

Comparando desempenho de threads Java aplicadas ao servidor Tomcat

Felipe Rosa, Walter Silvestre Coan

Bacharelado em Engenharia de Software
Universidade da Região de Joinville (Univille) Joinville – SC – Brasil

`feliperosa.1@univille.br, walter.s@univille.br`

Resumo. *Dada a relevância da linguagem Java na computação e a nova forma de execução com threads virtuais, este trabalho tem o objetivo de apresentar uma análise comparativa feita através da execução de um experimento com três cenários (cada um utilizando uma implementação de thread diferente da linguagem Java) aplicadas ao servidor Tomcat. Nos cenários, é utilizada uma ferramenta para disparo de requisições HTTP para o servidor, obtendo dados como percentual de sucesso das requisições e latência da resposta. Adicionalmente, enquanto as requisições são executadas, uma ferramenta de análise de memória, consumo de CPU e threads do SO é utilizada para extrair esses dados da aplicação. Com base no experimento, verificou-se que no cenário com threads virtuais não houve diminuição expressiva na latência média das requisições, além de observar-se um consumo maior de memória quando comparada às demais formas testadas.*

Palavras-chave: *Threads, concorrência, Java, Tomcat.*

Abstract. *Given the relevance of the Java language in computing and the new form of execution with virtual threads, this work aims to present a comparative analysis made through the execution of an experiment with three scenarios (each one using a different thread implementation of the Java language) applied to the server Tomcat. In the scenarios, a tool is used to trigger HTTP requests to the server, obtaining data such as the percentage of success of the requests and latency of the response. Additionally, while the requests are being executed, a tool for analyzing memory, CPU consumption and OS threads is used to extract the data of the application. Based on the experiment, it was verified that in the scenario with virtual threads there was no expressive decrease in the average latency of the requests, in addition to observing a higher consumption of memory when compared to the other forms tested.*

Keywords: *Threads, concurrency, Java, Tomcat.*

1. Introdução

Com a popularização dos multiprocessadores, as linguagens de programação têm fornecido mecanismos alternativos para implementar concorrência, visando melhor experiência de desenvolvimento e aproveitamento de recursos dos dispositivos. Nesse contexto, a linguagem de programação Java introduziu uma nova forma de utilização de *threads*, aplicadas neste trabalho ao servidor Tomcat e sendo comparada às demais implementações já existentes.

A linguagem de programação Java é constantemente elencada entre as mais populares da computação (TIOBE Index, 2023), o que se justifica por motivos como: neutralidade de arquitetura, portabilidade e robustez (SCHILDT, 2015). Em aplicações desenvolvidas em Java para o mundo web, o servidor de aplicação Tomcat é o mais utilizado (JREBEL, 2022), em grande parte por ser o servidor utilizado no popular *framework* para construção de aplicações web com Java - Spring Boot. Além disso, é reconhecido como uma opção de livre utilização (*open source*), leve e com amplo suporte da comunidade de software.

No servidor Tomcat, o modelo de *threads* utilizado ainda depende de um relacionamento direto com *threads* do sistema operacional, o que pode introduzir problemas de espera por recursos que são gerenciados pelo SO. Em um contexto em que o tempo de resposta das requisições deve ser otimizado, este modelo pode trazer problemas que as *threads* virtuais da linguagem se propõem a resolver e que foram comparados neste trabalho. Em experimentos realizados por outros pesquisadores (Beronić et al, 2021), verificou-se diminuição no tempo de resposta das requisições processadas com *threads* virtuais, quando comparadas ao modelo de *threads* tradicional da linguagem.

Portanto, dada a relevância da linguagem Java, a nova forma de implementação de *threads* disponível e a ampla utilização do servidor Tomcat em aplicações web, este trabalho tem como objetivo comparar as diferentes implementações de *threads* em Java no servidor Tomcat, visando responder à seguinte questão de pesquisa: quais as diferenças de desempenho entre as implementações de *threads* em Java, quando aplicadas ao servidor Tomcat?

