**Github:** <https://github.com/jandersonrafa/communication-rmi-rest/v2>

**1 - Introdução**

Iremos avaliar a performance da nova feature de Virtual Threads inserida de forma experimental no java 19 e agora presente no java 21. Lembrando que o Spring 6 agora também incorporou e permite habilitar.

Sendo assim, o objetivo deste benchmark é comparar spring com threads normais de sistema operacional e com threads de jvm as com a ativação das chamadas threads virtuais.

**2 - Métricas importantes**

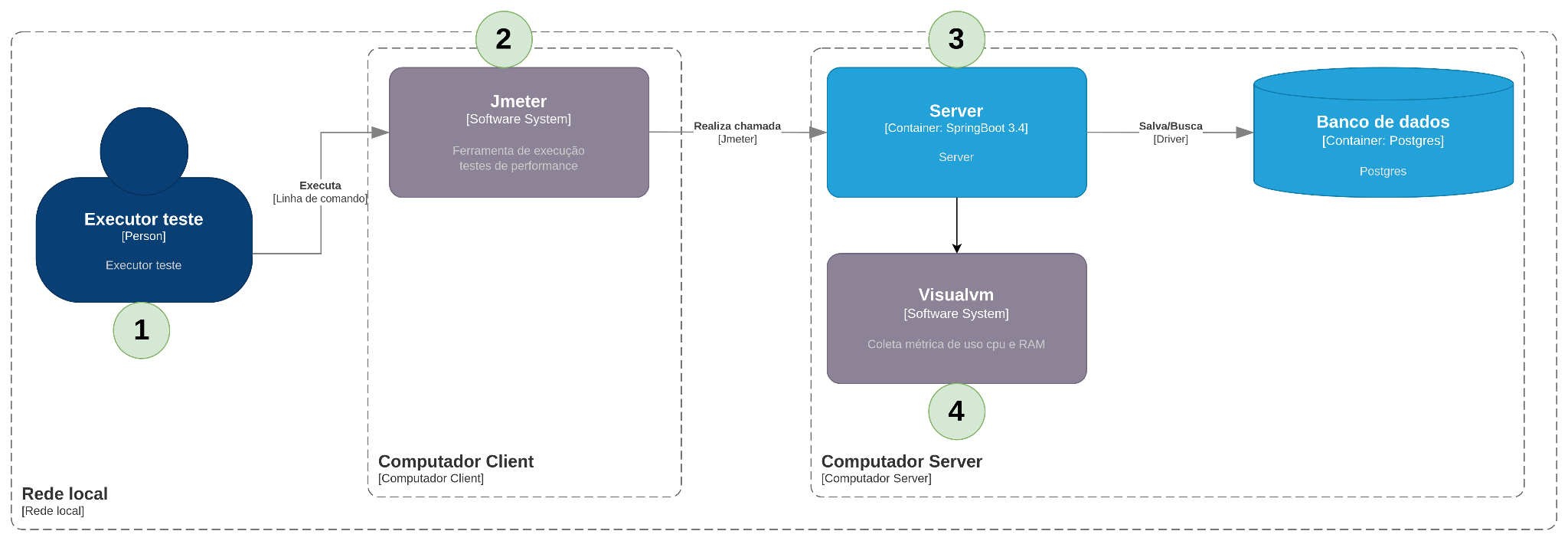
Para esse estudado foram consideradas as seguintes métricas:

* Desempenho
  + Tempo resposta
  + Taxa de erros
* Utilização de recursos do servidor
  + cpu
  + ram

**3 - Cenário de teste**

**3.1 - Componentes do teste**

A execução do testes de carga será realizada seguindo a seguinte o diagrama abaixo:



No diagrama podemos destacar os seguintes componentes:

* Executor teste: pessoa responsável pela execução dos testes manipulando Jmeter
* Jmeter: ferramenta de execução de testes de performance com cenários de requisições simultâneas
* Server: Aplicação servidor com spring 3.4 que responde a solicitação utilizando protocolo http
* Banco de dados: Banco postgres com tabela employee
* Visualvm: profiler que monitora as aplicações para avaliar estado de utilização dos recursos, tais como cpu e ram.

