UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE EDUCAÇÃO CONTINUADA

ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Rinaldo Darski

Busca de Indicadores de Eficiência no Uso de Recursos em Aplicações de Transmissão por Upload em um Estudo Comparativo de um Servidor Node.js para Middleware versos um Apache Tomcat

São Leopoldo

2016

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE EDUCAÇÃO CONTINUADA

ESPECIALIZAÇÃO EM QUALIDADE DE SOFTWARE

Rinaldo Darski

Busca de Indicadores de Eficiência no Uso de Recursos em Aplicações de Transmissão por Upload em um Estudo Comparativo de um Servidor Node.js para Middleware versos um Apache Tomcat

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Qualidade de Software, pelo curso de Pós-Graduação Lato Sensu em Qualidade de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Dra Margrit Reni Krug

São Leopoldo

2016

Agradeço a minha esposa e a meus dois filhos pelas horas que abriram mão da minha presença, e a paciência que tiveram comigo, para que eu consegui-se produzir esse texto.

Busca de Indicadores de Eficiência no Uso de Recursos em Aplicações de Transmissão por Upload em um Estudo Comparativo de um Servidor Node.js para Middleware versos um Apache Tomcat

Rinaldo Darski

Unidade Acadêmica de Educação Continuada – Universidade do Vale do Rio dos Sinos (Unisinos)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

Riddrk6@gmail.com

Resumo. *Notícias na imprensa de sistemas lentos são frequentes. Uma das razões é o crescente volume de dados nos últimos anos. É comum o envio de arquivos, fotos e vídeos por upload, entretanto diferentes tecnologias tendem a oferecer desempenhos não iguais. Esse estudo simula dez cenários e indica entre duas ferramentas, Node.js e o Apache Tomcat, qual apresenta melhores medidas de latência, variabilidade de latência e eficiência de CPU, se usado como Middleware para upload. Descrevem-se essas medidas e propõe como métrica o monitoramento da comutação de processos na CPU como forma de acompanhar a variabilidade dessa latência.*

Abstract. *Often we hear news of slow systems. One reason is that the generation of data is higher now. Nowadays it is common the user to post files, photos and videos, but different technologies not always deliver the same performance. This research simulates ten scenarios and points out between two tools, Node.js and Apache Tomcat. Results of this research show which has best measures latency, latency variability and efficiency of use CPU, if used as Middleware to upload. It is reported here this measures and it indicate as metric, to monitor the latency variability through the switching processes in the CPU.*

1. Introdução

Notícias de sistemas lentos são frequentes e frases como “a internet está lenta”, “o servidor está lento” são comuns no meio empresarial. Muitas vezes essas situações são aleatórias e não explicadas. Muitos usuários de sistemas possuem a percepção de que em determinado momento um software está funcionando bem, em outro o mesmo responde de forma muito mais lenta.

Uma das causas conhecidas desses problemas é o aumento de carga em determinados momentos em um servidor, normalmente relacionada ao volume de dados (Tian 2015) e não necessariamente a uma maior quantidade de usuários.

Nos últimos anos houve um crescimento muito maior do volume de dados gerados pelos usuários do que o crescimento da quantidade de usuários na rede. A IBM afirma que 90% dos dados globais foram criados nos últimos dois anos [IBM, 2016], outras organizações fazem projeções se não tão grandes também alarmantes [Chede 2012, Cisco 2015].

Atualmente aplicativos são baixados e usados simultaneamente por milhões de usuários, gerando uma crescente demanda na estrutura de servidores para suportar essas requisições. Qualquer sistema que necessite suportar um grande volume, deve conter no seu projeto medidas bem definidas de testes de desempenho [Pressman 2000], tendo em vista a garantia da qualidade.

Grande parte dos problemas de desempenho tem sua origem em servidores de aplicação, 40% no AS (*Application Server*) e 10% Web Server [Molinari 2009], portanto o aprimoramento de soluções que minimizem esses riscos de restrição operacional é importante para redução de incidentes nesses servidores.

Há várias tecnologias de desenvolvimento para servidores, sendo atualmente as mais comuns PHP e Java. As diferentes necessidades de hardware que essas tecnologias demandam criam a necessidade de avaliações previas do seu potencial consumo. Nesse aspecto de demanda prevista, um dos tipos de teste possíveis é o teste de carga [Myers 1979].

A variabilidade da entrega ao cliente final é um problema para os profissionais que monitoram constantemente o desempenho de servidores. Muitas vezes esses problemas de desempenho, para muitas tecnologias, são tratados pela aquisição de hardware, fato que nem sempre é uma solução mais eficiente e pode onerar projetos de forma desnecessária. Nesse aspecto, hoje existem outras tecnologias que vêm instrumentalizando desenvolvedores é o caso de servidores Node.js, que em outros estudos demonstram maior eficiência no uso de hardware [Mccune 2012].

Observa-se, de forma geral, que o recurso de *upload* de arquivos tornou-se comum em web sites. Com base na analise de carga desse tipo de recurso, relata-se aqui o comportamento de dois servidores submetidos a esse experimento: um servidor Node.js e um Servidor Java Tomcat. O objetivo deste projeto é mostrar qual deles apresenta um melhor comportamento em relação a essa variabilidade de serviço em determinado ambiente. Entenda-se comportamento como dois aspectos principais, a variação no tempo de resposta que os clientes efetivamente recebem as suas requisições, bem como uma menor carga de CPU no servidor para fazer a mesma atividade em ambas às tecnologias em situação de baixa restrição de IO (entrada e saída).

Considerando uma máquina específica, detalhada no item 4.1 (Ambiente de Teste e Limitações), as principais questões de pesquisa concentram-se em responder algumas questões, tais como:

a. Qual a tecnologia apresenta menor coeficiente de variabilidade no tempo de entrega em um ambiente de baixa restrição de IO?
b. É possível constatar em algum cenário que um investimento em disco SSD não é o suficiente para superar o tempo médio de resposta de outra tecnologia, mesmo utilizando um disco mecânico tradicional e tendo maior restrição de IO?

c. É possível a localização de indicadores de eficiência do uso de recursos os quais possam ser usados para acompanhar a variação dos dados obtidos do servidor?

O objetivo da busca por indicadores é contribuir com gestores de TI para ajudar a reduzir incidentes de desempenho, ao mesmo tempo em que se busca melhorar a previsibilidade dessas ocorrências com um melhor gerenciamento dos recursos.

Para alcançar os objetivos deste projeto, foram necessários os seguintes objetivos específicos:

a. Descrever em um conjunto de cenários a variabilidade do tempo de resposta, o tempo médio de respostas, quantidade de erros, uso de CPU em cada um dos testes;

b. Determinar qual das duas tecnologias é melhor cenário dos testes de carga. Entende-se melhor, como aquele que realizou a mesma atividade em menos tempo;

c. Em um grupo das amostras, averiguar se um disco SSD é suficiente para manter o tempo médio de resposta igual ou superior a outra tecnologia sem SSD;

d. Descrever qualitativamente indicadores, no lado do servidor, que possam melhorar a previsibilidade deste tipo de situação de variabilidade de latência com a carga de dados por *upload*.

2. Método de Pesquisa

Tradicionalmente testes de desempenho são pesquisas do tipo quantitativas [Molinari 2009]. Wainer (2007) classifica três grupos de *benchmarks* como típicas pesquisas quantitativas em ciência da computação, dentre esses três estão às avaliações de tempo de execução de programas.

Esse tipo de pesquisa caracteriza-se por dados objetivos e por avaliações de resultados estatísticos que tendem a serem mais ricos. Pode o pesquisador fazer uso de simuladores para verificação de modelos e obtenção de dados, devendo se cercar de alguns cuidados nesse uso, pois o viés estatístico deve "corresponder à realidade". No caso da hipótese não comprovar a sua relação com a realidade, ela pode ser nula, então o pesquisador deve delinear o tratamento dessa possibilidade na pesquisa[Wainer 2007].

Em um processo de criação de uma solução de programação, pessoas diferentes produzem algoritmos distintos, por isso é natural que linguagens de programação diferentes multipliquem esse efeito, pois no seu ciclo de evolução "natural" cada uma é concebida com centenas ou milhares de variáveis. O emprego de métodos de programação diferentes, para resolução de um mesmo problema, produz resultados físicos distintos no uso do hardware. É prudente que esses resultados sejam tratados com uma correlação somente por essa entrada de dados, uma vez que estudos não constatam essa correlação, apesar de não descartada essa possibilidade [Millani 2013].

Por ter dois grupos de arquiteturas heterogêneas (Node.js, Tomcat) essa pesquisa é uma pesquisa quantitativa transversal exploratória em relação a variabilidade da latência, pois ao comparar dois grupos distintos, ajuda a constatar fenômenos mais desejados em um grupo do que em outro, mediante a mesma realidade, no caso as

mesmas amostras. No âmbito da busca de indicadores de eficiência é uma pesquisa transversal correlacional causal [Sampieri 2013].

Averiguar e descrever comportamentos que melhor se saem em determinada realidade é um caminho para que mais informações possam ser obtidas através de pesquisas explicativas, mas algumas pesquisas descritivas vão mais adiante e descrevem a natureza das relações entre as variáveis [Gil 2010].

3. Referencial Teórico

3.1 Node.js

O Node.js é uma plataforma orientada a eventos assíncronos que processa Javascript do lado do servidor. Foi criado com o propósito de criar infraestruturas de redes escaláveis. Está presente em *smartphones*, televisores, aparelhos de *blu-ray* . [Mulder 2015, Krill 2015], em ferramentas de desenvolvimento como *framework* C QT [Kainpianen 2014], e no navegador Chrome [Ihring 2013].

Cabe ressaltar que Node.js não é uma nova linguagem, mas também não é uma plataforma como o Java. Trata-se de um aperfeiçoamento do Javascript que conseguiu inúmeros adeptos. Daum e Merten (2002) afirmam que é impossível ter uma única linguagem para resolução de todos os problemas de programação, pois a sua complexidade seria grande, bem como a responsividade necessária, tornando quase impossível o seu desenvolvimento por humanos.