2. Revisão da literatura

2.1 Sistemas operacionais e multiprogramação

Os sistemas operacionais (SOs) surgiram principalmente para gerenciar complexidade de hardware, sendo essencial que ele otimize os recursos disponíveis (TANENBAUM, 2016). Silberschatz (2016) destaca que um dos aspectos de um SO moderno é a capacidade de multiprogramação, ou seja, de poder executar diferentes tarefas e garantir que o processamento da CPU seja compartilhado entre elas. Para isso, é essencial que as tarefas estejam definidas de forma padronizada, papel de processos e *threads*.

2.2 Processos e *threads*

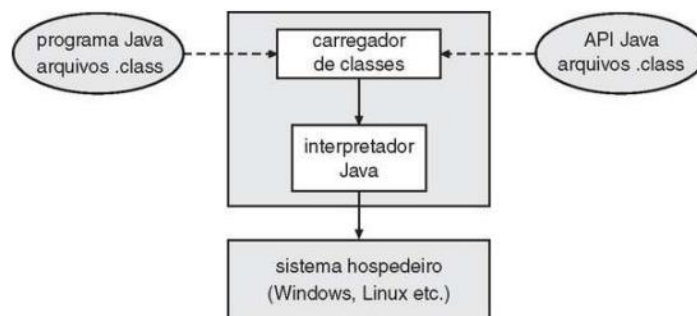
Um processo pode ser definido como um programa carregado na memória principal e em execução pelo computador (SILBERSCHATZ, 2016), contendo recursos associados a ele. Apesar de agregar informações, a abstração que executará na Unidade Central de Processamento (CPU) é a nível de *thread*.

Silberschatz (2016) define *threads* como unidades básicas de utilização da CPU, sendo essencialmente úteis quando se deseja executar mais de um procedimento dentro de um mesmo processo. Tanenbaum (2016) destaca alguns motivos para utilização de *threads*: em aplicações com muitas atividades ao mesmo tempo e com bloqueios, por bloquear apenas a *thread* e não a aplicação como um todo; por serem mais fáceis de criar e destruir quando comparado aos processos; e por serem especialmente úteis em sistemas com múltiplas CPUs.

2.3 A JVM e modelos de *multithreading* em Java

Java é executada a partir da máquina virtual da linguagem (JVM) no sistema operacional em que foi instalada (SILBERSCHATZ, 2016). Ao compilar um código Java escrito pelo desenvolvedor ou proveniente das classes utilitárias da linguagem (APIs), são gerados binários intermediários (*bytecode*, em arquivos “.class”), que são carregados pela JVM no carregador de classes e interpretados em tempo de execução para as instruções do hardware no sistema operacional hospedeiro, processo ilustrado na Figura 1.

Figura 1: Funcionamento da máquina virtual Java (JVM).



Fonte: Silberschatz et al. (2016, p. 100).

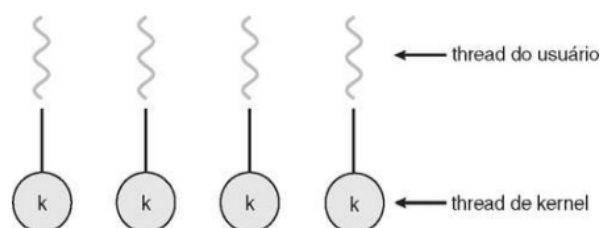
Dentre as classes utilitárias disponíveis para desenvolvimento de aplicações com mais de uma unidade de execução (*multithreading*), destacam-se: (1) através da instância da classe *Thread*, que cria a *thread* na JVM (*Java Virtual Machine*, ambiente de execução da linguagem) e no sistema operacional, sendo descartada após o fim da execução; e através de “*Executors*”, que oferecem uma interface com diferentes formas de criar e gerenciar *threads*, voltadas para submissão de tarefas (BLOCH, 2018).

Neste trabalho, *multithreading* é implementado apenas a partir da interface “*Executor*”, sendo utilizados: “*Executor*” de banco de *threads* (“*thread pools*”) em cache (*threads* criadas conforme demanda e reaproveitadas entre as tarefas submetidas) e “*Executor*” de *threads* de tipo virtual, por tarefa (que utiliza o novo conceito de *threads* virtuais para as execuções).

2.4 Tipos de *thread* em Java

A linguagem Java tradicionalmente utiliza o modelo de *multithreading* “um para um”, em que para cada *thread* criada dentro da JVM, outra deve ser criada no SO hospedeiro, estabelecendo relacionamento único e direto, conforme ilustrado na Figura 2.