**3.2 - Configurações dos servidores**

* **Servidor server:**
  + So: Windows 8
  + cpu: Intel(R) Core(TM) i7
  + ram: 32 GB DDR
* **Servidor client:**
  + So: Linux Ubuntu
  + cpu: AMD Ryzen 7 5700U - 16 threads virtuais
  + ram: 32,0 GB (utilizável: 30,73 GB)

**3.3 - Stack das aplicações**

* **Jmeter:** versão apache-jmeter-5.6.3
* **Visualvm**: versão **2.1.8**
* **Server:**
  + Server: Java 21, Spring boot 3.4.0, jetty server embedded

**3.4 - Fluxo do teste**

O teste será executado dentro do computador client, onde será executado o jmeter que fará uma série de requisições para o endpoints de teste employee que será ouvido pela aplicação do servidor (ms-server), utilizando protocolo http.

**3.5 - Endpoints e objetos de comunicação**

* GET thread Name
  + Endpoint de teste do client**:** [http://localhost:8080/thread/](http://localhost:8080/client/beneficiario/)name
  + Parâmetro de entrada: nenhum
  + Objeto de resposta: String com nome de thread
* POST employee
  + Endpoint de teste do client**:** [http://localhost:8080/](http://localhost:8080/client/beneficiario/)employee
  + Parâmetro de entrada: {"firstName": "foo33", "lastName": "bar33", "dateOfBirth": "2023-05-04", "age": "21","nickname": "nicknameyteste","ddd": "51","phone": "99925555","city": "canoas","position": "Developer"}
  + Objeto de resposta:{"id":12,"firstName":"foo33","lastName":"bar33","dateOfBirth":"2023-05-04", "age": "21","nickname": "nicknameyteste","ddd": "51","phone": "99925555","city": "canoas","position": "Developer"}
* GET employee by id
  + Endpoint de teste do client**:** [http://localhost:8080/](http://localhost:8080/client/beneficiario/)employee/{id}
  + Parâmetro de entrada: id no path
  + Objeto de resposta:{"id":12,"firstName":"foo33","lastName":"bar33","dateOfBirth":"2023-05-04", "age": "21","nickname": "nicknameyteste","ddd": "51","phone": "99925555","city": "canoas","position": "Developer"}
* GET all employee
  + Endpoint de teste do client**:** [http://localhost:8080/](http://localhost:8080/client/beneficiario/)employee/find-all
  + Parâmetro de entrada: nenhum
  + Objeto de resposta:[{"id":1,"firstName":"foo94903","lastName":"bar94903","nickname":"nicknameyteste","age":"21","ddd":"51","phone":"99925555","position":"Developer","city":"canoas","dateOfBirth":"2023-05-04"},{"id":2,"firstName":"foo83011","lastName":"bar83011","nickname":"nicknameyteste","age":"21","ddd":"51","phone":"99925555","position":"Developer","city":"canoas","dateOfBirth":"2023-05-04"}]

**3.6 - Aplicações**

As aplicações client e server de cada protocolo serão executadas em dois computadores diferentes, para deixar mais realista a questão de latência na comunicação.

O ideal é que as aplicações sejam executadas em formato docker, limitando a 512 mb de ram e 1 core de cpu, porém devido a questões técnicas executaremos direto da IDE Intellij.

Para quem quiser utilizar formado docker foi criado um dockerfile dentro de cada projeto para fazer a compilação e construção de uma imagem, fixando a definição da versão maven e java 8 para evitar problemas:

Exemplo dockerfile:

FROM maven:3.8.4-openjdk-8 AS *build*

WORKDIR /app

# Copia o código fonte do projeto para o diretório de trabalho

COPY . .

# Executa a construção do projeto, pulando os testes

RUN mvn clean install -DskipTests

FROM openjdk:8-jdk-alpine

WORKDIR /app

# Copia o jar gerado para a nova imagem

COPY --from=*build* /app/target/target/\*.jar .

