Programação Orientada a Eventos no lado do servidor utilizando Node.js

Francisco de Assis Ribeiro Junior (fjunior87@gmail.com/francisco.ribeiro@ifactory.com.br)
Ifactory Solutions, Fortaleza -CE

***Resumo* — A disponibilização de sistemas através da WEB trouxe a à tona uma problemática não existente anteriormente: como escalar sistemas com milhares, ou até milhões de usuários simultâneos. O modelo baseado em *threads* se mostra ineficiente em situações onde há várias solicitações simultâneas. Com isso, um modelo baseado em eventos tem se mostrado mais eficiente para situações onde há várias solicitações simultâneas e grande demanda por operações de *I/O*. *Node.js* é um *framework* que foi construído com o intuito de facilitar a construção de sistemas orientados a eventos, altamente escaláveis e com alto desempenho. O presente trabalho pretende mostrar conceitos de programação orientada a eventos, bem como mostrar a utilização do Node.js como ferramenta para tal modelo de programação.**

***Palavras-chave* — Node.js, Programação Orientada a Eventos, Assíncrono.**

***Abstract* — When the systems were made available through the WEB raised a kind of problem that was not experienced before, how to scale systems with thousand or even million concurrent users. The thread based model has been shown inefficient in situations where there are too many concurrent requests. Because of that, the event based model has been shown more efficient in situations where there are many concurrent requests and a high demand for I/O operations. Node.js is a framework that was built in order to make easy the building of high scalable and performatic event-driven systems. This article intends to present some concepts of event driven programming and to present the use of Node.js as tool for using this programming model.**

***Keywords* — Node.js, Event-Driven Programming, asynchronous.**

I. INTRODUÇÃO

Nos dias atuais, a Internet é um dos principais meios para disponibilização de dados, produtos, serviços, entre outros. Além de trazer um número imenso de possibilidades de sistemas que podem ser desenvolvidos e disponibilizados através dela, a Internet, também traz grandes desafios na construção dos mesmos. Entre os principais desafios enfrentados nos sistemas voltados para Internet é a escalabilidade: como manter o sistema com um bom tempo de resposta, robusto e disponível, mesmo com uma alta demanda de milhões ou bilhões de usuários simultâneos[1].

Welsh at. al [1] mencionam que o aumento no número de acessos resulta num aumento nas operações que envolvem *Input/Output* e requisições a recursos disponibilizados através de uma rede.

A medida que a complexidade e o número de conexões simultâneas crescem, há a necessidade de utilização de diferentes técnicas para se alcançar a escalabilidade necessária.

Kegel [2] descreve o "*C10k problem*" no qual menciona que o limite de conexões simultâneas suportadas pela maioria dos *web servers* é de 10.000 (dez mil). Além disso, menciona que para suportar a mencionada quantidade de conexões simultâneas, o *hardware* não é o único gargalo, mas que tal poderia ser alcançado através da mudança no modo em que as operações de *I/O* são realizadas.

Muitos *softwares* se utilizam do modelo de *threads* para alcançar concorrência de *I/O* e outros recursos. Um dos principais aspectos que fazem do uso de *threads* a escolha inicial quando fala-se de concorrência é a preservação da aparência de programação serial [3]. Um outro aspecto importante que leva ao uso de *threads*, é a capacidade de utilização de multiprocessadores ou *cores* [4].

Segundo Welsh at. al[1], embora haja um certo grau de facilidade no desenvolvimento de aplicativos baseados em *thread*, muito por conta da adoção das mesmas em boa parte das linguagens de programação, o *overhead* associado as *threads* pode levar a uma degradação do desempenho à medida em que o número de *threads* aumenta. Isso se dá devido ao fato de que nesse modelo, cada requisição é atendida por uma *thread* separada, ou seja, quanto maior o número de conexões simultâneas, maior será o número de *threads* abertas.

Devido aos limites de escalabilidade impostos pelo modelo baseado em *threads*, muitos desenvolvedores têm optado por um modelo baseado em eventos para gerenciar concorrência [1].

Há vários sistemas desenvolvidos utilizando um modelo baseado em eventos, entre os quais é possível citar: *Nginx* [5], *G- Wan* [6] *web servers*. Há também ferramentas que podem ser utilizadas para criação de sistemas baseados em eventos, entre os quais é possível citar: *Python Twisted* [7] e o Ruby *Event Machine* [8].

O presente trabalho irá apresentar conceitos de programação orientada a eventos, e utilizará como ferramenta o *Node.js* [9], o qual será descrito nas seções subseqüentes.

