THIAGO VIEIRA PULUCENO

ESTUDO DE CASO SOBRE UMA API REST UTILIZANDO A ABORDAGEM DE PROGRAMAÇÃO ORIENTADA E EVENTOS COM A PLATAFORMA NODE.JS

FLORIANÓPOLIS - SC
2012

THIAGO VIEIRA PULUCENO

ESTUDO DE CASO SOBRE UMA API REST UTILIZANDO A ABORDAGEM DE PROGRAMAÇÃO ORIENTADA E EVENTOS COM A PLATAFORMA NODE.JS

Trabalho de conclusão de curso apresentado para obtenção do Grau de Bacharel em Sistemas de Informação pela Universidade Federal de Santa Catarina.

ORIENTADOR:

VITÓRIO BRUNO MAZZOLA

DEPARTAMENTO DE INFORMATICA E ESTATÍSTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DE SANTA CATARINA

FLORIANÓPOLIS - SC

2012

SUMÁRIO

Lista de ilustrações

Lista de tabelas

Lista de símbolos, nomenclaturas e abreviações

API	Applications Programming Interface
BPM	Business Process Management
DOM	Document Object Model
ESB	Enterprise Service Bus
E/S	Entrada/Saída
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JMS	Java Message Service
JSON	JavaScript Object Notation
LAN	Local Area Network
MIME	Multipurpose Internet Mail Extension
MQ	Message Queue
REST	Representational State of Transfer
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP/IP	Transfer Control Protocol / Internet Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WS	Web Service
WSDL	Web Service Description Language
XML	eXtensible Markup Language

RESUMO

Estudo de caso sobre uma API REST utilizando a abordagem de programação orientada e eventos com a plataforma Node.js

O desenvolvimento deste trabalho é resultado da implementação de uma arquitetura híbrida para sistemas distribuídos, conhecida como REST - Transferência de Estado Representacional- e proposta por Fielding (2000), utilizando uma nova arquitetura de sistemas, chamada Node.js, onde o programa do lado do servidor é escrito na linguagem JavaScript além de ser orientado a um laço de eventos, isto é, novas threads não são criadas para cada nova requisição.

REST possui um estilo arquitetural orientado a recursos, enquanto que Node.js possui um estilo arquitetural orientado a eventos. A aplicação destes elementos permitiu desenvolver uma interface de comunicação entre quaisquer sistemas compatíveis com REST, através da disponibilização de recursos em um formato padrão e bem definido fazendo uso da Arquitetura Orientada a Recursos, e altamente escalável e veloz, fazendo uso da Arquitetura Orientada a Eventos.

Por fim, foi realizado um comparativo desta implementação, com uma implementação idêntica, porém escrita na linguagem JAVA (orientada a threads), a fim de avaliar os benefícios, onde e quando deve se fazer uso desta proposta.

Palavras-chaves: Arquitetura de software, REST, Arquitetura Orientada a Recursos, Node.js, Arquitetura Orientada a Eventos, JavaScript, escalabilidade.

1 Introdução

Aplicações Web são cada vez mais frequentes entre os incontáveis sítios espalhados pela Internet e podem ser implementadas utilizando-se um universo de padrões, ferramentas, tecnologias, arquiteturas e estilos arquiteturais disponíveis especificamente para tal tarefa.

Fielding (2000), em sua dissertação de doutorado, propõe um novo estilo arquitetural, chamado Representational State Transfer (REST). Embora REST possua um conjunto de restrições que o caracterizam como um estilo arquitetural, ainda permite muitas liberdades em sua aplicação a alguma arquitetura já existente. Por essa liberdade de aplicação, este estilo vem cada vez mais sendo utilizado por aquelas aplicações web, que com o decorrer do tempo necessitaram cada vez mais se comunicarem entre si. REST surge atualmente como uma forma rápida, eficiente e simples de solucionar esse problema de comunicação entre diferente sistemas construídos utilizando diferentes linguagens de programação.

Porém, como o volume de tráfego nestes sistemas tem crescido de forma vertiginosa, há uma busca incessantes para que possamos prover uma maior quantidade de serviços para mais usuário, a um custo menor de implementação e manutenção. Node.js surge no momento como uma opção muito atraente para solucionar este tipo de dificuldade, uma vez que suas aplicações são de fácil e rápido desenvolvimento, utiliza uma linguagem de programação que grande maioria dos programadores conhece, e é altamente escalar, permitindo maximizar a utilização dos recursos de um servidor.

1.1 Motivação

Em linguagens como Java e PHP, cada nova conexão inicia um novo encadeamento que potencialmente é acompanhado da utilização de dois Megabytes de memória. Em uma máquina que possua oito Gigabytes de memória RAM, isto define o número máximo teórico de conexões simultâneas em cerca de quatro mil usuários. Enquanto a base de usuários do nosso sistema cresce, nosso aplicativo web precisa fornecer suporte a mais usuários. Desta forma seria necessário

adicionar mais máquinas com mais servidores para suprir a demanda. Este é o gargalo de toda a arquitetura de aplicativos da web, o número máximo de conexões simultâneas que um servidor pode tratar.

O Node.js busca solucionar esta problemática mudando a forma como uma conexão é feita no servidor. E é com a implementação de uma API RESTful em Node.js que este trabalho propõe uma nova forma de tratarmos esta dificuldade atual.

1.2 Justificativa

Com a explosão da WEB, é notável o crescimento da automação das interações entre os web sites, aplicações web, banco de dados e toda a interconexão global. A tecnologia de web services tem o potencial de mudar a forma como a Internet funciona para as empresas. Embora atualmente os serviços web funcionem bem para empresas e consumidores, eles não o fazem para aplicações business-to-business. Por exemplo, um cliente individual pode facilmente comprar um livro de um vendedor on-line, mas para uma livraria querendo fazer uma compra de volume, é muito mais complicado. A livraria potencialmente precisaria usar vários aplicativos para acompanhar as vendas, determinar novas encomendas e acompanhar o transporte. Muitas vezes, os dados de um aplicativo precisam ser reinseridos em outros aplicativos, tornando todo o processo ineficiente. Web Services são capazes de resolver este problema.

A Arquitetura Orientada a Serviço tradicional (SOA) está se movendo lentamente para Web Oriented Architecture (WOA) (Dion Hinchcliffe), onde as aplicações utilizam uma rica rede de recursos REST. Ao invés de um ponto com alguns serviços, os dados da empresa serão expostos através de milhões de recursos granulares, como a própria web.

REST está se tornando rapidamente a tecnologia preferida para a construção de aplicações arbitrárias que se comunicam através da rede. REST aproveita totalmente protocolos e padrões que apoderam a Web, além de ser mais simples do que os tradicionais serviços Web baseados em SOAP. Com o surgimento da computação em nuvem e o crescente interesse em aplicações web, tecnologias

baseadas em REST podem ajudar tanto no desenvolvimento de interfaces ricas para os usuários utilizá-las de servidores remotos, quanto no desenvolvimento de servidores para manipular estruturas de dados numa aplicação do cliente (escrito em qualquer linguagem) ou diretamente no navegador.

Com todo este crescimento, Node.js oferece alto desempenho e altíssima escalabilidade para suas aplicações. Desta maneira, juntando a agilidade do REST com a escalabilidade e velocidade do Node.js, é possível criar Web Services de altíssimo desempenho, fornecendo recursos a uma quantidade de usuários muito além do que as arquiteturas convencionais.

1.3 Objetivos Geral e Objetivos Específicos

1.3.1 Objetivo Geral

Neste trabalho será apresentado uma breve introdução às duas arquiteturas aqui propostas. Elaborar um tutorial de como começar a construir aplicativos em Node.js, e compreender e propor uma interface RESTful, com o intuito de que esta sirva como ponto de partida para trabalhos futuros.

1.3.2 Objetivo Específico

Construir uma API REST em Node.js, com o intuito de averiguar a facilidade de construção, a velocidade de comunicação e a escalabilidade. Este trabalho deve servir para que trabalhos futuros o utilizem como um ponto de partida para desenvolvimento nesta nova arquitetura (Node.js), ou ainda como uma espécie de guia para que os conceitos aplicados tanto pelo Node.js quanto pela arquitetura REST possam ser aplicados em outras áreas.

Provar na prática a alta escalabilidade e suporte a um alto número de requisições simultâneas do Node.js, comparando a aplicação desenvolvida com uma aplicação semelhante escrita na linguagem JAVA.

1.4 Estrutura do Trabalho

Este trabalho está organiza em seis capítulos. Este capítulo introduziu a área de pesquisa, apresentando a motivação necessária para o desenvolvimento deste trabalho.

O capítulo 2 apresenta os principais conceitos relacionados às arquiteturas utilizadas na elaboração do projeto, incluindo histórico, casos de uso e quando ou não utilizar estes estilos arquiteturais.

No capítulo 3 estão descritos os elementos básicos para a construção da API proposta e também suas funcionalidades e métodos compreendidos. O capítulo 4 nos exibe como ocorreu esta implementação e quais seus resultados. Por fim, são apresentadas as conclusões e posteriormente as referências bibliográficas utilizadas ao longo deste documento.

2 Fundamentação Teórica

2.1 NODE.JS

2.1.1 Definição de Node.js

Node.js é muitas coisas, mas principalmente é uma maneira de rodar Javascript fora do navegador web. Podemos classificá-lo como uma arquitetura de sistemas, que dentre tantas, possui alguns objetivos específicos. Seu principal objetivo é permitir que fossem escritas aplicações altamente escaláveis para uso em rede, ou seja, na Internet.

Outras linguagens de programação que utilizam de diferentes arquiteturas, partem desta premissa, serem escaláveis. Porém Node.js é diferente, uma vez que ela utiliza a linguagem Javascript para escrever seus programas. A priori, podemos pensar que Javascript é uma linguagem de programação utilizada apenas para escrevermos programas que executarão no navegador do usuário que está visualizando nossa página, e que portanto não faz muito sentido escrevermos um sistema inteiro utilizando Javascript.

Mas Node.js é muito mais do que um simples interpretador de Javascript. Ele foi construído utilizando o interpretador de Javascript mais rápido da atualidade. O V8 é o interpretador de Javascript que é utilizado no navegador Chrome, do Google, e foi escrito em C++ pela equipe de desenvolvimento do Google Chrome.

A execução de Javascript no V8 é extremamente rápida e executa muito bem em diversas circunstâncias. O Node.js ajustou o V8 para trabalhar melhor em outros contextos além do navegador, principalmente, fornecendo APIs alternativa que são otimizadas para casos de uso específicos. Por exemplo, num contexto de servidor, a manipulação de dados binários é muitas vezes necessária. Isto é mal suportado pelo Javascript e, como resultado, no próprio V8. O Node.js adiciona a classe Buffer para suas implementações permitirem fácil manipulação de dados

binários, de uma maneira que se torna fácil seu uso, além de ser eficiente quando se trata de memória. O Node.js não apenas proporciona o acesso direto ao V8 em tempo de execução, como também o torna mais útil para os contextos em que as pessoas usam o Node.js.

O V8 utiliza das técnicas mais recentes de compiladores, permitindo que códigos escritos numa linguagem de alto nível, como o Javascript, possuam desempenho tão bom e custos reduzidos em relação a códigos escritos em linguagens de mais baixo nível como C, por exemplo. Este foco sobre o desempenho é um aspecto-chave do Node.js

Mas como o Node.js permite que minhas aplicações sejam escaláveis? Muitos programas escritos em diferentes linguagens utilizam threads ou até mesmo conseguem ser multitarefa, mas qual é o melhor que pode ser feito hoje? A resposta está na forma como o Node.js trata as operações de entrada e saída. Ele é direcionado a um laço de eventos, e as operações de entrada e saída não poderiam ser diferentes. Além de direcionadas a evento, as operações de entrada e saída não bloqueiam o sistema enquanto aguardam por uma resposta, do disco por exemplo.
Por que este tipo de configuração é ideal para o Node? O Javascript é uma excelente linguagem para programação direcionada a eventos, pois estes eventos permitem funções e fechamentos anônimos e mais importantes, a sintaxe é familiar para quase todos que alguma vez já programaram. As funções de call-back que são chamadas quando um evento ocorre podem ser escritas no mesmo local onde você captura o evento. Portanto, é fácil de codificar, fácil de manter, sem estruturas orientadas a objetos complicadas, sem interfaces e sem potencial para excessos na arquitetura. Basta aguardar um evento, escrever uma função de call-back e a programação direcionada a eventos toma conta de tudo. [Michael Abernethy]

Logo, de uma maneira sucinta, podemos dizer que o Node.js é uma maneira de escrevermos programas que possuirão altíssima concorrência, são altamente escaláveis e são puramente orientados a eventos que não bloqueiam a infraestrutura (banco de dados, por exemplo) a qual utilizam.

Apesar de não ser muito utilizada e conhecida no Brasil, o crescimento do Node.js é notável, sendo em janeiro de 2012 nomeada a Tecnologia do ano de 2011 pela Infoworld, e possuindo também o repositório mais acompanhando do Github (um servidor de repositórios na internet), tornando-o mais popular

internacionalmente do que Ruby on Rails (framework para programação na linguagem Ruby).

2.1.2 Histórico

Tudo começou em cinco de Janeiro de 2009, quando Ryan Dahl, o criador do Node.js teve algumas ideias sobre um novo webserver. Ele estava envolvido com trabalhos onde ele escrevia pequenos programas baseados a eventos. Ryan gostava do design dos servidores baseados a eventos, pois estes pareciam serem mais fáceis para compreender seu funcionamento, além do fato de estes programas não bloquearem as operações de entrada e saída.

Havia uma boa razão para a maioria das pessoas não desenvolver programas desta forma, apesar da simplicidade e eficiência: eles precisavam utilizar justamente as bibliotecas que realizam o bloqueio de entrada e saída. Programas baseados em eventos é uma proposição de tudo ou nada; você pode pegar várias operações de E/S em uma única thread do sistema operacional se você tiver cuidado para nunca bloqueá-la, mas se o fizer, então você irá bloquear todos os fluxos que estava manuseando. Usando uma thread para cada fluxo de E/S, é muito mais tolerante, pois não importa quão lento seu sistema é em uma dessas operações. Ryan queria criar um sistema onde as pessoas só poderiam utilizar operações de entrada e saída não bloqueáveis.

Originalmente, Ryan tentou desenvolver seu projeto utilizando outras linguagens de programação para escrever seus programas, como C, Lua e Haskell, mas a escolha pelo Javascript foi simplesmente por coincidência. O interpretador V8 foi lançado no mesmo período em que ele estava estudando que linguagem utilizar em seu projeto, então ele teve uma repentina ideia que o Javascript era realmente a linguagem perfeita para o que ele queria: uma thread, sem noções preconcebidas de "server-side" E/S e sem bibliotecas existentes.

A evolução do projeto aconteceu de maneira muito rápida. Rapidamente colaboradores começaram a surgir e o projeto Node.js começou a tomar forma de um projeto sério que poderia ser utilizado para vários fins e por várias grandes empresas.

Vejamos de forma rápida como a história se desenrolou:

- 5 de Janeiro de 2009: Ryan Dahl escreve suas novas ideias para desenvolver um novo tipo de webserver.
- 2 de Fevereiro de 2009: Ryan Dahl anuncia o novo projeto, sob o nome de Node.js, utilizando o interpretador de JavaScript V8 e uma arquitetura assíncrona muito rápida.
- 31 de Maio de 2009: É lançada a primeira versão do Node.js, versão 0.0.1.
- 30 de Junho de 2009: É lançada a versão 0.1.
- 8 de Novembro de 2009: Node.js é apresentado na JS Conference 2009 (Conferência anual sobre JavaScript), sob a versão 0.1.17.
- Março de 2010: A empresa Joyent começa a patrocinar o desenvolvimento do Node.js, contratando seu criador, Ryan Dahl.
- 20 de Agosto de 2010: É lançada a versão 0.2.
- 23 de Novembro de 2010: É lançada a versão 0.3.
- 10 de Fevereiro de 2011: É lançada a versão 0.4.
- 5 de Julho de 2011: É lançada a versão 0.5.
- 25 de Agosto de 2011: É realizada a primeira convenção unicamente sobre o Node.js.
- 4 de Novembro de 2011: É lançada a versão 0.6.
- 16 de Janeiro de 2012: É lançada a versão 0.7.
- A equipe de desenvolvimento do Node.js cresce de uma forma que seu criador, Ryan Dahl, deixa o time, prestando apenas suporte aos desenvolvedores e não mais escrevendo código propriamente dito.
- 15 de Junho de 2012: É lançada a versão 0.7.11, que é versão mais atual.

2.1.3 Funcionamento

Uma parte fundamental do Node.js é o ciclo de eventos, um conceito subjacente ao comportamento de JavaScript, bem como a maioria dos outros

sistemas interativos. Em muitas linguagens, modelos baseados em eventos são deixados de lado, mas os eventos JavaScript sempre formam uma parte essencial da maioria dos aplicativos web. Isso é porque o JavaScript sempre lidou com a interação do usuário. Qualquer um que tenha usado um navegador moderno está acostumado a páginas da web que fazem coisas "onclick", "onmouseover", etc. Estes eventos são tão comuns que dificilmente pensamos sobre eles ao desenvolvermos a interação da página web, mas ter esse suporte a eventos na linguagem é algo muito poderoso que podemos explorar de outras maneiras. No servidor, em vez de o conjunto limitado de eventos com base nas interações do usuário com o modelo de objeto de documentos da página web, temos uma variedade infinita de eventos com base no que está acontecendo no software de servidor que usamos. Por exemplo, o módulo HTTP do servidor fornece um evento chamado "request" (requisição), emitido quando um usuário envia para o servidor web um pedido.