# Comando para iniciar a aplicação

ENTRYPOINT ["java","-jar","/app.jar"]

Comandos utilizados:

* Compilar aplicação e gerar imagem com jar executável: docker build . -t best-answer-back-main
* Executar container docker
* Executar container docker client: docker run -d --cpus=1 --memory=512m --name <container\_name> --network minha-rede -p <port>:<port> <docker\_image\_name>
* ip addr show eth0: dentro do wsl2 para descobrir ip do wsl2

**4 - JMETER**

**4.1 - Parâmetros do teste**

* **Numero de requisicoes por thread:** 4
* **Number of Threads:** 400
* **Ramp-Up Period:** 50 segundos
* **Loop Count:** 200
* **Total de requisições:** 160000
* **Constant timer (thread delay)**: 0 ms (tempo de aguardo entre requisições) (não inserido)

**4.2 - Execução dos testes**

Foram executadas 4 baterias de teste para cada protocolo, descartando a primeira execução e salvando as 3 últimas. A execução dos testes no jmeter foi realizado via linha de comando

alias jmeter="C:/cwi/programs/apache-jmeter-5.6.2/bin/jmeter"

* rmi: ./jmeter -n -t ./jmeter/teste-carga.jmx -l ./jmeter/result-rmi-execucao-<numero\_execucao>.csv -e -o ./jmeter/result-rmi-execucao-<numero\_execucao>
* grpc: ./jmeter -n -t ./jmeter/teste-carga.jmx -l ./jmeter/result-grpc-execucao-<numero\_execucao>.csv -e -o ./jmeter/grpc-rmi-execucao-<numero\_execucao>/
* rest (rest template): ./jmeter -n -t ./jmeter/teste-carga.jmx -l ./jmeter/result-rest-rest-template-execucao-<numero\_execucao>.csv -e -o ./jmeter/rest-rest-template-execucao-<numero\_execucao>/
* rest (client-netflix-oss-feign): ./jmeter -n -t ./jmeter/teste-carga.jmx -l ./jmeter/result-rest-netflix-oss-feign-execucao-<numero\_execucao>.csv -e -o ./jmeter/rest-netflix-oss-feign-execucao-<numero\_execucao>/

**5 - Ressalvas**

O contexto de execução do teste foi construído tentando deixar o mais próximo de um cenário real de produção, porém dentro de um ambiente local e comum a um desenvolvedor, pois sabemos que nem sempre se tem o tempo e os recursos necessários para se fazer um teste super realista, então destaco abaixo principais pontos de atenção:

* O volume de carga do teste poderia ser maior, porém precisaríamos de mais máquinas e mais tempo para aguardar os resultados
* As aplicações e o jmeter que vai executar os testes estão rodando em uma mesma rede local, para se aproximar de produção o ideal seria que o jmeter executor estivesse em uma rede separada da rede das aplicações, para evidenciar pontos de latência.
* A aplicação client e o jmeter estão rodando na mesma máquina, o ideal seriam estar em máquinas separadas, pois o executor, mesmo em linha de comando irá consumidor recursos do computador que é o mesmo onde está hospedado a aplicação client.
* O ideal seriam as aplicações rodarem em docker, porém devido a problemas com windows em conjunto com wsl2 e comunicação entre computadores diferentes, foi optado por executar direto da IDE Intellij, sem limitar cpu e ram.