II. PROGRAMAÇÃO ORIENTADA A EVENTOS

Em sistemas *web* convencionais, o modelo padrão utilizado é o *Request-Response*. Um componente cliente faz uma requisição (*request*) a um componente servidor ou *provider*, que recebe a requisição e produz uma resposta (*response*).

Em contrapartida, em programas construídos utilizando um paradigma orientado a eventos, tudo gira em torno de eventos. Evento é uma indicação de algo que aconteceu, consequentemente a terminologia utilizada para os participantes muda, agora existe o produtor do evento (*event producer*) e o consumidor do evento (*event consumer*) [10].

Uma das principais diferenças encontradas em sistemas baseados em eventos, é que em geral, o cliente (*event producer*) não espera pela ação a ser executada pelo servidor (*event consumer*) [1], ou seja, trata-se de uma interação *non-blocking*, pode-se até dizer assíncrona.

É possível exemplificar essa diferença com a seguinte situação: em programas síncronos, ao ser feita uma requisição em um banco de dados há a necessidade de esperar a requisição ser completada para então processar os resultados. Já em um programa assíncrono, ao ser feita uma requisição a um banco de dados, será especificado o que deve ser feito com os resultados da requisição. O programa não espera a finalização da requisição, e passa para outras atividades. E apenas quando o resultado é retornado, a lógica de manipulação dos resultados que foi especificado no momento da requisição é executada [11]. A essa lógica que é executada após a finalização da requisição dá-se o nome de *callback*.

Um exemplo de chamada assíncrona e a especificação de um *callback* pode ser visto na Figura 1.

```
getData("select * from some_table", function(results){
//Faça algo com os resultados retornados
});
//Não irá esperar completar a operação acima
readFromFile();
```

Figura 1. Exemplo de chamada assíncrona e *callback*

Na Figura 1, é possível ver uma chamada a função *getData*, esta função recebe dois parâmetros: uma instrução *SQL* a ser executada, e uma função de *callback*. Esta função de *callback* é que será executado no momento da finalização da consulta *SQL*. O ponto principal é que a chamada da função *getData*, não bloqueia a execução do restante do código, ou seja, a chamada da função *readFromFile* não irá esperar até a finalização da chamada de *getData*.

Diferente de programas baseados em *threads*, geralmente programas orientados a eventos, consistem de um único *loop*, chamado de *event loop*, que espera por eventos, e repassa os mesmos para o manipulador do evento. Programas baseados em eventos executam os *callbacks* serialmente, por esse motivo, não há preocupação com controle de concorrência [4]. Dabek at. AL [4] mencionam que por se tratar de uma único *loop*, programas baseados em eventos não usufruem de multiprocessamento ou *cores*, e para tal se faz necessário a execução de cópias do mesmo aplicativo.

Programação orientada a eventos é algo bem natural em aplicativos no lado do cliente (*client-side*), como por exemplo *interfaces* gráficas, nas quais programam-se quais as ações deverão ser realizadas após a ocorrência de algum evento, tais como clique em botões, redimensionamento de uma janela, dentre outros. Isto também é algo bem natural no desenvolvimento *web*, onde eventos como *onclick*, *onload*, dentre outros são bastante comuns.

Por outro lado, este não é um paradigma tão natural no lado do servidor (*Server-side*), por não lidar com as interações do usuário com os componentes da *interface* gráfica. Com isso, há a necessidade de aplicação de técnicas de programação diferentes das usadas quando se desenvolve de forma serial. Sendo isto, um dos principais aspectos negativos mencionados a respeito deste paradigma de programação [4].

Por se tratar de comunicação assíncrona, muito dos padrões descritos por Hohpe e Woolf [12] podem ser aplicados em programação orientada a eventos.

III. NODE.JS

Node.js é uma plataforma cujo objetivo é a fácil construção de rápidas e escaláveis aplicações de rede. Para tal emprega um modelo baseado em eventos, e *non-blocking I/O* [9]. Foi desenvolvido por Ryan Dahl em 2009. Trata-se de um ambiente *javascript* no lado do servidor, *single-threaded*, implementado em *C/C++* [13].

Aplicações *Node.js* podem ser escritas utilizando *javascript*, para tal, o *framework* faz uso da *Javascript Engine V8* do *Google* [14]. A *V8* é a implementação *javascript* utilizada pelo navegador *Google Chrome*, é extremamente rápida, mas necessitou de algumas modificações para que tenha um melhor funcionamento em outros contextos que não sejam o *browser* [15]. Para que se tenha *I/O* baseado em eventos e *non-blocking*, o *Node.js* faz uso de bibliotecas C *libev* [16] e *libeio* [17], desenvolvidas por Marc Lehmann.