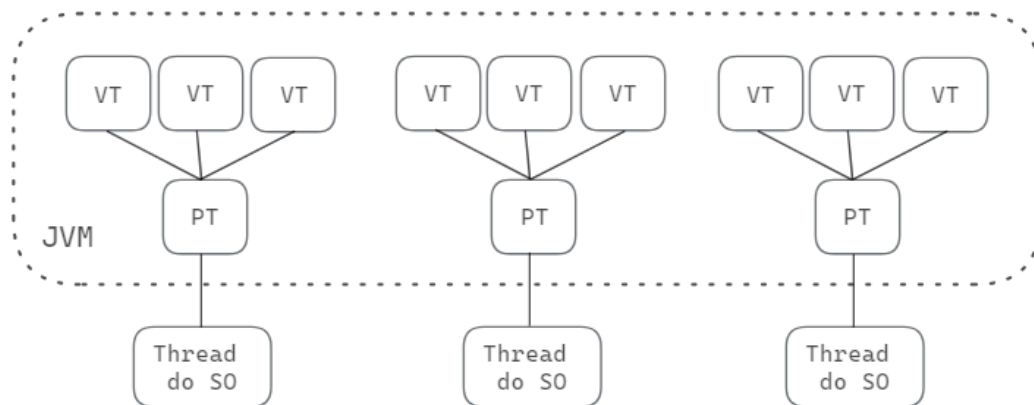
Figura 2: Modelo de *multithreading* um para um.



Fonte: Silberschatz et al. (2016, p. 120).

Apesar de facilitar a concorrência devido a relação direta com *threads* do SO, pode ocorrer redução no desempenho da aplicação quando criadas muitas *threads*, devido restrições do SO e riscos de esgotamento de recursos (SILBERSCHATZ, 2016). A nova proposta com *threads* virtuais implementada na JEP 444 (*JDK Enhancement Proposal*, forma de inserir mudanças no ambiente de execução da linguagem Java) é de que o código de uma *thread* Java continue sendo executado em uma *thread* do sistema operacional, mas que esta não seja exclusiva de uma única *thread* da JVM, mas utilizada por várias delas (PRESSLER, 2022), conforme ilustrado na figura 3.

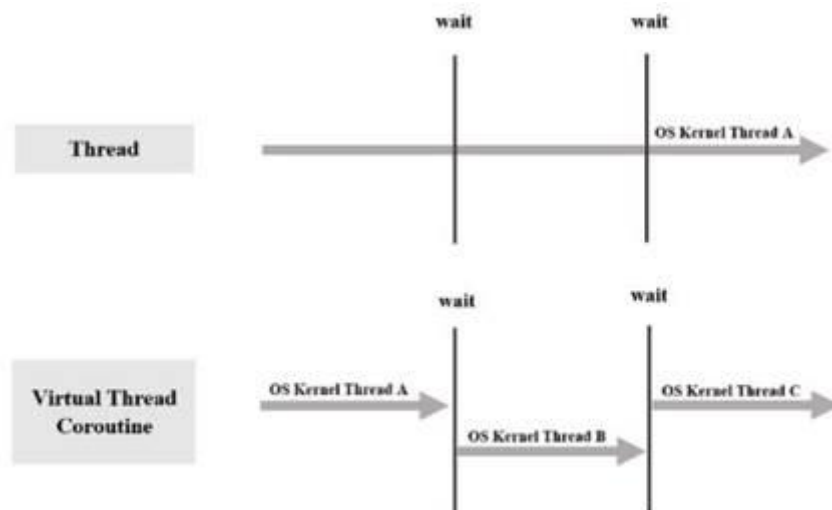
Figura 3: *Threads* virtuais (VT), *threads* de plataforma (PT) e *threads* do SO.



Fonte: Primária (2023).

Neste modelo, uma *thread* virtual pode ser executada em diferentes *threads* do sistema operacional, ocorrendo uma troca sempre que houver alguma operação que bloqueie ou que esteja esperando por resultado que não seja de processamento na CPU, ilustrado na Figura 4.

Figura 4: Relação de uma *thread* comum e de uma *thread* virtual executando na *thread* do SO.



