ARQUITETURA E TESTES DE SERVIÇOS WEB DE ALTO DESEMPENHO COM NODE.JS E MONGODB

Ricardo Schroeder[1] , Fernando dos Santos[1]
[1]Universidade do Estado de Santa Catarina - UDESC
ricardo.schroeder.bsi@gmail.com, fernando.santos@udesc.br

Resumo
Este artigo tem como objetivo apresentar e testar uma arquitetura de servidores web de alto desempenho utilizando Node.JS. A alta performance tem sido um requisito cada vez mais almejado em aplicações web, visto o aumento do número de usuários que possuem acesso a plataforma. Com o objetivo de maximizar o desempenho será utilizado outra categoria de banco de dados, conhecido como banco de dados NoSQL, sendo utilizado o MongoDB (orientado a documentos). Para mensurar o ganho ao se utilizar esta arquitetura, foi realizado um comparativo da mesma com outras arquiteturas web semelhantes: Java/Netty e também tradicionais: Apache/PHP. A forma de comparação das arquiteturas se dará através do desenvolvimento de uma pequena aplicação intitulada de "encurtador de URL's". Esta aplicação foi implementada e executada em cada arquitetura avaliada. Para comparar as arquiteturas os seguintes aspectos foram avaliados: número de requisições, número de requisições com erro, tempo de resposta médio, vazão, requisições por tempo, uso de memória e CPU. A arquitetura proposta obteve o melhor desempenho em todos os cenários exceto no uso de memória, tendo em alguns casos performance seis vezes superior a outras arquiteturas, comprovando o alto desempenho da mesma.

Palavras-chave: Servidores Web. Alto Desempenho. Teste de Software.

Abstract
This article presents and test a architecture of high-performance web servers using Node.js High performance has been an increasingly desired requirement in web applications, since the number of users who have access to the platform has been increased. In order to maximize the performance will be used another category database known as NoSQL, MongoDB being used (document-oriented). To measure the gain when using this architecture , a comparison of it with other similar web architecture: Java/Netty and also traditional: Apache/PHP. The way to compare the architectures will be through the development of a small application titled " URL shortener 's ". This application was implemented and executed on each evaluated architecture. To compare the architectures the following aspects were evaluated: number of requests, number of error requests, average response time, throughput, requests for time, memory usage and CPU. The proposed architecture achieved the best performance in all scenarios except in memory usage, and in some cases performance six times higher than other architectures, confirming the high performance of the same.

Keywords: *WebServer. High Performance. Software Test.*

1. Introdução

A escalabilidade de um sistema é um ponto importante na etapa de projeção de qualquer software na internet, pois o mesmo pode obter uma taxa de crescimento muito rápida se comparado a outros tipos de software.

> "Uma série de fatores pode induzir o crescimento, incluindo o mercado, o setor e os fatores da economia. Se uma organização cresce mais rápido do que o previsto, experimenta todos os tipos de degradações de desempenho, que vão desde falta de espaço em disco até uma desaceleração na velocidade de

> operação. Antever o crescimento esperado - e inesperado - é fundamental para a construção de sistemas escaláveis que possam suportá-lo.” (BALTZAN, 2012)

A escalabilidade é importante para a web, pois neste ambiente é comum haver a utilização de servidores adicionais para se manter a performance (conhecido como escalabilidade horizontal). De acordo com Fowler (2003), “um sistema escalável é aquele que lhe permite adicionar hardware e obter uma melhora de desempenho proporcional.” Um fato importante é que a arquitetura de um sistema pode influenciar em sua escalabilidade, impedindo por exemplo que novos recursos sejam adicionados, ou até mesmo que estes recursos não tenham impacto na performance do sistema.

Com este trabalho será introduzido uma nova abordagem para o desenvolvimento de serviços web onde tenha-se o requisito de “muitos acessos simultâneos”, o objetivo principal é a especificação de uma arquitetura de servidores web de alto desempenho utilizando Node.JS e banco de dados MongoDB de modo a extrair o máximo de capacidade de um hardware para determinado serviço, podendo ter escalabilidade se necessário.

A arquitetura proposta neste trabalho prevê a utilização de um servidor Web com utilização de IO não bloqueante (o que permite lidar com um maior número de requisições simultâneas), bem como a utilização de banco de dados NoSQL, que foram projetados para alto desempenho e para se manterem escaláveis.

Como forma de testar o desempenho da arquitetura foi especificada uma aplicação de exemplo, denominada de encurtador de URL’s. A escolha desta como cenário de execução se deve ao fato de ser uma aplicação existente no mercado, de baixa complexidade (podendo ser facilmente implementada em diversas arquiteturas) e que tem como requisito básico ser de alto desempenho. Por alto desempenho entende-se que irão existir muitos acessos simultâneos ao serviço, bem como espera-se um tempo de resposta o mais baixo possível do servidor nas requisições efetuadas. Esta especificação será bem objetiva, especificando o comportamento algorítmico. A especificação desses algoritmos é fundamental para que as implementações a serem realizadas em diferentes linguagens sejam o mais parecida possível.

Após o desenvolvimento da aplicação especificada na arquitetura proposta, a mesma foi desenvolvida nos seguintes ambientes: Node.JS/PostgreSQL, Netty/Java/MongoDB, Netty/Java/PostgreSQL, Apache/PHP/MongoDB e Apache/PHP/PostgreSQL.

A escolha da linguagem Java e PHP como referências a arquitetura proposta se deve ao fato de as mesmas serem largamente utilizadas para o desenvolvimento de aplicações web. E a escolha do banco de dados PostgreSQL em relação ao MongoDB se deve ao fato de que os banco de dados relacionais ainda são largamente utilizados.

O teste de desempenho consiste em avaliar os seguintes aspectos: número de requisições, número de requisições com erro, tempo de resposta médio, vazão, requisições por tempo, uso de memória e CPU. Com os resultados obtidos nos testes de desempenho pode ser verificado que as arquiteturas de alto desempenho obtiveram desempenho superior em quase todos os aspectos, como por exemplo, tempo médio de resposta e vazão, que chegou a ser seis vezes maior do que em outras arquiteturas.

Este artigo está organizado da seguinte forma: a seção 2 apresenta uma revisão teórica dos principais assuntos referenciados no artigo. A seção 3 especifica a definição da arquitetura proposta. A seção 4 traz a definição dos testes de desempenho bem como o desenvolvimento das arquiteturas. A seção 5 apresenta a análise dos resultados obtidos durante os testes de desempenho e por fim, as conclusões e trabalhos futuros são apresentados na seção 6.

2. Revisão teórica

2.1 NoSQL

O termo NoSQL (Not Only SQL) é aplicado de maneira bem ampla nos mais recentes bancos de dados que não são considerados relacionais como BigTable, Cassandra, MongoDB, Memcached, etc. “Os entusiastas destes bancos afirmam que determinados sistemas podem ser desenvolvidos mais facilmente, ser mais escalável e ter mais perfomance utilizando os mesmos” (SADALAGE, 2013). Porém, a ausência de algumas características (variando de banco para banco) como controle transacional, faz com que sua utilização na prática seja especifico para algumas aplicações.

> “Quando se analisa a possibilidade de se optar por uma estratégia NoSQL em detrimento de um SGBD tradicional, é preciso levar em consideração algumas questões básicas, como, por exemplo, os critérios de escalonamento, consistência de dados e disponibilidade” (BRITO, 2010)

Como o próprio nome já diz, não se usa SQL para fazer busca aos dados nestes bancos, tendo cada um sua maneira especifica de realizar “queries”. Outras características é que grande parte destes bancos são projetos “open-source” e que já nasceram com foco na escalabilidade, tendo como foco a alta performance, bem como capacidade básica a de rodar em cluster, garantindo mais facilmente a alta disponibilidade quando comparado com banco de dados relacionais.