O ciclo de eventos é o sistema que o Javascript utiliza para lidar com essas solicitações de entrada de várias partes do sistema de uma forma válida. Há várias maneiras como as pessoas lidam com problemas em tempo real ou paralelização na computação. Javascript tem uma abordagem simples que faz com que o processo se torne muito mais compreensível, mas introduz algumas restrições.

Node.js faz com que a abordagem de todas as atividades de entrada e saída sejam não bloqueáveis. Isto significa que as solicitações HTTP, consultas de banco de dados, operações de entrada e saída, e outras coisas que exigem que o programa espere, não interrompam a execução até que eles retornem os dados correspondentes. Assim, eles executam de forma independente, e depois emitem um evento quando seus dados estão disponíveis. Isto significa que a programação em Node.js tem muitas chamadas de call-backs que lidam com todo o tipo de E/S. Call-backs frequentemente iniciam outros call-backs em forma de cascata, que é muito diferente da programação para o browser. Há ainda uma certa quantidade de configuração linear, mas a maior parte do código envolve lidar com call-backs.

Devido a este estilo de programação pouco familiar, é preciso procurar padrões para nos ajudar a programar no servidor de forma eficaz. Isso começa com o ciclo de eventos. Pensamos que a maioria das pessoas intuitivamente entende programação orientada a eventos, porque é como a vida cotidiana. Imagine que

estamos cozinhando. Estamos cortando um tomate, quando uma panela começa a transbordar. Paramos de cortar aquele pedaço de tomate, e depois abaixamos o fogo, ao invés de tentar cortar o tomate e abaixar o fogão ao mesmo tempo. Assim, alcançamos o mesmo resultado de uma forma muito mais segura e eficiente, apenas mudando de contexto. A programação orientada a eventos faz a mesma coisa. Ao permitir que o programador escreva código que só funciona em um retorno de chamada de cada vez, o programa é compreensível e também capaz de executar rapidamente muitas tarefas de forma eficiente.

Na vida cotidiana, estamos acostumados a ter todos os tipos de call-backs internos para lidar com eventos, e ainda, como JavaScript, sempre fazemos apenas uma coisa ao mesmo tempo. É ótimo para deixar os eventos conduzir a ação, mas é "single-threaded" (uma thread), de modo que apenas uma coisa acontece ao mesmo tempo. Este conceito de single-threaded é realmente importante. Uma das críticas dirigidas à Node.js com bastante frequência é a falta de "concorrência". Ou seja, ele não usa todas as CPUs em uma máquina para executar o JavaScript. O problema com a execução de código em várias CPUs uma vez é que ele requer a coordenação entre vários fluxos de execução. Para que múltiplas CPUs possam efetivamente dividir o trabalho, elas teriam que “conversar” com cada outra sobre o estado atual do programa, o trabalho que cada uma tinha feito, etc. Embora isso seja possível, é um modelo mais complexo que exige mais esforço tanto do programador quanto do sistema. A abordagem JavaScript é simples: há apenas uma coisa acontecendo de uma só vez. Tudo que o Node.js faz é não-bloqueável, assim o tempo entre um evento ser emitido, e o Node.js ser capaz de agir naquele evento, é curtíssimo, justamente pelo fato de não haver tempo ocioso com operações de E/S.

Outra maneira de pensar sobre o ciclo de eventos é compará-lo a um carteiro. Para nosso carteiro, cada carta é um evento. Ele tem uma pilha de eventos para entregar em ordem. Para cada carta (evento) o carteiro caminha por uma rota para entregá-la. A rota, é a função de retorno de chamada atribuído a esse evento. No entanto nosso carteiro tem apenas um único par de pernas, logo ele só pode andar apenas por um caminho único de cada vez. Às vezes, enquanto o carteiro está caminhando por uma rota, alguém lhe entrega outra carta. Neste caso, o carteiro entrega a nova mensagem imediatamente (afinal, se alguém lhe entregou diretamente, ao invés de ir aos correios, deve ser uma mensagem urgente). O

carteiro vai sair da sua rota atual e caminhar por outro caminho adequado para entregar a nova carta. Após entregá-lo, ele então retorna para o ponto onde estava quando recebeu a nova carta e continua trilhando o caminho original emitido pelo evento anterior.

Vamos analisar o comportamento do nosso carteiro em um programa típico, imaginando algo simples. Suponha que temos um servidor web (HTTP) que recebe pedidos, recupera alguns dados a partir de um banco de dados, e os retorna para o usuário. Neste cenário, temos alguns eventos para lidar com eles. Primeiro (como na maioria dos casos), há o evento “request” do cliente requisitando a exibição de uma página web. O retorno de chamada que lida com o pedido inicial (vamos chamá-lo de call-back A) olha para a requisição do cliente e descobre quais os dados que necessita do banco de dados. Ele em seguida, faz um pedido ao banco de dados requisitando aqueles dados necessários para a exibição da página, passando outra função call-back, B, para ser chamado em caso de resposta do banco de dados. Depois de tratado o pedido, um call-back retorna. Quando o banco de dados encontrou os dados necessários, emite o evento resposta. O ciclo de eventos em seguida, chama call-back B, que envia os dados para o usuário. Isso parece bastante simples. A coisa óbvia a notar aqui é o "break" no código, que você não teria em um sistema processual. Devido ao fato de Node.js ser um sistema não-bloqueável, quando chegarmos ao momento de esperar uma resposta do banco de dados, nós simplesmente não esperamos, e retornamos por uma função de call-back. Isto significa que diferentes funções devem começar a processar o pedido e terminar de manuseá-lo quando os dados estiverem prontos para voltar. Então, precisamos nos certificar de passar qualquer estado que precisamos para o call-back ou disponibilizá-lo de alguma outra forma. A programação Javascript normalmente o faz através de encerramentos.

Por que isso faz o Node.js mais eficiente? Imaginemos que estamos num restaurante e realizamos um pedido para o garçom. O garçom por sua vez, pode se comportar de duas maneiras, uma delas é orientada a eventos, e a outra não. Vamos começar com a típica abordagem adotada pelo PHP, Java e muitas outras plataformas web. Quando fizermos nosso pedido ao garçom ele levará a ordem para a cozinha, mas não servirá outros clientes até que nossa ordem tenha sido concluída pela cozinha para que ele possa nos entregar. Existem algumas coisas que ele pode

fazer depois que ele enviou nossa ordem: processar parcialmente a conta, servir a bebida, atender outras pessoas, e assim por diante. No entanto, o garçom ainda vai ter que esperar uma quantidade desconhecida de tempo para a cozinha preparar nosso prato. Se num restaurante ocorresse como na abordagem tradicional da web, onde para cada garçom é atribuído a apenas um pedido de cada vez, a única maneira de escalar este processo seria adicionando mais garçons. No entanto, também é muito óbvio que nosso garçom não está sendo muito eficiente. Ele está gastando muito tempo, esperando que a cozinha prepare os alimentos.

Obviamente, na vida real os restaurantes utilizam um modelo muito mais eficiente. Quando um garçom terminou de tomar nosso pedido, receberíamos um número, onde poderíamos utilizá-lo para chamar o garçom novamente. Vamos chamar este número de retorno. É assim que o Node.js funciona. Quando as coisas lentas, tais como E/S iniciam, o Node.js simplesmente lhes dá uma referência de retorno de chamada e, em seguida, fica com outro trabalho que pode ser executado (enquanto não recebe a chamada de retorno), como o próximo cliente (ou evento, no caso do Node.js). Ao agir de forma orientada a ventos, os garçons são capazes de maximizar o seu rendimento.

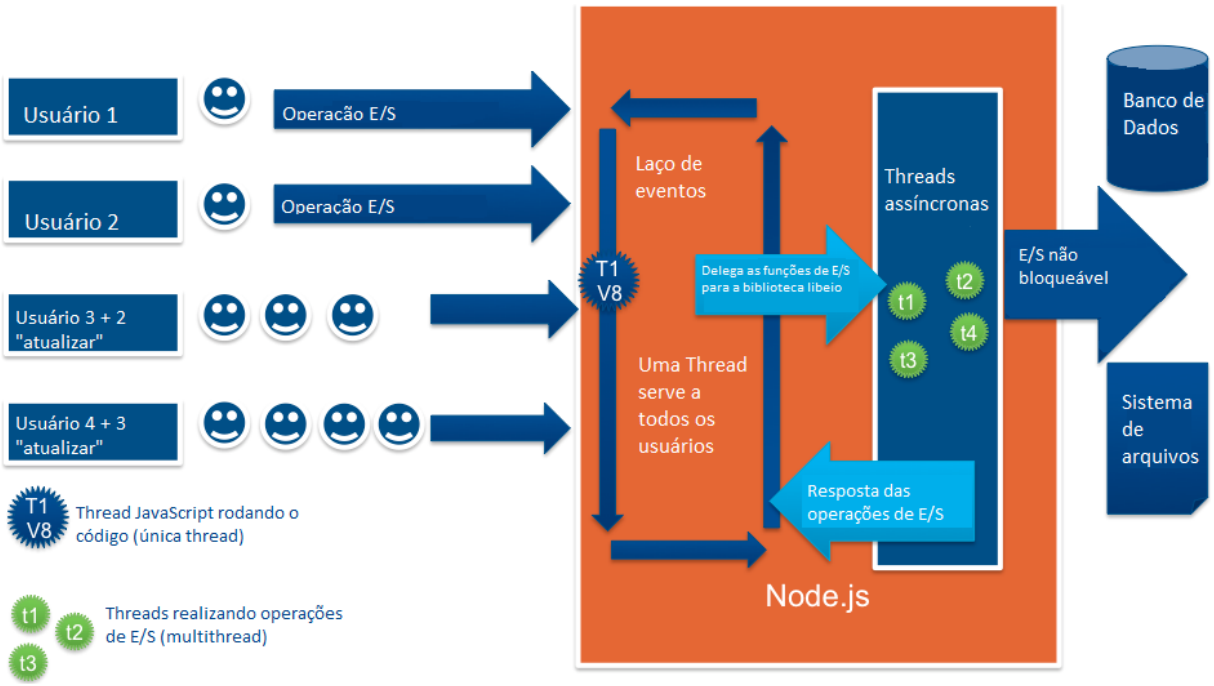

FIGURA 1 – Esquema de funcionamento do Node.js

Esta analogia ilustra também os casos onde está o Node.js se encaixa bem e aqueles em que não se encaixa bem. Em um pequeno restaurante onde o pessoal da cozinha e os garçons são as mesmas pessoas, nenhuma melhoria poderia ser feita tornando-os orientados a eventos. Como todo o trabalho está sendo feito pelas mesmas pessoas, a arquitetura orientada a eventos não acrescenta nada. Se todo (ou quase todo) o trabalho do servidor é realizar tarefas puramente computacionais (cálculos, resolução de algoritmos, etc.) Node.js pode não ser o modelo ideal.

No entanto, também podemos ver quando a arquitetura se encaixa. Imaginemos que existem dois garçons e quatro clientes em um restaurante. Se os garçons atendem apenas um cliente de cada vez, os dois primeiros clientes obterão seu pedido o mais rápido possível, mas o terceiro e quarto clientes vão ter que esperar até que os dois primeiros tenham seu pedido entregue. Os dois primeiros clientes terão a sua comida assim que estiverem prontas, pois os garçons têm dedicado toda a sua atenção para o cumprimento suas ordens. Isto se realiza a custo dos outros dois clientes. Em um modelo orientado a eventos, os dois primeiros clientes podem ter que esperar um curto espaço de tempo para que os garçons terminem de tomar as ordens dos demais clientes (terceiro e quarto) antes de chegar a sua comida, mas o tempo médio de espera (latência) do sistema será muito mais baixo.

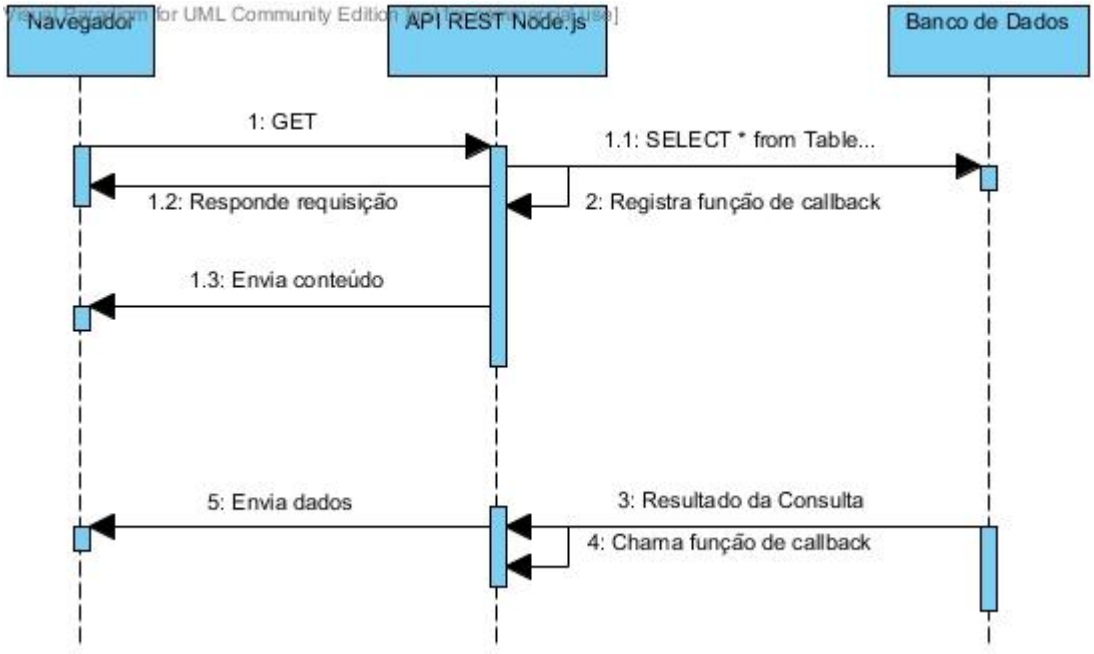

FIGURA 2 – Diagrama de sequência do funcionamento de um aplicativo Node.js

2.1.4 Casos em que devemos considerar sua utilização

2.1.4.1 APIs JSON

Construção de APIs REST que utilizam JSON (Javascript Object Notation) é algo onde Node.js realmente desempenha muito bem. Seu modelo não-bloqueável combinado com o Javascript lhe torna uma ótima opção para envolver outras fontes de dados, tais como diversos bancos de dados ou serviços da web ou WebService e expô-los através de uma interface JSON.

2.1.4.2 Aplicativos de uma única página

Se estivermos planejando escrever um página com um aplicativo pesado que utilize muitas requisições AJAX, como o Gmail, por exemplo, Node.js é excelente para tal tarefa. A capacidade de processar muitos pedidos por segundos com tempos de resposta baixos, assim como tarefas de compartilhamento como o código de validação entre o cliente e o servidor, tornam o Node.js uma excelente escolha para aplicações web modernas que fazem lotes de processamento no cliente.

2.1.4.3 Ferramentas Unix

Pelo fato de o Node.js ainda ser jovem, é tentador pensar em reinventar todos os tipos de software para ele. No entanto, uma abordagem melhor é mergulhar no vasto universo de ferramentas de linha de comando existentes. A capacidade do Node.js de gerar milhares de processos filhos e tratar os seus resultados como um fluxo, faz dele a escolha ideal para aqueles que procuram alavancar o desempenho de softwares existente.

2.1.4.4 Fluxo de dados

Uma pilha de protocolo de web services é uma pilha de protocolos que é usada para definir, localizar, implementar e fazer com que serviços web interajam entre si. Essas pilhas web tradicionais muitas vezes tratam solicitações e respostas HTTP como eventos atômicos. No entanto, a verdade é que eles são os canais, e muitos aplicativos legais podem ser construídos em Node.js para tirar proveito deste fato. Um grande exemplo é o upload de arquivos em tempo real, bem como a construção de proxys entre as camadas de dados diferentes.

2.1.4.5 Aplicações em tempo real

Outro grande aspecto do Node.js é a facilidade com que podemos desenvolver sistemas de tempo real. Sistemas com alto grau de interação com o usuário como o Twitter, programas de chat, apostas esportivas ou interfaces para redes de mensagens instantâneas.

Mas devemos ter cuidado aqui, pois como o Javascript é uma linguagem dinâmica que possui um coletor de lixo, os tempos de resposta poderão sofrer alterações dependendo da frequência e duração em que o coletor de lixo entra em ação. Então não é recomendável tentar construir sistemas muito rígidos em tempo real que exigem tempos de resposta extremamente consistentes, utilizando o Node.js.

2.1.4.6 Encontrando desenvolvedores

Podemos pensar do Javascript o que quisermos, mas neste momento ele está se tornando a língua franca da programação de computadores. Praticamente todos os computadores pessoais têm um ou mais interpretadores de Javascript (navegadores) instalados, o que significa que é improvável para a maioria dos desenvolvedores web não terem aprendido tal linguagem ou nunca sequer ter tomado conhecimento sobre a mesma.