O simples estudo de uma ferramenta leva a um maior contato e mergulho nas comunidades em seus conjuntos de padrões e princípios comuns de seu uso, e ditos como mais corretos por esses grupos. Toda plataforma ou linguagem tem sua forma de fazer as coisas, seja por um princípio ou por um mero conjunto de instruções únicas. Alguns princípios necessitam de mega sistemas para serem possíveis, outros são meramente uma evolução com uma ou duas linhas de código apenas, mas que ainda assim podem inspirar mais mudanças e tecnologias [Casciaro 2014].

3.2 Infra Estrutura e Middleware

Para entender melhor a importância da questão de uso em infraestrutura de rede, deve-se lembrar da mudança que levou "arquitetos de sistemas tornarem-se arquitetos de comunicação" [Daum e Merten 2002], assim como entender um pouco mais sobre o conceito de processamento assíncrono, base do funcionamento do Node.js.

Daum e Mertem (2002) citam que as organizações necessitam mais colaboração do que de integração, pois grandes sistemas integrados engessam as organizações, e uma nova era da Web orientada a serviços ocuparia esse espaço, ou seja, a era de megaprojetos cederia espaço a projetos menores e focados nos problemas das organizações com o mundo. Os sistemas empresariais devem permitir incorporar novas tecnologias, de forma que os projetos não sejam de longo prazo, caros e arriscados, devendo ser incrementais obtidos por uma estrutura flexível [Cummis 2002].

Não é possível entender o impacto de uma nova tecnologia sem levar em conta as tecnologias legadas que nos permitiram gerar um ponto de partida para uma melhoria. Os SGBDs (Sistemas de Gerenciamento de Banco de Dados) modernos hoje mantêm a antiga visão matemática sólida do SQL atendendo hoje milhões de transações, e chaveando seus dados pelo conceito de *lock*. Na visão de inúmeros

desenvolvedores, “segurar” uma quantidade de recursos de hardware enquanto ocorre alguma espera na execução de uma transação por um usuário, é ou era um evento aceitável. Em alguns casos situações de esgotamento de hardware ocorrem em função da elevada concorrência que gera os *deadlocks* [Cummis 2002].

Tradicionalmente operações de IO estão ligadas a operações com arquivos, acesso a banco de dados, comunicação de rede e outros recursos fora da CPU e da memória RAM disponível (exceto *swap*). A solução tradicional para problemas referentes à falta desses recursos inúmeras vezes é abordada de forma simplista, como a sugestão de colocar um hardware mais potente. Esses aspectos oneram muitos projetos que buscam melhorar desempenho com essa abordagem.

Outro ponto importante reforçado por Cummins (2002) é a importância das crescentes integrações entre os sistemas, dos componentes compartilhados e das especificações compartilhadas (*Model-Driven Architecture*), bem como a economia de escala desse processo de integração. Mas projetar um sistema desprezando os requisitos de desempenho entre diferentes agentes pode, em determinadas circunstâncias, demandar estruturas de hardware e software que tornam um projeto proibitivo. Tornou-se uma necessidade real e imediata para muitos negócios se preocupar com desempenho de seus sistemas [Molinari 2009].

Molinari (2009, p 27) destaca que, “sempre existirão gargalos que somente serão descobertos com o tempo, mas que muitos softwares já estão sendo construídos visando uma alta performance, quebrando o velho conceito de: -faz primeiro e otimiza depois” .

A sociedade mudou a forma como consome comunicação com a realidade da mobilidade. Isso criou novas necessidades e a informação passou a ser movimentada em tempo real. Hoje muitos aplicativos são projetados para o uso simultâneo de milhares ou milhões de usuários, capturando e transmitindo opiniões e outros dados importantes para as organizações. Daum e Merten (2002) caracterizam a evolução das topologias e classificam como topologia transacional para a relacional e posteriormente para topologia navegacional, e projetaram um novo conjunto de serviços inovadores em rede para essa fase.

Basicamente a transmissão de dados usa um conceito simples, uma mensagem quando deve ser transmitida vai para uma fila de mensagens, essa mensagem deve ter um destinatário e esse deve receber cada uma das mensagens de forma sequencial, uma única vez. Às vezes o destinatário é obrigado a dar uma resposta, ao remetente ou a alguém diferente, nem sempre imediatamente, e nesse casso de espera de uma resposta se denomina de resposta assíncrona [Cummins 2002]. Uma mensagem pode ter um evento de resposta, e sobre esse evento de resposta Cummins (2002) define um evento de *callback*, que é um evento fundamental no Node.js: “Quando uma mensagem é enviada de um cliente ou de um servidor, o objeto recebido, direta ou indiretamente, pode enviar uma mensagem que retorne para o objeto no processo de origem e que o processo de origem reconheça a resposta em um contexto”.

3.2.1. Multiprogramação em Sistemas Operacionais

Os sistemas operacionais abstraem para uma utilização mais conveniente uma série de recursos hardware e periféricos do programador, um exemplo disso é o conceito de arquivo, que na realidade é uma abstração de SO para acessar os dados

físicos no disco. Essas abstrações que são disponibilizadas em forma de bibliotecas, permitem ao sistema operacional intermediar e gerenciar os recursos entre vários processos através do escalonador de processos. A idéia da multiprogramação, que permite que o SO execute outro processo, quando um recurso está ocioso, surgiu com os sistemas operacionais de terceira geração [Tanenbaum e Woodhull 2000]. Isso surgiu no início de 1960.

O escalonador busca um processo apto, aguardando recurso, de uma fila e coloca-o em estado de execução até que um novo recurso desse processo demande uma espera, ou quando seu tempo de consumo esgotar. Essa interrupção faz o escalonador chavear o processo colocando-o em um estado de bloqueio, salvando os seus registradores antes de buscar outro processo na fila dos aptos [Carissimi, Toscani e Oliveira 2001].

Cada processo tem um bloco descritor, que contem as informações para que o SO possa retomar do ponto em que parou o processo antes de entrar em um estado de suspenso. Entre essas informações estão à lista de arquivos abertos, fatia (segmento) de memória que é utilizada de forma privativa ou compartilhada, tempo de processamento, prioridade e o conteúdo dos registradores [Carissimi, Toscani e Oliveira, 2001].

Para que o SO execute cada programa de forma sequencial dentro do seu fluxo de trabalho, o SO necessita evitar conflitos entre processos que disputam arquivos abertos, periféricos e memória compartilhada e outros recursos. Nesse aspecto, o SO bloqueia qualquer processo que solicite esses recursos, transferindo a tarefa para sua fila interna de eventos de sistema, nesse momento libera outros recursos de hardware, como a CPU, para que outros processos possam executar as suas tarefas. Nesse momento, o novo processo a receber esses recursos, não necessariamente executa um serviço de um servidor Web, como o Tomcat, ele pode nem mesmo estar relacionado com a função principal do servidor de aplicação (AS), e pode trocar o foco dos recursos entre vários outros serviços, até chegar novamente a vez do serviço principal da aplicação.

3.2.2. Promessa de Eficiência

Servidores Web normalmente são criados como um software que é um processo filho de uma sessão de SO. Esse SO libera recursos sequencialmente para os processos que funcionam de forma sequencial na sua lógica. O núcleo do sistema operacional, através de um laço varre continuamente a fila de processos, e processa em ciclos o próximo conjunto de instruções.

Enquanto servidores como o Tomcat e Apache usam essa camada extras de várias *threads* para processar seus códigos de forma sequencial, o Node.js o faz com um único processo e com um único segmento de memória [Ihring 2013], evitando perder o foco e o poder de processamento durante essas mudanças. Ele usa o conceito da multiprogramação processando instruções de forma assíncrona. Toda vez que alguma instrução necessita de algum recurso em operações IO, é delegado o evento a outro processo, evitando que o SO faça o bloqueio do processo principal e evitando as interrupções mais frequentes.

O Node.js tem um limite máximo de 16KB por processo, mas mesmo usando um único segmento de memória ele consegue processar arquivos gigantes, muitas vezes sendo usado como Middleware [Casciaro 2014].

Se por um lado ocorrem benefícios ao se reduzir as interações diretas com o SO com essa estrutura de eventos, por outro, o desenvolvedor de software com o Node.js passou a ter que se preocupar com um conjunto de instruções programadas que não possuem uma seqüência síncrona, além de criar controles e gerenciar recursos que normalmente são configuráveis em um servidor web [Casciaro 2014].

3.3. Teste de Performance

Normalmente nas fases de testes de um software o teste de desempenho é uma das últimas etapas e visa esclarecer se um determinado requisito de disponibilidade está contemplado. Mas como afirma Molinari (2009) à frase "faz primeiro, testa depois" já não é mais uma regra absoluta. Mas, quais os motivos que levam a se planejar e fazer um teste de performance?

Um sistema em tempo real deve ter no seu projeto especificado os requisitos de desempenho com valores de tempos e quantidades não arbitrários. [Pezze e Young 2008], sendo que a degradação de desempenho em um sistema pode ocorrer em muitos pontos, e muitas vezes estão associadas a componentes específicos que podem conter falhas de projeto ou de implementação não percebidas anteriormente [Molinari 2009; Jaskiel e Stefan 2001]. Especialmente em servidores WEB as atividades constantes de "ler mensagens de pedidos, gerar repostas" e transmiti-la, consomem uma quantidade representativa de recursos do servidor, existindo inúmeras variáveis que contribuem para essa demanda [Krishnamurty e Rexford 2001].

