Fazer *checkin*". Após o clique no botão é disparada uma requisição HTTP do tipo POST para o recurso "*/api/checkin*" que registra o *checkin* do usuário no banco de dados, faz uma nova consulta para trazer os últimos comentários e os apresenta em forma de lista na interface.

Este fluxo é apresentado em um contexto de alto nível no diagrama de atividade da Figura 43:

Figura 43 - Diagrama de atividade da funcionalidade de *checkin*



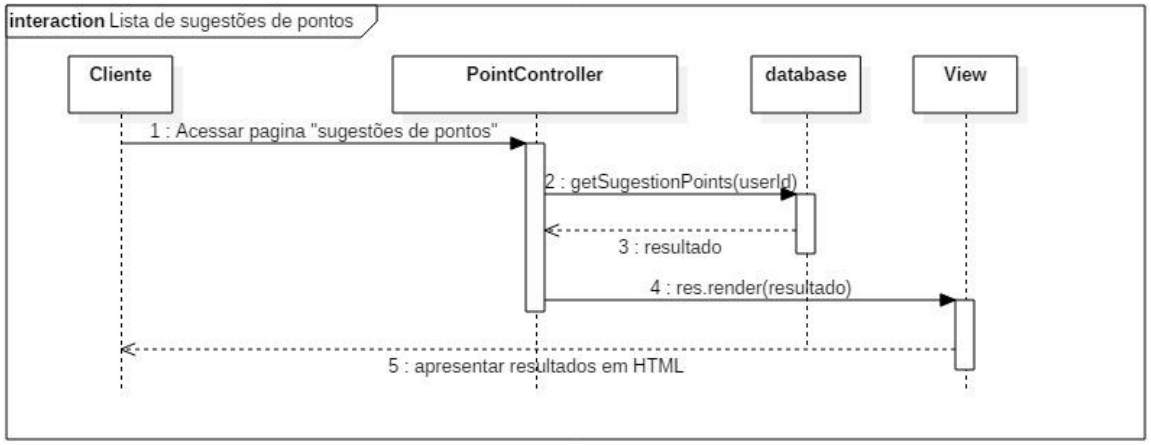
Fonte: O autor

3.6.4. Sugestão de pontos

A sugestão de pontos permite que o usuário visualize quais pontos foram sugeridos por ele e dentre eles quais já foram tomadas algumas providencias por parte dos responsáveis pelo local.

A lista é apresentada através de uma lista na página HTML com os dados dos pontos sugeridos. Esta solicitação é feita através do método *getSugestionPoints()* do *controller* Point*Controller* que consulta no banco de dados filtrando os pontos pela sugestão feita pelo usuário e os retorna no formato JSON que posteriormente é processado na página utilizando JavaScript no navegador cliente. A Figura 44 exemplifica o fluxo através do diagrama de sequência:

Figura 44 - Diagrama de sequência - Lista de sugestões de pontos



Fonte: O autor

3.7. Desafios e problemas no desenvolvimento do protótipo

A implementação do protótipo apresentou alguns desafios principalmente em relação às ferramentas e novos conceitos que foram necessários para a sua execução.

A programação orientada a eventos com Node.js faz com que o desenvolvedor adquira um novo meio de pensar a respeito do código a ser escrito, pois em vários momentos a forma de execução assíncrona pode gerar valores diferentes, ou até falhas se a visão do desenvolvedor estiver focada na sequência de execução síncrona de código.

A utilização de um módulo para estender a funcionalidade de geolocalização no banco de dados PostgreSQL, apesar de funcionar bem depois de configurado, exige que alguns passos

sejam feitos antes da utilização e pode apresentar problemas dependendo do tipo de pacote de instalação utilizado e seu sistema operacional. O ideal é que tipos relacionados a geolocalização fizessem parte do núcleo do PostgreSQL assim como em bancos de dados não relacionais.

A ORM Sequelize utilizada para a manipulação de dados ainda é pobre em recursos se comparada a outras ferramentas do mercado, e sua documentação é muito defasada.

3.8. Considerações finais

Neste capítulo foi apresentado o cenário no qual se faz necessário utilizar a arquitetura proposta afim de atingir um grande número de usuários. Foram apresentadas também as principais funcionalidades do protótipo, tecnologias utilizadas e os principais fluxos nos quais as funcionalidades realizaram suas atividades.

No próximo capitulo serão apresentados os resultados do trabalho desenvolvido, com resultados sobre o nível de escalabilidade da arquitetura em ambientes diferentes.

4. RESULTADOS

O propósito deste trabalho é medir a eficiência da arquitetura proposta utilizando técnicas que permitam avaliar o consumo de recursos da API através de uma simulação de uso de alta concorrência.

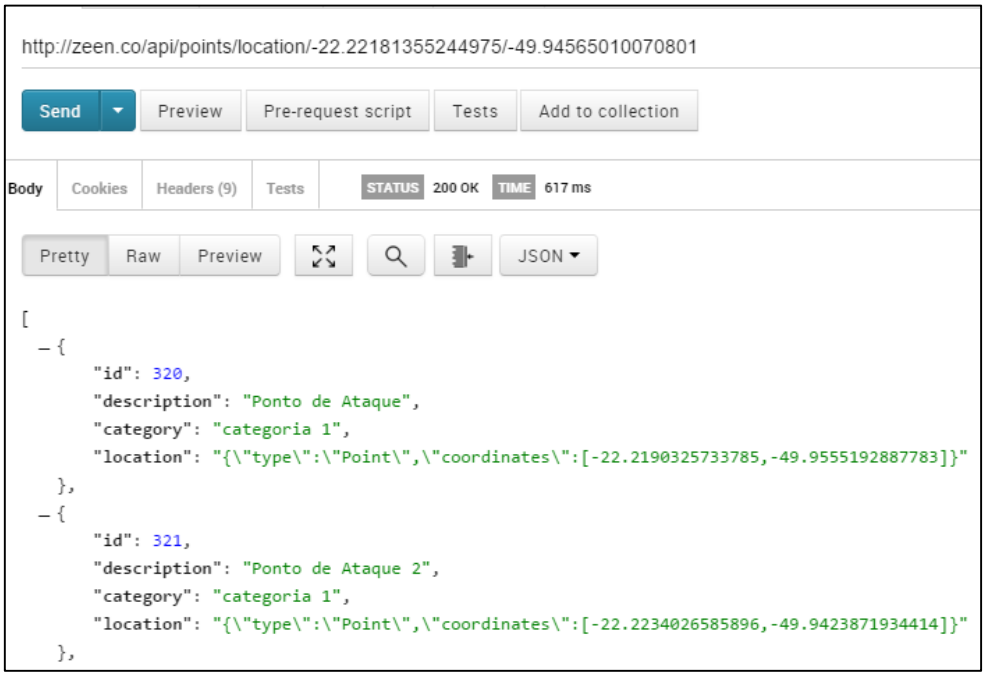
Para tal, será utilizada a ferramenta Siege (Siege, 2015), que permitirá a realização de testes de carga na arquitetura da seguinte forma:

Em um servidor Debian versão 7 será executado o seguinte comando:

```
siege -b -d1 -t5M -c50 http://zeen.co/api/points/location/-22.22181355244975/-49.94565010070801
```

Com este comando o Siege trabalhará em modo *benchmark* durante 5 minutos fazendo acesso concorrente simulando, no exemplo acima 50 usuários, com um atraso aleatório entre zero e 1 segundo a cada solicitação. Na URL do recurso acessado pelo Siege é passado como parâmetro a latitude e a longitude de um determinado ponto que retorna no formato JSON 100 resultados dos pontos próximos a este conforme o modelo simplificado mostrado na Figura 45:

Figura 45 - Exemplo de resposta da API no cenário de testes



Fonte: O autor

Para que o comportamento do VPS, *Virtual Private Server*, da arquitetura seja testado,

a configuração do servidor que hospeda aplicação sofrerá alterações na quantidade de núcleos, memória RAM e espaço em disco simulando uma escalabilidade vertical conforme a tabela 5:

Tabela 5 - Configuração de Hardware do VPS

CPU	Memória RAM	Espaço em Disco
1 Core	512MB	20GB (SSD)
2 Cores	2GB	40GB (SSD)
4 Cores	8GB	80GB (SSD)

Fonte: (DigitalOcean, 2015)

Na mudança de configuração do VPS, será alterado o número de *workers* do Nginx, onde o número de workers deverá ser igual ao número de núcleos da CPU do VPS (Nginx, Beginners Guide, 2015).