O *Node.js* é composto por diversos módulos: módulos que fazem parte do núcleo da plataforma, chamados *core modules*, e módulos desenvolvidos pela comunidade. A arquitetura padrão da plataforma pode ser vista na Figura 2.

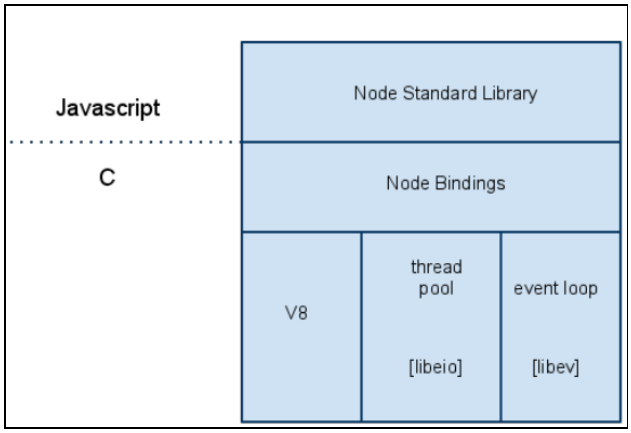

Figura 2. Arquitetura Node.js – [9]

O projeto Node.js teve influência de outros sistemas como o *Python Twisted* e o *Ruby Event Machine*. Uma das partes fundamentais da arquitetura Node.js é o *Event Loop*. O *event loop* é o método que o *javascript* utiliza para lidar com os eventos de uma melhor maneira [15]. O *Event Loop* permite que o *framework*, ao invés do sistema operacional, gerencie a mudança entre as tarefas a serem executadas [11]. Para o *event loop* o *Node.js* se utiliza da biblioteca *libev*.

A Figura 3. mostra um exemplo básico mostrado no *site* do *Node.js*, onde há a criação de um simples *Web Server*.

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

Figura 3. Web server feito com node.js

No código mostrado na Figura 3, é utilizado um dos módulos padrão do *Node.js*, o módulo *http*, que possui uma função para a criação do servidor. Esta função recebe como parâmetro uma outra função, esta é a função de *callback* que será executada a cada requisição feita ao *Web Server*. A resposta dessa requisição será basicamente a exibição da mensagem "*Hello World*".

Através deste exemplo, pode ser visto um dos principais motivos da escolha do *javascript* como linguagem utilizada, seu suporte a *callbacks* de eventos [13]. Isto se dá, pelo fato de que em *javascript* funções são "*first-class objects*", isso significa, que podem ser utilizadas como parâmetros em outras funções, como retorno de uma função, e atribuídas a variáveis [11]. Além disso, há a possibilidade de criação de funções anônimas, o que dá grande flexibilidade na criação de *callbacks*.

É possível observar a natureza assíncrona da *API* do *Node.js* através do exemplo mostrado na Figura 4.

```
var path = require("path"),
    fs = require("fs");

var filename =  "C:\\text.txt";
path.exists(filename, function(exists) {
    console.log("Running inside exists!");
    if(exists) {
        console.log("File exists!");
    } else {
        console.log("File doesn't exist!");
    }
});
console.log("After path.exists");
```

Figura 4. Exemplo de chamada assíncrona no Node.js

Ao executar o código mostrado na Figura 4. a saída gerada no console pode ser vista na Figura 5.

```
After path.exists
Running inside exists!
File doesn't exist!
```

Figura 5. Saída gerada pela execução do código de exemplo

Através da saída gerada no console, percebe-se a natureza *non-blocking* da *API Node.js*. A saída "*After path.exists*" não esperou a chamada a *path.exists* finalizar para ser executada, ou seja, a execução da aplicação não ficou bloqueada.

IV. EVENTOS COM NODE.JS

Um dos principais aspectos do *Node.js* é a questão do tratamento de eventos, ou seja, produção e consumo dos eventos. O *framework* provê um módulo específico para tratamento de eventos, o módulo "*events*". Este módulo disponibiliza uma classe genérica para manipulação de eventos, a classe *EventEmitter* [18].

