Destilando JMeter I: Introdução e Conceitos

Posted on July 16, 2013 by Camilo Ribeiro

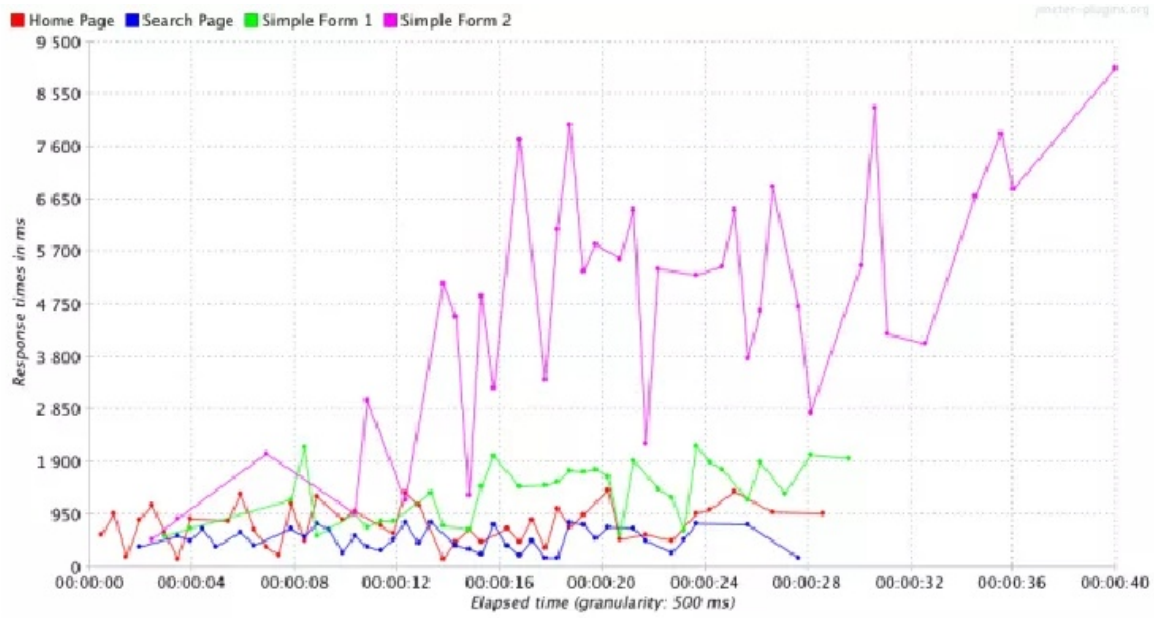
> *Leitura recomendada para antes de ler esse post:* *12 Lições Aprendidas em Teste de Performance Server Side*

Don't be evil! Use o JMeter para o bem :]

A muito tempo tenho recebido pedidos de pessoas sobre como usar o JMeter e sobre como fazer testes de performance! Apesar de não ser um *full time Performance Testing Engineer*, como generalista que sou, tenho alguma experiência com performance e por sorte acabei praticando bastante esse tipo de teste usando o JMeter nos últimos dois anos e passei por dor de cabeça o suficiente para escrever um tutorial completo com parte do que eu aprendi.

A boa notícia é que vou lançar alguns posts sobre JMeter no modelo conceitos e tutorial (passo a passo explicativo) totalmente gratuito e não comercial, e vou me esforçar para que **esse material seja mais detalhado e didático sobre JMeter na lingua portuguesa**. Com sua ajuda com dúvidas, feedback e compartilhamentos eu tenho certeza que isso é possível ;]

Nesse primeiro post vamos cobrir os conceitos básicos sobre os conceitos do teste de performance, defeitos e outras informações básicas que precisamos para começar os testes de performance. Além disso vamos falar um pouco sobre o projeto JMeter, sobre a interface gráfica e vamos preparar o JMeter com os plugins e ferramentas mínimas para executarmos *load tests, stress tests e soak tests*com profissionalismo. Ainda vamos ter uma visão geral dos relatórios, configurações e parâmetros que podemos usar através da UI do JMeter. (Sim é muita coisa e esse é mais um dos posts gigantes do bugbang.com.br :p)



Exemplo de gráfico de Tempo de Resposta por Tempo do JMeter [13]

Testes de performance é sobre tempo de resposta?

Quando falamos em testes de performance, sempre vem a cabeça medir o quão rápido é a resposta de uma página, mas testes de performance podem ir muito além disso.

A Microsoft define teste de performance como:

> *...a investigação técnica feita para determinar ou validar a velocidade, escalabilidade e/ou estabilidade características do produto sob teste.2[]*

A wikipedia tem uma definição que eu gosto muito:

> *...testes executados para determinar como um sistema performa em termos de responsivilidade e escalabilidade sob uma carga particular.*

e faz um adendo:

> *... também investigar, medir, validar ou verificar outros atributos de qualidade do sistema, tais como a escalabilidade, confiabilidade e uso de recursos.[1]*

Eu gosto de definir que testes de performance são:

> *Teste onde submetemos aplicações a cargas e condições específicas por tempo determinado afim de observar e avaliar os diferentes comportamentos que essas condições e cargas vão proporcionar.*

Esse comportamento pode ser observado de várias maneiras e o teste de carga pode ter como saída defeitos de performance (*timeouts*, tempo de resposta abaixo da expectativa, problemas de I/O, etc.), funcionais (falha no mecanismo de *caching*, inconsistências matemáticas ou de dados, etc), estruturais (armazenamento, *memory leak*, corrupção de dados, problemas de rede ou *load balancer*, etc) e de segurança (exposição de dados, exposição de *stack traces*, etc). Ou seja, o simples fato de uma aplicação ter um tempo de resposta abaixo de um valor estipulado pelo negócio não significa que essa aplicação tem qualidade quando submetida as condições e cargas dos nossos testes.

Indicadores de performance

De acordo com **Ian Molyneaux** em seu livro "**The Art of Application Performance Testing**"[3], existem basicamente quatro indicadores de performance em aplicações que podem ser classificados em dois grupos:

Indicadores orientados a serviços

O primeiro grupo é **orientado a serviços**, (que também podemos chamar de **fatores de percepção externa** ou **que afetam o usuário final**) define o quão bem ou não uma aplicação provê seus serviços para seu usuário final. Esses indicadores são:

Disponibilidade

A disponibilidade é a capacidade de uma aplicação manter-se operando sobre a carga, mesmo após muito tempo. Usualmente escutamos pessoas falando sobre "aplicações 24/7" ou de "alta disponibilidade", isso significa que essa aplicação não pode parar.

O custo de algumas aplicações quando paradas pode ser chegar a milhões em questão de minutos, especialmente em momentos críticos, quando o número de acessos é acima do normal. Um exemplo desse momento é a *Black Friday*, uma sexta feira especial nos Estados Unidos onde todos os comerciantes dão descontos significativos. O nosso teste *soak* (ver mais a frente) é um ótimo exemplo de como avaliar o comportamento da aplicação sob uma determinada carga durante horas

ou dias e em alguns casos também verificamos esse indicador nos testes de estresse.

Tempo de resposta:

O Tempo de resposta é o indicador mais conhecido e cobrado. O tempo de resposta é o tempo que a aplicação leva para dar o *feedback* apropriado para o usuário final.

Esse indicador é importante porque mesmo que a aplicação tenha alta disponibilidade, se ela não responder dentro do tempo esperado pelo seu usuário ela pode ser rejeitada por ele, abrindo oportunidades para concorrentes, mesmo que os produtos desses concorrentes sejam inferiores em termos de funcionalidade. Além disso algumas aplicações tem tempos máximos de resposta por contrato, como acontece com algumas apis de cartões de crédito. Usualmente medimos esse indicador em todos os programas de threads (ver a frente), embora não tenha muito valor durante o teste de estresse.

Indicadores orientados a eficiência

O segundo grupo é o **orientado a eficiência**, (que também podemos chamar de **fatores de percepção interna** ou **de natureza arquitetural**) definem o quão bem ou não a sua aplicação está utilizando os recursos.

Vazão

Mais conhecido como ***Throughput***, a vazão é a capacidade de a aplicação ou parte dela executar uma operação repetidamente em um período de tempo. Ou seja, podemos exemplificar o throughput de uma página como a quantidade de vezes que conseguimos receber uma resposta completa dessa página por segundo.