2.2 MongoDB

O MongoDB é “um banco de dados orientado a documentos que provê alta performance, alta disponibilidade e fácil escalabilidade” (MongoDB Manual, 2013). Seus dados são representados através de JSON (JavaScript Object Notation). Em comparação aos SGBDR (Sistema Gerenciador de Banco de Dados Relacional) algumas analogias podem ser feitas: uma tabela pode ser comparada com uma coleção e um documento pode ser comparado a uma linha de dados.

A escolha do banco de dados MongoDB como solução NoSQL se deve ao grande grau de maturidade que o mesmo já possui, bem como por uma maior adoção pela comunidade.

2.3 Input/Output Não Bloqueante

A maioria das operações de input/output feitas ao sistema operacional são bloqueantes. No cenário tradicional, isto significa que é feita uma solicitação de leitura ao SO e a mesma irá bloquear o programa até que a operação tenha sido efetuada. Quando se lida com muitas requisições simultâneas “bloquear apresenta um grande problema na programação cliente/servidor, pois um bloqueio, por menor que seja, pode fazer com que todo o programa trave, enquanto os outros pedidos tem que esperar” (STEIN, 2004)

O uso de I/O não bloqueante, também conhecido como I/O assíncrono ou I/O orientado a eventos, muda o paradigma tradicional, pois basicamente você registra um *callback* que é chamado quando a operação é concluída.

O Node.JS é um servidor para a internet desenvolvido para ser escalável. De acordo com seu fabricante (NODE.JS, 2013) “o uso de I/O não bloqueante é o que torna leve e eficiente”. Apesar de suportar diversos protocolos o mesmo é bastante utilizado para a construção de aplicações web, tendo como característica o suporte ao protocolo HTTP e WebSockets.

A linguagem utilizada para o desenvolvimento de aplicações no Node.JS é o JavaScript. Um exemplo de servidor HTTP utilizando o Node.JS pode ser visto no Quadro 1. O Node.JS tem como característica a programação orientada a eventos, utilizando de I/O assíncrono para garantir que não tenha bloqueios em sua *thread*, pois todos os processos são executados utilizando um único processo, fazendo com que o consumo de memória alocado por requisição seja menor. Segundo Abernethy (2013) “o Node é extremamente bem projetado para situações em que um grande volume de tráfego é esperado e a lógica e o processamento necessários do

lado do servidor não são necessariamente volumosos antes de responder ao cliente". Mesmo utilizando linguagem de *script* para rodar no servidor a performance é maximizada utilizando a V8 JavaScript Engine, que de acordo com seu fabricante "compila e executa código fonte JavaScript, lida com a alocação de memória para objetos e limpa-os quando não são mais necessários." (V8, 2013).