Com relação ao Node.js, a utilização do PM2 como gerenciador de processos será no padrão f*ork mode* onde é instanciado somente um processo do Node.js. A partir da configuração de 2 núcleos serão realizados testes tanto em *fork mode* quanto em *cluster mode*, sendo este segundo um recurso do próprio PM2 para instanciar a aplicação em vários núcleos disponíveis no VPS (PM2, 2015).

A tabela 6 indica todos os cenários de teste no qual o sistema será submetido:

Tabela 6 - Cenários de Teste

Cenário	Nº de núcleos	Nº de Clientes	Descrição
A	1	50	• 1 Worker Process Nginx • Node.js fork mode
B	1	150	• 1 Worker Process Nginx • Node.js fork mode
C	1	350	• 1 Worker Process Nginx • Node.js fork mode
D	2	50	• 2 Worker Process Nginx • Node.js fork mode
E	2	150	• 2 Worker Process Nginx • Node.js fork mode
F	2	350	• 2 Worker Process Nginx • Node.js fork mode
G	2	50	• 2 Worker Process Nginx • Node.js cluster mode
H	2	150	• 2 Worker Process Nginx • Node.js cluster mode
I	2	350	• 2 Worker Process Nginx • Node.js cluster mode
J	4	50	• 4 Worker Process Nginx • Node.js fork mode
K	4	150	• 4 Worker Process Nginx • Node.js fork mode
L	4	350	• 4 Worker Process Nginx • Node.js fork mode
M	4	50	• 4 Worker Process Nginx • Node.js cluster mode
N	4	150	• 4 Worker Process Nginx • Node.js cluster mode
O	4	35	• 4 Worker Process Nginx • Node.js cluster mode

Fonte: O autor

Tanto o sistema operacional quanto o banco de dados não foram configurados para que se obtenha algum tipo de otimização, mantendo as configurações padrões presentes na instalação dos mesmos.

Outros dados de consumo de CPU e memória da aplicação foram colhidos através da utilização da ferramenta de monitoramento Zabbix (Zabbix, 2015).

4.1. Métricas para se medir a escalabilidade de uma arquitetura

Sobre a escalabilidade da arquitetura Fowler (2006, p. 29) levanta alguns pontos que devem ser considerados em uma arquitetura corporativa:

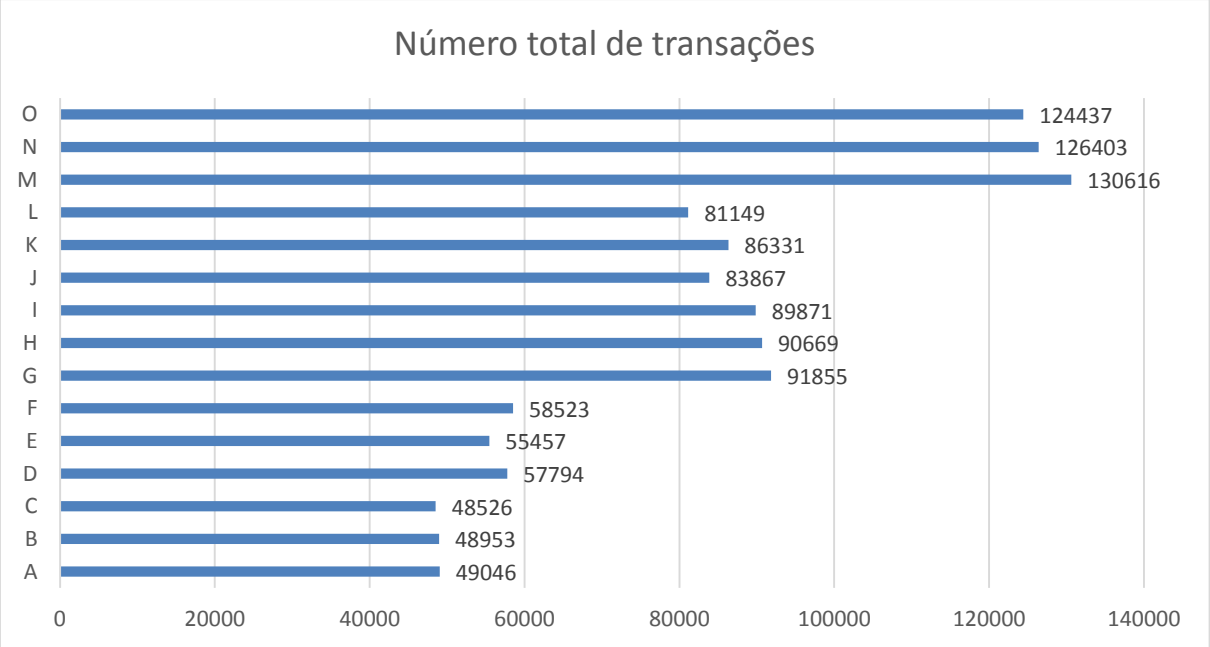
- ***Tempo de resposta***: que é a quantidade de tempo que o sistema leva para processar uma solicitação externa, por exemplo uma ação na interface com o usuário ou uma chamada de API do servidor.
- ***Agilidade de resposta***: que mede o quão rapidamente o sistema reconhece uma solicitação em oposição ao tempo que leva para processá-la. A agilidade de resposta influencia no nível de frustração do usuário caso o sistema demore a responder a sua solicitação.
- ***Latência***: é o tempo mínimo requerido para obter qualquer forma de resposta, mesmo se o trabalho a ser feito for inexistente. Bastante presente em sistemas remotos, é um ponto a se considerar quando se faz uma quantidade grande de chamadas remotas;
- ***Throughput***: é a quantidade de ações que podem ser feitas em uma dada quantidade de tempo, por exemplo a contabilização do tempo gasto na cópia de um arquivo, que neste caso o *throughput* pode ser medido em bytes por segundo. Geralmente em aplicações corporativas é utilizada a medida de número de transações por segundo.
- ***Carga***: é a medida da pressão a que o sistema está submetido, que poderia ser medida pelo número de usuários a ele conectados em um determinado instante de tempo. A carga é geralmente um contexto para alguma outra medida, como o tempo de resposta;
- ***Sensibilidade de carga***: é como o tempo de resposta varia de acordo com a carga, por exemplo considerando um cenário onde há dois sistemas, A e B, no sistema A o tempo de resposta é de 0,5 segundos para um número de usuários entre 10 e 20 usuários, e o sistema B tenha um tempo de resposta de 0,2 segundo para 10 usuários que aumenta para 2 segundos com 20 usuários. Neste cenário o sistema A tem uma sensibilidade de carga menor do que o sistema B. Pode ser usado o termo degradação para este caso, onde o sistema B degrada mais do que o sistema A.

- ***Eficiência***: é o desempenho dividido pelos recursos. Um sistema que obtenha 30 transações por segundo com duas CPUs é mais eficiente que um que obtenha 40 transações por segundo com quatro CPUs idênticas;
- ***Capacidade***: é a indicação do máximo *throughput* efetivo ou máxima carga efetiva, podendo ser um número máximo absoluto ou um ponto a partir do qual o desempenho caia abaixo do limite aceitável
- ***Escalabilidade***: medida de como o acréscimo de recursos, geralmente de hardware, afeta o desempenho da aplicação. Um sistema escalável é aquele que permite adicionar hardware e obter uma melhora de desempenho proporcional, por exemplo, dobrar o número de servidores disponíveis para dobrar o *throughput*. A escalabilidade pode ser dividida entre vertical, que significa adicionar mais poder a um único servidor (acrescentar memória) e a horizontal, que significa adicionar mais servidores.

4.2. Avaliação dos Testes

Os resultados obtidos nos testes foram organizados nos seguintes gráficos:

Figura 46 – Gráfico I - Número total de transações



No Gráfico I da Figura 46 é possível verificar que em cada uma das variações do

número de núcleos do cenário há uma pequena queda no número de transações contabilizadas conforme os o número de requisições por segundo foi aumentado. De modo geral o aumento dos núcleos permitiu que mais requisições fossem atendidas pela API.

