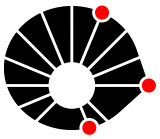

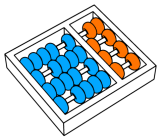

Testes de performance em sistemas web executados pela Java Virtual Machine

L. A. Ramalho *D. P. Mendes*

Relatório Técnico - IC-PFG-20-19
Projeto Final de Graduação
2020 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Testes de performance em sistemas web executados pela Java Virtual Machine

Lucas Alfonso Ramalho* Danilo Perina Mendes*

Resumo

O objetivo deste projeto é comparar a performance e escalabilidade de dois serviços web que utilizam linguagens executadas na Java Virtual Machine: Java e Kotlin. Iremos focar em uma situação na qual os dois sistemas irão realizar um algoritmo de execução paralela quando receber uma requisição HTTP. O sistema feito em Java utilizará *threads*, já o sistema em Kotlin Coroutines.

Para realizar a comparação, inicialmente fizemos uma análise teórica para entender como as aplicações deveriam se comportar em situações de alto *throughput*. Concluímos que o serviço que utiliza Coroutines deveria apresentar melhores resultados devido ao fato de seu algoritmo de execução paralela consumir menos memória e processamento.

Então colocamos os dois sistemas sob testes de carga utilizando diversas configurações diferentes, e comparamos os resultados para avaliar o comportamento na prática. O resultado foi que a aplicação Kotlin performou de maneira substancialmente melhor, comprovando o que havíamos entendido na análise teórica.

1 Introdução

1.1 Concorrência e paralelismo no desenvolvimento web

Nas primeiras tentativas de otimização do tempo de computação, não havia verdadeiro paralelismo com mais de um núcleo de processador. Isso exigiu criatividade nos primórdios da programação, resultando no desenvolvimento da ideia de *threads*. Essa criação possibilitou a programação concorrente, onde diferentes *threads* podem se revezar no uso dos recursos computacionais, alternando entre si de modo a otimizar o tempo de utilização deles.

Com o avanço dos meios físicos de computação, novas possibilidades de programação se abriram, atacando a dificuldade de minimizar tempo de comuptação. O desenvolvimento de processadores com mais de um núcleo possibilitou paralelismo verdadeiro, uma vez que duas *threads* diferentes agora podem utilizar processadores separados sem que haja concorrência entre elas. Essa inovação resultou em melhorias significativas no tempo de processamento de tarefas.

Mais recentemente, novas abstrações para lidar com concorrência e paralelismo foram se consolidando, como as corrotinas, que serão abordadas em outra seção deste relatório. Uma das linguagens que utilizam essa funcionalidade é o Kotlin, que teve seu início em

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

2016. "*Kotlin is an open-source statically typed programming language that targets the JVM, Android, JavaScript and Native. It's developed by JetBrains. The project started in 2010 and was open source from very early on. The first official 1.0 release was in February 2016*".[1]

No caso de sistemas web, com as grandes quantidades de acessos simultâneos recentes, os servidores tiveram que se reinventar para conseguir atender de forma satisfatória todos os *requests* de clientes concorrentes. Apenas o uso de *threads* não soluciona esse problema de maneira razoável. Portanto, foram criadas soluções para lidar com esse problema e uma das que mais causou impacto foi o sistema *NGINX*, que tem como base o uso de processos e chamadas de *I/O* assíncrono. Desde seu lançamento (2004) ainda é um dos sistemas mais rápidos e escaláveis que temos até hoje. Ele funciona com um processo *master* e vários processos *worker*. Cada *worker* consegue gerenciar milhares de *sockets* de conexão, implementando um *loop* de eventos que faz repetidamente *syscalls* não bloqueantes de entrada e saída, checando na repetição se chega um evento de término para daí executar o que precisa e devolver o resultado. Isso acontece com cada *worker* utilizando apenas uma *thread*, o que evita o uso exacerbado de recursos do sistema.

1.2 Motivação do projeto

Com o aumento exponencial de usuários de serviços *web*, um dos desafios encontrados pelos grandes servidores é o de atender números exorbitantes de requisições ao mesmo tempo. Com a ainda recente funcionalidade de *Coroutines* introduzida ao Kotlin, a comparação de sua performance com as já amplamente utilizadas *threads* acaba tornando-se inevitável. A motivação deste trabalho é para que haja um avanço na discussão de qual desses métodos é mais eficiente quando se trata dos grandes números, uma vez que nestes casos extremos uma pequena diferença no desempenho pode acabar custando muito mais barato.

2 Objetivo

O objetivo deste relatório é realizar um estudo teórico e prático sobre o comportamento dos sistemas web desenvolvidos nas linguagens Kotlin e Java, buscando entender quais são as diferenças entre os algoritmos de execução paralela dos mesmos.

Vamos analisar o comportamento dos serviços web em situações de estresse semelhantes às encontradas na indústria e avaliar quais foram as diferenças entre o esperado pela teoria e a prática. Deste modo, iremos aplicar o conhecimento adquirido ao longo do curso de Engenharia da Computação nos tópicos de sistemas operacionais, engenharia de software, redes, orientação a objetos e análise de algoritmos.

3 Base Teórica

Neste capítulo iremos abordar e detalhar os conceitos das ferramentas e as teorias que vamos utilizar como base neste projeto, além de buscar entender qual será o comportamento teórico dos sistemas durante o experimento.

3.1 Java

Java é uma linguagem de programação orientada a objeto que inicialmente foi desenvolvida para televisão interativa. Sua utilização veio a se provar avançada demais para as tecnologias da época, tendo sua primeira implementação lançada em 1996. Teve como princípios de desenvolvimento a simplicidade, robustez, alta performance, entre outras.

Um desses objetivos dos desenvolvedores de Java era que a linguagem tivesse portabilidade universal, ou seja, que pudesse ser utilizada em qualquer máquina sem detrimento qualquer. Para alcançar essa missão, foi desenvolvida a JVM (*Java Virtual Machine*), uma máquina virtual que permite a utilização da linguagem em qualquer sistema operacional. Sua versão mais primitiva foi lançada em 2006, até que em 2007 foi lançada por completo.

Java foi por muito tempo a linguagem mais utilizada em serviços web, uma vez que há bibliotecas e *frameworks* de suporte para desenvolver esse tipo de serviço. Essas funcionalidades adicionadas à portabilidade universal resultaram num amplo uso por parte de muitos desenvolvedores. Até hoje é muito utilizada, reforçado pelo fato de que há constante suporte e novas atualizações lançadas com certa frequência.

3.2 Kotlin

Kotlin é uma linguagem tanto orientada a objetos quanto funcional que teve seu desenvolvimento iniciado em 2010 para ter sua primeira versão estável lançada em 2016. Inicialmente foi pensada para ser uma nova linguagem para JVM, uma vez que muitos serviços web já utilizam essa máquina virtual, e que fosse tão rapidamente compilável quanto Java, gozando de funcionalidades que facilitassem seu uso e otimização.

Na versão v1.3 de 2018, Kotlin introduziu as chamadas *Coroutines*, que foram um ponto importante na decisão atual de muitos desenvolvedores para começar a desenvolver seus novos serviços utilizando esta linguagem. A ideia de corrotinas não foi uma invenção de Kotlin, mas ela trouxe uma nova possibilidade para a JVM com o suporte nativo a essa funcionalidade, facilitando seu uso.

Corrotinas tem como ideia básica a "imitação"de uma *thread*; são como se fossem *threads* mais leves que permitem programação assíncrona e concorrência entre trechos de código. Elas tornaram Kotlin uma linguagem bastante visada no mercado *web*, devido ao grande aumento no número de usuários, o que resulta em uma escalada das requisições simultâneas nos serviços atuais.

3.3 Java Virtual Machine

A JVM é uma máquina virtual que providencia um ambiente para rodar um código Java e aplicações, e que faz parte da JRE (Java Runtime Environment). O compilador de Java, diferentemente de outras linguagems, compila o código para a JVM e não para o sistema operacional. Primeiro o código Java é compilado e é gerado um Java *bytecode*. A JVM, por sua vez, transforma esse *bytecode* em linguagem de máquina a depender do sistema operacional sobre o qual ela está rodando.

Quando um arquivo *.java* é compilado, é gerado um arquivo *.class* de mesmo nome. Ao ser executado, esse arquivo passa por diversos passos, realizados pela JVM. A funcionalidade

da JVM pode ser resumida a esses passos, demonstrados na **Figura 1**:

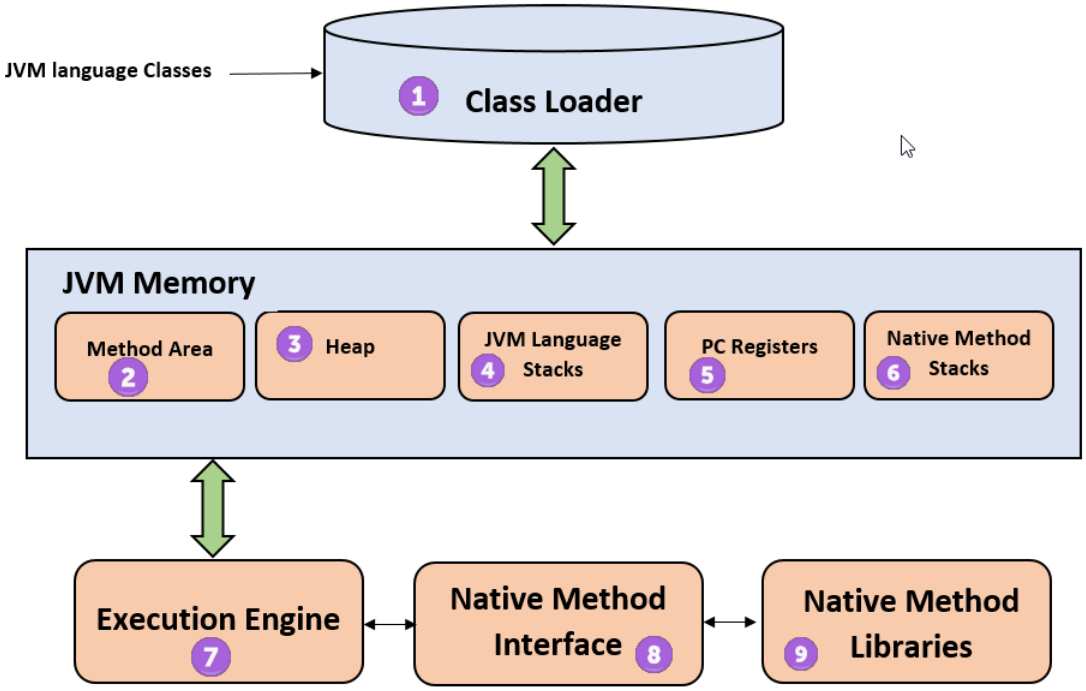

Figura 1 - Arquitetura da JVM

O *Class Loader* tem 3 principais funções: *Loading* (ler o arquivo, gerar o binário correspondente e salvá-lo na *Method Area*), *Linking* (faz a verificação e preparação, como por exemplo alocação de memória) e *Initialization* (inicialização das variáveis).