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(80, "127.0.0.1");
console.log('Server running at http://127.0.0.1/');
```

Quadro 1 – Exemplo de um servidor HTTP em Node.JS (Acervo do Autor, 2014)

O Netty é um framework para o desenvolvimento de aplicações web utilizando o conceito de I/O não bloqueante em Java. "Rápido e fácil, sem sofrer com problemas de desempenho, o Netty foi projetado cuidadosamente com a experiência adquirida em diversos protocolos, como: FTP, HTTP, SMTP e diversos protocolos binários ou baseado em textos" (Netty, 2014).

2.4 Trabalhos Relacionados

Os principais trabalhos correlatos que serão úteis ao trabalho se referem a utilização de Banco de Dados NoSQL e seus diferencias, e também a otimização de Servidores Web.

Bancos de Dados NoSQL x SGBDs Relacionais: Análise Comparativa. Neste artigo é demonstrado as principais características dos bancos de dados NoSQL em relação aos bancos de dados relacionais. De acordo com Brito (2010) "O propósito, portanto, das soluções NoSQL não é substituir o Modelo Relacional como um todo, mas apenas em casos nos quais seja necessária uma maior flexibilidade da estruturação do banco." Em seu artigo, é feita uma análise dos prós e contras de cada tipo de banco com o intuito de esclarecer a aplicabilidade de cada um. Porém, ao se ressaltar que os bancos NoSQL possuem uma maior performance, não foi feito nenhum teste para comprovar tal afirmação.

Otimizando Servidores Web de Alta Demanda. De acordo com Hirata (2002) o "objetivo deste trabalho é levantar precisamente todos os fatores inerentes ao desempenho na Web sob o ponto de vista de software do servidor, analisá-los isoladamente e em suas interações, apontando possíveis soluções". O foco de sua dissertação é a de otimizar servidores web de diversas arquiteturas envolvendo diversas características, desde o sistema operacional, particularidades do protocolo HTTP, monitoramento do servidores e protocolo TCP/IP. Uma ausência na dissertação é que a mesma não cita otimizações no processo de Banco de Dados.

Performance Comparison Between Node.js and Java EE. Neste artigo é realizado um comparativo de performance entre o Node.JS e Java EE (utilizando o conceito de *Servlets*) utilizando o Banco de Dados CouchDB (que possui funcionalidades muitas parecidas com o MongoDB). Nos ambientes de testes, de acordo com Fasel (2013) "o Node.JS obteve um desempenho 20% superior a mesma implementação utilizando Servlet".

3. Definição da Arquitetura

A arquitetura proposta é bem simples e funcional, tendo como requisito para sua implementação os seguintes componentes:

- Servidor Web com tratamento de conexões através de I/O não bloqueante. Linguagem de programação compatível com os requisitos do servidor web e que compartilhe dos mesmos princípios de I/O (não bloqueante).

- Servidor de Banco de Dados NoSQL que seja suportado pela linguagem de programação.

- Aplicação Web sem grande grau de complexidade, com requisições rápidas ao servidor que não exijam um forte processamento de dados. Além disso, os requisitos da aplicação devem ser compatíveis com a ausência de algumas funcionalidades na camada do Banco de Dados (que podem variar de acordo com o banco escolhido), como por exemplo: ausência de controle transacional, falta de integridade referencial.

Uma característica do Node.JS é a utilização de uma única *thread* para atender a todas as requisições, juntamente com a utilização de I/O não bloqueante, esta é a característica que irá garantir com que muitas requisições possam ser atendidas simultaneamente.

> "Em vez de iniciar um novo encadeamento do SO para cada conexão (e alocar a memória correspondente com ele), cada conexão cria um processo, que não requer que o bloco de memória o acompanhe. O Node alega que nunca ocorrerá um impasse de bloqueios, pois não são permitidos bloqueios e ele não bloqueia diretamente para realizar chamadas de E/S" (ABERNETHY, 2013).

A utilização de uma única *thread* pode representar uma má utilização de hardware em ambientes que possuem um grande número de processadores. Este problema pode ser resolvido através da utilização do módulo Cluster1. Apesar de maximizar os recursos quando se tem mais de um processador este recursos não será utilizado no momento.

A maioria dos recursos do Node.JS dispõe dos conceitos de alta performance, não sendo diferente com o MongoDB, que possui um *driver* de conexão bastante otimizado para o Node.JS. Após alguns testes, a melhor forma de conexão com o banco de dados encontrada foi a de manter uma única conexão com banco de dados (aberta antes de iniciar o servidor), e utilizando a mesma conexão para todas as requisições HTTP a serem atendidas.

Esta maneira de conexão, só é eficiente graças ao uso de I/O não bloqueante também pelo *driver* de conexão, fazendo com que *callbacks* sejam invocados quando uma *query* possui retorno.

É importante ressaltar que com a utilização do módulo cluster para o Node.JS talvez esta não seja a melhor técnica de manipulação de conexão com o Banco de Dados.

Um outro aspecto que pode interferir em um grande número de conexões HTTP é a utilização do *keep-alive*, que faz com que a mesma conexão TCP-IP seja utilizada pelo cliente em requisições distintas. No Node.JS o *keep-alive* já vem habilitado por padrão.

4. Definição dos testes de desempenho

Quando se fala em desempenho de sistemas, diversos fatores devem ser levados em consideração, pois desempenho não se refere somente a velocidade do sistema, e sim a outros fatores como disponibilidade e confiabilidade.

O teste de desempenho é fundamental para definir a capacidade do servidor e aplicação, bem como definir os limites do sistema de acordo com o hardware disponível, "O objetivo é duplo: entender como o sistema responde a carregamento (isto é, número de usuários, número de transações ou volume global de dados), e coletar métricas que vão levar a modificações de projeto para melhorar o desempenho". (PRESSMAN, 2006)

[1] http://nodejs.org/api/cluster.html

Para que possa ser realizado testes de desempenho de uma arquitetura de serviços web é imprescindível que exista um software a ser testado. Para testar o desempenho da arquitetura proposta em relação as demais foi especificado uma aplicação simples denominada de "Encurtador de URL's". Esta aplicação, que não possui qualquer tipo de interface, tem como objetivo simular um serviço básico, com um banco de dados simples, que possui como requisito básico o grande número de acessos simultâneos ao serviço.

Para se ter uma base de dados com registros significativos, foi utilizado uma tabela com um milhão de links/URL's dos sites mais acessados segundo Alexa (2013), juntamente com um *hash* de 6 caracteres gerados de maneira única e aleatória. Um exemplo de como os dados estarão dispostos no banco de dados pode ser visto na Tabela 1.

Hash	*URL*
dktPz4	www.ford.com.br
vOB9ro	www.betweenflashes.es
lWgSrt	www.japonmoda.net
ShqKWQ	www.somag.tk
fBkcRw	www.focus.com
hVo72q	www.helgeland-arbeiderblad.no

Tabela 1 – Exemplo da tabela de endereços do software de exemplo (Acervo do Autor, 2014)

A aplicação cujo fluxo do comportamento esperado pode ser visualizado na Figura 1, basicamente irá funcionar seguindo os passos:

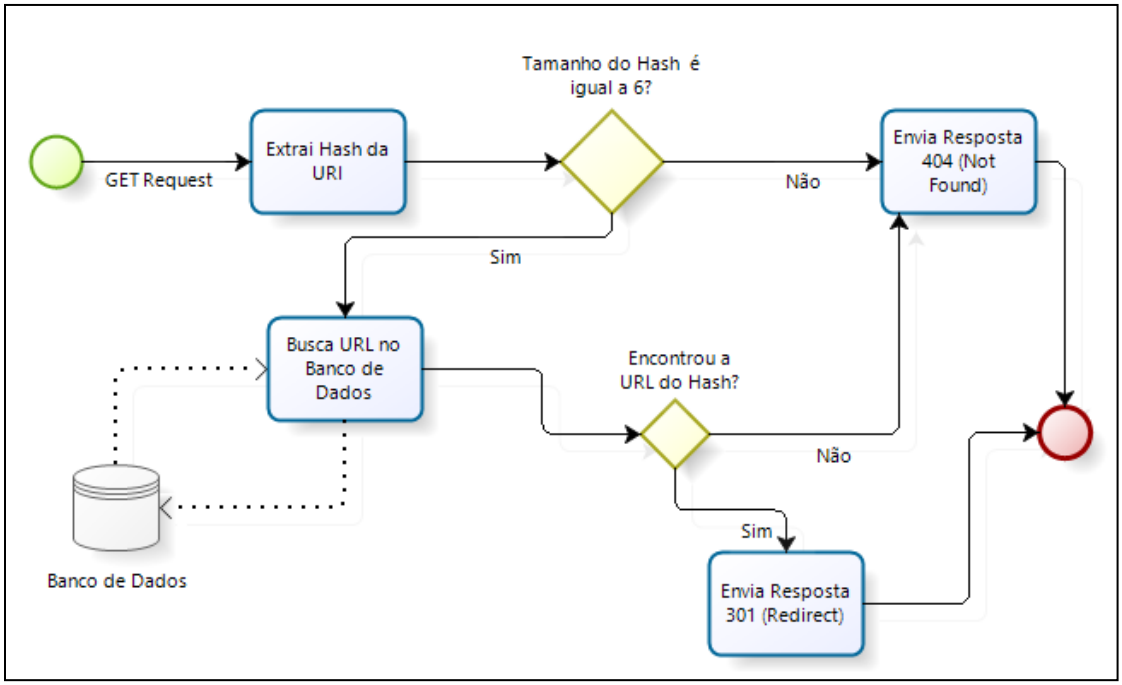

Figura 1 – Fluxo do Algoritmo da aplicação encurtador de URL's (Acervo do Autor, 2014)

1. Verifica se foi informado um *hash* válido como parâmetro da requisição (string com 6 caracteres), caso o *hash* seja inválido é enviado um código de resposta 404 (objeto não encontrado).
2. Busca no banco de dados a URL encontrada na primeira etapa, caso a mesma não tenha sido encontrada é enviado um código de resposta 404 (objeto não encontrado).
3. Envia um código de resposta 301 (objeto movido permanentemente), juntamente com o cabeçalho *Location*, indicando a URL para qual o objeto foi movido.

Este simples algoritmo possibilita com que a aplicação seja portada facilmente para outras arquiteturas utilizando funções básicas de servidores HTTP.

4.1 Desenvolvimento da Aplicação em Node.JS

O desenvolvimento basicamente seguiu o que foi definido na arquitetura proposta, em relação a utilização do MongoDB juntamente com o Node.JS alguns pontos que podem ser observados no código-fonte da aplicação no Quadro 2 merecem destaque: pelo fato de o servidor atender a todas as requisições em uma única Thread obtêm-se um melhor desempenho utilizando uma única conexão com o Banco de Dados (linhas 3 e 4). É solicitada a conexão ao servidor e registrado um *callback* que quando conectado, irá iniciar a execução do servidor (linha 5 a linha 28). Cada requisição utiliza a mesma conexão, isto é possível graças ao desenvolvimento orientado a eventos, pois cada *query* registra outro *callback* (linha 10) a ser executado quando a mesma estiver pronta, o que permite a execução de várias *queries* sem que as outras tenham terminado.

