UNIVERSIDADE FEDERAL FLUMINENSE

THIAGO OLIVEIRA DE SOUZA

COMUNICAÇÃO *REAL-TIME* COM NODE.JS E WEBSOCKETS

Niterói

2016

THIAGO OLIVEIRA DE SOUZA

COMUNICAÇÃO *REAL-TIME* COM NODE.JS E WEBSOCKETS

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Orientador:
CLEDSON OLIVEIRA DE SOUSA

NITERÓI
2016

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S729 Souza, Thiago Oliveira de
Comunicação *real-time* com Node.js e WebSockets / Thiago Oliveira de Souza. – Niterói, RJ : [s.n.], 2016.
71 f.

Projeto Final (Tecnólogo em Sistemas de Computação) – Universidade Federal Fluminense, 2016.
Orientador: Cledson Oliveira de Sousa.

1. Aplicação web. 2. Tempo real. 3. WebSocket. I. Título.

CDD 005.3

THIAGO OLIVEIRA DE SOUZA

COMUNICAÇÃO *REAL-TIME* COM NODE.JS E WEBSOCKETS

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, ___ de _____ de 2016.

Banca Examinadora:

_____

Cledson Olveira de Sousa, M.Sc. – Orientador

UFF - Universidade Federal Fluminense

_____

Douglas Paulo de Mattos, M. Sc. – Avaliador

UFF – Universidade Federal Fluminense

Dedico este trabalho a minha esposa.

AGRADECIMENTOS

A meu Orientador Cledson Sousa pelo estímulo e atenção que me concedeu durante a elaboração do trabalho.

Aos Colegas de Gazeus pelo incentivo e aprendizado diário.

A todos os meus familiares e amigos pelo apoio e compreensão.

“A diferença entre uma pessoa de sucesso e as outras não é a falta de força, nem a falta de conhecimento, mas particularmente a falta de determinação”.

Vince Lombardi

RESUMO

Este estudo trata da utilização da plataforma Node.js em conjunto com WebSockets para o desenvolvimento de aplicações em tempo real. Através do estudo teórico das particularidades de cada uma dessas tecnologias, é possível entender como elas contribuem para o desenvolvimento de aplicações no modelo estudado. Para validar as informações teóricas, o trabalho apresenta vantagens e desvantagens da plataforma Node.js, WebSockets e de outras tecnologias concorrentes.

Palavras-chaves: Aplicações em tempo real, Node.js e WebSocket.

LISTA DE ILUSTRAÇÕES

LISTA DE GRÁFICOS

LISTA DE ABREVIATURAS E SIGLAS

AJAX - A*synchronous JavaScript And XML*

API – *Application Programming Interface*

AST - *Abstract Syntax Tree*

DOM – *Document Object Model*

GC – *Garbage Collector*

HTTP – *HyperText Transfer Protocol*

JIT – *Just-in-Time*

JS – *JavaScript*

VM – *Virtual Machine*

XHR - XMLHttpRequest

SUMÁRIO

1 INTRODUÇÃO

A evolução dos sistemas Web e a disseminação dos serviços *mobile* tornaram a comunicação em tempo real um requisito indispensável para o sucesso das aplicações desenvolvidas. Redes sociais, aplicações de troca de mensagens instantâneas e jogos *online* são exemplos corriqueiros de aplicações que demandam resposta imediata a interações dos usuários. Mas, para atender essa demanda de forma eficiente, é preciso escolher o melhor conjunto de ferramentas na construção da aplicação, tendo em vista preocupações com o desempenho e escalabilidade do sistema.

De acordo com Fowler (2003), “um sistema escalável é aquele que lhe permite adicionar hardware e obter uma melhora de desempenho proporcional.” De fato, dentre as possibilidades de melhorar o desempenho da aplicação, a mais simples, mas nem sempre melhor, é melhorar o hardware do servidor, entregando maior poder de processamento à aplicação, o que é popularmente conhecido como escalabilidade vertical. Porém, a escolha correta das tecnologias adequadas ao sistema em desenvolvimento também constitui uma forma barata e eficiente de permitir que a aplicação não enfrente gargalos durante seu crescimento.

Tendo como base o clássico artigo feito por Kegel (1999) *The C10K problem*[1], no qual ele questiona a capacidade de escalabilidade e de suportar um grande número de usuários conectados aos servidores, faremos uma análise de duas tecnologias que vêm ganhando bastante atenção por parte dos desenvolvedores nos últimos anos, justamente por atender aos questionamentos levantados no artigo: Node.js e WebSockets. O intuito é verificar se a escolha pela utilização de ambas as tecnologias dentre outras opções, pode render benefícios para a escalabilidade e o desempenho da aplicação.

[1] http://www.kegel.com/c10k.html

Assim, foram realizados testes de avaliação comparativa com alguns casos de uso utilizando Node.js e PHP rodando sobre um servidor Apache, analisando a eficiência dessas tecnologias sob um elevado número de requisições simultâneas. Paralelamente, é feito um comparativo entre o uso de WebSockets e métodos de *polling*, que são utilizados para forjar uma comunicação bidirecional entre servidor e cliente para simular o *real-time*.

Cada tecnologia discutida nesse trabalho é analisada para entender os motivos que a tornam uma escolha mais apropriada para o tipo de aplicação proposta neste estudo. No Capítulo 2, é feita uma análise conceitual da plataforma Node.js, apresentando suas características que a distinguem de outras plataformas e servidores. O intuito é entender seus conceitos e como tirar o melhor proveito de seus paradigmas para desenvolver uma aplicação com melhor desempenho.

No terceiro capítulo, é feito um estudo conceitual sobre o protocolo WebSockets e sua API, implementada juntamente com a especificação HTML5. Nesse capítulo, busca-se entender a motivação da criação desse novo protocolo e o que o diferencia dos protocolos tradicionalmente utilizados no modelo cliente/servidor.

Finalmente no quarto capítulo, são realizados os testes comparativos para verificar se o desempenho obtido pelo conjunto Node.js e WebSockets constitui uma alternativa considerável para construção das aplicações almejadas.

2 NODE.JS

Segundo a documentação do site oficial [19] do projeto, Node.js é uma plataforma construída sobre o *Chrome V8 Engine*[2], compilador JIT *(just-in-time)* de JavaScript desenvolvido pelo Google Inc, com o objetivo de construir aplicações *web* de alta performance e escalabilidade. Utilizando o modelo de orientação a eventos e operações de I/O não bloqueantes da plataforma, é possível criar aplicações em tempo real com intensa troca de dados entre cliente e servidor de forma eficiente [19].

Hoje, a plataforma é utilizada nos serviços de dezenas de grandes empresas de tecnologia, tais como Linkedin, Walmart, PayPal e Groupon [16]. Para entender os fatores que levaram ao crescimento de interesse na plataforma, é preciso saber em quais circunstâncias históricas sua criação ocorreu e, principalmente, quais suas características que a tornam a tecnologia indicada para a utilização em aplicações em tempo real.

2.1 HISTÓRIA

Ryan Dahl [8], então programador em uma empresa especializada em serviços de *hosting* chamada Joyent, encontrava-se diante de um problema: como informar corretamente o progresso de *upload* a um usuário que tenta transferir um arquivo do servidor para o cliente. O que tornava o problema tão incômodo, na visão de Ryan, era o fato de que os métodos disponíveis para exibir esse *feedback* exigiam que fossem feitas consultas periódicas ao servidor para retornar um valor percentual que seria inserido no modelo de objeto de documentos (do inglês

[2]https://developers.google.com/v8/

document object model, ou DOM), a árvore de representação e interação de documentos de linguagem de marcação. Mas esse modelo não era eficiente para lidar com múltiplos uploads concorrentes, que gerariam, cada um, uma série de requisições para retorno do seu percentual completado. Outra forma muito utilizada à época era a técnica de long-polling, que possui como grande inconveniente o bloqueio dos processos no servidor por conta da operação de I/O até que haja uma mudança para ser enviada. Para Dahl, o caminho natural para resolver a questão era que o servidor enviasse atualizações de estado ao cliente baseado em mudanças, sem a necessidade de que o cliente pedisse tais requisições a todo instante [7].

Apesar de estar familiarizado com a linguagem Ruby e utilizar o módulo Mongrel[3] como inspiração, ele logo percebeu que a implementação do servidor utilizando Ruby como linguagem seria muito penosa, basicamente por conta de sua natureza síncrona e pelo desempenho de sua Máquina Virtual (do inglês Virtual Machine, ou VM) considerada demasiadamente lenta por Dahl, que a todo instante reescrevia trechos da plataforma em C para obter melhora na velocidade de execução. Como C era uma linguagem com grau de complexidade indesejado para o escopo do projeto que tinha em mente, Dahl buscou outras linguagens para implementação como Haskell e Lua, porém, por já apresentarem noções preconcebidas sobre o funcionamento de operações de I/O, a implementação nessas linguagens não obteve êxito [12].

Durante os estudos de Dahl para implementar seu projeto, os grandes navegadores investiam grandes quantidades de recursos em uma disputa pelo desenvolvimento de interpretadores JavaScript de melhor desempenho. Isso acarretou um crescimento acentuado na adesão e popularidade da linguagem, que se refletia no aumento do número de projetos *open source* voltados para sua *performance*, um gargalo histórico que sempre emperrara sua popularização. Nesse contexto, a decisão da equipe do Google Chrome de abrir o código de seu JavaScript *V8 Engine*, somada às características da linguagem, como a orientação à eventos nativa e a sintaxe familiar, que iam de encontro ao buscado por Dahl para

[3]http://www.rubydoc.info/gems/mongrel

implementar um servidor que atendesse sua necessidade inicial, tornaram a escolha do JavaScript uma decisão óbvia para o projeto.

É importante frisar que o Node.js não foi a primeira tentativa de implementação de JavaScript no lado do servidor. Projetos como Helma[4], AppEngine JS SDK[5] e RingoJS[6] surgiram antes do lançamento do Node.js, mas apresentam diferenças substanciais em sua construção, a começar pela adoção da Java Virtual Machine (JVM) como base no servidor. Mas, diferente da implementação de Dahl, os projetos apenas portavam a linguagem JavaScript para o servidor, sem se preocupar com I/O não bloqueante, assincronicidade, *single-threading* ou orientação a eventos, conceitos que pavimentaram o sucesso do Node.js. Portanto, qualquer comparação entre as implementações deve levar essa particularidade em conta, sendo mais adequada a comparação do Node.js a plataformas e *frameworks* que possuam características semelhantes, como o Twisted, para Python, o Event Machine, para Ruby e Swing, para Java [25].

Com o entendimento do contexto no qual se deu a criação do Node.js, é possível estudar cada uma das características que diferenciam a plataforma, começando pelo seu compilador de JavaScript, o *Chrome V8 Engine*.

2.2 CHROME V8 ENGINE

O *V8 Engine* é, segundo definição do site do projeto, o compilador de JavaScript de alto desempenho utilizado pelo navegador Google Chrome. Escrito em C++ e de código aberto, pode ser usado de forma independente ou embarcada em outros softwares escritos em C++ [31]. Ele foi projetado e implementado desde o início com foco em performance, particularmente, buscando resolver gargalos clássicos inerentes ao JavaScript, como a sobrecarga do operador “+” e sua própria característica de single *threading*, que sempre limitavam a aplicabilidade e

[4]http://helma.org/

[5]https://github.com/gmosx/appengine

[6]http://ringojs.org/

incremento da complexidade dos códigos que poderiam ser utilizados na aplicação, pois atrasavam a execução do código. A equipe de desenvolvimento do projeto V8 definiu três pilares fundamentais ao seu funcionamento: compilação do código fonte em JavaScript diretamente para linguagem de máquina nativa; gestão eficiente de memória, com alocação mais rápida de objetos e pequenas pausas para *garbage collection*; e, finalmente, introdução de classes ocultas utilizando *inline caching,* que aumentam a velocidade de acesso a propriedades e chamadas de funções [3].

2.2.1 COMPILAÇÃO JIT

Uma das características que deram ao *V8 Engine* uma liderança considerável na corrida pela melhor performance na execução de JavaScript foi a transição da interpretação dos códigos para a compilação JIT. O conceito de compilação *just-in-time* começou a tomar forma após pesquisas de McCarthy (1960) sobre a linguagem LISP, em que ele faz menção à compilação de funções de forma rápida o suficiente para que a saída do compilador não necessitasse ser salva [24]. Na compilação JIT, o código JavaScript é traduzido em código de máquina durante a execução do script, uma função por vez, à medida em que são executadas, sem geração de *bytecodes*, como em Java [24]. Isso significa que, mesmo em grandes bibliotecas de *scripts*, o compilador não se ocupará de compilar todas as funções, focando-se apenas nas que são chamadas em um dado momento.

No caso do V8, foi implementada uma infraestrutura complexa de compilação. Esta infraestrutura inclui o compilador base, chamado de *full codegen,* cuja tarefa é a produção de código nativo de máquina em alta velocidade, sem a realização de otimizações, e o compilador otimizador, denominado *Crankshaft*, responsável pela compilação não tão veloz, mas altamente otimizada do código. O *engine* compila o script utilizando o compilador base durante um período inicial, enquanto uma *thread* paralela é aberta para que o *profiler* (analisador de programas) interno ao V8 faça a seleção das chamadas *hot functions*, funções que são críticas e/ou usadas com frequência durante a execução. Finalmente, quando identificada

uma *hot function* pelo *profiler*, o compilador Crankshaft se encarrega de compilá-la no momento de sua próxima execução, de forma concorrente à compilação normal [5].