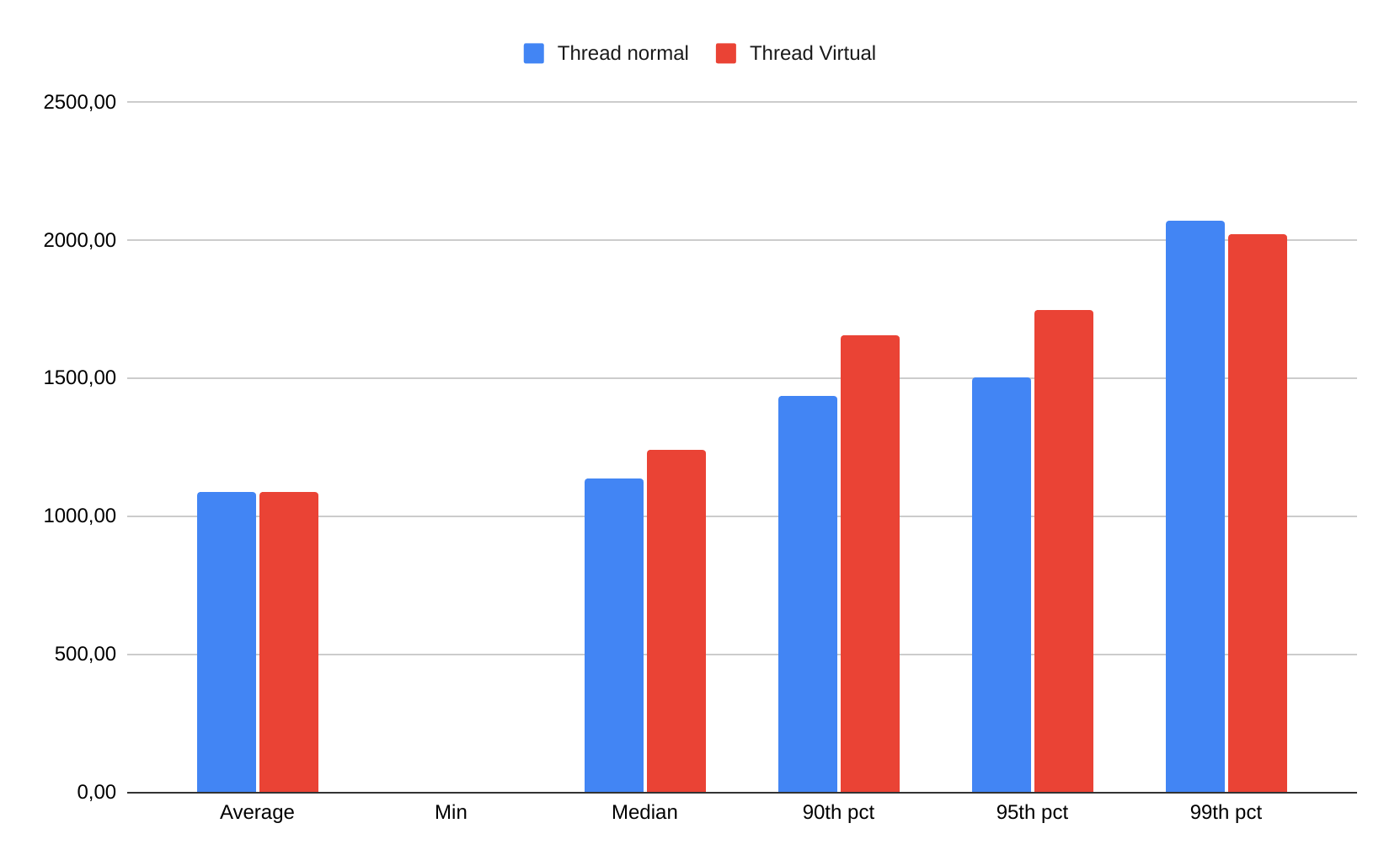
**6 - Resultados**

O script executado no teste ficará dentro da pasta jmeter/teste-carga.jmx e os relatórios de execução ficaram dentro da pasta jmeter em um subdiretório respectivo ao protocolo testado.