```
01 var http = require("http");
02 var url = require("url");
03 var MongoClient = require('mongodb').MongoClient;
04 MongoClient.connect("mongodb://127.0.0.1:27017/monografia",
05 function(err, db) {
06    http.createServer(function(request, response) {
07        var path = url.parse(request.url, true).path;
08        if(7==path.length) {
09            var col = db.collection('url');
10            col.findOne({hash:path},function(err,result) {
11               if(!err && result) {
12                    var sUrl = 'http://'+result['url'];
13                    response.writeHead (301, {'Location': sUrl});
14                }
15                else {
16                    response.writeHead(404);
17                }
18                db.close();
19                response.end();
20            });
21        }
22        else {
23            response.writeHead(404);
24            response.end();
25        }
26    }).listen(80);
27    console.log("Server Running 80...");
28 });
```

Quadro 2 – Encurtador de URL's implementado em Node.JS com MongoDB (Acervo do Autor, 2014)

Outro aspecto importante é que o Node.JS não é utilizado somente para a construção de servidores HTTP, sendo assim durante o desenvolvimento é necessário que se tenha bons conhecimentos do funcionamento do protocolo HTTP (o que em outros ambientes de desenvolvimento é mais facilitado). Alguns exemplos de "burocracia" que o desenvolvedor tem que fazer é sempre fechar explicitamente a resposta ao cliente, tratar nativamente a url e para qual recurso ela vai levar, dentre outros. Já existem algumas bibliotecas que podem ser incorporadas ao servidor que facilitam o desenvolvimento, e por se tratar de uma plataforma relativamente nova, a tendência é que futuramente novas facilidades sejam adicionadas.

4.2 Desenvolvimento da Aplicação em outros ambientes

Com o objetivo de comparar os resultados a serem obtidos com a solução do Node.JS com MongoDB optou-se por desenvolver o mesmo algoritmo utilizando outras linguagens de programação/servidor, bem como a utilização de outro Banco de Dados. Com estas alterações espera-se nos resultados comparativos poder classificar melhor qual a importância de cada um na performance da aplicação. A lista contendo os servidores e linguagens de programação escolhidos é encontrada na Tabela 2.

Servidor	*Linguagem de Programação*	*Banco de Dados*
Node.Js	JavaScript	MongoDB
Node.Js	JavaScript	PostgreSQL
Netty	Java	MongoDB
Netty	Java	PostgreSQL
Apache	PHP	MongoDB
Apache	PHP	PostgreSQL

Tabela 2 – Ambientes onde os testes serão executados e comparados (Acervo do Autor, 2014)

Todas as aplicações discriminadas na Tabela 2, foram desenvolvidas da maneira mais simplificada possível, levando em consideração o algoritmo do software na Figura 1. Para conexão com o Banco de Dados, os *drivers* nativos fornecido pelo próprio fabricante do banco foram utilizados. A aplicação de comparação com o desenvolvimento mais simples foi em PHP, resultando em um código-fonte em torno de 15 linhas, que pode ser visualizado no Quadro 3.

```php
<?php
$query = $_SERVER['PATH_INFO'];
if (7 == strlen($query)) {
    $query = substr($query, 1);
    $mongo = new Mongo('mongodb://127.0.0.1:27017');
    $res = $mongo->monografia->url->findOne(Array('hash' => $query));
    if ($res) {
        $redirect = 'http://' . $res['url'];
        header('Location: ' . $redirect, true, 301);
    } else {
        header('HTTP/1.0 404 Not Found',true,404);
    }
} else {
    header('HTTP/1.0 404 Not Found',true,404);
}
?>
```

Quadro 3 – Encurtador de URL's implementado em PHP com MongoDB (Acervo do Autor, 2014)

O desenvolvimento utilizando o framework Netty em Java foi o de maior complexidade. Pelo fato de ser uma plataforma mais robusta, e de o framework utilizado não ser voltado somente para aplicações HTTP. O desenvolvimento em Java exigiu um maior esforço de programação, sendo necessário a criação de algumas classes (fora do contexto da aplicação encurtador de URL's) para realizar o tratamento das conexões para o protocolo HTTP. O principal trecho de código da aplicação em Java pode ser visto no Quadro 4.

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        if (msg instanceof HttpRequest) {
            HttpRequest req = (HttpRequest) msg;
            String uri = req.getUri();
            FullHttpResponse response = null;
            if (7 == uri.length()) {
                uri = uri.substring(1);
                DB db = getClient().getDB("monografia");
                db.requestStart();
                db.requestEnsureConnection();
                DBCollection url = db.getCollection("url");
                //BasicDBObject query = new BasicDBObject("hash", uri);
                DBObject resul = url.findOne(new BasicDBObject("hash", uri));
                if (resul != null) {
                    String urlRedirect = "http://" + resul.get("url");
                    response = new DefaultFullHttpResponse(HTTP_1_1, MOVED_PERMANENTLY);
                    response.headers().set(LOCATION, urlRedirect);
                } else {
                    response = new DefaultFullHttpResponse(HTTP_1_1, NOT_FOUND);
                }
                db.requestDone();
            } else {
                response = new DefaultFullHttpResponse(HTTP_1_1, NOT_FOUND);
            }
            boolean keepAlive = isKeepAlive(req);
            response.headers().set(CONTENT_LENGTH, response.content().readableBytes());
            if (!keepAlive) {
                ctx.write(response).addListener(ChannelFutureListener.CLOSE);
            } else {
                response.headers().set(CONNECTION, Values.KEEP_ALIVE);
                ctx.write(response);
            }
        }
    }
```

Quadro 4 – Encurtador de URL's implementado em Netty com MongoDB (Acervo do Autor, 2014)

Para garantir que todas as aplicações tenham o mesmo comportamento foi criado uma aplicação de validação, com uma base de amostras contendo dois mil *hash's* (tanto *hash's* válidos como inválidos) fazendo a mesma requisição HTTP para cada uma das 6 arquiteturas especificadas. Os cabeçalhos de respostas são comparados e o teste tem sucesso quando são iguais em todas as arquiteturas.

Durante a execução da aplicação de validação das arquiteturas, pequenas inconformidades foram encontradas nas diversas aplicações, após a correção das mesmas os testes foram novamente executados até que todas as aplicações tivessem atingidos os mesmos resultados. Um fluxo do algoritmo de teste pode ser visualizado na Figura 2.

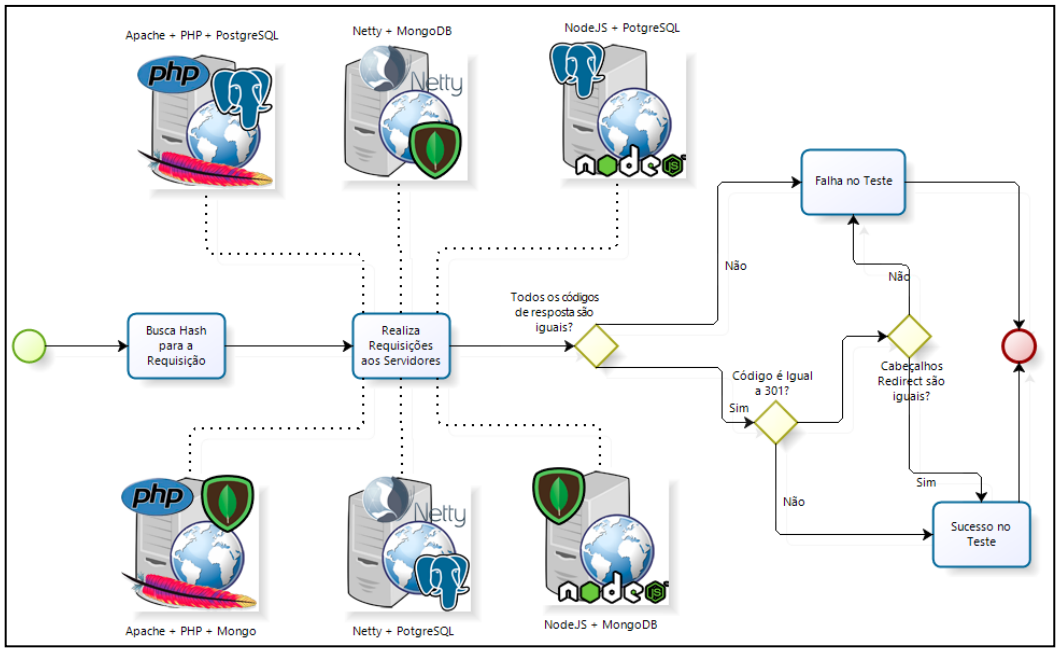

Figura 2 – Fluxo do Algoritmo da aplicação de validação das arquiteturas (Acervo do Autor, 2014)

Os códigos-fontes de todas as aplicações, inclusive a de validação da aplicação está disponível em um repositório no Google Code[2].