Além de *bugs* que podem causar gargalos, outro motivo para se analisar o desempenho refere-se ao fato de que os sistemas que tem uma razão de crescimento ou que sofrem *upgrades* de software ou hardware podem não corresponder às expectativas de desempenho após a mudança. Supondo que uma determinada atualização seja realizada, tendo em vista também objetivos de performance, o custo dessa atualização deve ser inferior a seus benefícios. A avaliação de desempenho nesse aspecto pode ser realizada anteriormente e posteriormente a mudança. Na primeira, além do reconhecimento da situação atual, também pode ser estimado objetivos de melhoria pretendidos, normalmente para isso se usa uma simulação. A segunda, em um ambiente mais próximo possível do ambiente de produção, busca-se quais os resultados obtidos, os gargalos, e a tolerância a falhas da nova situação [Jaskiel e Stefan 2001].

Realizar simulações para evidenciar qual o melhor algoritmo para determinada finalidade também é um dos motivos que levam ao teste de desempenho. Um exemplo conhecido dos servidores Web foi que nas implementações to protocolo HTTP 1.1, que ao se transmitir uma sequência grande de pacotes às várias conexões em paralelo não foram consideradas mais eficientes do que o uso dos pipelines. O benefício de obter uma mensagem de OK de cada pacote, em cada *thread*, não foi suficiente para superar a eficiência dos pipelines em uma conexão [Krishnamurthy e Rexford 2001]. Aos poucos os criadores de servidores WEB convergiram para a mesma solução em função desses testes realizados.

Portanto cabe um questionamento, como prever as situações na quais os usuários submeterão o software criado, uma vez que o teste é um teste de sistema (mais geral)? Nesse ponto o profissional com base na sua experiência pratica busca criar hipóteses que serão validadas em relação a seus objetivos, e planeja os testes sobre uma estrutura (hardware e software) para determinar se uma série de condições e ou

pressuposições são verdadeiras ou falsas, pois nenhum teste de performance é igual a outro [Molinari 2009].

Esse tipo de teste de sistema faz o uso potencial da especificação de cenários [Pezze e Young 2008], podendo ser modelados usando modelos visuais (diagramas, UML), matemáticos e mistos (Rede de Petri) [Molinari 2009]. Os tipos de variabilidade de carga em um teste de desempenho são definidos por Molinari (2009) como carga de trabalho constante, crescente e baseada em cenários e artificiais.

Muitas das variáveis de um teste de performance já possuem estudos que auxiliam a compreensão de determinados comportamentos, e algumas possuem uma tendência comum. O uso desses estudos pelo profissional de testes permite uma melhoria na qualidade dos planos de testes criados. Apesar de valores como média, mediana e desvio padrão poderem ser replicados para simular determinados comportamentos, em outros casos são mais bem representados usando expressões matemáticas. Uma função (f(x)) pode representar um comportamento que foi previamente validado em uma amostra, podendo um gerador de números aleatórios substituindo o x da função, gerar resultados da função com valores para teste mais próximos da realidade [Krishnamurthy e Rexford 2001].

Entre esses estudos encontra-se o emprego de regras e modelos matemáticos como o modelo de Markov para auxiliar a modelar comportamento de usuários, Modelo de Filas [Jaskiel e Stefan 2001, Molinari 2009], bem com vários estudos sobre comportamentos de uso, do tipo de recurso requisitado pelo cliente, do tamanho do recurso (quantidade de imagens), do tamanho da resposta (tamanho dos arquivos) e da popularidade de uma informação [Krishnamurthy e Rexford 2001].

A quantidade de combinações possíveis somente entre essas variáveis (tamanho de recurso, tipo de recurso, tamanho do recurso, tamanho de resposta), resulta em um grande número de cenários de testes possíveis que devem ser analisados e reduzidos em função de objetivos e limites existentes. Visando essa redução é realizado a analise das combinações sendo que algumas técnicas de testes combinatórios podem ser empregadas como no descarte de cenários. Nessa etapa, busca-se apontar as características variáveis e invariáveis do modelo, o estabelecimento das fronteiras de testes a serem executados [Pezze e Young 2008], e averiguar outros limites de recursos, como os recursos financeiros que normalmente estão atrelados a uma taxa de retorno esperada do projeto.

É importante destacar que pode ser estimando uma razão de melhoria ou de degradação em um teste de performance, mas essa variação deve ser comparada com os resultados esperados em termos quantitativos. Pode uma nova funcionalidade ter a previsão de degradar o desempenho em função de um benefício esperado. Nesse ponto, delimitar o universo de testes e ordenar os objetivos em uma lista de prioridades auxilia o planejamento e a realização dos testes, tornando-os mais objetivos e também contribui para escolha de ferramentas [Molinari 2009, Jaskil e Splaine 2001].

Para servidores WEB, especificamente na busca por melhoria de desempenho o analista de teste deve considerar algumas possibilidades antes de desenhar o projeto de teste, são elas [Jaskil e Splaine 2001]:

a. Reduzir a carga, deixando arquivos preparados para os momentos de pico;
b. Reduzir a carga do servidor, aumentado à taxa de banda de entrega;

c. Adicionar recursos ao gargalo que surgir, como memória e discos mais rápidos.

Para Krishnamurty e Rexford (2001), os analistas de teste devem considerar os seguintes aspectos relacionados à melhoria de desempenho dos servidores WEB:

a. Combinar chamadas em uma única chamada de sistema operacional deve melhorar a eficiência do transporte do dado, exemplo ao invés de escrever em um buffer temporário já escrever diretamente no local definitivo;

b. Tentar garantir que clientes e servidores utilizem o mesmo algoritmo de compactação;

c. Analisar os *logs* e localizar falhas do tipo mensagens HTTP 403 nos *logs*;

d. Alterar a aplicação para fazer chamadas no corpo da mensagem de forma única (consolidação de escritas);

e. Enviar uma mensagem já solicitando o fim de comunicação, pois elimina o *overhead* do servidor. Aqui há um problema da não confirmação recebida pelo cliente de que o dado foi recebido (influi no desenho do processo).

Em um cenário de teste de desempenho podem ser combinados vários tipos de testes em função dos objetivos. Entre os Testes de Sistema com essa finalidade tem-se [Jaskiel e Splaine 2001]:

a. *Smoke Test* – É um pré-filtro para determinar os limites de um teste mais aprofundado;

b. *Load Test* ou Teste de Carga – Avalia se a combinação de hardware ou software (web site, base de dados, rede, banda.) se vai atender aos requisitos de performance no mundo real. Pode ser requisitado na etapa de design para avaliar uma arquitetura em particular.

c. *Stress Testing* ou Teste de Vazão. Descobrir a quantidade máxima de sessões concorrentes que um componente suporta e se falha, em que ponto. Teste altamente dependente da natureza da aplicação e dos objetivos;

d. *Spike/Bounce Testing* ou *Overload*. Consistem em tentar prever e criar um cenário o mais pessimista possível, fazendo testes com variações de carga e picos o mais próximo possível de um cenário pessimista. Um problema pode surgir até mesmo com um comportamento regular, revelando um "*strobe effect*" que, com cargas com tempos mais randômicos não aconteceriam. Molinari (2009) classifica esse tipo de teste como *Stability Test* ou Teste de Estabilidade.

Molinari (2009) cita ainda o *Isolation Test* ou Teste de Isolamento. Uma vez que um problema seja descoberto esse problema é atacado para se descobrir mais informações e outras conseqüências dessa falha. Nesse ponto é uma boa prática de testes em geral, classificar as falhas em severas, críticas e moderadas, o que já auxilia na priorização de testes futuros [Young e Pezze 2008].

A escolha dos requisitos normalmente anda junta com o processo de construção das metas e objetivos no teste de performance. Cada requisito de um teste deve ser modelado como um processo com vários elementos como identificação, descrição, pré e pós condições, relacionamentos, dados de testes utilizados, casos de testes, ambiente de execução, dependências e métricas, resultados esperados e obtidos [Molinari 2009].

Entre alguns dos requisitos possíveis para um teste de performance para um site WEB tem-se como exemplo [Jaskil e Splain 2001, Molinari 2009]:

a. Lista de componentes que devem suportar uma carga especifica;
b. Tempo para o evento em HTTP e HTTPS;
c. Requisitos e tempo médio de resposta;
d. Número máximo de transações concorrentes;
e. Limite de vazão do componente;
f. Qual a banda necessária para o funcionamento com uma carga estimada;
g. Como o Web-site vai manipular grandes massas de dados;
h. Qual o *download* mais rápido;
i. Contingência ou balanceamento de carga;
j. Taxa de crescimento das tabelas no SGBD;
k. Uso de memória RAM e Cache;
l. Uso de CPU; e,
m. Preocupações já existentes antes dos testes.

Molinari (2009) propõe para Teste de Performance o modelo OTPM (*Open Test Performance Model*) aderente a macro atividades do modelo CMMI e as atividades de um desenvolvimento Ágil. Esse modelo é composto de nove etapas são elas: Identificação das metas, identificação do ambiente, planejamento de testes, configuração do ambiente, implementação dos cenários de teste, testes e retestes de cenários, monitoração de ambiente durante o teste, analise de resultados, elaboração dos relatórios.

Na elaboração de cenário de teste de desempenho é fundamental a escolha de ferramentas para realização e coletas de dados automatizados. Simular cenários com centenas ou milhões de usuários sem uma ferramenta adequada é praticamente impossível, da mesma forma a coleta de resultados, pois deve haver um sincronismo entre ambos.

Após a execução dos testes e coleta dos dados, na análise estatística dos dados obtidos é necessária apontar as possíveis causas de eventuais problemas. Na análise de resultados de um Teste de Performance localizar e identificar os gargalos é um dos resultados esperados. Esses pontos de estrangulamento podem estar em defeitos de software ou hardware, na subutilização de software ou hardware por problemas de configuração e mau uso ou na utilização em massa dos mesmos. Não necessariamente o gargalo seja interno, mas seus efeitos na estrutura analisada devem ser relatados [Molinari 2009].

Entre os problemas mais comuns na avaliação de resultados de Testes de Performance pode-se destacar a avaliação de cenários com margens de erros muito próximas, normalmente relacionadas ao fato que, o pesquisador não encontrou as variáveis necessárias para a avaliação. Delimitar as variáveis em cenários minimamente mais otimistas e pessimistas ajuda a minimizar esse tipo de evento [Jaskiel e Splaine 2001]. Um exemplo para isso é que margens muito pequenas de tempos não são

perceptíveis para os usuários finais, podendo não se revelar como um ganho real em um *benchmark,* por exemplo.