A *JVM Memory* é onde ficam guardadas as informações do arquivo. Na *Method Area* ficam as guardadas informações sobre variáveis e métodos. *Heap* é onde ficam salvos os objetos, e é uma área compartilhada por todas as *threads*. Cada *thread* tem a sua própria *Language Stack* onde ficam guardadas as variáveis locais, e essa informação não é compartilhada com outras *threads*; quando uma *thread* acaba, sua *Stack* é destruída. O *PC register* guarda o endereço da instrução sendo executada e não é compartilhado com outras *threads*. O *Native Mother Stacks* guarda as instruções de métodos nativos.

O *Execution Engine* lê o *bytecode* linha por linha e executa as instruções. O *Native Method Interface* habilita a JVM a chamar as bibliotecas das *Native Method Libraries* (uma coleção de bibliotecas nativas).

3.3.1 Spring framework

O Spring é um *framework open source* para a plataforma Java criado por Rod Johnson em Outubro de 2002 e que hoje é mantido por um time de quase 100 colaboradores[9]. O objetivo da ferramenta é facilitar o desenvolvimento de qualquer tipo de aplicação que é

executada na JVM. Para isto, ele oferece um grande conjunto de módulos com funcionalidades que são tipicamente utilizadas na indústria e que possuem grande flexibilidade de configuração. A partir da versão 5.0, o *framework* também passou a suportar Kotlin. [10]

O que diferencia o Spring de outros *frameworks* presentes no mercado é a grande variedade e a qualidade dos módulos, cujo desenvolvimento é feito visando os padrões de qualidade esperados de software presentes na indústria, como segurança e performance. Nesse sentido, é uma ferramenta que permite o desenvolvimento de aplicações em menor tempo e com qualidade.

Embora cada módulo seja independente dos outros, todos eles obedecem a alguns padrões de arquitetura que são comuns, dos quais se destaca o princípio de inversão de controle, também conhecido como injeção de dependência. Este padrão corresponde a um processo onde os objetos definem suas dependências apenas através de propriedades das mesmas, sem instanciá-las. O próprio Spring é responsável então por instanciar as propriedades e relacioná-las ao objeto. Este processo é fundamentalmente o oposto do próprio objeto controlar o comportamento das suas propriedades, daí o nome "Inversão de controle". O grande valor deste padrão está no fato de que o *framework* passa a ser o responsável em gerenciar o comportamento dos objetos dentro da aplicação, prevenindo problemas de desempenho como por exemplo excesso de memória no *heap* da JVM causado por má instanciação de objetos.

No nosso caso, o principal módulo utilizado pelos dois serviços que vamos testar foi o Spring Boot. Ele oferece a maioria dos componentes do Spring necessários para aplicações em geral de maneira pré-configurada, tornando possível termos uma aplicação sendo executada com o esforço mínimo de configuração e implantação. Para rodar uma aplicação baseada neste módulo, basta ter o Java na versão compatível instalada no sistema operacional. É possível gerar um *template* de um serviço web baseado neste módulo através do site https://start.spring.io/.

O *template* possui uma classe chamada NomeDoServico.Java que, se for executada, irá executar todos os passos necessários para deixar o servidor online e recebendo requisições. Veremos no próximo capítulo como isso foi feito nos dois serviços que vamos testar.

3.4 Protocolo HTTP

HTTP é um protocolo da camada de aplicação utilizado em sistemas *web* para fazer requisições de recursos para servidores, enviado sobre o protocolo da camada de transporte TCP. O cliente (agente-usuário), normalmente um navegador *web*, faz uma requisição (via HTTP) para um servidor de algum recurso (por exemplo uma página HTML). O HTTP é um protocolo sem estado, apenas com sessões. Para guardar o estado muitos serviços utilizam os *cookies*, conseguindo assim guardar informações da conexão.

O fluxo HTTP começa com o cliente abrindo uma conexão TCP com o servidor, geralmente na porta 80. Então o cliente envia uma mensagem HTTP, como por exemplo a demonstrada na **Figura 2**:

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Figura 2 - Exemplo de uma mensagem do tipo GET

Depois o cliente lê a resposta do servidor, como na **Figura 3**:

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the requested web pa
```

Figura 3 - Resposta HTTP

E por fim o cliente ou fecha a conexão ou reutiliza para requisições futuras.

O protocolo HTTP tem um conjunto de métodos, chamados de Verbos, cuja função é indicar qual ação deverá ser executada. Os mais utilizados são POST, usado para indicar mudanças no lado do servidor, PUT, substitui todas as atuais representações do recurso de destino pela carga de dados da requisição, DELETE, que remove um recurso específico e o método GET. O método que foi utilizado nesse projeto, é o GET, que é um método do HTTP usado para fazer uma requisição de recursos, como por exemplo: o cliente (navegador *web*) faz uma requisição pelo endereço *url* ao servidor e o servidor responde com uma página HTML, como podemos ver mais detalhadamente na **Figura 4**:

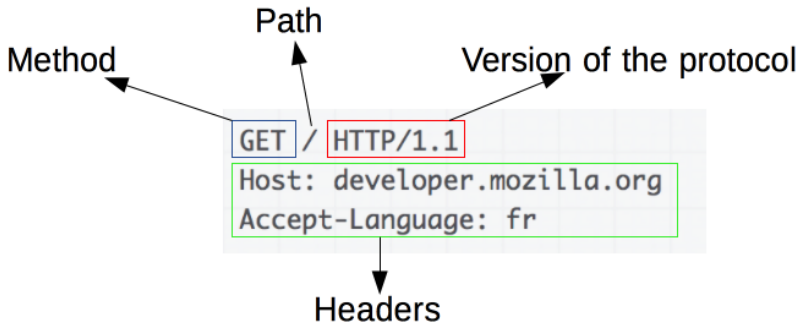

Figura 4 - Exemplo de mensagem do tipo GET

Vamos utilizar em um fluxo nesse projeto duas requisições GET; um é o que a ferramenta de teste de performance faz para a aplicação e o outro é o que o serviço faz para o servidor central, seja por uma *thread* separada ou por uma Coroutine. Os testes realizados neste projeto foram desenvolvidos de modo que a comparação seja feita da melhor forma possível, tendo como objetivo a menor variação de tempo entre as requisições. Para que isso ocorra, o ideal é que o cliente e o servidor estejam o mais próximo possível; por isso colocamos o *JMeter* na mesma rede dos serviços a serem testados.

3.5 Concorrência e paralelismo em sistemas web

O primeiro passo para entendermos este assunto é saber diferenciar os conceitos de concorrência e paralelismo no contexto de sistemas computacionais. O melhor jeito de sintetizar essa diferença é através da famosa fala do criador da linguagem Go, Rob Pike: "Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."[11]

Em outras palavras, concorrência está ligado a como o sistema lida com a execução de forma sequencial de um conjunto de tarefas independentes em um mesmo intervalo de tempo. Quando se discute concorrência em sistemas web, estamos tratando de um problema de escala exponencial, principalmente nos dias atuais em que bilhões de pessoas possuem acesso à rede, e uma simples promoção ou festas de final de ano podem ser motivo para que haja milhões de requisições simultâneas para um determinado sistema. Nessa escala, pequenas diferenças de desempenho podem escalar e se tornarem colossais.

Dentro do contexto do projeto, vamos estudar como um serviço web Java e um Kotlin lidam com esse cenário de múltiplas requisições simultâneas, onde existe o acesso concorrente aos recursos de hardware, em especial a memória e o processamento.

Já o conceito de paralelismo significa um sistema executar mais de uma atividade simultaneamente. Ou seja, paralelismo é um dos recursos que podemos utilizar para se lidar com situações de concorrência.

O número máximo de atividades que qualquer sistema computacional pode executar simultaneamente é sempre igual a quantidade de núcleos de processamento disponíveis para

o mesmo. O que irá variar de um ambiente para outro é como é feito o processo de alocação de recursos para a execução de uma determinada atividade.

Uma estratégia utilizada pelos principais sistemas operacionais presentes no mercado é dividir as aplicações que estão sendo executadas em processos. Cada processo tem um espaço de memória exclusivo e podemos executar dois processos simultaneamente em núcleos separados. Contudo, essa estratégia não é válida para o nosso caso, dado que necessitamos compartilhar memória entre o que está sendo executado paralelamente nos cores do processador.

Tendo em vista esse problema, foi criado o conceito de *thread*. A ideia é dividir um processo em pedaços - chamados de *threads* - que podem ser executados simultaneamente em núcleos diferentes. Nesse sentido, cada uma tem seu próprio contexto de hardware, porém compartilha o contexto de software e memória com as outras do mesmo processo.

Essa estratégia gera alguns desafios de programação, já que é possível que duas ou mais *threads* tentem e manipular ou acessar o mesmo endereço de memória ao mesmo tempo. Esses desafios são chamados de *race conditions*. São necessários mecanismos de controle para lidar com essa situação.

É possível encontrar em algumas literaturas o termo *threads* como "processos leves". Isso se dá também porquê a mudança de contexto de execução de uma *thread* no processador para outra necessita de menos recursos que a mudança de contexto de processos.

Os principais sistemas operacionais modernos possuem suas próprias implementações de processos e *threads*. A JVM, por sua vez, possui uma camada de abstração em cima dessas implementações para que seja possível utilizar este recurso através do Java e do Kotlin. Nas duas próximas subseções iremos detalhar como funciona essa abstração para cada linguagem, quais as vantagens e desvantagens, e como irá funcionar dentro do cenário prático.

3.5.1 Java Threads

O recurso mais baixo nível oferecido pela Oracle para lidar com paralelismo é a classe Java chamada Thread.[12] Essa classe possui métodos que permitem a criação e gerenciamento das mesmas dentro de um programa Java.

Porém, dado que essa interface não possui nenhum mecanismo ou algoritmo para lidar com as *race conditions* que podem aparecer, foram criadas algumas bibliotecas que possuem abstrações que ajudam a evitar esses problemas.

A biblioteca que utilizamos neste projeto foi também desenvolvida pela Oracle e se chama CompletableFuture. Essa biblioteca foi introduzida na versão 8 do Java e é uma evolução de uma biblioteca antiga que se chamava Future.[13] O objetivo dela é prover uma interface de desenvolvimento paralelo, onde é necessário apenas deixar explícito o código a ser executado em outra *thread* e qual é o formato da variável de retorno desse código. A única parte da memória que será compartilhada é essa variável de retorno. No caso do serviço que vamos testar, a rotina que será executada paralelamente é um HTTP GET para um terceiro serviço e a variável de retorno é o conteúdo que é retornado na requisição.

Dentro do contexto de um serviço *backend* Java ou Kotlin feito utilizando Spring, cada HTTP *request* que o mesmo recebe é responsável pela criação de uma *thread* nova dentro

do processo principal da aplicação. O conjunto de *threads* que estão pré-instanciadas para armazenar requisições é chamado de Thread Pool.

Vamos imaginar que enviamos 100 requisições simultâneas para um *backend* feito em Java. Se a *pool* do sistema estiver configurada para ter 1000 *threads*, 100 delas armazenarão as requisições, que serão executadas em ordem de chegada da requisição. Porém, se a *pool* estiver configurada para ter 10 *threads*, 90 dessas requisições não serão executadas e retornarão erro.

O número total de *threads* que um *pool* pode ter é determinado pela memória disponível para a aplicação, e deve ser balanceado com outras características do sistema como por exemplo capacidade de processamento.

Nesse sentido, dentro do fluxo do nosso projeto, quando o serviço Java receber uma requisição HTTP, serão abertas duas *threads*: uma para a execução da requisição em si e outra através da biblioteca CompletableFuture que irá executar uma nova requisição HTTP para um terceiro serviço.

Já quando o serviço Kotlin receber uma requisição, será aberta uma thread para a execução da requisição em si, porém será utilizada uma Coroutine para realizar a chamada HTTP para o terceiro serviço. Iremos explicar o comportamento da *coroutine* abaixo.

3.5.2 Kotlin Coroutines

Conceitualmente, as *Coroutines* são como *threads*; elas executam unidades de tarefa concorrentemente. com a diferença de que não são obrigatoriamente ligadas a uma *thread* específica. Elas podem ter a sua execução suspensa e ter a sua atividade retomada em um momento futuro, possivelmente em outra Thread.

As Coroutines também são muito mais leves, porque enquanto cada *thread* tem sua própria pilha de dados, as corrotinas compartilham memória, o que torna elas muito mais eficientes do ponto de vista de uso de recursos.

No caso do serviço Kotlin que desenvolvemos, quando a requisição HTTP é feita, fazemos uma chamada de suspensão, liberando a *thread* principal para ser utilizada por outra requisição enquanto não recebemos a resposta. Assim que temos o retorno, a atividade é retomada na *thread* principal. A **Figura 5** abaixo representa este fluxo:

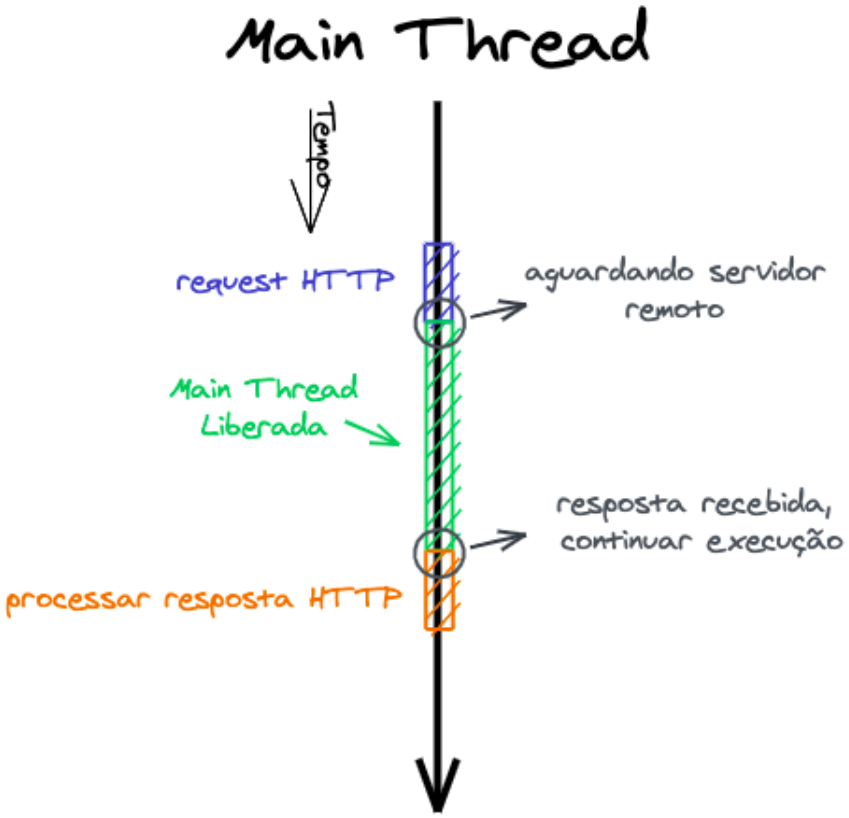

Figura 5 - Fluxo de uma Coroutine

3.6 Testes Automatizados

Para garantir a qualidade e funcionalidade de um programa, uma prática fundamental é a de testar o código, a fim de encontrar falhas no sistema e verificar se ele funciona como deveria. Testes manuais em geral são muito trabalhosos e pouco eficientes, portanto uma chave para ter um código amplamente testado são os testes automatizados. Existem inúmeros tipos de testes, como por exemplo testes unitários e de interface de usuário.