Isso significa que possuímos uma enorme e diversificada gama de pessoas que podemos contratar, e provavelmente já possuímos muitos talentos em nossa própria empresa ou ao nosso redor. Então, se estivermos trabalhando para uma empresa em crescimento, este é um forte argumento favorecendo Node.js.

2.1.4.7 Comunidade vibrante

Neste momento, a comunidade Node.js está crescendo em um ritmo acelerado, atraindo alguns dos mais inteligentes desenvolvedores da indústria. Isto também significa que o ecossistema Node.js está crescendo a cada dia, e também é fácil de obter suporte gratuito e comercial de várias fontes.

2.1.4.8 Desempenho

Este argumento tem de ser cuidadosamente tocado, mas se o desempenho é um aspecto crítico da sua aplicação, Node.js tem muito para oferecer. Com cinco empresas (Mozilla, Google, Apple, Microsoft, Opera) competindo sobre a melhor implementação Javascript, o interpretador incluso no Node.js (V8 do Google) tornou-se incrivelmente rápido, e ficando cada dia melhor. Combinando isso com o modelo não-bloqueável do Node.js, é necessário tempo para tentar criar uma aplicação que seja lenta. A maioria dos aplicativos feitos em Node.js são facilmente capazes de lidar com milhares de conexões simultâneas, e isso tudo utilizando o que poderia ser considerado hardware moderado por quaisquer padrões.

2.1.4.9 Apoio Corporativo

Um dos riscos com o uso de um projeto de código aberto jovem, é a falta de compromisso de longo prazo de seus autores. Este fato não é o que parece acontecer com o Node.js. Node.js é atualmente patrocinado pela Joyent, que contratou Ryan Dahl e vários outros colaboradores do núcleo do projeto, então há uma força econômica real apoiando o desenvolvimento futuro do projeto.

Entre outras coisas, isso já deu a empresas como Yahoo! e HP confiança suficiente para construir seus produtos de próxima geração utilizando o Node.js.

De maneira geral:

- Javascript é uma linguagem conhecida por muitos desenvolvedores;

- Node.js já está configurado com uma arquitetura orientada a ventos com fácil uso, devido às suas funções anônimas, fechamentos e call-backs.

- Seu rápido interpretador de Javascript possui três pilares para o desempenho: o acesso rápido a propriedade, geração de código dinâmico da máquina e coleta de lixo eficiente, que permite a execução de Javascript próximo de velocidades nativas.

- Comutação de threads tem um custo que faz com que os servidores orientados a threads fiquem mais lentos conforme o número de requisições simultâneas cresce.

- Com o número crescente de aplicações ricas para a internet utilizando Ajax e websockets frequentemente, pequenas requisições ao servidor têm se tornado mais comuns, e isto aumenta significantemente o número de requisições concorrentes.

- As operações de entrada e saída não bloqueáveis é a principal vantagem, pois a capacidade de resposta do servidor web é muito menos dependente do número de requisições simultâneas ao banco de dados (especialmente se estivermos utilizando algum tipo de NoSQL, como redis, memcached ou MongoDB).

2.1.5 Casos em que devemos descartar sua utilização

Mesmo parecendo que Node.js é útil para todas as situações, há algumas específicas onde sua utilização simplesmente não faz muito sentido. Vejamos:

2.1.5.1 Aplicativos pesados para a CPU

O caso mais óbvio é com aplicativos que são muito pesados quando falamos sobre utilização da CPU e muito leve na ordem de operações de E/S. Então se nosso objetivo é escrever algum tipo de software de codificação de vídeo, software de inteligência artificial ou algum software similar que exija muito da CPU, o uso do Node.js não é vantajoso. Provavelmente obteremos melhores resultados utilizando C ou C++.

2.1.5.2 Aplicativos HTML que sejam um simples CRUD

Apesar de Node.js ser uma útil e agradável ferramenta para escrever todos os tipos de aplicações web não devemos esperar que ele nos proporcionasse mais benefícios do que PHP, Ruby ou Python neste momento. Sim, o aplicativo pode acabar sendo um pouco mais escalável, mas não, a nossa aplicação não vai magicamente obter mais tráfego só porque vamos escrevê-la em Node.js. A verdade é que enquanto nós estamos começando a ver os quadros bons para Node.js, não há nada tão poderoso quanto o Rails, CakePHP ou Django em cena ainda. Se a maior parte do nosso aplicativo é simplesmente o processamento de HTML baseado em algum banco de dados, fazendo uso do Node.js não vai proporcionar muitos benefícios comerciais tangíveis ainda.

2.1.5.3 NoSQL + Node.js

Sim, Redis, CouchDB, MongoDB, Riak, Casandra, etc., tudo parece ser realmente tentador, mas se já estamos assumindo um risco tecnológico com o uso

do Node.js, não devemos multiplicá-lo com mais tecnologia que provavelmente não entendemos completamente ainda.

Claro, existem casos de uso legítimos para a escolha de um banco de dados orientado a documento, como o Facebook, Twitter, Google, Microsoft e outras inúmeras empresas que fazem uso de diversos bancos de dados deste tipo.

2.1.6 Instalação do Node.js

Instalar o Node.js é extremamente simples. Node.js é executado em Windows, Linux, Mac, e outros sistemas operacionais POSIX (como o Solaris e BSD). Node.js está disponível a partir de dois locais principais: o site do projeto (http://www.nodejs.org/) ou o repositório GitHub.

O recomendável é realizar o download a partir do site, pois lá se encontra disponível sempre a última versão estável. Os recursos mais recentes estão hospedados no GitHub, para a equipe de desenvolvimento ou qualquer outra pessoa poder copiar. Embora no repositório GitHub esteja disponível a última versão com as últimas atualizações, esta nem sempre é estável, e não consideramos confiável utilizar uma versão não estável em nosso projeto.

Começaremos então instalando o Node.js. A primeira coisa a fazer é realizar o download a partir do site http://www.nodejs.org. Há instaladores para Windows e Mac OS, além do último código fonte estável. Caso estejamos utilizando Linux, podemos instalar a partir do código fonte ou ainda utilizar um gerenciador de pacotes (apt-get, yum, etc.).

Instalando utilizando o código fonte:

2.1.6.1 Instalando no Mac OS**:**

Caso estejamos fazendo uso do homebrew package manager, podemos instalar o Node.js com apenas um comando: brew install node

Caso contrário, basta seguir os seguintes passos:

i. Instalar Xcode

ii. Instalar git
iii. executar os seguintes comandos:

git clone https://github.com/joyent/node
cd node
./configure
make
sudo make install

Feito isso podemos depois testar se tudo está correto, realizando um exemplo Hello World, que veremos a seguir.

2.1.6.2 Instalando no Ubuntu**:**

Basta seguir os seguintes passos:

1. Instalar as dependências necessárias:
   - *sudo apt-get install g++ curl libssl-dev apache2-utils*
   - *sudo apt-get install git-core*
2. Executar os seguintes comandos:

git clone https://github.com/joyent/node
cd node
./configure
make
sudo make install

Feito isso podemos depois testar se tudo está correto, realizando um exemplo Hello World, que veremos a seguir.

2.1.6.3 Instalando no Windows

Para instalar a partir do código fonte no Windows, é necessário utilizarmos programas de terceiros para isto, neste caso o cygwin (http://www.cygwin.com/):

1. Instalar o cygwin (há um tutorial disponível em http://www.mcclean-cooper.com/valentino/cygwin_install/)
2. Utilizar o setup.exe na pasta do cygwin para instalar os seguintes pacotes:
    - devel → openssl
    - devel → g++-gcc
    - devel → make
    - python → python
    - devel → git
3. Abrir o terminal de linha de comando do cygwin: “Iniciar -> Cygwin > Cygwin Bash SHell.
4. Executar os seguintes comandos para construir o Node.js:

git clone https://github.com/joyent/node
cd node
./configure
make
sudo make install

Feito isso podemos depois testar se tudo está correto, realizando um exemplo Hello World, que veremos a seguir.

2.1.7 Exemplo de utilização

Este é um exemplo básico, um simples servidor web que vai apenas responder “Hello World” para cada requisição. Uma maneira simples e rápida de certificarmo-nos de que o Node.js foi instalado e está funcionando corretamente. Basta criarmos um arquivo, utilizando nosso editor de texto preferido, inserir o seguinte conteúdo e salvá-lo com o nome de exemplo.js:

```
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('Servidor rodando em http://127.0.0.1:1337/');
```

TRECHO DE CÓDIGO 1 - Hello World Node.js

```
node example.js
```

TRECHO DE CÓDIGO 2 - Exemplo Node.js

Inserido o conteúdo acima e com o arquivo salvo, basta executar o comando a seguir como linha de comando:

E será obtido o seguinte resultado:

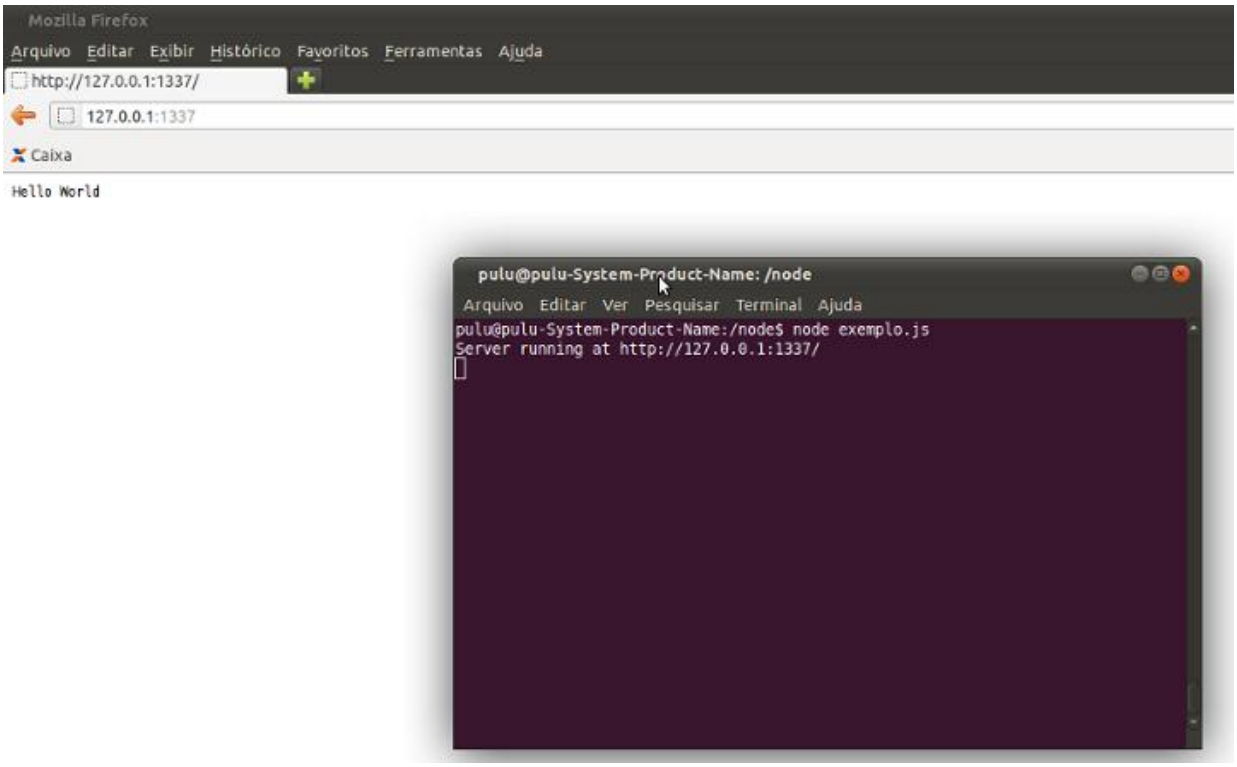

Figura 3 – Exemplo Hello World em Node.js

2.1.8 Quem utiliza

Alguns relatos de grandes empresas que utilizam o Node.js:
Linkedin:

- “Do lado do servidor, todo nosso software voltado para atender aos dispositivos móveis é totalmente construído utilizando Node.js. O primeiro motivo para utilizá-lo, foi a escalabilidade e o segundo foi que o Node.js nos mostrou imensos ganhos de desempenho.” - Kiran Prasad, Diretor de Engenharia de Móvel.

Ebay:

- “O modelo de E/S orientado a eventos nos livrou da preocupação sobre problemas com bloqueios e concorrência que eram comuns nos modelos com vários threads e operações de E/S assíncronas.” - Subbu Allamarju, Diretor Técnico.

Uber

- “Node.js nos permitiu construir um sistema em tempo real de logística sem precisarmos nos preocupar com problemas de bloqueios e concorrência.” - Curtis Chambers, Diretor de Engenharia.

Transloadit

- “Node.js nos permitiu executar muitos processos independentes em plano de fundo, de uma maneira que não ocorressem bloqueios. Isto é essencial para fazer com que façamos do upload de arquivos e codificação de vídeos uma experiência incrível para o usuário.” - Tim Koschuetzki, Fundador.

Microsoft

- Node.js deu aos usuários do Azure, a primeira experiência de desenvolvimento utilizando JavaScript de ponta-a-ponta, para o desenvolvimento de uma nova classe de aplicativos em tempo real. - Cláudio Caldato, Diretor de Programas, Setor de Estratégias de Interoperabilidade.

Voxer

- Node.js faz mágica nos lugares certos. Nós escrevemos nossas aplicações e o próprio Node.js se encarrega de enviar JSON utilizando o protocolo HTTP. - Matt Ranney, Diretor Técnico

Yahoo!

- Node.js é o coração do Manhattan, permitindo com que os desenvolvedores construam o código todo baseado em apenas uma linguagem de programação - isto é como um sonho para os programadores. - Renaud Waldura, Gerente Sênior.

Cloud 9 IDE

- Node.js nos permitiu construir uma IDE em tempo real na nuvem, utilizando apenas uma única linguagem de programação, deste o front como o backend. Ele tornou tanto as nossas vidas, como as do usuários mais fáceis para escrever, executar e debugar códigos em qualquer lugar, a qualquer momento. - Rik Arends, Diretor Técnico.

2.2 REST

REST nos propõe uma interface muito mais simples do que o SOAP, porém com algumas limitações. Desenvolver utilizando REST é mais fácil e mais rápido e esta é a razão pela qual APIs web mais famosas são baseados em REST.

As limitações estão relacionadas com o protocolo: com REST somos obrigados a utilizar o protocolo HTTP. Por isso, é difícil de manusear diferentes transportes como MQ ou JMS. Então, se estivermos criando um aplicativo para ser usado exclusivamente na web, REST pode ser uma opção. Se estivermos projetando uma arquitetura orientada a serviços para uma empresa, que precisa interligar vários sistemas que utilizam transporte de múltiplos canais e com operações sofisticadas para ser modelados com BPMs, então é melhor ir com SOAP.

Estamos pagando pela complexidade do desenvolvimento em troca da flexibilidade do SOAP. Interfaces SOAP são descritas pelo padrão WSDL que parece realmente complicado e não é suposto para ser lido por seres humanos.

As aplicações construídas nos moldes de REST são denominadas RESTful. Aplicações RESTful simplesmente contam com o built-in HTTP de segurança (protocolo HTTPS), com SOAP podemos ter opções mais sofisticadas, como o WS-Security, que nos permite criptografar apenas partes das mensagens e move o manuseio de segurança da camada de transporte para o camada de mensagem.
REST é mais jovem do SOAP, por isso não é bem suportado por produtos comerciais (tanto BPM e produtos ESB). No entanto as novas versões do REST recebem grande apoio de alguns ESBs, como o Mule e o OpenESB.

Serviços RESTful não tem um descritor estrito, foi tentado definir algo do mesmo tipo do WSDL, ele é chamado WADL (https://wadl.dev.java.net/). O WADL de momento não é um padrão comum, e sua adoção é muito baixa.

Se estivermos planejando executar uma implantação de um serviço muito grande com muitas interfaces de dados, teremos grandes benefícios usando os Serviços RESTful. REST é baseado 100% na rfc HTTP (http://www.w3.org/Protocols/rfc2616/rfc2616.html). Muitas das interfaces RESTful que vemos atualmente na Internet não suportam a especificação HTTP, e o resultado é que elas não poderiam se beneficiar desta especificação em termos de desempenho, cache, negociação de conteúdo e segurança.

Um grande motivo para a utilização de interfaces REST, é que elas pode facilmente suportar content-types/mime-types diferentes. Outra razão pela qual as pessoas estão usando interfaces REST, é para sua aplicação ser escalável como a web e o Google. Isso é possível quando estivermos utilizando o puro e simples protocolo HTTP. Temos o apoio de tantos produtos padrão como web proxy caches (como SQUID) ou outros para escalar nossa aplicação com um esforço muito baixo. Para as comunidades sociais, temos agora um novo padrão para serviços REST de segurança. Ele é chamado OAuth (http://oauth.net/), que faz da autenticação via HTTP uma tarefa muito fácil.