4. Experimento Para Busca de Dados

Esse trabalho teve como objetivo principal determinar qual dos dois serviços (Node.js e Java Tomcat) apresenta a menor variabilidade no tempo de entrega. Para medir essa variabilidade foi considerado o desvio padrão do tempo médio de entrega obtido em coletas no Jmeter, e para medir o uso de recursos no servidor se coletou amostras com o software Nmon da IBM.

Na avaliação de CPU considerou-se percentual utilizado pelo usuário o percentual utilizado pelo sistema bem como o consumo da CPU pelo sistema hospedeiro medido internamente no *host* virtual.

4.1. Ambiente de Teste e Limitações

Entre os recursos limitantes para a realização do trabalho está o ambiente principal de testes. Foi utilizado uma máquina CENTOS 7 com servidores virtualizados usando KVM. O servidor principal trata-se de uma máquina com:

a. Core i5-4670;
b. CPU 3.40GHz, x86, MHz 1096,234, Mips 6784,44;
c. Quatro núcleos;
d. 16 Gb de memória, SDRAM 1666MHz;
e. Disco SATA 3 - 7200 – 500GB, e;
f. Disco SSD (Solid State Drive), Kingkston 240 GB.

Nesse ambiente foram criadas duas máquinas virtuais, em cada uma instalou-se um sistema de arquivos independente, sendo que o sistema de arquivos utilizado para criação das máquinas virtuais no sistema hospedeiro, foi o XFS.

Os testes foram executados em um servidor virtual configurando em um *hypervisor* KVM. Essa máquina virtual tinha o núcleo um dedicado no sistema hospedeiro sendo usando o software chamado Tuna para realizar essa tarefa. É importante destacar que a escolha de um ambiente virtual foi uma imposição dos recursos disponíveis. Foram alocados 1,8Gb de memória do sistema hospedeiro para essa máquina de testes, sendo configurado nessa máquina virtual o número de arquivos abertos igual oito mil e quarenta e oito (8048).

As versões de Node.js e Apache Tomcat submetidos aos testes foram respectivamente o Node 4.4.7 e o Apache Tomcat 7.0.54 com o OpenJDK 64 bits (*runtime build* 1.8.0.91-b14 e *Server build* 25.91-b14). As bibliotecas usadas para *upload* do Node.js o Multer (versão 2.4.16) e o *framework* Express (versão 4.14.0) e a ainda o pacote denominado node-uuid (versão 1.4.7). Em contrapartida no Apache Tomcat usou-se com biblioteca os pacotes do projeto Apache commons-fileupload-1.3.2.jar e commons-io-2.5.jar.

O software utilizado na máquina virtual, para coletar as mostras de uso de hardware, foi o software originário da IBM Nmon (nmon_16e_x86_rhel72).

Em outra máquina virtual com o mesmo SO ocupando os núcleos dois e três, uma máquina de 2,5Gb memória que serviu para ser o simulador de clientes fazendo *uploads* de arquivos de diversos tamanhos. O software utilizado para fazer essa simulação foi o Jmeter (versão 3.0.r1743807).

Criaram-se duas rotinas para carga de *upload* e uma resposta simples com exatamente a mesma funcionalidade e tamanhos de recursos de respostas gerados. Essas rotinas foram criadas com as recomendações das suas respectivas documentações. Um detalhamento maior da construção e rotinas de testes encontra-se no Apêndice Processo de Criação de Arquivos e Rotinas de Testes desse artigo.

4.2. Criação das Amostras

Uma vez que se trata de uma pesquisa quantitativa no que se refere ao tempo de latência, e existe uma amplitude necessária para a busca de indicadores, é necessário respaldar essa busca com amostras mais similares possíveis a realidade para um Teste de Carga. No intuído de resolver esse problema prévio, buscou-se na literatura [Krishnamurty e Rexford 2001] as formas de melhor criar uma amostra de tamanho de recurso, no caso o tamanho dos *uploads* de arquivos.

Com essa proposta iniciou-se o processo homologando um conceito da literatura [Krishnamurty e Rexford 2001] que afirma que, em geral o tamanho de recurso no aspecto de tráfego para o corpo de um gráfico tende a um comportamento de uma curva normal (Lognormal), e a cauda do mesmo tende a um gráfico de Pareto representados no Quadro 1.

Quadro 1. Variáveis de comportamento conhecido de saída na literatura

Distribuição	**Parâmetro de carga de trabalho**
Exponencial	Tempos entre chegadas de sessão
Pareto	Tempo de respostas (cauda da distribuição) Tamanho de recurso (cauda da distribuição) Número de imagens embutidas Tempos entre chegadas de pedido
Lognormal	Tamanhos de resposta (corpo da distribuição) Tamanho de recurso (corpo da distribuição) Localidade temporal
Zipf	Popularidade do Recurso

Fonte: Krishnamurthy e Rexford (2001, p. 409)

Uma vez que na literatura não fica claro se o conceito aplica-se a tráfego de entrada por *upload*, é natural propor que o comportamento de *upload* possa ser similar ao tráfego de saída de um servidor.

Krishnamurty e Rexford (2001) expõem as situações e as fórmulas básicas dessas operações, nesse trabalho transcritas para o formato do Microsoft Excel no Quadro 2.

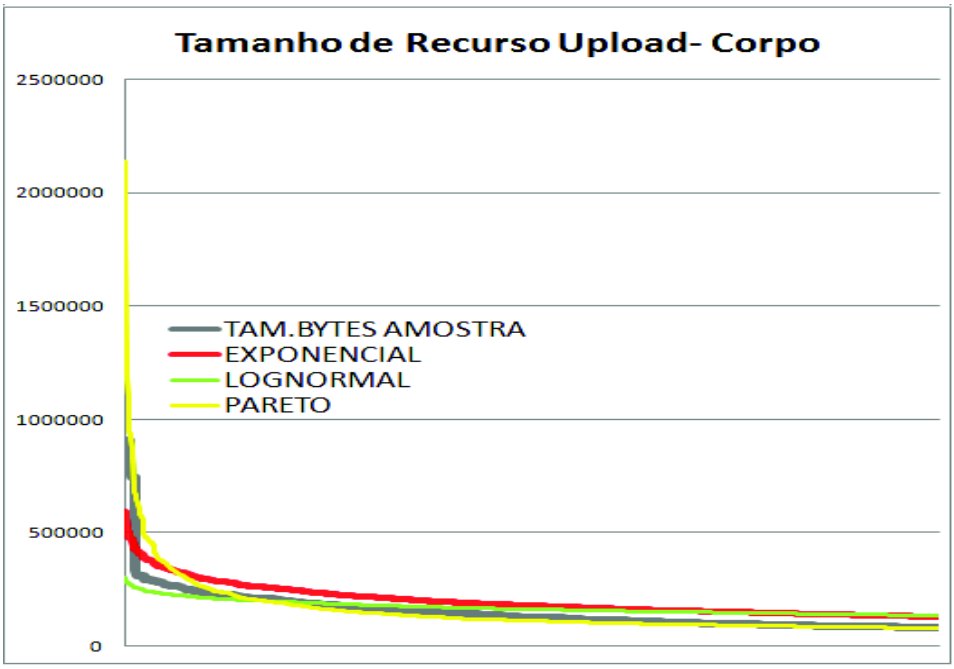
Quadro 2. Base das fórmulas usada

TIPO	FÓRMULA EXCEL
Pareto	**k/(ABS(aleatorio())^(1/a))**
Exponencial	**LN(ABS(aleatorio()))*-1*m**
Lognormal	**INV.normal(1-ABS(aleatorio())),m, desviopadrao_de_m)**
a= parâmetro de forma do gráfico de Pareto	a = 1,9 Valor ajustado de forma visual para a melhor aproximação das linhas do gráfico
k=parâmetro de escala do gráfico de Pareto	K =3000
k=m/(a-1)/a; m=(k*a)/(a-1)	m = média

Fonte: Adaptado de Krishnamurthy e Rexford (2001, p. 394 e 395)

Para validar as fórmulas do Quadro 2, obteve-se uma lista de *uploads* de cinco mil imóveis, feitos por diferentes agentes imobiliários (43 imobiliárias) em seus sites (nível de confiança de 99%, erro amostral de 1,8%). Após algumas avaliações e sobreposições de gráficos gerados pelas fórmulas do Quadro 2, obteve-se os gráficos das Figuras 1 e 2 conforme recomenda Krishnamurthy e Rexford (2001).

Figura 1. Validação de Amostra de Corpo do Comportamento de Upload

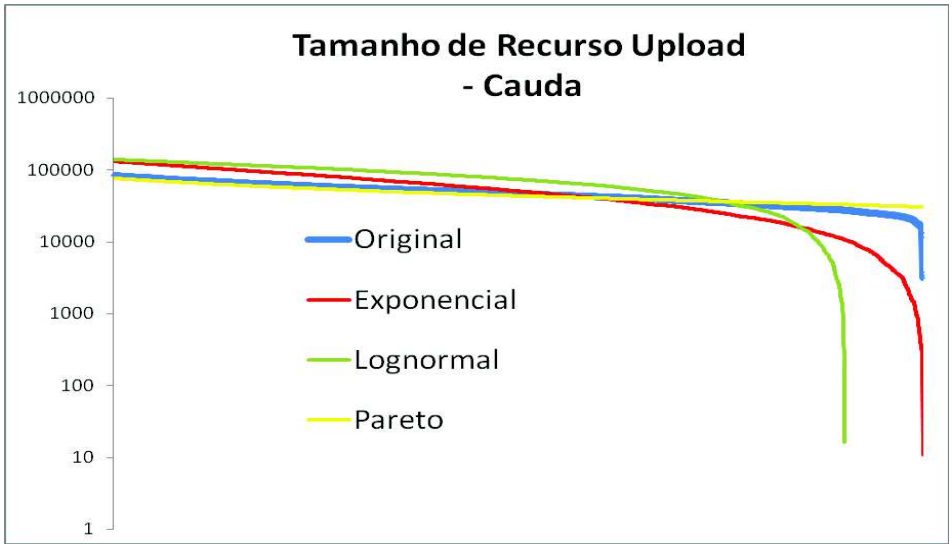


Fonte: Adaptado Krishnamurthy e Rexford (2001, p. 394)

O gráfico da Figura 1 demonstra que, com base no universo de fotos dessas empresas, é possível usar um simulador e criar um universo de arquivos de fotos de imóveis de forma aleatório seguindo esses estudos de tamanho de recursos.

