INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE – IFSUL, *CÂMPUS* PASSO FUNDO
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

FERNANDO LUIS BASSO

UM ESTUDO EXPERIMENTATIVO E COMPARATIVO DA PLATAFORMA NODE.JS

PASSO FUNDO, 2014

FERNANDO LUIS BASSO

UM ESTUDO EXPERIMENTATIVO E COMPARATIVO DA PLATAFORMA NODE.JS

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-Rio-Grandense, *Campus* Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador: Élder Francisco Fontana Bernardi

PASSO FUNDO, 2014

TÍTULO DO TCC

Trabalho de Conclusão de Curso aprovado em ____/____/____ como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

_____

Nome do Professor(a) Orientador(a)

_____

Nome do Professor(a) Convidado(a)

_____

Nome do Professor(a) Convidado(a)

_____

Coordenação do Curso

PASSO FUNDO, 2014

DEDICATÓRIA

Aos professores,
por terem pacientemente compartilhado
seus conhecimentos.

Ao meu amigo Victor Sebben, por ter me
encorajado e me ajudado durante todo o curso.

Ao meu orientador Élder Francisco Fontana Bernardi
por me guiar durante o processo de elaboração deste trabalho.
enriquecendo sobremaneira o resultado final.

EPÍGRAFE

"Uma linguagem que não influencia na
maneira que pensamos sobre programação
não vale a pena ser aprendida."
Alan J. Perlis

RESUMO

O presente documento é um estudo sobre a plataforma Node.js que visa tornar possível criar e implantar aplicações de rede em tempo real, rápidas e escaláveis. A pesquisa cobre os aspectos principais da plataforma Node.js, incluindo a linguagem de programação JavaScript, que é a base da tecnologia, diferenças entre utilização de múltiplas *threads* versus o *event loop*, operações I/O que não bloqueiam a execução, exemplos e testes de avaliação comparando Node.js com as linguagens Java e PHP.

Palavras-chave: thread. teste de avaliação. event loop.

ABSTRACT

The present document is a study of the Node.js plataform which aims to make it possible to create and deploy fast, realtime, scalable network applications. The paper covers the main aspects of the Node.js plataform, including the JavaScript programming language, which is the basis of the technology, the difference between using multiple threads versus the event loop, non-blocking I/O, examples and benchmark tests comparing Node.js against the PHP and Java Programming languages.

Keywords: thread. event loop. benchmark;

LISTA DE TABELAS

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

A Internet vem, cada vez mais, deixando de ser um meio cujo objetivo é o mero compartilhamento de documentos eletrônicos, e se tornou uma plataforma de transmissão de multimídia e entrega de aplicações completas em tempo real com as mais diversas finalidades.

Com o crescimento quase que imensurável da Internet e o grande número de dispositivos com acesso à rede, há, hoje, uma demanda crescente em se oferecer serviços web responsivos e altamente escaláveis. Para tal, um alto poder de processamento e recursos computacionais é exigido dos servidores de modo geral.

Buscando aperfeiçoar a maneira com a qual aplicações Web são desenvolvidas e implementadas, empresas, organizações e desenvolvedores em geral criam novas ideias, metodologias e tecnologias. Algumas dessas novas criações atingem um considerável nível de sucesso e aceitação e são largamente utilizadas no mercado ao passo que outras ficam obsoletas rapidamente. Este trabalho é um estudo sobre uma dessas tecnologias: Node.js.

A plataforma Node.js, foi desenvolvida com o intuito de suprir às demandas citadas. Na página inicial do site oficial do projeto, pode-se ler a seguinte definição:

> Node.js é uma plataforma construída sobre o *runtime* JavaScript V8 para a fácil implementação de aplicações de redes rápidas e escaláveis. Node.js utiliza um modelo orientado a eventos que não bloqueia em operações de entrada e saída (*non-bloking* I/O), o que o torna leve e eficiente, perfeita para aplicações de dados intensivas em tempo real que rodam distribuídas em dispositivos diversos (NODE.JSa, 2014, tradução nossa).

O estudo aqui proposto visa investigar e testar na prática as suas características, explorando conceitos que a tornam diferente dos demais servidores web e linguagens de programação. Testes de avaliação comparativa também são executados sobre algumas aplicações envolvendo Node.js, Java e PHP (cujas escolhas serão explicadas no momento oportuno) de forma a pôr em prova a eficácia da tecnologia (já que Node.js se apresenta como uma alternativa mais eficiente em relação a performance e velocidade ao mesmo tempo em que é mais econômica em relação a recursos computacionais). Também são feitas algumas considerações quanto a experiência de utilização da tecnologia. de modo a avaliar a viabilidade do uso desta plataforma e servir como um ponto de partida para aqueles que, por algum motivo, necessitem ou queiram fazer uso dela.

Node.js surgiu em 2009 e chamou atenção por utilizar a linguagem JavaScript (na

verdade, sua apresentação inicial ocorreu em uma conferência sobre JavaScript[1]), e durante o estudo, foi constatado que um conhecimento mais aprofundado da linguagem JavaScript é necessário para pode utilizar e entender a plataforma Node.js. Portanto, um apanhado dos pontos principais da linguagem JavaScript é também realizado e apresentado na Seção 2.1.

O trabalho é dividido em quatro capítulos: Introdução, Referencial teórico, Avaliação e comparação da plataforma Node.js e Considerações finais. Na Introdução são mencionados os motivos que levaram a execução do trabalho proposto e os objetivos a serem alcançados. O Capítulo dois, Referencial teórico, é dividido em duas grandes seções, uma abordando a linguagem JavaScript focando em Node.js especificamente. O terceiro capítulo descreve o processo dos testes, demonstrando os resultados obtidos e algumas observações a respeito deles. O trabalho é finalizado com o capítulo quatro, onde são feitas algumas considerações gerais sobre o estudo e são apontados alguns possíveis tópicos para pesquisas futuras.

1.1 MOTIVAÇÃO

A Informática é uma área do conhecimento que evolui e muda de maneira extremamente rápida. Tecnologias vão e vem o tempo todo. Algumas permanecem por muito tempo; outras não. Por esse motivo, é imperativo que desenvolvedores, de um modo geral, tomem conhecimento dessas novas tecnologias, conheçam suas características, viabilidade de utilização prática em contextos e aplicações reais, bem como suas particularidades positivas e negativas.

Talvez ainda mais importante seja a necessidade da experimentação em torno dessa tecnologia devido as características que possui. É importante instigar no meio acadêmico e nos profissionais da área da informática um espírito desbravador e curiosidade em relação à plataforma, já que esta é totalmente diferente do paradigma dominante quanto à construção de servidores e aplicações que rodam sobre eles. É curioso, ainda, o fato de que a partir de uma linguagem utilizada até então apenas no lado do cliente, interpretada (não compilada) e considerada de segunda categoria, surja uma plataforma para se construir servidores altamente escaláveis, eficientes e com baixo consumo de recursos[2].

Finalmente, acreditando que a curiosidade e a vontade de aprender são qualidades imprescindíveis a qualquer profissional, mas especialmente os da área informática, onde tudo

---

1 http://jsconf.com/

2 Que é o que o estudo pretende averiguar.

muda tão rapidamente, o presente trabalho busca, também, contribuir para difundir e ampliar o conhecimento dos leitores tanto no meio profissional quanto no meio acadêmico.

1.2 OBJETIVOS

Dando seguimento aos elementos motivacionais mencionados acima, este capítulo apresenta os objetivos gerais e específicos a serem alcançados com a conclusão deste trabalho.

1.2.1 Objetivos gerais

Fornecer conhecimento conceitual e prático sobre a plataforma Node.js.

1.2.2 Objetivos específicos

São quatro os objetivos principais que o trabalho visa alcançar:

- Conhecer os pontos principais da linguagem JavaScript.
- Conceitos sobre a plataforma Node.js.
- Mostrar exemplos práticos de código utilizando a tecnologia.
- Realizar testes de avaliação de desempenho.

2 REFERENCIAL TEÓRICO

Este Capítulo se divide em duas grandes partes: JavaScript e Node.js. A Seção 2.1 sobre JavaScript é um pré-requisito para o tópico principal, Node.js, já que esta é uma tecnologia inteiramente fundamentada nos preceitos e características (*event loop*, eventos, orientada a objetos, funções *callback*, entre outras coisas) da linguagem JavaScript.

Serão abordados, inicialmente, conceitos fundamentais da linguagem. O estudo será focado nas características do JavaScript que a tornam diferentes das demais linguagens utilizadas atualmente em grande escala. Serão deliberadamente omitidos aqueles pontos que se assemelham às linguagens comumente usadas, como Java, PHP e C.

Em seguida, o foco passa totalmente para o estudo da plataforma Node.js propriamente dita, suas características e conceitos, módulos, o *event loop* e outras particularidades e detalhes importantes da plataforma bem como alguns exemplos práticos em código.

2.1 JAVASCRIPT

Nesta Seção, abordar-se-á a história da linguagem JavaScript e são discutidas as principais características da linguagem.

2.1.1 História

Em 1995, a Netscape contratou Brendam Eich[1] para implementar em seu navagador, também chamado Netscape, uma linguagem que possibilitasse executar *scripts* e proporcionar interação do usuário com o navegador. Todo o processo de planejamento e desenvolvimento levou apenas 14 dias. Eich desejava desenvolver um dialeto simplificado de Lisp, mas a Netscape não gostou da ideia, e pediu a ele que criasse algo com sintaxe similar ao Java de modo que fosse algo mais familiar aos programadores de um modo geral. No entanto, Brandan Eich seguiu sua ideia de fazer algo no estilo funcional, mantendo a sintaxe do Java

[1] Engenheiro de microcontroladores do vale do silício na época, um dos criadores da fundação Mozilla, atual membro da direção e CTO (Chief Technology Officer, ou Diretor Técnico) da fundação Mozilla.

para satisfazer a Netscape. O nome inicial da linguagem era LiveScript, pois julgavam que tal nome refletia a natureza da linguagem, que era (e ainda é) fracamente tipada e oferecia interação com o usuário (ou seja, tornava as páginas web mais vivas).

O que aconteceu em seguida foi um acordo entre as empresas Sun e Netscape com o intuito de derrubar a Microsoft. Passaram a incluir Java *applets* no navegador, pois acreditavam que entregar aplicações através de um navegador tornaria a Microsotf obsoleta. Houve um impasse quanto continuar usando JavaScript ou Java *applets,* mas, no final, as duas opções foram mantidas no navegador. Ainda, para manter a aliança saudável, agradar a Sun (proprietária do Java na época) e também para facilitar o *marketing*, o nome acabou mudado para JavaScript (CHAMPEON, 2001), e por motivos legais do acordo entre as duas empresas, o nome JavaScript se tornou marca registrada da Sun (a qual concedeu direito exclusivo do uso do nome para a Netscape), e hoje pertence à Oracle.

Para dar mais credibilidade à linguagem, a Netscape tentou formalizar um padrão através da W3C (*World Wide Web Consortium*), o qual foi recusado[2]. Tentaram também obter auxílio da ISO (*International Organization for Standardization*), mas acabaram apenas conseguindo apoio da *European Computer Manufacteres Association* (ECMA) para consolidar o padrão. A Microsoft, que havia feito engenharia reversa e consegui imitar *bug* por *bug* da versão JavaScript no Netscape estava na primeira reunião do comitê de padronização, e, por muito tempo, ditou as regras da especificação. O nome do padrão aprovado pela ECMA acabou chamado de ECMAScript (já que JavaScript era registrado pela Sun). A Microsoft, por sua vez, usou o nome JScript (CROCKFORD, 2010).

A primeira versão da especificação ECMAScript ocorreu em 1997, seguida pela versão 2 em 1998. O padrão seguinte foi o 3, em 1999 e a versão 4 foi abandonada, passando direto para a versão 5 em 2011. Estão sendo desenvolvidas, atualmente, a versão 6 e 7 do padrão (ECMASCRIPT, 2014).

2.1.2 Principais Características

JavaScript é, atualmente, a opção mais utilizada[3] para possibilitar interatividade com o usuário em um navegador. Cada vez mais, aplicações são desenvolvidas com interface Web, o

[2]A Netscape era uma empresa com má reputação perante a sociedade e outras empresas e organizações, inclusive com a W3C.

[3]*Applets* Java foram uma promessa, mas nunca chegaram a ser amplamente utilizados, e mesmo assim, pode-se dizer que caíram em total esquecimento. O Flash da Adobe, também, (por motivos diversos que vão além do escopo desta discussão) vem sendo utilizado cada vez menos.

que logicamente requer um navegador para que possam ser utilizadas. Por outro lado, não é uma linguagem comumente ensinada em cursos superiores de informática. Mesmo quando há disciplinas relacionadas à linguagem JavaScript, o tempo empregado no assunto é insuficiente, e a profundidade de conhecimento atingida é mínima[4].

Outro fator a ser considerado é que se trata de uma linguagem com "cara" de C ou Java, o que leva o indivíduo (versado ou não em linguagens como as citadas) a pensar que por conhecê-las, está, implicitamente, apto a programar em JavaScript. Considere-se ainda o fato de que apesar da semelhança sintática com C ou Java (e outras similares), os conceitos fundamentais da linguagem divergem muito das linguagens mencionadas, pois herdou ideias das linguagens Self (objetos que atuam como protótipos) e Scheme (o modelo funcional). (CROCKFORD, 2010).

JavaScript é uma linguagem funcional. Nesta linguagem, funções são objetos de primeira classe (o que possui diversas implicações, como será visto adiante) (CROCKFORD, 2008, p. 3), não faz uso de *classical Inheritance* (RESIG, 2013, p. 143), mas sim de *prototypal inheritance* (RESIG, 2013, p. 150), suporta *lambdas* (funções anônimas) (CROCKFORD, 2008, p. 98), *closures* (CLOSURES, 2014)*, currying*, escopo é criado apenas por de funções (e não por blocos, como em outras linguagens), e o curioso fato de que o contexto da *keyword* `this` é definido dinamicamente (RESIG, 2006, p. 49)

Douglas Crockfor (2008, p. 3)[5] menciona em seu livro que JavaScript foi a primeira linguagem *lambda* (*lambda language*) a ser utilizada em larga escala[6], e ainda complementa em uma de suas palestras que isso era algo que o pessoal do MIT (*Massachusetts Institute of Technology*) vinha tentando fazer por cerca de quarenta anos (CROCKFORD, 2007).

Crockford (2008) afirma ainda que JavaScript é a linguagem mais mal compreendida e subestimada que existe. Segundo ele, até mesmo o nome da linguagem passa uma ideia errônea a seu respeito, dando a impressão de se tratar de uma linguagem imprópria para aplicações sérias (CROCKFORD, 2011).

Considerando-se todos os fatos mencionados, é seguro afirmar que não se pode querer utilizar JavaScript como se fosse java, C++, Python ou PHP. Pelo contrário, devido a sua natureza funcional e seus protótipos (em vez de classes), exige do programador raciocínio e

[4]Não temos a pretensão de insinuar quanto tempo deveria ser atribuída a cada disciplina. Estamos apenas constatando um fato e observando os efeitos causados por ele.

[5]Ex engenheiro de software no Yahoo! e atualmente Arquiteto Sênior JavaScript no Paypal, palestrante renomado reconhecido como uma das maiores autoridades em JavaScript, criador do JsLint, Json, e autor de um livro chamado JavaScript: The Good Parts.

[6]Há controvérsias quanto a esta afirmação. Há quem diz que o Perl 5, lançado em 1994, já possuía suporte a *lambdas*, e JavaScript foi lançado em 1995.

abordagem totalmente diferentes.

Após a introdução da linguagem, seguem-se algumas seções explicando suas principais características. Exemplos práticos de código são também utilizados com o intuito de facilitar o entendimento para o leitor.