Figura 47 - Gráfico II - Número total de transações com variação entre fork e cluster mode para 2 núcleos

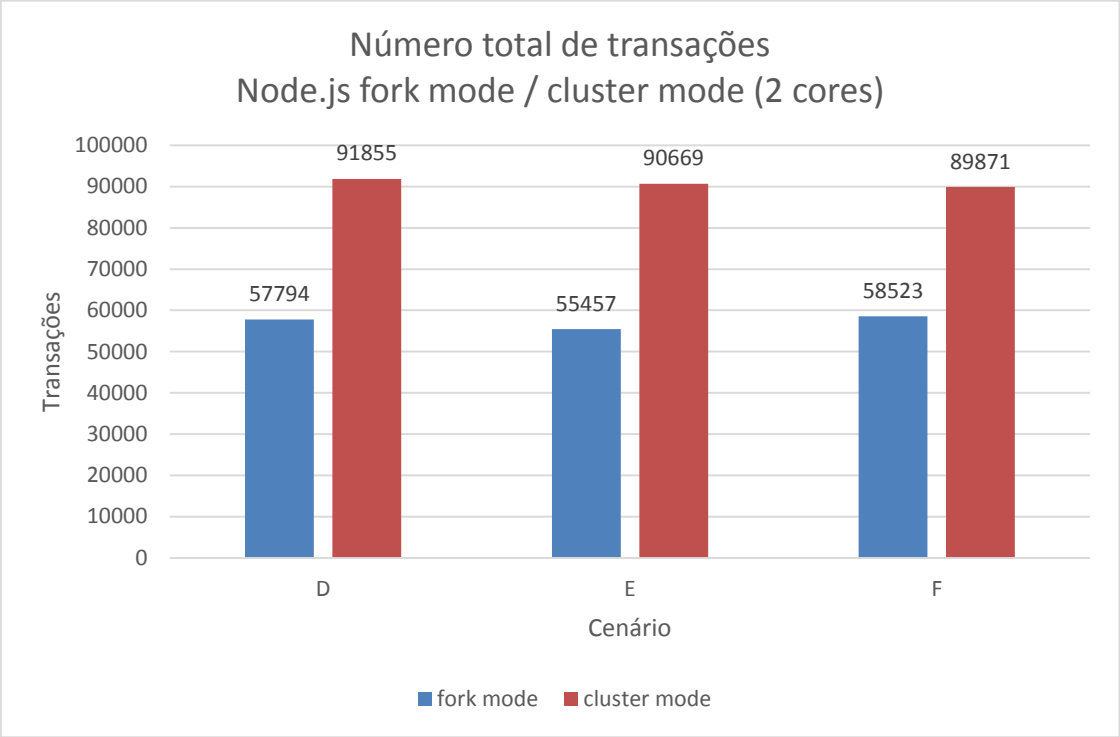
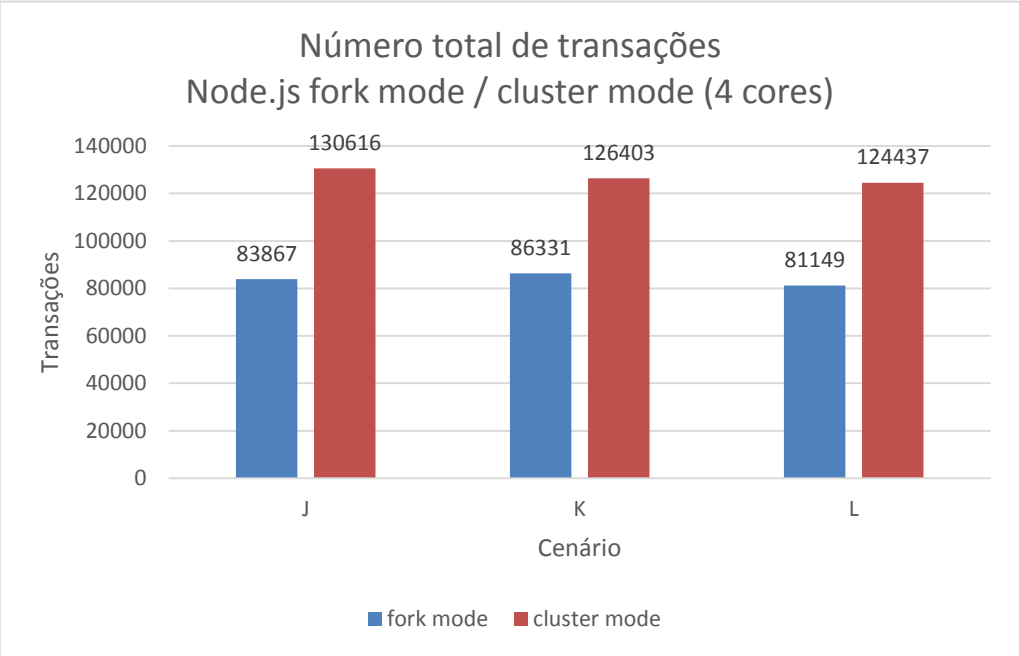
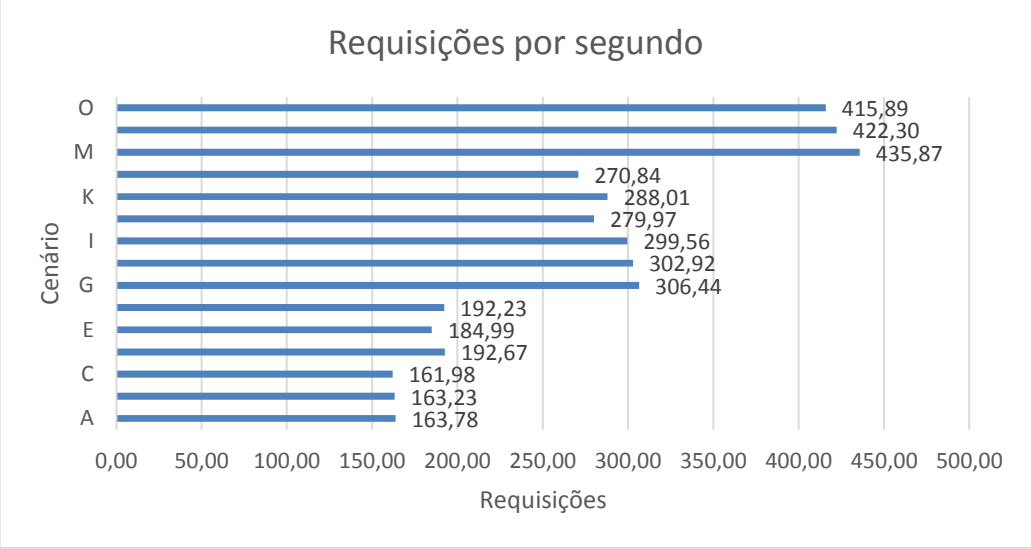


Figura 48 - Gráfico III - Número total de transações com variação entre *fork* e *cluster mode* para 4 núcleos



Nos gráficos II e III das figuras 47 e 48 são apresentados os resultados da variação entre a utilização da aplicação Node.js instanciada em *fork mode* e *cluster mode* nos cenários de dois e quatro núcleos. Ao utilizar o *cluster mode* a aplicação teve um aumento entre 55,4% 58,9% em média no número de requisições totais atendidas.

Figura 49 - Gráfico IV - Número de requisições por segundo



O gráfico IV da figura 49 mostra o número de requisições por segundo atendidas em todos os cenários de teste. Neste gráfico é possível verificar que o aumento no número de requisições do cenário mais simples e menos exigido "A" até o cenário "O" que é o mais robusto e mais exigido é de 153,93%.

Figura 50 - Gráfico V - Número de requisições por segundo em *fork* e *cluster mode* para 2 núcleos