Através da classe *EventEmitter* é possível adicionar um *callback* para tratar um determinado evento, através da função *addListener* ou *on*, bem como, lançar, emitir um determinado evento, através da função *emit*. Um exemplo de uso desta classe pode ser visto na Figura 6.

```
var event = require("events");
var eventEmitter = new event.EventEmitter();

//registrando um callback para o evento
eventEmitter.on("evento",function(){
    console.log("Callback associado ao evento!");
});

//Emitindo o evento
eventEmitter.emit("evento");
```

Figura 6. Utilização da classe *EventEmitter*

Este modelo de registro de *callbacks* associados a um determinado evento, e um criador de eventos para serem consumidos, pode ser definido como *Publish-Subscribe* [12]. Com isso, torna-se possível haver mais de um *callback* associado ao mesmo evento. Quando o evento ocorrer, o *node.js* irá executar todos os *callbacks* associados ao evento ocorrido. Isto pode ser visto na Figura 7.

```
var event = require("events");
var eventEmitter = new event.EventEmitter();

//registrando um callback para o evento
eventEmitter.on("evento",function(){
    console.log("Callback associado ao evento!");
});
//registrando um  novo callback para o evento
eventEmitter.on("evento",function(){
    console.log("Segundo Callback associado ao evento!");
    console.log("Sou executado depois do primeiro!!");
});

//Emitindo o evento
eventEmitter.emit("evento");
```

Figura 7. Mais de um *callback* associado a um mesmo evento.

No exemplo mostrado na Figura 7 há mais de um *callback* associado a um mesmo evento, quando este evento for emitido, todos os *callbacks* serão executados na ordem em que foram associados ao evento.

O *EventEmitter* é utilizado pela maioria dos módulos no *Node.js*. Os que precisam fazer uso de emissão de eventos, se utilizam desta classe. Isto pode ser feito de várias formas, dentre as quais podem ser citadas, o uso explícito do *EventEmitter* no código do módulo, ou através de herença, na qual a classe criada no módulo estende *EventEmitter*, dessa forma passa a possuir as funções definidas na mesma. Um exemplo da segunda opção pode ser visto através da Figura 8. Sendo esta a opção mais utilizada.

```
var events = require('events');
var util = require('util');

function Stream() {
  events.EventEmitter.call(this);
}
util.inherits(Stream, events.EventEmitter);
```

Figura 8. Classe herdando de *EventEmitter*

O Exemplo mostrado na Figura 8 foi retirado do código de um dos módulos do *Node.js*, o módulo "*stream*". Nela há uma função construtora *Stream*, e uma chamada a função *util.inherits*, que indica que a classe Stream está herdando de *EventEmitter*. Desta forma objetos criados utilizando a função construtora *Stream*, podem acessar funções definidas por *EventEmitter*, como *on*, *emit*, dentre outras.

V. Dificuldades com Node.js

Programação orientada a eventos se utiliza bastante de *callbacks*, como *javascript* permite a passagem de funções anônimas, não seria incomum encontrar códigos como o mostrado na Figura 9.

```
fs.open('results', 'w', function(err, fd) {
  fs.write(fd, results, function(err, written, f) {
    fs.close(fd, function(err) {
        done();
    });
  });
});
```

Figura 9. Código Sequencial com Node.js – [19]

Devido a sua natureza assíncrona, não há garantia na ordem de execução das chamadas efetuadas. O código acima tem a intenção de manter o controle na ordem de execução das chamadas. A medida em que se consegue manter o controle sobre a ordem de execução, acaba-se perdendo em legibilidade e manutenibilidade do código, gerando o chamado "*Spaghetti Code*".

Para solucionar essa problemática há uma grande quantidade de módulos desenvolvidos pela comunidade. Entre os quais podem ser citados, *Step*, *Flow-js*, dentre outros. É possível verificar a utilização da biblioteca Step na Figura 10.

```
Step(
  function readSelf() {
    fs.readFile(__filename, this);
  },
  function capitalize(err, text) {
    if (err) throw err;
    return text.toUpperCase();
  },
  function showIt(err, newText) {
    if (err) throw err;
    console.log(newText);
  }
);
```

Figura 10. Step – Exemplo de Uso

Verifica-se que através do uso do módulo *Step* obtém-se maior legibilidade no código, além de uma forma bem simples de executar atividades de maneira seqüencial. Um ponto a notar é que o framework não perde sua natureza assíncrona, o que o módulo faz é simplesmente passar a própria função como *callback*, dessa forma o módulo *Step* controla a ordem de execução.