Esse número é importante porque define a capacidade da aplicação. Quando falamos em usuários, devemos fazer estudos entender o quanto esses usuários estão acessando essa aplicação ou página, para então definir qual será a nossa carga. Esse número é importante para todos os nossos testes, mas ele é usado em sua essência para os testes de carga, onde a carga é o número de usuários que podem reproduzir o throughput esperado em produção.

Utilização de recursos

Um aspecto muito importante para entender o comportamento de uma aplicação é o quanto essa aplicação necessita dos recursos computacionais para realizar as tarefas necessárias. Esses recursos são vastos e podem afetar diretamente os demais indicadores ou mesmo outros fatores, como custo de hardware, custo de banda de internet, etc.

Todos esses indicadores devem ser avaliados juntos, justamente por afetarem uns aos outros de diversas formas. Por exemplo, quando a sua aplicação consome muito recurso, ou quando o recurso está abaixo do necessário, ela consegue processar menos requests, logo o throughput dessa aplicação é reduzido. Se ela não consegue responder todos os requests que ela respondia antes, vai existir uma lentidão do sistema (tempo de resposta alto). Os usuários vão começar a atualizar a página e fazer novos requests, o que vai consumir mais recursos e diminuir ainda mais o throughput e aumentar o tempo de resposta, até um momento em que a aplicação pode entrar em colapso e perder disponibilidade. (**Sim, isso é ciência e é emocionante**)

Para entender um pouco mais sobre esses indicadores, recomendo a leitura do livro "**The Art of Application Performance Testing**" citado anteriormente, ou do blog post "12 Lições aprendidas em Testes de Performance Server Side".

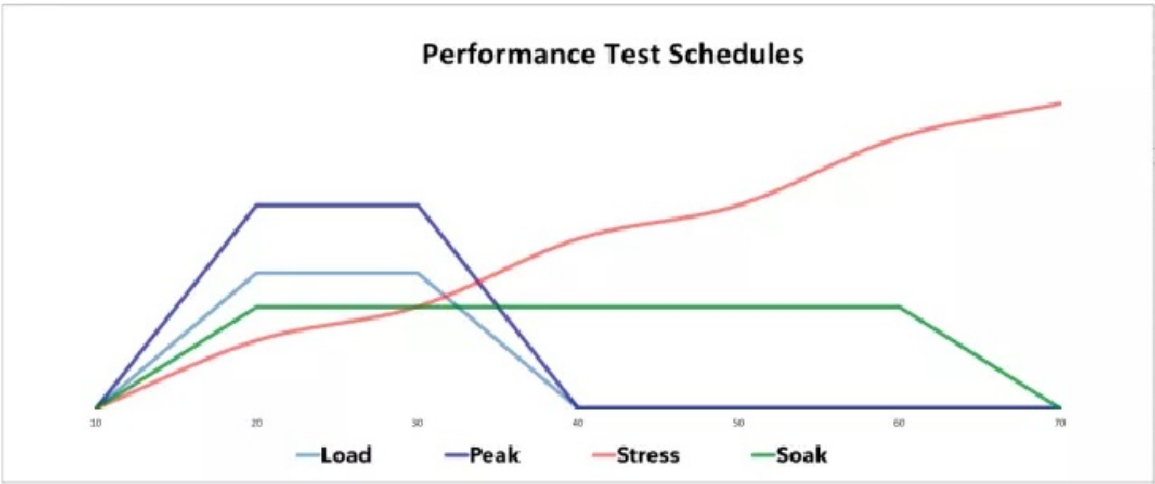
Schedules mais usados no mercado

Diferentes livros falam sobre diferentes "modelos de teste de performance", e inclusive usam diferentes nomes para esses modelos. Vamos usar nosso vocabulário baseado no que vamos aplicar na prática, logo vamos usar termos

aderentes ao JMeter. O que o jmeter chama de Programa de Threads (*Threads Schedule*) é basicamente uma estratégia para a execução de testes.

Como veremos a frente, usaremos um *plug-in* que vai nos proporcionar usar esse programa de forma bem intuitiva, relacionando tempo e *threads*. O tempo é tempo determinado em segundos, enquanto as *threads* é o número de "usuários virtuais" disponíveis. Os *schedules* mais usados são o teste de carga, o teste de pico, o teste de estresse e o teste de continuidade*.

* Não achei uma tradução adequada para *soak test*.



Modelos de schedules mais adotados de testes de performance

Load Test ou Teste de Carga:

A linha azul clara representa o teste de carga que pode ser considerado o modelo de teste onde submetemos uma aplicação a uma carga determinada e observamos o seu comportamento. A carga submetida neste teste deve ser a carga esperada em produção, para que possamos avaliar os indicadores de performance e ter uma visão antecipada dos problemas e riscos que estamos propensos a sofrer na entrada em produção, sua continuidade após um determinado período ou mesmo de acordo com nossas ultimas modificações.

Peak Test ou Teste de Pico:

Muitas literaturas gostam de dizer que o *peak test* é uma variação do *load test* onde avaliamos os mesmos indicadores, mas submetemos a carga do site a um momento de pico, como por exemplo a *black friday* que citamos anteriormente. Para isso usamos a nomenclatura *Nominal Load Test*, quando usamos o teste de carga com a carga normal e *Peak Load Test* para os momentos em que executamos a carga aumentada para simular pico de demanda da aplicação.

Eu separei os dois testes pois no meu ponto de vista, na maioria das aplicações não esperamos que os momentos de pico tenham os mesmos comportamentos e os mesmos resultados que temos em dias normais. Para esses momentos normalmente pensamos em estratégias como "elasticidade" dos nossos recursos de forma a garantir que a aplicação continue performance de forma satisfatória.

Stress Test ou Teste de Estresse:

O teste de estresse por sua vez assume uma característica muito mais focada em segurança do que em performance no meu ponto de vista, tanto é que o conceito de teste de estresse não fala sobre performance ou sobre carga, mas sim sobre exercitar o sistema sob condições hostis ao seu funcionamento, o que não exclui carga exageradas, mas inclui volume excessivo de dados, restrições de hardware ou software, sob ataques de segurança como *spidering*, etc.

Para executar um teste de estresse usando ferramentas de performance como o JMeter, podemos usar o programa de threads representado pela linha vermelha, em que aumentamos a carga progressivamente até o momento* em que a aplicação começa a sofrer com essa carga e finalmente cai ou falha**.

* Existem literaturas e pessoas que dizem que existe ainda um outro teste semelhante ao teste de estresse, que visa entender aonde é o ponto de maior resistência da aplicação.

** Um outro ponto interessante para avaliar é o pós teste de performance, para avaliar a recuperabilidade e integridade física dos dados, configuração e servidores.

Soak Test:

O Soak test (ou teste de continuidade), representado pela linha verde no nosso exemplo, é um teste que visa usar uma carga próxima a esperada em produção, mas manter essa carga por um longo período de tempo, que pode chegar a semanas dependendo da necessidade da aplicação. Esse teste exercita o sistema de uma forma bem diferente e é capaz de identificar falhas que não eram pegas antes, como por exemplo um *memory leak* resultante de uma *garbage collector* mau configurada ou mesmo uma restrição de banda pelo seu provedor .

Sobre a nomenclatura

Eu particularmente me preocupo muito mais com o entendimento dos conceitos e dos diferentes teste de performance, por isso não me preocupei em dedicar tanta atenção nem referências para esses testes neste primeiro post. Eu tenho certeza que se pesquisarem vão encontrar outros nomes, mas de uma certa forma os conceitos são bem aceitos na comunidade e na literatura. A mensagem que eu quero passar aqui é o que cada teste faz e quando usar esse teste, fique a vontade para mudar a nomenclatura na sua empresa.

Cenários, Volumetria e Ambiente

Eu diria que o sucesso de um teste de performance está tão relacionado ao balanço entre o ambiente, os cenários escolhidos e os dados criados quanto há definição da carga correta. Eu digo isso porque em certos casos você pode aumentar a carga vezes dez, que ainda assim vai ter um resultado muito melhor que que se tiver o volume de dados dobrado por exemplo.

Cenários

Os cenários podem variar de um simples acesso em uma determinada página, para um cadastro de um novo usuário ou mesmo upload de dados, imagens ou informações. Esses cenários devem ser elaborados pensando em

Google Analytics como ferramenta para definir cenários e distribuição de carga

situações semelhantes as reais, para isso pense que a *home page*, assim como todas as outras *landing pages*, vai ter mais acessos, que funcionalidades como de resetar senha ou de efetuar cadastros também vão precisar de uma atenção especial.