5. Análise de Resultados

A execução de testes de desempenho requer a utilização de ferramentas para este propósito. Para a execução dos testes a ferramenta escolhida foi o JMeter, que é "uma aplicação desktop, open-source, 100% desenvolvida em Java, projetada para a execução de testes funcionais e medição de desempenho" (Apache JMeter, 2014).

Os testes foram executados isoladamente no seguinte ambiente virtualizado: Sistema Operacional CentOS 6.2 x86, 1Gb de memória RAM, 2 núcleos de 2,4Ghz Intel Core I5.

Os softwares que foram avaliados e suas respectivas versões são as seguintes:

- Node.JS v.0.10.4
- MongoDB v.2.5
- Apache 2.2.15 + Módulo PHP 5.3.3
- PostgreSQL 9.2.2

Todos os aplicativos foram baixados de seus repositórios oficiais, e realizado as instalações padrão, não alterando nenhum tipo de configuração (com exceção de portas) após a instalação.

Para execução dos testes foi definido o seguinte cenário: quarenta *threads* (usuários), inicializando incrementalmente durante cinco segundos, ficando todos ativos durante quarenta e cinco segundos, reduzindo os mesmos durante os últimos dez segundos, totalizando sessenta segundos de teste. Este número de usuários foi escolhido após diversos testes e por identificar que um número superior ocasionava muitas requisições não atendidas, e um número inferior ocasionava hardware ocioso.

Cada usuário contém uma lista com 2007 (valor escolhido de maneira arbitrária) hash's (dentre válidos e inválidos), onde as requisições são feitas de maneira seqüencial Caso um usuário tenha feito todas as suas requisições e o teste ainda não tenha chegado ao fim, sua lista é iniciada novamente, garantindo com que todos os usuários fiquem ativos independente do

[2] Repositório disponível na URL https://code.google.com/p/monografia-node/

número de requisições que fizeram. A freqüência de usuários em relação ao tempo pode ser visualizada na Figura 3.

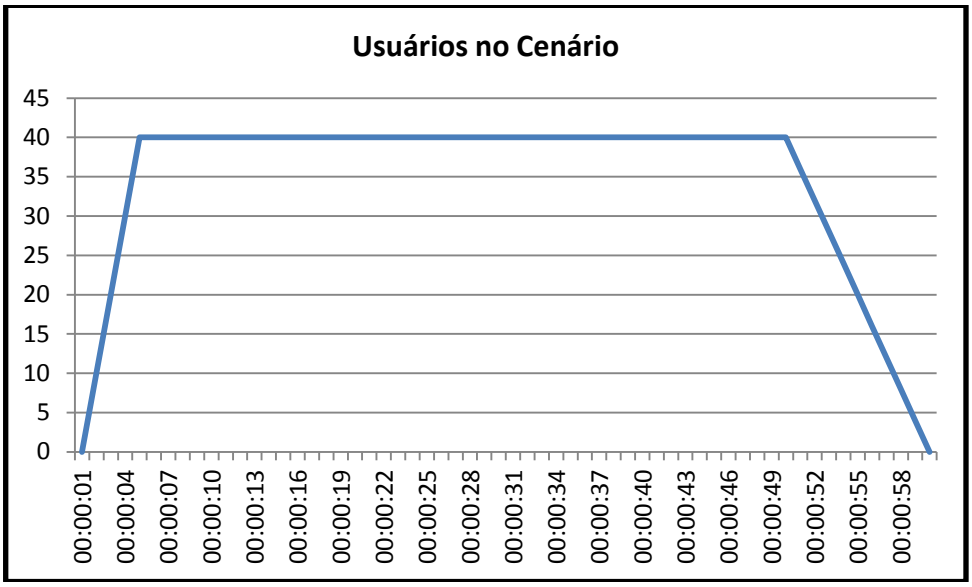

Figura 3 – Número de Usuários x Tempo no ambiente de teste (Acervo do Autor, 2014)

Durante a execução dos testes as seguintes métricas foram coletadas para posterior análise: número de requisições, número de requisições com erro, tempo de resposta médio, vazão, requisições por tempo, memória e CPU.

Os resultados obtidos durante a execução do teste serão expostos a seguir. É importante ressaltar que os dados contidos nos gráficos devem sempre ser analisados em conjunto e não isoladamente, pois se tratam de informações complementares.

5.1 Consumo de Memória

A memória RAM é um dos principais pontos de análise em um servidor web, pois esta ligada diretamente ao número de requisições que o servidor é capaz de atender em determinado tempo. O Gráfico contendo o comparativo de memória pode ser visto na Figura 4.

Durante a execução dos testes, praticamente todos os ambientes analisados tiveram o uso de memória constante e estável, porém, os dois ambientes utilizando Apache e PHP foram os que tiveram o maior consumo de memória, ocupando em média 40% a mais de memória que os outros ambientes. A arquitetura do Node.JS/MongoDB foi a que obteve o segundo melhor índice de consumo de memória, ficando atrás somente do Java/Netty/PostgreSQL.

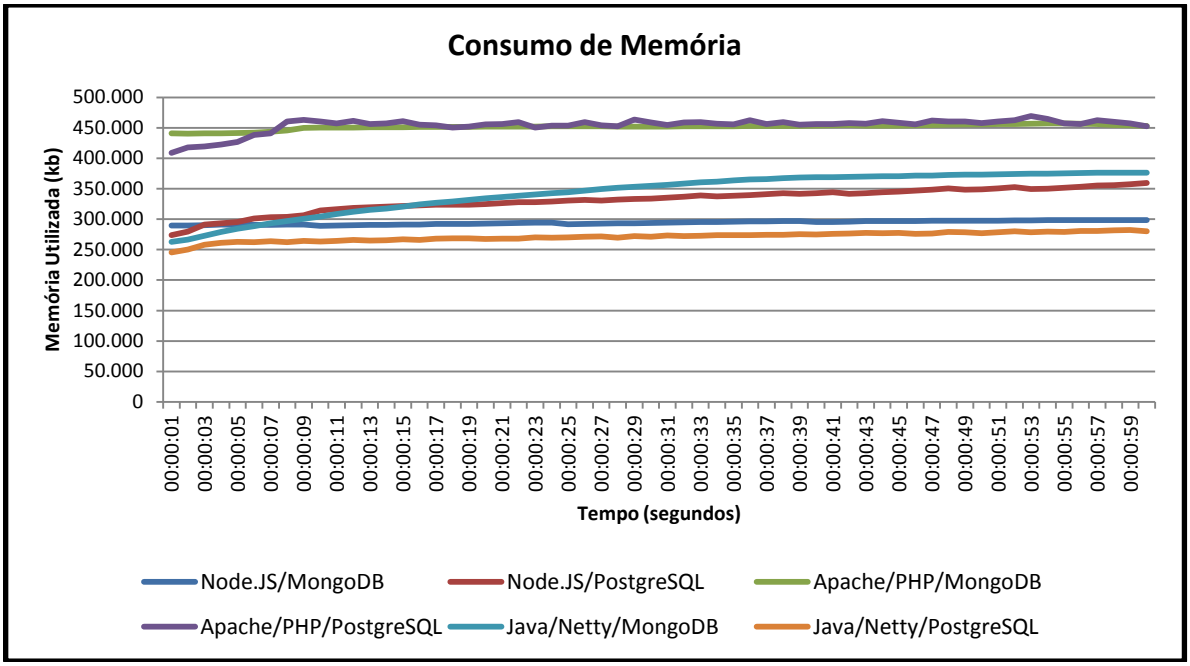

Figura 4 – Consumo de memória RAM durante a execução dos testes (Acervo do Autor, 2014)

5.2 Consumo de CPU

A utilização de CPU foi a métrica de hardware que mais sofreu variação, chegando a um aumento superior a 100% dependendo do ambiente a ser comparado. Um dado importante que pode ser analisado é que os dois piores ambientes (que mais utilizaram recursos da CPU) utilizavam o banco de dados PostgreSQL: Java/Netty e Apache/PHP.