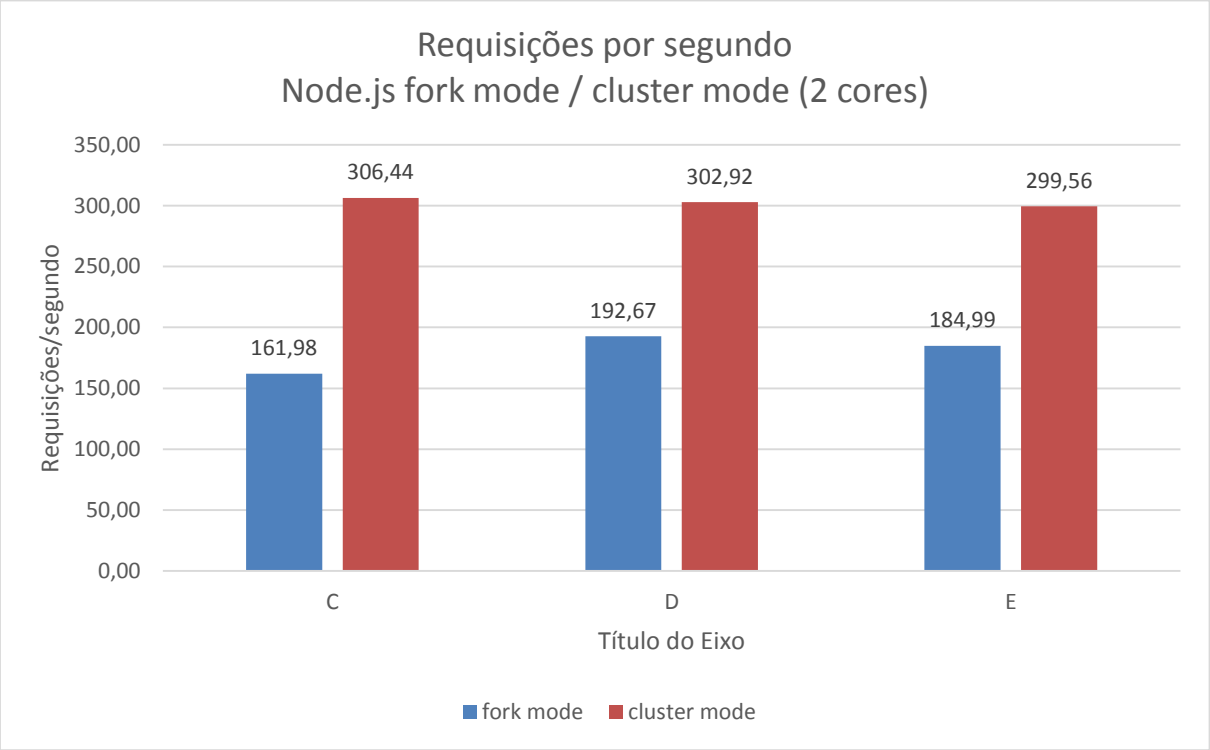
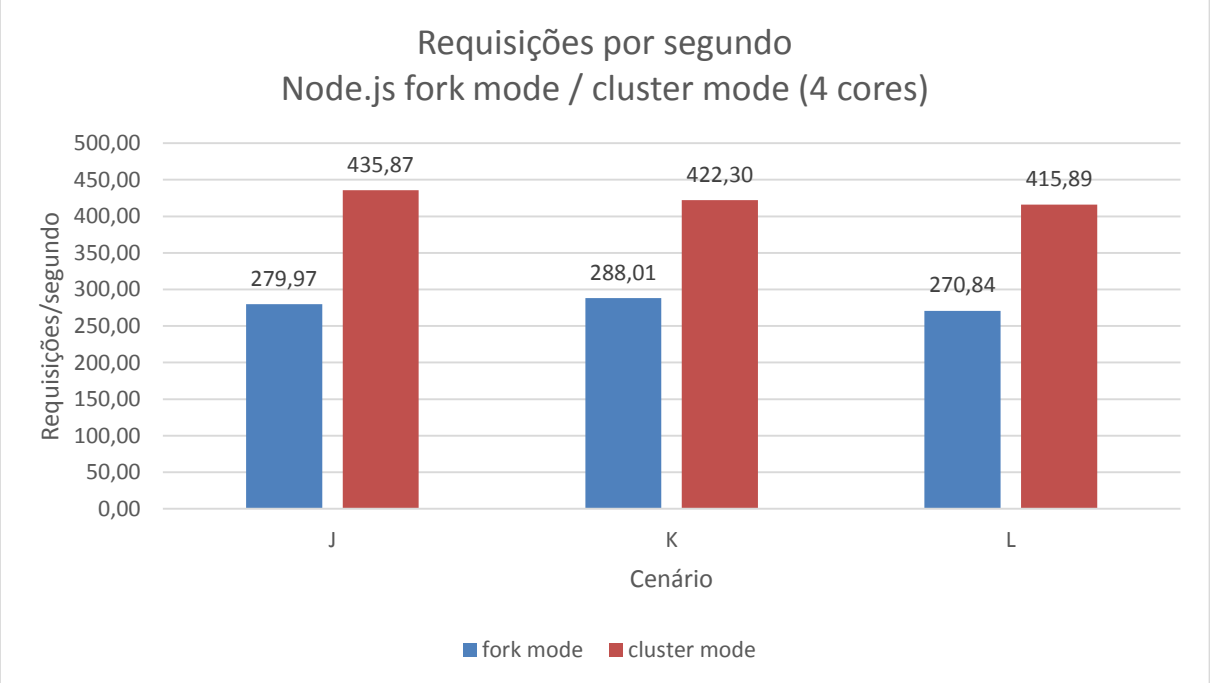
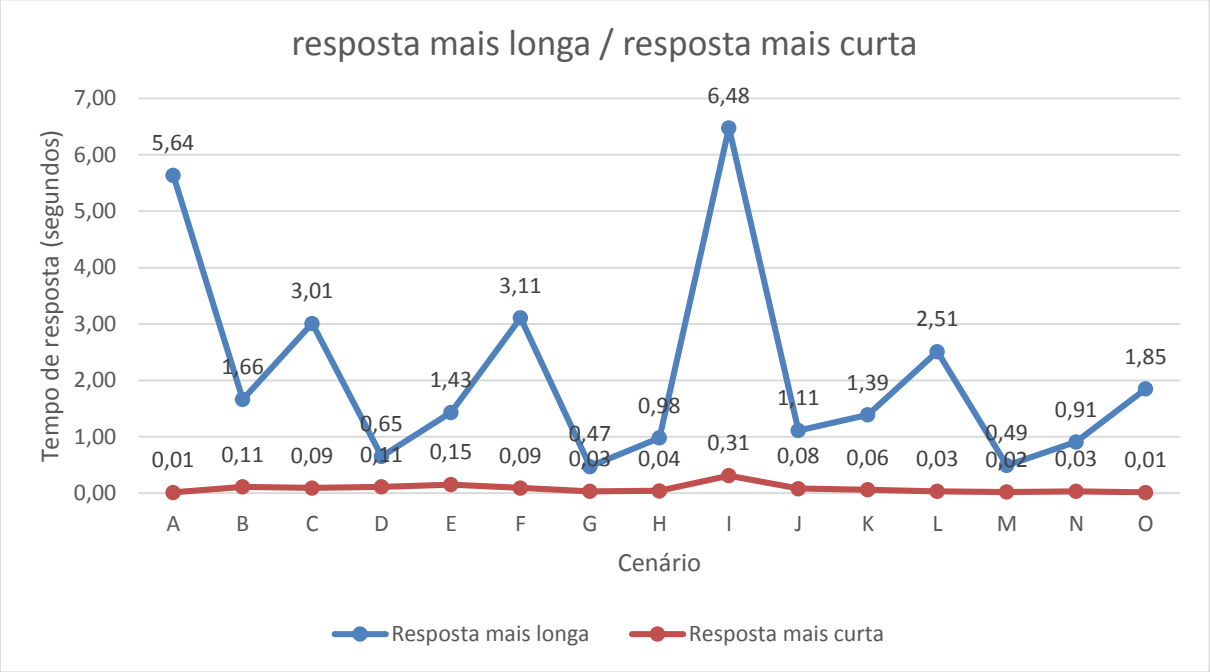


Figura 51 - Gráfico V - Número de requisições por segundo em fork e cluster mode para 4 núcleos



Nos gráficos IV e V das figuras 50 e 51 são apresentadas as variações tanto na instanciação da aplicação em *fork mode* e *cluster mode* quanto a variação dos núcleos entre dois e quatro. No cenário de dois núcleos a mudança para *cluster mode* trouxe um aumento médio de 69,44% no número de requisições por segundo atendidas, enquanto que no cenário de quatro núcleos o aumento médio foi de 51,95%.