Mike Cohn cunhou em seu livro *Succeeding with Agile* o termo Pirâmide de Testes. Na base da pirâmide ficam os testes mais básicos, rápidos e mais fundamentais, como os testes unitários. Já no topo da pirâmide, ficam os testes mais complexos, completos e de alto nível, que tem como objetivo testar o sistema como um todo, integrando diferentes funcionalidades no mesmo teste.

Neste projeto, o tipo de teste que iremos desenvolver será o de performance. É um teste de mais alto nível (topo da pirâmide), que visa comparar funcionalidades mais complexas e que não são as mais "essenciais"para o funcionamento de um sistema web. É um tipo de teste mais voltado para melhorias *high-end*, de forma que o sistema possa performar melhor

em situações muito específicas (situações de alto nível de *stress*).

3.6.1 Teste de performance

Neste projeto, os testes de performance (ou desempenho) que serão realizados tem como foco avaliar o tempo de resposta de múltiplas requisições, a velocidade de processamento e a quantidade de recurso de hardware exigido.

Entre os testes de desempenho, existem alguns subtipos, como por exemplo o teste de resiliência, que tem como objetivo determinar como o código reage a longos períodos de uso, o teste de carga, que serve para verificar como o sistema reage a um número grande de usuários, e o teste de *stress*, parecido com o teste de carga, mas tem o objetivo de esticar ao máximo os limites de requisições e capacidade. Neste projeto, o teste feito é mais parecido com um teste de carga, onde iremos fazer um grande número de requisições simultâneas, mas sem o objetivo de "quebrar"o código como num teste de *stress*.

3.6.2 Métricas de performance

Nos testes de performance algumas métricas serão utilizadas como parâmetro de comparação entre os serviços. Elas serão ligados uso de recursos de hardware, quantidade de erros e tempo médio de resposta das requisições. Todas esses dados permitem gerar estatísticas que darão mais clareza onde cada tipo de serviço tem mais dificuldades e problemas.

A variância do tempo médio de resposta é uma das estatísticas que iremos calcular para cada serviço. Dessa maneira, poderemos ter uma noção melhor da consistência desse tempo médio e checar se ele há muita discrepância entre os valores, para conseguir chegar a uma conclusão mais apropriada. O cálculo se dá pela fórmula:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n-1}$$

Onde n é o número total de requisições feitas, y_i é o tempo de resposta de cada requisição individual e $\bar{y}$ é a média aritmética de todos os y_i.

Outro conjunto de estatísticas que será levado em consideração é o percentil. Essa estratégia consiste em dividir todo o conjunto experimental em 100 blocos que possuem tempo de resposta crescente. A partir daí, o n-ésimo percentil Pn é o valor x (xn) que corresponde à frequência cumulativa de N .n/100, onde N é o número total de requisições feitas. Então, por exemplo, o 90 percentil corresponde ao maior valor de tempo de resposta dos 90% menores valores disponíveis. O JMeter disponibiliza o 90, 95 e 99 percentil da amostragem de teste automaticamente, o que nos permite ter uma métrica de comparação que elimina casos de borda.

Outro dado que iremos utilizar é o APDEX (Application Performance Index). Essa classificação é um padrão aberto criado por uma série de empresas de tecnologia para avaliar o desempenho de aplicações de software. Seu propósito é converter as métricas gerais de performance em um valor que possa dar um entendimento do quanto a aplicação está dentro das expectativas dos seus usuários. Este valor é calculado pelo JMeter durante a execução

de cada teste e leva em consideração que a expectativa do usuário é conseguir realizar uma requisição HTTP GET em menos de meio segundo. A fórmula da conta é a seguinte:

$$APDEX = \frac{SatisfiedCount + 0.5 * ToleratingCount}{TotalCount}$$

3.7 Conclusão teórica

Tendo em consideração os pontos levantados neste capítulo, concluímos que é esperado que o sistema Kotlin consiga escalar e receber mais requisições concorrentes que o serviço Java, já que a tendência é que tenha mais memória disponível para armazenar requisições no Thread Pool. Porém, não necessariamente irá existir diferença no tempo de requisição médio entre os dois sistemas.

4 Desenvolvimento

O objetivo deste capítulo é mostrar como foi feito o desenvolvimento da parte prática do projeto e como funcionam os sistemas criados.

4.1 Arquitetura geral

Para o desenvolvimento da parte prática de coleta de dados de desempenho foi prototipado um sistema que seja capaz de testar e gerar métricas em relação ao desempenho de dois serviços web semelhantes em uma situação de alta carga de trabalho.

Dentro desse fluxo de teste precisamos de no mínimo dois componentes atuando: um que realiza requisições HTTP com alta frequência e retorna métricas dos resultados e outro que seria o sistema que recebe as requisições e irá ser avaliado.

No caso do primeiro, como não se trata de interesse do projeto o desenvolvimento das ferramentas de teste, buscamos por opções *open source* que são consolidadas e utilizadas na indústria. Por isso, acabamos utilizando o JMeter. Detalhamos a ferramenta e suas características próxima seção.

No caso do segundo componente, o serviço web que vamos testar, temos que desenvolver dois projetos: um em Java e outro em Kotlin. Os dois vão ser executados na JVM e devem ser capazes de receber requisições web e de criar uma segunda *thread/coroutine* dentro de cada requisição recebida. Ambos foram desenvolvidos utilizando o Spring como *framework* base.

Cada chamada HTTP atendida pelo sistema deverá criar uma *thread/coroutine* que por sua vez executa uma determinada rotina. Este trecho que será executado separadamente precisa realizar algum tipo de tarefa que demore algum tempo fixo para executar, de modo que podemos comparar a performance em um cenário parecido com a realidade. Nesse cenário, haviam duas possibilidades de escolha para a tal tarefa: realizar uma operação de I/O ou realizar uma operação custosa - por exemplo uma iteração que realize milhões de contas. Como o ideal para a precisão do experimento é manter os dois ambientes de teste o mais parecido possível, optamos por atribuir a esta tarefa a realização de uma nova chamada HTTP para um terceiro que chamamos de servidor central, de modo que a tarefa

a ser executada separadamente terá um tempo de execução muito próximo nos dois casos. Na próxima subseção explica-se a escolha do servidor central.

As duas imagens abaixo (**Figura 6** e **Figura 7**) esquematizam a arquitetura descrita:

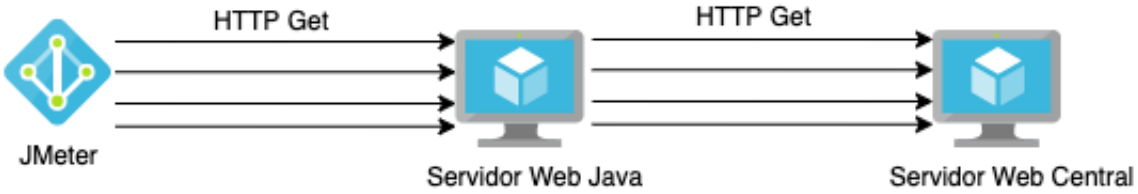

Figura 6 - Fluxo de requisições do serviço Java

Figura 7 - Fluxo de requisições do serviço Kotlin

4.2 Componentes

Abaixo vamos detalhar a implementação e o funcionamento dos componentes descritos na arquitetura geral. Todo o código do projeto está contido em um repositório central Git que está disponível em https://gitlab.ic.unicamp.br/ra135434/pfg-jvm-testing. Git é uma ferramenta *open source* utilizada para controle de versões de código. Mais informações em https://git-scm.com/. Ela foi utilizada pelo grupo para auxiliar no desenvolvimento conjunto do sistema. Criamos um repositório central onde sempre se encontra a versão mais atual do código, assim, temos controle de todas as modificações, quando elas foram feitas e quem foi o responsável.

No arquivo README.md dentro pasta raiz do projeto pode-se encontrar instruções sobre como executar cada componente localmente.

4.2.1 Servidor Web Java

O servidor *backend* foi desenvolvido utilizando o *framework* Spring na versão 5 como base. Foi utilizado como base um código gerado automaticamente pelo https://start.spring.io/. Optamos utilizar como configuração inicial as seguintes características: projeto Maven, Spring Boot versão 2.3.3 e versão 11 do Java.

Maven é uma ferramenta de gerenciamento de projeto desenvolvida pela Apache. A sua característica principal é que conseguimos gerenciar o *build*, *imports* externos e documentação dentro de um mesmo documento. Escolhemos utilizar Maven por conta da

familiaridade no uso por parte dos integrantes do projeto. Esta ferramenta não tem impacto no desempenho final do sistema, apenas impacta no tempo de desenvolvimento e *deploy* da aplicação.

Dentro da pasta central do código do serviço destacam-se dois componentes: a pasta *src*, e o arquivo pom.xml. Este arquivo está em formato XML e é o que o Maven utiliza como fonte para controle do projeto. Nele podemos encontrar quais bibliotecas externas estão sendo utilizadas em qual versão.

Já na pasta *src/* encontra-se o código Java responsável por receber as requisições e realizar abertura de uma segunda *thread* que irá realizar uma chamada HTTP para um terceiro serviço. São duas classes que realizam este trabalho: MainController.java e CentralServiceClient.java. Dentro da classe MainController temos dois métodos que correspondem à um *endpoint* cada um. Abaixo a classe:

```
public class MainController {

    @Autowired
    private CentralServiceClient centralServiceClient;

    Logger logger = LoggerFactory.getLogger(MainController.class);

    @GetMapping("/")
    public String getCentralServiceResult() {
        logger.info("Get central service message incoming request");
        return centralServiceClient.getCentralServiceMessage();
    }

    @GetMapping("/async")
    public String getAsyncCentralServiceResult() {
        logger.info("Get ASYNC central service message incoming request");
        return centralServiceClient.getAsyncCentralServiceMessage();
    }

}
```

A anotação @GetMapping("/caminho") indica que o método abaixo dele será invocado quando uma requisição GET chegar no endereço `https://ip_do_servico:porta/caminho`. Na seção "hospedagem"detalharemos as propriedades de rede dos serviços.

Portanto, o getCentralServiceResult é ativado quando o serviço recebe uma chamada HTTP GET no caminho "/"e chama o método de mesmo nome da classe CentralServiceClient, que é responsável por realizar uma nova chamada HTTP bloqueante dentro da *thread* principal da requisição e retornar exatamente o que vier do mesmo.

Já o getAsyncCentralServiceResult escuta o caminho "/async", e invoca um outro método que realiza a chamada HTTP para o servidor em uma segunda *thread* e também retorna o resultado.

A ideia de ter dois fluxos, um no qual a chamada HTTP bloqueia a *thread* principal e outro que bloqueia uma outra *thread*, é para avaliarmos se o comportamento do serviço em

situação de alta carga será de acordo com o que prevemos. É esperado que o fluxo que apenas bloqueie a *thread* principal na realização da chamada HTTP se comporte de maneira mais performática que o segundo, pois teoricamente apenas bloquear a *thread* principal consome menos recursos (principalmente memória) que abrir uma *thread* extra e bloquear a mesma.

A classe CentralServiceClient, representada abaixo, é responsável por implementar os fluxos descritos acima. Aqui é válido notar que o método assíncrono utiliza a mesma implementação do síncrono, ou seja, os dois executam o mesmo código, a diferença está apenas no fato de que o assíncrono irá abrir uma segunda *thread* que será bloqueada enquanto a requisição HTTP para o servidor central é feita.

```
public class CentralServiceClient {

    public String getAsyncCentralServiceMessage(){
        try {
            CompletableFuture<String> future
                   = CompletableFuture.supplyAsync(() ->
                       getCentralServiceMessage());
            return future.get();
        } catch (Exception e) {
            e.printStackTrace();
            return "Error";
        }
    }

    public String getCentralServiceMessage(){
        try {
            URL url = new URL("http://172.20.0.59:8081");
            HttpURLConnection con = null;
            con = (HttpURLConnection) url.openConnection();
            con.setRequestMethod("GET");
            con.setConnectTimeout(5000);
            con.setReadTimeout(5000);

            BufferedReader in = new BufferedReader(
                   new InputStreamReader(con.getInputStream()));
            String inputLine;
            StringBuffer content = new StringBuffer();
            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }
            in.close();
            return content.toString();
        } catch (Exception e) {
            e.printStackTrace();
            return "Error";
        }
    }
}
```

4.2.2 Servidor Web Kotlin

O código do *backend* desenvolvido em Kotlin foi estruturado de maneira semelhante ao Java. Utilizamos o *framework* Spring como base para o serviço e criamos o *template* inicial através do start.spring.io.

Dentro do arquivo Maven pom.xml, presente na pasta raiz do projeto, podemos encontrar todas as dependências externas utilizadas no projeto. Destacam-se duas: org.jetbrains.kotlinx.kotlinx-coroutines-core e org.jetbrains.kotlinx.kotlinx-coroutines-reactor. Estes módulos vem de uma biblioteca chamada Kotlinx Coroutines desenvolvida pela Jetbrains [5], e possuem a implementação da criação de uma corrotina. Neste sentido, do mesmo jeito que utilizamos a CompletableFuture no projeto Java para facilitar a criação de uma *thread* nova, aqui estamos utilizando a Kotlinx Coroutines.

No caminho *src/main/kotlin/org/unicamp/pfg/kotlinservice* podemos encontrar as classes Kotlin que implementam os dois fluxos também presentes no serviço que vamos comparar a este. São duas classes, a MainController.kt e a CentralServiceClient.kt. A MainController.kt é responsável por mapear os dois *endpoints* para responder a chamadas HTTP GET nos caminhos "/"e "/async".

A CentralServiceClient.kt é responsável por implementar os dois fluxos, um deles realizando a requisição HTTP na *thread* principal da requisição e o outro dentro de uma Coroutine criada. O método getAsyncCentralServiceMessage() tem na assinatura o prefixo 'suspend', que indica à JVM que esse trecho deve ser executado numa corrotina.

Nesta classe, diferentemente do que acontece no serviço Java, o código dos fluxos de requisição HTTP para o servidor central não são os mesmos. Isso acontece porque foi necessário utilizar uma biblioteca que fosse compatível de ser utilizada dentro de uma Coroutine. Para tal, é preciso que a implementação tenha claro quais são os momentos de "pausa", ou seja, onde que o fluxo pode liberar o recurso do sistema para executar outros fluxos.

```
class CentralServiceClient {

    suspend fun getAsyncCentralServiceMessage(): String {
        val request = getWebClient()
                .get()
                .uri("/")
                .retrieve()
                .awaitBody<String>();

        return request;
    }

    private fun getWebClient(): WebClient {
        val tcpClient = TcpClient
                .create()
                .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000)
                .doOnConnected { connection: Connection ->
                    connection.addHandlerLast(ReadTimeoutHandler(5000,
                        TimeUnit.MILLISECONDS))
```

```
                connection.addHandlerLast(WriteTimeoutHandler(5000,
                    TimeUnit.MILLISECONDS))
            }

    return WebClient
            .builder()
            .baseUrl("http://localhost:8081")
            .defaultCookie("cookieKey", "cookieValue")
            .defaultHeader(HttpHeaders.CONTENT_TYPE,
                MediaType.APPLICATION_JSON_VALUE)
            .clientConnector(ReactorClientHttpConnector(HttpClient.from(tcpClient)))
            .defaultUriVariables(Collections.singletonMap("url",
                "http://localhost:8081"))
            .build();
}

fun getCentralServiceMessage(): String {
    return httpGet("http://localhost:8081");
}

private fun httpGet(myURL: String?): String {
    val inputStream: InputStream
    val result:String
    val url: URL = URL(myURL)
    val conn: HttpURLConnection = url.openConnection() as HttpURLConnection

    conn.connect()
    inputStream = conn.inputStream
    if(inputStream != null)
        result = convertInputStreamToString(inputStream)
    else
        result = "Erro"

    return result
}

private fun convertInputStreamToString(inputStream: InputStream): String {
    val reader = BufferedReader(inputStream.reader())
    val content = StringBuilder()
    try {
        var line = reader.readLine()
        while (line != null) {
            content.append(line)
            line = reader.readLine()
        }
        return content.toString();
    } finally {
        reader.close()
    }
```