Outra análise que pode ser feita é que o ambiente Java/Netty/PostgreSQL foi o que menos havia consumido memória, porém, foi o que mais teve uso de CPU.

A arquitetura do Node.JS/MongoDB foi a que obteve o melhor desempenho em utilização de CPU, mantendo uma média de uso em torno de 40%. O gráfico contendo o comparativo de uso da CPU pode ser visto na Figura 5.

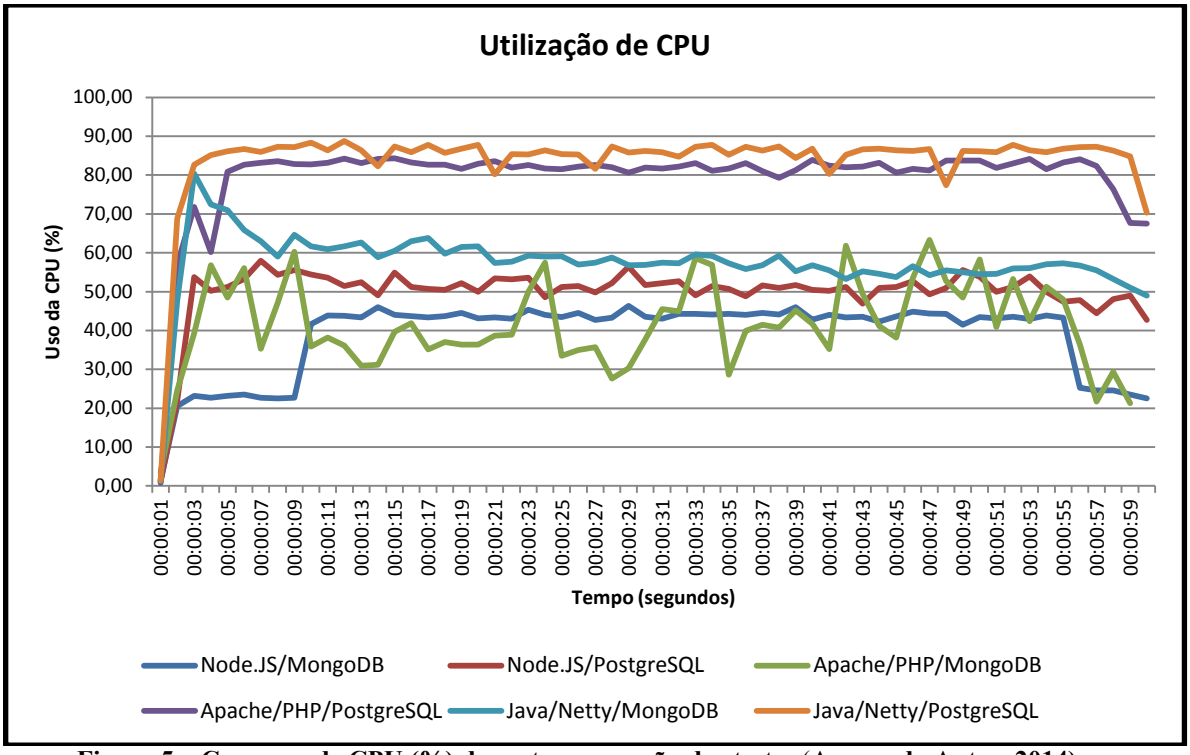

Figura 5 – Consumo de CPU (%) durante a execução dos testes (Acervo do Autor, 2014)

5.3 Requisições por tempo

Um dos principais fatores a serem analisados no teste é o número de requisições que o servidor foi capaz de atender durante determinado período de tempo. Este número basicamente indica o quanto de usuários (requisições) o servidor é capaz de absorver. A Figura 6 apresenta um dado importante: os dois ambientes que mais responderam requisições (Node.JS/MongoDB e Netty/MongoDB) utilizam o mesmo banco de dados: MongoDB, considerado um banco de alto desempenho. Vale ressaltar também, que ambos são servidores de alto desempenho, que tem como característica básica atender o maior número de requisições simultâneas.

A arquitetura proposta obteve o melhor índice de requisições, obtendo em grande parte do teste um número superior a 1400 requisições por segundo. O pior desempenho ficou com as arquiteturas: Apache/PHP/PostgreSQL e Java/Netty/PostgreSQL, tendo uma média de 200 requisições por segundo.

É possível notar que neste ambiente de teste o banco de dados MongoDB obteve performance muito superior em todos os ambientes do que os que utilizavam PostgreSQL. Também fica visível a grande diferença entre as plataformas, chegando a ter um aumento de praticamente 500% no número de requisições utilizando basicamente os mesmos recursos de hardware.

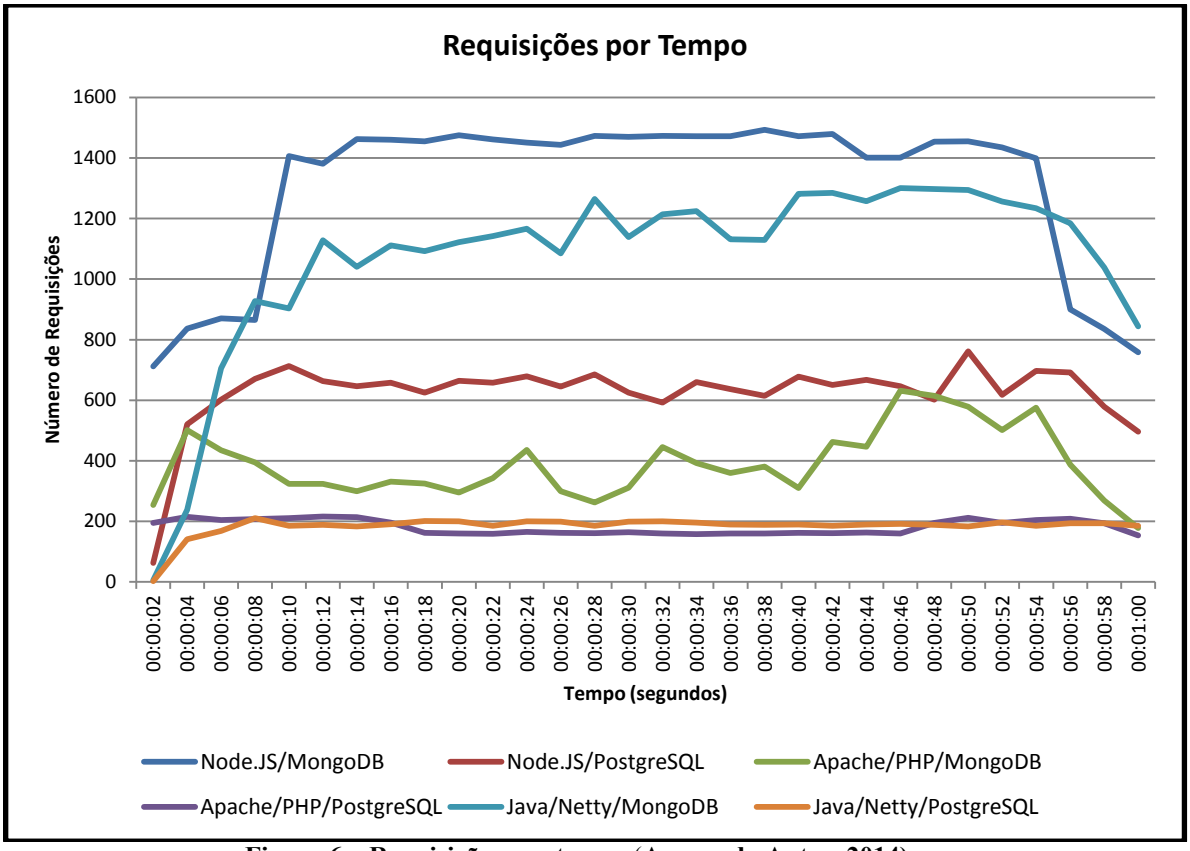

Figura 6 – Requisições por tempo (Acervo do Autor, 2014)

5.4 Número de Requisições

O número de requisições efetuadas durante o período total do teste (são computadas nesta métrica tanto amostras com sucesso, quanto amostras inválidas), disponível na Figura 7, basicamente acompanha a informação de requisições por tempo, pois não houve alterações (tanto para mais quanto para menos) significativas no número de requisições em relação ao tempo, tendo se mantido estável. Assim a arquitetura Node.JS/MongoDB foi a que conseguiu atender o maior número de requisições, chegando perto de 80mil requisições, um número 8 vezes superior as de pior desempenho: Apache/PHP/PostgreSQL e Java/Netty/PostgreSQL com 10 mil cada.