O gráfico da Figura 2 demonstra a necessidade de não usar uma única fórmula. Apesar de ser um estudo preliminar, ele aponta um caminho para a geração das amostras para simulação de *uploads*.

Figura 2. Validação de Amostra de Cauda do Comportamento de Upload



Fonte: Adaptado Krishnamurthy e Rexford (2001, p. 395)

No intuído de usar uma metodologia diferente das Figuras 1 e 2 para averiguar as diferenças criou-se o Quadro 3. O Quadro 3 mostra a diferença entre a área média da amostra simulada gerada aleatoriamente pelo Excel, usando as fórmulas, e a média da amostra real, e comprova a necessidade de uma avaliação visual para essa finalidade. Apesar da área de corpo ser menor com a fórmula de Pareto visualmente na Figura 1, percebe-se que existe uma quantidade maior de itens na amostra simulada, que seguem uma curva normal.

Quadro 3. Diferenças de áreas para escolha de fórmulas foi descartada

CORPO	EXPONECIAL	NORMAL	PARETO
AMOSTRA GERADA	238.244.087	202.341.666	180.103.982
MEDIA AMOSTRA REAL	181.833.466	181.833.466	181.833.466
DIFERENÇA	56.410.621	**20.508.200**	**-1.729.484**
CAUDA	**EXPONECIAL**	**NORMAL**	**PARETO**
AMOSTRA GERADA	209.743.279	273.933.738	174.809.178
MEDIA AMOSTRA REAL	181.872.659	181.872.659	181.872.659
DIFERENÇA	27.870.620	92.061.079	**-7.063.481**

Fonte: Elaborado pelo Autor

Após a validação inicial, criaram-se dez listas com dez mil elementos cada uma, no formato CSV, tendo essas listas diferentes tamanhos médios, para submeter o servidor de testes a essa carga. Foram criados alguns Scripts Unix, que utilizando essas listas CSV criaram arquivos com os tamanhos exatos de cada amostra (pedaços menores de uma imagem maior, criada pelo comando *split*). Após a criação dos arquivos das amostras, esses arquivos foram instalados na máquina com Jmeter e as

listas CSV foram configuradas no Jmeter para depois simular clientes realizando *uploads* no servidor de teste de cada arquivo.

Todas as amostras foram derivadas de um arquivo JPG de tamanho maior. Imagens JPG, em função da sua compressão, são reconhecidas como um bom formato para Web, pois a compressão de arquivos é recomendada antes de qualquer *upload*.

4.3. Cenários Simulados e Escolha das Amostras Resultantes

As dez amostras foram criadas em uma proporção Fibonacci, sendo que amostras em proporções maiores foram calculadas e desconsideradas para esse artigo, em função de limitações dos recursos existentes.

Realizaram-se alguns testes preliminares para determinar valores de carga possíveis para não extrapolar o ponto de vazão. Manter os serviços com um consumo inferior a oitenta por cento (80%), do uso de CPU, é considerada uma boa prática em um servidor, sendo esse o ponto de controle para evitar chegar ao ponto de vazão.

Para cada amostra de cada tecnologia foram realizadas quatro execuções com coletas de dados no Jmeter no lado cliente, e no lado servidor os dados foram coletados com Nmon. Dentre essas quatro, selecionou-se a amostra mais adequada para analise. Os critérios de seleção de uma amostra, dentre as quatro execuções, foram:

a. Menor quantidade de erros;
b. Desempate, aquela que possuir o tempo médio mais próximo da média do conjunto das quatro execuções.

Na Tabela 1 encontram-se a quantidade de vezes que foram repetidos um teste, a quantidade de usuários (*threads*) simultâneos no Jmeter, a carga em Gigabytes, e a amostra que foi selecionada no final desse processo. A amostra cinco foi escolhida para comparar o uso de um disco SSD e um disco mecânico tradicional. Qual dentre as duas tecnologias (Node.js ou Tocamt), usando um disco mecânico tradicional, pode ter um melhor desempenho em relação ao tempo médio e o coeficiente de variação do tempo médio em relação a outra tecnologia que esteja usando um disco SSD.

Tabela 1 Amostras de Entrada e Selecionada na Saída

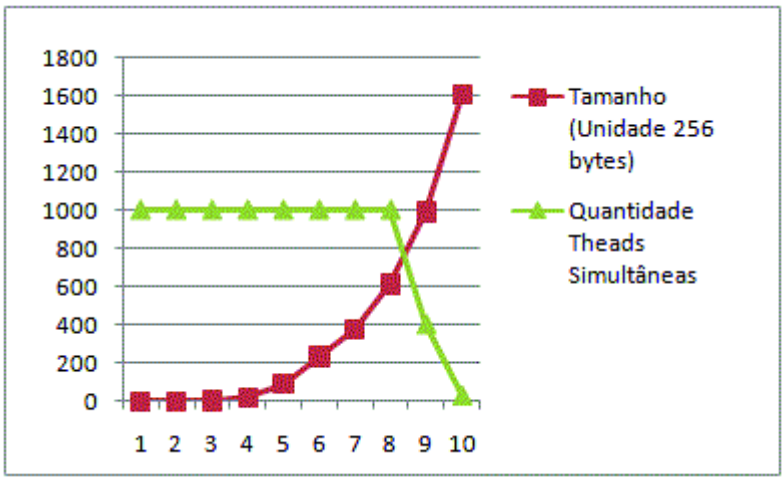
Amostra	Fibonacci	Entrada						Saída	
		Bytes		a=1,9	Jmeter		Tamanho	Seleção	
		Média	Desvio	k	Loop	Threads	Carga em GB	NodeJs	Tomcat
1D	3	4	5	1,2	500	1000	0,00	C	a
2D	5	55	89	22,1	500	1000	0,03	c	b
3D	8	790	451	243,4	500	1000	0,39	d	b
4D	13	5403	3137	1673,7	500	1000	2,64	B	d
5DS	21	22857	13272	7081	500	1000	11,16	Cc	a
6S	34	59864	34759	18545	500	1000	29,23	C	c
7S	55	96838	56227	30000	500	1000	47,28	D	d
8S	89	156702	90986	48545	500	1000	76,51	D	a
9S	144	253540	147214	78545	500	400	49,52	B	b
10S	233	410243	238200	127091	500	25	20,03	D	b

D=Disco S=SSD **Fonte: Elaborado pelo Autor**

A amostra cinco foi à amostra que iniciou um maior gargalo de IO no servidor, passando a gravação para um disco SSD, a partir desse momento todos os outros testes foram realizados usando esse hardware. Manteve-se para o Node.js os dados dessa amostra sem uso do SSD em função dos objetivos.

As duas variáveis desse conjunto de testes de carga são o tamanho médio do *upload* e a quantidade de *threads* simultânea. Uma vez já esgotado a capacidade do sistema de suportar arquivos maiores e a mesma quantidade de *threads* na amostra nove, então foi configurado a amostra dez com um tamanho de arquivo maior, mas com reduzida quantidade de *threads* simultâneas, e dentro das margens suportadas pelo hardware.

Figura 3. Limites explorados nos testes



Fonte: Elaborado pelo Autor

O objetivo de testar esses extremos foi verificar se os comportamentos das variáveis físicas coletadas no lado servidor, de ambas as tecnologias, se manteriam os mesmos em relação às demais amostras, ou se algum novo comportamento mais latente surgiria nessa situação. Esses limites estão representados da Figura 3.

5. Resultados e Analises

Este capítulo tem como objetivo apresentar os dados coletados após a realização dos experimentos propostos neste trabalho.

Para cada linha mencionada na Tabela 2 foram coletados resultados em uma linhas em outras duas tabelas, uma tabela para os resultados Node.js e outra para os resultados do Tomcat. As informações de latência, percentual de erros foram coletadas no Jmeter do lado cliente. As informações de uso de CPU e percentual de espera foram coletadas pelo Nmon no lado servidor.

A Tabela 2 contém os onze testes realizados no Node, pois a amostra cinco foi realizada com disco SSD e com um disco comum. Os somatórios do tempo médio e do desvio padrão são utilizados para obter a métrica física da variabilidade da latência

obtida pela razão dessas duas variáveis. Nessa tabela chama atenção a representação decrescente do uso de CPU enquanto crescia o tamanho médio dos *uploads*.

Tabela 2. Tabela com resultados no Node.js

Amostra	Amostras coleta Node.js							
	Latência milissegundos							
	Tempo Médio	Desvio Padrão Tempo	Desvio Médio	%CPU Médio	Desvio Padrão CPU	%Erro Amostra	Maior %Erro 4 amostras	%Espera
1D	428	728	381	69	15	0	0,01	1
2D	453	782	401	70	17	0	0	1
3D	463	829	414	68	24	0	0	0
4D	491	944	455	66	16	0	0	1
5D	*887*	*2778*	*987*	*43*	*25*	*0*	*0,03*	*32*
5S	785	1144	606	62	14	0	0	2
6S	984	1668	820	54	11	0	0,06	2
7S	1251	1866	1032	36	11	0	0,07	7
8S	1544	2190	1066	36	10	0	0,04	4
9S	814	847	458	26	6	0	0	5
10S	93	243	110	4	1	0	0	3
Soma	7305	11240	6730	491	125			
	Razão	1,54			25%			

D=Disco S=SSD **Fonte: Elaborado pelo Autor**

A Tabela 3 contem os resultados coletados dos testes com o Tomcat. Os dados apresentados na Tabela 3 o que chama atenção é o maior percentual de erros coletados pelo Jmeter enquanto o tamanho médio dos *uploads* crescia.