**6.1 - Desempenho**

* + Tempo resposta

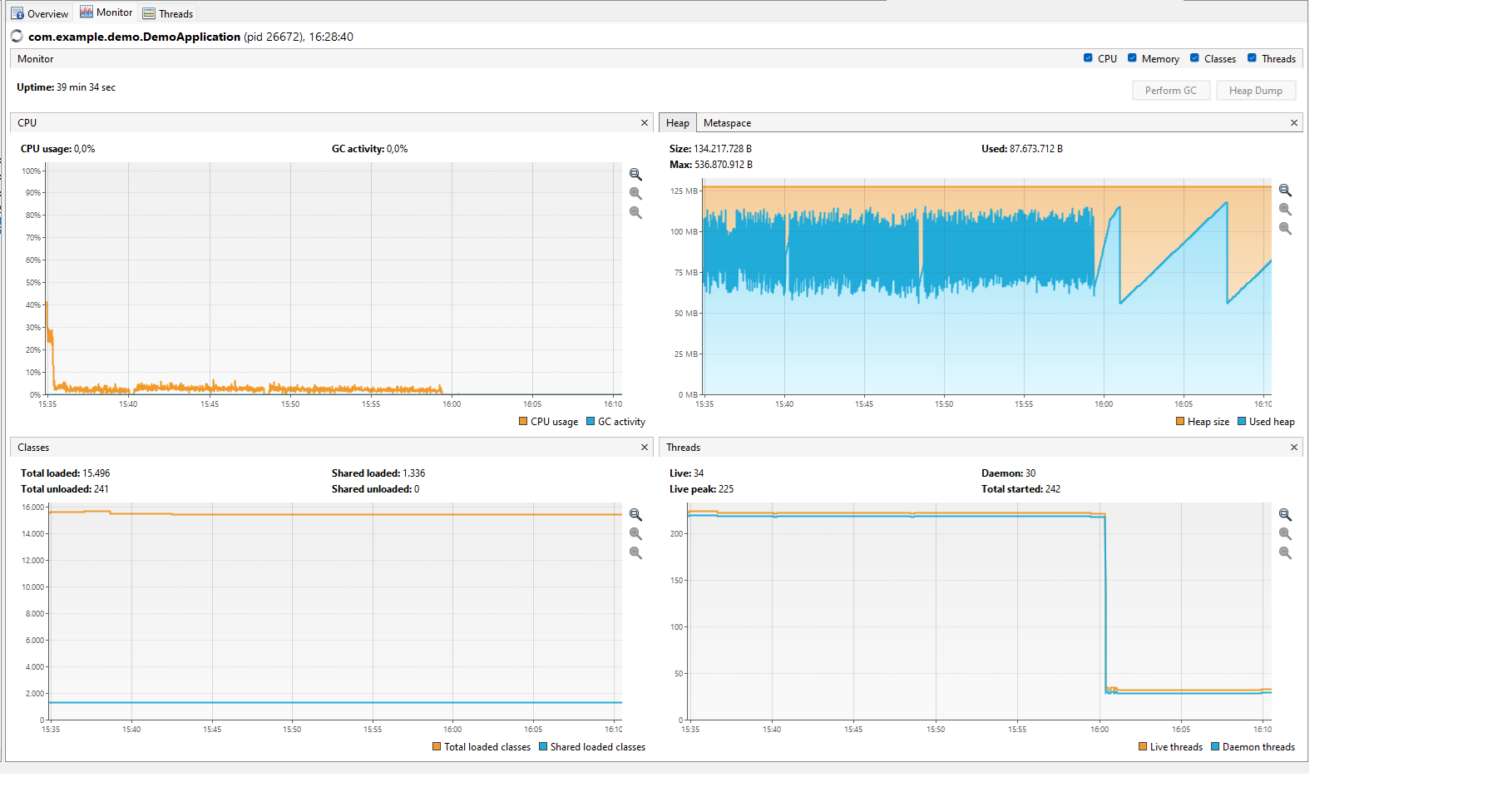
| **GERAL** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Response times (ms)** | | | | | |
| **Protocolo/client** | **Average** | **Min** | **Median** | **90th pct** | **95th pct** | **99th pct** |
| Thread normal | 1088,10 | 1,00 | 1135,17 | 1438,30 | 1504,33 | 2074,33 |
| Thread Virtual | 1090,75 | 1,00 | 1243,00 | 1657,30 | 1750,32 | 2024,99 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |



* + Taxa de erros
    - Nenhuma das execuções resultou em alguma falha nos requests

**6.2 - Utilização de recursos do servidor**

* **thread normal**

****

* **thread virtual**

****

**6.3 - Facilidade de implementação**

A implementação de código de ambos é praticamente igual, no spring boot só precisa acionar via properties com a propriedade:

***spring.threads.virtual.enabled=true***

**7 - Avaliação resultados**

**7.1 - Avaliação resultados- Facilidade de implementação**

A implementação de código de ambos é praticamente igual, no spring boot só precisa acionar via properties.

**7.2 - Avaliação resultados- Uso de recursos**

Em geral foi observado que as threads virtuais utilizam mais memória e utilizam muito menos threads.

**7.3 - Desempenho**

O servidor nesse benchmark teve um tempo de resposta muito similar em ambos os casos, tendo uma leve melhora com o uso de threads normais em relação a threads virtuais.

| **Protocolo/client** | **Average** |
| --- | --- |
| Thread normal | 1088,10 |
| Thread virtual | 1090,75 |
|  |  |
|  |  |

**8 - Conclusão**

Em relação ao teste com múltiplas requisições http não foi observado melhora de response time e uso de recursos em atendimentos de requisições em api. Mesmo tendo criado cenários que simulasse i/o não foi notado melhora.