2.1.3 Objetos

Objetos podem ser criados de diversas formas. Uma delas é com o operador `new` (FLANAGAN, 2011, p. 117).

```
var pessoa = new Object();
```

O objeto pessoa herda de `Object.prototype`, o que lhe confere algumas propriedades e métodos (na verdade, métodos são funções associadas a propriedades), provenientes da herança, como o atributo `prototype` e os métodos `toString` e `valueOf`. Pode-se fazer a constatação através da seguinte linha de código (FLANAGAN, 2011, p. 118,):

```
Object.getOwnPropertyNames( Object.getPrototypeOf( person ) );
```

De acordo com a documentação sobre objetos da Mozilla (WORKING, 2014), é possível adicionar e recuperar propriedades e métodos aos objetos de forma muito simples, e há duas maneiras de fazer isso.

Dot notation:

```
person.nome = 'Mestre Yoda';
person.acao = function () {
    return this.nome + ' conhece os caminhos da força.';
};
person.acao();
```

Subscript notation (mais útil quando o nome da propriedade possui caracteres especiais, como espaço em branco, acentos, etc):

```
person[ 'nome completo' ] = 'Mestre Yoda';
person[ 'ação!' ] = function() {
    return this[ 'nome completo' ] +
      ' conhece os caminhos da força.';
};
```

```
person[ 'ação!' ]();
```

Como um último exemplo de como objetos são flexíveis em JavaScript, pode-se tomar o caso do método `trim`, o qual está disponível em praticamente todas as linguagens modernas, e inclusive no JavaScript implementado sobre a especificação do ECMAScript 5. No entanto, até o ECMAScript 3 o método `trim` não fazia parte da especificação, e por isso muitos navegadores não o implementavam, como era o caso do navegador Internet Explorer 8 ou anterior (TRIM, 2014). Como em JavaScript funções são objetos de primeira classe, é possível verificar se o protótipo do objeto `String` possui a função `trim`. Se não possui, cria-se uma função que faça o trabalho de remover espaços no início e final de uma *string* e assina-se essa função a um membro de `String.prototype` chamado `trim`, conforme o exemplo abaixo (STRING, 2014).

```
// Pega o argumento da linha de comando.
var str = process.argv[ 2 ];

// Se não temos trim() nativamente, podemos facilmente
// adicioná-lo e a partir daí utilizá-lo como se
// fizesse parte da própria linguagem.
if ( ! String.prototype.trim ) {
    String.prototype.trim = function () {
        return this.replace( /^\s+|\s+$/g, '' );
    };
}

// Run: node script.js ' hello '
console.log( str.length );           // 7.
console.log( str.trim().length );    // 5.
```

É importante observar que o método criado só é adicionado a `String.prototype` caso já não exista nativamente. Além disso, da forma como foi implementado, não há diferença alguma para o programador utilizando o método `trim`. Isto é, a sintaxe, parâmetros e comportamento são iguais aos da implementação nativa. O mesmo processo pode ser repetido para qualquer funcionalidade, e é útil em casos nos quais uma aplicação deve ser capaz de rodar em ambientes diferentes, que implementam versões diferentes da especificação ECMAScript, o que faz que características da linguagem que estão presentes em uma situação possam não estar presentes em outra.

2.1.4 Arrays e objetos literais

Arrays e objetos literais são aqueles criados sem o uso da *keyword* `new`. Eles não precisam sequer ser atribuídos a um identificador. (STEFANOV, 2010, p. 47). São muito úteis quando é necessário passar um parâmetro do tipo *array* que por ventura não necessite estar associado a uma variável.

```
function printArray( arr ) {
    var i = 0, cur;
    for ( ; cur = arr[ i++ ] ; ) {
        console.log( cur );
    };
};
printArray( [ 'foo', 'bar', 'js' ] );
```

Objetos literais seguem o mesmo conceito. Esse tipo de criação de objetos é extensivamente utilizado como parâmetro para funções nos mais diferentes tipos de situações (VALUES, 2014). Como exemplo, pode-se citar uma função que serializa dados para que possam ser enviados através de uma requisição Ajax, pelo método *post* (RESIG, 2006, p. 219).

```
function serialize( values ) {
    var arr = [];
    for ( var key in values ) {
        arr.push( key + '=' +
          encodeURIComponent( values[ key ] ) );
    };
    return arr.join( '&' );
};
serialize({
    id: 9,
    nome: 'Obiwan Kenobi',
});
```

O resultado seria a seguinte *string*:

```
id=9&nome=Obiwan%20Kenobi
```

E utilizado da seguinte maneira:

```
var xhr = new XMLHttpRequest();
xhr.open( 'POST', 'serv.php', true );
```

```
xhr.setRequestHeader( 'Content-Type',
  'application/x-www-form-urlencoded' );
xhr.send( serialize( { id: 9, nome: 'Obiwan Kenobi' } ) );
```

Pode-se citar também o caso de uma função específica para executar requisições Ajax, que é provavelmente a utilização mais recorrente de objetos literais como parâmetros de funções. Deve-se observar que a função recebe o objeto `options` como parâmetro, e cria um novo objeto chamado `args` a partir de testes feitos em propriedades do parâmetro passado. O que não é explicitamente definido nos argumentos recebe um valor *default*. O objeto `args` é então utilizado dentro da própria função.

```
function ajax( options ) {
    var args = {
        type: options.type || 'GET',
         url: options.url || '/',
        data: options.data || '',
        time: options.timeout || 60000,
        onSuccess: options.onSuccess || function(){}
    };
    // Linhas não relevantes omitidas...
}
```

Segue a maneira como a função `ajax` pode ser usada. É importante observar que alguns parâmetros url, data, e time foram omitidos. No caso, das cinco propriedades possíveis que poderiam ser passados no objeto do parâmetro, apenas duas foram efetivamente utilizadas. As demais ficam com os valores padrão.

```
var elem = document.getElementById( 'fill' );
ajax({
    url: 'file.txt',
    onSuccess: function ( dados ) {
        elem.style.display = 'inline-block';
        elem.innerHTML = dados;
    }
});
```

Tudo o que está entre colchetes externos é um objeto literal, o qual contém até mesmo um método (nesse caso, uma função anônima associada a uma propriedade) que é responsável por trabalhar nos dados retornados pela requisição.

2.1.5 Funções – objetos de primeira classe

Em JavaScript, funções são objetos de primeira classe. Isso implica que funções podem:

1. ser assinadas à variáveis, *arrays*, e propriedades de outros objetos;
2. ser passadas como argumentos para outras funções;
3. ser retornadas por outras funções;
4. possuir propriedades que podem ser dinamicamente criadas e assinadas (RESIG, 2013, p. 34).

Por exemplo, tem-se a função `fnFunc` que recebe duas outras funções como parâmetro, as quais são por sua vez armazenadas em variáveis.

```
var fn1 = function () {
    return 10;
};

var fn2 = function () {
    return 20;
};

// Recebe duas funções como parâmetro.
function fnFunc( fnArg1, fnArg2 ) {
    return fnArg1() + fnArg2();
}

// Chama função, passando os parâmetros, que nesse caso são
// funções assinadas à variáveis.
fnFunc( fn1, fn2 ); // 30.
```

E um caso onde se tem um *array* de funções criado a partir de um *loop*. Aqui, tem-se uma *Immeditaly-Invoked Anonymous Function Expression* (IIFE) (JAVASCRIPT, 2014), que, por sua vez, retorna uma função. A função retornada pela IIFE é armazenada no *array*.

```
function maker() {
    var arr = [], i = 0;

    for ( ; i < 3; ++i ) {
        arr[ i ] = (function ( n ) {
            // Retorna uma função.
            return function () {
                return n + 10;
            };
        }( i )); // Passa o "i" como parâmetro.
    }
```

```
    return arr;
}

var callbacks = maker();
callbacks.forEach( function ( callback ) {
    callback();
});
```

Seria possível, também, chamar cada função acessando o índice do *array* da seguinte maneira:

```
callbacks[ 0 ]();
callbacks[ 1 ]();
callbacks[ 2 ]();
```

Este conceito de funções como objetos de primeira classe, originado das linguagens funcionais, já estava presente na linguagem JavaScript desde o seu lançamento. Dado o seu poder expressivo, outras linguagens mais "tradicionais" vem cada vez mais implementando essa funcionalidade. Esse é o caso, por exemplo, da nova linguagem da Apple, Swift, que é a sucessora da Objective-C, lançada em junho de 2014. Na verdade, Swift disponibiliza várias outras funcionalidades encontradas na linguagem JavaScript, como *closures* (A SWIFT, 2014), algo que o Java 8, lançado em 2014 também acabou por implementar, como mostrado na Seção 2.1.8. Java 8 também possibilita passar funções como parâmetro e tratar funcionalidades como dados (LEARN, 2014).

2.1.6 Prototypal inheritance vs Classical Inheritance

Já foi mencionado na Seção 2.1.3 que objetos podem ser criados através da *keyword* `new`, e fazendo isso o objeto criado herda propriedades e métodos (mas na verdade, métodos são funções associados a propriedades) de `Object`. Outra maneira de criar objetos é através do método `create()` do objeto `Object`. Isso permite herdar propriedades e métodos tanto de objetos nativos quanto de objetos criados pelo usuário (OBJECT, 2014).

```
var Pessoa = {
    nome: 'Fernando',
    email: 'contato@fernandobasso.net'
};
```

```
// Cria um objeto usando Person como protótipo.
var desenvolvedor  = Object.create(
    Pessoa,
    {
        // E acrescenta uma propriedade a mais...
        habilidade: { value: 'Programar em JavaScript' },
        // E um método.
        programar: { value:
            function () {
                return this.nome +
                      ' está programando em javascript';
            }
        }
    }
);

desenvolvedor.nome;          // Fernando Basso
desenvolvedor.email;         // contato@fernandobasso.net
desenvolvedor.habilidade;    // Programar em JavaScript
desenvolvedor.programar();   // Fernando está programando
                             // em javascript
```

2.1.7 Funções anônimas

Função anônima é uma função que não possui nome, e é especialmente útil como parâmetro para outras funções (FUNCTIONS, 2014). Vários desses exemplos já foram vistos em meio aos exemplos anteriores, e esta Seção apresenta mais um exemplo bastante comum onde este recurso é utilizado.

A JavaScript possui uma função chamada `sort`, que é utilizada para ordenar *arrays*. Por padrão, ela ordena alfabeticamente, o que faz com que, por exemplo, o valor 80 seja considerado antes do 9, já que o número 8 tem um valor de representação interno menor do que o 9. Para ordenar valores numéricos, deve-se passar uma função que instrui a função `sort` como proceder para ordenar os valores. Essa função parâmetro deve receber dois argumentos, os quais, ao sofrer uma determinada operação, retornam um valor. Esse valor retornado é utilizado pela função `sort` para determinar como a ordenação deve ocorrer[7]. No código que segue, o *array* de números inteiros é ordenado descendentemente.

```
var res = [ 3, 79, 91, 34 ].sort( function( a, b ) {
    return ( a - b ) * -1;
```

[7]Não é objetivo mostrar detalhes da função `sort()`, mas apenas mostrar o conceito de *lambdas*. Mais informações sobre ordenação de *arrays* em JavaScript podem ser encontradas no link: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/sort

```
});
// res agora contem os valores na seguinte ordem:
// 91, 79, 34, 3.
```

Esse conceito, apesar de já estar no JavaScript desde o seu lançamento em 95, está cada vez mais aparecendo em outras linguagens utilizadas em larga escala, visto a sua grande utilidade. Cita-se como exemplos a linguagens Java e PHP. Em Java, *lambdas* foram integradas à linguagem apenas na versão 8 (JAVA, 2014), cujo lançamento ocorreu em 2014 (JDK, 2014), e no PHP a partir da versão 5.3 (ANONYMOUS, 2014), lançada em 2009 (UNSUPORTED, 2014).

2.1.8 Closures

Graças a diversas características inerentes a JavaScript (funções como objetos de primeira classe e funções anônimas), é possível que funções contenham outras funções. *Closure* é um meio pelo qual uma função mais interna pode acessar variáveis presentes em uma função mais externa mesmo após ela (a função mais externa) ter retornado (sua execução ter sido finalizada) (RESIG, 2006, p. 29, tradução nossa). Veja-se um exemplo do uso de *closures* para simular uma classe como em linguagens orientadas a objetos tradicionais.

```
function Radio() {
    var marca = 'Sony';
    this.getMarca = function () {
        return marca;
    }
    this.setMarca = function( m ) {
        marca = m;
    }
}

var radio = new Radio();
radio.getMarca(); // Sony
radio.setMarca( 'Philips' );
radio.getMarca(); // Philips
```

Closures são úteis também para solucionar problemas inerentes ao próprio paradigma ao qual a JavaScript pertence. Por exemplo, uma função interna lembra o contexto da função externa no momento em que a função interna é executada, e não quando é criada. Isso quer dizer que uma função interna sempre acessa o último valor que uma variável da função externa possuiu. Para solucionar o problema, utiliza-se uma *closure*, o que induz o escopo e

mantém o valor desejado das variáveis (RESIG, 2006, p. 29).

O código abaixo não funciona como esperado, pois quando o evento `focus` acontece, o valor de `item` é o último objeto literal do *array*.

```
function mostraAjuda( ajudaMsg ) {
    document.getElementById( 'ajuda' ).innerHTML = ajudaMsg;
}

function formAjuda() {
    var i = 0, item;
    var textos = [
        { id: 'nome', 'help': 'Seu nome completo.' },
        { id: 'email', 'help': 'Providencie um email válido.' },
        { id: 'idade', 'help': 'Você deve ser maior que
           16 anos' }
    ];

    for ( i = 0; i < textos.length; i++ ) {
        item = textos[ i ];
        var elem = document.getElementById( item[ 'id' ] );
          elem.addEventListener( 'focus', function() {
            // Sempre mostra a juda sobre ser maior que 16 anos.
            mostraAjuda( item[ 'help' ] );
        }, false );
    }
}

formAjuda();
```

No entanto, fazer o *loop* utilizando uma *closure*, como no exemplo abaixo, força um escopo diferente em cada iteração, e tudo funciona como esperado.

```
for ( i = 0; i < textos.length; i++ ) {
    item = textos[ i ];

    (function ( itemAtual ) {
        var elem = document.getElementById( item.id );
        elem.addEventListener( 'focus', function() {
            mostraAjuda( itemAtual.help );
        }, false );
    }( item ) );
}
```

2.1.9 Currying

Currying é o nome dado a um conceito que ocorre quando uma função é utilizada para gerar outra função, a qual é criada com argumentos pré-definidos (RESIG, 2006, p. 28). O assunto pode se tornar bastante complexo, mas aqui é apresentado um exemplo simples. Trata-se de uma função que gera e retorna uma função de soma. O quanto será somado é definido durante a criação da função gerada. No momento de usar a função gerada, é passado apenas um operando, pois o outro operando já foi definido durante a geração da função de soma.

```
var geradorDeSoma = function ( num ) {
    return function ( addEste ) {
        return num + addEste;
    };
};

var add5 = geradorDeSoma( 5 );
var add80 = geradorDeSoma( 80 );
console.log( add5( 3 ) ); // 5 + 3, mostra 8
console.log( add80( 1 ) ); // 80 + 1, mostra 81
```

2.1.10 Keyword “this” tem contexto dinâmico

Para encerrar os conceitos essenciais da linguagem JavaScript será abordado o comportamento bastante peculiar da *keyword* `this`, cujo contexto é definido dinamicamente. Com isso, é possível, por exemplo, utilizar o construtor de um objeto em outro objeto, ou simplesmente utilizar uma função qualquer como método de um objeto e fazer o valor de `this` apontar para aquele objeto (FLANAGAN, 2011, p. 187).

```
// Cria uma função genérica.
var msg = function ( str ) {
    return this.nome + ' diz: ' + str;
};
// Cria um objeto com apenas a propriedade nome.
var obiwan = {
    nome: 'Obiwan Kenobi'
}
// Utiliza a função msg como método do objeto obiwan.
// this.nome é o atributo nome do objeto obiwan.
msg.call( obiwan, 'May the force be with you.' );
```

Outro exemplo comum é utilizar o construtor de uma função *constructor* em outros objetos similares.

```
function Produto( descricao, preco ) {
    this.descricao = descricao;
    this.preco = preco;
}

function Brinquedo( descricao, preco ) {
    // Força o "this" em Produto apontar para o objeto Brinquedo
    // em vez do objeto Produto.
    Produto.call( this, descricao, preco );
}

function Eletrodomestico( descricao, preco ) {
    // Fora o "this" em Produto apontar para o objeto
    // Eletrodoméstico em vez do objeto Produto.
    Produto.apply( this, [ descricao, preco ] );
}

var brinquedo = new Brinquedo( 'Hot Wheels', 35 );
var eletrodom = new Eletrodomestico( 'Liquidificador', 40 );

brinquedo.descricao; // Hot Wheels
brinquedo.preco;      // 35

eletrodom.descricao; // Liquidificador
eletrodom.preco;      // 40
```

2.1.11 Considerações

Foi apresentado um breve apanhado das características principais da linguagem JavaScript que mostra como se trata de uma linguagem diferente e que exige a interpretação de um paradigma diferente por parte do programador. Embora o conhecimento em outra linguagem seja útil àqueles que pretendem aprender e utilizar JavaScript, como estruturas de controles e laços, que são sim, similares, a maioria dos conceitos diferem muito em relação às linguagens mais conhecidas e utilizadas hoje em dia. Há certamente uma curva de aprendizado não muito tênue para que a linguagem possa ser dominada. Os livros e tutoriais encontrados na rede sobre a linguagem são, em sua grande maioria, escritos por pessoas que desconhecem essas características e tentam ensinar a linguagem como se fosse uma linguagem tradicional, como Java ou C. Os próprios livros, com algumas poucas exceções, limitam-se a fornecer exemplos para “copiar e colar”, com algumas explicações superficiais.