Figura 52 - Gráfico VI - Tempo de Resposta

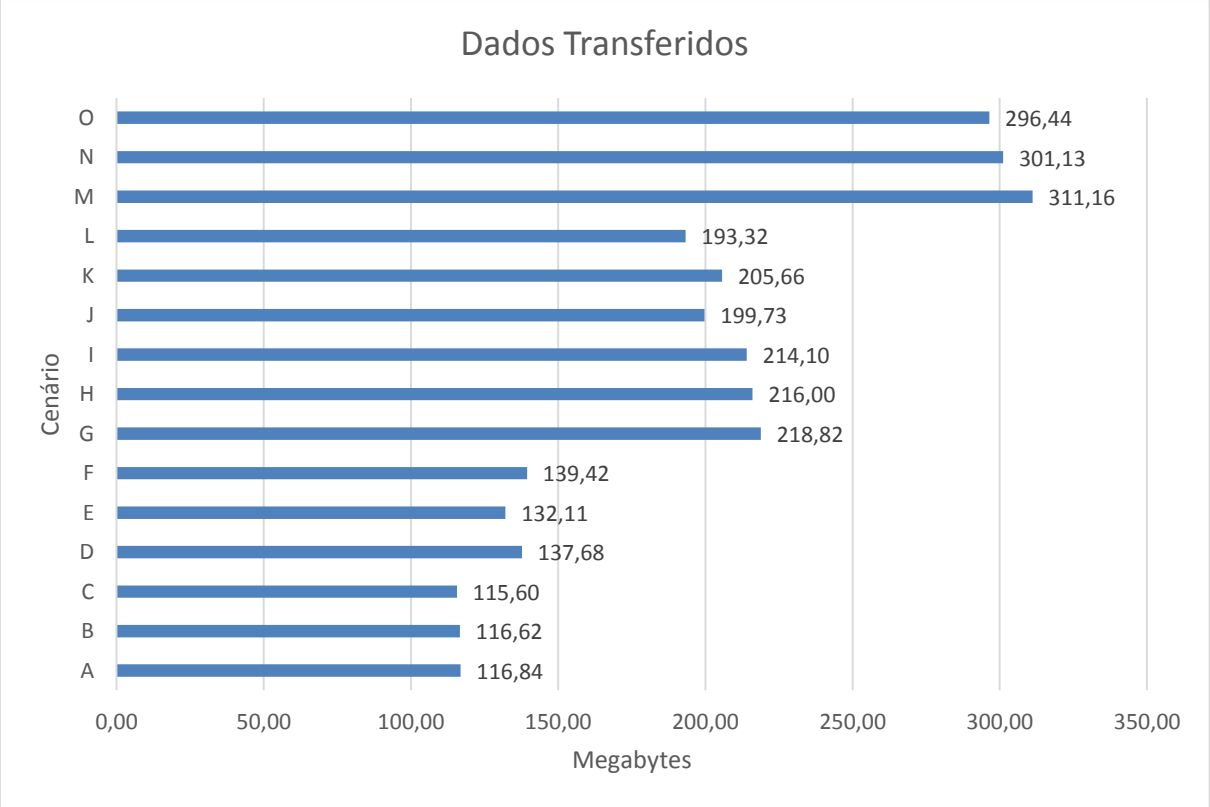


No gráfico VI da figura 52 é apresentado o tempo de resposta das requisições em todos os cenários, inclusive o menor e o maior tempo registrado nas requisições. Pode ser observado que em alguns momentos há um pico entre as respostas mais longas. De todos os testes o que apresentou a menor diferença entre o menor e o maior tempo foi do cenário M.

Figura 53 - Gráfico VII - Taxa de erros

No gráfico VII da figura 53 pode ser verificado que em todos os testes a API teve um índice zero de erros, o que quer dizer também que em todos os cenários houve 100% de disponibilidade da API.

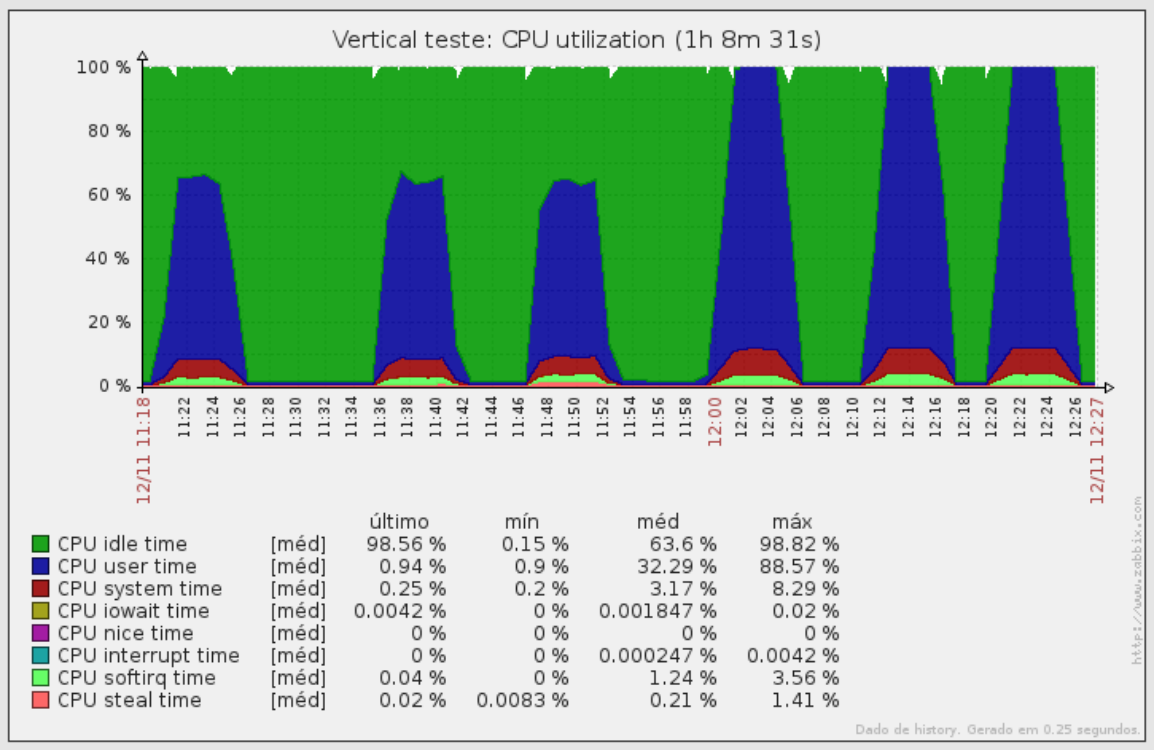
Figura 54 - Gráfico VIII - Dados transferídos



O Gráfico VIII da figura 54 mostra a quantidade de dados transferidos durante cada um dos testes. Estes dados são importantes para se calcular o consumo de banda que a aplicação pode sofrer durante um determinado número de requisições.

Os gráficos a seguir mostram como o hardware se comportou durantes os testes realizados nos cenários com quatro núcleos de processamento em um determinado tempo e variação entre *fork mode* e *cluster mode*. Todas estas informações foram colhidas da ferramenta de monitoramento Zabbix instalada em um outro servidor e seu agente instalado no servidor da aplicação.

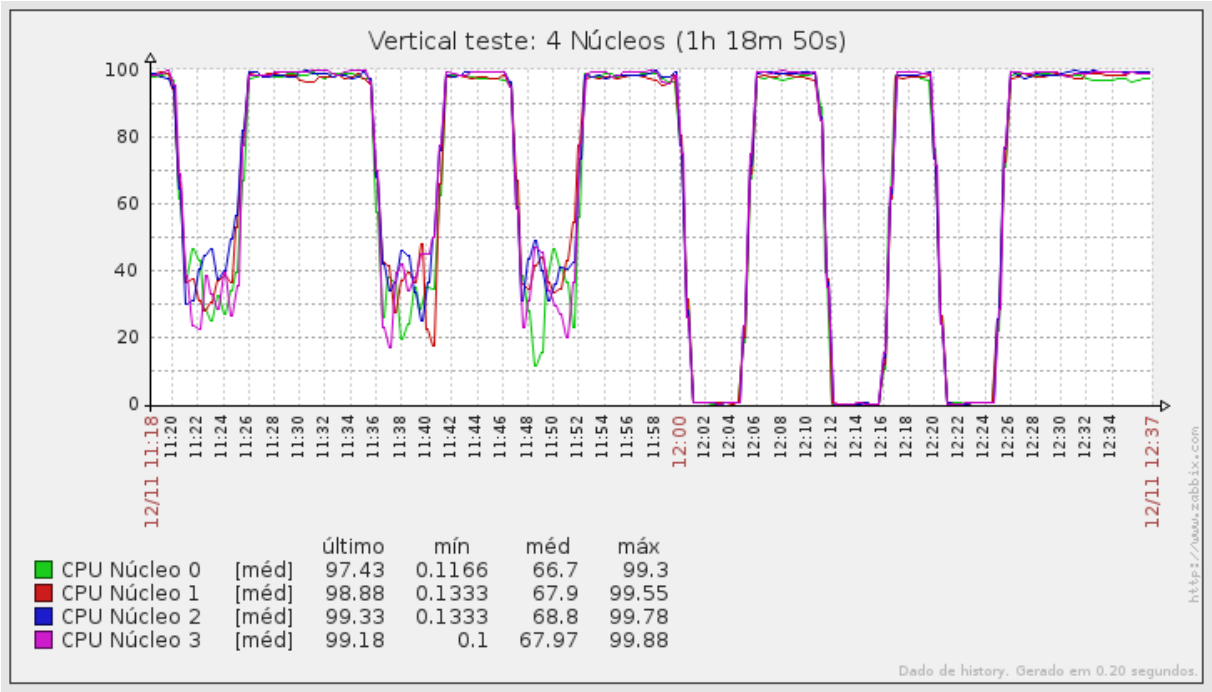
Figura 55 - Grafico IX - Utilização da CPU



Neste IV da figura 55 é apresentado a utilização geral da CPU sendo os três primeiros cenários “J”, “K” e “L” utilizando o *fork mode* e os cenários “M”, “N” e “O” utilizando o *cluster mode* como forma de instanciar a aplicação.