Alguns outros exemplos onde os serviços REST são dominantes são a Amazon AWS, onde são oferecidos serviços para armazenamento (S3) que possuem interfaces SOAP e REST, mas mais de 90% dos clientes estão usando

REST e o Google Maps, que está usando REST URIs para fornecer partes de mapas e o Yahoo API é em sua maior parte construída com interfaces REST.

Serviços REST são predominantes nas APIs WEB 2.0 para Mashups, como o OpenSocial API. Interfaces em REST são muito boas para aplicativos que lidam diretamente com o banco de dados, e principalmente quando algum cliente necessita de uma rápida e ágil integração.

2.2.1 Definição da arquitetura REST

REST (Representational State Transfer, Transferência de Estado Representacional) é um estilo arquitetural de software voltado para sistemas de hipermídia distribuídos, tais como a World Wide Web. Este estilo arquitetural, que foi proposto por Roy Fielding (2000) em sua dissertação de doutorado, tem sido largamente usado para guiar o design e o desenvolvimento de diversas aplicações da Web moderna. O estilo arquitetural REST, enfatiza a escalabilidade das interações entre componentes, a generalidade das interfaces, o desenvolvimento independente de componentes e a utilização de componentes intermediários para reduzir a latência das interações, reforçar a segurança e encapsular sistemas legados (Fielding 2000).

Um dos principais usos de REST é a criação de Web services (conhecidos como RESTful Web Services ou RESTful Web APIs) simples, construídos através do uso de padrões populares e bem estabelecidos como HTML, URI e XML. Esta prática surgiu como uma alternativa às opções de arquitetura de Web services baseadas em padrões complexos, que ignoram ou reinventam elementos e características que já tornaram a World Wide Web uma tecnologia de sucesso. Deste modo, os RESTful Web Services são propostos como um modelo simplificado que busca padronizar a forma como a Web é utilizada, tanto por humanos como por máquinas, pois é estabelecido que as características que tornam um website fácil de usar também são as características que tornam a API de um Web service fácil de ser utilizada pelo programador (Richardson e Ruby 2007).

Para entender melhor o quê isso significa, vamos considerar uma aplicação web simples:

1. Um usuário visita a página inicial da aplicação, digitando o endereço no navegador.
2. O navegador envia uma solicitação HTTP para o servidor.
3. O servidor responde com um documento HTML contendo alguns links e formulários.
4. O usuário digita seu status em um formulário e envia o formulário.
5. O navegador envia uma requisição HTTP para o servidor.
6. O servidor processa a solicitação e responde com outra página.

Esse ciclo continua até que o usuário pára de navegar. Exceto por algumas poucas exceções, a maioria sites e aplicações baseadas na Web seguem o mesmo padrão.

Transferência de Estado Representacional (REST) é uma descrição técnica de como a Web funciona. Especificamente, REST nos diz como a Web atinge sua grande escala. se fosse capaz de dizer que a Web utiliza um sistema operacional, seu estilo arquitetônico seria REST. Uma Interface de Programação de Aplicativos (Application Programming Interface - API) REST é um tipo de servidor web que permite que um cliente, seja operado por um usuário ou automatizado, possa acessar os recursos de um modelo dados do sistema e suas funções.

O que o usuário digitou no navegador no início do exemplo anterior, é um Identificador Uniforme de Recursos (Uniform Resource Identifier - URI). Outro nome comum utilizado é Localizador Uniforme de Recursos (Uniform Resource Locator - URL). URI é um termo mais geral, que podemos utilizar tanto para referir uma localização (URL) como um nome. Uma URI é um identificador de recursos, onde na maioria dos casos elas não são visíveis aos clientes.

As restrições que compõem a arquitetura de sistemas REST foram desenvolvidas de forma incremental, através da análise das restrições de outros estilos arquiteturais baseados em rede, as quais cumprem o papel de enfatizar os principais objetivos e prerrogativas do estilo arquitetural. Estas restrições, bem como seus objetivos, são propostas e descritas por Fielding em seu trabalho (Fielding 2000):

Cliente-Servidor - Ao separar os aspectos de interface do usuário dos aspectos de armazenamento de dados, aumentamos a portabilidade da interface através de várias plataformas e aprimoramos a escalabilidade ao simplificar os

componentes do servidor. Esta separação de interesses permite que os componentes evoluam independentemente, uma característica que é especialmente significativa para a Web.

Comunicação sem estado - A comunicação entre cliente e servidor deve ser desprovida de estado. Isto é, cada requisição enviada ao servidor pelo cliente deve conter toda a informação necessária para se fazer entendida, e não pode tirar vantagem de algum contexto armazenado no servidor. Deste modo, o estado da sessão é mantido inteiramente no lado do cliente. Esta restrição é responsável pelas propriedades de visibilidade (um sistema monitorador não precisa olhar para além dos dados de uma requisição para determinar a natureza daquela requisição), confiabilidade (torna mais fácil a recuperação de falhas parciais) e escalabilidade (por não precisar armazenar e tratar informação de estado entre as requisições, o componente servidor pode mais rapidamente liberar seus recursos, além de ter sua própria implementação simplificada). Como a maioria das decisões arquiteturais, a opção pela comunicação stateless tem suas desvantagens, sendo a principal delas o decréscimo no desempenho da rede pelo aumento da quantidade de dados repetidos enviados em uma série de requisições, já que estes dados não podem ser mantidos no servidor em um contexto compartilhado.

Cache - A fim de melhorar o desempenho da rede, restrições de cache são adicionadas ao cliente. Estas restrições requerem que os dados contidos em uma resposta do servidor a uma requisição sejam marcados como cacheable ou non-cacheable. Se uma resposta é cacheable, então o cliente pode reusar os dados desta resposta em futuras requisições equivalentes. A grande vantagem da adição de restrições de cache ao cliente é possibilitar potencialmente a eliminação, parcial ou completa, de algumas interações, melhorando eficiência, escalabilidade e o desempenho percebida pelo usuário ao reduzir a latência média de uma série de interações. Uma desvantagem natural é a possível diminuição da confiabilidade, visto que um conjunto de dados armazenados na cache do cliente pode, em um determinado momento do tempo, ser significativamente diferente dos dados que o cliente obteria diretamente do servidor por meio de uma requisição direta.

Interface uniforme - A ênfase em uma interface uniforme entre componentes é a principal característica que difere REST dos demais estilos arquiteturais baseados em rede. Ao aplicar o princípio de Engenharia de Software da

generalidade à interface dos componentes, a arquitetura do sistema em geral é simplificada e a visibilidade das interações é melhorada. Uma desvantagem, contudo, é que a uniformidade das interfaces diminui a eficiência, já que a informação é transferida em um formato padronizado ao invés de um formato específico que atende às necessidades de uma aplicação.

Sistema em camadas - O estilo de sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas ao restringir o comportamento dos componentes de forma que cada componente não possa "enxergar"além do domínio da camada com a qual interage. Esta restrição limita a complexidade do sistema como um todo e promove a independência entre as camadas. As camadas também podem ser usadas para encapsular serviços legados e proteger novos serviços de clientes legados, simplificando componentes ao mover funcionalidade raramente usada para um intermediário compartilhado. Camadas intermediárias também podem ser usadas para melhorar a escalabilidade do sistema ao permitir o balanceamento de carga de serviços através de várias redes e processadores. A principal desvantagem da arquitetura em camadas é a adição de latência ao processamento dos dados, reduzindo o desempenho percebida pelo usuário, desvantagem essa que pode ser sanada pelo uso de caches compartilhadas entre os intermediários.

Código sob demanda - REST permite que a funcionalidade do cliente seja estendida, baixando e executando código na forma de applets ou scripts. Esta possibilidade simplifica a implementação dos clientes, que podem ter um menor número de funcionalidades pré-implementadas, e aumenta a capacidade de extensão do sistema, embora diminua sua visibilidade. Portanto, esta é uma restrição opcional na arquitetura REST.

REST é um estilo arquitetural híbrido, derivado da combinação de diversos outros estilos arquiteturais baseados em rede e restrições adicionais que definem uma interface uniforme de conexão. Como resultado, REST foi concebido como um estilo orientado a recursos para a abstração de elementos arquiteturais dentro de um sistema de hipermídia distribuído. REST ignora detalhes da implementação dos componentes e sintaxe de protocolo e concentra-se nos papéis desempenhados pelos componentes, nas restrições de sua interação com outros componentes e na sua interpretação de elementos de dados significativos. Deste modo, REST engloba

as principais restrições em torno dos componentes, conectores e dados que definem a base da arquitetura da Web e, assim, definem também a essência de seu comportamento como uma aplicação baseada em rede (Fielding 2000).

Os elementos arquiteturais de REST são os seguintes (Fielding 2000):

Elementos de dados - A principal abstração de dados em REST é o recurso. Um recurso é definido como um mapeamento conceitual para um conjunto de entidades. Usualmente, um recurso é algo que pode ser armazenado em um computador e representado como uma sequência de bits: um documento, uma linha em um banco de dados, ou o resultado da execução de um algoritmo. Contudo, um recurso também pode ser um objeto físico, como uma pessoa, ou mesmo um conceito abstrato. Ou seja, em REST um recurso é qualquer coisa suficientemente importante para ser referenciada por si própria (Richardson e Ruby 2007). Para nomear e referenciar um recurso, REST faz uso de identificadores de recurso. Para que as restrições da arquitetura se mantenham, é necessário que todo e qualquer recurso possua ao menos um identificador válido. Em implementações baseadas na Web, os identificadores são implementados pelas URIs e URLs que endereçam os recursos do sistema. Estes recursos podem ser apresentados em diferentes representações, que podem ser entendidas como formatos alternativos de encapsulamento dos dados. Por exemplo, um mesmo recurso (e.g., uma lista de nomes) pode ser recuperado como um documento XML, ou como uma página da web, ou mesmo como um arquivo de texto separado por vírgulas (Richardson e Ruby 2007).

Conectores - Conectores, em REST, são elementos que encapsulam as atividades de acesso a recursos e transferência de representações de recurso. Os conectores apresentam uma interface abstrata para a comunicação de componentes, reforçando a simplicidade ao prover uma separação de interesses e esconder a implementação interna de recursos e mecanismos de comunicação. Os principais tipos de conectores são cliente e servidor; a diferença essencial entre estes é que, enquanto o cliente inicia a comunicação ao efetuar uma requisição, o servidor espera ativamente por conexões e responde a requisições para fornecer acesso aos seus serviços. Outro importante tipo de conector é a cache, que pode ser localizada na interface do cliente ou do servidor para salvar respostas marcadas

como cacheable a fim de reusá-las posteriormente a fim de reduzir a latência das interações. Demais tipos de conectores incluem o resolvedor (de nomes de recursos) e o túnel, usado para transmitir a comunicação através de uma fronteira (por exemplo, um firewall).

Componentes - Os componentes de REST são tipificados de acordo com seus papéis desempenhados na aplicação. Um exemplo comum de componente é o agente de usuário, que utiliza um conector cliente para iniciar uma requisição e torna-se também o receptor final da resposta esperada, papel que é comumente realizado pelo browser ou navegador Web. Outros tipos de componentes incluem servidores de origem, gateways e proxies.

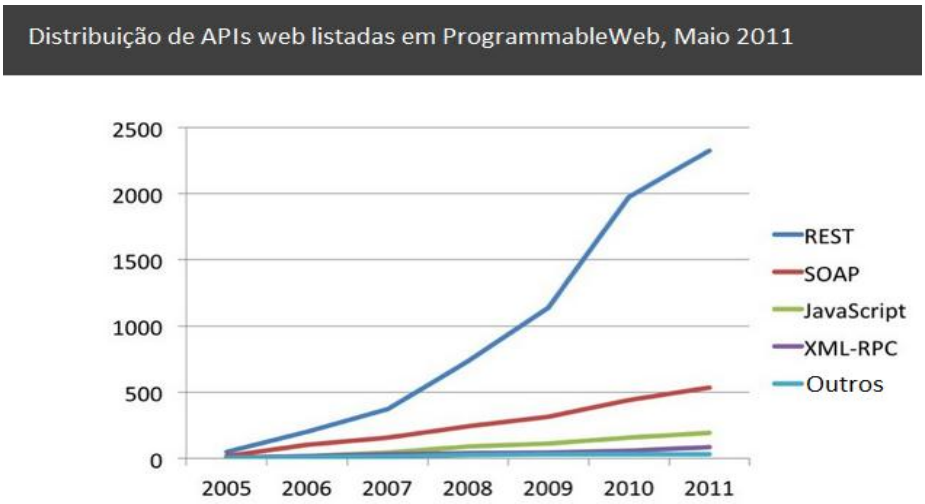

Figura 4 – Crescimento das APIs REST

2.3 Vantagens e desvantagens das diferentes abordagens de Web Services

Proporcionar interoperabilidade entre tecnologias heterogêneas e promover o acoplamento fraco entre o consumidor do serviço (cliente) e o prestador de serviços são os principais objetivos da arquitetura SOA com base nos conceitos e tecnologia de web services. Porém, muitos estilos diferentes de web services podem ser usados para integrar aplicações empresariais (Pautasso et al., 2008). A escolha é uma decisão importante de arquitetura, que influencia os requisitos e as propriedades do sistema integrado. A tecnologia de web services SOAP oferece a interoperabilidade, assim como o Remote Procedure Call (RPC) e estilos de integração de mensagens.

2.3.1 Web Services SOAP

Apesar da sua complexidade, o formato de mensagem SOAP e a linguagem de definição de interface (WSDL) ganharam ampla adoção de tecnologias como o gateway capaz de oferecer interoperabilidade entre os sistemas de middleware heterogêneos. Uma das vantagens do WS-* é a transparência e independência de protocolo. Usando SOAP, a mesma mensagem no mesmo formato pode ser transportada através de uma variedade de sistemas middleware que dependa do HTTP (Pautasso et al., 2008).

A utilização de WSDL para descrever uma interface de serviço ajuda a abstrair o protocolo de comunicação e os dados de serialização, bem como a plataforma de implementação do serviço (sistema operacional e linguagem de programação). Contratos WSDL fornecem uma descrição de processamento da sintaxe e da estrutura do pedido correspondente e mensagens de resposta para as máquinas, além de definir um caminho evolutivo e flexível para o serviço. Assim como os negócios e os requisitos tecnológicos mudam, a interface abstrata do mesmo serviço pode ser vinculada a diferentes protocolos de transporte e endpoints de mensagens. Em particular, WSDL pode modelar interfaces de serviços para sistemas baseados em padrões de interação síncronos e assíncronos. Este tipo de flexibilidade torna-se fundamental quando gateways de construção de sistemas legados já existentes, ou que nem sempre usam protocolos web amigáveis (Pautasso et al., 2008).

Além disso, os motores SOAP e as ferramentas WSDL efetivamente encapsulam a complexidade do programador da aplicação e do integrador. De acordo com a sua experiência pessoal, Pautasso (Pautasso et al., 2008) enfatiza que não é necessário estudar as especificações para ser capaz de desenvolver serviços inter operáveis, assumindo que o tempo de execução e ferramentas selecionadas mantenham o perfil básico WS-I 12. Uma característica importante é que os clientes de teste podem ser gerados a partir dos contratos WSDL automaticamente.

As ferramentas WS-* atuais favorecem a transformação dos componentes de software existentes em serviços da web, porém podem ser facilmente confundidas. Dessa forma, é importante evitar perdas entre os diferentes níveis de abstração. No entanto a tradução entre o XML e as estruturas de dados correspondentes de memória tem sido problemática e é a principal fonte de ineficiência e de desempenho (Pautasso et al., 2008).

Os problemas de interoperabilidade podem ocorrer quando, por exemplo, tipos de dados nativos e construções de linguagem da execução de serviços estão presentes em sua interface. Esse problema pode ser atenuado por afirmar e fazer cumprir determinadas diretrizes de codificação, tais como o desenvolvimento contract-first .

2.3.2 RESTful XML

Os web services RESTful são conhecidos pela sua simplicidade, pois o estilo REST aproveita as normas existentes e conhecida da W3C/IETF (HTTP, XML, URI, MIME) baseada em uma infra-estrutura generalizada. Os clientes e servidores HTTP já estão disponíveis para todas as linguagens de programação e principais plataformas de sistema operacional/hardware, e a porta HTTP padrão 80 é comumente deixada em aberto por padrão na maioria das configurações de firewall (Pautasso et al., 2008).

Essa infra-estrutura leve, em que os serviços podem ser construídos com ferramentas mínimas, é de baixo custo para a aquisição e, portanto, tem pouca resistência para a sua adoção. O esforço necessário para construir um cliente para um serviço RESTful é muito pequeno, os desenvolvedores podem começar a testar esses serviços a partir de um navegador da web comum, sem ter que desenvolver o software de cliente personalizado.

Implantar um serviço web RESTful é muito similar á construção de um site dinâmico (Pautasso et al., 2008).

Além disso, graças aos URIs e hiperlinks, o REST demonstrou que é possível descobrir os recursos da web sem uma abordagem baseada em registro obrigatório para um repositório centralizado. Do ponto de vista operacional, os web

services RESTful oferecem suporte para cache, clustering e balanceamento de carga. O REST possibilita também maior liberdade para aperfeiçoar o desempenho de um web service com um formato de mensagem mais leve, por exemplo, o Javascript Object Notation (JSON) (Pautasso et al., 2008).