Tabela 3. Tabela com resultados no Tomcat

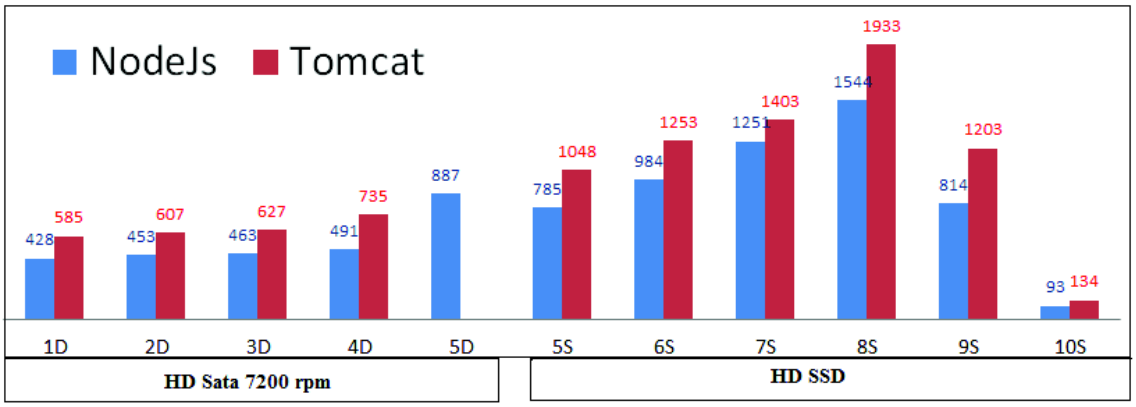
Amostra	Amostras coleta Tomcat							
	Latência milissegundos							
	Tempo Médio	Desvio Padrão Tempo	Desvio Médio	%CPU Médio	Desvio Padrão %CPU	%Erro Amostra	Maior %Erro 4 amostras	% Espera
1D	585	1923	762	16	16	0,00	0,03	1
2D	607	1848	764	12	8	0,00	0,03	1
3D	627	1976	802	8	8	0,00	0,09	2
4D	735	2441	913	14	9	0,00	0,13	2
5S	1048	3469	1132	56	18	0,10	0,38	2
6S	1253	3553	1340	47	13	0,17	0,06	3
7S	1403	3579	1482	47	13	0,27	0,53	1
8S	1933	4685	1973	54	12	0,54	1,08	2
9S	1203	2607	1102	62	14	0,00	0,03	1
10S	134	260	164	2	5	0,00	0,00	2
Soma	9527	26341	10434	318	116			
	Razão	2,76			37%			

D=Disco S=SSD **Fonte: Elaborado pelo Autor**

Comparado os dados apresentados nas Tabelas 2 e 3, observa-se em relação à latência de entrega, que em todas as amostras o Node.js obteve um tempo médio de entrega melhor que o Tomcat representado na Figura 4. O Node.js apresentou uma

latência de 23% (vinte três por cento) menor que o Tomcat nesses dez conjunto de testes que foram submetidas ambas as tecnologias.

Figura 4. Comparativo do Tempo Médio de Resposta (ms)



Fonte: Elaborado pelo Autor

A Figura 4 demonstra que inclusive na amostra cinco (5D), na qual aconteceu a mudança de hardware com o uso de tecnologia SSD, o Node.Js apresentou um melhor tempo de resposta, mesmo com um aumento considerável de IO (30%).

Em relação à variabilidade dessa latência, calculadas e apresentadas na Tabela 4, obtidas através da razão do desvio padrão pela média, observa-se que em somente uma amostra o Tomcat apresentou um menor coeficiente de variação de latência, na amostra dez, com baixa concorrência o Tomcat obteve um melhor coeficiente de variabilidade de latência, apesar de não ser algo perceptível para qualquer usuário.

Tabela 4. Coeficiente de Variação da Latência

Amostra	Node.js	Tomcat
1D	1,70	3,29
2D	1,73	3,05
3D	1,79	3,15
4D	1,92	3,32
5D	*3,13*	
5S	1,46	3,31
6S	1,70	2,84
7S	1,49	2,55
8S	1,42	2,42
9S	1,04	2,17
10S	*2,59*	*1,94*

D=Disco S=SSD

Fonte: Elaborado pelo Autor

Observou-se que nas situações de maior tamanho e maior volume de requisições o Tomcat apresentou uma maior quantidade de erros. Esse comportamento pode tanto aumentar o processamento, quando diminuí-lo, o que torna problemático eventual comparativos sem a localização das causas. Nos registros de erros do Tomcat mostra-se somente que as requisições falharam.

Na análise do uso de recursos pelo servidor chama atenção um maior consumo de CPU do Node.js em situações de grande volume de requisições e arquivos pequenos (primeira amostra). Constata-se também que no Node.js mostra uma correlação inversa no uso CPU e tamanhos de arquivos, enquanto que o Tomcat aparenta uma correlação direta, não sendo objeto desse estudo o motivo dessa aparente correlação. Uma causa provável seria que para obter uma maior prontidão da CPU, uma vez que não havia demanda, o Node.js deve ter mantido um laço (*loop*) com algum tempo constante que onerou mais o hardware.

Dentro dos diversos dados coletados no servidor observou-se um indicador promissor para a questão de monitorar a variabilidade de latência, o indicador é o número de mudanças de contexto por segundo (*pswitch*, quantidade de vezes que a CPU descarrega os dados em memória e carrega os dados de outro processo). Nos casos de testes de carga analisados, essa mudança foi menor sempre na tecnologia que apresentou menor coeficiente de variabilidade de latência em todas as amostras. Isso pode ser visto na Tabela 5.

Tabela 5. Quantidade Média de Comutação de Processos por Segundo (pswitch)

Amostra	Node.js	Tomcat
1D	656	2586
2D	641	2326
3D	646	2334
4D	628	2114
5D	458	
5S	657	1263
6S	642	1320
7S	669	1153
8S	596	947
9S	735	796
10S	2008	799

D=Disco S=SSD

Fonte: Elaborado pelo Autor

Pelo cálculo da correlação entre comutação de processos e a variação da latência para cada tecnologia e pelo cálculo entre as diferenças de ambas as tecnologias pelos valores contidos nas Tabelas 4 e 5, obtêm-se os coeficientes de correlação, contidos na Tabela 6.

Tabela 6. Coeficientes de Correlação entre Comutação e Coeficiente de Variação de Latência

Node.js	Tomcat	Diferenças
0,76	0,81	0,79

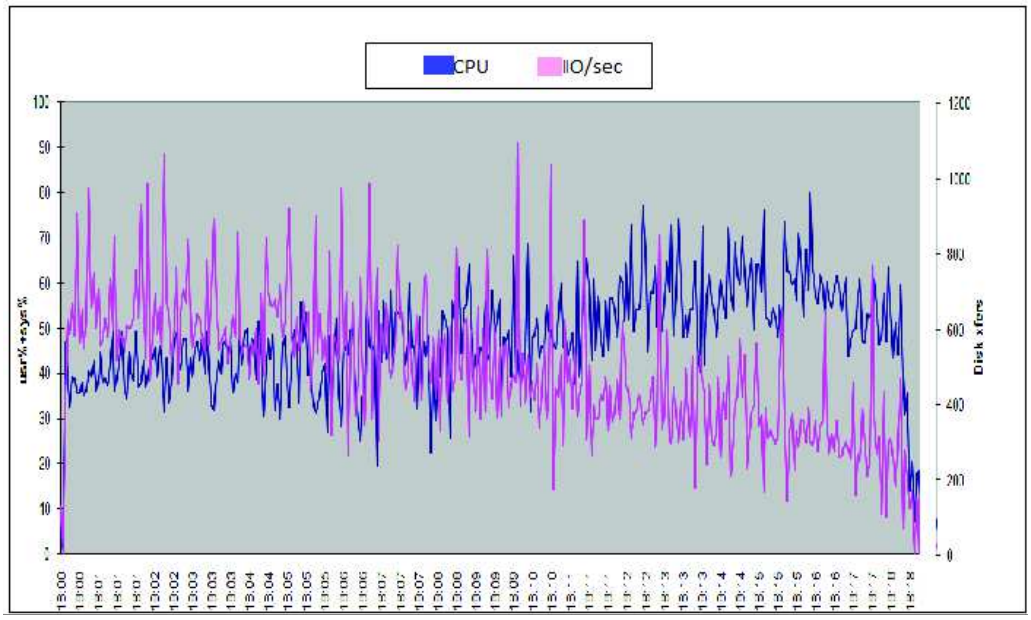
Fonte: Elaborado pelo Autor

Pela descrição da forma com que ambas as ferramentas trabalham e pelo índice de correlação, é possível supor que esse custo de abrir e fechar processos (*threads*), afeta o usuário final com decréscimos e acréscimos de tempos maiores. Isso pode estar relacionado a diversos fatores, como o custo de troca pelo hardware, mas também pode estar associada a chamadas indevidas no software.

Em relação ao consumo de recursos constata-se que ambas as tecnologias possuem comportamento similar em relação ao consumo de memória, exemplo no Apêndice Uso de Memória do Lado do Servidor. Após o início do processo, o consumo de memória mantém-se estável e a quantidade de memória utilizada em patamares muito próximos de ambas as tecnologias.