Fonte: Beronić et al. (2022, p. 1468).

O modelo de *threads* virtuais traz alguns benefícios (GOETZ, 2022), como: as informações são armazenadas no espaço de memória gerenciado pela JVM, não mais aguardando alocação do SO; são mais fáceis de criar e destruir por não necessitar

aguardar a criação de recursos no SO hospedeiro; utilizam as mesmas APIs (classes utilitárias) do modelo tradicional e por possibilitar manter o estilo de programação síncrono (em contrapartida ao modelo assíncrono/reactivo).

A partir dessa proposta, são introduzidos novos conceitos (PRESSLER, 2022): *platform threads* (modelo tradicional de *threads*, uma *thread* da JVM é vinculada a uma *thread* do SO), *virtual threads* (*thread* que é criada e gerenciada pela JVM, mas sempre que precisar ser executada na CPU, é executada na *thread* do sistema operacional) e *carrier threads* (são *threads* criadas pela JVM no SO para executar tarefas das *virtual threads*).

2.5 Servidor Tomcat

Tomcat é um servidor de aplicações de código aberto construído em Java que possibilita, dentre outras funcionalidades, criar aplicações web que são executadas quando alguma URL mapeada for recebida através de uma requisição HTTP.

Segundo documentação do Apache Tomcat, internamente as requisições são recebidas através do componente *Connector*, que escuta na porta TCP configurada e extrai as informações contidas na requisição. Estas informações são recebidas pelo componente *Engine*, que as repassa para a *servlet* (aplicação Java) correspondente, onde são definitivamente processadas. Após ser tratada em uma das *threads* do componente *Executor*, a requisição é finalizada e a *thread* utilizada no processamento é devolvida para ser reutilizada.

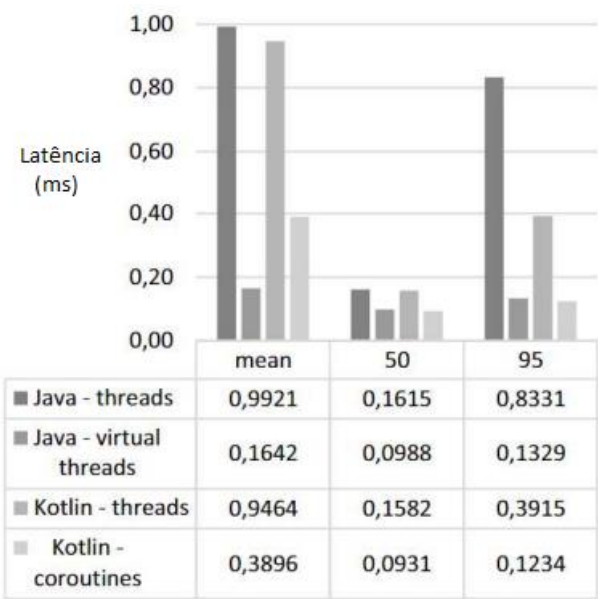
Através de arquivos de configuração é possível customizar características dos componentes mencionados. Por exemplo: por padrão, o componente *Executor*, mencionado anteriormente, é configurado contendo apenas duzentas *threads* de plataforma para processar todas as requisições recebidas, podendo ser modificado para outro modelo ou quantidade de *threads*, mudança esta que ocorre neste trabalho.

2.6 Trabalhos relacionados

Apesar das *threads* virtuais terem sido recentemente disponibilizadas em Java (2021), alguns pesquisadores já realizaram experimentos comparando com as demais formas existentes. Em um teste (Beroni^ć et al, 2021) foram executados algoritmos de ordenação e comparadas métricas como velocidade de criação de *threads* e de execução. Nos resultados coletados, observou-se que: o tempo para criar um milhão de *threads* comuns (*platform threads*) foi 2,23 vezes mais lento que a criação do mesmo número de *threads* virtuais. Quanto a execução, no algoritmo de *Merge Sort* em um *array* com 64000 elementos, observou-se que com *threads* virtuais a ordenação foi 30,82% mais rápida do que utilizando *threads* de plataforma.