Os web services REST costumam ser implementados de duas formas diferentes: Hi-REST e Lo-REST. Hi-REST é a implementação conforme descrito por Fielding (Fielding, 2000), também chamada de RESTful. A implementação Lo-REST, por outro lado, utiliza apenas os métodos HTTP GET e POST. Existem algumas confusões sobre as melhores práticas comumente aceitas para a construção de web services RESTful. As recomendações Hi-REST foram estabelecidas de maneira informal, mas nem sempre são totalmente seguidas (Pautasso et al., 2008).

Assim, apenas dois verbos, (GET para pedidos e POST para tudo o mais), são usados. Isto é devido ao fato de que proxies e firewalls nem sempre permitem conexões HTTP que usam qualquer outro verbo. Como a maioria das soluções não-padrão, eles não podem ser entendidos por todos os servidores web, e exigem desenvolvimento adicional e esforço de teste (Pautasso et al., 2008).

Outra limitação torna impossível seguir estritamente a regra GET / POST para pedidos repetitivos que tenham grandes quantidades de dados de entrada (mais de 4 KB na maioria das implementações atuais). Não é possível codificar esses dados no recurso URI, pois o servidor irá rejeitar como URIs “mal formadas” ou, no pior caso, será acidental, expondo o serviço a ataques de buffer overflow. O tamanho do pedido, não obstante, também pode ser um desafio para codificar estruturas complexas de dados em um URI como não há nenhum mecanismo que aceite o marshalling. Essencialmente, o método POST não sofre tais limitações (Pautasso et al., 2008).

2.3.3 RESTful JSON

Quando se trata de gerenciar a evolução de um web service, o acoplamento fraco implica a capacidade de fazer modificações no serviço sem afetar seus

clientes. No caso de web services RESTful, é evidente que os quatros verbos GET, POST, PUT, DELETE são uniformes e os mesmos para todos os serviços e nunca mudam. O congelamento do protocolo básico permite a dissociação total de clientes e servidores, como qualquer alteração no servidor nunca vai quebrar um cliente, tal mudança simplesmente não acontece.

Para os web services de WS-*, existe um formato de mensagem simples e padronizada: SOAP. Por outro lado, os web services RESTful atualmente não usam um único formato para representar recursos. Em vez disso, contam com a flexibilidade proporcionada pelas características e conteúdo do REST para escolher entre uma variedade de MIME types do documento que pode também incluir SOAP. Tal situação pode complicar e dificultar a interoperabilidade de serviços web RESTful, como por exemplo, os clientes esperando os dados JSON não serão capazes de analisar um arquivo XML. Além disso, um web service RESTful é capaz de prover os recursos em formatos de representação múltipla, o que requer mais esforço de manutenção. No entanto, o formato JSON sobre o XML pode compensar o esforço extra e a falta de interoperabilidade com uma redução significativa da sobrecarga (Pautasso et al., 2008).

2.3.4 XML - RPC

O RPC é a sigla para Remote Procedure Calls. Essa técnica é usada na construção de aplicações distribuídas. Baseia-se no conceito convencional de funções e procedimentos locais de um programa, estendendo-o. Utilizando RPC é possível chamar um procedimento que não esteja implementado localmente, tampouco escrito na mesma linguagem. Semelhante á serialização, é o processo de transformar a representação da memória de um objeto para um formato de dados adequados para o armazenamento ou transmissão.

A abordagem RPC facilita o trabalho do programador, pois poupa o trabalho de ter que aprender sobre protocolos subjacentes, redes e vários detalhes de implementação. As bibliotecas RPC são geralmente projetadas para serem relativamente transparentes e muitas vezes são operadas com uma única função de chamada em vez de uma API complexa. Programas escritos em mainframes,

minicomputadores, workstations e computadores pessoais, até mesmo de diferentes fornecedores, podem se comunicar se estiverem em uma rede comum (Laurent et al., 2001).

Efetivamente, RPC fornece aos desenvolvedores um mecanismo para definição de interfaces que pode ser chamado através de uma rede. Essas interfaces podem ser tão simples como uma única chamada de função ou tão complexas e extensas como uma API grande. O RPC é um mecanismo que permite ao desenvolvedor, usufruir o máximo dela, limitado apenas pelos custos de sobrecarga da rede e preocupações arquitetônicas (Laurent et al., 2001).

2.4 JSON

O JSON (Javascript Object Notation) é um mecanismo de codificação/decodificação de valores para intercâmbio de dados. Possui uma sintaxe de alto nível, fácil de ser entendida, facilitando o trabalho dos programadores e dos computadores. Ele é nativo da linguagem JavaScript. O JSON é também um formato de texto que é completamente independente da linguagem de programação, mas usa convenções que são familiares para os programadores como Java, PHP, C/C++ e muitas outras. Estas propriedades fazem do JSON um formato ideal para transmissão de dados orientado a objetos através da rede.

O formato JSON é um modelo de formato para intercâmbio de dados, não é um documento, é uma linguagem textual, e um subconjunto de JavaScript, leve e muito fácil de analisar. Sendo assim JSON tem sido amplamente adotado no meio acadêmico e por diversas empresas, principalmente devido ao fato de ser altamente produtivo em aplicações distribuídas e no desenvolvimento de serviços.

Além de possuir a capacidade de representar as estruturas de dados gerais como: registros, listas e árvores, a sintaxe JSON é significativamente mais simples, então o processamento é mais eficiente em comparação com o XML. Portanto, esta é a diferença mais significativa entre os dois formatos no intercâmbio de dados.

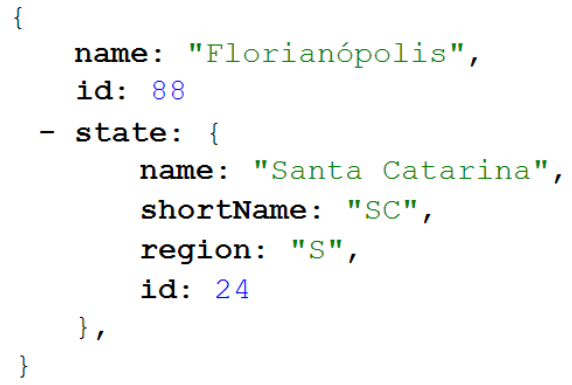

Figura 5 - Exemplo de um objeto no formato json

2.5 Interface Uniforme

REST especifica uma interface uniforme, porém sua generalidade não especifica muito a forma como ela é. Na arquitetura explorada neste trabalho, a interface REST é proveniente do protocolo HTTP.

O protocolo HTTP define nove métodos, dos quais seis são utilizados na arquitetura REST: GET, PUT, POST, DELETE, HEAD e OPTIONS. Cada método está associado a uma função, com exceção do método PUT, que pode ser utilizado de mais de uma maneira.

2.5.1 GET

Realiza uma requisição da representação atual do estado de um recurso. Requisições GET requisitam apenas dados, e não devem ter outro efeito. Sempre que se insere um endereço no navegador web, é realizada uma requisição GET junto ao servidor com o intuito de receber a representação para aquela URL.

2.5.2 PUT

Ao contrário do GET, o método PUT é utilizado para enviar representações ao servidor de forma que ele disponibilize-a através de um recurso.
Quando um método PUT é executado em recurso já existente, a representação deste recurso é substituída pelos dados enviados pelo cliente, junto com a requisição. Deste jeito o PUT atua como forma de atualizar a representação de um recurso. Caso este método seja submetido a um URI que ainda não existe, ele poderá ter a função de criar o recurso e ao mesmo tempo atribuir uma representação a ele.

2.5.3 POST

O método POST é utilizado para a criação de recursos subordinados a outros recursos, isto é, recursos que existirão em relação a um recurso pai.
Embora semelhante em função com o método PUT, o que o diferencia deste último é a incapacidade do cliente de inferir a URI do recurso que está sendo criado, ou seja, será o servidor que irá decidir qual será a URI resultante desta operação e que dará acesso ao recurso desejado. Este é o caso, por exemplo, quando é inserida uma nova linha em um banco de dados onde o identificador desta linha é um número gerado automaticamente pelo servidor, isto não irá permitir ao cliente inferir qual é o identificador das informações recém incluídas por ele.

Como resposta, o servidor irá enviar ao cliente um código de informação que irá dizer a respeito do sucesso ou falha da operação. Juntamente, é possível enviar um outro dado, como uma mensagem de erro no caso de falha ou um dado de localização que informará onde está situado o novo recurso recém criado.

2.5.4 DELETE

Quando é necessária a destruição de um recurso existente, o método DELETE deverá ser utilizado. Opcionalmente poderá ser retornado ao cliente um

código de confirmação da remoção ou erro, embora o não-retorno, em caso de sucesso, seja perfeitamente aceitável.

2.5.5 HEAD

Este método funciona de maneira idêntica ao método GET com a diferença de retornar ao cliente apenas os metadados do recurso, descartando totalmente todo o resto da representação. Isto é útil quando se deseja saber que tipo de recurso é aquele, ou ainda, saber se o recurso existe sem a necessidade de obter o conteúdo completo de sua representação.

2.5.6 OPTIONS

Por fim, o método OPTIONS permite ao cliente descobrir quais métodos um determinado recurso está apto a responder. Caso a resposta de um recurso para este método fosse GET e HEAD, seria possível acessá-lo apenas para leitura, sem a possibilidade de executar um PUT, POST ou DELETE. O conjunto dos métodos permitidos pode ser diferente caso o cliente seja autenticado pelo servidor, possibilitando por exemplo, que visitantes apenas tenham acesso a leitura de informações, enquanto que administradores, após autenticados, tenham o poder de atualizar, incluir ou remover recursos.

3 Estudo de caso – Implementação de uma API Restful em Node.js

Neste capítulo vamos abordar a implementação de uma api REST utilizando Node.js, e comprovar de fato se sua arquitetura não bloqueável é capaz de melhorar o desempenho de uma aplicação com alto índice de concorrência.

3.1 Descrição da api de testes

A implementação será simples e objetiva, restringindo a implementação dos métodos HTTP a apenas o GET, pois este método por si só é capaz de demonstrar a performance e escalabilidade do Node.js pelo motivo deste ser não-bloqueável.

Se fará uso do Node.js em sua última versão estável até o momento (versão 0.8.14) e seu servidor embutido.

O desenvolvimento será realizado utilizando o sistema operacional Ubuntu, na versão 11.04 32bits e também Windows, na versão 7 SP1. A ferramenta Aptana Studio será utilizada para escrever os códigos em Javascript, e a IDE Eclipse será utilizada para escrever e executar as implementações em JAVA.

Com o intuito de compararmos o desempenho da aplicação, será realizada uma implementação nos mesmos padrões utilizando a linguagem JAVA, em sua última versão, v1.7b09, e a especificação REST mais recente para esta linguagem, a JAX-RS 1.1. Será utilizado também a implementação da especificação JAX-RS, RestEasy, em sua última versão estável, 2.3.5.Final. Como servidor, será utilizado o JBoss Application Server na versão 7.1.1.Final.

O banco de dados, que será comum a ambas aplicações, será o MySQL na versão 5.5.28.

A máquina utilizada para realizar as implementações e os testes, é um Intel Core i7 3.5ghz com 6gb DDR3 1600mhz

3.2 Testes

Para realizar os testes, serão utilizadas duas ferramentas de código aberto. São elas, o ApacheBench (http://httpd.apache.org/docs/2.2/programs/ab.html) e o JMeter (http://jmeter.apache.org/), ambas distribuídas pela Apache Software Foundation.

O ApacheBench é um programa single-thread que é executado através de linha de comando. Seu objetivo é medir o desempenho de servidores web enquanto à requisições, sejam estas concorrentes ou sequenciais. Seu uso é bastante simples, bastando apenas um comando, onde é definido o número de requisições, o número de threads simultâneas que farão as requisições ao servidor, e o endereço web do servidor:

```
ab –n 100 –c 5 http://ufsc.br/
```

TRECHO DE CÓDIGO 3 - ApacheBench

E sua saída será sempre semelhante a saída seguir:

```
server@server:~$ ab -n 100 -c 5 http://ufsc.br/
This is ApacheBench, Version 2.3 <$Revision$>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ufsc.br (be patient).....done


Server Software:        nginx/1.1.13
Server Hostname:        ufsc.br
Server Port:            80

Document Path:          /
Document Length:        7116 bytes

Concurrency Level:      5
Time taken for tests:   1.822 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      753500 bytes
HTML transferred:       711600 bytes
Requests per second:    54.88 [#/sec] (mean)
Time per request:       91.101 [ms] (mean)
Time per request:       18.220 [ms] (mean, across all concurrent requests)
Transfer rate:          403.86 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        9   32   8.2     31      55
Processing:    34   58   9.2     58      80
Waiting:       14   31   7.9     31      49
Total:         45   91  11.0     92     113

Percentage of the requests served within a certain time (ms)
  50%     92
  66%     95
  75%     98
  80%     99
  90%    102
  95%    104
  98%    111
  99%    113
 100%    113 (longest request)
server@server:~$
```

FIGURA 6 - ApacheBench

Em sua saída, podemos observar qual o servidor e sua versão que foi testada, seu hostname e a porta na qual foi efetuada a conexão. Mas o que realmente nos interessa são as métricas encontradas pelo apachebench, como a quantidade de requisições por segundo que o servidor respondeu (Requests per second), e o tempo médio para responder uma requisição (Time per request).

Já o Jmeter possui uma interface gráfica, onde é possível construirmos um plano de testes muito mais elaborado. O Apache JMeter pode ser usado para testar o desempenho tanto em recursos estáticos como dinâmicos (arquivos, Servlets, scripts Perl, objetos Java, bases de dados e consultas, servidores FTP e muito mais). Ele pode ser usado para simular uma carga pesada em um servidor de rede, ou objeto para testar a sua força ou para analisar o desempenho global no âmbito de diferentes tipos de carga.

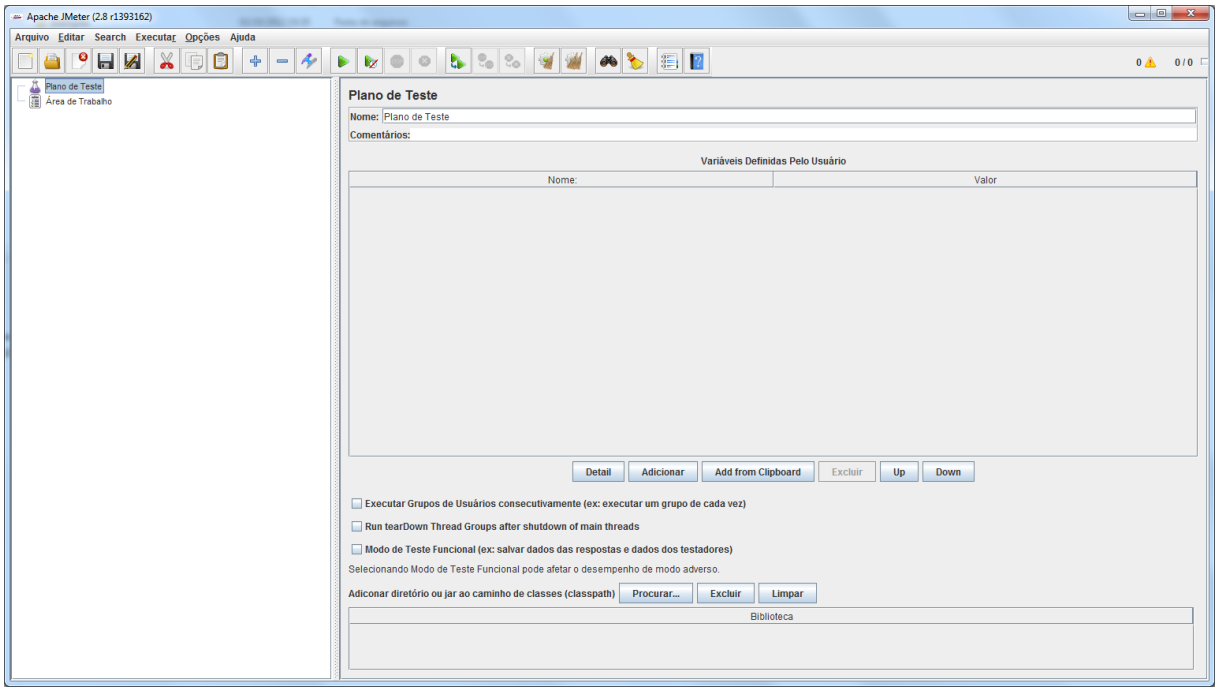

FIGURA 7 - Apache Jmeter

3.3 Node.js e MySQL

Um dos desafios do Node.js desde o começo, era a interação com banco de dados relacionais, como o MySQL, Postgres e SQL Server, pois ele não possuía suporte para tais bancos. Em seu surgimento, o foco era nos bancos NoSQL, pelo fato de os mesmos utilizarem do mesmo princípio do Node.js, serem não-bloqueáveis. Com o passar do tempo e o crescimento exponencial do Node.js, foi

constatada a necessidade deste interagir também com bancos de dados relacional. A partir de então, vários drivers, frameworks e bibliotecas foram propostas, mas seu desempenho sempre deixava a desejar, ficando atrás de linguagens convencionais como Java e PHP. Recentemente um driver ganhou espaço entre os desenvolvedores, pelo fato deste ser de fácil utilização e possuir um desempenho tão bom quanto os já consagrados para outras linguagens. Atualmente na versão 0.9 alpha, o Node-MySQL é um driver que não necessita ser compilado e é escrito totalmente em Javascript.