A Figura 5 representa outro comportamento, observado no lado servidor do Tomcat, chama a atenção que em todas as amostras obtidas conforme ocorria uma carga com maior volume de dados, tamanho médio dos arquivos maiores, também ocorria um decréscimo de IO. Mesmo não havendo uma restrição de IO iniciava-se um decréscimo do uso desse recurso no Tomcat, no lado do servidor. Esse fato gerou um efeito perceptível nos resultados coletados, com um acréscimo no tempo médio e uma maior variação da latência do Tomcat. Esperava-se que uma mesma amostra repetida cinqüenta vezes durante o teste, resultaria em um comportamento mais linear dessas medidas físicas.

Figura 5. Variação de Uso de CPU e IO na Amostra Sete Tomcat



Fonte: Elaborado pelo Autor

Observa-se na Figura 5 que essa constância linear esperada não aconteceu, pois existe um acréscimo do uso de CPU e um decréscimo do IO durante as cinqüenta repetições da mesma amostra de teste. A Figura 5 refere-se aos resultados da amostra sete do Tomcat.

Essa queda de desempenho no Tomcat deve ser vista sobre dois pontos de vistas:

- Alguma falha pode estar associada e gerar o comprometimento sucessivo de desempenho;
- Em situações com menor carga sucessiva, a ferramenta deve ter um desempenho melhor do que os testes a qual foi submetida.

6. Conclusões

Na busca dos objetivos, considerando-se o ambiente em questão (Físico e Lógico com máquinas virtuais KVM), os mesmos foram atingidos, no que se referem à avaliação da tecnologia com menor latência, o consumo de CPU e memória de cada tecnologia, a comprovação de que nem sempre um melhor hardware supera um melhor software, e a localização de indicadores físicos para acompanhar a variabilidade da latência.

Em relação à latência o Node.js apresentou um melhor desempenho em todas as dez situações de carga que ele foi submetido, sendo superior ao Tomcat, nessas amostras em 23% (vinte e três por cento) com nível de confiança de 99% (noventa e nove por cento) obtido de amostras de 500.000 *uploads* de carga de teste. Também em relação à variabilidade dessa latência o Node.js apresentou uma razão de 1,54 contra uma razão de 2,76 no Tomcat, nessas dez amostras sintetizadas. Somente em uma única amostra, a mostra dez, o Node apresentou uma variabilidade da latência maior que o Tomcat.

Para obter essa maior eficiência de rendimento em relação à latência (menor prazo de entrega e menor variação), o Node.js usou mais CPU nas amostras de um a seis e muito menos nas demais, ao inverso do Tomcat que usou mais CPU nas amostras maiores. Notou-se que muito desse maior consumo de CPU do Node ocorreu com arquivos de pequena quantidade de bytes.

Estudos em projetos futuros podem melhor qualificar essas medidas físicas buscando o entendimento do Node.js em relação ao consumo de CPU quando submetido a cargas pequenas, bem como um maior estudo do porquê o Tomcat apresenta uma queda de desempenho de IO quando repetido o mesmo teste.

A amostra cinco comprovou que nem sempre a aquisição de mais hardware supera uma solução mais enxuta, embora o exemplo aqui seja somente a carga por *upload* em diferentes tecnologias com essa especifica configuração de hardware. Comprova-se também como em outras pesquisas, o mérito da estrutura de eventos do Node.js nas tratativas de IO, que possui um processo desvinculado ao processo principal para tratar esses eventos. Nesse caso, usando um núcleo, essa estrutura foi mais eficiente do que com as *threads* Java.

Em relação à qualificação de uma métrica no lado servidor para acompanhar a variabilidade de latência, o uso da quantidade de processos comutados por segundo é um dado conhecido de administradores de sistema operacionais. Especificamente para a variabilidade de latência essa métrica possui para esses casos estudados uma forte correlação, quando não tem gargalo de IO, mostrando-se uma métrica promissora para estudos em projetos futuros com essa finalidade.

Com a realidade atual de aplicativos de tempo real, essa variabilidade da latência encontrada, reforça a importância dos testes de carga ainda mais quando é agregada uma quantidade desconhecida de componentes de *frameworks* de terceiros.

Apesar do Javascript ser amplamente conhecido, o desenvolvimento de sistemas assíncronos não é trivial, mas é possível. Nem sempre a tecnologia mais simples em primeiro momento, ou enxuta, é a mais fácil ou de menor custo. A escolha de uma ferramenta, além de questões relacionadas a desempenho, também está relacionada ao custo de manutenção de profissionais aptos a prestar assistência na mesma. Esses

fatores devem ser considerados, não somente com o Node.js, mas na escolha de qualquer ferramenta ou framework que se agregue a alguma solução de software.

Referencias

CASCIARO, Mario. (2014) NODE.JS Design Patters, Packt Publishing Ltda.

CCUNE, Robert Ryan; Node.js Paradigms and Benchmarks. University of Notre Dame, rmccune@nd.edu.2012.http://netscale.cse.nd.edu/cms/pub/Edu/GradOSF11FinalProjects/final.pdf

CISCO. Tráfego Cloud vai multiplicar-se por quatro entre 2014 e 2019. http://www.cisco.com/c/pt_pt/about/press/articles-2015/20151028.html (2015)

CHEDE, Cezar. Big Data volume + variedade + velocidade de dados. IBM; https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/big_data_volume_variedade_velocidade_de_dados (2012**)**

CRESWELL, John W. Projeto de Pesquisa. Métodos qualitativo, quantitativo e misto. Artmed Editora S.A. 2.ed. (2007).

CUMMINS, Fred. Integração de Sistemas EAI – Enterprise Application Integration. Arquitetura para Integração de Sistemas e Aplicações Corporativas. Rio de Janeiro. Editora Campus, (2002)

DAUM, Berthold; MERTEN, Udo. Arquitetura de Sistemas com XML, Rio de Janeiro, Editora Campus (2002)

FRINZELLE, John. A Scalability Study into Server Push Technologies with regard to Server Performance. Waterford Institute of Technology (2011).

GIL, Antonio Carlos. Como Elaborar Projetos de Pesquisa. Editora Atlas. 5.ed. São Paulo. (2010).

IBM, What is big data? ; https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html (2016**)**

IBM, Nmon Sourgeforge Project. Https://sourgeforge.net/projects/nmon (2016**)**

IHRIG, C.J.; Pro Node.js para Desenvolvedores. Rio de Janeiro. Editora Ciência Moderna (2014)

JASKIEL, Stefan P.; SPLAINE, Steven. The Web Testing Hand Book. STQE Publishing, Florida (2001).

KAINPIANEN, Miska. Qt Weekly #10: Creating Server-side, Cross-Platform and N-Screen Todo Application with Qt Cloud Services. Qt. https://blog.qt.io/blog/2014/05/13/creating-server-side-todo-application-with-qt-cloud-services/ (2014)

KRISHNAMURTHY, Balachander; REXFORD, Jennifier. Redes para a Web – Http/ 1.1; Protocolos de Rede Caching e Medição de Tráfego. Editora Campus Ltda. (2001)

KRILL, Paul. IoT.js is a lightweight version of Node.js that can run on devices with resource constraints. InfoWord.com. http://www.infoworld.com/article/2953719/javascript/samsung-banks-on-javascript-node-js-for-iot.html (2015)

MILLANI, Luis F.G. Análise de Correlação Entre Métricas de Qualidade de Software e Métricas Físicas. UFRS, TC 2013.

MOLINARI, Leonardo. Testes de Performance, Visual Books - Florianópolis (2009)

MULDER, Patrick, BRESEMAN, Kelsey. Node.js for Embedded Systems. Building Web Interfaces for Connected Devices. http://embeddednodejs.com (2015).

MYERS, Glenford. The Art Of Software Testing, 1979.

PRESSMAN, Roger S. Engenharia de Software, Sexta Edição. Editora MCGrawHill: Porto Alegre, (2010).

RIBEIRO, Camilo. 12 Lições aprendidas em Testes de Performance Server Side. http://www.bugbang.com.br/12-licoes-aprendidas-em-testes-de-performance-server-side/ (2013)

SEBESTA, Robert. Conceitos de Linguagens de Programação. Editora Bookman, 9 ed. Porto Alegre, (2011)

SAMPIERE, Roberto Hernández, COLLADO, Carlos Fernandez. LUCIO, Maria de Pilar Baptista. Metodologia de Pesquisa.; Editora Penso. 5.ed. – Porto Alegre (2013)

YOUNG, Michal; PEZZE, Mauro. Teste e Análise de Software processos, princípios e técnicas. Bookman, Porto Alegre (2008)

TANENBAUM, Adrew S., WOODHULL, Albert S. Sistemas Operacionais Projeto e Implementação. Bookman. Porto Alegre. Edição 2. (2002).

TIAN, Xinhui; HAN, Rui, WANG, Lei, GANG, Lu, ZHAN, Jianfeng. Latency critical big data computing in finance. The Journal Of Finance and Data Science, v.1, n.1 p. 33-41. (2015) http://www.sciencedirect.com/science/article/pii/S2405918815000045

WAINER, Jacques. Métodos de pesquisa quantitativa e qualitativa para a ciência da computação, 01/2007, "Atualização em informática 2007", Capítulo, ed. 1, Sociedade Brasileira de Computação e Editora PUC RJ, Vol. 1, pp. 42, pp.221-262. (2007)

APÊNDICES I

Processo de Criação Arquivos e Rotinas de Testes

Trecho de script que gerou arquivos dos tamanhos necessários com dados obtidos de uma lista CSV gerada pelos cálculos das amostras.

$ cat **awk.script**

#Gera arquivos do tamanho recebido no parâmetro $1

com o nome composto *tamanhobytes*.**split**

#raiz.jpg = nome de arquivo JPG maior em bytes que o maior tamanho da amostra

BEGIN { Nomearq=”raiz.jpg” }

{

Print “**split** “Nomearq” **–a10 –b**”$1

Print “**echo** “$1

Nomearq=$1”.**split**”

Print “**mv xaaaaaaaaaa** “Nomearq

Print “**rm xa***”

}

Chamada do script. Le uma lista de tamanhos numéricos e ordena de forma única decrescente executando o script passando como parâmetro o tamanho.