Embora seja importante testar cada uma dessas funcionalidades individualmente, é muito importante reproduzir um cenário próximo ao real, executando um teste de "integração de performance" similar ao esperado em produção.

A imagem ao lado é de um gráfico de comportamento gerado pelo Google Analytics, que pode ser uma ótima fonte de recursos caso esteja construindo novos testes para uma aplicação legada em processo de reconstrução.

Volume

O volume de dados deve ser planejado para o que espera-se da aplicação, não somente no momento do *release*, mas também quando ela estiver produzindo e funcionando a todo vapor. Segundo o livro **The Art of Application Performance Testing**"[3], é importante pensar nesse volume para o dia do *release*, seis meses depois do *release*, um ano depois do *release* e dois anos depois do *release*.

Essa é uma das partes mais "ciência" do planejamento, porque temos que estimar o volume de dados que vais usar para cada um dos modelos de dados do sistema, ou seja, se estivermos pensando em uma loja virtual temos que prever qual é o número de lojas, quantos produtos, quantos usuários, quantas peças de cada produto, quantas vendas, etc., para cada um dos nossos marcos de teste.

Ambiente

É quase impossível reproduzir o ambiente de produção, não só porque é caro, mas também porque vários detalhes só poderão ser implementados ou configurados, como dados de acesso a cartões de crédito, serviços de mailing, etc., por isso perseguir o ambiente de produção pode ser um tiro no pé.

Ao invés disso você pode entender a sua arquitetura e tentar replicá-la de uma maneira reduzida em um ambiente controlável. Se a sua aplicação e infra estrutura foram desenvolvidas para escalar horizontalmente (através da adição de novas máquinas), e seu ambiente de produção tem vinte máquinas por exemplo, você pode simular um micro ambiente de produção com duas máquinas e "imaginar" que seu teste vai escalar algo em torno de seis a oito vezes. Performance não é exatamente linear, portanto tenha sempre uma margem de segurança para fazer esse calculo. Caso o seu sistema tenha sido desenhado para escalar verticalmente (através da adição de recursos como memória e CPU ao(s) servidor(es)) você pode simular um ambiente parecido com o de performance mas com a infra estrutura mais básica.

O mais correto é fazer experimentos para calibrar essas métricas. Caso tenha a oportunidade de executar uma ferramenta de acompanhamento da performance em produção, como o New Relic [9], use-a para entender qual é a relação entre a performance de produção e do seu ambiente emulado.

Defeitos que podemos encontrar

Como comentado anteriormente podemos encontrar dezenas de defeitos durante os testes de performance. Vou tentar sumarizar alguns dos tipos de problemas que já encontrei com uma breve descrição, exemplo e problemas resultantes desses defeitos:

Tempos de Resposta Baixos: O problema clássico que encontramos mais frequentemente associado a teste de performance. Quando o tempo de resposta de um alvo de teste de performance está abaixo do tempo esperado pelo Business, podemos dizer que esse é um problema e precisamos estudá-lo. Bem possivelmente outros problemas listados abaixo também podem influenciar nessa lentidão.

Timeouts: Quando o tempo de resposta fica mais lendo do que a API está configurada para suportar ele pode começar a retornar erros de *timeout*, quando o servidor não é mais capaz de processar o *request* e simplesmente devolve um erro.

Esse tipo de problema quando não tratado pode ser raiz para outros problemas como a exposição de dados ou de *stack traces*.

Falha no mecanismo de caching: Testes de performance também podem ajudar a identificar quando um mecanismo de *caching* não está funcionando. O Mecanismo de cache pode por exemplo ter o objetivo de armazenar o conteúdo processado inicialmente na memória RAM de um servidor configurado para ser *caching* server e em seguida devolver para os *requests* seguintes o conteúdo já armazenado em memória. Testes de performance podem ajudar a entender se esse cache está funcionando corretamente.

Erros de load balancer: Outro teste que podemos fazer é monitorar todos os servidores que estamos usando para balancear a carga (incluindo o próprio *node balancer* que deve distribuir a carga entre os demais servidores) e em seguida executar testes de performance e avaliar qual é o *throughput* que está chegando em cada um e qual é o consumo de recursos, identificando por exemplo quando o *load balancer* não estiver calibrado.

Memory Leak: Como o teste de performance pode executar uma sequência de operações milhares de vezes, ele também pode nos ajudar a identificar se alguma rotina não está tratando a memória que não é mais necessária, ou seja, se o sistema está desalocando memória na hora certa. Caso não esteja, vamos ver um crescente consumo de memória mesmo quando terminamos as tarefas e a memória poderia voltar para o *pool* de memória livre.

Perda de Dados: Uma quantidade excessiva de *requests* pode causar problemas como bloquear uma tabela, incluir demasiados registros ao mesmo tempo em um repositório, substituir arquivos se por algum motivo estranho como por exemplo se usamos o timestamp para gerar o nome ou diretório desse arquivo, etc. Como testes de performance podem executar tarefas de uma maneira bem rápida, principalmente em sistemas distribuídos, esses tipos de problemas podem aparecer durante os nossos testes.

Exposição de Dados: Em alguns casos o teste de performance pode gerar erros 50x que espoem detalhes como mecanismos de cache, dados do servidor, *stack trace* com informações do banco de dados entre outras informações.

Todos os problemas acima podem existir em sua aplicação, e normalmente não paramos para pensar sobre todos eles e muito menos sobre como encontrá-los ou investigá-los no dia a dia. O teste de performance pode nos ajudar a encontrar esses problemas de forma mais fácil. Aliado a uma ferramenta de monitoramento de erros como o New Relic [9] podemos guardar o log desses erros e investigar depois que terminarmos o teste. Através do log podemos então criar testes específicos para essas situações e evitar problemas mais graves em produção.

Mão na massa com JMeter

O Apache JMeter (aka JMeter) é um software livre de código aberto para geração automatizada de carga criado e mantido pela Apache Software (http://www.apache.org/) como parte do projeto Jakarta [12]. Por ser uma ferramenta madura (criada em 2001 e RC em 2007) ela tem se mantido como a primeira ferramenta que nos vem a cabeça quando falamos de testes de carga ou performance.

Escolhi o JMeter como ferramenta por alguns motivos. O primeiro motivo é que o JMeter é uma ferramenta livre e open-source, ou seja, não só é gratuita como também tem o código disponível caso queiramos mudar alguma coisa, conferir alguma métrica, desenvolver uma nova funcionalidade, etc., assim todos podemos baixar e usar sem nenhuma restrição. O segundo motivo é que ela é 100% baseada em Java, dessa forma não importa qual o sistema operacional você esteja usando, o comportamento funcional vai ser o mesmo*. O terceiro motivo é que ele é a ferramenta mais usada no seu seguimento, inclusive nas nossas comunidades como o #DFTests [4]. O quarto motivo é que o JMeter oferece uma interface de linhas de comando muito bem organizada, o que permite que ele seja usado por máquinas (servidores de integração continua por exemplo) e pode ser distribuído em clusters para escalar com muita facilidade.

Caso ainda tenha dúvidas consulte o wiki oficial da ferramenta [11]. Para saber mais sobre outras ferramentas de teste de performance você pode consultar o blog "**Software Testing Help**" no post "**Top 15 Performance Testing Tools – Comprehensive List of In-Demand Tools with Download Link**" [8]

* Diferentes sistemas operacionais podem tratar as *threads* de forma diferente. Sistemas baseados em Unix tendem a trabalhar melhor com *threads* e processos computacionais do que outros sistemas operacionais.

Como "instalar" o JMeter?

O JMeter não é uma ferramenta para instalar. Você precisa executá-lo a partir de um terminal ou usando uma bat caso esteja usando Windows. Particularmente, eu sugiro que use um sistema operacional baseado em Unix como Linux ou OSX para seguir esse tutorial e mesmo para rodar os testes na sua empresa.