Por outro lado, um mundo de novidades se abre para aqueles que conseguem entender esses conceitos, e descobre-se uma linguagem simples e expressiva, possibilitando ao programador encontrar maneiras novas e estimulantes de desenvolver algoritmos e resolver problemas computacionais.

2.2 NODE.JS

O modelo de criação de aplicações Web atual possuem um processo de funcionamento bastante similar ao que se segue: instala-se um servidor como o Apache (para PHP, Python, Perl, etc.) ou GlassFish (Java), colocam-se os arquivos da aplicação em diretórios específicos, inicia-se o servidor e este executa o código dos arquivos. Para cada pedido de requisição, o servidor em questão utiliza um novo processo ou uma *thread* (dependendo do servidor utilizado) com o objetivo único de atender àquela requisição. Esse é um método que não oferece grandes dificuldades e tem funcionado de maneira satisfatória. O maior problema enfrentado, no entanto, é a dificuldade de escalabilidade, já que para cada cliente conectado, um novo processo ou uma nova *thread* (processo leve) são criados exclusivamente para manter a conexão do cliente, e o consumo dos recursos computacionais aumenta consideravelmente (APACHE, 2014). É este o problema que a Node.js tenta resolver.

A Node.js é uma plataforma para criação de servidores rápidos e escaláveis, buscando manter um consumo reduzido de recursos computacionais, principalmente no que diz respeito à memória (TILKOV; VINOSKI, 2010). Diferencia-se dos demais servidores populares, como o Apache, por utilizar apenas uma única *thread* de execução para tratar requisições, invés de criar uma *thread* específica para cada conexão. Isso traz a principal vantagem de possibilitar que um servidor trate de milhares de conexões sem que sejam necessários recursos computacionais exagerados.

Nesta Seção serão abordadas as características principais da tecnologia Node.js. Vale ressaltar que alguns conceitos se sobrepõem à própria linguagem JavaScript, já que a plataforma em si tem toda a sua base fundada sobre essa linguagem.

2.2.1 Uma única thread

Só há uma *thread* de execução para cada *runtime* JavaScript (DAGGET, 2013, p. 80).

Brendan Eich, criador da linguagem, diz, “de forma alguma eu colocaria *threads* de estado mutável, compartilhadas e preemptivas” porque julgava ser “inadequado para o público ao qual a linguagem se destinava” (EICH, 2014, tradução nossa).

Rodar sobre uma única *thread* oferece certas vantagens, como por exemplo, a ocorrência de *deadlocks* é impossível, e o desenvolvedor não precisa se preocupar com problemas de *race condition* e proteção da região crítica da memória[8]. Por outro lado, há, certamente, um limite de processamento que pode ser obtido em uma única *thread* (embora o processamento possa ser delegado a processos filhos através da utilização do módulo child_process[9]). Tem-se ainda os casos em que um *script* pode demorar muito para completar, tornando a interface de usuário não responsiva (embora esses casos sejam raros, notadamente com as funções `alert()` e `confirm()`) (DAGGET, 2013, p 80).

É importante observar que quando se fala de uma única *thread*, isto está se referindo à interface Node.js com o programador. Ou seja, o programador não precisa criar *threads* para possibilitar a execução concorrente de tarefas. Isso acontece automaticamente em um nível mais baixo, e se deve ao funcionamento da linguagem JavaScript, que é orientada a eventos e trabalha com o chamado *event loop*.

2.2.2 Event loop

Quando se trabalha com *threads*, ocorre o chaveamento de contexto, que é um processo complexo onde registros do processo em execução são passados para o descritor do processo, e os registros de um processo em espera são transferidos para os registradores do processador. O sistema operacional é responsável por essas tarefas e elas possuem, obviamente, um certo custo computacional (BERNARDI, 2012). Isso implica ainda programar pensando em evitar problemas de *deadlock* (impasse, ou, bloqueio de processos) e é necessário proteger a região crítica da memória contra *race conditons* (corridas de concorrência), o que não é uma tarefa trivial (BERNARDI, 2013).

Com Node.js, por ser uma característica da linguagem JavaScript, a única *thread* de execução sobre a qual o desenvolvedor tem contato é chamada *event loop*. Por isso, não há chaveamento de contexto, logo, é computacionalmente mais barato. Evita-se, também, os problemas já mencionados relacionados à programação com *threads* como *deadlocks* e *race*

[8] Área da memória que deve ser protegida contra o acesso de mais de um processo ou *thread* simultaneamente de modo a evitar a corrupção dos dados ou alteração incorreta dos dados.

[9] http://nodejs.org/api/child_process.html

condition pela região crítica da memória. As próximas subseções tratam desse assunto com mais detalhes, tentando mostrar suas características principais, explicar mais detalhadamente o funcionamento do *event loop*.

2.2.3 O que é o evento loop

O *event loop* é uma fila FIFO (*First In, First Out*) utilizada para sequenciar a execução de tarefas. Essa fila armazena mensagens (pedaços de código) que devem ser executadas. Programas registram *event listeners*, os quais ficam escutando por eventos. Quando um evento acontece, o código associado a ele é adicionado ao final da fila de forma que possa ser executado (DAGGET, 2013, p. 82).

Como existe apenas um *event loop* por *runtime*, cada mensagem deve ser processada completamente antes da próxima mensagem da fila poder também ser selecionada e processada (CONCURRENCY, 2014). Uma vez que uma função inicia a execução, ela não pode jamais sofrer chaveamento de contexto (preempção) de forma a dar oportunidade para que outra mensagem seja executada por um certo número de *clocks* do processador.

2.2.4 O event loop internamente

Mensagens (código a ser executado) são extraídas e transformadas em *frames*. Esses, por sua vez, são adicionados em uma pilha chamada *call stack* (pilha de execução) (DAGGET, 2013, p. 82). Cada *frame é* composto por pedaços de código que executam uma ação específica, e são executados isoladamente. No entanto, muitas vezes, para que o código de um *frame* possa ser executado com sucesso, ele depende do resultado do processamento de um outro *frame*. Como a pilha trabalha em um paradigma LIFO (*Last In, First Out*), se a execução de um *frame* A depende do resultado do processamento de um *frame* B, o *frame* A é colocado antes do *frame* B. Dessa forma, quando A é executado, ele já terá acesso ao resultado do processamento de B (CONCURRENCY, 2014).

2.2.5 Non-blocking I/O

As próximas três Seções focarão especificamente no funcionamento do paradigma

non-blocking I/O com Node.js, bem como em sua vantagem em relação à I/O tradicional (*blocking* I/O). Será abordado também como a plataforma se comporta quando trabalha com aplicações CPU-*bound*, nas quais o Node.js não oferece vantagens em relação a outras tecnologias, provando que nenhuma tecnologia é a solução ideal para todos os tipos de problemas.

2.2.5.1 Funcionamento

Com Node.js, todo tipo de I/O é, por padrão, *non-blocking* (NODE.JSa, 2014). Isso, como se pretende mostrar nos próximos parágrafos, é um conceito extremamente importante nessa tecnologia. Conforme o próprio Rayn Dahl menciona na sua apresentação original na JSConf, em 2009, o código seguinte exemplifica como uma linguagem de programação tradicional se comporta quando se trata de I/O:

```
result = db.query( 'select …' );

// 1. espera o resultado, bloqeuando a execução nesse ponto.
// 2. milhões de ciclos depois, usa o result.

// 3. finalmente, executa as próximas instruções.
```

E durante o tempo em que a aplicação está esperando o resultado da consulta, nada mais é feito, ou então se utiliza algum tipo de mecanismo de *threads*. Por outro lado, como Node.js, o paradigma seria algo como o que se segue:

```
// 1.
db.query( 'select …', function ( result ) {
    // 3. Usar o result.
});

// 2. Outras instruções são executadas enquanto se espera
//    pelo resultado da consulta.
```

No exemplo acima, o programa retorna para o *event loop* imediatamente, o que o habilita a continuar executando outras tarefas enquanto espera pelo resultado da consulta, e o mais importante, sem que o programador seja forçado a lidar diretamente com *threads*. Quando o resultado da consulta está finalmente disponível, a função *callback* é então

executada (DAHL, 2009).

Nesse ponto, é imperativo mencionar o porquê da linguagem JavaScript ter sido escolhida em detrimento de tantas outras linguagens extremamente bem estabelecidas e utilizadas em larga escala, como Java, Python, Ruby, Perl ou C. Novamente, segundo Ryan Dahl, outras linguagens foram testadas, mas o problema era que todas elas já possuíam um conceito e implementação de I/O: todas as funcionalidades relacionadas a I/O (consultas a bancos de dados, requisições e respostas HTTP, consultas DNS, etc.) sempre seguiam o paradigma *blocking*, o que colocava por terra todas as suas tentativas de construir um servidor extremamente rápido e leve. Por outro lado, JavaScript era uma linguagem, conforme suas palavras, "simples e pura", sem qualquer noção de I/O, o que o permitiria implementar do zero essas funcionalidades, e sempre seguindo o paradigma *non-blocking* por padrão[10].

Vale ressaltar, ainda, que há maneiras de utilizar *event loop* em Ruby com a biblioteca *EventMachine* (ABOUT, 2014), Python com *Twisted* (WHAT, 2014) e em Perl com *AnyEvent* (ANYEVENT, 2014). Quanto a isso, Ryan argumenta que programadores versados nessas linguagens já são acostumados a desenvolver da maneira tradicional, e o problema inerente é que o *event loop* é apenas uma biblioteca nessas linguagens, ou seja, são uma exceção à regra e ao paradigma de programação tradicional, e o desenvolvedor ainda tem que fazer uso de várias outras bibliotecas que são *blocking* por padrão, sendo que, por mais que essas bibliotecas ajudem, no final das contas, boa parte do trabalho ainda fica preso ao paradigma de *blocking* I/O.

2.2.5.2 Importância das operações I/O non-blocking

Mas por que é tão importante que operações de entrada e saída sejam *non-blocking*? O fato é que não se pode (ou não se deveria) tratar o acesso ao HD e a recursos na Web, como banco de dados e resolução DNS, da mesma maneira que o acesso às memórias cache (como L1 e L2) e memória RAM. Abaixo são apresentados alguns números que mostram, aproximadamente, o número de *clocks* que o processador tem que esperar para acessar certos recursos (KUNKLE, 2012):

---

[10] Ryan Dahl – History of Node.js. Uma palestra que aconteceu pelo *hangouts* da Google, a qual foi gravada e postada no youtube, e pode ser visualizada em https://www.youtube.com/watch?v=SAc0vQCC6UQ.

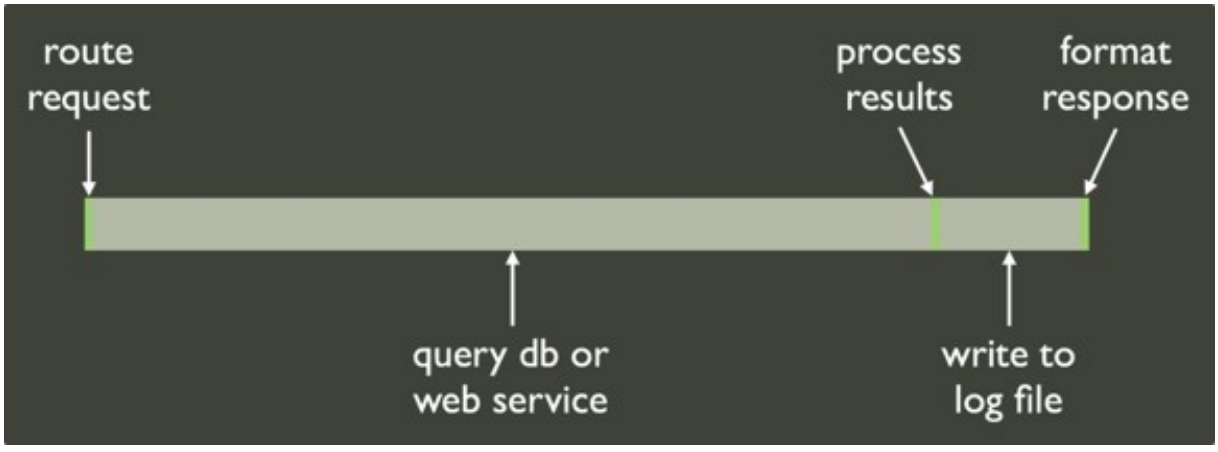
Tabela 1: Número de ciclos por tipo de operação

Operação	Número de ciclos
L1	3
L2	14
RAM	250
Disco	41.000.000
Rede	240.000.000

Fonte: KUNKLE, 2012

Tendo como base as informações da Tabela 1, pode-se, então, observar a Figura 1 que descreve como uma *thread* de execução passa a maior parte do tempo apenas esperando por resultados de operações que estão totalmente fora do escopo da *thread* (KUNKLE, 2012). Isto é, a *thread* está relacionada à conexão em si, e outras *threads* executam para realizar consultas a bancos de dados, ler e escrever arquivos no disco, etc.