A compilação realizada pelo V8 é subdividida em fases. A primeira etapa é a de *parsing* (análise). Nessa etapa, o código é traduzido em uma árvore sintática abstrata, ou AST (*abstract syntax tree*), que será utilizada tanto pelo *full codegen* e pelo Crankshaft, e descartada após uso, pois seu armazenamento toma espaço na memória, mas sua utilização não é frequente e sua reconstrução é simples. Na etapa seguinte, é realizada a análise de escopo, em que o V8 define o uso das variáveis de acordo com seu escopo. Assim como na fase anterior, a análise de escopo também é utilizada pelos dois compiladores. Usando a AST, informações de escopos e o *feedback* de tipos, o compilador dá início à terceira etapa, que consiste na construção do grafo de controle de fluxo usado pelo *Hydrogen*, uma representação intermediária de alto-nível usada no V8. Na fase seguinte, ocorrem as otimizações que são aplicadas sobre o grafo do *Hydrogen*. No final da etapa de otimização, o grafo do *Hydrogen* é utilizado como base para a geração do grafo do *Lithium*, a representação de baixo-nível usada no V8. Finalmente, o Crankshaft emite uma série de instruções para cada instrução presente no grafo *Lithium.* Ao fim dessa etapa, o código é embrulhado em um objeto Code e a execução pode ser realizada pelo V8 [4].

2.2.2 ACESSO EFICIENTE USANDO HIDDEN CLASSES

Por se tratar de uma linguagem dinâmica, as propriedades podem ser adicionadas aos objetos JavaScript a qualquer momento durante a execução. Apesar de garantir flexibilidade ao programador, essa característica necessita de executar pesquisas dinâmicas (*dynamic lookups*) sempre que uma propriedade é chamada para encontrar sua posição em memória. Portanto, comparativamente, o acesso a propriedades em JavaScript tende a ser muito mais demorado do que o

acesso a variáveis de instâncias em outras linguagens como Java[7] ou C++[8]. Nestas linguagens, as variáveis de instância ficam alocadas em blocos fixos de memória determinados pelo compilador. Portanto, nesses casos, o acesso é uma questão de carregamento ou armazenamento usando uma única instrução [1].

Para otimizar o tempo de busca por propriedades, o V8 não faz uso de pesquisas dinâmicas. Para isso, foi implementado o conceito de *hidden classes*, que são representações dos objetos em memória [30]. Dessa forma, o acesso às propriedades de um objeto é feito de forma direta, mimetizando linguagens compiladas. A cada adição de uma nova propriedade ao objeto, ele muda de *hidden class*.

```
1  function Retangulo(lado, altura) {
2      this.lado = lado || 0;
3      this.altura = altura || 0;
4  }
5  var a = new Retangulo(5, 10);
6  var b = new Retangulo(1);
7  b.area = b.lado * b.altura;
```

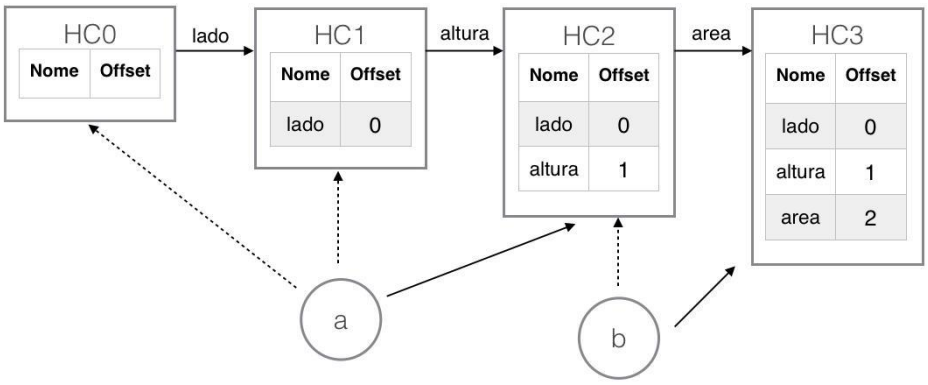

Figura 1: Exemplo de funcionamento e representação interna das hidden classes da classe Retangulo

Fonte: (AGUIAR, 2015)

Na Figura 1, é possível entender o processo de geração das *hidden classes* do objeto Retangulo. Durante a execução da linha 5, o comando *new* faz com o V8 crie uma nova instância do objeto. Como ainda não existe representação

[7]https://www.java.com

[8]http://www.cplusplus.com/

da classe na memória, é criada a HC0 (*hidden class 0*) sem qualquer propriedade definida. Ao instanciar o Retangulo, é executada a linha 2, que define a propriedade lado. Como trata-se de uma propriedade não existente na HC0, ela é estendida gerando a HC1, que recebe o *offset* 0. O valor de *offset* é utilizado como índice de acesso direto da propriedade na memória. Um ponteiro de nome *lado* é acrescentado à HC0 indicando que caso algum objeto aponte para HC0 receba o campo *lado* como nova propriedade, este objeto deverá utilizar a HC1. Ao executar a linha 3, o mesmo fluxo dá origem à HC2, acrescentando a propriedade *altura* com *offset* 1 e adicionando um ponteiro *altura* em HC1 indicando a HC2. Retornando ao fluxo do código, chega a vez de instanciar a variável b, porém, como já existem classes que atendem às propriedades necessárias de b, nenhuma *hidden class* será criada, assim b apontará para a HC2. Ao inserir a nova propriedade *area* no objeto da variável b, HC2 deixa de atender sua particularidade e uma nova *hidden class* é criada, HC3. A variável b passa a apontar para ela e HC2 ganha um ponteiro indicando HC3 [1].

Portanto, é visível o grau de reúso disponibilizado utilizando a abordagem demonstrada. Se uma nova instância for criada utilizando a classe Retangulo, ela compartilhará alguma *hidden class* criada anteriormente. É importante frisar que, para otimizar os ganhos com tal abordagem, é preciso que as propriedades dos objetos sejam declaradas em sua definição inicial, pois objetos com propriedades iguais, mas declaradas em ordem diferente uma do outra, possuiram *offsets* diferentes, constituindo, portanto, *hidden classes* diferentes e desvinculadas. Apesar disso, como vantagens dessa abordagem, é possível destacar que o acesso às propriedades não precisa ser feito através de pesquisas dinâmicas, além de permitir que o V8 aplique a técnica de *inline caching* para otimização [1].

Inline caching, ou IC, é uma função com múltiplas possibilidades de implementação normalmente gerada durante a execução, que pode ser chamada para lidar com operações específicas [5]. No caso do V8, o *full codegen* usa IC para operações de carregamentos, armazenamentos, chamadas, binárias, unárias, comparação e booleanas. Seu funcionamento é simples: é gerado um *stub*[9] com

[9] *Stub* é um trecho de código utilizado para substituir um programa mais longo que possui alguma funcionalidade.

comportamento semelhante a uma função que pode ser chamada e retorna um valor. Normalmente sua geração é feita durante a execução, mas também pode ser cacheado e reutilizado por outros ICs, além de conter código otimizado para lidar com tipos de operandos que aquele IC particular encontrou anteriormente. Ao encontrar um caso que não possuía tratamento, o *stub* retorna um erro e chama *runtime* C++ para lidar com a situação. O *runtime* lida com o caso e gera um novo *stub* preparado para lidar com o novo caso e todos os vistos anteriormente. As chamadas para o *stub* antigo são reescritas para utilizar o novo *stub* e a execução prossegue normalmente [5].

2.2.3 GARBAGE COLLECTION

Durante a execução do programa, a utilização da memória deve ser cuidadosamente gerenciada para evitar um grande número de problemas, incluindo vazamentos de memória[10]. O uso de *garbage collector* (GC)[11] reduz a preocupação que o programador deve ter com essa gestão, mas pode ser causa de longas interrupções na execução da aplicação caso seja implementado de maneira incorreta. Apesar de não fazer parte da especificação do ECMAScript[12] utilizada como base para o V8, sua equipe de desenvolvimento optou pelo uso de GC para gestão da memória [6]. Sua função é identificar áreas “mortas” da memória e reutilizá-las para outras alocações ou liberá-las para o sistema operacional.

Objetos vivos consistem em objetos que são apontados por objetos raíz ou por outros objetos ainda vivos. Objetos raíz são os apontados diretamente pelo V8, como por exemplo variáveis globais. Por definição, qualquer objeto raíz é considerado vivo. Quando um objeto não é mais apontado, ele é considerado morto

[10] Trata-se de um fenômeno que ocorre quando um trecho da memória, alocado para uma determinada operação, não é liberado quando não é mais necessária.
[11] *Garbage collection* é um processo usado para liberar uma área de memória não mais utilizada por um programa.
[12] ECMAScript é a especificação baseada no JavaScript, padronizada pela Ecma International no documento ECMA-262.

e passa a ocupar de forma desnecessária a memória. A tarefa do GC é então identificar e liberar a memória utilizada por esses objetos [6].

O GC empregado é bloqueante, geracional e preciso. Em outras palavras, ele para a execução do programa ao executar um ciclo de *garbage collection* e processa somente uma pequena parte do *heap* de objetos na maioria dos ciclos. Aproveitando-se do fato de que os objetos tendem a ter uma vida curta, o *heap* é dividido em duas partes principais: o espaço novo, em que os novos objetos são criados, e o espaço antigo, no qual objetos antigos que sobreviveram a ciclos anteriores são promovidos . A cada promoção de objetos e consequente mudança de espaço, todos os ponteiros daquele objeto são atualizados. Cada espaço é composto por páginas contíguas de 1MB até 8MB, de acordo com a heurística utilizada [6].

A alocação no espaço novo é bem simples e utiliza somente um ponteiro que é incrementado sempre que um espaço deve ser reservado para um novo objeto. Quando o ponteiro atinge o fim do espaço novo, é feita uma limpeza rápida (ciclo menor) que remove objetos mortos. Quando um objeto sobrevive às duas limpezas, ele é promovido ao espaço antigo, que só sofre uma varredura completa (ciclo maior) após a promoção de muitos objetos, mas esse limite é variável de acordo com o tempo de execução do programa, de seu comportamento e do tamanho alocado ao espaço [6].

2.3 DEMAIS ELEMENTOS DA ARQUITETURA NODE.JS

Somente com o *engine* V8, não é possível implementar um servidor completo, pois necessita-se da interface de suas funcionalidades com o sistema operacional. Dessa forma, Dahl acrescentou uma série de camadas extras em C e C++ para servirem de colagem aos elementos centrais da arquitetura do Node.js, suprindo lacunas de necessidades não atendidas pelo V8 ou estendendo suas bibliotecas [18]. O diagrama da Figura 2 ajuda a compreender como estão organizadas as camadas na pilha interna do Node.js.

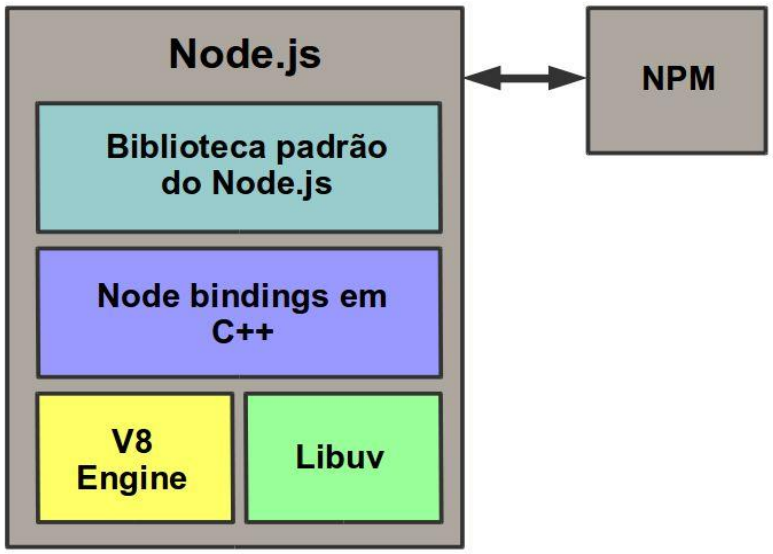

Figura 2: Arquitetura interna das camadas que compõem o Node.js

Enquanto o V8 tem como responsabilidade servir como *engine* de compilação JIT que garante a implementação da linguagem JavaScript, uma biblioteca fica responsável pela gestão do I/O assíncrono e pelo e*vent-loop*, pilares fundamentais no funcionamento do Node.js e que serão detalhados mais adiante.

Escrita em C, a biblioteca Libuv é uma abstração multi-plataforma utilizada para garantir a integração do Node.js com as operações de baixo-nível executadas junto ao sistema operacional, como operações de leitura e saída, interface de rede, *fork* de processos e do próprio *event-loop* [18]. Portanto, qualquer operação que demande interface com sistemas de arquivos, *sockets* e eventos de sistema passa, obrigatoriamente, pela Libuv, que faz internamente a gestão de processos através de uma *pool* de *threads,* que disponibiliza as *threads* para execução de funções assíncronas [13]. O Libuv provê, em conjunto com o *loop* de eventos, duas abstrações para uso: *handles* e *requests*. Os *handles* representam objetos de longa duração capazes de executar operações como chamadas ao *callback* a cada iteração do *loop*, desde que estejam ativos. Já os *requests* são

objetos de vida útil curta normalmente aplicados sobre um *handle,* mas que também podem ser executados diretamente sobre o *loop* [13].

Acima da camada composta pelo V8 e Libuv, se encontram os Node *bindings*, uma camada que fornece as interfaces de programação de aplicativos (do inglês *Application Programming Interface*, ou API) necessárias [1]. Um caso típico dos *binding* presentes no Node.js é a API de banco de dados. Como o V8 não provê uma interface para trabalhar com banco de dados, um *binding* é criado e utilizado para realizar a interface entre o V8 e a biblioteca do banco de dados utilizado. De modo geral, essa camada atua sobre as APIs fornecidas pela biblioteca padrão do Node.js, mas que não possuem interface nativa no V8 ou Libuv [1].

Finalmente, a camada mais externa do Node.js expõe sua biblioteca padrão contendo seus módulos próprios, todos escritos utilizando JavaScript. No momento do desenvolvimento, cada módulo nativo precisa ser importado para a aplicação a ser desenvolvida, bastando utilizar a diretiva `require('nome_do_modulo')` antes de qualquer instrução no código fonte, tornando os métodos do módulo importado disponíveis para o desenvolvedor. A aplicação roda, então, sobre toda essa pilha, utilizando os módulos de alto nível disponibilizados pelo Node.js. É importante mencionar que o *Node Package Manager*, conhecido como NPM, atua ao lado dessa estrutura para facilitar a busca, disponibilização e instalação de módulos desenvolvidos por terceiros [1]. O funcionamento do NPM será discutido em detalhes mais adiante.