Um fato que merece destaque, é que somente um ambiente apresentou uma taxa considerável de requisições com erro (foram considerados erros qualquer cabeçalho de requisição diferente de 301 e 404, bem como problemas relacionados à falha de conexão), em torno de 4%, no ambiente Apache/PHP/MongoDB e os erros apresentados não se referiam a falha de conexão e sim de falha do ambiente no gerenciamento de memória.

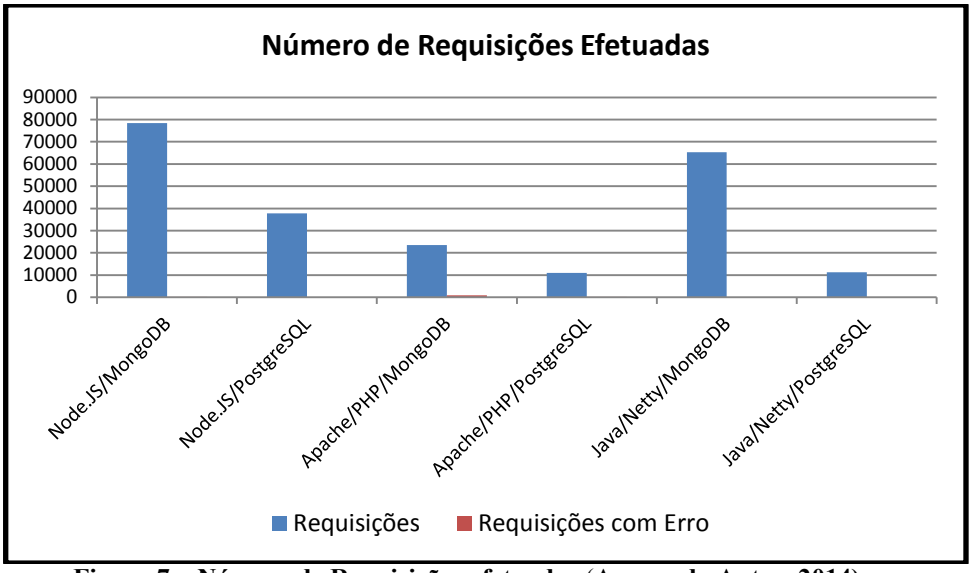

Figura 7 – Número de Requisições efetuadas (Acervo do Autor, 2014)

5.5 Tempo médio de resposta

Esta métrica é uma das mais importantes para o usuário do sistema, pois basicamente define o tempo que o servidor irá demorar para atender a requisição feita pela usuário (sem levar em consideração velocidade da rede de conexão). Basicamente o tempo médio de resposta de todas as requisições acompanhou o número total de requisições efetuadas. Isto se deve ao fato de quanto mais rápido o servidor responder, mais tempo livre ele tem para atender novas requisições, elevando assim o número de requisições que podem ser efetuadas em relação ao tempo de resposta das mesmas.

Novamente a arquitetura Node.JS/MongoDB foi a que obteve o melhor desempenho, tendo um tempo médio de resposta de 27 milissegundos, valor muito abaixo do obtido pelo Apache/PHP/PostgreSQL que teve o pior desempenho, com um tempo médio de resposta de 190 milissegundos. O gráfico contendo os comparativos de tempo médio de resposta pode ser visto na Figura 8.

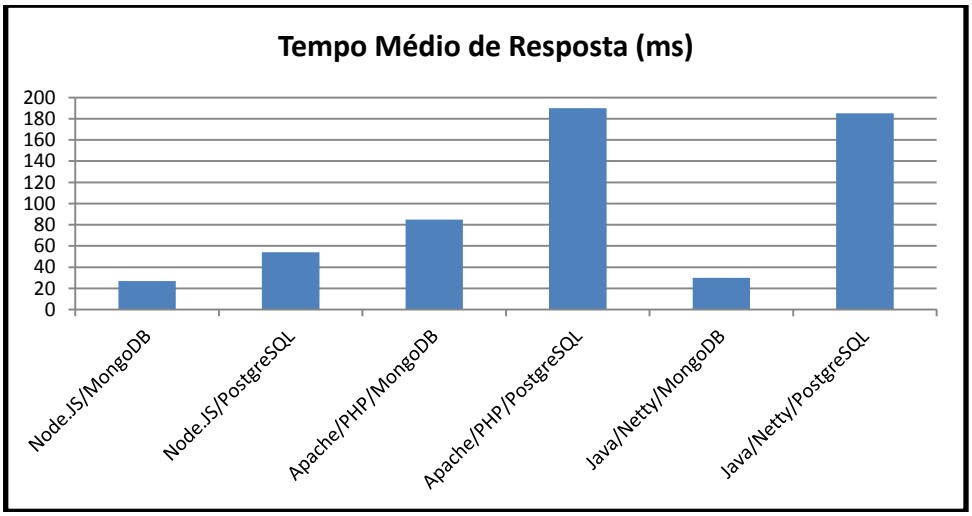

Figura 8 – Tempo médio de resposta em milissegundos (Acervo do Autor, 2014)

5.6 Vazão do servidor

Novamente a vazão dos servidores, que é o número de operações que o servidor é capaz de processar em determinado período de tempo (sendo calculado pelo número total de requisições efetuadas divido pelo período de execução do teste) vista na Figura 9 acompanha os gráficos de tempo médio de resposta e de número de requisições efetuadas. Destaque para as plataformas Node.JS/MongoDB (que obteve o melhor desempenho) e Java/Netty/MongoDB que obtiveram uma taxa de vazão superior a 1000 requisições por segundo, comprovando a performance destas plataformas. O pior desempenho ficou com o Apache/PHP/PostgreSQL e Java/Netty/PostgreSQL, com taxas de vazão inferior a 200 requisições por segundo.