Para executar o JMeter você precisa de ter o java instalado no seu computador, para isso vá até o site da Oracle [7] e baixe a versão mais recente do JDK pasta baixar a versão mais recente no site da Apache [5], extrair os arquivos e ir até a seguinte pasta:

```
$ jmeter/bin/
```

Neste diretório, execute o comando:

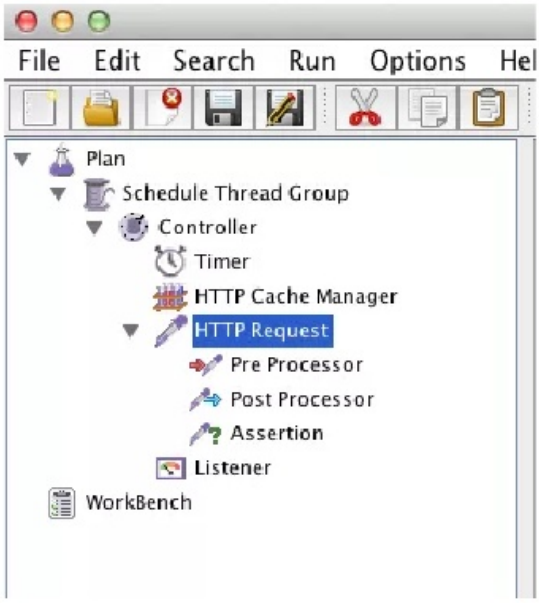
```
$ ./jmeter
```

Turbinando o JMeter

Agora você deve ver a interface gráfica do JMeter. Nesse ponto, feche o JMeter e faça o download dos plugins **JMeterPlugins-Standard-1.1.1.zip**, **JMeterPlugins-Extras-1.1.1.zip**, **JMeterPlugins-ExtrasLibs-1.1.1.zip** [9]. Extraia os arquivos e mova os arquivos ".jar" para dentro da pasta "jmeter/lib/ext", ou siga as instruções no link [9].

Feito isso, vamos ter novos recursos que serão utilizados neste e nos próximos posts, como o Ultimate Thread Group, o CMDJmeter e algumas bibliotecas para tratar dados por exemplo.

Entendendo os componentes básicos



Arvore de Componentes

O JMeter tem uma arvore de componentes bem intuitiva que fica a esquerda na interface gráfica. Essa árvore de componentes fornece as ferramentas necessárias para escrever testes, criar verificações, cuidar de situações especiais como cookies e controles de tempo, controlar e distribuir o fluxo de execução das threads, gerar relatórios, etc.

Nesta sessão vamos entender pra que cada um desses componentes é utilizado e quais são suas principais opções:

Plan

O Plano é um elemento único e requerido para qualquer atividade dentro do JMeter. Esse elemento agrupa todos os outros elementos e controla a execução de *thread groups*.

Não existem muitas opções para falarmos nesse tutorial inicial, mas uma característica muito importante desse componente é a capacidade de armazenar variáveis globais, chamadas de "*User Defined Variables*". Dentro desse elemento podemos armazenar um grande número de variáveis que podem ser usadas livremente em todo o *script* para flexibilizar a nossa implementação ao armazenar dados que variam de acordo com o ambiente sob teste, por exemplo *strings* de conexão com bancos de dados, url do servidor, etc.

Schedule Thread Group

O *Schedule Thread Group* é o componente que controla o nosso teste. Ele tem o poder de controlar a execução e por esse motivo incluímos o *Ultimate Thread Group* mais cedo no nosso tutorial, pois dessa forma estendemos o nosso controle para tempo de execução e não somente para número de execuções.

Exemplo do uso do Ultimate Thread Gorup

O exemplo acima representa um teste que deve aguardar dez segundos para começar (Initial Delay, Sec), iniciar uma thread por segundo (Startup time é igual ao número de threads), usar as vinte e cinto threads durante 20 segundos (hold Load for, sec) e finalmente desligar uma thread por segundo (Startup time é igual ao número de threads). O Ultimate Thread Group disponibiliza uma visão gráfica do número de threads, que inclusive inspirou o meu gráfico que exemplifica os programas de thread.

Você pode ainda controlar o teste em caso de falha, desligando a thread que encontrou o problema, abortando o teste, ignorando a falha, ou, como no exemplo acima, simplesmente abortando essa execução dessa thread, e começando um novo teste com essa thread na configuração "*Action to be taken after a sampler error*".

Logic Controllers

Os controladores lógicos são a maneira de direcionar os seus testes de performance para executar uma determinada sequência ou fluxo de eventos, tomar decisões baseadas em dados colhidos durante o teste, agrupar *samplers* ou outros elementos, etc.

O JMeter não é uma ferramenta de testes funcionais, embora muitas pessoas o usem como tal e mesmo no plano de testes citado acima ele tenha uma opção para "Emular testes funcionais de software". Algumas pessoas, assim como eu, não acham interessante emular testes funcionais em uma ferramenta como o JMeter e por essa razão evitam usar os controladores lógicos disponíveis (não é o meu caso). Mas eu posso listar uma vasta quantidade de situações onde eles são muito úteis, e aqui vão alguns exemplos:

Simple Controler: Um controller que só agrupa elementos, ajudando você a manter o seu teste sempre organizado e segmentado.

Once Only Controller*: Existirão dezenas de situações onde você vai querer executar algum comando uma única vez, para isso você precisaria sair do seu *thread group*, criar um *thread group* com uma *thread* só, e continuar o seu teste com outro *thread group* que permita o seu teste a continuar como anteriormente. Para evitar todo esse trabalho, existe um controlador de fluxo chamado "Execute uma só vez", que independente da quantidade de *threads* vai executar somente uma vez tudo que estiver agrupado com ele.

Throughput Controller: Outro controlador muito útil que tem o poder de decidir, aleatoriamente, quanto ao percentual ou mesmo número absoluto máximo de execuções que um determinado grupo de componentes vai receber. Esse controlador é muito útil quando temos que fazer um teste de integração de performance** e por isso precisamos distribuir a carga em proporções diferentes para cada um dos elementos sob teste.

Random Order Controller: Por default o JMeter executa seus testes como um script, ou seja, de cima para baixo. Caso precise executar alguns elementos em ordens aleatórias você pode usar esse controller.

Random Controller: Diferente do *controller* citado acima, o *Random Controller* não executa todos os itens sob ele de forma randômica, mas escolhe um elemento de forma randômica e o executa. Uma situação que eu usei esse controlador foi em um caso onde no meu projeto eu tinha que simular uma fila de requests aleatórios para algumas apis, mas segundo a especificação esse requests deveriam ser feitos um

por um de forma irregular. Setei uma thread e coloquei esse controler sobre todos os meus samplers e voila.

Controladores de repetição e condição: Ainda temos os controladores para funções básicas de lógica de programação, como o IF, LOOP, FOR EACH, SWICH e WHILE. Não vamos entrar em detalhes agora ainda porque as funções de cada um são bem

conhecidas por todos aqui.

* O **Once Only Controller** não tem a capacidade de executar uma só vez para clusters. Caso esteja rodando mais de um JMeter ao mesmo tempo usando o client server JMeter, ele executará uma vez o*Once Only Controller* para cada máquina com o script.

** Teste realizado com várias páginas ou apis ao mesmo tempo, para definir o comportamento de um ou mais desses itens quando outros itens sofrem um grande número de acessos ao mesmo tempo.

Timers

Várias vezes temos que usar pausas para execução de um script, por exemplo quando precisamos emular um determinado thinking time*. Para isso temos alguns timers como o **constant timer** que espera por um tempo absoluto em milissegundos ou o **uniform random timer** onde informamos uma variação entre X e Y para a nossa pausa, além de outros timers.