2.4 O *LOOP* DE EVENTOS

Dentre todas as características que definem o Node.js, nenhuma é mais importante e tão significativa para o funcionamento da plataforma quanto a implementação do *loop* de eventos. Inspirado por projetos como o Python Twisted e o Ruby Event Machine [25], o Node.js faz uso de um *loop* de eventos, disponível graças à presença da Libuv, que roda sobre uma única *thread*, com uma estrutura de fila FIFO (First In, First Out), responsável por duas funções: detectar eventos

gerados na aplicação e disparar as funções associadas à cada evento. Sempre que um novo evento ocorre na aplicação, ele é colocado na fila. A cada iteração do laço, um único evento é retirado da fila e processado. Quaisquer outros eventos que possam vir a ser gerados são colocados ao fim da fila. Quando o processamento do evento termina, o laço retoma o controle e na iteração seguinte um novo evento é processado [1].

É importante frisar que apesar do *loop* rodar em uma única *thread*, o processamento dos eventos nem sempre é realizado na mesma *thread* utilizada pelo laço. Quando uma chamada requisita o uso de algum recurso que gere um bloqueio enquanto aguarda a resposta, como um banco de dados ou sistema de arquivos, o servidor inicia a chamada, mas anexa uma função denominada *callback* que é executada quando a requisição está pronta ou encerrada. Nesse momento um evento é emitido disparando a função *callback*, que executa algum processamento com a resposta (ou erro) gerado pela chamada [1]. O fluxo de execução do *loop* de eventos é exemplificado no diagrama da Figura 3.

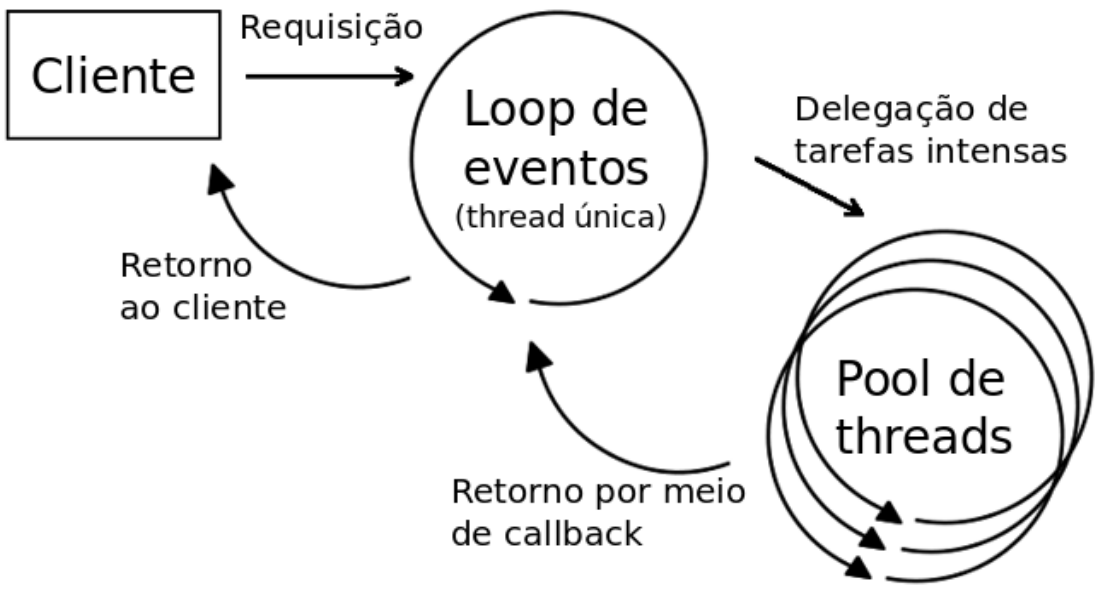

Figura 3: Esquema de funcionamento do loop de eventos

Como é visível no diagrama 3, o laço, operando sobre uma única *thread*, é responsável por processar a fila contendo as requisições originadas pelos clientes,

delegando as tarefas de alto custo computacional para uma das *threads* auxiliares disponíveis no *pool*. Quando a operação é completada, a *thread* retorna as informações ao laço de eventos por meio das funções *callback* e fica disponível no *pool* novamente. Finalmente, o laço de eventos retorna as informações geradas ao cliente ou ao processo que originou a chamada inicialmente. Qualquer outro evento originado em decorrência do processo é colocado no final da pilha de eventos. O funcionamento do *loop* de eventos evidencia as características definitivas para a compreensão das aplicabilidades do Node.js.

2.5 PROGRAMAÇÃO ORIENTADA A EVENTOS E I/O NÃO BLOQUEANTE

Voltando ao problema inicial enfrentado por Ryan Dahl, a emissão de atualizações de estado por parte do servidor, a existência de API's nativas no JavaScript que permitem registrar escutas específicas para eventos foi preponderante para a escolha da linguagem para implementação no projeto [12]. Como consequência dessa escolha, o desenvolvimento da plataforma acabou convergindo para a adoção do paradigma de orientação a eventos.

Como o nome já deixa explícito, a programação orientada a eventos é um paradigma tipicamente usado na programação de interfaces interativas, em que o usuário dispara eventos através de cliques ou arrasto de itens, criando o fluxo de execução [27]. Portando, esse paradigma gera eventos que são tratados por *handlers* ou por funções *callback*. Para exemplificar o funcionamento da programação orientada a eventos, considere uma consulta hipotética a um banco de dados em um paradigma tradicional de programação em que a *thread* fica ociosa aguardando o retorno da pesquisa no banco de dados para somente após a resposta retomar o fluxo de processamento da aplicação [27]. No contexto de programação orientada a eventos, a consulta seria realizada como exibida na Figura 4.

```
consulta_finalizada = function (resultado) {
                executar_algo_usando(resultado);
}
query('SELECT * FROM postagens WHERE id = 1', consulta_finalizada);
```

Figura 4: Exemplo de consulta ao banco orientada a eventos

O trecho acima executa uma consulta em banco de dados, mas, ao invés de bloquear a execução da aplicação enquanto aguarda o resultado, repassa o controle ao *loop* de eventos e somente após o retorno da informação obtida pela consulta no banco, a função *callback* é disparada para prosseguir com o processamento do resultado obtido, resultando em um funcionamento assíncrono [27].

Em um cenário comum, operações de entrada ou saída realizadas pelo sistema operacional são bloqueantes, ou seja, uma solicitação de leitura ou escrita bloqueia a execução do programa até que haja o retorno de uma resposta. Em servidores que lidam com volumes grandes de requisições simultâneas, o bloqueio representa um grave problema na interface cliente e servidor, pois um pequeno período de bloqueio pode gerar o travamento do programa devido ao acúmulo de requisições. Na arquitetura concebida para o Node.js, processos de alto custo computacional, que fatalmente gerariam bloqueios prolongados em um sistema bloqueante, são delegados às *threads* auxiliares, operando de forma assíncrona ao laço de eventos. Dessa forma, o *loop* permanece livre e atendendo as demais requisições da fila até que o resultado da operação de entrada/saída seja repassado via *callback* [27].

A primeira vantagem que se traduz desse modelo de funcionamento é a redução expressiva do uso de memória do servidor e da capacidade de processamento da UCP. Enquanto em uma arquitetura tradicional baseada em *threads*, há a necessidade de alocação de uma nova *thread* para cada processo iniciado, a orientação a eventos do Node mantém apenas a *thread* do laço de eventos ativa ininterruptamente e somente abre novas *threads* de acordo com a necessidade. Com o uso menos intensivo de memória e processamento, há um

aumento significativa na capacidade do servidor de lidar com requisições simultâneas [27].

Porém, a flexibilidade proporcionada pela escrita de um código com execução assíncrona pode ser também um problema durante o desenvolvimento de aplicações mais complexas. Graças a natureza do paradigma, um evento pode disparar um ou diversos *callbacks* e cada evento disparado como resultado de um *callback* pode possuir suas próprias funções de retorno. Esse aninhamento recorrente de funções é conhecido como *callback hell* [23].

2.6 CALLBACK HELL

O aninhamento de diversas funções consequentes, apesar de possível do ponto de vista funcional, é um inconveniente recorrente de aplicações em Node.js, mas que deve ser evitado pelo programador por representar um aumento exponencial na dificuldade de compreensão e manutenção do código da aplicação. A causa para a ocorrência de *callback hells* é a forma instintiva com a qual o programador tende a escrever o código de forma sequencial, acompanhando o fluxo de execução do início ao fim [23].

```
var form = document.querySelector('form');
form.onsubmit = function (submitEvent) {
    var name = document.querySelector('input').value;
    request({
        uri: "http://example.com/upload",
        body: name,
        method: "POST"
    }, function (err, response, body) {
        var statusMessage = document.querySelector('.status');
        if (err) {
             return statusMessage.value = err;

             statusMessage.value = body;
        }
    });
}
```

Contudo, evitar esse inconveniente é fácil graças às particularidades do JavaScript e do próprio ecossistema Node.js. A primeira maneira e mais simples forma de evitar um *callback hell* é nomeando as funções utilizadas. Dessa forma evita-se o aninhamento de funções anônimas [23]. O código exibido na Figura 5 e 6, retirado do site CallbackHell.com, dedicado a apresentar boas práticas de desenvolvimento em Node.js, mostra dois códigos que produzem o mesmo resultado, mas com níveis diferentes de aninhamento.

Figura 5: Exemplo de callback hell causado por aninhamento

Nessa primeira versão, é possível que seja feita uma chamada AJAX assíncrona ao servidor com o retorno de uma mensagem de status. Para facilitar a compreensão do código basta nomear cada uma das duas funções utilizadas e reduzir a profundidade de aninhamento. Aproveitando-se do *hoisting*[13] de funções, particular do JavaScript, pode-se reorganizar o código como exibido na Figura 6 de forma a se obter maior clareza.

[13] Hoisting é um comportamento padrão do JavaScript e trata-se do içamento das declarações de variáveis e funções para o topo do escopo.

```
document.querySelector('form').onsubmit = formSubmit;
function formSubmit (submitEvent) {
    var name = document.querySelector('input').value;
    request({
        uri: "http://example.com/upload",
        body: name,
        method: "POST"
    }, postResponse);
}
function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status');
    if (err) {
        return statusMessage.value = err;
    } else {
        statusMessage.value = body;
    }
}
```

Figura 6: Exemplo de código refatorado para evitar callback hell

Além do evidente benefício de organização e clareza do código, a refatoração do exemplo anterior também ajuda no período de desenvolvimento, pois erros durante a execução da aplicação terão a descrição do erro seguida pelo nome da função que os originou, diferente de funções anônimas [23].

Outro modo de organizar o código e evitar o *callback hell* é através da modularização da aplicação. A linguagem JavaScript sempre foi criticada devido ao *namespace,* ou escopo, compartilhado por todos os scripts, que podem gerar conflitos ou problemas de segurança. Como forma de minimizar esse problema, o Node.js implementa o padrão de modularização conhecido como *CommonJS*, em que cada módulo importado para a aplicação recebe seu próprio *namespace* e exporta somente as propriedades desejadas para uso. Dessa forma, as funções já vêm nomeadas, tendo o mesmo benefício do exemplo anterior e cada função ou módulo é separada em seu próprio pacote ou arquivo, sem adicionar pacotes que exercem funções secundárias ou complementares aos arquivos de sua aplicação [23].

Quando se fala em modularização de código para Node.js, é inevitável explicar o funcionamento da ferramenta que contribuiu para a adoção e popularização de seu ecossistema: o *Node Package Manager* ou popularmente conhecido como NPM.

2.7 NODE PACKAGE MANAGER

NPM é uma ferramenta *open source* criada em 2009 para gerir o armazenamento e disponibilização de módulos e bibliotecas JavaScript versionadas, facilitando o compartilhamento e reúso de código por parte dos programadores. Apesar de escrito para uso e instalado juntamente com o Node.js, trata-se de um projeto independente. Seu funcionamento é simples: o programador encapsula o código de seu módulo em um pacote e o envia para registro no NPM, tornando-o disponível para que outros desenvolvedores o utilizem [22].

A ideia é que os desenvolvedores submetam módulos de pequeno tamanho que resolvam um problema pequeno e bastante específico de forma eficiente. Assim, através da agregação de diversos blocos, é possível construir aplicações maiores, complexas e customizadas, mas com a certeza de utilizar dependências testadas e difundidas pela comunidade. Um dos benefícios dessa modularização é o fato de que uma equipe ou desenvolvedor pode se aproveitar de conhecimento externo através da adoção dos módulos apropriados, enquanto focam os esforços em outras áreas da aplicação. Além disso, o uso dos pacotes auxilia na adoção de padrões mais estritos de projeto, acarretando em ganho de desempenho quando há trabalho em conjunto entre diferentes programadores [22].

Todos os pacotes registrados possuem um arquivo chamado *package.json* que lista todas as características relevantes do pacote do qual faz parte como nome, descrição, autor(es), número de versão e dependências internas. Por meio das informações cadastradas na plataforma, um desenvolvedor pode buscar por módulos que resolvam problemas específicos encontrados durante o desenvolvimento de sua aplicação. Da mesma forma, através da lista de dependências presente no arquivo, é possível compartilhar as dependências de seu próprio projeto com outros programadores que contribuem no seu desenvolvimento [22].

A partir do comando *npm install,* que deve ser realizado na linha de comando a partir da pasta do projeto em que se econtra o arquivo *package.json*, o Node.js varre o arquivo, lendo e baixando qualquer dependência nele listada,

incluindo versões específicas. Os módulos baixados são salvos em uma pasta específica, nomeada *node_modules*, em que estarão disponíveis para importação no código da aplicação. Além de módulos importados para aplicações, alguns pacotes encontrados no registro do NPM são ferramentas independentes e completas que funcionam através da interface da linha de comando. É o caso de gerenciadores de tarefas como *GruntJS* e *GulpJS*. Nesses casos, a instalação é feita globalmente disponibilizando a ferramenta para qualquer usuário daquele ambiente.