Em outro trabalho (Beroni^ć et al, 2022), os pesquisadores fizeram comparações com: *threads* de plataforma e virtuais das linguagens Java e Kotlin. A comparação faz sentido visto que a linguagem Kotlin utiliza o mesmo ambiente de execução (JVM). Diferente do artigo anterior, o teste continha um servidor web que recebia as requisições e dentre as métricas colhidas, a latência observada apresentou diferença, sempre em favorecimento às *threads* virtuais, conforme Figura 5. Em ambos os testes mencionados, os pesquisadores não compararam outras métricas, como consumo de memória ou fizeram testes utilizando outras soluções de *multithreading* em Java, como através dos “*Executors*”.

Figura 5: Latência média de resposta às requisições utilizando diferentes formas de *threads*.



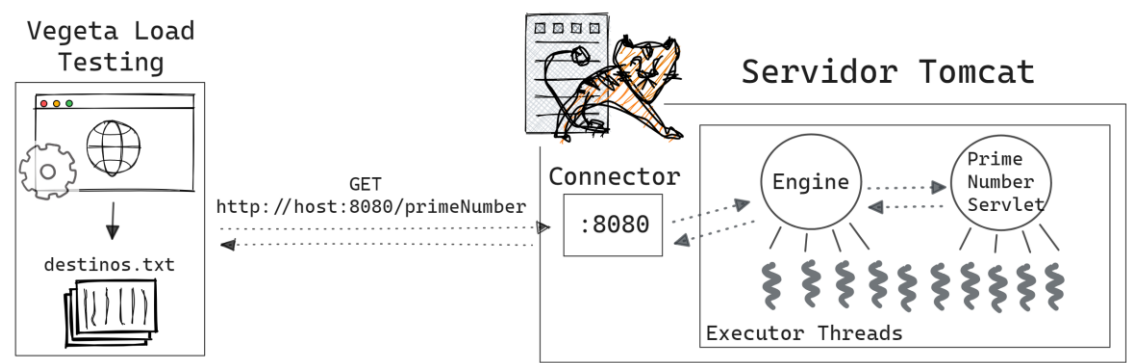
Fonte: Beronić et al. (2022, p. 1470).

3. Procedimentos metodológicos

Este trabalho faz uma análise comparativa a partir da execução de três cenários de teste: dois cenários substituindo o componente *Executor* do servidor Tomcat com tipos distintos de *threads* Java, e um cenário utilizando o componente *Executor* padrão.

Nos experimentos foi utilizado um ambiente de testes com as seguintes características: computador com sistema operacional Linux Ubuntu versão 22.04 LTS em um hardware de 8GB de memória RAM e processador Intel(R) Core (TM) i7, composto de 8 núcleos virtuais de processamento. Além disso, foram utilizados: linguagem Java na versão prévia OpenJDK 21 contendo as *threads* virtuais; a ferramenta VisualVM na versão 2.1.6 para extrair os indicadores de consumo de recursos computacionais (*profiling*); a ferramenta Vegeta na versão 7.0.0 para disparo de requisições HTTP; e o servidor Tomcat na versão 11.0.0 como servidor de aplicação, ambos sendo executados no mesmo hardware, conforme ilustrado na Figura 6.

Figura 6: Fluxograma de execução dos testes.



Fonte: Primária (2023).

O experimento segue uma estrutura de cliente/servidor, onde o cliente (ferramenta Vegeta) lê o arquivo texto contendo as URLs de destino e realiza o disparo de 5000 requisições HTTP por segundo, durante 10 segundos seguidos. O servidor Tomcat contém a aplicação Java (*servlet*) *PrimeNumberServlet*, desenvolvida neste trabalho, que é chamada para processar todas as requisições HTTP de tipo GET que correspondam a URL “/primeNumber” mapeada, processo ilustrado na Figura 4.

Internamente, o servidor Tomcat é composto de um *Connector* que escuta a porta 8080 da máquina e redireciona as requisições para o componente *Engine* sempre que receber uma requisição HTTP. Neste componente é identificada a *servlet* responsável pela requisição (a partir das URLs mapeadas) e feita a chamada do método da *servlet* para definitivamente processá-la. Ambos (*Engine* e *PrimeNumberServlet*) utilizam *threads* “Executor” disponíveis para realizar o processamento necessário para responder a requisição.