Figura 1 – Exemplo de execução de uma thread

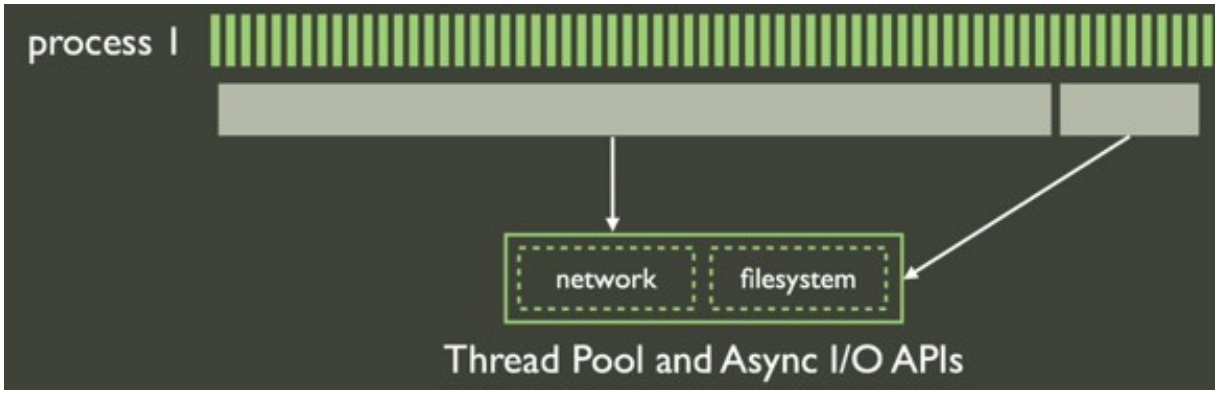


Font: KUNKLE, 2012

As pequenas barras verticais indicam os momentos nos quais a *thread* responsável pela conexão está realmente trabalhando. Todo resto, em cinza, é o tempo que a *thread* não faz nada. Mesmo assim, deve-se lembrar que ela fica utilizando recursos computacionais e sofrendo chaveamento de contexto (o que por si só têm um certo custo computacional) juntamente a outras *threads*. Para processar várias conexões simultaneamente, utilizando esse

método, maior é o tempo desperdiçado sem fazer nada, e cada conexão está atrelada a uma *thread* em execução (DAHL, 2009). Por outro lado, observa-se, através da Figura 2, como o *event loop* lida com essa questão:

Figura 2 – Exemplo de execução do event loop



Fonte: KUNKLE, 2012

Pode-se perceber como este método de tratar conexões é muito mais econômico em questão de recursos computacionais, pois utiliza de maneira muito eficiente a única *thread* de execução com a qual o desenvolvedor tem contato direto. Para ilustrar o quão efetiva esta abordagem pode se tornar, observe-se o caso do LinkedIn, que substituiu toda sua infraestrutura *mobile* para Node.js, e conseguiu reduzir 15 servidores com 15 instâncias de máquinas virtuais em cada servidor para apenas 4 instâncias Node.js, sendo capaz de suportar o dobro de tráfego[11] (DAGGET, 2013, p. 2).

2.2.5.3 CPU-bound vs IO-bound

Finalizando o assunto da importância de I/O *non-blocking*, é imprescindível realçar um ponto extremamente importante a respeito da plataforma Node.js. É uma tecnologia desenvolvida explicitamente para atender a uma certa categoria de aplicações: são aquelas que trabalham muito com I/O, como leitura e gravação de arquivos em disco, comandos SQL, resolução DNS, e tratamento de conexões TCP e UDP. (DAHL, 2009). Porém, não é especialmente indicada para situações onde o foco seja CPU-*bound*. Como se trabalha com uma única *thread*, para calcular primos, por exemplo, esta *thread* seria, possivelmente,

[11] Mais informações em http://venturebeat.com/2011/08/16/linkedin-node/.

utilizada por um longo período de tempo, o que faria o programa “bloquear” enquanto a operação fosse processada. O exemplo abaixo prova essa situação.

```
/**
 * Verifica se um número é primo.
 * @param Integer num O número a ser verificado.
 * @return Boolean true se é primo, senão false.
 */
function isPrimo( num ) {
    var max = Math.sqrt( num );
    var i = 2;

    for ( ; i <= max; ++i ) {
        if ( num % i === 0 ) {
            return false;
        }
    }
    return true;
}

/**
 * Descobre N números primos.
 * @param Integer quantidade
 * @return Array Um array de N números primos.
 */
function descobrePrimos( quant ) {
    var primos = [];
    var i = 0, n = 2;
    for ( ; quant > 0; ++n ) {
        if ( isPrimo( n ) ) {
            primos.push( n );
            --quant;
        }
    }
    return primos;
}

// Mostra o número de primos encontrados.
console.log( descobrePrimos( process.argv[ 2 ] ) );

// Somente após toda a operação anterior ter sido concluída
// a linha abaixo é executada.
console.log( 'Operação Finalizada.' );
```

No entanto, veja-se como Node.js trata operações I/O (seu ponto mais forte) e permite que tudo funcione de uma maneira *non-blocking*. Observe primeiramente um exemplo em que é feita uma requisição HTTP sobre um determinado domínio. É importante observar que a linha de código abaixo da função é executada imediatamente (ao contrário do que acontece com o caso do cálculo dos primos).

```
// Exemplo de uso:
// node async-io.js 'http://inf.passofundo.ifsul.edu.br/'
var http = require( 'http' );

http.get( process.argv[ 2 ], function ( res ) {
    res.setEncoding( 'utf8' );
    res.on( 'data', function ( parte ) {
        console.log( parte.length );
    });
});

// Executado imediatamente.
console.log( ' --- Início do processamento --- ' );

// Milhões de ciclos no futuro, o callback é executado
// e pode-se ver o tamanho de cada parte da requisição.
```

A seguir há mais um exemplo em que é feita a leitura de um arquivo no disco. Aqui, assim como no exemplo anterior, a linha logo abaixo da função que lê o arquivo é executada imediatamente. Em algum momento no futuro, o *kernel* do sistema operacional executa a função *callback*, a qual faz a sua parte e mostra os dados do arquivo.

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

fs.readFile('arq.txt', 'utf8', function (err, dados) {
   if (err) throw err;
    console.log(dados);
});
// Imprimido imediatamente.
console.log('Lendo arquivo.');

// Em algum momento no futuro, o conteúdo do
// arquivo é imprimido.
```

Funções *callback* não precisam ser anônimas. É possível definir uma função com um nome, e passá-la como parâmetro. O exemplo abaixo possui a mesma funcionalidade do exemplo anterior. A única diferença é que o código foi refatorado para utilizar uma função com nome, definida separadamente, invés de usar uma função anônima:

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

var myCallback = function (err, dados) {
    if (err) throw err;
    console.log(dados);
};

fs.readFile('arq.txt', 'utf8', myCallback);
```

2.2.6 Componentes

Ao se baixar e descompactar o código fonte do Node.js do site oficial[12], pode-se constatar que ele é composto de diversas outras bibliotecas e programas. São eles: c-ares, http_parser, npm, openssl, libuv, v8 e zlib. A lista a seguir, sumarizada dos arquivos "readme" e "licence" de cada um dos componentes empacotados com o Node.js, explica brevemente qual a função de cada um desses componentes.

1. C-ares: Biblioteca para resolução assíncrona de DNS e é útil para aquelas aplicações que necessitam realizar consultas DNS de forma *non-blocking* ou em paralelo. Escrita em C e distribuída sobre a licença MIT.
2. Http_parser: *Parser* de mensagens HTTP. Trata tanto de respostas quanto de requisições. Projetado para aplicações que utilizam o protocolo HTTP e necessitam de alta performance. Escrito em C e distribuído sobre a licença MIT.
3. NPM (Node Package Manager): Gerenciador de pacotes (modulos) para Node.js (instalar, atualizar e remover modulos). Escrito em JavaScript e distribuído sobre a licença *Artistic License 2.0*.
4. Openssl: É um *toolkit* que implementa os protocolos SSL (*Secure Sockets Layer*) versões 2 e 3 e TLS (*Transport Layer Security*) versão 1. Escrito em C e C++ e distribuído sobre as licenças próprias OpenSSL e SSLeay[13].
5. Libuv: É uma camada da plataforma Node.js que tem o objetivo de abstrair IOCP (*Input/Otput Completion Port*) em sistemas Windows e Unix, o que efetivamente permite operações I/O que não bloqueiam, que é o objetivo principal da plataforma em si. Escrito em C e distribuído sobre a licença própria Node.
6. V8: A *engine* (motor ou interpretador) JavaScript desenvolvido pela Google e utilizado no navegador Google Chrome e Opera, entre outros menos conhecidos. Implementa a 5ª edição da especificação ECMAScript. Pode ser executado separadamente ou embarcado em qualquer aplicação em C ou C++. Escrito em C++ e distribuído sobre a licença BSD.
7. Zlib: Biblioteca de compressão de dados de propósito geral. Escrita em C e distribuída sobre a licença própria Zlib.

---

12 http://nodejs.org/.

13 http://www.openssl.org/related/

2.2.7 Módulos

A plataforma implementa as suas funcionalidades e recursos através de módulos. Por exemplo, módulo `fs` (*file system*) seria utilizado para trabalhar com leitura e gravação de arquivos no disco[14], o módulo `net` para servidores TCP[15], e o módulo `http` para criar servidores HTTP[16].

A Node.js conta com uma série de módulos *core* e ainda milhares de outros módulos escritos pela comunidade, escritos em JavaScript. Os módulos *core* são oficialmente parte da plataforma e obviamente os mais seguros, robustos e úteis. Na versão 0.10.28, a Node.js conta com 40 desses módulos (NODE.JSb, 2014). Os módulos da comunidade, durante a escrita desta Seção, somaram um total de 78875, o que chega a ser um exagero. Este número pode ser conferido a qualquer momento acessando-se a URL www.npmjs.org (NPM, 2014). Obviamente, um número tão grande de módulos não oficiais remete à questão de confiabilidade. Para aliviar essa situação, o próprio site do NPM possibilita que usuários *logados* possam votar positiva ou negativamente os módulos, e observando o número de votos positivos e negativos é possível ter uma noção da qualidade de um módulo.

2.2.8 NPM – Node Package Manager

Como mencionado anteriormente, a Node.js conta com uma ferramenta para instalar módulos chamada *Node Package Manager* e vem na própria instalação da Node.js. Embora o NPM tenha sido escrito especificamente para a Node.js, é um projeto independente[17] também é escrito em JavaScript.

A utilização do NPM se dá através da linha de comando e é suficientemente simples, como será visto nos próximos exemplos, os quais foram tirados do próprio *help* da ferramenta.

---

14 http://nodejs.org/api/fs.html

15 http://nodejs.org/api/net.html

16 http://nodejs.org/api/http.html

17 https://www.npmjs.org/

2.2.9 Read, eval, print, loop

A Node.js possui um REPL (*Read*, *Eval*, *Print*, *Loop*, ou, em uma tradução livre, ler, interpretar, imprimir e repetir). Este REPL é uma interface em linha de comando utilizada para executar JavaScript interativamente, o que possibilita depurar (*debug*) *scripts*, testar pedaços de código e fazer experimentações (REPL, 2014).

Para entrar nesse modo, basta, a partir de um terminal ou *prompt* de comando, digitar `node --interactive`. Nesse ponto o usuário estará apto a executar código JavaScript, ou imprimir dados sobre o ambiente da plataforma. Quando o *prompt* do REPL mostrar um símbolo de "maior que", significa que ele está aguardando por entrada de comandos. Se o usuário pressiona enter antes de terminar um comando, o *prompt* muda para "`...`" indicando que o comando ainda não terminou. Alguns poucos exemplos serão mostrados a seguir.

Entrar no REPL e visualizar o sistema o seu sistema de ajuda:

```
$ node --interactive
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.exit     Exit the repl
.help     Show repl options
.load     Load JS from a file into the REPL session
.save     Save all commands in this REPL session to a file
>
```

Percebe-se pelo exemplo acima que os comandos do REPL propriamente ditos devem iniciar com um ponto.

Variáveis e operadores:

```
> var a = 3.14, b = 1;
> ( b + 1 ) * a;
6.28
```

A variável especial "`_`" armazena o resultado da última expressão interpretada:

```
> _
6.28
_ + 10
16.28
```

Um laço com os caracteres de uma *string*:

```
> var str = 'Node';
> str.lenght;
4
>   for ( var i = 0, chr; chr = str[ i++ ] ; ) {
...     console.log( '\t' + chr );
... }
    N
    o
    d
    e
```

Carregar um arquivo contendo código JavaScript para dentro da sessão REPL e executá-lo:

```
> .load for.js
> var str = 'Node.js'.split().reverse().join( '' );
> for ( var i = str.length - 1, chr; chr = str[ i-- ] ; ) {
...     for ( var j = i; j >= 0; --j ) {
.....         process.stdout.write( '\t' );
.....     }
...     process.stdout.write( chr );
...     process.stdout.write( '\n' );
... }
                                s
                           j
                      .
                 e
           d
     o
N
```

Observando-se a saída da execução do exemplo acima, percebe-se que o REPL carrega o arquivo tornando o seu conteúdo visível e executando cada linha em sequência, como se o código estivesse sendo escrito naquele momento.

Visualizar variáveis de ambiente, configuração, e o PID do REPL:

```
process.env;
process.config;
process.pid
```

Para ver tudo:

```
process;
```

Sair do REPL (observar o ponto):

```
> .exit
$
```

Um recurso dessa interface que ajuda muito é a possibilidade de utilizar a tecla *Tab*

para autocompletar comandos, sejam eles da linguagem JavaScript ou do próprio REPL. Por exemplo, ao digitar `process` seguido de um ponto e pressionar *Tab* duas vezes, todas as possibilidades de autocompletar são mostradas. Ou então, ao pressionar, por exemplo `i` seguido de *Tab*, pode-se observar todas as possibilidades e palavras reservadas, comandos e *keywords* da linguagem JavaScript que iniciam com essa letra. Para usuários acostumados com linha de comando em sistemas *Unix-like*, o REPL vai parecer algo bastante familiar. Até mesmo o modo de funcionamento do teclado é igual ao do editor Emacs, o que é o padrão (embora possa ser modificado) em sistemas como Linux, OSX e BSD (REPL, 2014).

2.2.10 Node.js como servidor

No paradigma atual de desenvolvimento para a Web, desenvolvedores utilizam um servidor para alguma linguagem ou fim específico. Esses servidores são como uma "caixa-preta", não oferecendo outra forma de controle senão através de arquivos de configuração. A plataforma Node.js quebra este paradigma oferecendo apenas as ferramentas necessárias para que o servidor seja criado. Não é um software que é instalado, inicializado e fica automaticamente executando arquivos PHP ou Python ou outra linguagem qualquer.

Outra característica que difere muito dos demais servidores, como Apache ou Tomcat, por exemplo, é que o servidor e a lógica de programação são implementados ao mesmo tempo. Em outras palavras, o código responsável por dar existência ao servidor é também o código responsável pelas regras de negócio da aplicação. Isto é diferente de se ter, por exemplo, o Apache executando de um lado, e os *scripts* PHP sendo interpretados pelo Apache do outro lado. O programador utiliza a linguagem PHP para tratar das regras de negócio, mas não toca em uma linha de código do Apache. Com Node.js, o desenvolvedor constrói o servidor e a lógica do negócio.

Finalmente, essa plataforma é voltada à linha de comando, principalmente durante o desenvolvimento.

Na Seção 2.2.5.3, foi mostrado um exemplo em que uma página da Internet e era requisitada e mostrado o tamanho de cada parte da resposta HTTP, e um exemplo no qual um arquivo no disco era lido e seus dados eram simplesmente imprimidos terminal. As próximas seções serão utilizadas para demonstrar, através de exemplos práticos, como a Node.js pode ser utilizada para se criar servidores Web diversos. Juntamente, serão expostos alguns outros pontos importantes da tecnologia quando estes se fizerem pertinentes.

2.2.10.1 Servidor HTTP

O módulo `http` pode ser utilizado para escrever tanto servidores quanto clientes HTTP. Segundo sua documentação, ele trabalha em um nível bastante baixo de forma a suportar recursos do protocolo que tem sido, tradicionalmente, difíceis de usar. Esse módulo se atém em fornecer uma API simples (HTTP, 2014) sobre a qual *frameworks* podem então ser escritos sobre a tecnologia Node.js de modo a facilitar a construção de clientes/servidores HTTP de uma maneira mais prática.

Primeiramente será mostrada uma porção mínima de código, seguida de uma discussão sobre os pontos importantes.

```
01. var http = require('http');
02.
03. http.createServer(function (req, resp) {
04.     resp.writeHead(200, { 'Content-Type': 'text/plain' });
06.     resp.write('Hello World!\n');
07.     resp.end();
08. }).listen(8888, 'localhost');
```

O código acima importa o módulo `http`, e o associa à variável também chamada `http`. Essa variável é, na verdade, um objeto com as funcionalidades HTTP nativas da Node.js. Na linha 3, o método `createServer` é chamado. Sua assinatura requer uma função *callback* com duas referências, uma para o objeto da requisição e outra para o objeto da resposta.