Com a popularização do ecossistema, alguns pacotes ganharam destaque e tornaram-se padrão para adoção em projetos maiores [21]. A seguir, esses pacotes que ganharam relevância pelo volume de utilização ou por determinarem padrões de projetos serão apresentados e descritos.

2.7.1 EXPRESS.JS

Um dos pacotes mais populares, de acordo com as estatísticas de registro do NPM [21], o Express.js é um *framework* que provê uma abstração sobre o módulo http, nativo do Node.js, oferecendo novos métodos e uma sintaxe mais amigável para construção de aplicações *web* ou híbridas. O objetivo do módulo é facilitar a criação de APIs amigáveis, de forma rápida concisa e robusta [9]. Com o Express.js, o roteamento, validação de URLs e fornecimento de recursos é favorecido através de métodos menos verbosos e mais compreensíveis, enquanto oferece suporte à adoção de funções *middleware* nativas ou desenvolvidas por terceiros [9].

Dentre os métodos disponibilizados pelo *framework*, os mais importantes são http, *get, post, delete* e *put*, bastante utilizados em arquiteturas *RESTful*[14]. Dessa forma, o programador já inicia o projeto tendo um esqueleto sobre o qual pode desenvolver sua aplicação, enquanto mantém a coesão e uniformidade no código [9].

[14]Representational State Transfer (REST), em português Transferência de Estado Representacional

O *framework* se provou tão popular na comunidade de Node.js que foi adotado como componente padrão de roteamento na pilha denominada MEAN[15] (um acrônimo formado pelos quatro componentes utilizados: MongoDB, Express.js, Angular.js e Node.js).

2.7.2 UNDERSCORE.JS

Esta biblioteca oferece mais de uma centena de funções auxiliares em conjunto com funcionalidades naturais do JavaScript, mas sem estender objetos nativos da linguagem [29]. Dessa forma, o uso de funções semelhantes existentes nativamente ou a partir da biblioteca fica a critério do programador.

Através de uma abordagem funcional, as funções disponibilizadas vão desde mapas e filtros, passando por *binding* de objetos, *templating*, indexação e teste de igualdade de tipos, funcionalidade essa conhecidamente de difícil assimilação para iniciantes na programação e programadores oriundos de outras linguagens [29]. Do ponto de vista instrumental, é possível afirmar que a facilidade na construção de operações lógicas oferecida pela biblioteca Undercore.js é semelhante à facilidade que o jQuery introduziu na manipulação de elementos do DOM.

Com foco em obter o melhor desempenho possível enquanto mantinha um tamanho reduzido (menos de 60kb na versão de desenvolvimento, cerca de 16kb minificada) [29], a biblioteca se tornou relevante dentro do ecossistema Node.js, sendo adotada como dependência do pacote Cordova[16], um popular *framework* utilizado para criação de aplicativos *mobile* utilizando HTML, CSS e JavaScript, além de servir de base para o *framework* MV*, Backbone.js[17] [21].

[15] http://mean.io/

[16] http://cordova.apache.org/

[17] http://backbonejs.org/

2.7.3 LODASH

Nascida como um *fork* da biblioteca Underscore.js, a biblioteca Lodash possui um objetivo bastante semelhante em oferecer uma gama de funções auxiliares com foco na programação funcional, mantendo desempenho, modularidade e consistência em primeiro plano. Apesar de não oferecer novas capacidades em relação ao Underscore.js, o pacote Lodash melhora a usabilidade dos desenvolvedores através de uma sintaxe mais concisa para encadeamento e modularização de seus métodos, facilitando a importação somente das partes realmente relevantes para o projeto em desenvolvimento [14].

Diferente do Underscore.js, em que grande parte dos métodos nativos são reescritos, os criadores do Lodash optaram por uma implementação que trouxesse não somente o melhor desempenho de execução, mas principalmente uma forma de manter o funcionamento das funções o mais consistente possível entre diferentes versões de navegadores e plataformas [14]. Tal preocupação fez do Lodash o pacote mais dependido[18] do NPM em 2016 [21].

2.7.4 ASYNC

A biblioteca Async é um módulo de utilidades que provê funções de simples implementação e entendimento elaboradas especificamente para trabalhar com assincronicidade e controle de fluxo no código da aplicação. A biblioteca disponibiliza cerca de 70 funções que incluem desde auxílios para programação funcional sobre coleções, além de padrões de fluxo de controle assíncrono, como paralelização e serialização [2]. Dessa forma, o desenvolvedor pode abstrair a

[18]Um pacote é dependido quando um outro pacote o lista como sendo uma dependência para seu funcionamento.

complexidade de lidar com métodos assíncronos e escrever sua aplicação como faria normalmente em Node.js.

Dessa forma, o módulo facilita a etapa de desenvolvimento ao oferecer mecanismos que identificam e protegem o fluxo de execução da aplicação, evitando erros típicos ocasionados pela falta de conhecimento da programação orientada a eventos do Node.js, como execução de funções que bloqueiam o laço de eventos ou uso de variáveis antes que valores sejam atribuídos [2]. Assim, é mais fácil garantir que um método invocado para atuar sobre os dados recuperados de um banco de dados, por exemplo, sempre irá ser chamado após o retorno dos dados consultados.

2.7.5 MONGODB

Seguindo o módulo Express.js, que passou a integrar o MEAN stack devido à sua popularidade, o banco de dados não-relacional MongoDB também ganhou notoriedade graças ao desempenho quando utilizado em conjunto com o ecossistema Node.js. Dessa forma, é compreensível a popularidade alcançada pelo pacote de *drivers* do MongoDB. Seu objetivo é oferecer suporte a interações baseadas em chamadas *callback* ou através de *promises*[19], conceito introduzido a partir da especificação EcmaScript 6 [17].

Assim como outros pacotes de sucesso, o pacote de *drivers* permite ao programador importar todas as funcionalidades disponíveis ou somente o núcleo necessário para seu funcionamento [17]. Dessa forma, o desenvolvedor tem liberdade para importar somente funções relevantes para funcionamento de sua aplicação. Seu uso foi tão difundido que acabou gerando uma série de pacotes de terceiros que visam a facilitar sua implementação. Um exemplo que ganhou notoriedade e que hoje figura na lista de pacotes mais baixados do NPM é a

[19]Uma *promise* (traduzido como promessa) representa um valor que pode estar disponível agora ou futuramente.

biblioteca Mongoose[20], que auxilia na modelagem de objetos com aplicação direta na MongoDB.

2.7.6 NODEMAILER

Criado em 2010, o módulo tinha por objetivo suprir um vazio existente no ecossistema para o disparo de e-mails a partir de um servidor Node.js, enquanto oferecia uma sintaxe concisa e amigável. A simplicidade de uso o transformou na opção padrão para executar a tarefa. O módulo oferece suporte para disparo de e-mails em formato HTML, envio de anexos, protocolo seguro utilizando SSL/STARTTLS, definição de *templates* para uso automático e até mesmo diferentes métodos de transporte, em adição ao protocolo SMTP [20].

Seu uso acabou tão difundido na comunidade Node.js que diversos plugins externos foram criados: API's de serviços de e-mail como Mandrill, SendGrid, Sailthru e Sparkpost, além de suporte a diferentes formatos de texto utilizando Markdown, Dkim ou até mesmo imagens convertidas para o formato base64 [20].

2.7.7 SOCKET.IO

Socket.io é a primeira opção de pacote utilizável quando a aplicação realiza comunicação cliente-servidor através de *sockets*. O pacote permite a comunicação bidirecional em tempo real baseada em eventos com suporte a qualquer plataforma ou dispositivo, de *desktops* a *smartphones*, focando em estabilidade e velocidade de conexão. O pacote abstrai a implementação da comunicação em tempo real ao utilizar diferentes API's de transporte sempre tentando as que entregam melhor desempenho primeiro e, caso não haja suporte, passando à técnica seguinte. Dessa forma, o pacote mantém a compatibilidade com

[20]http://mongoosejs.com/

navegadores mais antigos e independe do dispositivo sendo utilizado pelo usuário [26].

Através da manipulação dos eventos no lado do servidor, o pacote controla o envio de informações para um usuário específico ou para um grupo de usuários. Sendo carregada também no lado do cliente, a biblioteca permite monitorar eventos cadastrados pelo desenvolvedor como mensagens enviadas para canais específicos de um *chat* e disparar ações específicas na *view* do usuário conectado. De forma semelhante, eventos podem ser emitidos no lado do cliente e os eventos gerados são processados e/ou repassados a outros usuários conectados. A biblioteca suporta a transmissão de qualquer tipo de objeto JSON serializado incluindo *strings*, *numbers, arrays* e booleanos. Objetos do tipo *buffer* de Node.js também são suportados, assim como o *stream* de leitura [26].

A sua implementação será mais discutida e explicada adiante juntamente com os conceitos e definições de WebSockets.

3 WEBSOCKETS

No modelo de requisição e resposta, representado pelo protocolo HTTP (*hypertext transfer protocol),* o conteúdo é solicitado pelo cliente e carregado no navegador do usuário, além da interação ser limitada ocorrendo somente através de novos carregamentos de página. A partir da introdução do AJAX (*asynchronous JavaScript and XML)*, o conteúdo das páginas passou a oferecer mais opções de interação, sendo disparados sem um carregamento completo de página ou um envio de formulário. Contudo, o fluxo básico da comunicação permanecia inalterado, tendo o cliente como único agente capaz de iniciar uma comunicação com o servidor, ainda utilizando o mesmo protocolo HTTP para realizar as requisições XHR (XMLHttpRequest) [28].

Mesmo com as limitações decorrentes do uso do HTTP, diversas técnicas surgiram para permitir uma comunicação de sentido servidor/cliente sempre que novos dados estiverem disponíveis. Uma das primeiras técnicas difundidas e que mais se destacou em um momento inicial foi a técnica conhecida como *polling*. Ela consistia no envio sistemático e periódico de requisições ao servidor para receber atualização de informações caso elas existissem. Assim, a comunicação bidirecional era simulada de forma rudimentar, pois o envio de novas informações do servidor para o cliente ocorria de forma invisível para o usuário. Logicamente, a solução não era ideal do ponto de vista computacional, já que diversas requisições inúteis poderiam ser geradas, caso o servidor não tivesse novos dados para enviar. Além disso a comunicação não ocorria em tempo real por depender do intervalo entre as requisições periódicas agendadas pelo sistema [15].

Em seguida, um conjunto de soluções denominado *Comet* ganhou adesão. Sua premissa era baseada em efetuar uma requisição HTTP, mas, diferente de uma requisição padrão, ao receber a resposta, não ocorria o encerramento da conexão, mantendo-a aberta para que o servidor envie eventos subsequentes,

utilizando uma mesma conexão, reduzindo o *overhead* gerado por uma nova requisição. A técnica era possível utilizando diversos métodos e tecnologias, sendo as mais utilizadas um *iframe* oculto de tamanho infinito que permanece em estado de carregamento indefinidamente, o que permitiria um canal aberto do qual o servidor poderia se aproveitar para enviar seus dados. Além dessa técnica, o conjunto *Comet* possui o *long-polling*, que é uma variação do *polling*, mas cada requisição de atualização de dados era mantida aberta até que o servidor tivesse informações novas para enviar. Ao receber os dados de atualização, o cliente encerrava a conexão e deveria estabelecer uma nova requisição. Com isso, reduzia-se o número de requisições inúteis [15].

Apesar de terem avanços em relação ao *polling* tradicional, ambas as soluções apresentavam pontos negativos. O *iframe* oculto não possibilitava saber o estado atual do objeto, assim como não permitia um método para realizar o tratamento de erros e distorcia a semântica e objetivo na utilização do *iframe*. Já o *long-polling* ainda sofria do excesso de requisições e troca de cabeçalhos HTTP. Além disso, no caso de um volume grande de atualizações vindas do servidor, a técnica perde sua vantagem em relação ao método anterior, pois a velocidade de abertura e fechamento de conexões é elevado. Em ambos os casos, a eficiência computacional acaba prejudicada principalmente pela necessidade de criação de dois canais de comunicação *half-duplex* para simular a comunicação bidirecional. Outro fator ineficiente compartilhado por todas as técnicas descritas é o *overhead* gerado pelo cabeçalho HTTP. A cada requisição iniciada, um novo cabeçalho é construído e transmitido juntamente com *cookies*, gerando trânsito de dados desnecessários para atender a requisição do cliente, elevando a latência do aplicativo em uso, o que impacta diretamente na escalabilidade e tempo de resposta da aplicação [15][28].

Com o constante avanço na complexidade e necessidade das aplicações *web* modernas, foi preciso criar uma forma de conexão persistente e de baixa latência, que oferecesse suporte a comunicações iniciadas por qualquer um dos atores envolvidos na conexão. Uma solução simples para satisfazer essas necessidades foi utilizar uma conexão TCP para suportar o tráfego em ambas as direções. Assim foi criada a especificação WebSockets, alternativa mais eficiente

aos métodos de *polling* que utilizam o HTTP. Os WebSockets proveem uma comunicação bidirecional *full-duplex*, ou seja, ambos os atores envolvidos na comunicação podem transmitir dados simultaneamente em ambos os sentidos, sem necessidade de múltiplas conexões HTTP como os métodos mencionados e reduzindo o *overhead* gerado pelo cabeçalho do protocolo. Sua premissa básica é a de que o enquadramento da requisição deve ser mínimo, simplificando sua implementação e economizando recursos [10].

Para compreender o funcionamento e implementação dos WebSockets, serão analisadas separadamente as duas partes integrantes da especificação: o protocolo WebSocket e a HTML5 WebSocket API.

3.1 PROTOCOLO WEBSOCKET

O protocolo WebSocket pode ser dividido basicamente em duas partes, o *handshake* e a transferência de dados. Antes de enviar e receber dados, cliente e servidor devem estabelecer a conexão através do *handshake*, que consiste de um pedido de atualização a partir do HTTP padrão. Isso se dá para que haja compatibilidade entre os *softwares* do lado do servidor e possíveis intermediários que sejam baseados no modelo de requisições HTTP, para que todos os clientes, HTTP ou WebSockets, conectados com servidor utilizem uma mesma porta. Caso seja aceito o pedido de atualização, uma mensagem de aceitação de conexão será enviada pelo servidor [10].