Sua utilização é trivial, como podemos ver no trecho de código a seguir:

```
var connection = mysql.createConnection('mysql://usuario:senha@localhost');
connection.connect();
var queryString = 'SELECT * FROM Usuarios';
connection.query(queryString, function(err, rows, fields) {
    if (err) throw err;
    for (var i in rows) {
        console.log('Usuários: ', rows[i].nome);
    }
});
```

TRECHO DE CÓDIGO 4 – Node-MySQL

3.4 Aplicação e testes

Desta forma, foi proposto a implementação de um web servisse restful simples, onde podemos comparar a performance de uma aplicação REST utilizando JAVA e uma aplicação REST utilizando Node.js. Existem duas funcionalidades distintas em ambas aplicações. Uma que realiza acesso ao banco de dados MySQL e retorna os dados no formato json, e outra que realiza um pequeno processamento e retorna o resultado para o usuário também utilizando o formato json. Assim, será possível analisarmos o comportamento das aplicações e servidores quando solicitados desta forma.

3.4.1 Resultados esperados

Chamaremos a funcionalidade que não realiza acesso ao banco de dados, de primeiro caso, e a funcionalidade a qual realiza acesso ao banco de dados, de segundo caso.

Para o primeiro caso, justamente pelos motivos citados ao longo deste trabalho, esperamos um resultado superior por parte do Node.js em relação ao JAVA. Estes resultados englobam tanto o número de requisições atendidas por segundo, quanto utilização do processador e memória. Neste caso será considerado o melhor resultado, a aplicação que atingir a menor a utilização do processador e da memória.

Já para o segundo caso, apesar de o Node.js ser não bloqueável, esperamos um desempenho semelhante de ambas aplicações no que diz respeito a número de requisições atendidas por segundo, pois estas teoricamente estão limitadas pela capacidade de resposta do banco de dados e pela qualidade da consulta executada. Neste exemplo, temos uma tabela contendo todas as cidades do Brasil, totalizando 5.565 registros. A busca executada deve retornar todos os registros encontrados onde o nome da cidade contenha uma letra, gerada aleatoriamente, e enviada por parâmetro de consulta.

3.4.2 Resultados obtidos

3.4.2.1 Primeiro caso

No primeiro teste com o Node.js, rodando uma aplicação rest simples, que não realiza acesso ao banco de dados, e o Tomcat 7, também rodando uma aplicação rest simples, foram realizados algumas variações de disparos de requisições simultâneas. Utilizando o Jmeter e o ApacheBench, é possível escolher a quantidade de threads (que representam usuários) e quantas vezes cada thread realizará uma requisição ao servidor. Desta forma, foram realizados 50 requisições para cada thread, iniciando com 16 threads e subindo este número até 128. Porém, para comprovar a eficiência e escalabilidade do Node.js, necessitamos ir além, e os testes se estenderam por mais 3 maneiras distintas. 128 threads com 500 repetições cada, totalizando 64 mil requisições, 512 threads com mil repetições cada,

totalizando 512 mil requisições, e por último uma rodada de testes com 1024 threads e 2 mil requisições para cada thread associada, totalizando 2 milhões e 48 mil requisições.

Usuários (Threads)	Desvio Padrão	Vazão (req/s)	Transferência (KB/s)	Máximo CPU (%)	Máximo de memória (MB)
16	0,29	761,2	110,01	76,00	202,88
32	0,47	1387,7	200,56	80,00	203,33
48	0,54	1933,9	279,51	84,00	207,23
64	0,66	2365,1	341,83	87,00	209,25
128	1,64	3234,1	467,41	90,00	228,08
128*	11,02	6138,5	888,34	96,00	248,40
512**	189,70	4028,7	662,44	99,00	293,25
1024***	454,48	3236,8	540,06	99,00	335,78

Dados 1 - * 500 repetições, ** mil repetições, *** duas mil repetições

TABELA 1 – Caso 1: Dados da aplicação REST em JAVA

Tanto nossa aplicação JAVA, quanto o Tomcat se comportaram muito bem, não havendo falhas, nem erros de memória ou de resposta. A vazão de requisições foi sólida, obtendo excelentes resultados em todas as situações. Foram atingidos números impressionantes em alguns casos, e pudemos perceber a mais otimizada faixa operacional de nossa aplicação neste servidor, que foi para 128 threads com 500 repetições cada, obtendo a maior vazão dos testes. O destaque fica para a utilização de memória e processador, que foram significativas em todos os casos.

Usuários (Threads)	Desvio Padrão	Vazão (req/s)	Transferência (KB/s)	Máximo do CPU	Máximo de memória ()
16	0,53	759,1	120,82	81,0%	48,42
32	0,19	1385,3	221,66	83,0%	48,69
48	0,22	1940,2	308,84	83,3%	49,14
64	0,26	2442,7	3888,84	86,0%	50,13
128	0,83	3841,5	611,49	89,2%	51,39
128*	4,76	7177,3	1142,48	93,2%	53,37
512**	26,91	5977,1	951,44	98,2%	58,95
1024***	104,34	5109,7	813,36	98,8%	61,11

Dados 2 - * 500 repetições, ** mil repetições, *** duas mil repetições

TABELA 2 – Caso 1: Desempenho da aplicação REST em Node.js

A aplicação REST em Node.js se comportou tão bem quanto a aplicação JAVA, porém com algumas diferenças significantes, como o consumo de memória, que foi cerca de um quinto do consumo da aplicação JAVA, e a utilização do CPU, que apesar de alta, em momento algum chegou a 100% de utilização. Entretanto, é uma taxa de utilização muito alta, mas o que é perfeitamente compreensivo, pois pelo fato de ter que lidar com múltiplas funções de retorno, muito mais ciclos de processamento são necessários. Em contrapartida, a aplicação Node.js entrega um número muito maior de requisições por segundo. Até 64 threads, podemos perceber que o desempenho de ambas as aplicações são quase que idênticas, porém conforme maior a carga no servidor, a diferença de desempenho entre o Node.js e o JAVA se torna evidente, especialmente o desvio padrão do tempo das respostas das requisições, onde no Node.js este é menos acentuado, demonstrando principalmente além de uma capacidade muito grande de atender requisições, atendê-las com grande qualidade.

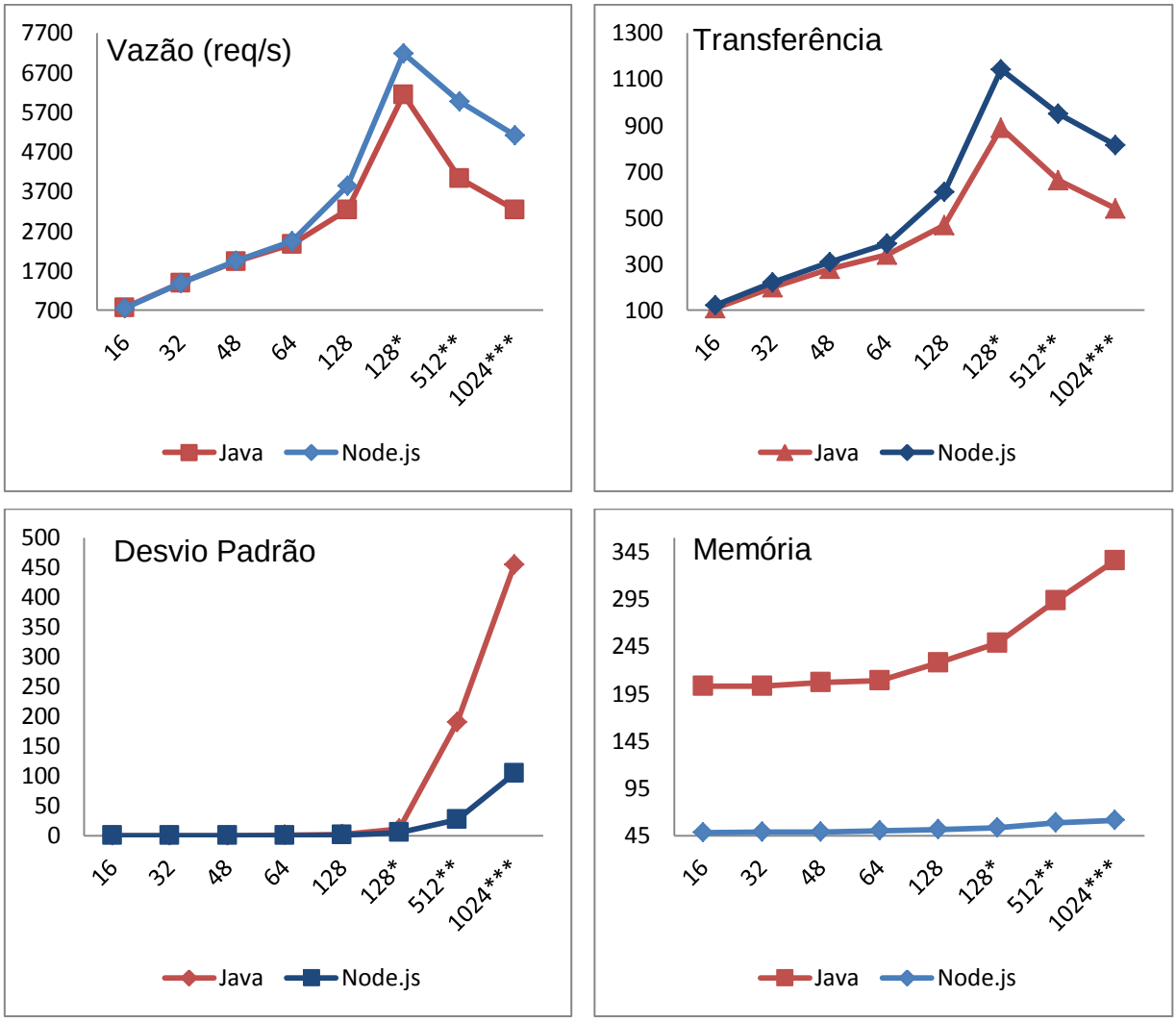

FIGURA 8 – Caso 1: Desempenho da aplicação REST: JAVA x Node.js

3.4.2.2 Segundo caso

No segundo teste, alteramos a implementação para se comunicar com o banco de dados, a fim de comprovar na prática o quão performático Node.js pode ser quando utilizando banco de dados relacionais tradicionais, apesar de esperarmos números semelhantes à ambas aplicações. Neste caso, foram realizados testes até 128 threads com 50 repetições cada, pois apesar de serem executados testes com mais threads e repetições, os resultados permaneceram praticamente inalterados.

Usuários (Threads)	Desvio Padrão	Vazão (req/s)	Transferência (KB/s)	Máximo do CPU	Máximo de memória (MB)
16	120,07	502,20	1245,19	75,80%	270,5
32	189,95	642,68	1593,52	80,10%	271,1
48	280,66	685,77	1700,36	83,70%	276,3
64	286,63	671,83	1665,79	84,00%	279
128	320,23	526,20	1304,61	86,10%	304,1

TABELA 3 – Caso 2: Desempenho da aplicação REST em JAVA

Neste caso, tanto o servidor Tomcat quanto a aplicação JAVA se mostrou estável e confiável, com uma vazão e uma taxa de transferência quase que constante. Como era esperado, houve uma pequena variação entre a quantidade de threads, requisições e a vazão. Ficou comprovado na prática que o banco de dados interfere diretamente no desempenho da aplicação, limitando a vazão da mesma à vazão do banco de dados. Mais uma vez o destaque ficou para o uso da memória, atingindo números consideráveis se comparados com a aplicação Node.js.

Usuários (Threads)	Desvio Padrão	Vazão (req/s)	Transferência (KB/s)	Máximo do CPU	Máximo de memória (MB)
16	94,88	525,62	1089,16	81,00%	53,8
32	159,95	686,78	1423,09	83,00%	54,1
48	210,74	769,30	1594,09	83,30%	54,6
64	197,58	703,85	1458,46	86,00%	55,7
128	208,43	562,79	1166,16	89,20%	57,1

TABELA 4 – Caso 2: Desempenho da aplicação REST em Node.js

A aplicação Node.js obteve um desempenho muito semelhante à aplicação JAVA, obtendo uma vazão quase que constante em torno de seiscentas requisições por segundo. Pelo fato de a aplicação Node.js apenas realizar acesso ao banco de dados e retornar um resultado, não foi possível fazer uso de seu assincronismo para este caso. No entanto, num sistema completo, este assincronismo pode representar um ganho significativo no carregamento da página como um todo, uma vez que os dados do banco serão carregados assincronamente em relação ao restante do conteúdo. O número de requisições por segundo provavelmente continuará

inalterado para popular uma página inteira, mas certamente atingirá números semelhantes ao caso 1 para responder às requisições dos usuários. Isto é, apesar de minha aplicação estar limitada à vazão do banco de dados, todo o restante do conteúdo que se encontra em minha aplicação Node.js pode ser entregue assincronamente a milhares de usuários simultaneamente, restando a eles receberem apenas os dados do banco.

O destaque positivo do Node.js ficou para o baixo consumo de memória e por não chegar a utilizar 100% do CPU, possibilitando ainda assim pequenos intervalos de processamento para outras aplicações.

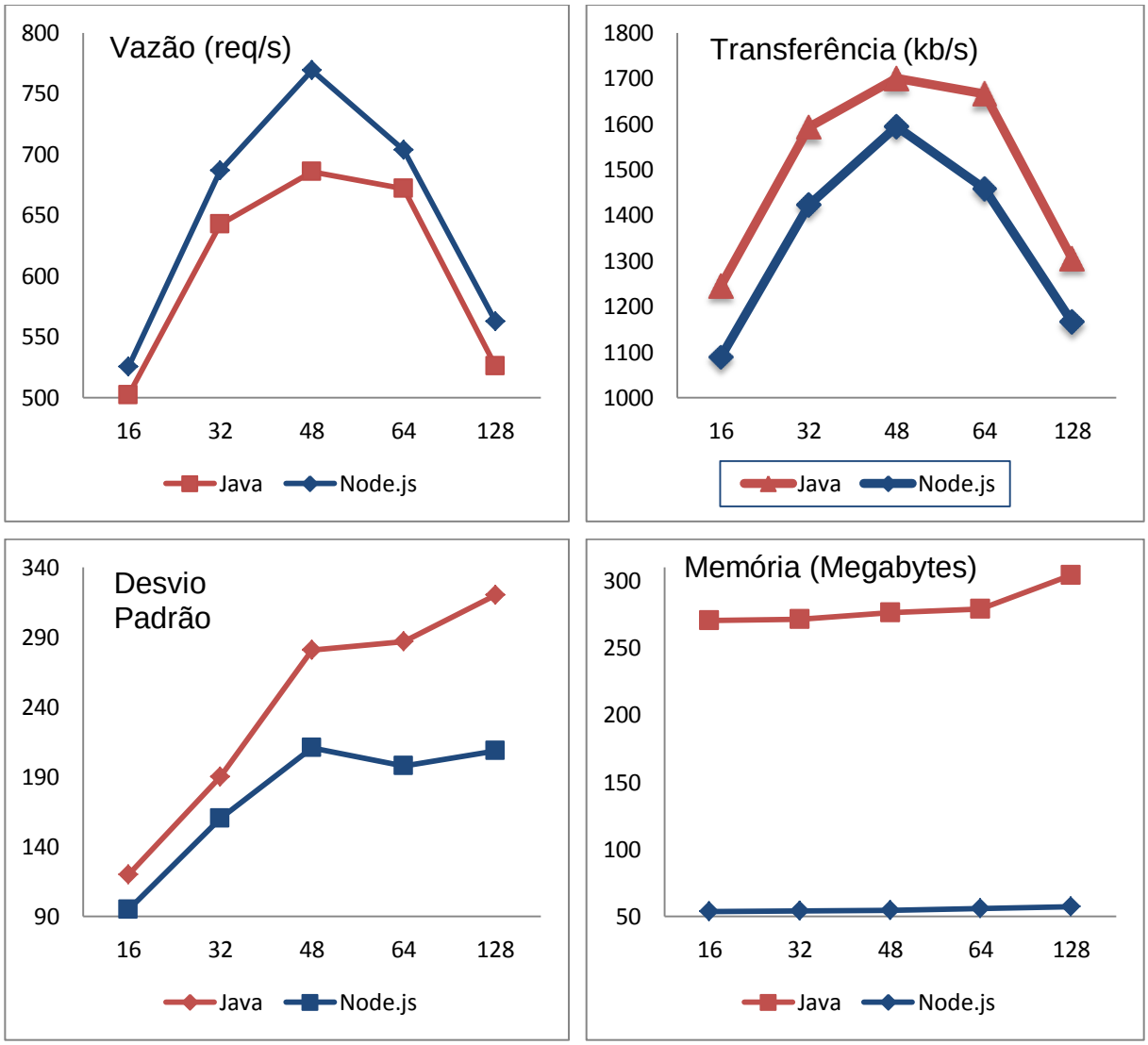

FIGURA 9 – Caso 2: Desempenho da aplicação REST: JAVA x Node.js

4 Conclusão

Neste trabalho foi desenvolvido uma API Restful em Node.js. Esta api foi validada em termos práticos através de testes práticos simulando situações reais de uso, onde se foi adotada a utilização de dois programas para tal fim, o ApacheBench e o JMeter. Para o desenvolvimento e implementação da interface foi necessário procurar o estado da arte em REST, nos quais os pressupostos de Fielding (2000) ao apresentar o estilo REST formaram o eixo que norteou esta aplicação. Os resultados desta aplicação alcançaram todos os objetivos propostos por este trabalho, possibilitando efetivamente testar e avaliar a eficácia, simplicidade e facilidade em desenvolver uma API Restful, seja esta em Node.js ou JAVA.