Como foi mencionado que o módulo `http` trabalha em um nível baixo, a linha 4 pode, de certa forma, provar isso, já que, como pode ser observado, é necessário escrever o cabeçalho HTTP com o *status code* (código de status) 200 (OK), que significa que o recurso foi encontrado no servidor, e o `Content-Type`, que indica o tipo de dados contidos no documento sendo enviado na resposta.

Na linha 6, é enviado o corpo da resposta HTTP, e, na linha 7, é enviado um sinal para o cliente de que a resposta encerrou, e nada mais será enviado. Finalmente, na última linha, faz-se com que o servidor escute na porta 8888. É necessário poderes de *root* fazer o servidor escutar nas portas baixas (HTTP, 2014).

Coloca-se a aplicação para funcionar com a seguinte linha de comando:

```
node servidor.js
```

Para testar o servidor, pode-se utilizar alguma ferramenta em linha de comando, como o cURL[18] (C*lient for URLs*), de código livre e disponível para muitas plataformas.

No exemplo que se segue, `curl` é invocado com o parâmetro `--include`, que o instrui a incluir informações de cabeçalhos. Isto possibilita visualizar o cabeçalho HTTP enviado pelo pequeno servidor, pois algumas dessas informações serão discutidas em seguida. O resultado da operação é mostrado abaixo:

```
$ curl --include http://localhost:8888
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Thu, 19 Jun 2014 22:40:23 GMT
Connection: keep-alive
Transfer-Encoding: chunked

Hello World!
```

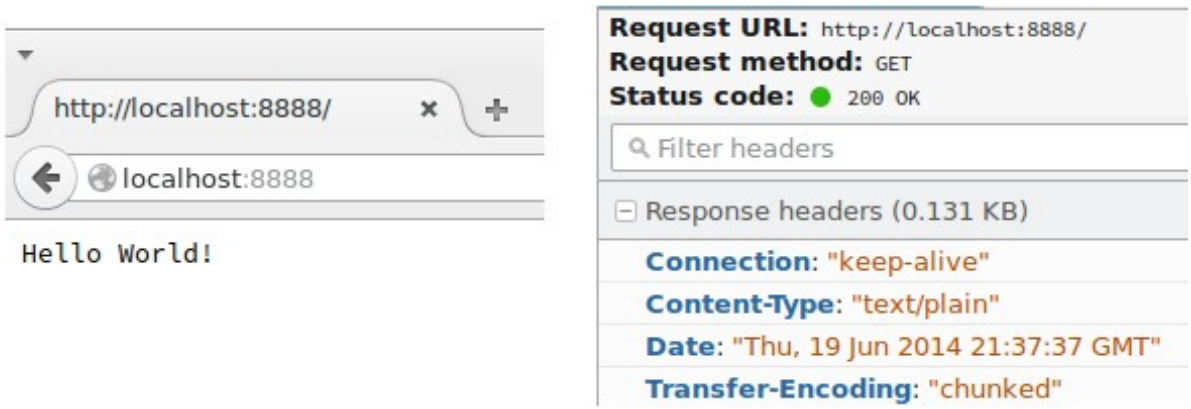
O cabeçalho de resposta reflete o código de status 200 que foi escrito explicitamente no código do servidor. Ainda, a versão do protocolo HTTP utilizada é a 1.1, cuja RFC determina a utilização de conexões persistentes (a conexão não é fechada e reaberta a cada envio de resposta) por padrão, contrário a versões anteriores do protocolo HTTP (HYPERTEXT, 1999, p. 44). O cabeçalho HTTP enviado na resposta pelo servidor do exemplo segue essa norma, como pode ser comprovado pelo parâmetro `Connection: keep-alive`, o que indica uma conexão persistente. Finalmente, e muito importante, o cabeçalho também mostra `Transfer-Encoding: chunked`, o que permite que o servidor envie corpo da mensagem HTTP em partes, à medida que eles são gerados dinamicamente pela aplicação. Nesse caso, o servidor não precisa saber o tamanho da mensagem antes de poder iniciar a transmissão (HYPERTEXT, 1999, p. 24) e também não obriga o servidor a armazenar a mensagem em um *buffer* no servidor antes de poder enviá-la, o que consumiria mais memória (INTRODUCTION, 2011). O fim da transmissão do corpo da mensagem é indicado pelo envio de um *chunk* (pedaço) final com tamanho zero (HYPERTEXT, 1999, p. 24). Finalmente, é mostrado o corpo da mensagem, “Hello World!”, e a requisição é encerrada.

Para finalizar este exemplo, é mostrada uma imagem do acesso feito ao servidor pelo navegador Firefox (Figura 3), onde são mostrados o resultado na página (extremamente

[18] http://curl.haxx.se/docs/faq.html#What_is_cURL

simples) e os dados do cabeçalho HTTP de resposta, que podem ser visualizados ao se pressionar o atalho de teclado Ctrl+Shift+Q para se acessar a aba *Network* das ferramentas de desenvolvimento do Firefox.

Figura 3 – Teste realizado no navegador Firefox



Fonte: do autor

2.2.10.1.1 Concorrência

Para testar a capacidade de atender a requisições concorrentes, é possível utilizar o programa *ab* (*Apache HTTP server benchmarking tool*). Pode-se, por exemplo, fazer um teste com 1000 requisições e 1000 conexões concorrentes, como mostra o comando a seguir:

```
ab -n 1000 -c 1000 http://127.0.0.1:8888/
```

E, a seguir, a parte relevante do resultado do comando acima:

```
Concurrency Level:      1000
Time taken for tests:   0.521 seconds
Complete requests:      1000
Failed requests:        0
```

Como é possível observar, a saída do comando mostra que foram executadas 1000 conexões concorrentemente, o que levou um tempo aproximado de meio segundo. Todas as requisições completaram com sucesso.

O ponto mais relevante na experiência demonstrada é que em momento algum houve a necessidade da utilização de *threads* no que diz respeito à programação do servidor. O *event loop*, como mencionado em sua Seção 2.2.2, trata de estabelecer a conexão e colocar o código

das funções *callback* na pilha de execução. Internamente, há *threads* sendo executadas para determinadas tarefas (e isso fica escondido do desenvolvedor), mas não é, de forma alguma, utilizada uma *thread* para cada conexão, o que aumentaria significativamente a quantidade de recursos computacionais utilizados (DAHL, 2009).

2.2.10.2 Servidor TCP

O módulo `net` (também um módulo *core* da Node.js) pode ser utilizado para criar clientes e servidores TCP, fornecendo *sockets* para a manipulação de *streams* (fluxos) de dados entre cliente/servidor (NET, 2014). O exemplo a seguir foca na construção de um servidor TCP simples.

```
01. var net = require('net');
02.
03. var clientes = [];
04.
05. net.createServer(function (socket) {
06.
07.     // Quando um cliente conecta, colocamos ele no array.
10.     clientes.push(socket)
11.
12.     // Quando dados chegam no socket, manda esses
13.     // dados para todos os clientes conectados.
14.     socket.on('data', function (dados) {
15.         for (var i = 0, sock; sock = clientes[i++] ; ) {
16.             // Não manda dados para o cliente que os enviou.
17.             if ( sock !== socket ) {
20.                 sock.write(dados);
21.             }
22.         }
23.     });
24.
25.     // Se um cliente desconecta, remove do array.
26.     socket.on('end', function () {
27.         var idx = clientes.indexOf(socket);
30.         clientes.splice(i, 1);
31.     });
32.
33. }).listen(8888);
```

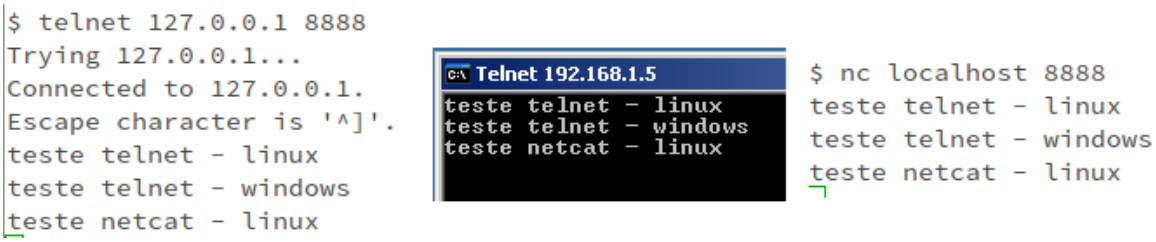
Na linha 5, é criado um servidor com o método `createServer` e passado uma função *callback* como parâmetro. Essa função define, como parâmetro, uma referência que representa o *socket* que origina a conexão, o qual, na linha 10, é adicionado em um *array* chamado `clientes`. Na linha 14 e dentro do bloco iniciado por ela, é definido que quando

ocorrer o evento `data`, deve-se retransmitir os dados vindos pelo *socket* para os demais clientes conectados, com exceção daquele que enviou a mensagem.

No final do programa, mais especificamente na linha 26, o programa é instruído a remover *sockets* que se desconectam do *array* `clientes` para evitar a tentativa de envio de dados para *sockets* já encerrados no bloco da linha 14. A última linha fecha o bloco principal e também faz com que o servidor, ao ser iniciado, escute por conexões na porta 8888.

Para testar o servidor TCP, programas como `telnet`[19] ou a versão GNU `netcat`[20] podem ser utilizadas. Nos testes realizados, foram feitas várias conexões utilizando-se as ferramentas citadas a partir de terminais de linha de comando.

Figura 4 – Conexão no servidor TCP com netcat e telnet



Fonte: do autor

Embora o servidor TCP do exemplo seja extremamente simples, pode-se seguramente afirmar que a Node.js permite, com poucas linhas de código, construir aplicações de rede de maneira bastante econômica em relação ao número de linhas de código escritas. Ainda, assim como no servidor HTTP criado no exemplo da Seção anterior, não foi necessário a utilização de *threads* para permitir a conexão concorrente de clientes. Novamente, o *event loop* faz o trabalho inicial de estabelecer a conexão e colocar as mensagens (código a ser executado proveniente das funções *callback*) na pilha de execução. Quando eventos acontecem, a Node.js, em trabalho conjunto com o sistema operacional, processa o que é necessário e manda o resultado para o *event loop* de modo que possa ser finalmente enviado para o cliente que deu origem ao evento (DAHL, 2009).

2.2.11 Site, documentação e comunidade

[19] http://www.telnet.org/

[20] http://netcat.sourceforge.net/

O site oficial do projeto é http://nodejs.org. A partir dessa página é possível acessar vários outros *links* como o do *blog* oficial da plataforma, onde são postadas notícias e novidades do interesse a indivíduos e organizações de alguma forma relacionados à tecnologia. Outro *link* interessante pode ser acessado ao clicar no item *Community* do menu de navegação, o que leva a uma página com diversas opções para aqueles que desejam aprender a desenvolver com a Node.js, como a *NodeSchool.io* e a *How To Node*. Há, também, um *link* que leva para o repositório no *github*[21] onde o código do projeto é armazenado e gerenciado. Entre outros *links* para páginas úteis em determinadas situações, há, finalmente, um que aponta para uma página explicando como proceder para contribuir para o projeto.

Quanto à documentação, a URL é http://nodejs.org/api/. Nessa página, encontra-se a documentação sobre a Node.js e seus módulos *core*. Documentação sobre os módulos da comunidade são encontrados no próprio site do NPM, https://www.npmjs.org. Ao acessar a página de um módulo, várias informações sobre ele são mostradas, e logo abaixo toda a documentação disponível sobre aquele módulo específico.

Finalmente, a documentação do executável `node`, utilizado pela linha de comando pode ser visualizada através dos comandos `man node` (sistemas *Unix-like* apenas) ou `node --help`. Alguns módulos também oferecem a possibilidade de utilizar a flag `--help`, como é o caso do módulo `uglifyjs`[22] e sem flag alguma como para o `uglifycss`[23].

2.2.12 Licença de utilização

Node.js está disponível sob da licença MIT, e "empacota outros componentes *open source* liberalmente licenciados" (DOWNLOAD, 2014, tradução nossa). Isso significa que a plataforma pode ser livremente utilizada, alterada, melhorada e redistribuída sem que seja necessário pagar por ela.

---

21 https://github.com/

22 https://www.npmjs.org/package/uglify-js

23 https://www.npmjs.org/package/uglifycss

3 AVALIAÇÃO E COMPARAÇÃO DA PLATAFORMA NODE.JS

O Capítulo 2 cobriu parte dos objetivos descritos na Seção 1.2, mais especificamente no que diz respeito a apresentar conhecimento conceitual e prático a respeito da linguagem JavaScript, e também a parte conceitual e exemplos práticos sobre a utilização da Node.js. Naquele Capítulo, foi apresentada, ainda, a infraestrutura que compõem a plataforma, como componentes, módulos, NPM, documentação, licença de uso e outros detalhes. Este Capítulo visa alcançar outro objetivo: avaliar a eficácia da plataforma, fazendo um comparativo entre Node.js e outras duas tecnologias, as quais serão apresentadas nas próximas seções.

O texto que segue, portanto, descreve o processo utilizado para a execução dos testes avaliação de desempenho da plataforma Node.js e os resultados obtidos. Para finalizar, é feita uma descrição sobre o processo de criação de uma das aplicações e a experiência resultante. Essa parte é bastante pessoal e o autor dá a sua própria opinião a respeito da tecnologia.

3.1 OBJETIVOS DA AVALIAÇÃO

O objetivo dos testes de avaliação de desempenho é medir o tempo de resposta, o número de clientes que o servidor é capaz de atender dentro de um determinado tempo e o consumo de recursos computacionais utilizados para tal, mais especificamente o uso da CPU e da memória RAM.

3.2 AMBIENTE DE TESTES

Os testes efetuados para o presente estudo foram executados em um sistema Linux, rodando a distribuição Arch Linux[1] 32 bits com 3GB de RAM com processador Pentium Dual Core com 2.10GHz cada núcleo. Para rodar as aplicações (descritas mais adiante) foram instalados o servidor HTTP Apache 2.4.10, o servidor de banco de dados PostgreSQL 9.5.1 e o próprio Node.js, na versão 0.10.33, todos dos repositórios oficiais da distribuição. O

[1] Distribuição Linux totalmente orientada a linha de comando, da instalação a manutenção. Mais informações no site oficial: https://www.archlinux.org/.

servidor Tomcat 8.0 foi instalado a partir do arquivo disponibilizado no site oficial do projeto[2].

3.3 FERRAMENTAS UTILIZADAS E METODOLOGIA

Uma parte dos testes de *benchmarking* foca em avaliar o número de requisições que o servidor/aplicação é capaz de atender por segundo e o tempo médio tomado para atender cada requisição individual. Para este fim, foi utilizado um programa específico para a execução de *benchmarks* chamado Apache Bench[3], o qual é instalado automaticamente ao se instalar o servidor Apache. Esse programa apresenta uma interface em linha de comando, é um projeto da Fundação Apache, *open source*, e roda em diversos sistemas operacionais, tais como Linux, Windows, BSD e OSX. Optou-se por executar cada teste quatro vezes, somar os resultados e dividir por quatro para chegar uma média sobre cada um dos itens a serem mensurados.

A outra parte dos testes visa medir o consumo de memória RAM e de CPU utilizados durante a execução de cada aplicação. A implementação prática deste tipo de mensuração se mostrou extremamente complexa. O plano inicial era medir exatamente a quantidade de CPU e RAM utilizados por aplicação em cada caso de testes, mas após várias tentativas frustradas de se atingir esse objetivo, ficou claro que a execução de testes de avaliação e comparação de desempenho entre diferentes tecnologias necessitam um estudo separado e mais aprofundado.

Cada uma das plataformas utilizadas para os testes utilizam diferentes mecanismos de execução. Por exemplo, Node.js, apesar de fornecer ao desenvolvedor o acesso a apenas uma *thread*, o *event loop* (ver 2.2.2), as utiliza internamente para execução de operações I/O. O Apache utiliza um novo processo para cada requisição (como pode ser comprovado executando-se o comando `ps u -C httpd`[4]), e o Tomcat utiliza um único processo principal, mas cada nova requisição é atendida por uma nova *thread* (visualizadas com o comando `ps -efL | sed -n '/tomcat/p'`[5]).