$ **cat lista_tamanhos_bytes_amostra1.CSV | sort –u –n –r | awk –f awk.script | sh**

Configuração do Jmeter com tempo de inicialização em 200ms (0,2s) e para o Jmeter ler os arquivos de obtidos de uma lista CSV, de forma sucessiva, usou-se a função __CSVRead, segue exemplo logo abaixo. Busca o primeiro registro que está posicionando o ponteiro e logo após já se posiciona no próximo registro. (*Files Upload* no Jmeter)

${___CSVRead(amostra9.CSV,1)}${___CSVRead(amostra9.csv,next)}

Rotina de upload Node.js

```
var express      =   require("express");
var bodyParser =    require("body-parser");
var multer   =   require('multer');
var uuid     = require('node-uuid')
var app      =   express();
var _fs      = require("fs");
var _config = require("./config");
app.use(bodyParser.json());
var storage =   multer.diskStorage({
  destination: function (req, file, callback) {
    callback(null, _config.tempDir);
  },
```

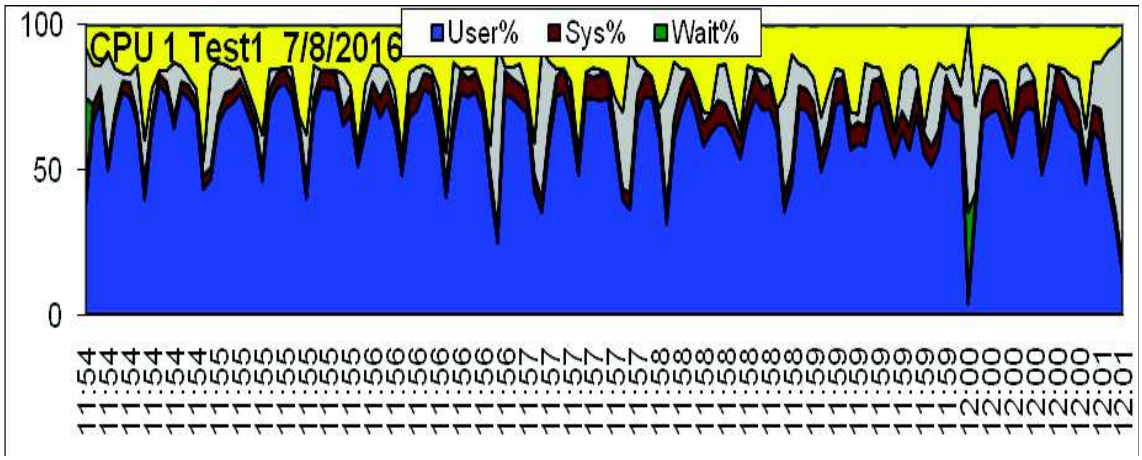
```
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + uuid.v1());
  }
});
var upload = multer({ storage : storage }).array('avatar',2);
app.post('/getimages',function(req,res){
    upload(req,res,function(err) {  if(err) {
            return res.end("Erro  in upload "+err );
        }
        res.sendFile(__dirname + "/resposta.html");
     });
});
app.listen(3000,function(){
    console.log("Working on port 3000");
});
```

Rotina de upload Tomcat.

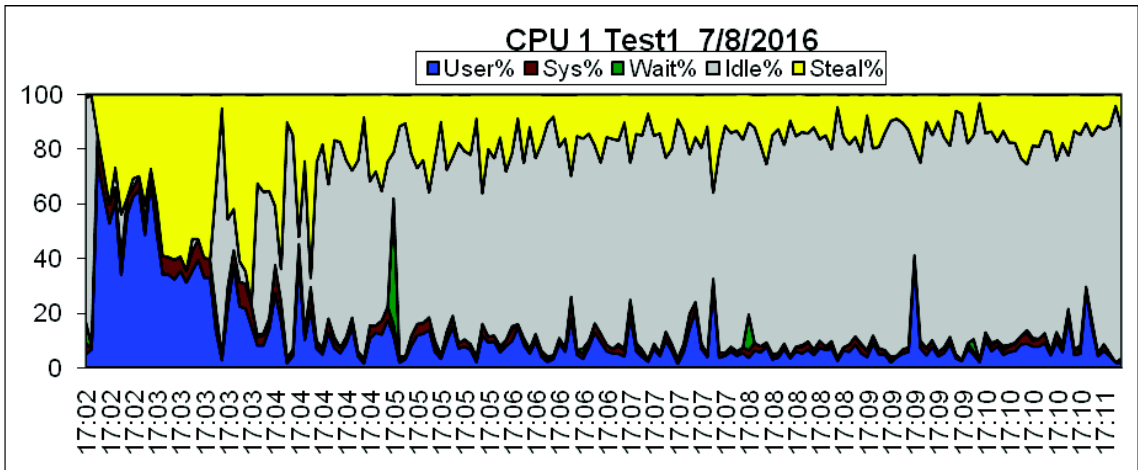
```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;
import java.util.UUID;
import javax.servlet.*
import org.apache.commons.fileupload.FileItem;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;

public class FileUploadServlet extends HttpServlet {
    private static final int MEMORY_THRESHOLD   = 1024 * 1024 * 3;  // 3MB
    private static final int MAX_FILE_SIZE      = 1024 * 1024 * 200; // 200MB
    private static final int MAX_REQUEST_SIZE   = 1024 * 1024 * 230; // 230MB
    ...
        try {
            List<FileItem> formItems = upload.parseRequest(request);
            if (formItems != null && formItems.size() > 0) {
                for (FileItem item : formItems) {
                    // processes only fields that are not form fields
                    if (!item.isFormField()) {
                        String fileName = new File(item.getName()).getName();
                        UUID uuid = UUID.randomUUID();
                        fileName=new String(uuid.toString());
                        String filePath = uploadPath + File.separator + fileName;
                        File storeFile = new File(filePath);
                        // saves the file on disk
                        item.write(storeFile);
                        request.setAttribute("message",
                            "Upload has been done successfully!");
                    }
                }
            }
        } catch (Exception ex) { ...
        }
        getServletContext().getRequestDispatcher("/message.jsp").forward(
                request, response);
    }
}
```

Amostras de Uso de CPU no lado Servidor

Exemplo de uso de CPU do Node na amostra um.

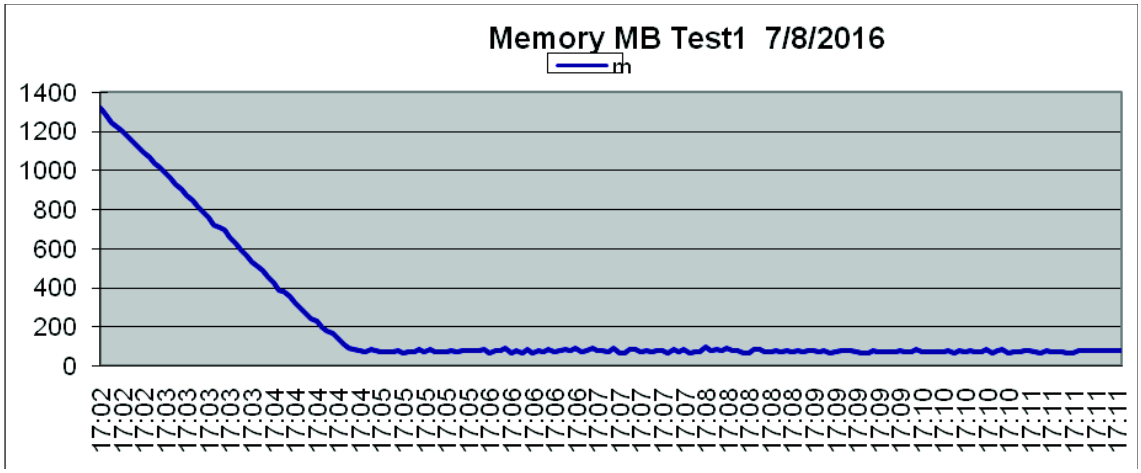


Exemplo de us de CPU do Tomcat na amostra um.

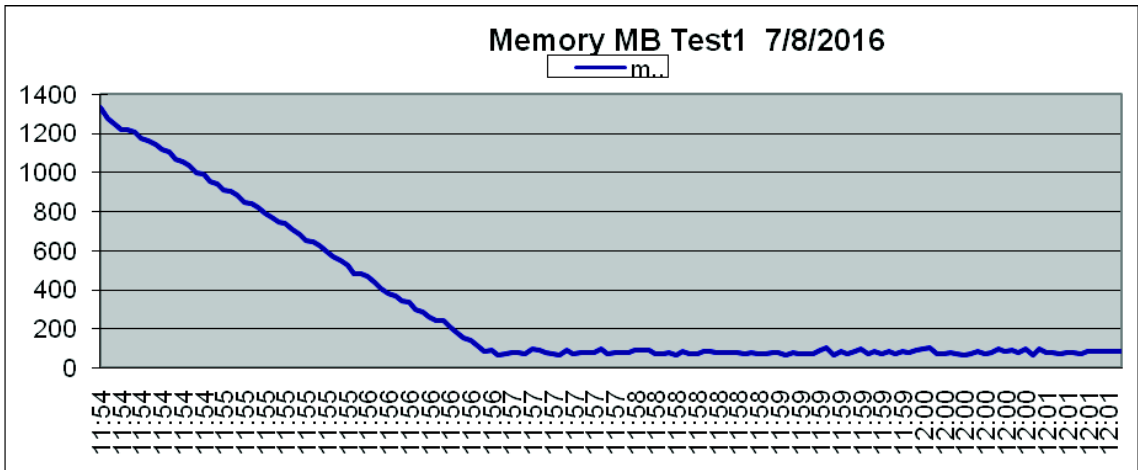


Uso de Memória no Lado Servidor

Exemplo de Uso de Memória Tomcat.



Exemplo de Uso de Memoria no Node.js



Exemplo de Uso de IO no Gráfico cinco sem SD do Node.js.