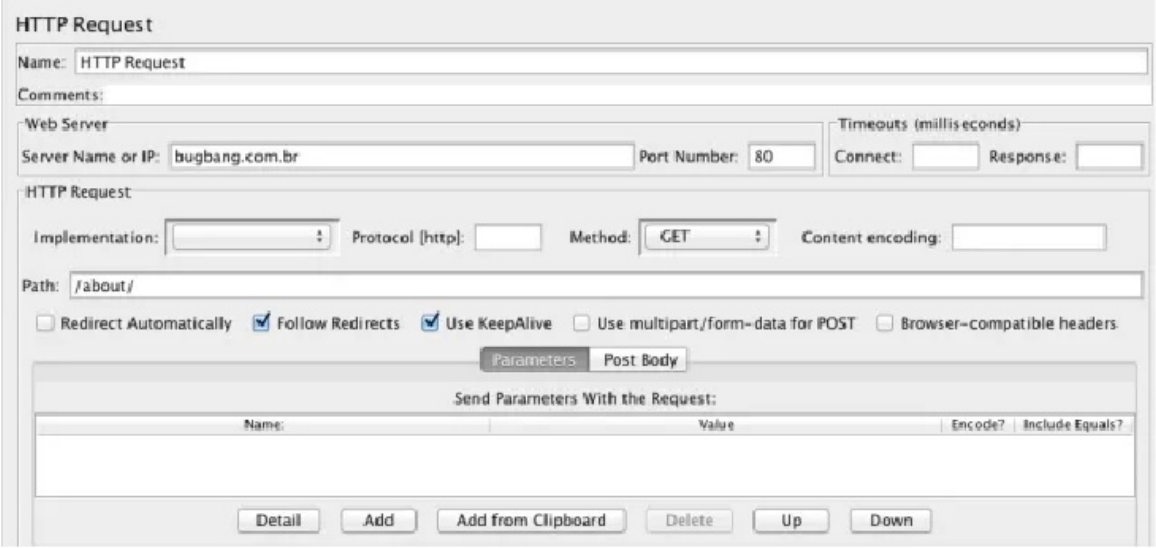
* Thinking Time é o nome do tempo que determinamos em alguns scripts funcionais e não funcionáis para o tempo que o usuário espera antes de realizar uma nova operação, por exemplo lendo, movendo o mouse, ou mesmo pensando sobre o que vai fazer.

Samplers

Samplers são onde determinados a forma como vamos acessar o alvo de testes, com quais dados vamos acessa-lo e quem é o alvo dos testes. Cada sampler corresponde a uma* página, um serviço, uma consulta sql, etc. Um sampler é submetido a um Thread Group, mas um Thread Group pode ter vários samplers

diferentes. Dessa maneira é que podemos usar um mesmo programa de performance para rodar os testes de integração, quando avaliamos os comportamentos de páginas quando seus pares estão sob a mesma carga simultaneamente.

* Uma página pode consumir dezenas de serviços, incluir outras páginas, assets, consultas sql, etc., logo essa afirmação diz respeito a um request para algum elemento, desconsiderando o seu comportamento interno que pode fazer novas chamadas.



HTTP Request Sample

Os campos básicos são:

Name: Um nome descritivo e intuitivo sobre a página ou serviço sob teste deste sampler.

Server Name or IP: O server ou IP do server, por exemplo bugbang.com.br, google.com, thoughtworks.com, etc.

Port Number: A porta usada, comumente 80 para páginas web. Você pode conseguir essa informação com os desenvolvedores, olhando no código ou tentando acessar pelo browser, ferramenta de requisições SOA, etc.

Method: Para a maioria das páginas vai ser GET, mas deve ficar atendo para formulários que normalmente usam POST.

Path: O que vem depois do servidor, por exemplo quando vamos para a página "Sobre" deste blog, ele nos direciona para a rota "/about/".

Parameters: caso trabalhe com query string*, com parâmetros no POST, ou com quaisquer informações que devem ser parametrizadas você vai ter que incluí-las aqui.

O exemplo acima faz uma consulta para a página sobre deste blog, usando a porta 80 que é a padrão e sem passar nenhum parâmetro. A opção de "Follow redirects" basicamente redireciona automaticamente quando recebemos uma resposta do tipo 30x. Não se preocupe com resposta HTTP agora, vamos falar disso em futuros posts dessa sequencia

* Query String são parâmetros que passamos pela URL como por exemplo os parâmetros de busca do google para a busca "Camilo Ribeiro" http://www.google.com.br/#q=camilo+ribeiro

Preprocessors e Postprocessors

Esses elementos são usados para executar ações antes (preprocessors) ou depois de um sampler (postprocessors). Eles são importantes para ajudar o engenheiro de testes de performance a criar condições especiais ou a executar ações dentro de um sampler.

Preprocessors: Um exemplo de bom uso de um preprocessor é

Postprocessors: Um ótimo exemplo do uso de postprocessor é quando temos uma aplicação que usa um CSRF id Token* (Cross-Site Request Forgery). Por definição, esse mecanismo evita que máquinas visitem ou postem informações em aplicações web. Para isso frameworks como o Rails implementam em seus formulários um campo oculto com um hash que é postado pelo próprio framework "garantindo" que não existe uma máquina por trás. Obviamente, como tudo do lado cliente é manipulável, usualmente realizamos um get no formulário e usamos um postprocessor como o "Regular Expression Postprocessor" para acessar o HTML desse get, armazenar o conteúdo em uma variável e mais tarde submeter esse valor como parte dos parâmetros do nosso post, simulando o comportamento do

framework. Falaremos sobre como burlar esse tipo de verificação de segurança em um post futuro.

Asserssions

Assim como testes funcionais, podemos usar mecanismos de verificação para ter certeza que o nosso teste está funcionando corretamente. Essas verificações servem basicamente para garantir que está tudo ok e que podemos continuar os nossos testes e principalmente para evitar falsos positivos.

Por padrão, o JMeter considera respostas 20x e 30x como sucesso e respostas 40x e 50x como falha, mas ambos os casos podem não ser erros em determinadas situações. Imagine que está executando um teste e o objetivo é realizar um post para logar na aplicação antes de continuar os testes, mas por algum motivo o usuário e senha do script de teste de performance está errado. A aplicação vai tratar esse problema e retornar a mensagem de não login com o código 200, ou seja, o nosso teste não deveria continuar pois existe um problema, mas para o JMeter está tudo bem pois tivemos uma resposta "200 Ok". Ou (para os paranoicos) se estivermos testando a nossa página de "Page Not Found", na verdade estamos esperando uma resposta "404 Page Not Found".

Para esses casos temos verificações que tem o poder de "mudar" o resultado de um script avisando ao JMeter que está tudo bem (ou não). Os principais modelos de verificações que temos são:

XPath: xpath classico para pesquisar no response por algum texto.

Duration Assertion: Você vai ter um relatório no final, mas se quiser falhar um request por ultrapassar um determinado tempo de resposta você pode usar uma verificação de quanto tempo ele está levando para terminar de carregar.

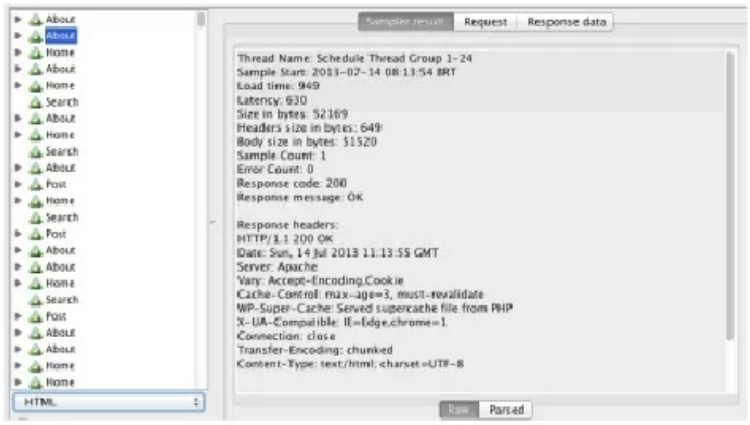
Response Assertion: Você pode definir qual é o código HTTP correto para uma página, dizendo ao JMeter que não deseja um 200, mas sim um 404 como no nosso exemplo anterior.

Eu quero desencorajar você se pensar em usar o JMeter para testes funcionais. O JMeter não é a melhor ferramenta para isso e nunca vai ser. Apesar da característica de verificação dele, ele não desempenha esse papel da melhor forma. Use essa ferramenta para testes de carga como estamos descrevendo aqui.

Listeners

Os listeners são os nossos relatórios. São vários relatórios e com a inclusão do nosso Ultimate Thread Group ganhamos alguns novos. Aqui eu vou falar sobre os mais úteis na minha opinião:

View Results in a Tree



Listener View Results in a Tree

Esse é, na minha opinião, o amigo número um durante a elaboração do script de performance. Esse Listener captura o request realizado (URL, parâmetros, etc.), o response do servidor (tempo de resposta, latência, código HTTP, etc) e ainda carrega o HTML/Javascript/JSON/XML gerado pelo request. Isso dá ao engenheiro de performance uma ferramenta essencial para "debuggar" o script e entender "Porque ele não foi de acordo com o planejado".

Além disso ele marca de verde e vermelho os requests de acordo com o resultado de sucesso ou falha. Depois que o script está pronto você pode também avaliar os requests realizados um a um caso seja necessário, por exemplo tentando entender porque um determinado request foi muito mais rápido ou muito mais lento que os demais.