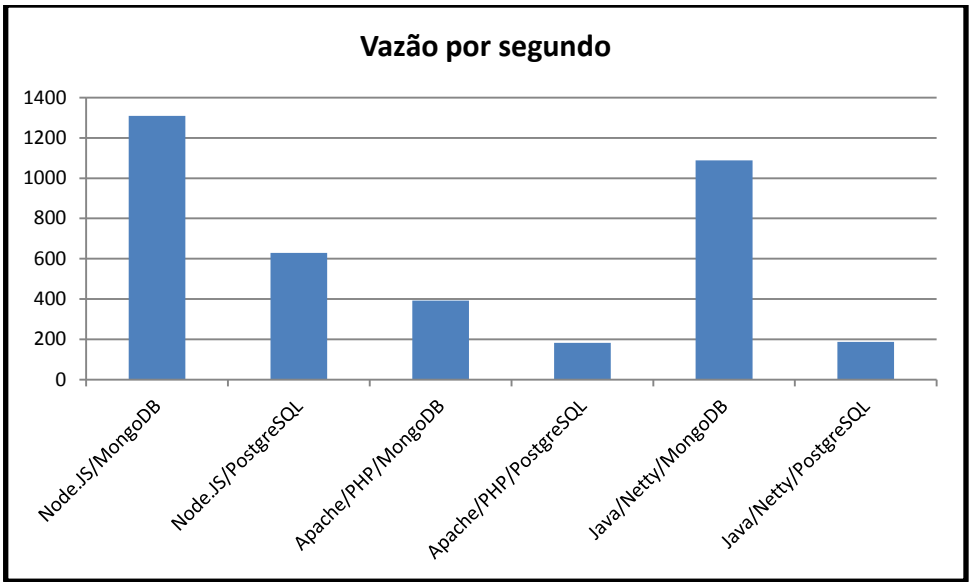

Figura 9 – Vazão dos servidores por segundo (Acervo do Autor, 2014)

De maneira geral, a arquitetura proposta obteve os melhores desempenhos em todos os critérios, com exceção do uso de memória. Este fato é justificável devido ao maior número de requisições que a mesma conseguiu atender no período de execução do teste.

A arquitetura utilizando Java/Netty/MongoDB obteve resultados próximos aos do Node.JS/MongoDB. A utilização de Apache/PHP/PostgreSQL foi a que obteve o pior desempenho de maneira geral.

6. Considerações Finais

A implementação e a avaliação da arquitetura proposta utilizando Node.JS e MongoDB foi realizada com sucesso, tendo resultados satisfatórios, dentro do esperado.

Analisando os resultados obtidos com a aplicação que foi especificada pode-se notar que o Node.JS e o MongoDB realmente são servidores de alto desempenho, tendo uma performance em alguns aspectos 6 vezes superior quando comparado as outras arquiteturas abordadas no teste. Isto se deve principalmente ao fato de terem sido construídos com este propósito, deixando de lado recursos que podem ser considerados importantes para maximização da performance. Nota-se que o desenvolvimento em Java, utilizando Netty possui um desempenho praticamente semelhante ao obtido com o Node.JS. Isto se deve principalmente pela utilização de I/O não bloqueante em sua construção.

A escolha de um Banco de Dados compatível com os requisitos da aplicação também tem um papel muito importante na arquitetura. Notou-se que a utilização do MongoDB por si só já é

responsável por grande parte do ganho de performance obtido e outros bancos de dados NoSQL podem ter uma performance até superior.

Apesar de recentes, estes servidores já são largamente utilizados no mercado. De acordo com Node.JS (2013) o mesmo é utilizado por "grandes empresas como: Linked-in, e-bay, Microsoft, PayPal, etc".

Além do Node.JS e do Netty, existem no mercado diversos servidores que compartilham da mesma característica, dentre os quais podemos destacar: Jetty[3] (Java), Vertex.IO[4] (multi-plataforma), Grizzly[5]. Até mesmo servidores mais tradicionais, como o Apache Tomcat, oferecem suporte a conectores não bloqueantes, que pode maximizar o desempenho do mesmo.

É importante destacar que estes servidores muitas vezes não substituem os tradicionais do mercado, pois possuem algumas características distintas, sendo necessário um estudo com muita cautela de onde os mesmos podem ser aplicados dentro das organizações. Um exemplo são os banco de dados NoSQL que em sua grande maioria não possuem recursos como controle de transação, garantia de integridade, etc, características estas que podem ser fundamentais em muitas aplicações. O fato dos ambientes serem relativamente novos, sem muitos recursos presentes nas plataformas de desenvolvimento mais robustas, faz com que o desenvolvimento de aplicações com muitos requisitos praticamente não tenha sido efetuado, se limitando ao desenvolvimento de pequenas aplicações/serviços.

A arquitetura proposta pode muitas vezes ser utilizada em conjunto com uma arquitetura já existente e tradicional, podendo servir como "válvula de escape" para aqueles requisitos que são responsáveis pelos "gargalos" do sistema. Esta solução hibrida pode inclusive ser implementada parcialmente, podendo utilizar somente o servidor web ou somente o banco de dados, cabendo um teste afim de viabilizar o uso de tal solução de acordo com o ambiente aonde a mesma poderá ser utilizada.

A sugestão para continuidade do trabalho é o desenvolvimento de algoritmos/aplicações de grande complexidade em ambientes de alto desempenho, isto irá permitir uma análise de performance da arquitetura em ambientes mais complexos. Outra continuidade a ser realizada é separação dos testes de Banco de Dados e de Servidor Web afim de ponderar qual a performance isolada de cada um.

Como portar uma aplicação já existente para outra arquitetura muitas vezes irá envolver o re-desenvolvimento dos códigos-fontes já existentes, sugere-se também uma pesquisa abordando a melhoria de performance em arquiteturas tradicionais como: Tomcat/Java/Servlet e Apache/PHP.

[3] http://www.eclipse.org/jetty/
[4] http://vertx.io/
[5] https://grizzly.java.net/

Referências

ABERNETHY, Michael. **O que exatamente é o Node.js?** 2013. Disponível em <http://www.ibm.com/developerworks/br/library/os-nodejs/>. Acesso em: 11 nov. 2013.

ALEXA, **Top 500 Global Sites**. 2013. Disponível em < http://www.alexa.com/topsites>. Acesso em: 11 nov 2013

BALTZAN, Paige; PHILLIPS, Amy. **Sistemas de Informação**. Porto Alegre: Bookman, 2012.

BRITO, Ricardo W. **Bancos de Dados NoSQL x SGBDs Relacionais:** Análise Comparativa. 2010. Disponível em <http://www.infobrasil.inf.br/userfiles/27-05-S4-1-68840-Bancos%20de%20Dados%20NoSQL.pdf>. Acesso em: 01 jun. 2013.

FASEL, Marc. **Performance Comparison Between Node.js and Java EE**. 2013. Disponível em < http://java.dzone.com/articles/performance-comparison-between>. Acesso em 01 nov. 2013.

FOWLER, Martin. **Padrões de Arquitetura de Aplicações Corporativas**. Porto Alegre: Bookman, 2003.

HIRATA, Renato. **Otimizando Servidores Web de Alta Demanda**. 2002. Disponível em <http://www.las.ic.unicamp.br/paulo/teses/20020411-MSc-Renato.Hirata-Otimizando.servidores.Web.de.alta.demanda.pdf>. Acesso em: 05 jun. 2013.

JMETER, **Apache JMeter**. 2013. Disponível em < http://jmeter.apache.org/>. Acesso em: 20 dez. 2013.

MONGODB. **The MongoDB Manual**. 2013. Disponível em <http://docs.mongodb.org/manual/>. Acesso em: 05 jun. 2013.

NETTY. **Java NIO Framework**. 2013. Disponível em < http://netty.io/>. Acesso em: 19 dez. 2013.

NODEJS. **Evented I/O for V8 JavaScript** 2013. Disponível em < http://nodejs.org/>. Acesso em: 05 jun. 2013.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Makron Books, 6. Ed. 2006: McGraw-Hill 2006.

SADALAGE, Pramod J; FOWLER, Martin. **NoSQL distilled**: a brief guide to the emerging world of polyglot persistence. Upper Saddle River,USA: Pearson Education, 2013.

STEIN, Lincoln D. **Network Programming with Perl**. USA: Addison Wesley, 2001.

V8. **JavaScript Engine**. 2013. Disponível em < https://code.google.com/p/v8/>. Acesso em: 05 jun. 2013.