Decidiu-se então adotar uma estratégia mais simples e exequível. Para medir uso de

---

2 http://tomcat.apache.org/

3 http://httpd.apache.org/docs/2.2/en/programs/ab.html

4 A opção -C do comando ps mostra todos os processos com o nome passado como argumento, conforme o seu manual documenta (man 1 ps).

5 A opção L do comando ps mostra as *threads* (e não somente o processo principal). Essa opção também é descrita no manual do comando ps.

memória e CPU foi utilizado um utilitário chamado gnome-system-monitor e observado do uso de recursos durante a execução de cada teste. Dessa forma, foi possível fazer uma observação mais geral (embora mais imprecisa) a respeito do uso de recursos. As tabelas criadas com os resultados mostram os picos de uso de memória e CPU.

As aplicações em Node.js e PHP foram criadas utilizando o editor de textos Vim (Vi Improved[6]), já no local onde os arquivos são executados pelo servidor (embora com Node.js pode-se executar os arquivos a partir de qualquer diretório). Já as aplicações em Java foram desenvolvidas na IDE Netbeans. No entanto, para execução e testes dessas aplicações em Java, estas foram devidamente instaladas no diretório *webapps* do Tomcat (o qual, como foi mencionado, foi instalado manualmente) e executadas sem qualquer ajuda do Netbeans justamente para não permitir que a IDE (que possui um alto consumo de recursos computacionais) pudesse influenciar na mensuração dos testes.

3.4 TECNOLOGIAS UTILIZADAS

As comparações são feitas utilizando três linguagens/plataformas: Node.js (que como foi mencionado na em 2.2.10, faz o papel de servidor e da programação da lógica das regras do negócio simultaneamente), PHP, executado sobre o servidor Apache, e Java executado através do Tomcat. Node.js é utilizado por se tratar do objeto principal do presente estudo. PHP e Java, no entanto, foram escolhidas por serem as duas linguagens ensinadas no curso Tecnologia em Sistemas para Internet, por serem amplamente utilizadas para construção de sistemas e aplicações Web[7] em geral e também por serem, ambas, assim como o Node.js, multiplataforma, não dependendo de ferramentas pagas ou algum sistema operacional específico e principalmente por utilizarem historicamente um mecanismo de funcionamento oposto ao Node.js. Outro fator importante quanto à escolha de Java e PHP é o fato de que Java é uma plataforma ampla, a qual é utilizada para outras finalidades além de sistemas Web, tem uma base sólida e conta com muitos anos de evolução e amadurecimento, enquanto que o PHP foi escolhida por ser uma linguagem programação específica para aplicações Web[8].

---

[6] http://www.vim.org/about.php

[7] A IEEE publicou em 3 de Julho de 2014 uma lista com o *ranking* de popularidade das principais linguagens. É necessário pagar para ter acesso aos dados, mas um site relativamente conceituado publicou um resumo no link http://www.infoq.com/br/news/2014/10/ranking-linguagens-ieee.

[8] Embora PHP possa ser utilizada para construir aplicações em linha de comando. Mais informações em: http://php.net/manual/en/features.commandline.php

Em todos os casos criados para os testes, foi deliberadamente tomado o máximo de cuidado para não executar linhas de código desnecessárias, consultas diferentes, ler arquivos diferentes ou executar algoritmos dissimilares em nem uma das versões das aplicações de modo evitar o favorecimento inadvertido de alguma das linguagens envolvidas nos testes. É importante ressaltar também nem o Apache ou o Tomcat tiveram configurações alteradas. Essa decisão foi tomada com o objetivo de não influenciar na performance da aplicação por conta de configurações extras nos servidores utilizados.

Todos os casos de estudo utilizados para os testes utilizam algum tipo de conexão por rede, isto é, o cliente sempre chega ao servidor através do protocolo HTTP. Após a conexão ser estabelecida, algum tipo de operação é, então, realizada no servidor, onde dados são recuperados ou produzidos e enviados para o cliente.

3.5 APLICAÇÃO 1 – I/O DE DISCO RÍGIDO

Para os testes de I/O de disco rígido, o objetivo é criar uma aplicação que atende requisições concorrentes de clientes, lê um arquivo de texto no HD, e retorna o conteúdo do arquivo para o cliente.

O arquivo a ser enviado contém 368 Kilobytes de texto puro. As três versões da aplicação (Node.js, PHP e Java) fazem exatamente o mesmo trabalho, isto é, simplesmente atender a requisição do cliente, abrir e ler o arquivo, e enviar seu conteúdo para o cliente.

A aplicação Node.js, por utilizar HTTP/1.1, trabalha com *Transfer-Encoding chunked*. Isso significa que não é necessário que o servidor leia o arquivo todo, salve na memória para só então enviar o arquivo. Ao contrário, conforme o arquivo vai sendo lido do HD, as partes (*chunks*) são enviadas ao cliente, o que permite que uma quantidade muito pequena de RAM seja necessária durante o processo (pelo menos em teoria), conforme foi explicado de maneira mais detalhada em 2.2.10.1.

3.6 APLICAÇÃO 2 – I/O DE BANCO DE DADOS

Similar aos testes da aplicação anterior, mas com a diferença de que neste caso os dados são recuperados de um banco de dados, e não de um arquivo no HD. Na Seção 3.9,

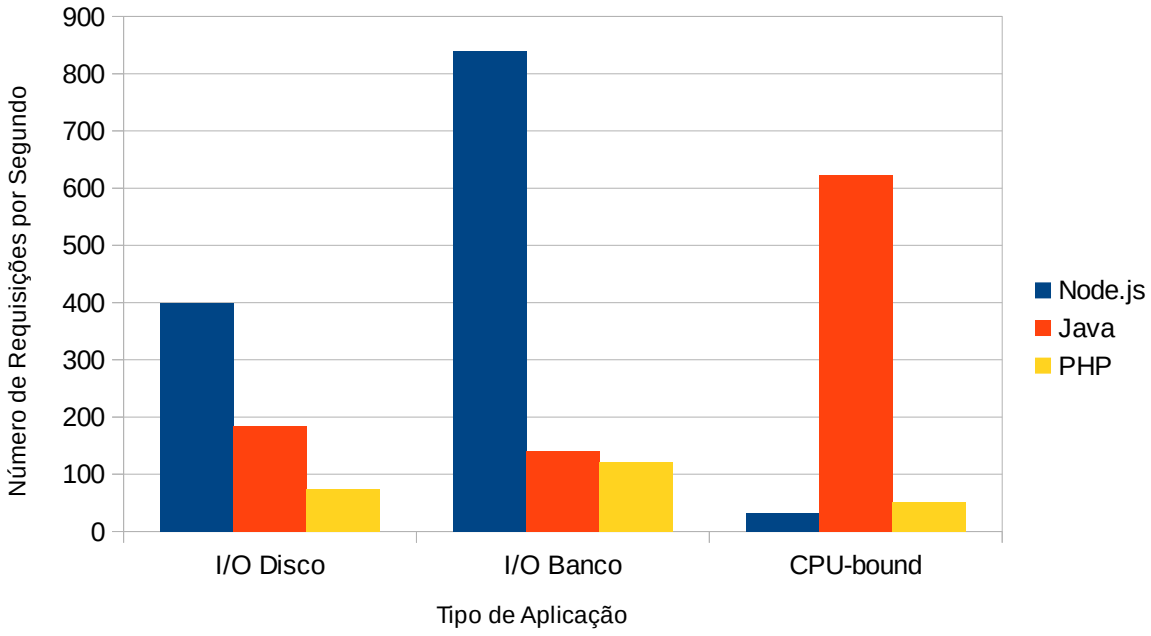
Experiência de Uso, a versão em Node.js desta aplicação é explicada detalhadamente.

3.7 APLICAÇÃO 3 – CPU-BOUND

O objetivo desta aplicação é fornecer um servidor onde os clientes podem efetuar requisições pedindo por respostas a um problema matemático, como realizar um cálculo complexo ou encontrar os primeiros 1000 números primos. Esta aplicação também atende os clientes por meio do protocolo HTTP.

3.7.1 Resultados para avaliação de número de requisições por segundo

Figura 5: Resultados para teste de número de requisições por segundo

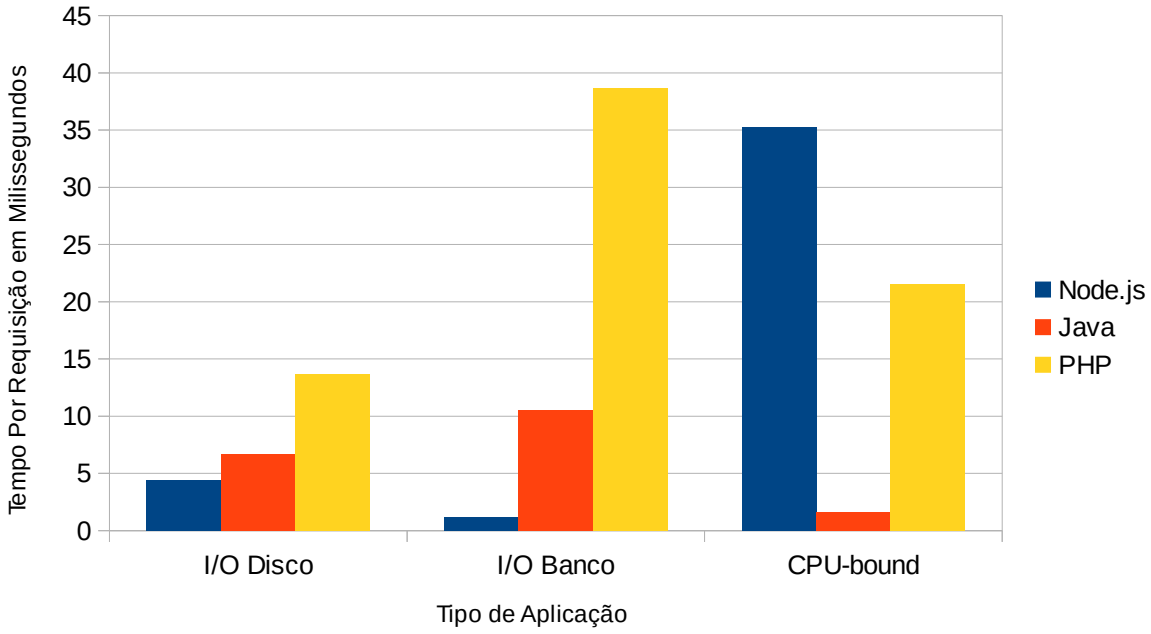


Fonte: do autor

Percebe-se claramente como a Node.js é bastante rápida para atender requisições quando se trata de operações I/O, sendo capaz de atender a mais que o dobro de requisições por segundo quando comparada com Java e PHP. Para a aplicação CPU-*bound*, Java, em contrapartida, Java foi muito mais eficaz do que Node.js e PHP.

3.7.2 esultados para avaliação de tempo por requisição

Figura 6: Resultados para teste de tempo por requisição em milissegundos

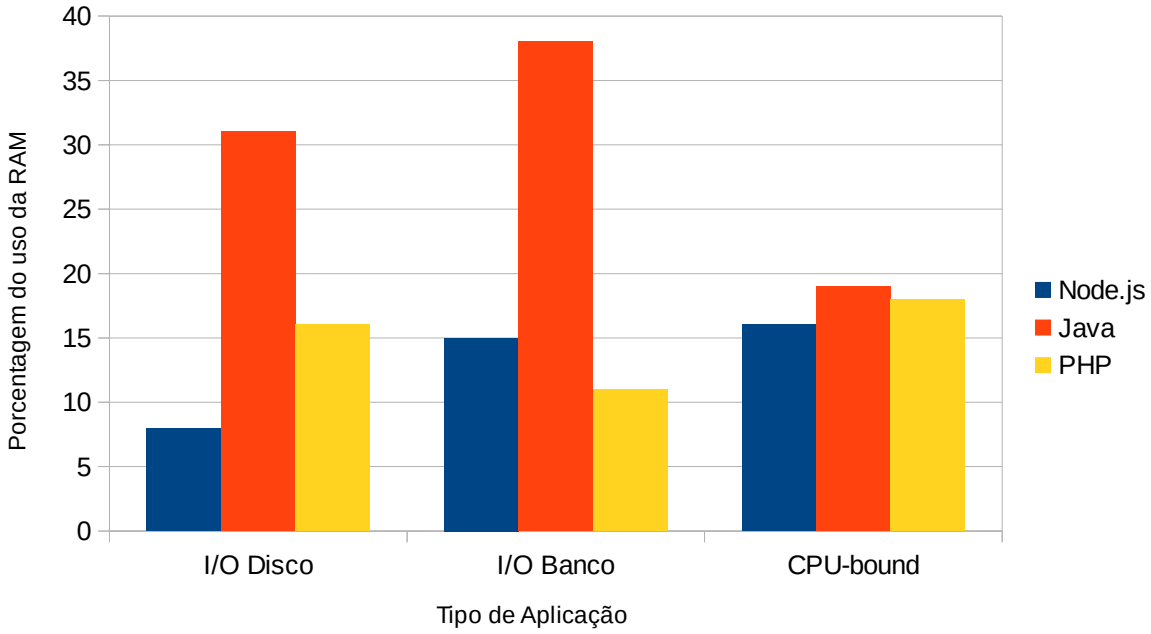


Fonte: do autor

No quesito tempo por requisição, a Node.js mais uma vez se mostrou superior quando a aplicação envolve algum tipo de I/O. Não chegando a 5 milissegundos para atender cada requisição na aplicação I/O de disco, e menos de dois milissegundos para a aplicação I/O de banco de dados. Para a aplicação I/O, Java novamente se saiu melhor, PHP ficando em segundo lugar.

3.7.3 Resultados para avaliação de uso da memória RAM

Figura 7: Resultados para teste de consumo de RAM

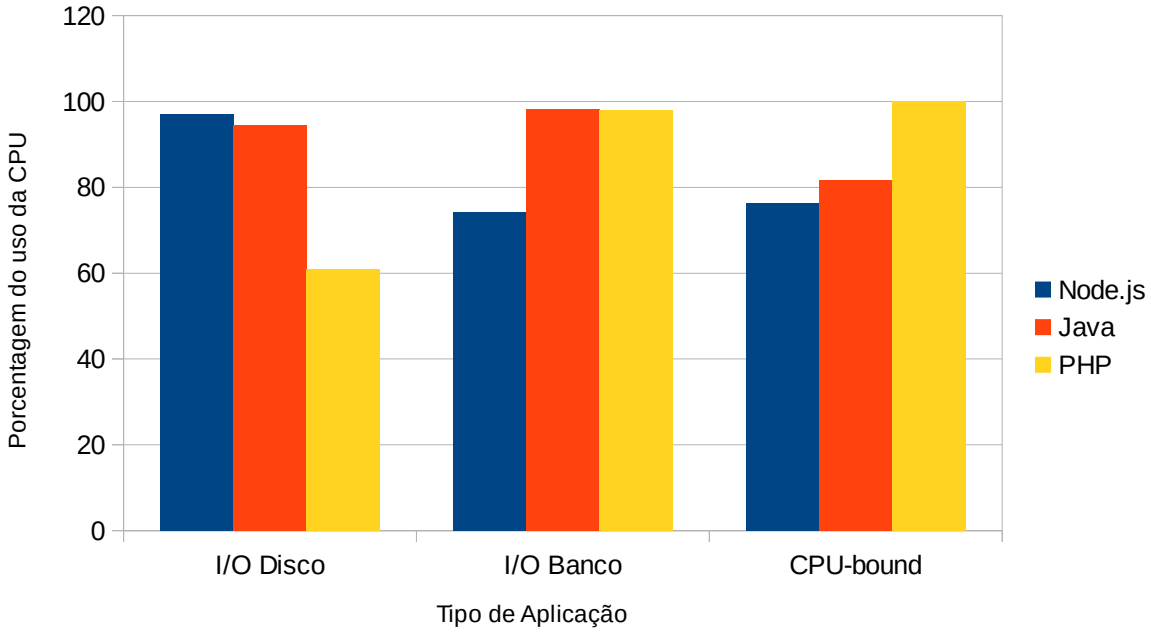


Fonte: do autor

Em relação ao consumo de memória RAM, a Node.js se manteve abaixo das demais tecnologias, com exceção da PHP, na aplicação de I/O de banco de dados e mais uma vez se mostrou bastante eficiente quando se trata de aplicações I/O. Apesar de ter tido a menor utilização da RAM na aplicação CPU-*bound*, não podemos esquecer que a versão em Node.js utiliza uma única *thread* de execução, o que consequentemente restringe a quantidade de memória que aquela *thread* vai necessitar, ao contrário de quando há diversas *threads* em execução, o que obviamente vai requerer mais uso de memória.