No *fork mode* é possível notar a distribuição dos recursos de CPU de forma que em alguns momentos a CPU fica livre para antender outras demandas, enquanto que no *cluster mode* a utilização da CPU é 100% eficiente.

Figura 56 - Gráfico XI - Disponibilidade da CPU - 4 Núcleos



No Gráfico XI da figura 57 é possível mais uma vez verificar a eficiência do uso dos núcleos da CPU no *cluster mode* visto que não há disponibilidade nos núcleos ao mesmo tempo que em outros gráficos são apresentados nível de disponibilidade em 100%, ou seja a arquitetura consegue atender toda a demanda do teste sem apresentar indisponibilidade da aplicação.

Figura 57 - Gráfico XII - Disponibilidade da memória

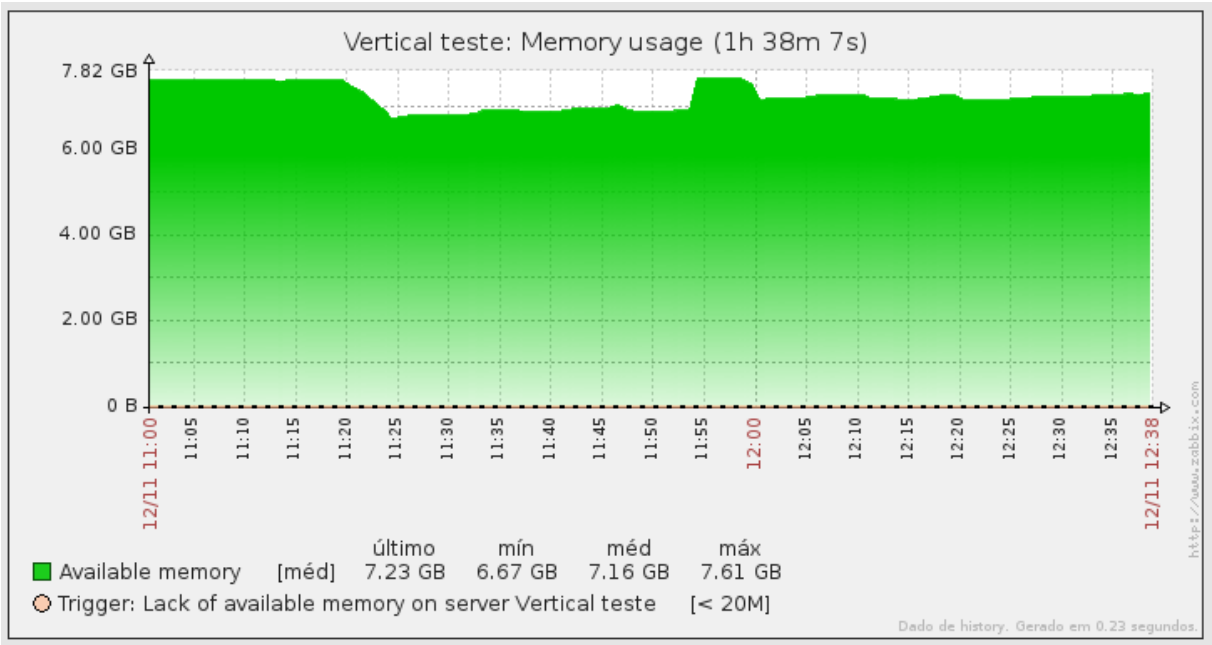
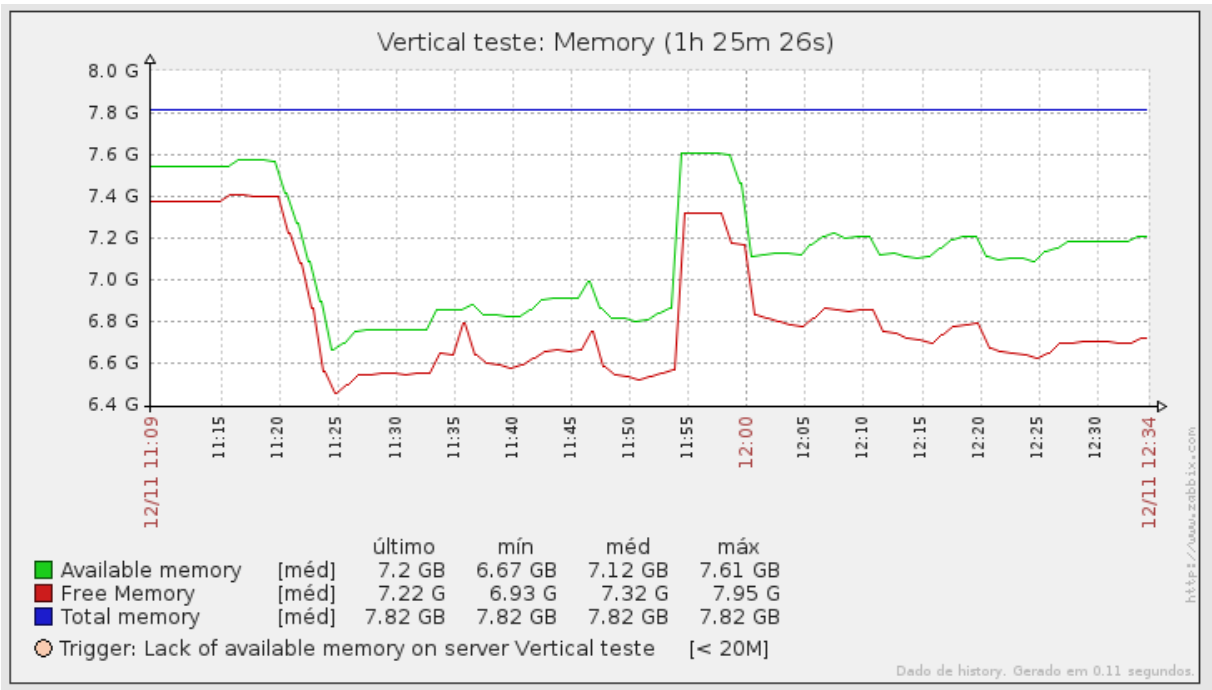
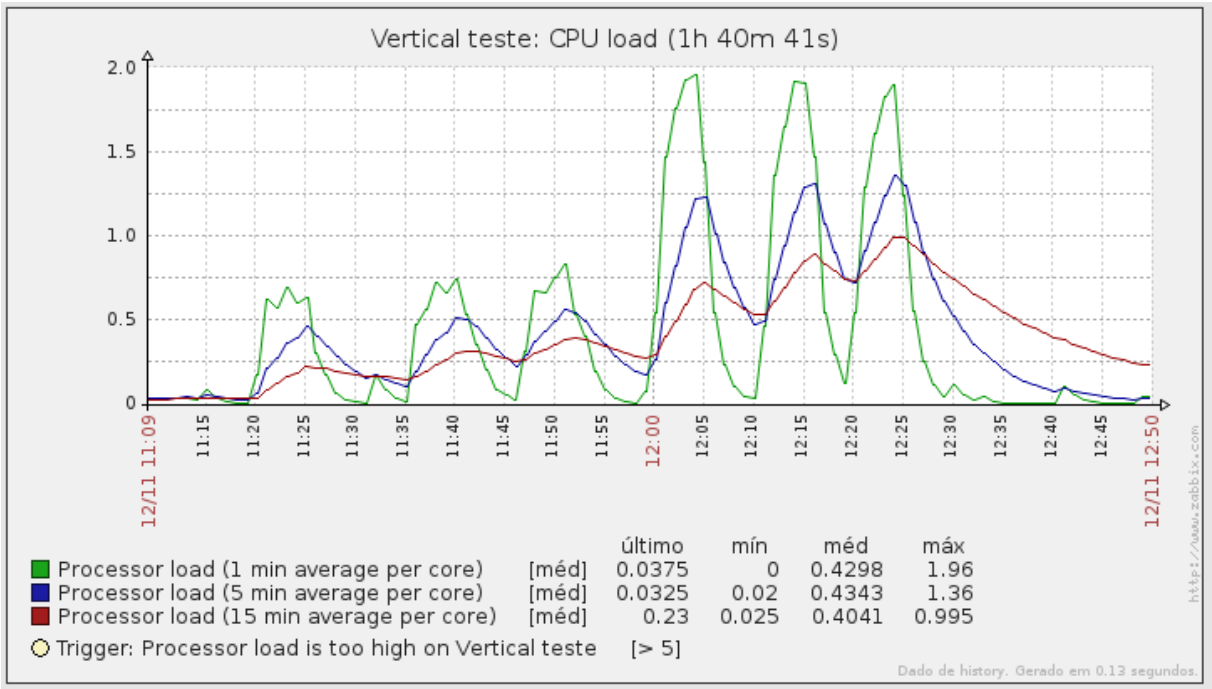