Um outro ponto a ser trabalhado no desenvolvimento de aplicações com *Node.js*, é a questão da utilização dos vários processadores ou *cores* disponíveis. O *Node* executa em um único processo, ou seja, é *single-threaded*. Uma das soluções comuns utilizadas no mundo *Node.js* é a execução de múltiplas instâncias do processo [13]. Há um módulo desenvolvido pela comunidade para este problema específico, o *multi-node* [20]. Este módulo faz uso da capacidade dos sistemas operacionais em compartilhar *sockets* entre processos. Um exemplo de uso deste módulo pode ser visto na Figura 11.

```
var server = require("http").createServer(function(request, response){
        ... standard node request handler ...
    });
var nodes = require("multi-node").listen({
        port: 80,
        nodes: 4
    }, server);
```

Figura 11. Multi-node exemplo de utilização.

Por se tratar de um *framework* relativamente novo, há ainda diversos pontos a serem trabalhados para facilitar o desenvolvimento de aplicações utilizando o *Node.js*. A cada dia a comunidade desenvolve novos módulos visando atacar os pontos a serem melhorados no *framework*.

VI. Considerações Finais

Neste artigo é possível constatar como o paradigma de programação orientada a eventos pode ser utilizada no lado do servidor. Verifica-se também que o *Node.js* é uma ferramenta viável para construção de aplicações orientadas a eventos. Foi possível perceber também algumas dificuldades que podem ser encontradas no desenvolvimento das mesmas.

O *Node.js* é ainda um *framework* em processo de maturação, sendo assim, o uso do mesmo deve passar por um processo de estudo para verificar se o mesmo se aplica as necessidades da aplicação em questão.

Baseado nas referências citadas, é possível constatar que quando aplicado ao contexto correto e utilizando as melhores práticas e padrões para programação orientada a eventos, bem como *javascript*, *Node.js* pode ser utilizado para produzir sistemas orientados a eventos altamente escaláveis e com alto desempenho.

VII. Referências

[1] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", ACM Symposium on Operating Systems Principles, 2001.

[2] D. Kegel, *The C10K Problem*, http://www.kegel.com/c10k.html, 2011, acessado em Fevereiro de 2012.

[3] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazièresm and R. Morris, "Event-Driven Programming for Robust Software", n the Proceedings of SIGOPS European Workshop 2002, Saint-Emilion, France, 2002.

[4] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazièresm and R. Morris, “Multiprocessor Support for Event-Driven Programs ”, In the Proceedings of USENIX 2003, San Antonio, Texas, 2003.
[5] Nginx, “Nginx”, http://nginx.org/, 2012, acessado em Fevereiro de 2012.
[6] G-WAN, “G-WAN Web Application Server”, http://gwan.com/, 2012, acessado em Fevereiro de 2012.
[7] Twisted, “Twisted”, http://twistedmatrix.com/trac/, 2012, acessado em Fevereiro de 2012.
[8] Event Machine, “eventmachine @ Github”, http://rubyeventmachine.com/, 2012, acessado em Fevereiro de 2012.
[9] Node.js, “Node.js”, http://nodejs.org/, 2012, acessado em Fevereiro de 2012.
[10] O. Etzion, P. Niblett,, “Event Processing in Action”, Manning,, 2010.
[11] M. Cantelon, and TJ. Holowaychuk, “Node.js in action”, Manning, 2011.
[12] G. Hohpe, B. Woolf, “Enterprise Integration Patterns: Desigining, Building, and Deploying Messagin Systems”,Addison-Wesley,2003.
[13] S. Tikov, S. Vinoski. “Node.js: Using Javascript to Build Gugh Performance Network Programs ”. Internet Computing, IEEE ,2010.
[14] Google. “V8 javascript Engine”. http://code.google.com/p/v8/, 2012, acessado em Fevereiro de 2012.
[15] T. Hughes-Crouch. “Up and Running With Node.js”. O’Reilly, 2010.
[16] M. Lehmann. “Libev”. http://software.schmorp.de/pkg/libev.html, 2012, acessado em Fevereiro de 2012.
[17] M. Lehmann. “Libeio”. http://software.schmorp.de/pkg/libeio.html, 2012, acessado em Fevereiro de 2012.
[18] Node.js. “EventEmitter”. http://nodejs.org/docs/latest/api/events.html, 2012, acessado em Fevereiro de 2012.
[19] B. Michel. “Evented Programming and its Patterns”. http://nono.github.com/Presentations/20110923_Evented_Programming/, 2012, acessado em Fevereiro de 2012.
[20] K. Zyp. “Multi-Node”. httphttps://github.com/kriszyp/multi-node, 2012, acessado em Fevereiro de 2012.