Assim, a requisição de atualização do protocolo de conexão é semelhante ao seguinte:

```
GET ws://servidor.exemplo.com HTTP/1.1
    Host: servidor.exemplo.com
    Upgrade: websocket
    Connection: Upgrade
    Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
    Origin: http://exemplo.com
```

Figura 7: Requisição de upgrade para o protocolo websocket

Como há uma atualização do protocolo, as URLs para uso de WebSockets utilizam o esquema *ws://,* equivalente a *http://*, ou *wss://*, equivalente ao *https://.* O cliente inclui o nome do *host* para que ambos concordem sobre qual servidor deverá ser utilizado. O campo *origin* é incluído como forma de proteção contra uso *cross-origin* não autorizado de *scripts* que utilizem a API de WebSockets no navegador. O servidor precisa, então, ser informado sobre qual é a origem dos *scripts* que serão executados para gerar a conexão via WebSocket. Caso a origem seja desconhecida para o servidor, o pedido de atualização poderá ser rejeitado, através da emissão de um código de erro adequado [10].

O resultado da resposta de aceitação de conexão é semelhante ao exibido na figura 8.

```
HTTP/1.1 101 Switching Protocols
    Upgrade: websocket
    Connection: Upgrade
    Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Figura 8: Resposta de upgrade para o protocolo websocket

Qualquer outro código retornado na primeira linha que não seja 101 indica que o *handshake* não foi completado com sucesso e o protocolo em uso ainda é o HTTP. O cliente então realiza a checagem do campo *Sec-WebSocket-Accept.* No caso do valor não corresponder ao esperado ou estiver vazio, o *handshake* não é efetivado. Além desses campos obrigatórios, outros campos podem ser utilizados no *handshake*, definindo subprotocolos ou versão de WebSocket disponível para uso [10].

Com o *handshake* completo, a conexão HTTP inicial é atualizada para WebSocket, que é apenas uma camada sobre o TCP/IP básico, oferecendo um modelo de segurança *web* baseado no endereço de origem do *script*, semelhante ao utilizado pelos navegadores, mecanismos de endereçamento e nomes, para suportar múltiplos serviços em uma mesma porta e múltiplos nomes de *host* em um mesmo endereço de IP, uma camada de enquadramento simples sobre o TCP além de reimplementar o *handshake* de fechamento para funcionamento na presença de

proxies e outros intermediários. À parte dessas mudanças, o WebSocket não inclui nada mais sobre o modelo bruto TCP, mantendo-se simples para coexistir com elementos de sua infraestrutura já existente, como *proxies*, mas ao mesmo tempo mantendo a maior segurança possível oriunda do TCP [10].

Com a conexão estabelecida, para garantir que cada mensagem seja reconstruída propriamente ao chegar ao seu destino, cada quadro na camada de aplicação é prefixado com 2 a 14 *bytes* de dados sobre o conteúdo da mensagem. O receptor só é notificado sobre a chegada de uma nova mensagem após todos os quadros relativos a tal mensagem sejam recebidos e reconstruídos. Esse sistema reduz o volume de dados transmitidos que não são relacionados diretamente ao conteúdo da mensagem que se deseja transmitir, o que diminui significativamente a latência de enfileiramento. O benefício dessa arquitetura é o ganho não somente em performance, mas principalmente em escalabilidade no lado do servidor [10].

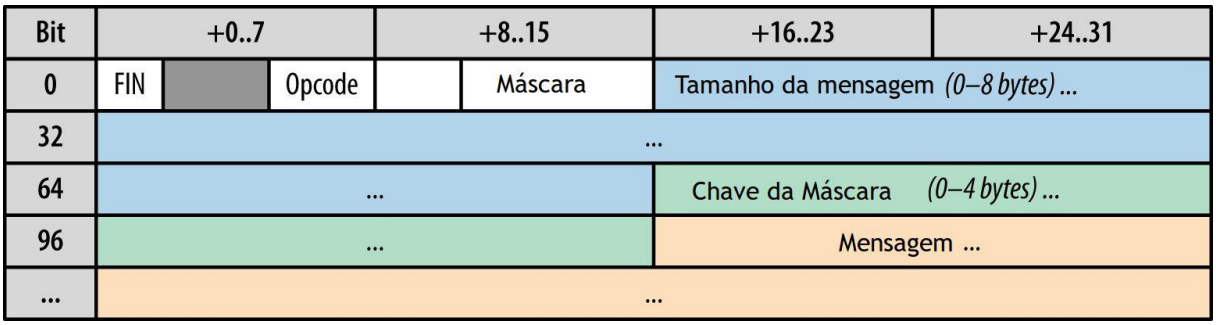

Figura 9: Quadro da mensagem utilizada pelo WebSocket

Após a utilização do *socket*, qualquer um dos pares pode solicitar o fechamento da conexão. O *handshake* de fechamento complementa o protocolo de fechamento utilizado no TCP, através dos pacotes FIN e ACK, pois o *handshake* utilizado no TCP nem sempre é confiável de ponta a ponta particularmente quando utilizados proxies ou outros intermediários. Para encerrar a conexão, o cliente ou servidor envia um quadro iniciado com um código “0x8”, contendo o código de estado do fechamento, além de um campo *reason*, que define a razão para o fechamento[10].

3.2 HTML5 WEBSOCKET API

Além de um novo protocolo, uma nova API foi desenvolvida disponibilizando o uso dos WebSockets aos navegadores através de JavaScript. Com uma interface simples, é possível gerenciar a conexão com o servidor de forma concisa e clara. Para solicitar uma nova conexão, basta instanciar um novo objeto WebSocket, passando a URL com o endereço de onde se deseja conectar como parâmetro do objeto e opcionalmente uma *string* ou um *array* de *strings* contendo os subprotocolos [28]. Como descrito na seção anterior, a URL utilizada deve conter como protocolo os prefixos *ws://*, para conexões normais, ou *wss://*, para conexões utilizando WebSockets seguros. Na Figura 10, temos um exemplo de instanciação utilizando WebSockets seguros.

```
var MeuWebSocket = new WebSocket( "wss://www.exemplo.com", ["soap",
"xmpp"] );
```

Figura 10: Exemplo de criação de websocket usando sua API

Com isso já há uma instância da conexão com o servidor. Através dessa instância podemos utilizar *listeners* que serão disparados de acordo com eventos gerados. O *listener* `onopen` é disparado sempre que a conexão é estabelecida e o *handshake* de abertura foi concluído com sucesso, indicando que a conexão já aceita envio e recebimento de dados. Com o *listener* `onerror` é possível capturar erros gerados durante a conexão. O *listener* `onmessage` é o principal e mais utilizado, pois é disparado sempre que novos dados são recebidos pelo cliente. Finalmente, o *listener* `onclose` é disparado quando a conexão entre os pares é encerrada completamente [28].

Após o estabelecimento da conexão, o envio de dados e mensagens para o servidor pode ser feito utilizando o método `send()`, disponível no objeto da conexão. Ele aceita *strings* e mensagens binárias como parâmetro. É importante notar que o método é assíncrono, pois a informação a ser transmitida é enfileirada

pelo cliente e a função retorna imediatamente. Assim, o desenvolvedor precisa ter em mente que o retorno dessa função não representa o efetivo envio da mensagem completa principalmente em casos em que a mensagem é demasiadamente extensa, mas o retorno é rápido. Como as mensagens são entregues exatamente na ordem em que são geradas, uma longa fila pode se acumular e atrasar a entrega de mensagens subsequentes [11]. Da mesma forma, mensagens enviadas pelo servidor são recebidas como objetos do tipo evento e seus conteúdos podem ser acessados através da propriedade `data`. Ao término do uso do WebSocket, é possível mandar um sinal de fechamento utilizando o método `close()` sobre a instância da conexão. Assim, um pedido de encerramento de conexão é iniciado entre as partes envolvidas e o WebSocket é fechado [28].

Além dos atributos e métodos citados, objeto WebSocket possui ainda o atributo *readyState*, através do qual é possível saber o estado atual da conexão. No momento da criação, o *readyState* do WebSocket é 0, que representa o estado de “conectando”. Após a execução do *handshake* de abertura a onexão passa então ao estado 1, que representa “conexão aberta”. Ao término de sua utilização e pedido de fechamento, a conexão passa pelo estado 2, que representa o andamento do fechamento do WebSocket, até finalmente atingir o estado 3, indicando que a conexão está encerrada. Através do *readyState* é mais fácil para o desenvolvedor depurar erros e entender o processo de inicialização da conexão [11].

3.3 APLICAÇÕES DO PROTOCOLO WEBSOCKET

Provendo uma interface simples, bidirecional e com fluxo de texto ou dados binários orientados a mensagens, aliado à extensão do protocolo TCP, os WebSockets adicionam uma gama de possibilidades de casos de uso ao desenvolvedor. Jogos multijogador on-line, aplicativos de *chat* em tempo real, agregador de notícias, atualizações de postagens de redes sociais, *streaming* de áudio e vídeo em formato binário, processamento de textos e planilhas de forma colaborativa, aplicações financeiras, ou seja, basicamente qualquer aplicação que

requeira uma conexão de baixa latência ou atualização constante de informações terá um ganho de desempenho e, principalmente, escalabilidade com a adoção dos WebSockets [11].

O principal ponto de comparação entre os métodos existentes de comunicação cliente servidor, como *polling*, *long-polling* e WebSockets é o *overhead* gerado pelos seus cabeçalhos. Mensagens trocadas a partir do protocolo WebSocket após o estabelecimento do *handshake* inicial são divididas em um ou mais quadros, dependendo do tamanho da mensagem, e carregam entre 2 e 14 *bytes* de *overhead*. Por conta dessa particularidade, os dados podem ser transmitidos tanto em formato binário, caso de áudio, vídeo ou imagens, ou em formato texto UTF-8. Comparativamente, requisições HTTP/1.x, utilizado nos métodos descritos de *polling* e *long-polling*, carregam entre 500 e 800 *bytes* extras de metadados incluindo *cookies* [11].

Inevitavelmente, a diferença no *overhead* dos protocolos impacta diretamente no desempenho de transmissão de ambos. Ao analisar diferenças de desempenho entre métodos de transporte baseados em XHR (XMLHttpRequest) e WebSockets, cabe salientar que a melhora de desempenho proporcionada pela adoção do último nada tem a ver com uma possível redução do *roundtrip time*, tempo que uma mensagem leva para trafegar entre cliente/servidor. A latência gerada pela propagação não é afetada pela mudança de protocolo, pois o número de mensagens trafegadas não se altera. O gargalo criado pelas requisições baseadas em HTTP e reduzido pelo WebSocket é relativo principalmente à latência de enfileiramento, tempo que uma mensagem precisa aguardar antes de ser transmitida para o outro par na relação cliente/ servidor. No caso das requisições XHR, a latência de enfileiramento é gerada pelo intervalo de aguardo entre as requisições de *polling* do cliente ao servidor. Uma mensagem pode já estar pronta para emissão, mas é necessária uma requisição por parte do cliente para que ele seja notificado sobre a existência dessa mensagem. O uso de *long-polling* reduz essa latência, mas a frequência de mensagens pode afetar negativamente a latência nesses casos. Já os WebSockets eliminam quase por completo essa latência, pois as mensagens são propagadas assim que disponíveis [11].

Porém, é importante destacar que requisições HTTP podem negociar formatos otimizados de transferências, como gzip para dados em texto, o que também influencia diretamente no desempenho. Em contrapartida, os WebSockets por suportar transferências de texto e dados binários, não implementam negociações de compressão, pois os dados binários já vêm otimizados. De todo modo, caso haja a necessidade, mecanismos de compressão devem ser implementados de forma seletiva a cada mensagem para alcançar a mesma economia de largura de banda e aumentar a velocidade da aplicação [11].

De forma semelhante, o uso de algum método de conexão baseado no protocolo HTTP pode ser mais recomendado do que os WebSockets, caso o tipo de dado que se deseja transmitir for armazenável em *cache* ou seja, armazenado pelo cliente ou um intermediário para acesso posterior, sem a necessidade de consulta ao servidor. Imagens, vídeos e áudios são exemplos de dados que se beneficiam do armazenamento em *cach*e, dependendo do objetivo da aplicação, trariam mais ganhos se transmitidos através de HTTP. É possível implementar uma arquitetura que priorize a entrega de texto e outros dados não armazenáveis em *cache* somente via WebSockets, enquanto mensagens de controle recebidas via WebSockets disparam requisições XHR para obtenção dos recursos armazenáveis em cache via protocolo HTTP. Dessa forma ganha-se os benefícios oferecidos pelos dois modelos sem abrir mão das vantagens oferecidas pelos WebSockets [11].

Outro ponto de precaução que deve ser observado no desenvolvimento da aplicação é o suporte de *proxies* e intermediários ao protocolo WebSocket. Devido a algumas políticas restritas de segurança, alguns servidores *proxy* ou outros intermediários podem não suportar o funcionamento do protocolo WebSocket, levando a uma série de casos de falha como *upgrade* de conexão às cegas, *buffer* indesejado de quadros de mensagens, modificações no conteúdo de mensagens sem o conhecimento do protocolo e, principalmente, classificação errônea da conexão através do WebSocket como tentativa de conexão HTTP insegura [11].

Apesar do uso da chave secreta no *handshake* de abertura minimizar a questão do *upgrade* às cegas em *proxies* explícitos, ou seja, *proxies* cujas existências são de conhecimento do navegador do usuário, a solução não resolve o problema em *proxies* transparentes, que ficam ocultos para o cliente, que podem

analisar ou modificar dados na conexão sem aviso. Umas das soluções defendidas para contornar esse problema é o uso de túneis seguros fim-a-fim ou, no caso dos WebSockets, o uso do protocolo *wss://*, pois ao negociar uma camada segura de transporte, SSL/TLS (*Secure Socket Layer*)/(T*ransport Layer Security*), ao executar o *upgrade* de conexão, o cliente e o servidor estabelecem um túnel encriptado que resolve uma série dos casos descritos principalmente para clientes móveis cujo tráfego normalmente passa através de uma série de *proxies* que podem não suportar o protocolo [11].

O entendimento das particularidades e restrições pertinentes ao modelo WebSocket permite ao desenvolvedor melhorar o desempenho de sua aplicação web. Para entender o impacto de sua utilização em conjunto com o Node.js para aplicações em tempo real, testes serão realizados de forma a estimar o desempenho e escalabilidade do conjunto em um sistema.

4 AVALIAÇÃO DO CONJUNTO NODE.JS E WEBSOCKETS

Nas seções anteriores foram discutidos as vantagens e desvantagens da utilização do Node.js, levando em consideração suas características como a programação orientada a eventos e as aplicações que se beneficiariam da implementação de WebSockets. Como a discussão inicialmente teve uma abordagem teórica, nas próximas seções tais teorias serão avaliadas. Para avaliação do desempenho da arquitetura estudada, duas séries de testes serão conduzidas. O primeiro objetivo é avaliar como o Node.js se comporta sob situações de alta carga e concorrência, assim como delimitar o seu desempenho quando são realizadas requisições HTTP normais. A segunda etapa tem por objetivo entender a diferença entre o desempenho da arquitetura Node.js em conjunto com WebSockets em comparação com outras soluções de comunicação bidirecional, basicamente requisições HTTP.

O ambiente de teste foi implementado em um computador com processador Intel i7 5500U de 2,4GHz, 8GB DDR3 de memória RAM e sistema operacional Ubuntu 16.04LTS. Para os testes foram utilizados os servidores Node.js versão 6.9.1 e Apache 2.4.18, além do módulo PHP 7.0. Já os testes foram realizados a partir de outro computador com processador Intel i3 3110M de 2,4GHz, memória RAM de 4GB DDR3 e sistema operacional Windows 10 Home Edition através de uma rede doméstica. Os testes de carga foram executados no *software* ApacheBench 2.4.23.

É importante salientar que evitou-se configurar melhorias e otimizações em todos os servidores e softwares de teste para não haver interferência nos resultados, assim sendo, todos os servidores serão testados “*out-of-the-box*”. A melhoria de desempenho é atingível em todos os servidores em maior ou menor escala e dependem de diversos fatores como maturidade da plataforma, arquitetura e, principalmente, finalidade da aplicação para a qual se destina sua instalação. Portanto, realizar o *benchmark* após melhorias arbitrárias seria prejudicial ao resultado final.

Também é importante mencionar que o uso de uma rede doméstica Wi-Fi também tem influência, pois adiciona a latência da rede aos resultados. Para reduzir potenciais discrepâncias originadas por falhas na conexão ou redução na qualidade de sinal, cada cenário de todos os testes foi repetido ao menos 3 vezes, sendo o resultado obtido a média aritmética entre cada um dos valores obtidos.

A escolha pelo conjunto Apache+PHP para comparação com o Node.js se deve ao fato de que o PHP é a linguagem *server-side* mais utilizada para construção de sites no mundo [32], além de ser livre e de código aberto, assim como Node.js. O uso do Apache é obvio, por fazer parte do *stack* mais comum para uso da linguagem, no caso o LAMP (Linux, Apache, MySQL e PHP) ou WAMP quando utilizando em distribuição Windows.

4.1 AVALIAÇÕES DE DESEMPENHO POR PLATAFORMA

A primeira bateria de testes tem como objetivo entender como o desempenho Node.js se coloca frente ao conjunto Apache e PHP. Para isso serão analisados os resultados de requisições em diferentes cenários: primeiro uma página simples, com o clássico exemplo “Olá mundo”. Depois um cenário de alto custo computacional representado pelo cálculo recursivo do trigésimo número na sequência de Fibonacci. Finalmente, um teste em que requisições disparem operações de E/S, realizado com um arquivo de dados para leitura com o objetivo de avaliar a diferença gerada pela natureza assíncrona das operações de E/S do Node.js [27]. Em todos os casos foram feitas rajadas de requisições concorrentes. Portanto, em um cenário de 10.000 requisições sendo 5.000 concorrentes, foram feitas duas rajadas para se alcançar o total de requisições desejado.

4.1.1 TESTE “OLÁ MUNDO”

Na Figura 11, é demonstrada a implementação do código e Node.js utilizado no primeiro teste. O intuito é gerar um servidor HTTP e responder à requisição com uma mensagem “Olá Mundo”.