Summary Report:

O sumário é outra ferramenta bem poderosa. Ele sintetiza algumas das informações mais valiosas do teste e agrega em uma só planilha de informações. Aqui podemos coletar informações de cada sampler como tempos máximo e mínimo, throughput, tamanho da resposta, quantidade de requestes realizados, tempo médio e desvio padrão.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
About	198	1029	721	2743	202.34	0.00%	2.9/sec	146.19	52168.9
Home	179	1237	797	2132	260.18	0.00%	2.8/sec	337.26	121842.0
Search	176	1435	955	2527	287.29	0.00%	2.8/sec	328.40	121546.0
Post	175	1185	786	1916	220.51	0.00%	2.8/sec	290.48	108080.9
TOTAL	728	1216	721	2743	284.34	0.00%	10.6/sec	1025.30	99512.9

Sumário do teste de performance

Eu gosto muito de usar esse relatório como uma primeira visão do resultado. Caso identifique que alguma informação está estranha, como por exemplo um tempo mínimo de 2 milissegundos e um tempo médio de 3 segundos, e então você pode se aprofundar e consultar o request de forma individual pelo "Results in a Tree"

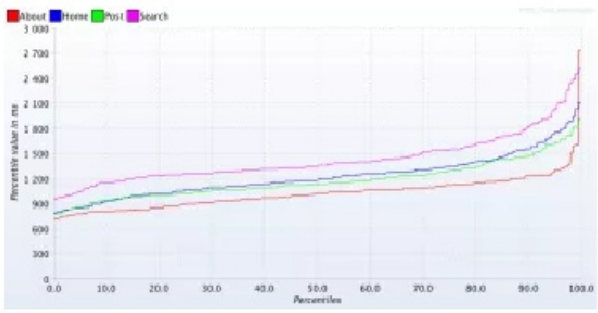
Respose Time over Time

O relatório de tempo de resposta por tempo de execução pode te mostrar dados interessantes sobre o comportamento diversificado das suas páginas/serviços durante o tempo de uma forma gráfica. Ele também tem a capacidade de te mostrar de uma maneira bem simples se você teve um outlier (valor atípico) tanto para cima quanto para baixo.

Response Time over Time

No nosso exemplo a esquerda, podemos ver que em algum momento um request vermelho (About page) respondeu com mais de 2/7 segundos. Pelo relatório podemos ver que essa página estava numa tendência de estabilidade mantendo seus requestes a menos de 1.2 segundos.

Response Time Percentils



Response Time Percentiles

Um dos relatórios menos lembrados, mas na minha opinião um dos mais importantes quando falamos de previsão e análise de tempo de resposta de uma aplicação.

Esse relatório nos da uma visão gráfica dos percentis por página, ou seja, ele enfileira em ordem crescente e cumulativa todos os tempos de resposta e cria uma linha no gráfico que aumenta no eixo X de acordo com a porcentagem e aumenta no eixo Y de acordo com o tempo de resposta. Em outras palavras, ele nos permite saber que 90% das nossas transações para a página About responderam com o tempo inferior a 1.8 segundos.

Essa informação é muito mais tangível do que dizer que o tempo médio da página about foi de 1.029 por exemplo. Quando falamos que o tempo médio foi de quase um segundo, algumas pessoas assumem que a maioria das pessoas teve um tempo de resposta de ate 1 segundo, quando na verdade elas tiveram tempos de resposta variáveis e ao mesmo tempo tão baixos e tão altos que o tempo fez parecer que estava baixo. Dessa maneira e com a ajuda desse listener você pode criar critérios de aceite mais concretos, por exemplo criando um critério que diz que "95% dos usuários devem receber a página antes de 1.2 segundos".

Hits Per Second

O relatório de Hits por Segundo também é bem útil, ele mostra na forma de um gráfico o número de hits (transações) que o servidor respondeu por segundo de acordo com o tempo que o teste foi executado.

Hits per Second

Com esse relatório
podemos identificar por exemplo, quando em um determinado tempo o servidor começa a responder menos hits. Essa informação é especialmente importante para ajudar a diagnosticar quando uma aplicação começa a piorar o throughput ao longo do tempo durante um soak test por exemplo.

ps: Esse número é a soma de todas as páginas e aqui não estamos avaliando o comportamento das páginas, mas sim da forma como o servidor processa todas as transações que estamos passando. Mais a frente vamos ver um listener que tem a capacidade de avaliar páginas individualmente.

Response Time Over Threads

Response Time over Threads

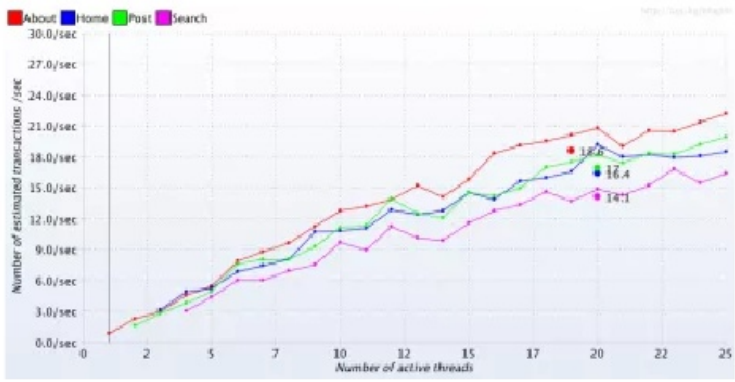
Esse listener nos ajuda a entender a relação entre o número de threads como especificado para o teste e seu tempo de resposta. Essa informação é importante para entender quando uma aplicação começa a perfromar abaixo do esperado na escala esperada para o teste de carga ou estresse por exemplo.

Para entendermos o comportamento é usamos o Ultimate Thread Group, que permite aumentar e dimiinuir o número de threads ao longo do tempo. No exemplo a esquerda podemos observar que existe um aumento progressivo do tempo de resposta ao longo do nosso aumento de número de threads.

Em testes de estresse por exemplo, onde uma abordagem pode ser aumentar a carga indefinidamente até que o sistema entre em colapso, esses números são importantes para entendermos quando o nosso sistema começa a apontar esse comportamento, também para sabermos qual o melhor momento para coletarmos os logs.

Transactions Throughput vs Threads

Assim como o Hits per Second, esse relatório tem como ideia coletar as informações de throughput, mas separando essa informação por sampler, de acordo com a imagem a direita.



Transactions Throughput over Threads

Esse relatório é essencial para entender onde estão os nossos gargalos de performance durante um teste integrado de performance.

Vamos imaginar que uma das nossas transações esteja muito lenta. Essa transação, possivelmente, está consumindo muito recursos do servidor, ou no mínimo está bloqueando uma ou mais das nossas threads por muito tempo, diminuindo o throughput de todas as demais transações. Nesse momento, esse relatório tem a capacidade de nos mostrar de uma forma muito simples qual é essa transação, e então podemos estudar uma abordagem especial para observar o comportamento interno e externo dela.

Além disso, esse relatório tem o mesmo benefício citado anteriormente no "Response TIme over Threads", que habilita o engenheiro de performance a entender quando o sistema (ou nesse caso até uma funcionalidade) entra em colapso durante um teste de integração de performance.

PerfMon Metrics Collection

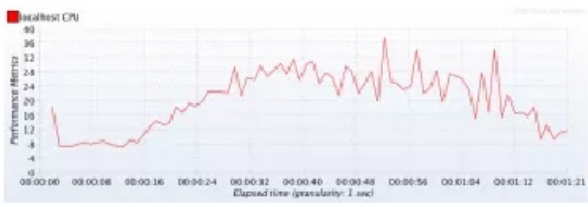
Esse é um coletor de métricas do servidor automatizado. Eu particularmente gosto de verificar os dados do jeito antigo, consultando logs, acompanhando pelo próprio servidor e usando algumas ferramentas como o New Relic que proporciona uma visão mais detalhada, mas caso não tenha ainda uma solução como o New Relic ou não consiga acompanhar os logs, você pode usar o PerfMon Metrics Collector.