Figura 58 - Gráfico XIII - Consumo de memória



Os gráficos XII e XIII das figuras 58 e 59 mostram o consumo e disponibilidade da memória RAM disponível nos cenários com quatro núcleos. Estes resultados mostram um baixo consumo de memória pela aplicação devido à arquitetura orientada a eventos, que diferente da arquitetura monolítica orientada por *threads*, que não cria vários processos por requisição da aplicação.

Figura 59 - Grafico XIV - Load da CPU



Os resultados do gráfico XIV da figura 59 mostram que nos três primeiros testes onde foi utilizado o *fork mode* o consumo de recursos computacionais do sistema a saber, CPU, memória e disco, foram comprometidos em um pouco mais de 25% da sua capacidade, enquanto que no *cluster mode* o consumo dos recursos computacionais foi de quase 50%.

A análise destes dados junto aos resultados dos gráficos anteriores mostra que em nenhum momento houve sobrecarga dos recursos de máquina, o que mostra mais uma vez o uso eficiente destes recursos pela arquitetura proposta.

4.3. Contribuição do projeto

Os resultados dos testes realizados neste projeto contribuem para que arquitetos de sistemas possam considerar a arquitetura orientada a eventos em diversos projetos que necessitam de certa escalabilidade, utilização eficiente dos recursos de hardware disponíveis, diminuído o custo que outras arquiteturas podem agregar ao sistema.

4.4. Lições aprendidas

Há disponível no mercado muitas ferramentas gratuitas e pagas que permitem a realização de testes de desempenho de sistemas desenvolvidos, bem como ferramentas de

monitoramento que permitam a análise dos dados dos testes realizados. Testes como estes devem ser considerados em cenários corporativos pois permitem medir a escalabilidade das ferramentas antes mesmo de serem colocadas em produção.

A não utilização de cálculos complexos no protótipo tiveram grande influência no resultado deste tipo de arquitetura, que é o mais comumente utilizado em sistemas para web. Em situações onde há necessidade de cálculos complexos, como vimos anteriormente no capítulo 2, esse tipo de arquitetura pode não ser o mais indicado.

4.5. Considerações finais

Outros testes podem e devem ser realizados afim de que sejam mitigadas várias dúvidas em relação à arquitetura.

Os testes e resultados colhidos neste trabalhado já permitem que desenvolvedores considerem a arquitetura proposta como viável em vários cenários de alta concorrência para aplicações e serviços web.

5. CONCLUSÃO

Devido as mudanças cada vez mais rápidas que vem acontecendo na Web, as aplicações e arquiteturas de software conhecidas são colocadas diariamente à prova.

Conforme apresentado neste trabalho, o modelo arquitetural orientado a eventos, aplicado ao contexto de aplicações web, é uma ótima opção a se considerar quando a necessidade envolve concorrência entre um grande número de usuários, devido à relativa facilidade de se abstrair a programação orientada a eventos utilizando a linguagem JavaScript, que possui recursos nativos para tal.

A plataforma Node.js possui uma ótima documentação e muitos desenvolvedores tem se empenhado para criar módulos e ferramentas de forma a contribuir com a comunidade. Várias empresas que possuem destaque na Web e que necessitam de um nível de escalabilidade cada vez maior tem utilizado Node.js em alguns dos seus projetos e os resultados atingidos supriram a demanda. No entanto, ao se trabalhar com Node.js desenvolvedores mais experientes e que utilizam frameworks *full stack* podem sentir falta de ferramentas que tragam agilidade ao processo de desenvolvimento.

A construção de *Web API's* em Node.js utilizando o framework Express torna o desenvolvimento simples devido aos seus *middlewares* e implementação de roteamento de *endpoints*, mas deve ser levada em consideração a sua característica de ser "não opinativo" pois se por um lado o framework permite grande flexibilidade, por outro lado deixa por conta do desenvolvedor qual estrutura ele irá utilizará para a organização da base de código, qual ferramenta irá manipular dados, se aplicação utilizará MVC (*Model-View-Controller*) ou outro paradigma.

Sobre a integração com sistemas gerenciadores de banco de dados, existe um vasto material sobre a utilização de Node.js com bancos de dados não relacionais, como o MongoDB em conjunto com a ODM Mongoose por exemplo. Já informação a respeito da integração com banco de dados relacionais, como o PostgreSQL ou MySQL, é bastante escassa e com módulos para manipulação de dados bastante simples e pouco ágeis se comparadas com outras ferramentas e frameworks do mercado, obrigando muitas vezes o programador a ter que escrever *queries* mais simples utilizando SQL. A utilização do Sequelizejs como ferramenta de ORM para este projeto foi feita com certa dificuldade pois a curva de aprendizado é acentuada devido principalmente à sua documentação bastante pobre, não acompanha os seus *releases* e possui poucos exemplos de implementação e boas práticas de uso.

Os resultados dos testes realizados na API e apresentados no capítulo 4 mostraram um

grande índice de aproveitamento tanto de recursos de hardware quanto de recursos computacionais do protótipo. O protótipo também obteve grande desempenho em relação à sua disponibilidade, pois como os testes mostraram, a API mesmo recebendo grande concorrência, não gerou nenhum erro ou queda da aplicação.

5.1. Trabalhos Futuros

Tem como trabalho futuro a abordagem dos seguintes tópicos:

- Validação de arquiteturas clássicas utilizando o mesmo conceito do protótipo para efeito de comparação de desempenho e produtividade utilizando outros frameworks;
- Utilização de banco de dados não relacionais para fins de comparação de desempenho, escalabilidade e confiança no que diz respeito a armazenamento e recuperação de dados de geolocalização;
- Estudos de usabilidade para a utilização da interface por deficientes visuais.

BIBLIOGRAFIA

Abbott, M. L., & Fisher, M. T. (2011). *Scalability Rules.* Boston: Addison-Wesley Professional.

ANATEL. Agencia Brasileira de Telecomunicações. (25 de Fevereiro de 2015). *Anatel Dados.* Acesso em 6 de Março de 2015, disponível em http://www.anatel.gov.br/Portal/exibirPortalNoticias.do?acao=carregaNoticia&codigo=36556

Arango, M., & Kaponig, B. (2009). Ultra-scalable Architectures for Telecommunications and Web 2.0 Services. *Intelligence in Next Generation Networks, 2009. ICIN 2009. 13th International Conference on* (pp. 1-7). Bordeaux: IEEE.

Atlassian. (30 de outubro de 2015). *Atlassian Bitbucket.* Fonte: Atlassian: https://bitbucket.org/

Barbosa, A. F. (2014). *Pesquisa sobre o uso das tecnologias da informação e comunicação no Brasil.* São Paulo: Comitê Gestor da Internet no Brasil.

Beltran, V., Carrera, D., Torres, J., & Ayguade, E. (2004). Evaluation the scalability of Java event-driven Web servers. *Parallel Processing, 2004. ICPP 2004. International Conference on* (pp. 134-142). Barcelona: IEEE.

Bootstrap. (30 de outubro de 2015). *History.* Fonte: Bootstrap: http://getbootstrap.com/about/

Candy, K. M., & Schulte, W. R. (2009). *Event Processing: Designing IT Systems for Agile Companies.* Chicado: McGraw-Hill Education.

Cantelon, M., Harter, M., Holowaychuck, T. J., & Rajlich, N. (2014). *Node.js in Action.* Shelter Island: Manning Publications Co.

Chieu, T., Mohindra, A., & Karve, A. A. (2011). Scalability and Performance of Web Applications in a Compute Cloud. *e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on* (pp. 317-323). Pequin: IEEE.