3.7.4 Resultados para avaliação de uso da CPU

Figura 8: Resultados para teste do uso da CPU



Fonte: do autor

Quanto a utilização da CPU, todas as tecnologias avaliadas tiveram um consumo bastante alto, não chegando a apresentar diferenças extremas. No entanto, vale lembrar que apesar do consumo da CPU ter sido similar em todas as plataformas, a Node.js se saiu muito bem no quesito número de requisições por segundo e tempo por requisição, e isto quer dizer que com praticamente a mesma carga na CPU, a Node.js foi capaz de atender a um número muito maior de requisições por segundo e com um tempo proporcionalmente menor para atender cada requisição.

3.8 CONSIDERAÇÕES GERAIS SOBRE OS TESTES

Após observar as tabelas com os resultados de todos os testes, pode-se dizer que Node.js não é necessariamente mais eficaz que outras tecnologias. Isso vai depender da finalidade da aplicação, como foi o caso da aplicação para banco de dados e aplicação de I/O de disco, onde a Node.js se saiu muito bem, ao passo que foi a pior das três para trabalhar

com uma aplicação CPU-*bound*. Em teoria, Node.js é mais inteligente quanto ao uso da CPU, como foi explicado na Seção 2.2.5.2, importância das operações I/O *non-blocking*. Mesmo assim, existem diversos outros fatores além do uso da CPU envolvidos em uma aplicação Web.

Há também que se considerar que aplicações Web podem ser otimizadas de diferentes formas, desde o *tuning* do banco de dados até o *cache* e compactação de recursos estáticos e distribuição de carga, sem mencionar otimizações que podem (e devem) ser feitas nos próprios servidores. Ainda, executar uma aplicação PHP sobre o servidor Nginx ou Lighttpd e aplicações Java sobre o Glassfish ou JBoss resultaria em uma performance diferente. Pode-se especular que um especialista em aplicações Java ou PHP obteria melhores resultados do que um novato com Node.js

De um modo geral, Node.js se mostrou bastante capaz. Se saiu especialmente bem na aplicação com banco de dados e também teve uma excelente performance na aplicação de I/O de disco. Quanto a aplicações CPU-*bound*, seu próprio criador, Ryan Dahl, dize que Node.js não é plataforma criada com esse tipo de aplicação em mente. Em fim, para aplicações Web, Node.js pode sim ser utilizada sem maiores problemas até com alguns ganhos potenciais em diversas situações.

Tendo como base os resultados dos testes e as características da plataforma Node.js, pode-se enumerar alguns casos onde a ela poderia ser utilizada com sucesso:

1. Aplicações que trabalham com operações a leitura e escrita de arquivos em disco.
2. Aplicações que trabalham com operações relacionadas a bancos de dados.
3. Aplicações onde o servidor atualiza a aplicação no lado do cliente em tempo real.
4. Aplicações onde o cliente fica por longos períodos de tempo conectado ao servidor, técnica conhecida como *long pooling*[9], o que causaria um imenso consume de recursos em outras tecnologias (uma *thread* ou processo por cliente, versus uma única *thread* para muitos clientes com Node.js).
5. Aplicações de *chat*.
6. Aplicações que tratam de muitas requisições concorrentes e que façam uso de poucos ciclos de CPU.

[9] http://en.wikipedia.org/wiki/Push_technology#Long_polling

3.9 EXPERIÊNCIA DE USO

A seguir será exposto o processo de criação da aplicação utilizada para servidor com banco de dados em Node.js. O objetivo é explicar o código, descrevendo os passos necessários e a experiência resultante do processo de criação.

3.9.1 Construção da aplicação servidor com banco de dados em Node.js

Abaixo, é mostrado o código completo da aplicação servidor com banco de dados. Em seguida, partes do código são explicadas individualmente. É importante lembrar que todas as aplicações foram criadas com o objetivo principal de servir para os testes. Por exemplo, nenhum sistema de *templates* foi utilizado para exibir dados (camada *view*) e também não houve preocupação com modularização e organização de código. Todos os exemplos foram criados da maneira mais direta possível dentro de cada tecnologia[10]. Deve-se ter essas considerações em mente ao acompanhar o código que segue.

```
01. var http = require('http');
02. var pg = require('pg').native;
03. var constr = 'postgres://devel:1234@127.0.0.1/tcc';
04.
05. var server = http.createServer(function(req, res) {
06.
07.   // Aqui recuperamos um cliente do banco de dados
08.   // para execução das consultas.
09.   pg.connect(constr, function(err, client, done) {
10.     // Se houve um erro, para tudo e retorna.
11.     if(err) {
12.       done(client);
13.       res.statusCode = 500;
14.       return res.end('An error occurred: ' + err.message);
15.     }
16.
17.     var txtsql = 'SELECT \
18.                   pessoa.cod, \
19.                   pessoa.nome, \
20.                   pessoa.nasc, \
21.                   cidade.nome AS cidade \
22.                   FROM pessoa INNER JOIN cidade \
23.                   ON pessoa.cidade = cidade.cod \
24.                   AND cidade.cod IN ($1, $2, $3) \
25.                   LIMIT 10;';
26.
```

[10] Dentro dos limites de conhecimento do autor.

```
27.     // Executa o prepared statement.
28.     client.query({
29.       text: txtsql,
30.       values: [1, 2, 3],
31.       name: 'myquery'
32.     }, function (err, result) {
33.       // Se houve erro na consulta, fecha o cliente.
34.       done(err ? client : false);
35.       // E manda um HTTP 500 (Internal Server Error) para os
36.       // clientes HTTP (navegador, apache bench, jmeter, etc).
37.       if (err) {
38.         console.error(err);
39.         res.statusCode = 500;
40.         // Para a execução da requisição atual.
41.         return res.end(err.message);
42.       }
43.       // Manda o cabeçalho HTTP para html.
44.       res.writeHead(200, {'content-type': 'text/html'});
45.       // Monta o resultado em uma tabela html.
46.       var html = result.rows.reduce(function (html, row) {
47.         return html += "<tr><td>" +
48.           row.nome + "</td><td>" +
49.           row.cidade + "</td><td>" +
50.           row.nasc.toLocaleString() + "</td></tr>";
51.       }, "<table border='1'>");
52.       // Manda o html, encerrando a requisição HTTP atual.
53.       res.end(html + "</table>");
54.     });
55.   });
56. });
57.
58. server.listen(8080, '127.0.0.1');
```

3.9.2 Explicação do código

Durante o texto desta Seção são feitas algumas menções a códigos de *status* HTTP. Mais informações sobre esses códigos podem ser encontrados na RFC 2616, disponível em http://tools.ietf.org/html/rfc2616.

Dar-se-á início a explicação do código pelas três primeiras linhas do programa, mostradas abaixo.

```
01. var http = require('http');
02. var pg = require('pg');
03. var constr = 'postgres://devel:1234@127.0.0.1/tcc';
```

A linha 01 importa o módulo `http`, que é um módulo oficial, não sendo necessário instalar pelo NPM. O módulo é importado e um objeto já é automaticamente criado, o qual é assinado a variável `http`.

Na linha 02, é importado o módulo `pg`[11], o qual deve ser instalado com o comando a seguir:

```
npm install pg
```

O módulo é instalado em um subdiretório chamado `node_modules` na raiz do projeto, e nada mais é necessário para que possa ser utilizado.

Em seguida, na linha 03, é construída a *string* que será posteriormente utilizada para a conexão com o banco.

```
03. var server = http.createServer(function(req, res) {
```

A linha 05 é responsável criar o servidor HTTP, o qual recebe uma função *callback* com uma referência para o objeto `request` e outra para o objeto response. O objeto `request` possui informações sobre a requisição efetuada pelo cliente (não foi necessário utilizar o objeto `request` para a aplicação em questão), e o objeto `response` é utilizado para enviar dados para o cliente, bem como para sinalizar o fim da comunicação.

```
09.   pg.connect(constr, function(err, client, done) {
10.     // Se houve um erro, para tudo e retorna.
11.     if(err) {
12.       done(client);
13.       res.statusCode = 500;
14.       return res.end('An error occurred: ' + err.message);
15.     }
```

O bloco de código acima é responsável por criar uma conexão com o banco, mais especificamente na linha 09, onde é utilizada a *string* com os dados da conexão. Aqui, mais uma vez, é utilizada uma função *callback*. Os parâmetros dessa função são um objeto de erro (que é `null`, caso não ocorram erros), o cliente com o banco (que será utilizado a seguir) e ainda outra função *callback*, que recebeu o nome `done`. Ela será chamada após consulta SQL ser executada, de modo a fechar a conexão com o PostgreSQL.

É importante ressaltar que função `done` é uma variável (na verdade, todos os tipos não

[11] https://www.npmjs.org/package/pg

primitivos são referências em JavaScript) como qualquer outra (ver 2.1.5, Funções, objetos de primeira classe). Ela é passada como parâmetro, e quando for necessário executar essa função, basta colocar parênteses após o nome: `done()`. Esta maneira de utilizar funções é característica de linguagens funcionais e foi explicado em 2.1.5. Finalizando o bloco iniciado na linha 09, se houve um erro, mostra-se o erro no console do servidor, fecha-se a conexão atual com o banco, configura-se a resposta para o código HTTP 500, e a requisição é encerrada, informando que houve um erro.

```
17.     var txtsql = 'SELECT \
18.                   pessoa.cod, \
19.                   pessoa.nome, \
20.                   pessoa.nasc, \
21.                   cidade.nome AS cidade \
22.                   FROM pessoa INNER JOIN cidade \
23.                   ON pessoa.cidade = cidade.cod \
24.                   AND cidade.cod IN ($1, $2, $3) \
25.                   LIMIT 10;';
```

O código acima constrói a consulta SQL. A *string* SQL da consulta foi quebrada em diversas linhas para facilitar a leitura (caso contrário a linha seria extremamente longa). As barras invertidas no final das linhas são utilizadas para dizer ao motor JavaScript que a *string* continua na próxima linha (herança da linguagem C, onde é possível *escapar new lines*). Isto não é necessário em PHP (basta pressionar Enter, continuar digitando, e fechar a aspa quando for necessário), e em Java não é sequer possível, sendo necessário concatenar a *string* diversas vezes (algo computacionalmente caro, pois *strings* são imutáveis em Java, sendo que cada concatenação força a criação de um novo objeto, onde o objeto da *string* antiga fica elegível para ser destruído pelo *garbage collector*)[12] ou utilizar uma classe como a StringBuilder (também acarretando mais custos computacionais). Caracteres de escape como **`\`** ou **`\n`** funcionam tanto dentro de aspas simples quanto de aspas duplas em JavaScript e consequentemente com Node.js. Embora *strings* sejam imutáveis em JavaScript também, temos essa possibilidade de evitar concatenações desnecessárias.

Os símbolos `$1`, `$2` e `$3` são os *placeholders* (característicos de consultas parametrizadas e *prepared statements*), os quais são parâmetros que serão substituídos posteriormente, quando da execução da consulta.

---

[12] https://docs.oracle.com/javase/tutorial/java/data/strings.html

```
28.     client.query({
29.       text: txtsql,
30.       values: [1, 2, 3],
31.       name: 'myquery'
```

O bloco acima efetivamente cria o *prepared statement*, fazendo uso da *string SQL*, um *array* com os valores que serão substituídos pelos *placeholders* da *string*. A linha 31 atribui um nome para o *statement.* Isso é um requisito do módulo pg[13] por razões que não fazem parte do escopo do presente documento.

```
32.     }, function (err, result) {
33.       // Se houve erro na consulta, fecha o cliente.
34.       done(err ? client : false);
35.       // E manda um HTTP 500 (Internal Server Error) para os
36.       // clientes HTTP (navegador, apache bench, jmeter, etc).
37.       if (err) {
38.         console.error(err);
39.         res.statusCode = 500;
40.         // Para a execução da requisição atual.
41.         return res.end(err.message);
42.       }
```

Após a criação da *prepared statement*, uma nova função *callback* é utilizada para dar continuidade ao programa. Nessa função, o parâmetro `err` é utilizado para denotar um possível erro (ou `null` se não houve erro). O parâmetro `result` possui o resultado da consulta. Se houve erro, fecha-se o cliente (foi utilizado o condicional ternário para fazer o teste). Em seguida, também em caso de erro, configura o código HTTP 500 e encerra a requisição atual, enviando a mensagem de erro ao cliente.

```
44.       res.writeHead(200, {'content-type': 'text/html'});
45.       // Monta o resultado em uma tabela html.
46.       var html = result.rows.reduce(function (html, row) {
47.         return html += "<tr><td>" +
48.           row.nome + "</td><td>" +
49.           row.cidade + "</td><td>" +
50.           row.nasc.toLocaleString() + "</td></tr>";
51.       }, "<table border='1'>");
52.       // Manda o html, encerrando a requisição HTTP atual.
53.       res.end(html + "</table>");
```

A linha 44 utiliza o objeto `res` para enviar o cabeçalho HTTP com o código de *status* 200 (OK) e informar o cliente que o que conteúdo a ser enviado é do tipo HTML. A partir do

[13] https://github.com/brianc/node-postgres/wiki/Prepared-Statements

objeto `res`, é acessado o método `reduce`[14] no *array* `rows`. O resultado é uma *string* contendo os campos do banco dentro de uma tabela HTML. Finalmente, na linha 53 o objeto `res` é novamente utilizado, mas desta vez para chamar o método `end`, o qual é utilizado para encerrar a conexão. Este método aceita uma *string* como parâmetro, sendo possível enviar os dados e encerrar a conexão no mesmo comando. Isso produz o mesmo efeito de fazer `res.write(dados)` seguido de `res.end()`.

```
58. server.listen(80, '127.0.0.1');
```

Na linha 58, o servidor é iniciado na porta 8080, *localhost*. A partir desse ponto quaisquer requisições serão atendidas pelo servidor. Para suportar IPv6, basta substituir `'127.0.0.1'` por `'::1'`.

Finalmente, é possível observar que várias variáveis são definidas em um escopo, e utilizadas em funções mais internas sem que sejam passadas como parâmetro. Esse foi o caso das variáveis definidas nas primeiras três linhas (`http`, `pg` e `constr`) e as variáveis `res`, `err`, `client` e `done` que são definidas em um escopo mais externo, e utilizadas livremente dentro dos escopos mais internos. Esse tipo de comportamento da linguagem nos remete ao conceito de *closures*, discutido na Seção 2.1.8.

3.9.3 Considerações

Analisando-se o código exposto para o exemplo de servidor com conexão e consulta a banco de dados, algumas características podem ser observadas, as quais serão discutidas a seguir.

Primeiramente, conforme foi mencionado na introdução deste Capítulo, o código não foi escrito pensando em organização e modularização. O código foi escrito pensando primariamente em atender requisições HTTP, consultar o banco de dados e enviar o resultado para os clientes da maneira mais simples e direta possível.