Esse é um listener que precisa de um setup especial. Para coletar essas métricas precisamos que o servidor tenha java1.6+ instalado e precisamos baixar um agente e executá-lo. Você pode encontrar o passo a passo em inglês aqui [10], mas caso esteja usando um servidor linux você pode executar os seguintes comandos:

$ wget http://jmeter-plugins.org/downloads/file/ServerAgent-2.2.1.zip

$ tar -zxvf ServerAgent-2.2.1.zip

$ ./startAgent.sh



Perfmon Metrics Collector para CPU

Com esses comandos você vai iniciar o agente que permite ao JMeter coletar informações como consumo de memória, de rede, de I/O, CPU entre outras. Para coletar essas informações você ainda vai precisar adicionar o listener e colocar o IP da máquina e a informação que deseja coletar.

Particularmente eu gosto de instanciar um coletor para cada métrica que vou analisar, porque como os valores são muito diferentes, o JMeter cria umas conversões malucas para manter a estabilidade da escala de cada um dos conversores. No meu ponto de vista essas informações são importantes o suficiente para terem atenção dedicada dos engenheiros de performance.

Os resultados nos ajudam a entender os motivos de lentidão derivados do hardware ou do consumo excessivo de recursos computacionais. Essas informações são importantes para avaliarmos o que está acontecendo com o nosso servidor quando um determinado número de threads chega no que o negócio espera, ou mesmo para entender o porque o nosso teste soak deu problema.

O JMeter ainda possui vários outros listeners como o **Transactions per second** e **Active Threads Over Time**, que podem te dar novas perspectivas sobre os resultados sob avaliação. Você pode experimentá-los a vontade para entender quais são os mais importantes para você nos seus testes de performance. O

importante é que antes de usar qualquer um desses relatórios como uma visão definitiva, você pense sobre o que quer medir e então estabeleça um conjunto de métricas.

Estamos falando somente dessas métricas neste primeiro tutorial, mas é importante lembrar que esse é apenas o JMeter, ou seja, estamos falando de uma das ferramentas que usamos dentro de um conjunto de ferramentas para coletar informações sobre performance. Apesar de o JMeter ter listeners e relatórios para coletar dados, a sua principal função ainda é a de **gerar carga**, por isso é importante ter outras maneiras de monitorar a performance da aplicação sob carga.

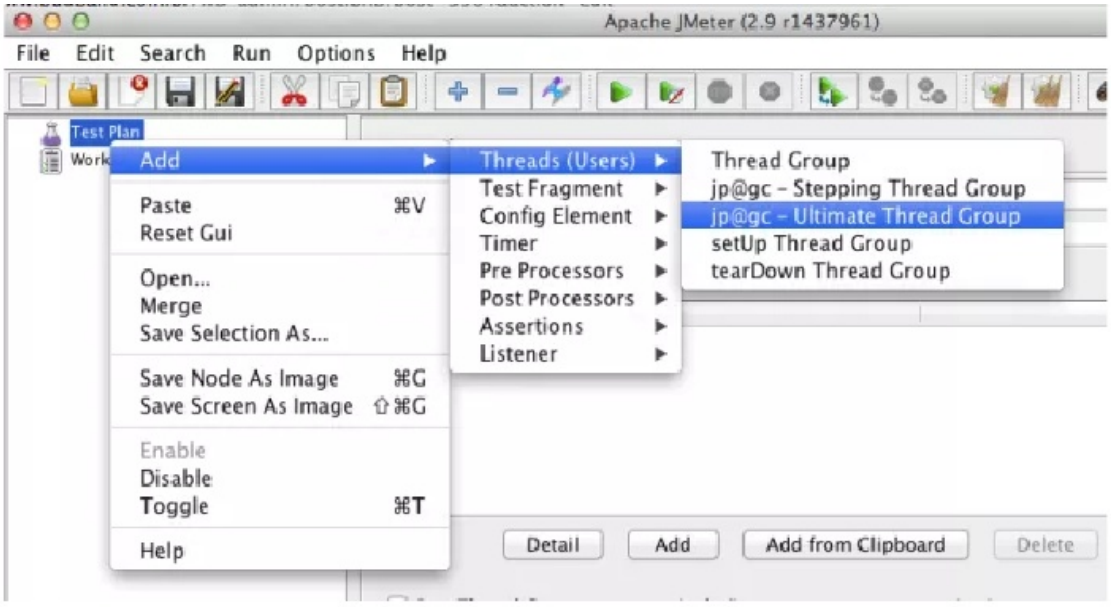
Hello Performance com JMeter

Agora vamos executar o JMeter contra um servidor e ver alguns desses dados. Para isso use o ambiente que criamos no início deste post.

Abra o JMeter, caso esteja usando o linux, basta executar o aplicativo ou o arquivo shell script dentro da pasta bin do JMeter, se estiver no Windows use o arquivo .bat. (lembre-se que precisa do java instalado no seu computador!)

```
$ ./jmeter
```

Agora você vai ver o plano de teste a sua esquerda. Clique com o botão direito nele e vejas as opções disponíveis. Vamos iniciar adicionando um Ultimante Thread Group:



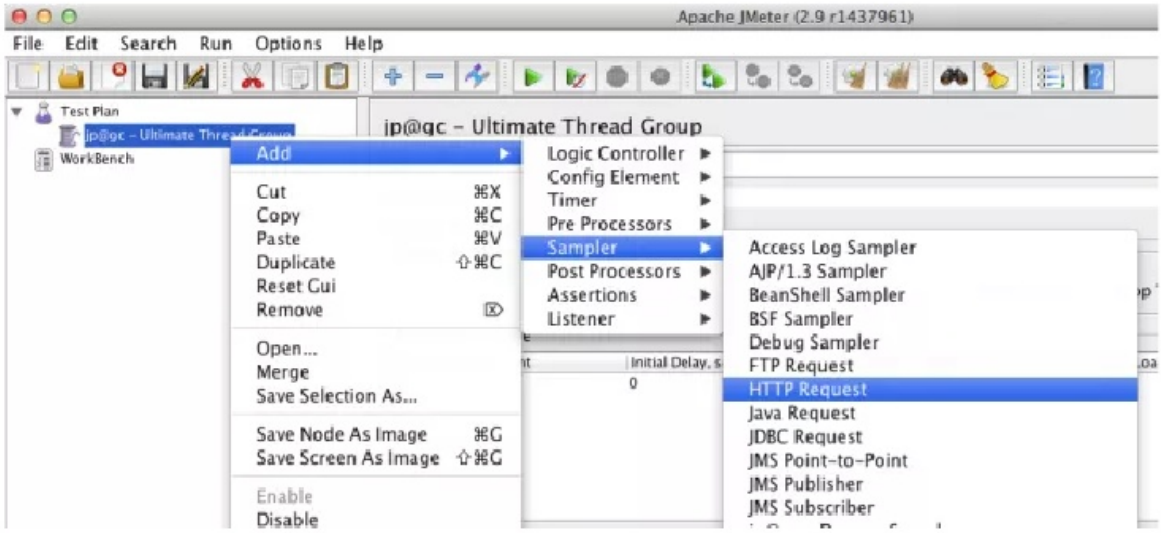
Passo 1: Adicionando um Ultimate Thread Group ao nosso plano

Agora que temos o nosso plano, vamos pensar em qual programa queremos simular. Eu recomendo que simulemos um load teste curto, somente como prova de conceito. No meu caso vou usar 3 threads para não gerar muita carga e vou usar um intervalo de dois minutos para o teste, dividindo 30 segundos para rumpup (subida da carga), um minuto para manter a carga e mais trinta segundos de rumpdown (descida da carga). Para adicionar essa carga temos que usar o botão "add row".

Passo 2: Adicionando o programa que queremos usar no teste de carga

Agora que temos o programa preparado com o Ultimate Thread Group, vamos adicionar um sampler do tipo HTTP Request clicando com o botão direito sobre o

Thread Group que acabamos de adicionar, selecionando as opções “Add” > “Sampler” > “HTTP Request”.