Acredita-se que este trabalho terá relevância para estudos posteriores, seja de outros alunos ou de pesquisadores, por se tratar de um assunto atual e de interesse à comunidade científica e aos profissionais de Ciências da Computação e Sistemas de Informação, pelo fato de apresentar informações técnicas e científicas sobre a arquitetura de software baseada no estilo REST. Também acredita-se que seu estudo sobre esta nova tendência de aplicações assíncronas, onde o Node.js foi a plataforma escolhida, seja de importância para ajudar a expor esta tecnologia empregada neste trabalho, e a encorajar os desenvolvedores a utilizá-la, seja para fins acadêmicos ou profissionais.

De maneira simplista, o Node.js é uma ferramenta muito útil quando bem utilizada, podendo evitar o escalonamento de um serviço web devido à elevada capacidade de atender requisições, especialmente pela maneira assíncrona como realiza suas tarefas. A arquitetura REST é hoje a mais utilizada para as interfaces de aplicações web, sendo de rápido desenvolvimento e fácil compreensão. Juntando estas duas tecnologias, é possível produzir serviços de alta qualidade com um desempenho muito elevado e por um custo inferior ao de desenvolvimento de aplicativos convencionais que utilizam outras arquiteturas.

Portanto, os resultados obtidos através deste trabalho apresentam contribuições importantes para a diminuição do tempo, custo e esforço necessários para a implementação de uma API Restful, facilitando a adoção desta arquitetura no desenvolvimento de sistemas para a web. Desta forma, estes resultados contribuem também de forma direta para a eficiência e a produtividade do desenvolvimento de

serviços web de maior qualidade, que por sua vez influi diretamente na melhoria da utilização das boas práticas da arquitetura REST.

5 Referências bibliográficas

Erl, T. Service-oriented architecture - a _eld guide to integrating xml and web services. 1st. ed. Prentice Hall, 2004.

Erl, T. Service-oriented architecture: Concepts, technology, and design. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

FIELDING, R. Architectural Styles and the Design of Network-based Software Architectures. 100 p. Tese (Doutorado) — University of California, 2000.

How to Node. Acessado em junho de 2012. Disponível em <http://howtonode.org/>.

IBM Standards and web services. Acessado em junho de 2012. Disponível em <http://www.ibm.com/developerworks/webservices/standards/>

Jun, Y.; Zhishu, L.; Yanyan, M. Json based decentralized sso security architecture in e-commerce. In: Proceedings of the 2008 International Symposium on Electronic Commerce and Security, ISECS '08, Washington, DC, USA: IEEE Computer Society, 2008, p. 471-475 (ISECS '08,).

Mastering Node.js. Acessado em junho de 2012. Disponível em <http://visionmedia.github.com/masteringnode/>.

Meng, J.; Mei, S.; Yan, Z. Restful web services: A solution for distributed data integration. In: Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on, 2009, p. 1 -4.

MCMILLAN, R. A RESTful approach to Web services. Fev 2003. Acesso em junho de 2012. Disponível em: <http://www.networkworld.com/ee/2003/eerest.html>.

REST Consolo 2011, REST Consolo 2011. Acessado em junho de 2012. Disponível em <http://www.restconsole.com>.

Node.js homepage. Acessado em junho de 2012. Disponível em <http://nodejs.org/>.

Node.js Manual. Acessado em junho de 2012. Disponível em <http://nodemanual.org/latest/>.

Pautasso, C.; Zimmermann, O.; Leymann, F. Restful web services vs. "big"' web services: making the right architectural decision. In: Proceeding of the 17th international conference on World Wide Web, WWW '08, New York, NY, USA: ACM, 2008, p. 805-814 (WWW '08).

Projects, Applications and Companies using Node. Acessado em junho de 2012. Disponível em <https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>.

RICHARDSON, L.; RUBY, S. Restful Web Services. Beijing: O'Reilly Media, 2007. ISBN 0596529260.

Stal, M. Web services: beyond component-based computing. Commun. ACM, v. 45, p. 71-76, 2002b.

W3C Web services architecture. Acessado em junho de 2012. Disponível em http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

W3C Xml schema. Acessado em junho de 2012. Disponível em <http://www.w3.org/XML/Schema.html>

W3C Web services description language (wsdl). Acessado em junho de 2012. Disponível em <http://www.w3.org/TR/2004/WD-wsdl20-patterns-20040326/>

WEBBER, J.; PARASTATIDIS, S. REST in Practice: Hypermedia and Systems Architecture. [S.1]: O'Reilly Media, 2010. ISBn 0596805829.

6 Apêndice

6.1 Artigo

ESTUDO DE CASO SOBRE UMA API REST UTILIZANDO A ABORDAGEM DE PROGRAMAÇÃO ORIENTADA E EVENTOS COM A PLATAFORMA NODE.JS

Thiago Vieira Puluceno

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

pulu@inf.ufsc.br

Abstract. *The development of this work is the result of implementing a hybrid architecture for distributed systems, known as REST - Representational State Transfer, and proposed by Fielding (2000), using a new system architecture, called Node.js, where the server side application is written in the JavaScript language beyond being walked to a tie events, ie no new threads are created for each new request. Finally, we performed a comparison of this implementation with an identical implementation but written in JAVA language (thread oriented), to assess the benefits, where and when to make use of this proposal.*

Resumo. *O desenvolvimento deste trabalho é resultado da implementação de uma arquitetura híbrida para sistemas distribuídos, conhecida como REST - Transferência de Estado Representacional- e proposta por Fielding (2000), utilizando uma nova arquitetura de sistemas, chamada Node.js, onde o programa do lado do servidor é escrito na linguagem JavaScript além de ser orientado a um laço de eventos, isto é, novas threads não são criadas para cada nova requisição. Por fim, foi realizado um comparativo desta implementação, com uma implementação idêntica, porém escrita na linguagem JAVA (orientada a threads), a fim de avaliar os benefícios, onde e quando deve se fazer uso desta proposta.*

1. Introdução

Aplicações Web são cada vez mais frequentes entre os incontáveis sítios espalhados pela Internet e podem ser implementadas utilizando-se um universo de padrões, ferramentas, tecnologias, arquiteturas e estilos arquiteturais disponíveis especificamente para tal tarefa.

Fielding (2000), em sua dissertação de doutorado, propõe um novo estilo arquitetural, chamado Representational State Transfer (REST). Embora REST possua um conjunto de restrições que o caracterizam como um estilo arquitetural, ainda permite muitas liberdades em sua aplicação a alguma arquitetura já existente. Por essa liberdade de aplicação, este estilo vem cada vez mais sendo utilizado por aquelas aplicações web, que com o decorrer do tempo necessitaram cada vez mais se comunicarem entre si. REST surge atualmente como uma forma rápida, eficiente e simples de solucionar esse problema de comunicação entre diferente sistemas construídos utilizando diferentes linguagens de programação.

Porém, como o volume de tráfego nestes sistemas tem crescido de forma vertiginosa, há uma busca incessantes para que possamos prover uma maior quantidade de serviços para mais usuário, a um custo menor de implementação e manutenção. Node.js surge no momento

como uma opção muito atraente para solucionar este tipo de dificuldade, uma vez que suas aplicações são de fácil e rápido desenvolvimento, utiliza uma linguagem de programação que grande maioria dos programadores conhece, e é altamente escalar, permitindo maximizar a utilização dos recursos de um servidor.

2. Node.js

Node.js é muitas coisas, mas principalmente é uma maneira de rodar Javascript fora do navegador web. Podemos classificá-lo como uma arquitetura de sistemas, que dentre tantas, possui alguns objetivos específicos. Seu principal objetivo é permitir que fossem escritas aplicações altamente escaláveis para uso em rede, ou seja, na Internet.

Outras linguagens de programação que utilizam de diferentes arquiteturas, partem desta premissa, serem escaláveis. Porém Node.js é diferente, uma vez que ela utiliza a linguagem Javascript para escrever seus programas. A priori, podemos pensar que Javascript é uma linguagem de programação utilizada apenas para escrevermos programas que executarão no navegador do usuário que está visualizando nossa página, e que portanto não faz muito sentido escrevermos um sistema inteiro utilizando Javascript.

Mas Node.js é muito mais do que um simples interpretador de Javascript. Ele foi construído utilizando o interpretador de Javascript mais rápido da atualidade. O V8 é o interpretador de Javascript que é utilizado no navegador Chrome, do Google, e foi escrito em C++ pela equipe de desenvolvimento do Google Chrome.

A execução de Javascript no V8 é extremamente rápida e executa muito bem em diversas circunstâncias. O Node.js ajustou o V8 para trabalhar melhor em outros contextos além do navegador, principalmente, fornecendo APIs alternativa que são otimizadas para casos de uso específicos. Por exemplo, num contexto de servidor, a manipulação de dados binários é muitas vezes necessária. Isto é mal suportado pelo Javascript e, como resultado, no próprio V8. O Node.js adiciona a classe Buffer para suas implementações permitirem fácil manipulação de dados binários, de uma maneira que se torna fácil seu uso, além de ser eficiente quando se trata de memória. O Node.js não apenas proporciona o acesso direto ao V8 em tempo de execução, como também o torna mais útil para os contextos em que as pessoas usam o Node.js.

O V8 utiliza das técnicas mais recentes de compiladores, permitindo que códigos escritos numa linguagem de alto nível, como o Javascript, possuam desempenho tão bom e custos reduzidos em relação a códigos escritos em linguagens de mais baixo nível como C, por exemplo. Este foco sobre o desempenho é um aspecto-chave do Node.js

Mas como o Node.js permite que minhas aplicações sejam escaláveis? Muitos programas escritos em diferentes linguagens utilizam threads ou até mesmo conseguem ser multitarefa, mas qual é o melhor que pode ser feito hoje? A resposta está na forma como o Node.js trata as operações de entrada e saída. Ele é direcionado a um laço de eventos, e as operações de entrada e saída não poderiam ser diferentes. Além de direcionadas a evento, as operações de entrada e saída não bloqueiam o sistema enquanto aguardam por uma resposta, do disco por exemplo.

Por que este tipo de configuração é ideal para o Node? O Javascript é uma excelente linguagem para programação direcionada a eventos, pois estes eventos permitem funções e fechamentos anônimos e mais importantes, a sintaxe é familiar para quase todos que alguma vez já programaram. As funções de call-back que são chamadas quando um evento ocorre podem ser escritas no mesmo local onde você captura o evento. Portanto, é fácil de codificar,

fácil de manter, sem estruturas orientadas a objetos complicadas, sem interfaces e sem potencial para excessos na arquitetura. Basta aguardar um evento, escrever uma função de call-back e a programação direcionada a eventos toma conta de tudo. [Michael Abernethy]

Logo, de uma maneira sucinta, podemos dizer que o Node.js é uma maneira de escrevermos programas que possuirão altíssima concorrência, são altamente escaláveis e são puramente orientados a eventos que não bloqueiam a infraestrutura (banco de dados, por exemplo) a qual utilizam.

3. REST

REST nos propõe uma interface muito mais simples do que o SOAP, porém com algumas limitações. Desenvolver utilizando REST é mais fácil e mais rápido e esta é a razão pela qual APIs web mais famosas são baseados em REST.

As limitações estão relacionadas com o protocolo: com REST somos obrigados a utilizar o protocolo HTTP. Por isso, é difícil de manusear diferentes transportes como MQ ou JMS. Então, se estivermos criando um aplicativo para ser usado exclusivamente na web, REST pode ser uma opção. Se estivermos projetando uma arquitetura orientada a serviços para uma empresa, que precisa interligar vários sistemas que utilizam transporte de múltiplos canais e com operações sofisticadas para ser modelados com BPMs, então é melhor ir com SOAP.

Estamos pagando pela complexidade do desenvolvimento em troca da flexibilidade do SOAP. Interfaces SOAP são descritas pelo padrão WSDL que parece realmente complicado e não é suposto para ser lido por seres humanos.

As aplicações construídas nos moldes de REST são denominadas RESTful. Aplicações RESTful simplesmente contam com o built-in HTTP de segurança (protocolo HTTPS), com SOAP podemos ter opções mais sofisticadas, como o WS-Security, que nos permite criptografar apenas partes das mensagens e move o manuseio de segurança da camada de transporte para o camada de mensagem.

REST é mais jovem do SOAP, por isso não é bem suportado por produtos comerciais (tanto BPM e produtos ESB). No entanto as novas versões do REST recebem grande apoio de alguns ESBs, como o Mule e o OpenESB.

Serviços RESTful não tem um descritor estrito, foi tentado definir algo do mesmo tipo do WSDL, ele é chamado WADL (https://wadl.dev.java.net/). O WADL de momento não é um padrão comum, e sua adoção é muito baixa.

Se estivermos planejando executar uma implantação de um serviço muito grande com muitas interfaces de dados, teremos grandes benefícios usando os Serviços RESTful. REST é baseado 100% na rfc HTTP (http://www.w3.org/Protocols/rfc2616/rfc2616.html). Muitas das interfaces RESTful que vemos atualmente na Internet não suportam a especificação HTTP, e o resultado é que elas não poderiam se beneficiar desta especificação em termos de desempenho, cache, negociação de conteúdo e segurança.

Um grande motivo para a utilização de interfaces REST, é que elas pode facilmente suportar content-types/mime-types diferentes. Outra razão pela qual as pessoas estão usando interfaces REST, é para sua aplicação ser escalável como a web e o Google. Isso é possível quando estivermos utilizando o puro e simples protocolo HTTP. Temos o apoio de tantos produtos padrão como web proxy caches (como SQUID) ou outros para escalar nossa aplicação com um esforço muito baixo. Para as comunidades sociais, temos agora um novo

padrão para serviços REST de segurança. Ele é chamado OAuth (http://oauth.net/), que faz da autenticação via HTTP uma tarefa muito fácil.

Alguns outros exemplos onde os serviços REST são dominantes são a Amazon AWS, onde são oferecidos serviços para armazenamento (S3) que possuem interfaces SOAP e REST, mas mais de 90% dos clientes estão usando REST e o Google Maps, que está usando REST URIs para fornecer partes de mapas e o Yahoo API é em sua maior parte construída com interfaces REST.

Serviços REST são predominantes nas APIs WEB 2.0 para Mashups, como o OpenSocial API. Interfaces em REST são muito boas para aplicativos que lidam diretamente com o banco de dados, e principalmente quando algum cliente necessita de uma rápida e ágil integração.

4. Estudo de caso – Implementação de uma API Restful em Node.js

A implementação será simples e objetiva, restringindo a implementação dos métodos HTTP a apenas o GET, pois este método por si só é capaz de demonstrar a performance e escalabilidade do Node.js pelo motivo deste ser não-bloqueável.

Se fará uso do Node.js em sua última versão estável até o momento (versão 0.8.14) e seu servidor embutido.

O desenvolvimento será realizado utilizando o sistema operacional Ubuntu, na versão 11.04 32bits e também Windows, na versão 7 SP1. A ferramenta Aptana Studio será utilizada para escrever os códigos em Javascript, e a IDE Eclipse será utilizada para escrever e executar as implementações em JAVA.

Com o intuito de compararmos o desempenho da aplicação, será realizada uma implementação nos mesmos padrões utilizando a linguagem JAVA, em sua última versão, v1.7b09, e a especificação REST mais recente para esta linguagem, a JAX-RS 1.1. Será utilizado também a implementação da especificação JAX-RS, RestEasy, em sua última versão estável, 2.3.5.Final. Como servidor, será utilizado o JBoss Application Server na versão 7.1.1.Final. O banco de dados, que será comum a ambas aplicações, será o MySQL na versão 5.5.28. A máquina utilizada para realizar as implementações e os testes, é um Intel Core i7 3.5ghz com 6gb DDR3 1600mhz. Chamaremos a funcionalidade que não realiza acesso ao banco de dados, de primeiro caso, e a funcionalidade a qual realiza acesso ao banco de dados, de segundo caso.

4.1. Primeiro Caso

Tanto nossa aplicação JAVA, quanto o Tomcat se comportaram muito bem, não havendo falhas, nem erros de memória ou de resposta. A vazão de requisições foi sólida, obtendo excelentes resultados em todas as situações. Foram atingidos números impressionantes em alguns casos, e pudemos perceber a mais otimizada faixa operacional de nossa aplicação neste servidor, que foi para 128 threads com 500 repetições cada, obtendo a maior vazão dos testes. O destaque fica para a utilização de memória e processador, que foram significativas em todos os casos.