Há também que se observar como as características da linguagem JavaScript são realmente importante para programação orientada a eventos e ao uso da tecnologia Node.js. Por exemplo, onde, em uma outra linguagem qualquer, uma consulta ao banco seria algo como:

---

[14] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce

(paradigma dominante atualmente)

```
// 1.
result = banco.query('SELECT …');
// 2.
foreach (result as row) {
    // usa a row.
}
// 3.
// Milhões de ciclos no futuro, executa as próximas instruções.
```

Com Node.js, no entanto, a lógica muda:

(paradigma orientado a eventos com uso de funções *callback*)

```
// 1.
banco.query(function(result) {
    // 3. Quando o banco retorna o resultado, faz uso dele.
    foreach (result as row) { … }
});
// 2.
// Executa outras instruções enquanto aguarda pelo resultado.
```

Essa abordagem de programação não é tão linear quanto o paradigma atualmente dominante, e certamente requer um tipo de raciocínio diferente. Existe um artigo escrito por David Ungar (criador da linguagem Self), Craig Chambers, Bay-Wei Chang e Urs Hölzle (nomes que ajudaram a implementar e evoluir a linguagem Self) chamado Organizing Programs Without Classes[15] (Organizando Programas Sem Classes). Certamente uma excelente leitura e extremamente pertinente ao tópico abordado neste documento. A importância da linguagem Self para JavaScript (objetos atuam como protótipos e classes são uma camada desnecessária) foi descrita em 2.1.1.

Outro ponto que chama a atenção é o fato de que o número de funções *callback* utilizadas acaba por tornar o código um tanto quanto difícil de acompanhar:

```
http.createServer(function(req, res) {
    client.query(function(err, res) {
        res.rows.reduce(function(arg1, arg2) {
            // Mais situações com funções callback
            // poderiam ocorrer.
        });
    });
});
```

[15]http://bibliography.selflanguage.org/organizing-programs.html

O estilo de código resultante da utilização de inúmeras funções *callback* é conhecido como *Callback Hell* ou pirâmide da destruição (do inglês, *pyramid of the doom*, pois a indentação do código lembra o formato de uma pirâmide) (IHRIG, 2013, p. 31). Além do documento escrito pelos desenvolvedores da linguagem Self, há um *website* devotado a evangelizar boas práticas de programação com JavaScript (e consequentemente Node.js) no que diz respeito aos *Callback Hell* com o intuito de minimizar esse problema. O referido conteúdo pode ser visualizado acessando-se o link http://callbackhell.com/. O texto possui diversos links para outros artigos e documentos pertinentes ao assunto.

Outro fator a ser observado é que, pelo menos por enquanto, não existe muita literatura formal a respeito (livros e artigos acadêmicos). No entanto, a comunidade é bastante ativa e existe muita documentação em sites, blogs, cursos online, vídeos de conferências relacionadas a tecnologia e até mesmo palestras de desenvolvedores de grandes empresas como LinkedIn, Walmart e Paypal falando sobre como eles utilizam a tecnologia para resolver problemas específicos.

De um modo geral, pode-se dizer que desenvolver aplicações com Node.js realmente requer mentalidade e raciocínio diferentes, pois estamos diante de um paradigma de programação diferente. A experiência do autor durante todo o processo de criação das aplicações Node.js para os testes mostrou que esta tecnologia é tão fácil ou tão difícil de ser utilizada quanto for o conhecimento do desenvolvedor sobre ela (e isto envolve saber JavaScript). Apesar de óbvio, não custa ressaltar que para alguém treinado apenas em Java ou PHP, essa tecnologia terá uma curva de aprendizado maior do que para aqueles que já conhecem JavaScript mais a fundo, ou, pelo menos, alguma outra linguagem funcional ou orientada a eventos.

Com esta Seção, é encerrado o estudo sobre a Node.js e acredita-se que os objetivos expostos na Seção 1.2 tenham sido alcançados. Há, no entanto, outras questões que não foram abordadas no presente documento e poderiam ser objeto de futuros estudos:

- Segurança em aplicações desenvolvidas com Node.js;
- *Frameworks*;
- Criação de módulos em JavaScript[16] e em C/C++[17];
- Execução simultânea de mais de uma instância do servidor com o módulo *cluster*[18];

[16] http://quickleft.com/blog/creating-and-publishing-a-node-js-module

[17] http://nikhilm.github.io/uvbook/introduction.html

[18] http://nodejs.org/docs/latest/api/cluster.html. Executar instâncias simultâneas da aplicação Node.js.

- Utilização de *web workers*[19].
- Utilização do módulo *Socket.IO*[20];
- Otimização de aplicações em Node.js;

Em fim, não faltam assuntos em torno da plataforma Node.js e há muito mais para ser explorado. A curiosidade vontade de aprender são o limite.

---

19 Execução de scripts em threads no background.
https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers,
https://www.npmjs.org/package/webworker-threads.

20 http://socket.io/, Utilizado para comunicação em tempo real entre um servidor Node.js com a aplicação rodando no cliente através emissão e captura de eventos.

4 CONSIDERAÇÕES FINAIS

Como foi mencionado na Seção 1.1, Motivação, tecnologias vão e vem, e parece que Node.js é uma das tecnologias que irá nos acompanhar por um bom tempo, pois já está sendo utilizada por grandes empresas como Paypal, LinkedIn e Walmart, e os próprios testes de avaliação mostraram que não estamos diante de uma tecnologia inferior. Durante os estudos e pesquisas realizados e mesmo a implementação e execução sobre casos de estudo utilizados para o presente trabalho, foi possível perceber que a plataforma Node.js é uma tecnologia que, apesar de nova, é potencialmente tão robusta quanto qualquer outra tecnologia já consolidada no mercado. O fato de utilizar JavaScript não desmerece nem um pouco a plataforma, até porque Google investe bastante em otimizar a performance e robustez do motor JavaScript V8, o qual é a base do Node.js, e devemos lembrar também que a própria linguagem JavaScript é desenvolvida sobre um padrão: o ECMAScript (ver seção 2.1.1).

Pode-se perceber que o trabalho fica bastante atrelado à linguagem JavaScript, por momentos podendo causar a impressão que o foco é mais a linguagem do que a plataforma Node.js propriamente dita. Neste quesito, é importante observar, conforme foi mencionado em 2.2.5.1, que o próprio criador da Node.js disse após tentar outras linguagens, JavaScript foi a linguagem cujas características se mostraram mais adequadas para desenvolver o que ele tinha em mente. Durante a demonstração de exemplos em código, é possível perceber como essas características são realmente importantes para possibilitar o entendimento e mesmo a programação com Node.js.

Quanto a questão de performance, especificamente, os testes comprovam o que a própria tecnologia afirma: foi desenvolvida com aplicações I/O-*bound* em mente, e não é aconselhável tentar utilizá-la para aplicações que façam uso intensivo do processador. Há que se considerar também que aplicações de médio e grande porte raramente fazem uso de apenas uma única linguagem de programação ou tecnologia, mas uma combinação de tecnologias diversas onde cada uma é utilizada na área (ou nas áreas) onde se sai melhor. Portanto, seria inconcebível pensar que Node.js é a solução ideal para tudo.

Outro fator interessante é que Node.js possibilita aos desenvolvedores conhecer um outro paradigma de programação. Trabalhar com uma linguagem funcional e orientada a eventos no lado do servidor é uma experiência bastante enriquecedora que possibilita aprender e criar soluções novas para resolver problemas antigos. Enfim, está aí uma

tecnologia que vale a pena estudar e pode perfeitamente ser utilizada em aplicações específicas.

A opinião pessoal do autor é que a tecnologia tem um grande potencial. O fato não ser necessário a utilização de *threads* é tanto um ponto positivo quanto negativo, dependendo do ponto de vista ou do tipo de aplicação. Não necessitar a utilização de *threads* tem seus benefícios, já que o conhecimento e utilização prática delas não é um assunto trivial (tanto que foi o motivo pelo qual Brendan Eich não as introduziu na linguagem JavaScript, como mencionado na Seção 2.2.1). No entanto, não deixa de ser uma restrição, pois *threads* são úteis para certas situações e um pré-requisito para outras, sem contar o fato de que qualquer programador sério deve conhecê-las. A Node.js, porém, por fazer uso eficiente do *event loop* torna o uso de *threads* desnecessário para aplicações IO-*bound*, e isso é um ponto forte.

Por outro lado, se trabalhar com *threads* e as dificuldades que elas apresentam (*deadlocks*, corridas de concorrência e proteção da região crítica da memória) é algo não trivial, trabalhar com um modelo de programação assíncrono induz a outro conjunto de dificuldades, pois o programa não é executado linearmente, e não se tem controle sobre quando uma linha de código será executada, o que causa problemas quando uma parte de código de pende do resultado da execução de uma outra parte (algo similar ao que acontece com a programação com *threads*). Ou seja, algumas dificuldades são evitadas, mas com o preço de se introduzir outras, embora as vezes menos severas.

Conclui-se, assim, o estudo proposto. Os objetivos mencionados na seção 1.2 foram alcançados e espera-se que este documento possa ser usado por outras pessoas que necessitem aprender sobre a plataforma Node.js e servir como ponto de partida e referência para futuros estudos na área.

REFERÊNCIAS

A SWIFT tour. iOS Developer Library. Disponível em: <https://developer.apple.com/library/prerelease/ios/documentation/swift/conceptual/swift_programming_language/GuidedTour.html#//apple_ref/doc/uid/TP40014097-CH2-XID_1>. Acesso em: 14 jun. 2014.

ABOUT EventMachine. Disponível em: <http://eventmachine.rubyforge.org/>. Acesso em: 10 mai. 2014.

ANONYMOUS functions. PHP Manual. Disponível em: <http://php.net/manual/en/functions.anonymous.php>. Acesso em: 5 dez. 2014

ANYEVENT. Disponível em: <http://search.cpan.org/dist/AnyEvent/lib/AnyEvent.pm>. Acesso em: 10 mai. 2014.

APACHE mpm worker, Multi-Processing Module implementing a hybrid multi-threaded multi-process web server. Apache HTTP Server Project. Disponível em: <http://httpd.apache.org/docs/2.2/mod/worker.html>. Acesso em: 31 mai. 2014.

BERNARDI, Élder Francisco Fontana. Gerência do processador. *Notas de Aula*. 01 mar. 2012, 15 jun. 2012.

BERNARDI, Élder Francisco Fontana. Gerência do processador. *Notas de Aula*. 13 jul.. 2013, 08 dez. 2013.

CHAMPEON, Steve. JavaScript: How Did We Get Here? O'Reilly WEB Devcenter, 2001. Disponível em: <http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html>. Acesso em: 25 mai. 2014.

CLOSURES. Mozilla Developer Network. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures>. Acesso em: 03 abr. 2014.

CONCURRENCY model and Event Loop. Mozilla Developer Network. Disponível em <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/EventLoop>. Acesso em: 15 mar. 2014.

CROCKFORD, Douglas. *JavaScript*: The Good Parts. Gravenstein Highway North, Sebastopol: O'Reilly, 2008.

CROCKFORD, Douglas. *JS Everywhere*. JavaScript: The Most Misunderstood Programming Language. JS.Everywhere conference, Boston, 2011. Disponível em: <http://vimeo.com/31188962>. Acesso em: 04 abr. 2014.

CROCKFORD, Douglas. Yahoo! YUI Theater. Crockford on JavaScript, Chapter 2: And Then There Was JavaScript. Crockford on JavaScript Series. Yahoo! Inc. 2010. Disponível em: <http://yuiblog.com/crockford/>. Acesso em: 04 mar. 2014.

CROCKFORD, Douglas. Yahoo! YUI Theater. Douglas Crockford, The JavaScript

Programming Language. YUI Theater , 2007. Disponível em: <http://yuiblog.com/blog/2007/01/24/video-crockford-tjpl/>. Acesso em: 06 abr. 2014.

DAGGET, Mark. *Expert JavaScript*. New York: Apress, 2013.

DAHL, Ryan: Node.js: JSConf, 2009. Disponível em: <http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html>. Acesso em: 10 de mai. 2014.

DOWNLOADS. Disponível em: <http://nodejs.org/download/>. Acesso em: 19 mai. 2014.

ECMAScript, Documentation. ECMAScript. Disponível em: <http://www.ecmascript.org/docs.php>. Acesso em: 24 mai. 2014.

FUNCTIONS, and function scope. Mozilla Developer Network. Disponível em: <https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions_and_function_scope>. Acesso em: 12 mai. 2014.

DAHL, Ryan - History of Node.js. Phx Tag Soup, 2011. Disponível em: <http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/>. Acesso em: 10 mai. 2014.

EICH, Brendan. 2013. Disponível em <https://twitter.com/BrendanEich/status/36747660724758184>. Acesso em: 30 mar. 2014.

FLANAGAN, David. *JavaScript*: The Definitive Guide, Sixth Edition. Gravenstein Highway North, Sebastopol: O'reilly, 2011.

HYPERTEXT transfer protocol – HTTP/1.1, 1999. Disponível em: <http://www.ietf.org/rfc/rfc2616.txt>. Acesso em: 19 jun. 2014.

HTTP. Disponível em <http://nodejs.org/api/http.html>. Acesso em: 19 jun. 2014.

INTRODUCTION to node.js with Ryan Dahl. Disponível em: <https://www.youtube.com/watch?v=jo_B4LTHi3I>. Acesso em: 19 jun. 2014.

IHRIG, Colin. Pro Node.js for Developers. New York: Apress, 2013.

JAVA, SE 8: Lambda Quick Start. Oracle. Disponível em <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>. Acesso em 12 abr. 2014.

JDK, 8 is released. Oracle, 2014. Disponível em: <https://blogs.oracle.com/thejavatutorials/entry/jdk_8_is_released>. Acesso em: 12 abr. 2014.

JAVASCRIT, The Right Way. Disponível em <http://jstherightway.org/>. Acesso em: 12 abr. 2014.

KUNKLE, Jeff: Node.js Basics Explained. 2012. Disponível em: <https://www.altamiracorp.com/blog/employee-posts/nodejs-basics-explained>. Acesso em: 10 mai. 2014.

LEARN More About Lamba Expressions. Scott Hommel, 2014. Disponível em: <https://blogs.oracle.com/thejavatutorials/entry/learn_more_about_lambda_expressions>. Acesso em: 14 jun. 2014.

NET. Disponível em <http://nodejs.org/api/net.html>. Acesso em: 21 jun. 2014.

NODE.JSa. Disponível em: <http://nodejs.org/>. Acesso em: 03 abr. 2014.

NODE.JSb v0.10.29 Manual & Documentation. Disponível em: <http://nodejs.org/api/modules.html#modules_core_modules>. Acesso em: 17 jun. 2014.

NPM node packaged modules. Disponível em <https://www.npmjs.org/>. Acesso em : 17 jun. 2014.

OBJECT, .create(). Mozilla Developer Network. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create>. Acesso em: 12 abr. 2014.

RESIG, John. *Secrets Of The JavaScript Ninja*. Shelter Island, NY: Manning, 2013.

RESIG, John. *Pro JavaScript Techniques*. New York: Apress. 2006.

REPL. Disponível em: <http://nodejs.org/api/repl.html>. Acesso em: 19 jun. 2014.

STEFANOV, Stoyan. *JavaScript Patterns*, First Edition. Gravenstein Highway North, Sebastopol: O'reilly, 2010.

STRING.prototype.trim(). Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/Trim>. Acesso em: 17 jun. 2014.

TILKOV, Stefan; VINOSKI, Steve. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, v. 14, n. 6, p. 80-83, nov. 2010.

TRIM Method (String) (JavaScript). Disponível em: <http://msdn.microsoft.com/en-us/library/ie/ff679971%28v=vs.94%29.aspx>. Acesso em 17 jun. 2014.

UNSUPORTED, Historical Releases. Disponível em: <http://php.net/releases/>. Acesso em: 12 abr. 2014.

VALUES, variables, and literals. Mozilla developer Network. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals#Object_literals>. Acesso em: 05 abr. 2014.

WHAT, is Twisted. Disponível em: <https://twistedmatrix.com/trac/>. Acesso em: 10 mai. 2014.

WORKING with objects. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects>. Acesso em: 05 abr. 2014.