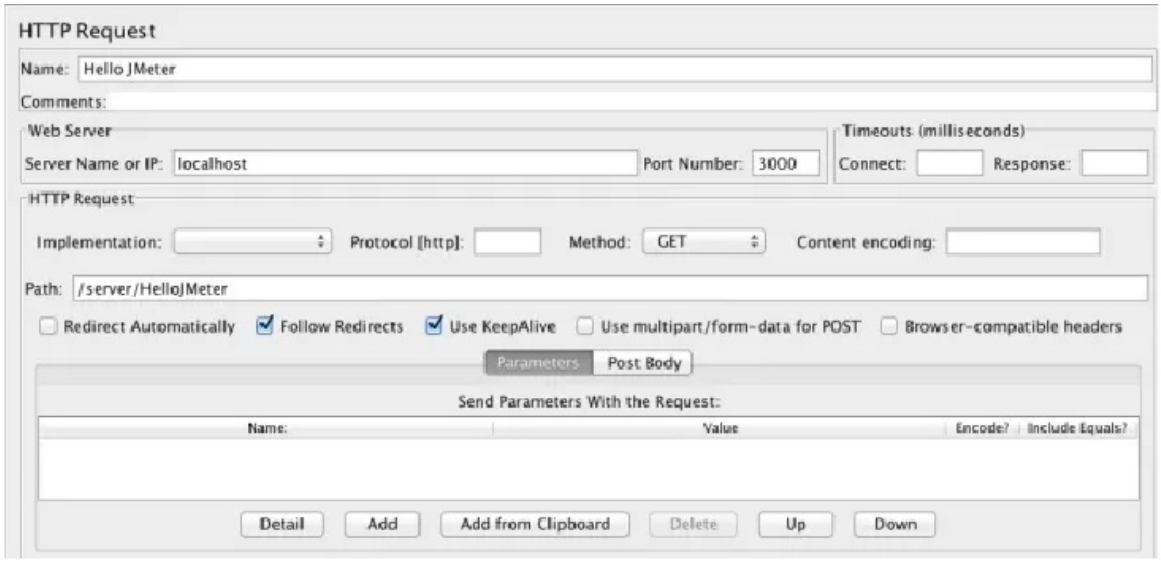
Passo 3: Adicionando um Sampler do tipo HTTP Request ao nosso Thread Group

Nesse momento peço que você tenha bastante discernimento para escolher um site seu* ou algum site que tenha autorização para emular carga contra ele. Lembre-se que podem existir implicações legais contra tentativa de ataques contra espaços virtuais e uma execução de ferramentas de repetição como o JMeter pode ser considerado uma tentativa de DoS (Denial of Service) e é facilmente identificado através de logs do servidor com informações do attacker.

Para evitar esse tipo de problema, eu sugiro que você use um apache local ou uma aplicação simples no seu computador local, e use esse site como ataque, é o que eu vou fazer nesse tutorial.

* Mesmo que o site seja seu, consulte o seu provedor para ter certeza que esse tipo de ataque é permitido. Caso use um servidor compartilhado possivelmente vai gerar problemas para outros usuários.

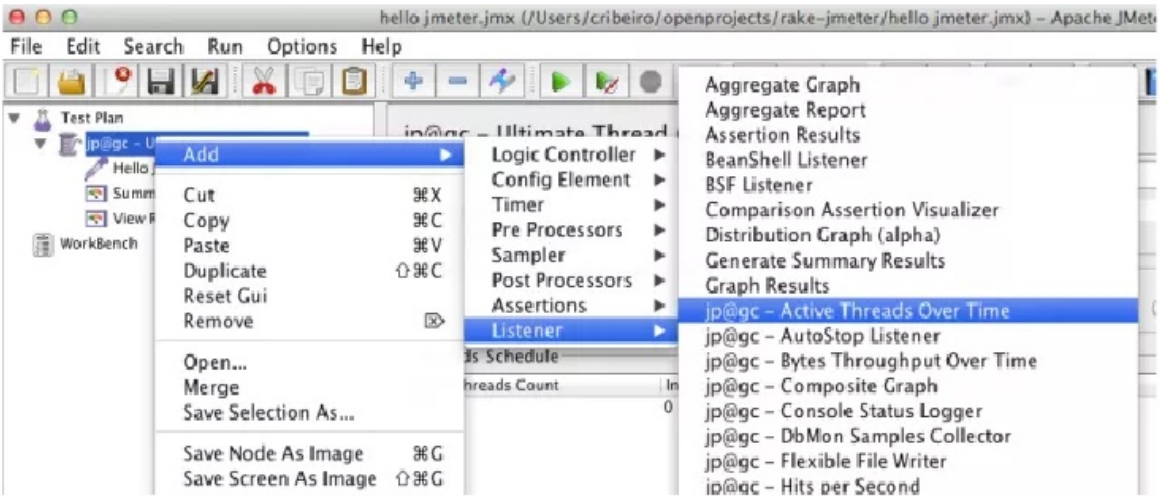
Caso não tenha ideia do que usar e tenha conhecimento em Node.JS, você pode usar uma aplicação que eu criei para testar framewoks de perofrmance chamada Hitz (https://github.com/camiloribeiro/hitz). Ela é bem simples de instalar e executar, as instruções estão na própria página da ferramenta.



Passo 4: Adicionar as informações de Server, Port Number, Path, Method e Prameters se necessário

Agora já estamos habilitados a rodar o teste, mas do jeito que está, não vamos saber o que está acontecendo. Para acompanhar o teste, vamos adicionar listeners a vontade, mas eu recomendo que pelo menos o "Summary Report", "Active Threads Over Time" e o "View Results in a Tree". Para isso clique com o botão direito no Thread Group e então "Add">"Listener".

Dica: Todos os elementos do JMeter, como os listeners, config element, timers, assertions, pre e post processors podem ser incluídos no mesmo nível dos samplers, ou abaixo de um deles individualmente. Quando eles estão no mesmo nível eles executam sua operação para todos os samplers e quando estão abaixo de um sampler eles só executam sua operação para esse sampler. No nosso exemplo com um sampler só não faz diferença, mas você pode criar novos samplers e brincar com esses elementos, assim vai entendendo como criar diferentes cenários de teste.



Passo 5: Adicionar os listeners que geram os relatórios para acompanharmos o teste

Agora vamos iniciar os nossos testes de performance. Para isso basta usar o atalho "control r" ou ir no menu "Run" > "Start".

Passo 6: Agora você pode observar os resultados nos relatórios que adicionou.

Caso esteja usando o Hitz que eu indiquei mais cedo, você vai observar que o número de requests vai aumentando progressivamente com os hits que o JMeter lança contra o servidor no seu localhost. Você vai poder comparar também os resultados

Resposta do Hitz aos hits provocados pelo JMeter em tempo real

Concluindo

O JMeter é uma ferramenta fantástica, e esse tutorial só te mostrou o básico das suas configurações e execução. O JMeter tem um poder extraordinário quando usado em conjunto com outras ferramentas ou linguagens de programação, como por exemplo shell script, ruby, virtualizadores, servidores de integração contínua, apis de cloud computing, etc.

Nas próximas lições você vai aprender mais sobre como escrever scripts de performance usando as ferramentas que conheceu hoje, como executar o JMeter de mais de uma máquina ao mesmo tempo e como rodar o JMeter usando linhas de comando, mas não se desespere! **É muito mais simples do que você imagina!** Além disso vamos integrar o JMeter com o Jenkins e mostrar como você pode manter seus testes de performance rodando todo o tempo e ser avisado quando alguma coisa está ficando diferente.

Espero que esse tutorial para escrever um pequeno teste de performance com JMeter tenha sido útil e simples de acompanhar para você. Caso tenha algum feedback, sugestão de melhoria, correção ou informação que poderia estar aqui, entre em contato comigo que vou melhorar/corrigir/adicionar com prazer.

Caso tenha gostado desse tutorial e queira ver o novo tutorial que explica como criar testes com JMeter, compartilhe e vamos avançar para o próximo nível

Referencias

[1] http://en.wikipedia.org/wiki/Software_performance_testing (acessado em 12 de julho de 2013)

[2] http://msdn.microsoft.com/en-us/library/bb924357.aspx (acessado em 12 de julho de 2013)

[3] http://www.amazon.com/The-Art-Application-Performance-Testing/dp/0596520662 (acessado em 12 de julho de 2013)

[4] http://br.groups.yahoo.com/group/DFTestes/

[5] http://jmeter.apache.org/download_jmeter.cgi

[7] http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html

[8] http://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools/

[9] http://newrelic.com/

[10] http://jmeter-plugins.org/wiki/PerfMonAgent/?utm_source=jpgc&utm_medium=link&utm_campaign=PerfMonAgent

[11] http://wiki.apache.org/jmeter/

[12] http://en.wikipedia.org/wiki/Jakarta_Project

[13] https://github.com/camiloribeiro/rake-jmeter

Camilo Ribeiro

Test Engineer at Klarna

Desenvolvedor, testador e agilista desde 2005, atualmente trabalhando na Suécia.