```
var http = require('http');
http.createServer(function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Olá Mundo!');
}).listen(3000);
```

Figura 11: Exemplo de código Node.js para o teste "Olá Mundo"

Na Figura 12, é demonstrada a implementação do código em PHP, que resulta em uma resposta idêntica ao Node.js.

```
<?php
    echo "Olá Mundo!";
?>
```

Figura 12: Exemplo de código PHP para o teste "Olá Mundo"

É importante salientar que, apesar de mais verboso, o código da implementação em Node.js não somente produz a resposta desejada, mas também é responsável por criar um servidor que escutará por requisições e as responderá.

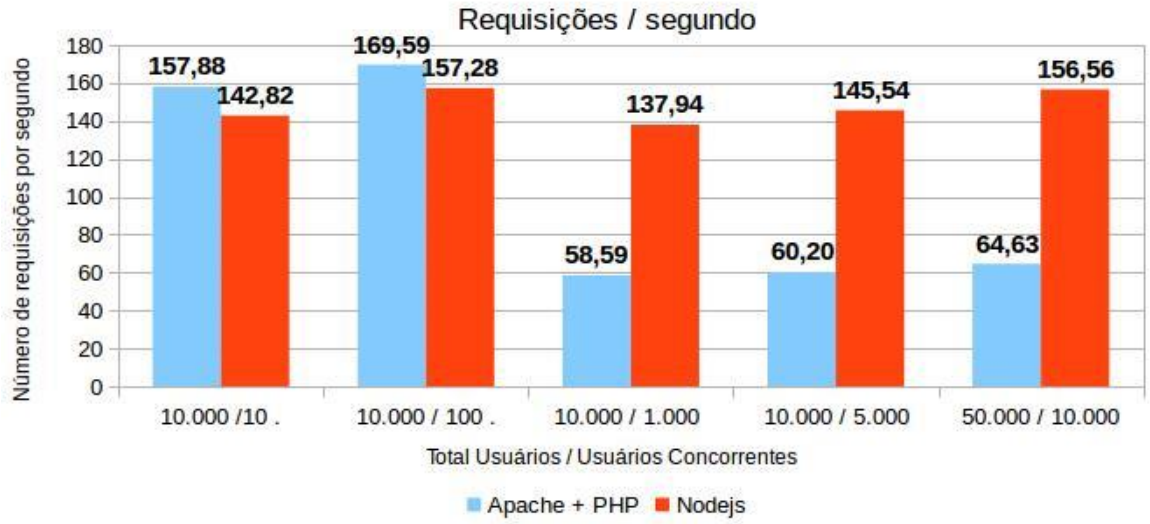

Gráfico 1: Requisições por segundo no cenário "Olá Mundo"

A partir do Gráfico 1, é possível notar a degradação no desempenho sofrido pelo conjunto Apache + PHP conforme o número de usuários simultâneos se eleva. Apesar de apresentar um desempenho menor que a pilha Apache + PHP com uma taxa reduzida de concorrência, o Node.js se mostrou mais estável mesmo quando submetido a um grande número de requisições simultâneas. Da mesma forma, o Gráfico 2 demonstra que o tempo médio de resposta às requisições apontam a mesma conclusão, sendo o tempo de resposta do Node.js 2 vezes mais rápido que o do conjunto Apache + PHP em cenários de alta concorrência.

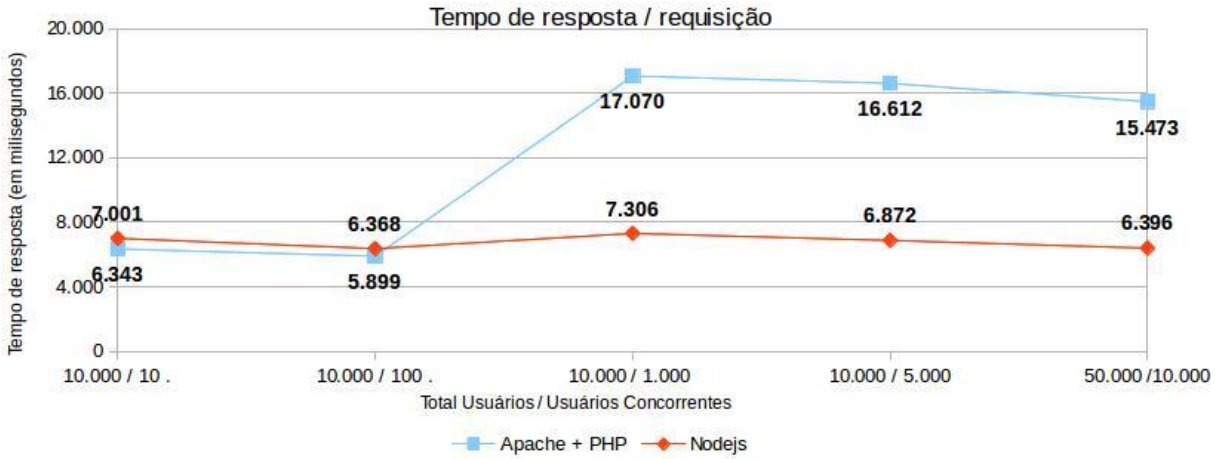

Gráfico 2: Tempo médio de resposta em milissegundos no cenário “Olá Mundo”

4.1.2 TESTE FIBONACCI

O cálculo dos números de Fibonacci é um problema matemático recorrente para o aprendizado da programação. A escolha da versão recursiva do algoritmo para tal cálculo se deve à complexidade exponencial comparada aos algoritmos iterativos lineares. Dessa forma, é possível submeter os ambientes testados a tarefas de elevado custo computacional. Ambos os algoritmos foram implementados de forma a retornar o trigésimo número da sequência de Fibonacci através de recursividade.

A Figura 13 demonstra a implementação do algoritmo de cálculo em Node.js, incluindo o código para estabelecer um servidor que responde através da porta 3000.

```
var http = require('http');
http.createServer(function(req, res) {
    function fibonacci(n) {
        if (n ==1 || n == 2){
            return 1;
        } else {
            return fibonacci(n-1) + fibonacci(n-2);
        }
    }

    res.writeHead(200, {'Content-Type': 'text/plain'});
    var num30Fibonacci = fibonacci(30);
    res.end(num30Fibonacci.toString());
}).listen(3000);
```

Figura 13: Implementação em Node.js do cálculo do vigésimo número da sequência de Fibonacci

Na Figura 14, a implementação da função de Fibonacci para retorno do trigésimo número da sequência em PHP.

```
<?php

    function fibonacci($n) {
        if ($n ==1 || $n == 2){
            return 1;
        } else {
            return fibonacci($n-1) + fibonacci($n-2);
        }
    }

    echo(fibonacci(30));
?>
```

Figura 14: Implementação em PHP do cálculo do vigésimo número da sequência de Fibonacci

Os testes utilizando a sequência de Fibonacci tiveram resultados até certo ponto surpreendentes, pois a plataforma Node.js, em seu início, era criticada por não ser capaz de lidar com tarefas de alto processamento com desempenho satisfatório. Os resultados no Gráfico 3, indicam que a equipe por trás do desenvolvimento do

Node.js está ativamente trabalhando na sua melhoria a cada versão, apresentando um grande salto desde sua versão 0.12.

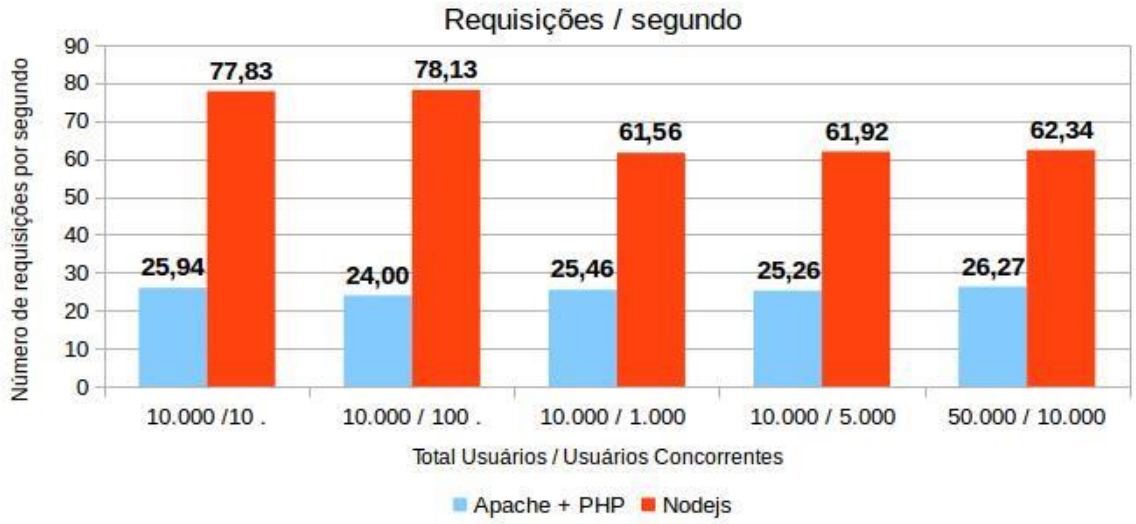

Gráfico 3: Requisições por segundo no cenário de Fibonacci

Novamente, o desempenho do Node.js foi mais que 2 vezes superior em todos os cenários testados, incluindo quando havia pouca concorrência. O tempo de resposta do conjunto Apache + PHP também foi mais de 2 vezes mais lento que o Node.js, sendo em alguns casos até mais de 3 vezes mais lento, segundo o Gráfico 4.

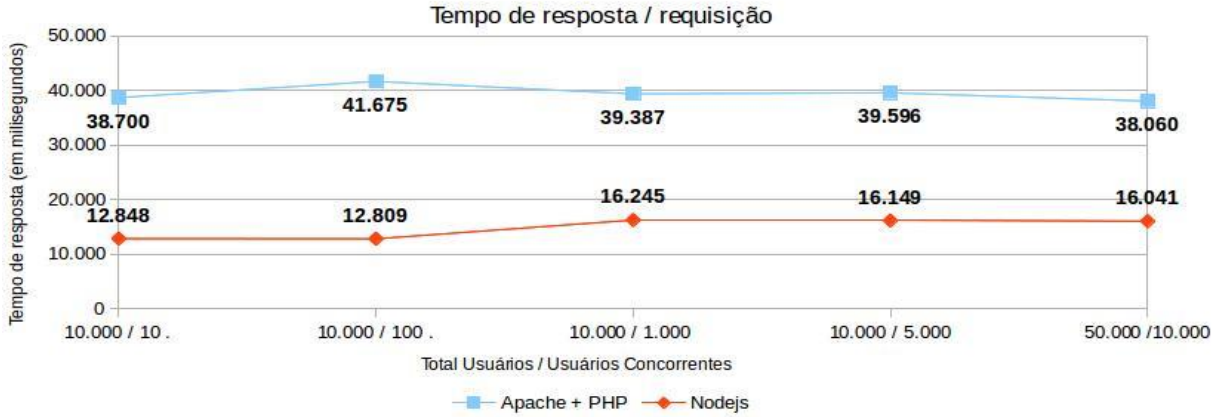

Gráfico 4: Tempo médio de resposta em milissegundos no cenário de Fibonacci

4.1.3 TESTE DE E/S

Para o teste de E/S, foi criado um arquivo de texto composto de 100.000 linhas contendo *strings* de tamanho e conteúdo aleatório. O teste realizado consiste na leitura do conteúdo desse arquivo e exibição de seu conteúdo na página retornada pelo servidor. Assim, espera-se entender o impacto das operações de I/O para a escalabilidade em sistemas web, e, particularmente, como o Node.js se comporta com E/S assíncrona. Para o teste, optou-se pela não utilização de banco de dados, pois o que se deseja é verificar o desempenho puro e simples dos *stacks* relacionados sem a adição de complexidades e/ou gargalos provenientes da escolha de determinado banco de dados ou modelo relacional ou não relacional.

O arquivo *textfile.txt* utilizado para o teste contém 100.000 linhas, em que cada linha é composta por 10 sequências de 10 caracteres aleatórios, com um trecho específico "*Foobarbazz"*, que deverá ser encontrado e retornado para o cliente, como exemplificado no trecho da imagem 14:

```
adhtxyxmub eplzneyhhz adhtxyxmub eplzneyhhz … (10 colunas no total)
msefvicrob vqlnlzjqfg msefvicrob vqlnlzjqfg … (10 colunas no total)
djmyaiqfug rbftqzsxln Foobarbazz rbftqzsxln … (10 colunas no total)
bummuehapl dcauprhfqi bummuehapl dcauprhfqi … (10 colunas no total)
tiyfwgcnbt stywzcsnsr tiyfwgcnbt stywzcsnsr … (10 colunas no total)
```

Figura 15: Trecho do conteúdo do arquivo utilizado no teste de I/O

A figura 16 demonstra a implementação do código em Node.js para a realização do teste.

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res) {
    fs.readFile('textfile.txt', 'UTF-8', function(err, data) {
        if (err) {
            return console.log(err);
        }
        res.writeHead(200, {'Content-Type': 'text/plain'});
        var idx = data.indexOf('Foobarbazz');
        if (idx !== -1) {
            res.write( data.substr(idx, 10) );
        }
        res.end();
    });
}).listen(3000);
```

Figura 16: implementação do código em Node.js para teste de I/O

A título de entendimento sobre impacto do funcionamento assíncrono pregado pelo Node.js, nesse teste também foi efetuada a comparação com a versão de leitura síncrona do próprio Node.js através da função *readFileSync.* Na figura 17, a implementação do código de teste de I/O em PHP.

```
<?php
    $content = file_get_contents("textfile.txt");
    $idx = strpos($content, "Foobarbazz");
    echo substr($content, $idx, 10);
?>
```

Figura 17: implementação do código em PHP para teste de I/O

A expectativa de melhor performance do Node.js em cenários de elevado número de requisições simultâneas foi confirmada com exibição de tendência semelhante no Gráfico 5, se comparado ao Gráfico 1.

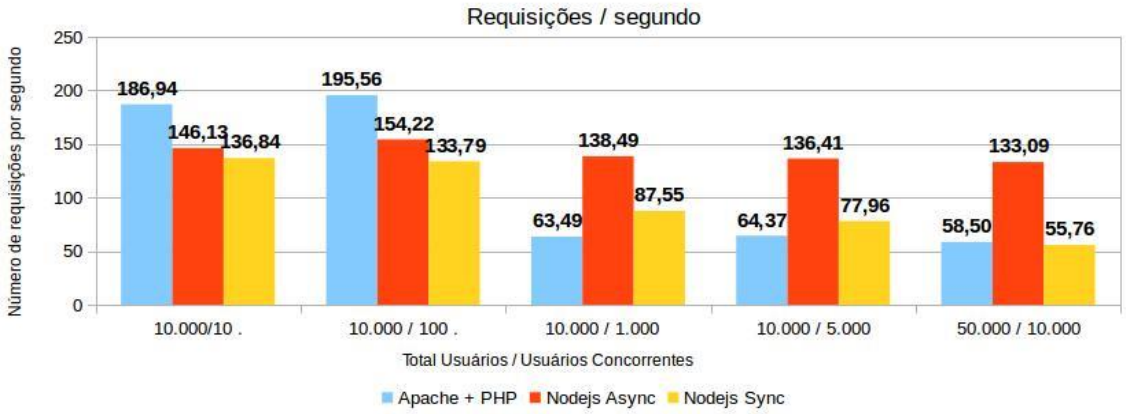

Gráfico 5: Requisições por segundo no cenário de I/O

Como no teste “Olá Mundo”, o desempenho do conjunto Apache + PHP se degradou devido à elevada concorrência, enquanto o Node.js manteve-se estável em todos os casos, devido ao uso de uma *thread* não bloqueante que delega as tarefas de E/S para *threads* auxiliares. O teste serviu também para mostrar que, mesmo sendo a plataforma otimizada para funções de E/S, sua utilização de forma equivocada, realizando leituras síncronas e bloqueantes, prejudica seu desempenho. Utilizando a versão síncrona da função de leitura de arquivos, o desempenho do Node.js foi bastante semelhante ao Apache + PHP, degradando seu desempenho de forma constante. O tempo de resposta exibido pelo Gráfico 6 demonstra o impacto gerado pelo uso de leituras bloqueantes.

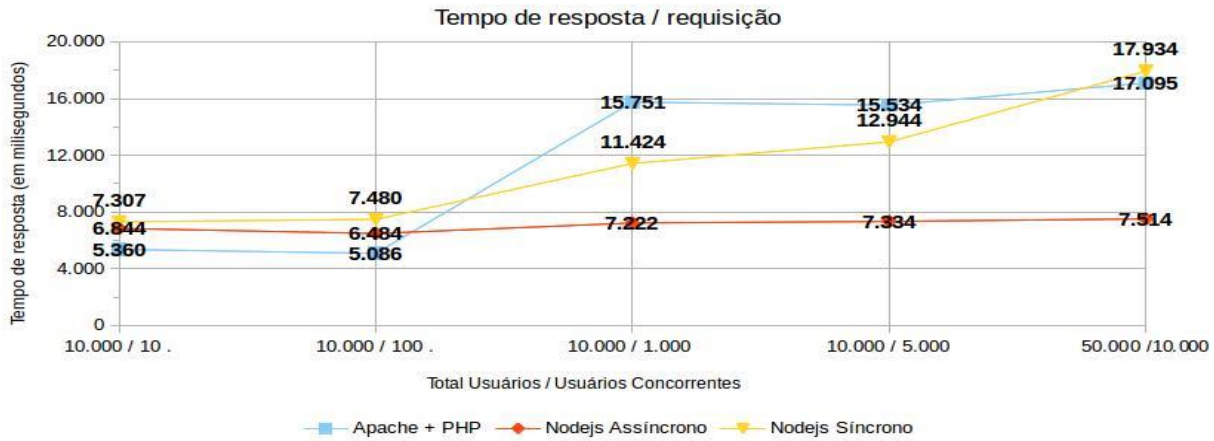

Gráfico 6: Tempo médio de resposta em milissegundos no cenário de I/O

Além disso, é importante notar o impacto que o paradigma de orientação a eventos tem influência nos resultados. O uso de um *loop* de eventos que delega funções de leitura e escrita a *threads* auxiliares prova-se estável em cenários de alta concorrência e, consequentemente, é um fator chave para considerações a respeito da escalabilidade da aplicação a ser desenvolvida.

4.2 COMPARATIVO ENTRE LONG-POLLING E WEBSOCKETS

Com um entendimento mais profundo sobre as capacidades e restrições do uso do servidor Node.js em uma aplicação *web*, é possível fazer um paralelo do impacto no desempenho gerado pelo tipo de transporte utilizado. Para isso, foram realizados testes de carga em um mesmo servidor em Node.js que opera como servidor de eco, em que um cliente se conecta e envia uma mensagem que será respondida com a mesma mensagem pelo servidor, e o processo se repete indefinidamente. O teste tem por objetivo verificar o volume de mensagens que pode ser transmitida e processada em cada modelo de transporte, além de verificar o consumo de recursos computacionais por ambos

Para realização dos testes foram montados dois geradores de clientes em Node.js que fazem as conexões no servidor utilizando os protocolos de transporte escolhidos para o teste: o primeiro utiliza o protocolo websocket e o segundo realiza a conexão utilizando a técnica de *long-polling* através de requisições XHR. Para monitoramento do estado do processador e memória do servidor, foi utilizada a ferramenta PM2 desenvolvido pela Keymetrics. Com ele é possível analisar consumo de processamento e memória em tempo real de cada processo gerado pelo Node.js no servidor.

Gráfico 7: Número médio de mensagens transmitidas por segundo

O volume de mensagens transmitidas por segundo pelo servidor apresentado pelo Gráfico 7 é bastante claro na velocidade que o servidor consegue atender e repassar as mensagens utilizando cada modelo de transporte. É importante notar que, enquanto o volume de mensagens transmitidas aumentava de acordo com o número de usuários conectados simultaneamente usando websockets, o mesmo não foi verificado utilizando o XHR *polling*, que apresentou uma redução no número de mensagens que podiam ser processadas e repassadas. Isso se deve ao uso das requisições sucessivas que acabam gerando uma sobrecarga no servidor.

Gráficos 8: Mensagens recebidas pelo servidor em relação ao número de usuários por segundo

Como esperado, o desempenho em ambos os modelos sofre uma degradação decorrente do alto número de usuários simultâneos. Mesmo assim, o desempenho entregue pelos websockets prova-se extremamente superior em casos de menor concorrência e consegue processar cerca de 10 vezes mais mensagens do que o método de *polling*, tendência exibida no Gráfico 8. Como observação, durante um dos testes realizados com o método de *polling*, o servidor não suportou a carga e desconectou gradativamente todos os usuários no cenário de 5.000 conexões simultâneas. O cliente de testes que utilizava websockets, entretanto, não apresentou anomalias durante a realização dos testes.

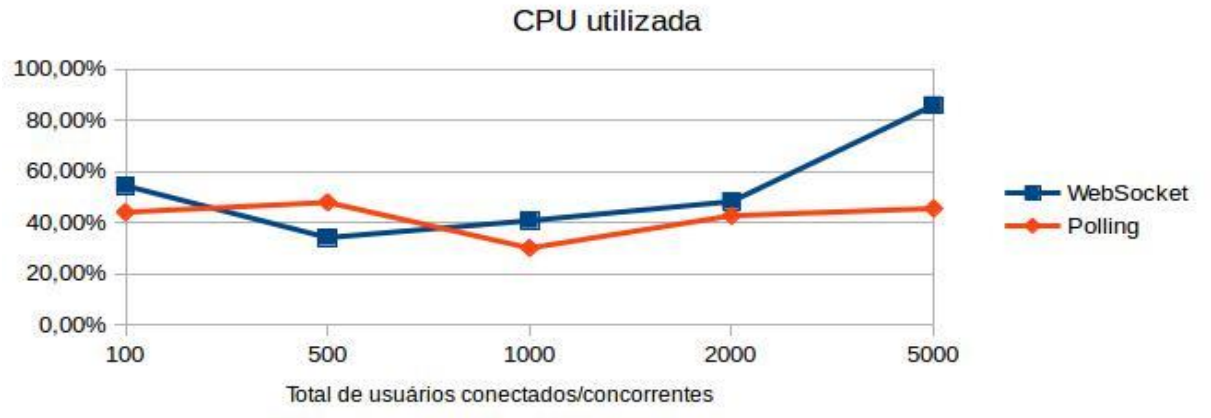

Gráfico 9: Uso do processador pela aplicação

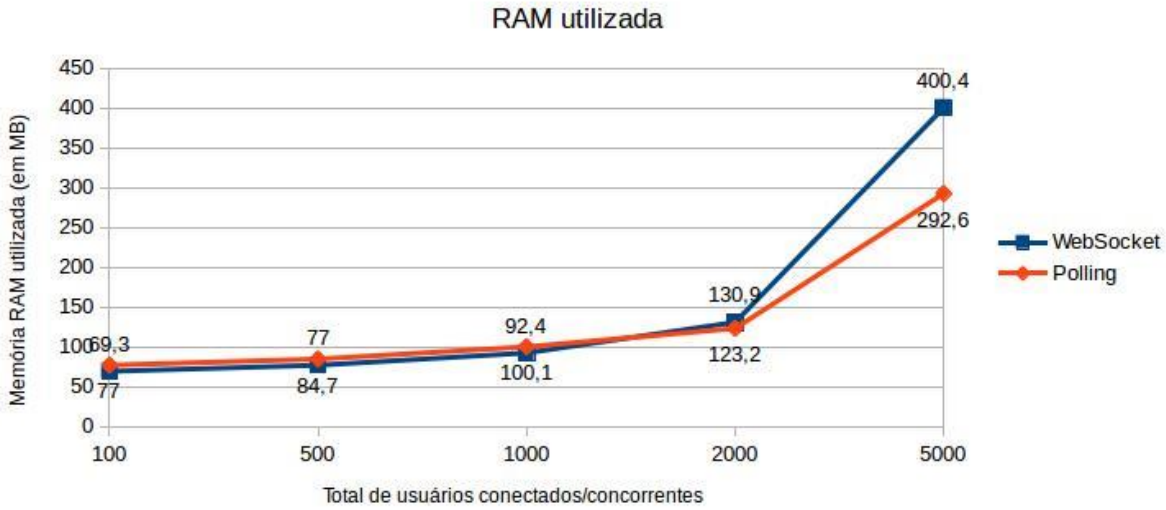

Gráfico 10: Uso de memória RAM pela aplicação

O teste demonstrou também que, apesar de oferecer um desempenho significativamente superior, o uso de websockets demanda muito mais o uso dos recursos computacionais como mostrado nos gráficos 9 e 10. Em alguns momentos específicos o uso do processador chegou a 98% na utilização dos websockets. Ao desenvolver uma aplicação que utilize esse modelo de transporte é importante estar atento para que o servidor esteja adequado para que o uso da CPU não seja um gargalo.