```
    }
}
```

4.2.3 Ferramenta de teste de performance

Para a realização das comparações entre os serviços, precisamos de uma ferramenta que execute requisições HTTP em alta escala e retorne métricas sobre o desempenho do sistema que recebeu.

As métricas de performance que estamos procurando são tempo médio de resposta de uma chamada HTTP, quantidade de respostas num determinado intervalo e quantidade de erros recebidos.

Após pesquisa de referências, acabamos escolhendo o Apache JMeter™. Este é um software *opensource* feito totalmente em Java e foi desenvolvido para realizar testes de carga funcionais e medir performance[4]. Trata-se de um dos softwares mais utilizados tanto na academia quanto na industria.

Para instalar, basta apenas baixar uma pasta compressa no site oficial e descomprimir dentro de algum diretório da máquina que vai realizar as requisições. Para que a comparação de desempenho seja o mais fiel possível, é necessário que a máquina que está o JMeter seja diferente da que o servidor sendo testado no momento.

O modo de funcionamento da ferramenta gira em torno do conceito de plano de testes. Cada plano contém todas as configurações de execução e métricas de performance de determinado tipo de execução de carga. Podemos importar e exportar planos através de arquivos com a extensão .jmx. Todos os arquivos correspondentes aos testes feitos dentro deste projeto estão dentro do repositório Git apresentado no começo deste capítulo, na pasta */test-plans*.

Por conveniência, instalamos a ferramenta localmente para criar os planos, porém é necessário que os testes sejam executados numa máquina sem concorrência por recursos (memória e processamento), de modo que seja possível criar um alto número de requisições simultâneas. Portanto, instalamos também o JMeter em uma VM situada na mesma rede do serviço. Detalharemos melhor na próximas seções.

O JMeter possui uma interface gráfica que pode ser utilizada para a criação, edição e exportação de planos. Para abrir a GUI, basta executar o jmeter.bat presente na raiz da pasta da ferramenta. Na **Figura 8** abaixo podemos ver a tela inicial:

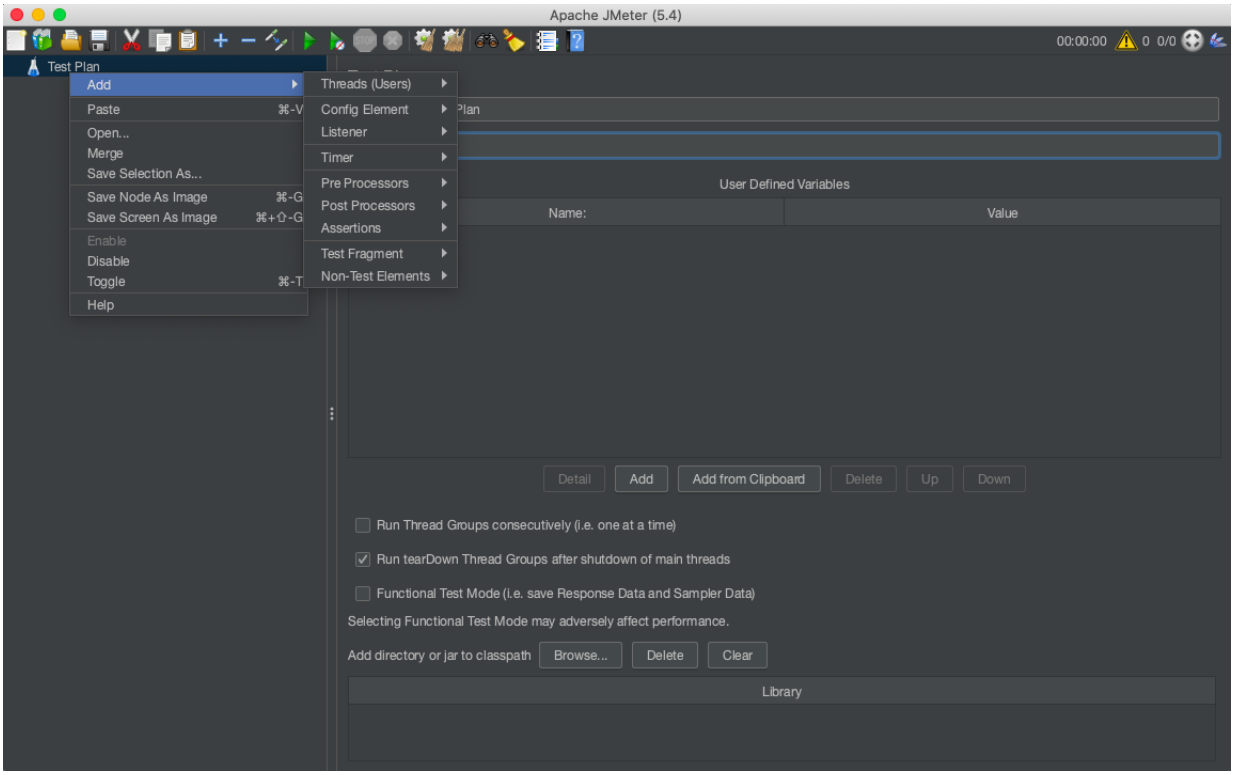

Figura 8 - Interface gráfica do *Test Plan*

Na subseção 4.4 iremos detalhar quais foram os planos criados e como foi feita a execução dos mesmo.

4.2.4 Servidor Central

A ideia de criar um terceiro servidor (que chamamos de central) é de que a *thread* ou a Coroutine tenha tempo de execução parecido nos dois cenários de teste. Partimos do princípio que a requisição http realizada para um outro serviço não deve ter variações de tempo para dois serviços que estão hospedados na mesma VM.

Inicialmente, criamos um terceiro projeto java com características semelhantes ao que será testado, contendo apenas um *endpoint* que recebe uma chamada HTTP GET e retorna o resultado da conta 2+2. Para que haja a menor variação de tempo possível entre as chamadas, este servidor deve estar na mesma VPN do serviço a ser testado e que a comunicação ocorra de maneira direta - sem sair da rede.

Porém, avaliamos que as oscilações de desempenho que podem ocorrer em condições de estresse neste serviço comprometem a veracidade do teste. Por isso, optamos em realizar testes também realizando requisições HTTP para um terceiro fora da rede que teoricamente não deve sofre oscilação de tempo na resposta, como por exemplo acessar o HTML estático https://www.csus.edu/indiv/m/merlinos/enron.html ou alguma API aberta. Descreveremos os testes e as diferenças de resultados na seção abaixo.

4.3 Hospedagem

Utilizamos como serviço de hospedagem e rede o Cloud.IC, hospedado no Instituto de Computação da UNICAMP. Este pode ser acessado apenas por alunos do instituto através do site https://cloud.ic.unicamp.br/. Através dessa solução podemos hospedar os nossos serviços em máquinas virtuais (VMs) e dispor de poder de processamento e armazenamento para o funcionamento dos mesmos.

O Cloud.IC é feito utilizando o Openstack como base. Este é um sistema operacional em nuvem que é capaz de controlar grandes *clusters* de computadores através de virtualização. Também é possível criar estruturas de comunicação como filas e *streams*. Neste projeto utilizaremos apenas VMs - que também chamamos de instâncias - com uma configuração específica e determinado provisionamento de recursos. O Openstack também possui um módulo de segurança com ferramentas para garantir que o acesso aos componentes possua autenticação.

Para o propósito de comparação, os ambientes que hospedam os serviços a serem testados devem ser idênticas. Por isso, podemos utilizar uma mesma instância para hospedar as duas aplicações. Além desta, também iremos utilizar mais dois tipos diferentes de configuração neste projeto: um para a ferramenta de teste e outro para o servidor central.

O Cloud.IC possui um *dashboard* que pode ser acessado pelo site. Neste painel conseguimos realizar todas as atividades de gerenciamento de recursos necessários, como criar instâncias e ver quais estão sendo utilizadas. Abaixo temos uma imagem que mostra parte dele, mais especificamente a parte que indica o total de recursos alocados perto do limite disponível. As duas instâncias que estão sendo utilizadas nesse contexto estão hospedando o serviço a ser testado e o JMeter, respectivamente. O número de vCPUs é uma unidade de medida para a quantidade de processamento que podemos utilizar.

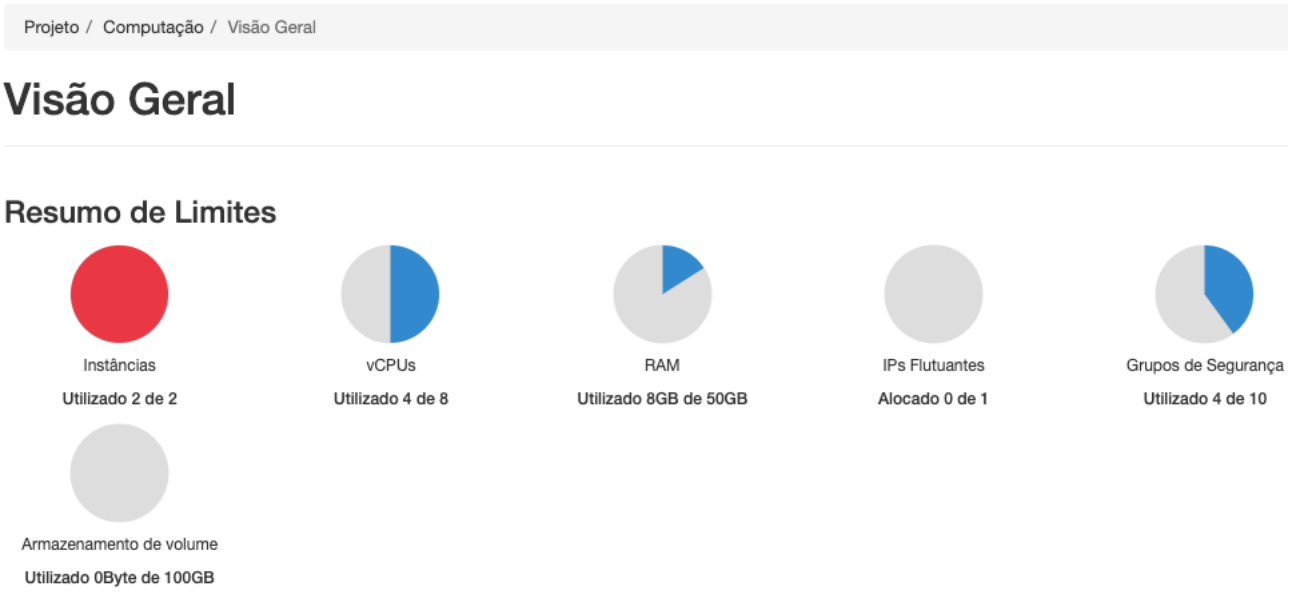

Figura 9 - Infraestrutura das máquinas utilizadas.

Ainda dentro do *dashboard*, também podemos listar as instâncias ativas, como mostra a imagem abaixo. Para este projeto será necessário inicialmente apenas duas, que chamaremos de *sender-instance* e *receiver-instance*. A *sender-instance* é responsável por hospedar a ferramenta que realiza as requisições, que no nosso caso será o JMeter. Já a *receiver-instance* hospeda a aplicação que recebe as requisições e tem o desempenho testado.

Aqui somos introduzidos ao conceito de *flavor* - ou sabor. Dentro do IC.Cloud, *flavors* definem a capacidade de processamento, armazenamento e memória de uma instânciaFLA. Ou seja, o sabor é a configuração de hardware da máquina virtual que será disponibilizada. Deve-se escolher um *flavor* no momento da criação de uma instância e não é possível alterar.

☐	Nome da instância	Nome da Imagem	Endereço IP	Flavor	Par de chaves	Status	Zona de Disponibilidade	Tarefa	Estado de energia	Tempo desde a criação
☐	sender-instance	Ubuntu-bionic-server-amd64-18.04-LTS	172.20.0.59 2801:8a:40c0:c11d:f816:3eff:fe86:8778	m1.medium.cpu-dedicated	agora vai	Ativo	nova	Nenhum	Executando	2 meses, 3 semanas
☐	receiver-instance	Ubuntu-bionic-server-amd64-18.04-LTS	172.20.0.37 2801:8a:40c0:c11d:f816:3eff:fe63:4d0e	m1.medium.cpu-dedicated	agora vai	Ativo	nova	Nenhum	Executando	2 meses, 3 semanas

Figura 10 - Instâncias ativas no *dashboard*.

A *receiver-instance*, como pode ser vista na **Figura 10**, é a VM que irá hospedar a aplicação cuja performance será testada. As principais configurações de recursos hardware provisionados podem ser encontradas na imagem abaixo. O *flavor* utilizado foi o m1.medium-cpu-dedicated, que não era o com maior poder de computação que havia disponível, porém era o único que garantia os recursos dedicados exclusivamente para a aplicação, pois é fundamental que não haja nenhuma variação de desempenho no hardware.

Visão Geral | Log | Console | Log da Ação

Nome	receiver-instance
Descrição	-
ID	5f868a3a-d214-4dbf-aeee-0438142c2615
Status	Ativo
Bloqueado	False
Zona de Disponibilidade	nova
Criado	1 de Outubro de 2020 às 21:38
Tempo Desde Criado	2 meses, 3 semanas

Especificações

Nome do Sabor	m1.medium.cpu-dedicated
ID do Sabor	1dd12bf1-ee1f-4359-8941-7be0bee69754
RAM	4GB
vCPUs	2 vCPU
Disco	25GB

Figura 11 - Detalhes da *receiver-instance*.

A *sender-instance* é a instância que irá hospedar a ferramenta de teste de performance. Também foi escolhido o flavor m1.medium-cpu-dedicated, que era a melhor configuração de hardware disponível dentro do limite do projeto.

O sistema operacional utilizado nas duas instâncias foi o Ubuntu-bionic-server-amd64-18.04-LTS. Escolhemos o Ubuntu Server pois se trata de um SO feito para ser executado em serviços de hospedagem baseados no *openstack* como o Cloud.IC, é amplamente utilizado na indústria e de fácil configuração.

Os recursos de hardware disponíveis para utilização estão conectados a uma rede privada que possui acesso à internet. É possível configurar uma instância para ter acesso externo ou não. Este conceito aqui é tratado como grupo de segurança (**Figura 12** abaixo). No momento da criação da instância, devemos selecionar quais os grupos de segurança a mesma pode acessar. Abaixo está a listagem da *receiver-instance*. No nosso caso, apenas a *receiver-instance* precisa estar conectada à web para realizar a requisição HTTP para o servidor central. Para isso, temos o grupo chamado "default", que faz com que a instância possa acessar a web. Os outros dois permitem a conexão da instância através do usuário cadastrado na rede do instituto de computação.

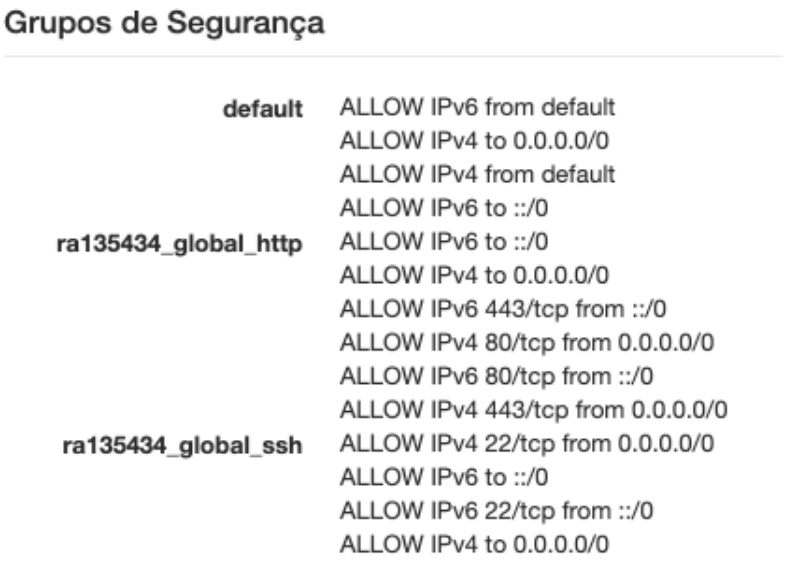

Figura 12 - Grupos de segurança

4.3.1 Deploy

Uma vez que as instâncias estão criadas e configuradas, é necessário instalar o JMeter na *sender-instance* e executar as aplicações Java e/ou Kotlin na *receiver-instance*.

O primeiro passo para o *deploy* é acessar as instâncias. Para isto, é necessário fazer via *ssh* da máquina cuja chave *ssh* foi inserida na hora de criar a instância. Por questão de segurança, adicionamos a chave *ssh* referente ao usuário de um dos integrantes dentro da

rede própria do IC. O comando necessário é:

```
# ssh -i id-rsa ubuntu@ip-da-instância
```

Para ser capaz de executar tanto os serviços quanto o JMeter, é necessário ter instalado apenas o Java instalado no Ubuntu. Vamos instalar o Java na versão 11 utilizando a biblioteca aberta *open-jdk*. Os comandos utilizados foram:

```
# sudo apt install openjdk-11-jre-headless
```

```
# sudo apt install default-jre
```

Uma vez que estamos dentro da VM do *receiver-instance*, para executar o serviço web, basta realizar dois passos:
1) Baixar o repositório git com o comando:

```
#  git clone https://gitlab.ic.unicamp.br/ra135434/pfg-jvm-testing.git
```

2) Executar o arquivo deploy.sh presente na raiz do projeto utilizando:

```
# sh deploy.sh
```

O código do arquivo deploy.sh está presente na **Figura 13** abaixo. Com este *script*, podemos executar os três serviços que criamos: java, kotlin e o central. Nas duas primeiras linhas do código, ele aguarda por um *input* que indica qual dos serviços deve ser executado. A partir daí, independente de qual foi a escolha, são três passos. O primeiro faz a navegação até a pasta do projeto, o segundo realiza a instalação das bibliotecas externas declaradas no arquivo pom.xml, e o terceiro realiza a execução do serviço.

```
1  #!/bin/bash
2  echo "Insert service name for deployment: java, kotlin, central"
3  read service_name
4
5  if [ $service_name = "java" ]; then
6      cd javaservice/
7      ./mvnw clean install
8      java -jar target/javaservice.jar
9  fi
10
11 if [ $service_name = "central" ]; then
12     cd centralservice/
13     ./mvnw clean install
14     java -jar target/centralservice.jar
15 fi
16
17 if [ $service_name = "kotlin" ]; then
18     cd kotlinservice/
19     ./mvnw clean install
20     java -jar target/kotlinservice.jar
21 fi
```

Figura 13 - Código do arquivo deploy.sh

Essa estratégia de *deploy* permite a troca de serviços a serem testados com facilidade. Na **Figura 14** abaixo temos uma representação mais completa da arquitetura, inserido os componentes de hospedagem vistos nessa seção.

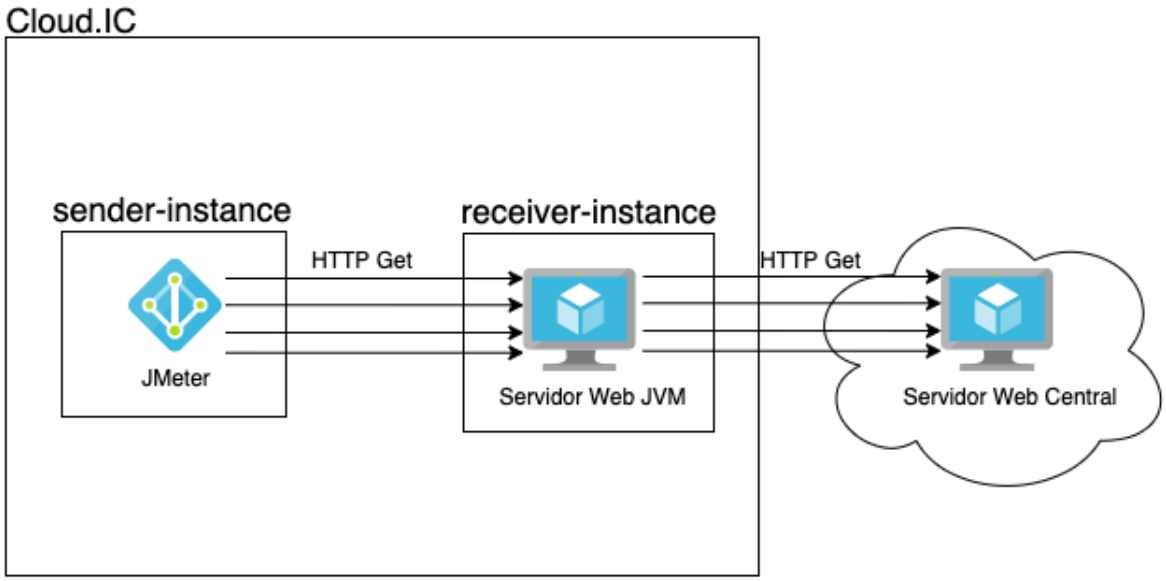

Figura 14 - Arquitetura do *deploy*

4.4 Testes de performance

4.4.1 Instalando o JMeter na Receiver instance

Para efetuar a instalação do JMeter, deve-se primeiramente logar na máquina via ssh:

```
# ssh -i id-rsa ubuntu@ip-da-instância
```

O próximo passo é fazer o download da ferramenta através do comando:

```
# wget https://downloads.apache.org//jmeter/binaries/apache-jmeter-5.4.tgz
```

O último passo é extrair a pasta. Para isso, basta executar o seguinte comando:

```
# tar -xf apache-jmeter-5.4.tgz
```

Seguindo esses passos, a ferramenta estará pronta para uso.

4.4.2 Criação de um test plan

Para criar um arquivo .jmx com as configurações de um teste dentro da interface gráfica do JMeter devemos primeiro criar uma *Thread Group* e especificar o número de *threads* (que seriam como se fossem os usuários acessando o serviço), o período de tempo ao longo do qual esse número de *threads* será executado (no exemplo abaixo, 5000 threads serão executadas

ao longo de 60 segundos) e quantas vezes esse teste será repetido (*loop count*), como pode ser visto na **Figura 15**:

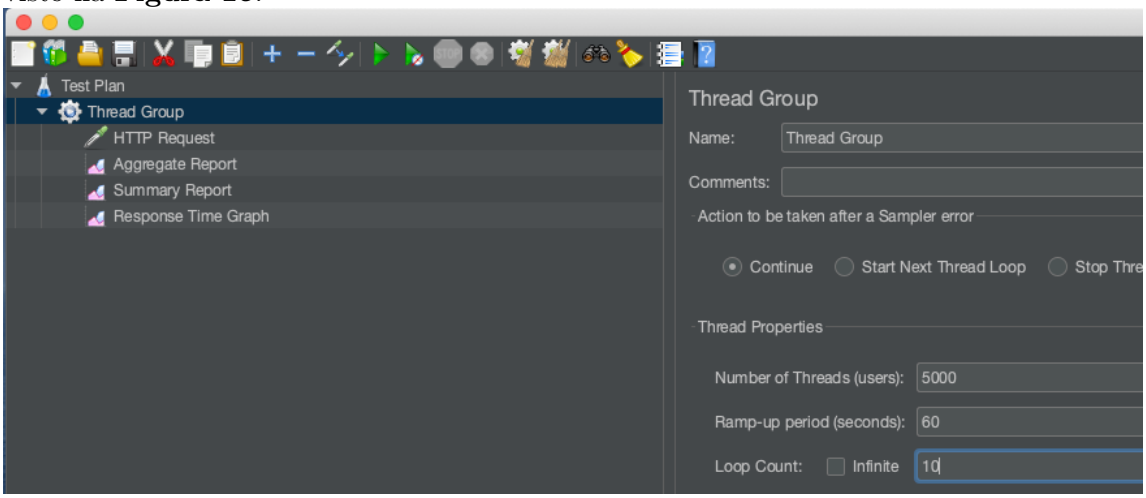

Figura 15 - Tela de configuração de uma Thread Group

Então deve-se adicionar o tipo de requisição que as *threads* irão executar. No caso, foi adicionado um HTTP Request, incluindo nas configurações o tipo de protocolo, o IP do servidor, a porta, o tipo de requisição (GET) e o caminho, como na **Figura 16**:

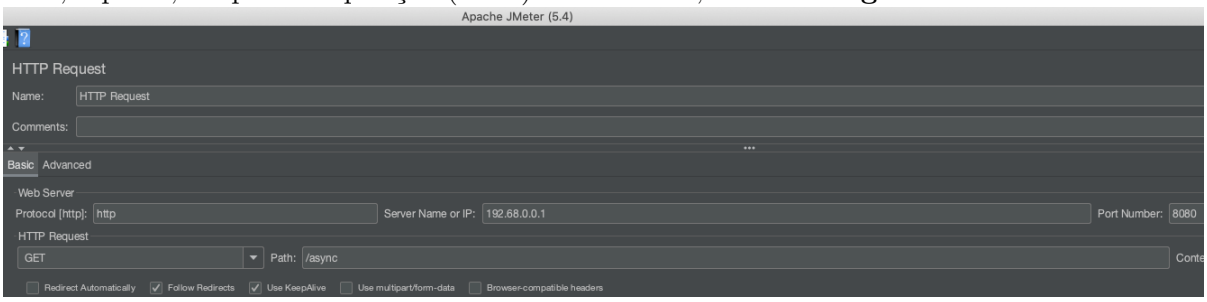

Figura 16 - Tela de configuração do HTTP Request

Por último, é necessário adicionar os *listeners*, que são usados para captar os resultados de um teste de carga. Nos testes realizados, foram utilizados o Aggregate Report, que gera dados estatísticos, o Summary Report, que é um resumo dos dados obtidos e um *listener* que coleta dados de tempo de resposta, como pode ser visto na **Figura 17**:

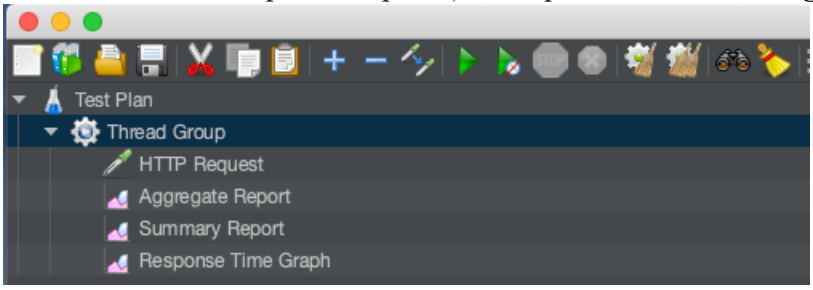

Figura 17 - Listeners adicionados no Thread Group

Após o término de todas as configurações acima, basta salvar o *test plan* como um arquivo .jmx dentro do repositório na pasta */teste-plans*. Esse arquivo permite que esse teste seja executado em qualquer máquina com JMeter instalado. Todos os *test plans* utilizados no projeto estão salvos na pasta */test-plans*.

4.4.3 Execução e Análise do resultado

Para facilitar a execução de múltiplos testes, foi desenvolvido um *script* que faz um disparo do JMeter na máquina requisitada e gera os resultados numa pasta designada. Para executar um teste, basta entrar no repositório e executar o arquivo:

```
# ./test.sh
```

O *script* funciona em 5 passos:
1 - Pergunta para o usuário o nome do experimento (por exemplo, java-load-test-1)
2 - Cria uma subpasta dentro da pasta *test-results/* com o nome desse experimento.
3 - Executa o arquivo .jmx que foi gerado pelo *test plan* (roda os testes)
4 - Coloca os resultados dentro da subpasta
5 - Faz um *commit* dos resultados no repositório *git*
Os resultados de todos os experimentos realizados estarão disponíveis na pasta *test-results/*, cada um em sua subpasta correspondente, onde haverá um arquivo log.jtl com as informações do teste. Para gerar um *dashboard* em HTML com essas informações a partir desse arquivo deve-se rodar o comando:

```
# jmeter -g /caminho-para-arquivo/log.jtl -o pasta que vai conter o html do dashboard
```

5 Resultados

Nessa seção temos os resultados dos testes práticos feitos nos serviços Kotlin e Java, assim como as análises e comparações dos mesmos. Todos os arquivos .jmx que são citados aqui estão disponíveis no repositório do projeto, dentro da pasta */test-plans*. A estratégia utilizada foi dividir o experimento rodadas de teste. Em cada rodada iremos executar um mesmo teste carga (mesmo *test-plan*) nos dois serviços e comparar os resultados. A carga dos experimentos será em ordem crescente, ou seja, vamos acrescentando *throughput* nos serviços de uma bateria para a outra.

5.1 Primeira rodada

5.1.1 Configurações do teste

- Valores utilizados:
  - Number of users (threads): 1000
  - Ramp up period (seconds): 60s

  - Loop Count: 10

- Arquivo: first-test-plan.jmx
- Número de threads na thread pool:
  - Java: 200
  - Kotlin: 5

5.1.2 Resultado Serviço Java

Pasta: java-1k-enron-1
Apdex = 0,504

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**10000**	**0**	**0.00%**	**1082.58**	**198**	**4159**	**1140.00**	**1550.00**	**1668.00**	**1910.00**	**153.70**	**1010.45**	**18.76**
HTTP Request	10000	0	0.00%	1082.58	198	4159	1140.00	1550.00	1668.00	1910.00	153.70	1010.45	18.76

Figura 18 - Estatísticas do serviço Java com 1000 usuários

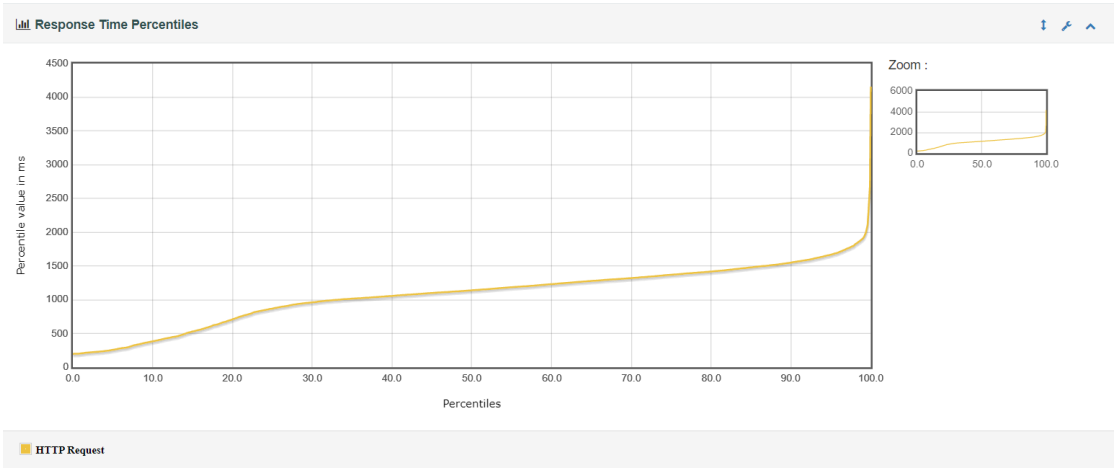

Figura 19 - Gráfico do tempo de resposta por percentil do serviço Java com 1000 usuários

5.1.3 Resultado Serviço Kotlin

Pasta: kotlin-1k-enron-3
Apdex = 0,786

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**10000**	**0**	**0.00%**	**484.30**	**321**	**2769**	**479.00**	**598.00**	**613.00**	**680.98**	**154.69**	**61.48**	**18.88**
HTTP Request	10000	0	0.00%	484.30	321	2769	479.00	598.00	613.00	680.98	154.69	61.48	18.88

Figura 20 - Estatísticas do serviço Kotlin com 1000 usuários

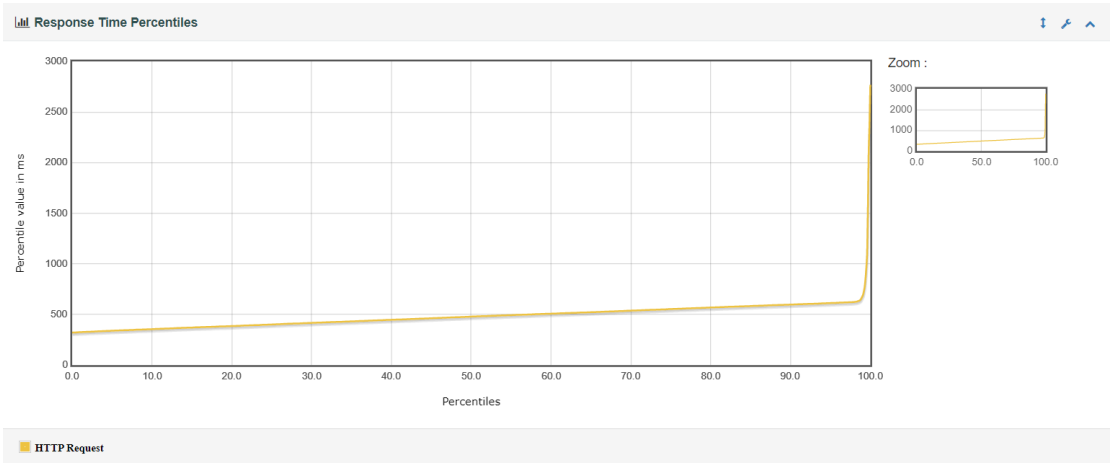

Figura 21 - Gráfico do tempo de resposta por percentil do serviço Kotlin com 1000 usuários

5.1.4 Análise e comparação

Observando os dois resultados, notamos que o desempenho do serviço Kotlin foi significantemente melhor que a do serviço Java em todas as métricas. O APDEX é quase 50% maior, o tempo médio de resposta é a metade e o 99 percentil é um terço. Apesar disso, a performance dos dois ainda é minimamente satisfatória nesse cenário, visto que não houve nenhum erro.

5.2 Segunda rodada

5.2.1 Configurações do teste

- Valores utilizados:
  - Number of users (threads): 5000
  - Ramp up period (seconds):60s
  - Loop Count: 10
- Arquivo: second-test-plan.jmx

- Número de threads na thread pool:
  - Java: 200
  - Kotlin: 5

5.2.2 Resultado Serviço Java

Pasta: java-5k-enron-0

Apdex = 0,001

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**50000**	**0**	**0.00%**	**20660.09**	**226**	**29186**	**21641.00**	**25475.00**	**25742.00**	**26148.99**	**188.15**	**1236.92**	**22.97**
HTTP Request	50000	0	0.00%	20660.09	226	29186	21641.00	25475.00	25742.00	26148.99	188.15	1236.92	22.97

Figura 22 - Estatísticas do serviço Java com 5000 usuários

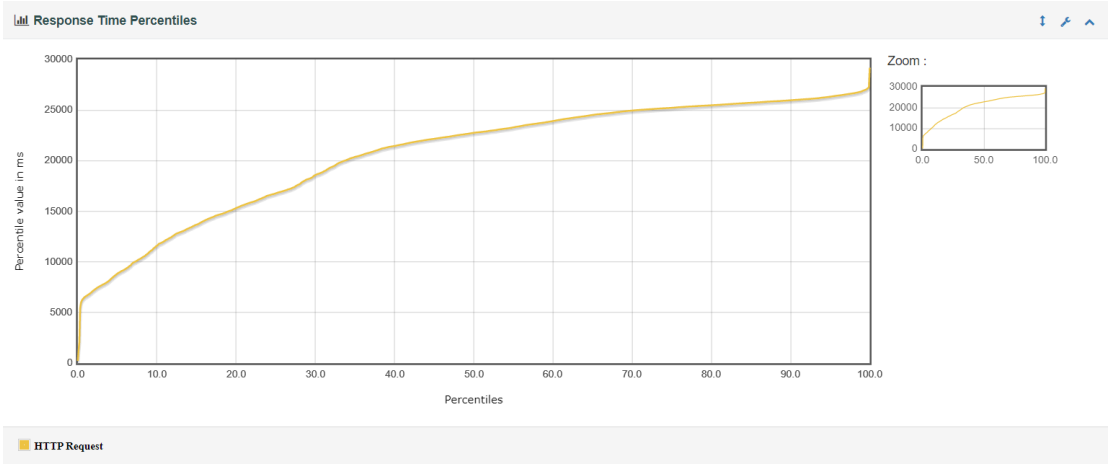

Figura 23 - Gráfico do tempo de resposta por percentil do serviço Java com 5000 usuários

5.2.3 Resultado Serviço Kotlin

Pasta: kotlin-5k-enron-0

Apdex = 0,437

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**50000**	**0**	**0.00%**	**1160.99**	**321**	**7390**	**634.00**	**973.90**	**1041.95**	**1124.00**	**779.16**	**309.68**	**95.11**
HTTP Request	50000	0	0.00%	1160.99	321	7390	634.00	973.90	1041.95	1124.00	779.16	309.68	95.11

Figura 24 - Estatísticas do serviço Kotlin com 5000 usuários

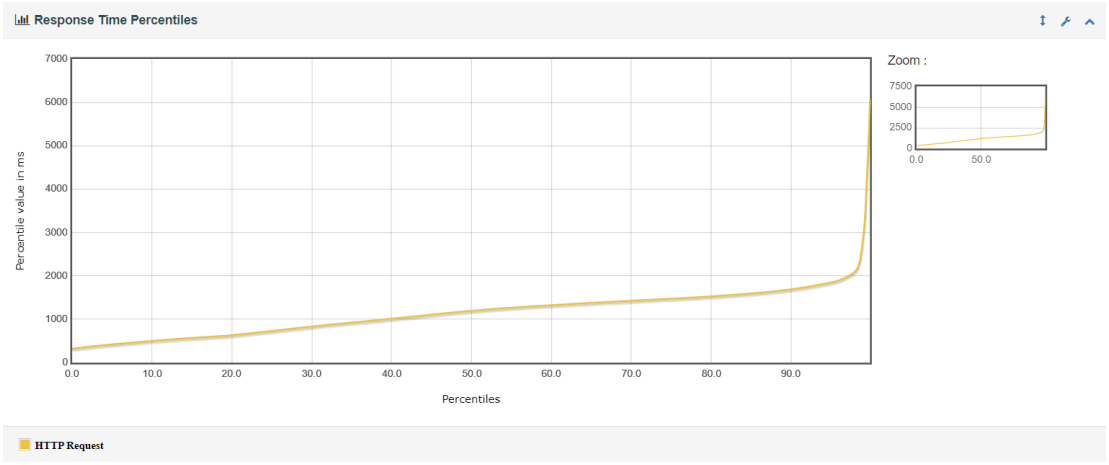

Figura 25 - Gráfico do tempo de resposta por percentil do serviço Kotlin com 5000 usuários

5.2.4 Análise e comparação

Aqui multiplicamos por 5 a carga do experimento anterior, e acabou ficando claro que a diferença de desempenho dos sistemas é bem grande. A aplicação Java teve $tempomédioderesposta = 20s$ e $APDEX = 0.0001$, o que indica que ela já não consegue suportar essa carga. Já o Kotlin aguentou a carga e teve seu $97percentil < 2segundos$, o que consideramos uma performance satisfatória. Um ponto importante de se observar é que, mesmo que lentamente, as plataformas ainda conseguem responder todas as requisições do teste. O índice de erro de ambas ainda é 0%.

5.3 Terceira rodada

5.3.1 Configurações do teste

- Valores utilizados:
  - Number of users (threads): 10000
  - Ramp up period (seconds): 60s
  - Loop Count: 10
- Arquivo: third-test-plan.jmx
- Número de threads na thread pool:
  - Java: 400
  - Kotlin: 5

5.3.2 Resultado Serviço Java

Pasta: java-10k400t-enron-0

Apdex = 0,000

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**32811**	**878**	**2.68%**	**36683.92**	**393**	**132677**	**39489.00**	**76319.90**	**95417.60**	**131464.97**	**178.44**	**1154.03**	**21.20**
HTTP Request	32811	878	2.68%	36683.92	393	132677	39489.00	76319.90	95417.60	131464.97	178.44	1154.03	21.20

Figura 26 - Estatísticas do serviço Java com 10000 usuários e uma pool de 400 Threads

Errors

Type of error	Number of errors	% in errors	% in all samples
Non HTTP response code: org.apache.http.conn.HttpHostConnectException/Non HTTP response message: Connect to 172.20.0.37:8081 [/172.20.0.37] failed: Connection timed out (Connection timed out)	612	69.70%	1.87%
Non HTTP response code: java.net.SocketException/Non HTTP response message: Connection reset	266	30.30%	0.81%

Figura 27 - Erros obtidos no serviço Java com 10000 usuários e uma pool de 400 Threads

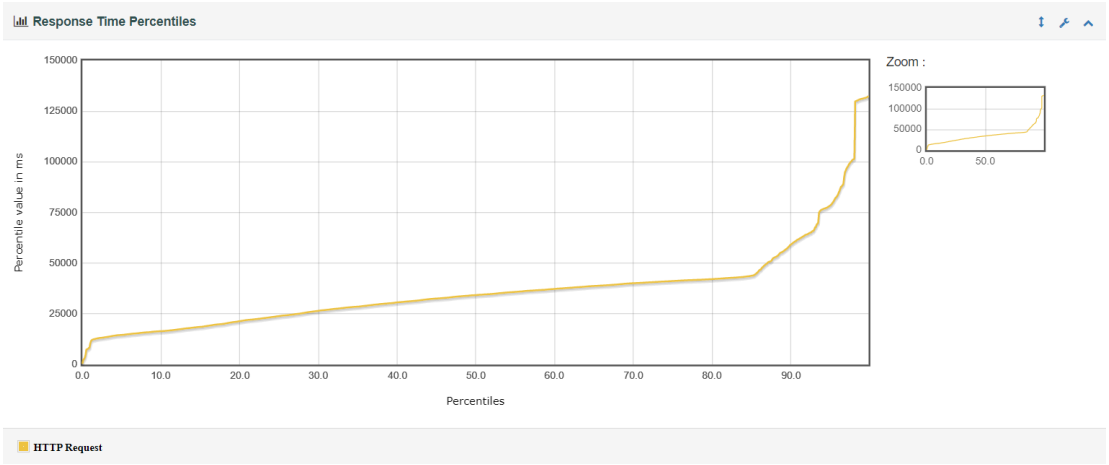

Figura 28 - Gráfico do tempo de resposta por percentil do serviço Java com 1000 usuários e uma pool de 400 Threads

5.3.3 Resultado Serviço Kotlin

Pasta: kotlin-10k-enron-0

Apdex = 0,007

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**100000**	**79**	**0.08%**	**7621.15**	**333**	**79466**	**5738.00**	**6668.00**	**6781.00**	**6916.00**	**781.20**	**2264.41**	**95.29**
HTTP Request	100000	79	0.08%	7621.15	333	79466	5738.00	6668.00	6781.00	6916.00	781.20	2264.41	95.29

Figura 29 - Estatísticas do serviço Kotlin com 10000 usuários

Errors

Type of error	Number of errors	% in errors	% in all samples
Non HTTP response code: java.net.SocketException/Non HTTP response message: Connection reset	79	100.00%	0.08%

Figura 30 - Erros obtidos no serviço Kotlin com 10000 usuários

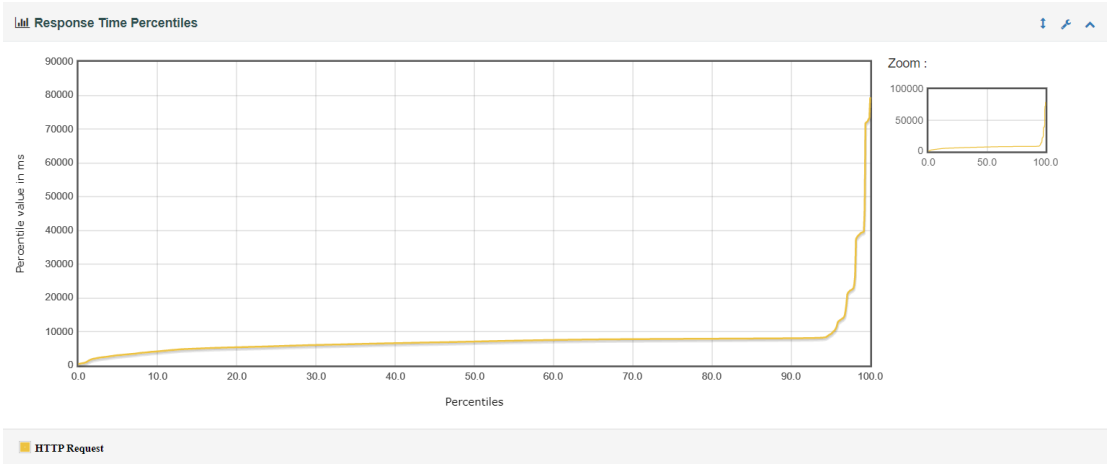

Figura 31 - Gráfico do tempo de resposta por percentil do serviço Kotlin com 10000 usuários

5.3.4 Análise e comparação

Neste teste dobramos a carga do teste passado para buscar o limite do serviço Kotlin. Nesse cenário, o desempenho da aplicação piorou consideravelmente, de modo que $2percentil > 2segundos$. A APDEX caiu para quase zero e houve incidência do erro: "Non HTTP response code: java.net.SocketException/Non HTTP response message: Connection reset", que significa que atingiu o *timeout* da requisição.

Já para a aplicação Java, resolvemos aumentar o número de *threads* na *thread* pool para 400 para ver se a aplicação conseguia performar melhor com um *threadpool* maior. Porém, não houve melhora na performance, o que indica que o sistema realmente não comporta cargas dessa escala.

6 Conclusão

Durante a análise teórica, entendemos que por estarem sendo executados em sistemas semelhantes a nível de software e hardware, não haveriam grandes diferenças entre o desempenho dos dois serviços. Acreditamos que haveria apenas uma vantagem pelo uso de Coroutines em vez de Threads.

Porém, após a realização dos testes de carga nas aplicações com diversas configurações e análise dos resultados, concluímos que o serviço web desenvolvido em Kotlin performa de maneira substancialmente melhor que o serviço Java. As duas métricas que levamos em consideração para essa conclusão são tempo de resposta médio e escalabilidade, que são as características mais visadas para serviços *backend* web atualmente.

No teste que consideramos de carga baixa - 1000 usuários a cada minuto -, a performance dos dois sistemas foi aceitável, com $APDEX > 0,5$. Quando multiplicamos o *throughput* para valores 5 e 10 vezes maiores, a aplicação Java não conseguiu lidar e passou a ter um comportamento que consideramos inaceitável para os requerimentos de hoje, com a maior parte das requisições levando mais que 10 segundos para retornarem ($20 percentil > 10 segundos$). Já o outro se manteve estável com 5000 usuários/segundo, porém também não aguentou a carga maior.

Os resultados, portanto, sugerem que Kotlin é uma escolha mais razoável em situações similares às realizadas neste projeto, tendo em vista as ferramentas e cenários utilizados nos testes comparativos.

Referências

[1] https://kotlinlang.org/docs/reference/faq.html

[2] https://spring.io/projects/spring-boot

[3] https://maven.apache.org/

[4] https://jmeter.apache.org/

[5] https://github.com/Kotlin/kotlinx.coroutines

[6] https://github.com/JetBrains/kotlin

[7] https://www.openstack.org/software/

[8] https://docs.openstack.org/nova/latest/user/flavors.html

[9] https://spring.io/team

[10] https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html

[11] https://blog.golang.org/waza-talk

[12] https://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html

[13] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html