Colbran, S., & Schulz, M. (2015). An update to the software architecture of the iLab Service Broker. *Remote Engineering and Virtual Instrumentation (REV), 2015 12th International Conference on* (pp. 90-93). Bangkok: IEEE.

Cordeiro, D. d. (2006). Estudo de escalabilidade de servidores baseados em eventos em sitemas multiprocessados: um estudo de caso completo. *Dissertação (Mestrado em Ciência da Computação) - Instituto de Matemática e Estatística, Universidade de São Paulo*. São Paulo, São Paulo. Acesso em 5 de Setembro de 2015, disponível em http://www.teses.usp.br/teses/disponiveis/45/45134/tde-19062012-163305

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2013). *Sistemas Distribuídos - Conceitos e Projeto.* Porto Alegre: Bookman.

Dabek, F., Zeldvich, N., Kaashoek, F., Mazière, D., & Morris, R. (1 de Julho de 2002). Event-driven programming for robust software. *EW 10 Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pp. 186-189.

Debian. (30 de outubro de 2015). *Sobre o Debian.* Fonte: Debian: https://www.debian.org/intro/about

DigitalOcean. (30 de outubro de 2015). *Pricing.* Fonte: Digital Ocean: https://www.digitalocean.com/pricing/

Etzion, O., & Niblett, P. (2011). *Event Processing in Action.* Stamford: Manning Publications Co.

Fowler, M. (2005). *UML Essencial.* Porto Alegre: Bookman.

Fowler, M. (2006). *Padrões de arquitetura de aplicações corporativas.* Porto Alegre: Bookman.

GIT. (30 de outubro de 2015). *GIT-SCM.* Fonte: GIT: https://git-scm.com/

Google. (30 de outubro de 2015). *Google Maps APIs.* Fonte: Google Developers: https://developers.google.com/maps/

Hapi. (1 de outubro de 2015). *Hapi*. Fonte: Hapi: http://hapijs.com/

Ihrig, C. J. (2013). *Pro Node.js for Developers.* Nova Iorque: Apress.

Instituto Brasileiro de Geografia e Estatística - IBGE. (2012). *Censo Demográfico 2010.* Rio de Janeiro: IBGE.

Jaramillo, D., Duy, N. V., & Newhook, R. (2014). Real-time experience techniques for collaborative tools on mobile. *SOUTHESTCON 2014, IEEE* (pp. 1-6). Lexington: IEEE.

Joyent. (30 de outubro de 2015). *About Node.js.* Fonte: Node.js: https://nodejs.org/en/about/

jQuery. (30 de outubro de 2015). *jQuery.* Fonte: The JQuery Foundation: http://jquery.com/

Keymetrics. (30 de outubro de 2015). *PM2.* Fonte: Keymetrics: http://pm2.keymetrics.io/

Krakenjs. (1 de outubro de 2015). *Krakenjs*. Fonte: PayPal Open Source: http://krakenjs.com/

Laudon, K., & Laudon, J. (2011). *Sistemas de Informação Gerenciais.* São Paulo: Pearson Education do Brasil.

Liu, D., & Deters, R. (21 de Abril de 2009). The Reverse C10K Problem for Server-Side Mashups. *Service-Oriented Computing - ICSOC 2008 Workshops*, pp. 166-177.

Liu, H. H. (2009). *Software performance and scalability: a quantitative aproach.* New Jersey: John Wiley & Sons, Inc.

Mardan, A. (2014). *Pro Express.js.* Nova Iorque: Apress.

MDN. (30 de outubro de 2015). *Promise.* Fonte: Mozilla Developer Network: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise

Nedelcu, C. (2013). *Nginx HTTP Server Second Edition.* Birmingham: Packt Publishing.

Newman, S. (2015). *Building Microservices.* Sebastopol: O'Reilly Media.

Nginx. (30 de outubro de 2015). *About Nginx.* Fonte: Nginx: http://nginx.org/en/

Nginx. (30 de outubro de 2015). *Beginners Guide.* Fonte: Nginx: http://nginx.org/en/docs/beginners_guide.html

NPM. (1 de outubro de 2015). *NPM*. Fonte: Node Package Manager: https://www.npmjs.com/

P, P., R, B., & Kamath, M. (2015). Performance analysis of process driven and event driven web servers. *Intelligent Systems and Control (ISCO), 2015 IEEE 9th International Conference on* (pp. 1-7). Coimbatore: IEEE.

Palácio do Planalto. (19 de dezembro de 2000). *Casa Civil. Subchefia de Assuntos Jurídicos.* Acesso em 10 de Fevereiro de 2015, disponível em https://www.planalto.gov.br/ccivil_03/leis/l10098.htm

Pasquali, S. (2013). *Mastering Node.js.* Birmingham: Packt Publishing.

Paudyal, U. (2011). *Scalable Web Application using Node.JS and CouchDB (student paper).* Uppsala: Uppsala universitet.

Pereira, C. R. (2013). *Aplicações web real-time com Node.js.* São Paulo: Casa do Código.

PM2. (30 de outubro de 2015). *Cluster Mode.* Fonte: PM2: http://pm2.keymetrics.io/docs/usage/cluster-mode/

PostGIS. (30 de outubro de 2015). *Features.* Fonte: PostGIS Project: http://postgis.net/features/

PostgreSQL. (30 de outubro de 2015). *About PostgreSQL.* Fonte: The PostgreSQL Global Development Group: http://www.postgresql.org/about/

Pressman, R. S. (2002). *Engenharia de Software.* Rio de Janeiro: McGraw-Hill.

Project, T. E. (30 de outubro de 2015). *Home.* Fonte: Express: http://expressjs.com/

Restify. (1 de outubro de 2015). *Restify*. Fonte: Restify: http://restify.com/

Richardson, L., & Ruby, S. (2007). *RESTful Web Services.* Sebastopol: O'Reilly Media Inc.

Sails.js. (1 de outubro de 2015). Fonte: Sails: http://sailsjs.org/

Savchenko, D., Radchenko, G., & Taipale, O. (2015). Microservices validation: Mjolnirr platform case study. *Information and Communication Technology, Eletronics and Microeletronics (MIPRO), 2015 38th Internation Convention* (pp. 235-240). Opatija: IEEE.

Sequelize. (30 de outubro de 2015). *Home.* Fonte: Sequelize: http://docs.sequelizejs.com/en/latest/

Siege. (30 de outubro de 2015). *Siege Home.* Fonte: Joe Dog Software: https://www.joedog.org/siege-home/

Silveira, P., Silveira, G., Lopes, S., Moreira, G., Steppat, N., & Kung, F. (2012). *Introdução à Arquitetura e Design de Software.* São Paulo: Casa do Código.

Smith, B. (2015). *Beginning JSON.* Nova Iorque: Apress.

Sommerville, I. (2011). *Engenharia de Software.* São Paulo: Pearson Pretince Hall.

Strongloop. (7 de setembro de 2015). *What Makes Node.js Faster Than Java?* Fonte: StrongLoop - An IBM Companny: https://strongloop.com/strongblog/node-js-is-faster-than-java/

Stubbs, J., Moeira, W., & Dooley, R. (2015). Distributed Systems of Microservices Using Docker and Serfnode. *Science Gateways (IWSG), 2015 7th International Workshop on* (pp. 34-39). Budapest: IEEE.

Tanembaum, A. S., & Steen, M. v. (2007). *Sistemas Distribuídos: principios e paradigmas.* São Paulo: Pearson Pretince Hall.

Teixeira, P. (2013). *Professional Node.js: Building JavaScript-Based Scalable Software.* Indianapolis: John Wiley & Sons, Inc.

Vladutu, A. (2014). *Mastering Web Application Development with Express.* Birmingham: Packt Publishing.

W3C. (30 de outubro de 2015). *Geolocation API Specification.* Fonte: W3C Editors Draft 11 July 2014: http://dev.w3.org/geo/api/spec-source.html

Welsh, M., Culler, D., & Brewer, E. (21 de Outubro de 2001). SEDA: an architecture for well-conditioned, scalable internet services. *SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles*, pp. 230-243.

Young, A., & Harter, M. (2015). *Node.js in Practice.* Nova Iorque: Manning Publications Co.

Zabbix. (2015 de outubro de 2015). *What is Zabbix.* Fonte: Zabbix: http://www.zabbix.com/product.php

Zeldovich, N., Alexander, Y., Dabek, F., Morris, R. T., Mazières, D., & Kaashoek, F. (2003). Multiprocessor Support for Event-Driven Programs. *USENIX 2003 Annual Technical Conference.* Texas.