5 CONCLUSÃO

Por meio deste estudo, fez-se uma análise do uso conjunto do Node.js e WebSockets no desenvolvimento de aplicações que dependam de interações em tempo real entre os diversos clientes conectados. Através do detalhamento teórico da plataforma Node.js, foi possível compreender as particularidades de sua implementação como a orientação a eventos e as operações assíncronas, que permitem à plataforma suportar um número maior de usuários simultâneos quando comparado ao conjunto Apache e PHP. Os resultados obtidos apontam que o uso do Node.js apresenta desempenho superior ao oponente quando submetido a um número elevado de usuários concorrentes, mantendo a estabilidade do sistema, com baixa degradação no tempo de resposta.

O crescimento da comunidade de desenvolvedores que contribuem para o ecossistema Node.js tem alavancado a adoção da plataforma em sistemas cada vez mais complexos e bem sucedidos. Empresas como LinkedIn, Walmart, PayPal, Uber e Netflix já utilizam soluções desenvolvidas em Node.js em produção[16]. Algumas dessas empresas possuem produtos com grande volume de conexões simultâneas, centenas de milhares de requisições por minuto e são referências em seus mercados de atuação.

Importante ressaltar que um dos maiores entraves para o crescimento na adoção da plataforma é a natureza do seu paradigma. Bastante diferente do modelo atual de requisição-resposta baseado em *threads*, a orientação a eventos ainda não é bem compreendida por muitos desenvolvedores oriundos de outras linguagens, como Java, Python ou o próprio PHP, trazendo consigo costumes que, quando replicados no ambiente Node.js, produzem códigos que ferem as diretrizes de seu desenvolvimento e degradam o seu desempenho. No teste realizado para leitura de arquivos externos utilizando os módulos síncrono e assíncrono, foi possível dimensionar o quanto o uso de estruturas não recomendadas pode impactar no desempenho que se espera da plataforma. Nesse sentido, ações que ajudem a esclarecer a metodologia da plataforma, como o site *callback hell*, citado neste estudo, são fundamentais para ajudar no crescimento de seu ecossistema.

Finalmente, o entendimento do funcionamento dos WebSockets e das opções utilizadas até então para mimetizar a comunicação bidirecional entre cliente e servidor, foi importante para esclarecer em que cenários sua utilização é recomendada. Através dos testes realizados, foi possível perceber que o volume de mensagens trafegadas e processadas é consideravelmente maior do que o método de XHR Polling, um dos mais tradicionais utilizados. Como reflexo, pode-se dizer que a conjunção do Node.js com WebSockets pode render um desempenho bastante satisfatório para aplicações e tempo real, unindo escalabilidade e velocidade de entrega de informações.

O estudo pode servir de ponto de partida para testes de otimização do conjunto buscando atingir um determinado volume de informações trafegadas e processadas entre servidor e clientes ou economia de recursos do servidor quando submetido a altas cargas. Apesar de serem duas tecnologias relativamente recentes, se comparadas às suas alternativas listadas ao longo do texto, estudos e materiais para uso em desenvolvimento têm sido produzidos em ritmo acelerado, o que não só contribui para o amadurecimento de ambas, mas principalmente apontam para possibilidades de aplicação ainda mais promissoras.

REFERÊNCIAS BIBLIOGRÁFICAS

[1]. AGUIAR, Gustavo Stor. Node.js: estudo tecnológico e desenvolvimento full-stack javascript de plataforma de competições em problemas algorítmicos. Recife, 2015. 66 p.

[2]. ASYNC. Disponível em: <http://caolan.github.io/async/docs.html>. Acesso em: 12 out. 2016.

[3]. BAK, Lars. Google Chrome's Need for Speed. 2008. <http://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html>. Acesso em: 04 abr. 2016.

[4]. CONROD, Jay. A tour of V8: Crankshaft, the optimizing compiler. 2013. Disponível em: <http://jayconrod.com/posts/54/aFtourFofFv8FcrankshaftFtheFoptimizingFcompiler>. Acesso em: 08 abr. 2016.

[5]. CONROD, Jay. A tour of V8: full compiler. 2012. Disponível em: <http://jayconrod.com/posts/51/a-tour-of-v8-full-compiler>. Acesso em: 08 abr. 2016.

[6]. CONROD, Jay. A tour of V8: garbage collection. 2013. Disponível em: <http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection>. Acesso em: 10 abr. 2016.

[7]. DAHL, Ryan. Deep inside Node.js with Ryan Dahl. [dez. 2010]. Entrevistador: Dio Synodinos. 2010. Entrevista concedida ao portal InfoQ. Disponível em: <http://www.infoq.com/interviews/node-ryan-dahl> Acesso em 02 abr. 2016.

[8]. DAHL, Ryan. Node.js. JSConf, 2009. Disponível em: <http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html>. Acesso em: 02 abr. 2016.

[9]. EXPRESS.JS. Disponível em: <http://expressjs.com/>. Acesso em 12 out. 2016.

[10]. FETTE, Ian & MELKINOV, Alexey. The WebSocket Protocol. RFC 6455, Internet Engineering Task Force, 2011. Disponível em: <https://tools.ietf.org/html/rfc6455>. Acesso em 28 out. 2016.

[11]. GRIGORIK, Ilya. High Performance Browser Networking. 2013. O'Reilly Media. Disponível em: <https://hpbn.co/websocket/>. Acesso em: 30 out. 2016.

[12]. HARRIS, Amber. The birth of Node: where did it come from? Creator Ryan Dahl shares the history. 2013. Disponível em: <http://siliconangle.com/blog/2013/04/01/the-birth-of-node-where-did-it-come-from-creator-ryan-dahl-shares-the-history/> Acesso em 03 abr. 2016.

[13]. LIBUV. Disponível em: <http://docs.libuv.org/en/v1.x/design.html>. Acesso em: 18 abr. 2016.

[14]. LODASH.JS .Disponível em: <https://lodash.com/>. Acesso em 18 out. 2016.

[15]. LUBBERS, Peter & GRECO, Frank. HTML5 WebSocket: A Quantum Leap in Scalability for the Web. Disponível em: <https://www.websocket.org/quantum.html>. Acesso em: 30 out. 2016.

[16]. MAIDÍN, Cian. Why Node.JS is becoming the go-to technology in the enterprise. 2014. Disponível em: <http://www.nearform.com/nodecrunch/node-js-becoming-go-technology-enterprise/>. Acesso em 02 abr. 2016.

[17]. MONGODB. Disponível em: <http://mongodb.github.io/node-mongodb-native/>. Acesso em: 12 out. 2016.

[18]. NODE.JS v5.10.1 DOCUMENTATION. Disponível em: <https://nodejs.org/dist/latest-v5.x/docs/api/addons.html>. Aceso em: 18 abr. 2016.

[19]. NODE.JS. Disponível em: <http://nodejs.org/>. Acesso em: 02 abr. 2016.

[20]. NODEMAILER. Disponível em: <https://nodemailer.com/>. Acesso em: 12 out. 2016.

[21]. NPM MOST DEPENDED UPON PACKAGES. Disponível em: <https://www.npmjs.com/browse/depended#product-navigation/>. Acesso em: 12 out. 2016.

[22]. NPM. Disponível em: <https://docs.npmjs.com/getting-started/what-is-npm>. Acesso em: 30 set. 2016.

[23]. OGDEN, Max. Callback Hell. 2012. Disponível em: <http://callbackhell.com/>. Acesso em 15 set. 2016.

[24]. OLIVEIRA, George Souza; SILVA, Anderson Faustino. Compilação Just-In-Time: Histórico, Arquitetura, Princípios e Sistemas. 2013.

[25]. RIBEIRO, Francisco de Assis. Programação Orientada a Eventos no lado do servidor utilizando Node.JS. Fortaleza, 2012.

[26]. SOCKET.IO. Disponível em: <http://socket.io/>. Acesso em: 12 out. 2016.

[27]. TEIXEIRA, Pedro. Professional Node.JS: Building JavaSript based scalable software. Indiana: John Wiley & Sons, Inc. 2013

[28]. UBL, Matle & KITAMURA, Eiji. Introducing WebSockets: Bringing Sockets to the Web. 2010. Disponível em: <https://www.html5rocks.com/en/tutorials/websockets/basics/>. Acesso em: 30 out. 2016.

[29]. UNDERSCORE.JS. Disponível em: <http://underscorejs.org/>. Acesso em 12 out. 2016.

[30]. V8 ENGINE. Disponível em: <https://developers.google.com/v8/design>. Acesso em: 10 abr. 2016.

[31]. V8 ENGINE. Disponivel em: <https://developers.google.com/v8/intro>. Acesso em: 04 abr. 2016.

[32]. W3TECH. Usage of server-side programming languages for websites. 2016. Disponível em: <https://w3techs.com/technologies/overview/programming_language/all>. Acesso em: 20 nov. 2016

APÊNDICE

APLICAÇÃO DE SERVIDOR PARA COMPARATIVO

```
var profiler = require('v8-profiler');
var io = require('socket.io').listen(3000);
var exec = require('child_process').exec;
var getCpuCommand = "ps -p " + process.pid + " -u | grep " + process.pid;
var users = 0;
var countReceived = 0;
var countSended = 0;

function roundNumber(num, precision) {
  return parseFloat(Math.round(num * Math.pow(10, precision)) /
Math.pow(10, precision));
}
setInterval(function() {
  var auxReceived = roundNumber(countReceived / users, 1);
  var msuReceived = (users > 0 ? auxReceived : 0);
  var auxSended = roundNumber(countSended / users, 1);
  var msuSended = (users > 0 ? auxSended : 0);
  var child = exec(getCpuCommand, function(error, stdout, stderr) {
    var s = stdout.split(/\s+/);
    var cpu = s[2];
    var memory = s[3];
    var l = [
      'CUsers: ' + users,
      'MReceived/S: ' + countReceived,
      'MSended/S: ' + countSended,
      'MReceived/S/User: ' + msuReceived,
      'MSended/S/User: ' + msuSended,
      'CPU: ' + cpu,
      'Mem: ' + memory
    ];
    console.log(l.join(',\t'));
    countReceived = 0;
```

```
    countSended = 0;
  });
}, 1000);
io.sockets.on('connection', function(socket) {
  users++;
  socket.on('message', function(message) {
    socket.send(message);
    countReceived++;
    countSended++;
  });
  socket.on('disconnect', function() {
    users--;
  })
});
```

APLICAÇÃO DE CRIAÇÃO DE CLIENTES WEBSOCKETS

```
var profile = require('v8-profiler');
var io = require('socket.io-client');
var message = "o bispo de constantinopla nao quer se desconstantinopolizar";
var argvIndex = 2;
var transport = 'websocket';
var users = parseInt(process.argv[argvIndex++]);
var rampUpTime = parseInt(process.argv[argvIndex++]) * 1000;
var newUserTimeout = rampUpTime / users;
var host = process.argv[argvIndex++] ? process.argv[argvIndex - 1] : 'localhost';
var port = process.argv[argvIndex++] ? process.argv[argvIndex - 1] : '3000';

function user(transport, shouldBroadcast, host, port) {
  var socket = io.connect('http://' + host + ':' + port, {'forceNew': true, transports: [transport]});
  socket.on('connect', function() {
      socket.send(message);
    }
    socket.on('message', function(message) {
      socket.send(message);
    });
    socket.once('disconnect', function() {
      socket.connect();
    });
  });
};
for(var i=0; i<users; i++) {
  setTimeout(function() { user(transport, host, port); }, i * newUser-Timeout);
};
```

APLICAÇÃO DE CRIAÇÃO DE CLIENTES XHR-POLLING

```
var profile = require('v8-profiler');
var io = require('socket.io-client');
var message = "o bispo de constantinopla nao quer se desconstantinopolizar";
var argvIndex = 2;
var transport = 'polling';
var users = parseInt(process.argv[argvIndex++]);
var rampUpTime = parseInt(process.argv[argvIndex++]) * 1000;
var newUserTimeout = rampUpTime / users;
var host = process.argv[argvIndex++] ? process.argv[argvIndex - 1] : 'localhost';
var port = process.argv[argvIndex++] ? process.argv[argvIndex - 1] : '3000';

function user(transport, shouldBroadcast, host, port) {
  var socket = io.connect('http://' + host + ':' + port, {'forceNew': true, transports: [transport]});
  socket.on('connect', function() {
      socket.send(message);
    }
    socket.on('message', function(message) {
      socket.send(message);
    });
    socket.once('disconnect', function() {
      socket.connect();
    });
  });
};
for(var i=0; i<users; i++) {
  setTimeout(function() { user(transport, host, port); }, i * newUser-Timeout);
};
```