A aplicação REST em Node.js se comportou tão bem quanto a aplicação JAVA, porém com algumas diferenças significantes, como o consumo de memória, que foi cerca de um quinto do consumo da aplicação JAVA, e a utilização do CPU, que apesar de alta, em momento algum chegou a 100% de utilização. Entretanto, é uma taxa de utilização muito alta,

mas o que é perfeitamente compreensivo, pois pelo fato de ter que lidar com múltiplas funções de retorno, muito mais ciclos de processamento são necessários. Em contrapartida, a aplicação Node.js entrega um número muito maior de requisições por segundo. Até 64 threads, podemos perceber que o desempenho de ambas as aplicações são quase que idênticas, porém conforme maior a carga no servidor, a diferença de desempenho entre o Node.js e o JAVA se torna evidente, especialmente o desvio padrão do tempo das respostas das requisições, onde no Node.js este é menos acentuado, demonstrando principalmente além de uma capacidade muito grande de atender requisições, atendê-las com grande qualidade.

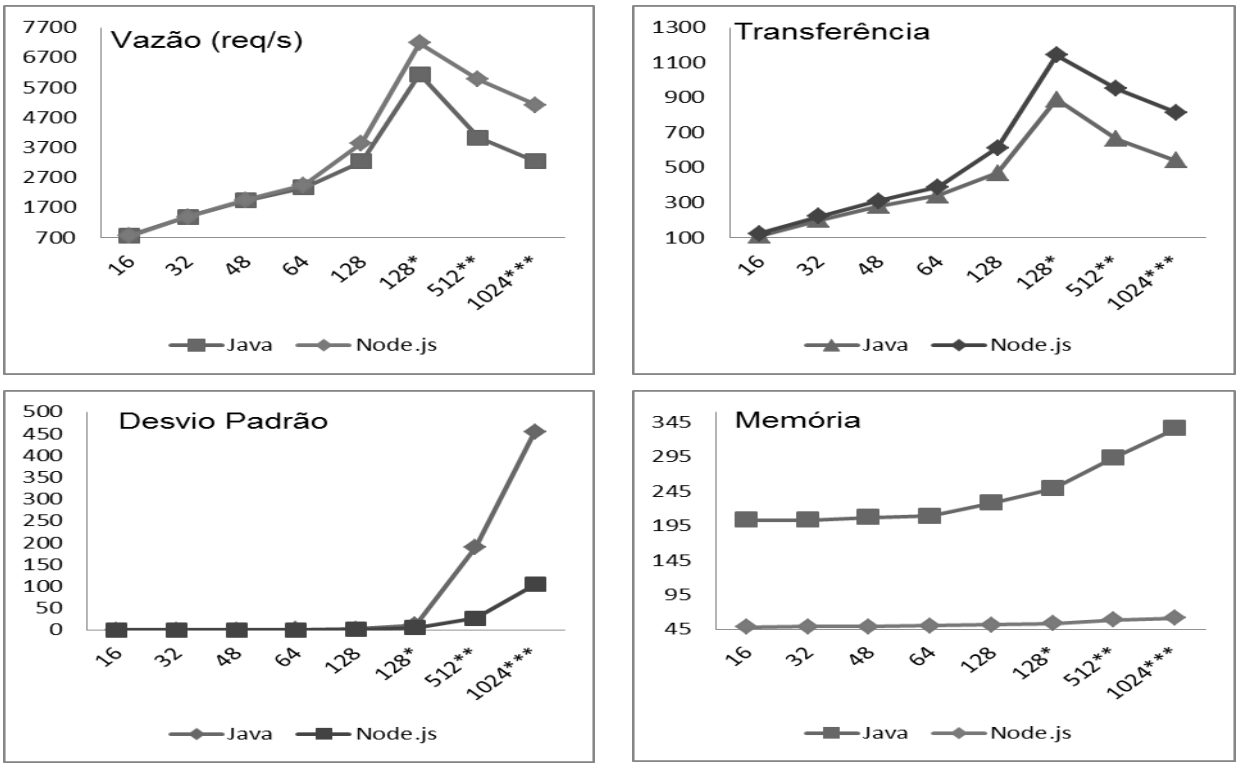

Figura 1. Caso 1: Java x Node.js

4.2. Segundo Caso

No segundo teste, alteramos a implementação para se comunicar com o banco de dados, a fim de comprovar na prática o quão performático Node.js pode ser quando utilizando banco de dados relacionais tradicionais, apesar de esperarmos números semelhantes à ambas aplicações. Neste caso, foram realizados testes até 128 threads com 50 repetições cada, pois apesar de serem executados testes com mais threads e repetições, os resultados permaneceram praticamente inalterados.

Neste caso, tanto o servidor Tomcat quanto a aplicação JAVA se mostrou estável e confiável, com uma vazão e uma taxa de transferência quase que constante. Como era esperado, houve uma pequena variação entre a quantidade de threads, requisições e a vazão. Ficou comprovado na prática que o banco de dados interfere diretamente no desempenho da aplicação, limitando a vazão da mesma à vazão do banco de dados. Mais uma vez o destaque ficou para o uso da memória, atingindo números consideráveis se comparados com a aplicação Node.js.

A aplicação Node.js obteve um desempenho muito semelhante à aplicação JAVA, obtendo uma vazão quase que constante em torno de seiscentas requisições por segundo. Pelo

fato de a aplicação Node.js apenas realizar acesso ao banco de dados e retornar um resultado, não foi possível fazer uso de seu assincronismo para este caso. No entanto, num sistema completo, este assincronismo pode representar um ganho significativo no carregamento da página como um todo, uma vez que os dados do banco serão carregados assincronamente em relação ao restante do conteúdo. O número de requisições por segundo provavelmente continuará inalterado para popular uma página inteira, mas certamente atingirá números semelhantes ao caso 1 para responder às requisições dos usuários. Isto é, apesar de minha aplicação estar limitada à vazão do banco de dados, todo o restante do conteúdo que se encontra em minha aplicação Node.js pode ser entregue assincronamente a milhares de usuários simultaneamente, restando a eles receberem apenas os dados do banco.

O destaque positivo do Node.js ficou para o baixo consumo de memória e por não chegar a utilizar 100% do CPU, possibilitando ainda assim pequenos intervalos de processamento para outras aplicações.

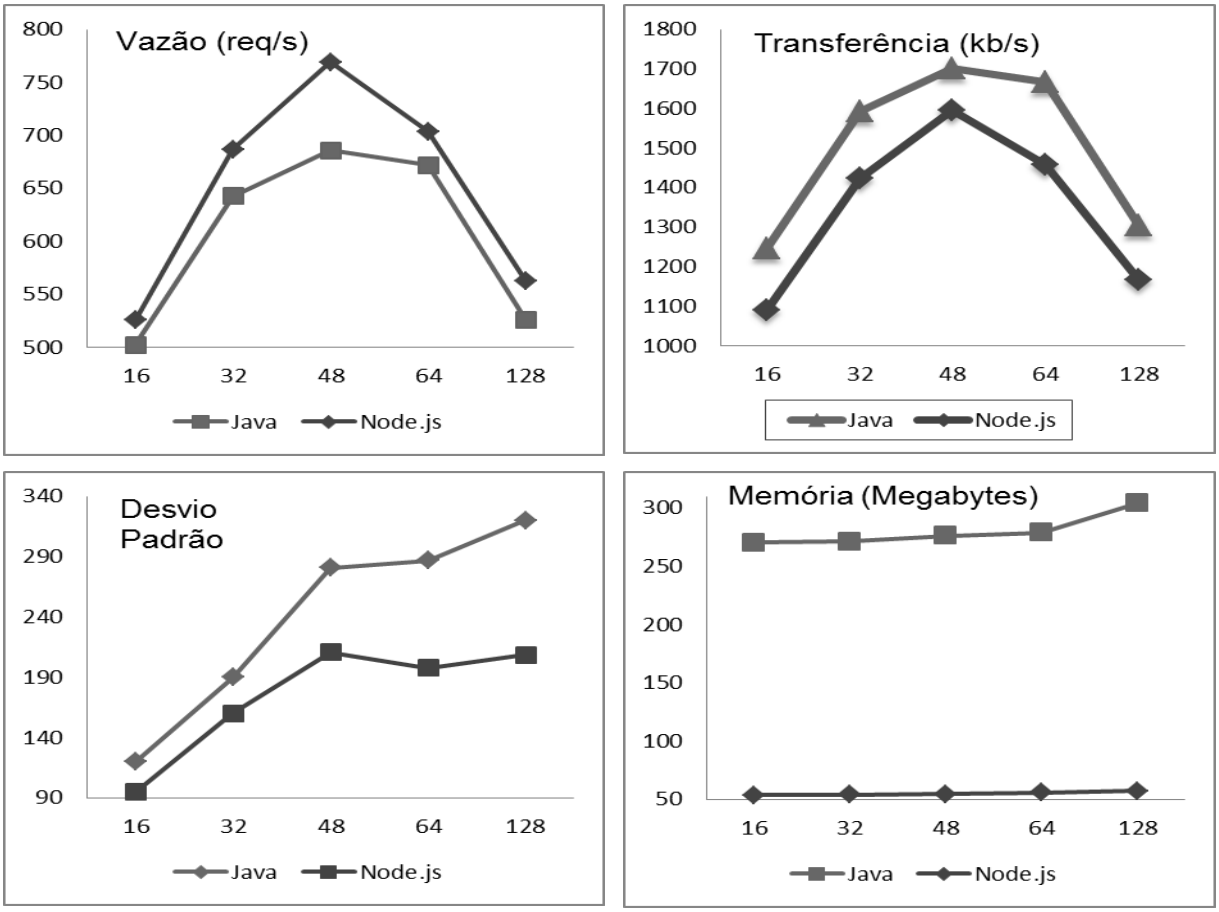

Figura 2. Caso 2: Java x Node.js

5. Conclusões

Neste trabalho foi desenvolvido uma API Restful em Node.js. Esta api foi validada em termos práticos através de testes práticos simulando situações reais de uso, onde se foi adotada a utilização de dois programas para tal fim, o ApacheBench e o JMeter. Para o desenvolvimento e implementação da interface foi necessário procurar o estado da arte em REST, nos quais os pressupostos de Fielding (2000) ao apresentar o estilo REST formaram o eixo que norteou

esta aplicação. Os resultados desta aplicação alcançaram todos os objetivos propostos por este trabalho, possibilitando efetivamente testar e avaliar a eficácia, simplicidade e facilidade em desenvolver uma API Restful, seja esta em Node.js ou JAVA.

Acredita-se que este trabalho terá relevância para estudos posteriores, seja de outros alunos ou de pesquisadores, por se tratar de um assunto atual e de interesse à comunidade científica e aos profissionais de Ciências da Computação e Sistemas de Informação, pelo fato de apresentar informações técnicas e científicas sobre a arquitetura de software baseada no estilo REST. Também acredita-se que seu estudo sobre esta nova tendência de aplicações assíncronas, onde o Node.js foi a plataforma escolhida, seja de importância para ajudar a expor esta tecnologia empregada neste trabalho, e a encorajar os desenvolvedores a utilizá-la, seja para fins acadêmicos ou profissionais.

De maneira simplista, o Node.js é uma ferramenta muito útil quando bem utilizada, podendo evitar o escalonamento de um serviço web devido à elevada capacidade de atender requisições, especialmente pela maneira assíncrona como realiza suas tarefas. A arquitetura REST é hoje a mais utilizada para as interfaces de aplicações web, sendo de rápido desenvolvimento e fácil compreensão. Juntando estas duas tecnologias, é possível produzir serviços de alta qualidade com um desempenho muito elevado e por um custo inferior ao de desenvolvimento de aplicativos convencionais que utilizam outras arquiteturas.

Portanto, os resultados obtidos através deste trabalho apresentam contribuições importantes para a diminuição do tempo, custo e esforço necessários para a implementação de uma API Restful, facilitando a adoção desta arquitetura no desenvolvimento de sistemas para a web. Desta forma, estes resultados contribuem também de forma direta para a eficiência e a produtividade do desenvolvimento de serviços web de maior qualidade, que por sua vez influi diretamente na melhoria da utilização das boas práticas da arquitetura REST.

6. Referências

Erl, T. Service-oriented architecture - a _eld guide to integrating xml and web services. 1st. ed. Prentice Hall, 2004.

Erl, T. Service-oriented architecture: Concepts, technology, and design. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

FIELDING, R. Architectural Styles and the Design of Network-based Software Architectures. 100 p. Tese (Doutorado) — University of California, 2000.

How to Node. Acessado em junho de 2012. Disponível em <http://howtonode.org/>.

IBM Standards and web services. Acessado em junho de 2012.

Disponível em <http://www.ibm.com/developerworks/webservices/standards/>

Jun, Y.; Zhishu, L.; Yanyan, M. Json based decentralized sso security architecture

in e-commerce. In: Proceedings of the 2008 International Symposium on Electronic

Commerce and Security, ISECS '08, Washington, DC, USA: IEEE Computer Society,

2008, p. 471-475 (ISECS '08,).

Mastering Node.js. Acessado em junho de 2012. Disponível em <http://visionmedia.github.com/masteringnode/>.

Meng, J.; Mei, S.; Yan, Z. Restful web services: A solution for distributed data integration. In: Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on, 2009, p. 1 -4.

MCMILLAN, R. A RESTful approach to Web services. Fev 2003. Acesso em junho de 2012. Disponível em: <http://www.networkworld.com/ee/2003/eerest.html>.

REST Consolo 2011, REST Consolo 2011. Acessado em junho de 2012. Disponível em <http://www.restconsole.com>.

Node.js homepage. Acessado em junho de 2012. Disponível em <http://nodejs.org/>.

Node.js Manual. Acessado em junho de 2012. Disponível em <http://nodemanual.org/latest/>.

Pautasso, C.; Zimmermann, O.; Leymann, F. Restful web services vs. "big"' web services: making the right architectural decision. In: Proceeding of the 17th international conference on World Wide Web, WWW '08, New York, NY, USA: ACM, 2008, p. 805-814 (WWW '08).

Projects, Applications and Companies using Node. Acessado em junho de 2012. Disponível em <https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>.

RICHARDSON, L.; RUBY, S. Restful Web Services. Beijing: O'Reilly Media, 2007. ISBN 0596529260.

Stal, M. Web services: beyond component-based computing. Commun. ACM, v. 45,

p. 71-76, 2002b.

W3C Web services architecture. Acessado em junho de 2012. Disponível em http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

W3C Xml schema. Acessado em junho de 2012. Disponível em <http://www.w3.org/XML/Schema.html>

W3C Web services description language (wsdl). Acessado em junho de 2012. Disponível em <http://www.w3.org/TR/2004/WD-wsdl20-patterns-20040326/>

WEBBER, J.; PARASTATIDIS, S. REST in Practice: Hypermedia and Systems Architecture. [S.1]: O'Reilly Media, 2010. ISBn 0596805829.

6.2 Códigos-fonte

RestServer.js

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

var client = mysql.createClient({
        user: 'teste'
        password: 'teste'
        host: '1.1.1.10'
        port: '3306'
        schema: 'RedeFood'
});
client.query('USE `rest-test`');

http.createServer(function (req, res) {
        res.writeHead(200, {'Content-Type': 'application/json'});
        var url_parts = url.parse(req.url, true);
        var query = url_parts.query;
        var beginning = query['id'];
        client.query('SELECT * FROM City c WHERE c.iCity = '"+id+"'", function
selectCb(error, results, fields) {
        var result = [];
        for (var i=0; i<results.length; i++) {
        var row = results[i];
        result.push({'id': row['id'], 'name': row['name']});
        }
        res.end(JSON.stringify({'names': result}));
        });
}).listen(8080, '127.0.0.1');
var http = require('http');
var url = require('url');
```

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


http.createServer(function (req, res) {
	res.writeHead(200, {'Content-Type': 'application/json'});
	var url_parts = url.parse(req.url, true);
	var query = url_parts.query;
	var nome = query['nome'];
	res.end(JSON.stringify({'names': nome}));
}).listen(8080, '0.0.0.0');
console.log("Server listening na porta 8080");
```

JaxApplication.java

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/")
public class JaxApplication extends Application {
}
```

Name.java

```
import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;


@Entity
public class Name implements Serializable {
	private static final long serialVersionUID = 3741178770908674782L;
```

```
	@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
	private Long id;

	private String name;

	public Long getId() {
	return id;
	}

	public void setId(Long id) {
	this.id = id;
	}

	public String getName() {
	return name;
	}

	public void setName(String name) {
	this.name = name;
	}
}
```

Response.java

```
import java.util.List;


public class Response {
	private List<Name> names;

	public List<Name> getNames() {
	return names;
	}
```

```
	public void setNames(List<Name> names) {
	this.names = names;
	}
}
```

TestService.java

```
import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

@Stateless
@Path("/test")
public class TestService {
	@PersistenceContext
	private EntityManager entityManager;

	@GET
	@Produces(MediaType.APPLICATION_JSON)
	@SuppressWarnings("unchecked")
	public Response action(@QueryParam("beginning") @DefaultValue("")
String beginning) {
	Response response = new Response();
```

```
        Query query = getEntityManager().createQuery("FROM Name WHERE
name LIKE :name");
        query.setParameter("name", beginning + "%");
        response.setNames((List<Name>)query.getResultList());
        return response;
        }

        public EntityManager getEntityManager() {
        return entityManager;
        }

        public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
        }}
```