Na URL de cada requisição disparada pela ferramenta Vegeta existe um número aleatório que é lido pela *servlet* e o parâmetro “delayDuration”, onde é informado o tempo que a *thread* de execução será suspensa (instrução “*sleep()*”), em milissegundos. Nas requisições disparadas, os números utilizados foram gerados de forma aleatória com a linguagem Java, tendo valor entre 100.000 e 1.000.000 e no parâmetro que suspende a execução da *thread*, o valor de 100ms.

O método da *servlet* que processa a requisição executa um algoritmo na *thread* do *Executor* contendo os seguintes passos: (1) ler a URL recebida pelo servidor e identificar o número inteiro informado; (2) criar um objeto na memória; (3) fazer a chamada do método que verifica se o número é primo; (4) fazer a chamada de método estático responsável por escrever em arquivo “.CSV” as informações da requisição processada; (5) suspender a requisição no tempo informado na URL; (6) e escrever o retorno no objeto de resposta da requisição, contendo o resultado se o número era primo ou não, e a *string* que foi gravada no arquivo “.CSV”.

Nas etapas 4 e 5, o processamento da requisição é bloqueado, ou seja, enquanto não houver retorno, a *thread* fica aguardando o resultado da chamada executada. Estas interrupções têm por motivação simular uma situação que é recorrente em aplicações web, quando por exemplo é feita uma conexão no banco de dados e a *thread* de execução permanece aguardando o retorno da conexão realizada.

Para cada requisição recebida no servidor Tomcat, os passos descritos acima são executados. A diferença entre cada cenário está no tipo de *thread* que irá processar a requisição e sofrer as interrupções do algoritmo, sendo: Cenário 1 (com *threads* virtuais), Cenário 2 (com banco de *threads* de plataforma em cache), Cenário 3 (com *threads* do servidor Tomcat).

Para utilizar os diferentes modelos, antes do servidor Tomcat ser inicializado, o arquivo “server.xml” do diretório de configurações é substituído pelo arquivo contendo o modelo de *threads* a ser testado. Nos três cenários, a *tag* do arquivo XML que configura o componente *Connector* é alterada, visando manter a uniformidade dos testes (ambos recebem os parâmetros “maxConnections=100000” e “maxThreads=100000”, para evitar que o limite de requisições e *threads* em utilização interfira nos testes).

Semelhante ao processo realizado na pesquisa relacionada, para coleta de dados do lado do cliente a ferramenta de disparo obtém a resposta de todas as 50.000

requisições executadas (5000/segundo) e faz o cálculo do tempo médio de resposta das requisições (latência). Do servidor, a ferramenta VisualVM coleta dados do processo do servidor Tomcat no SO, extraíndo a cada segundo informações de: memória utilizada e reservada pela JVM na memória RAM (em bytes), percentual de uso de CPU e número de *threads* do sistema operacional em uso pelo processo. Ao fim da execução é exportado o arquivo “.CSV” contendo todas as informações coletadas.

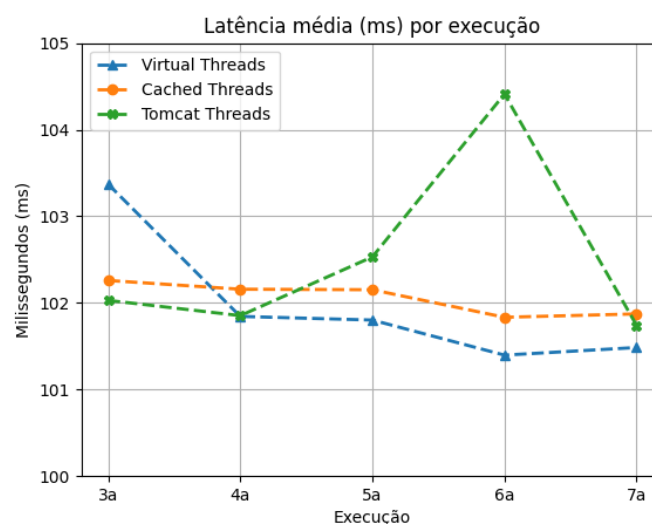
Para executar o experimento com cada tipo de *thread*, um *script* Bash Linux foi criado com as etapas: trocar o arquivo de configuração do servidor Tomcat (server.xml) contendo as informações do tipo de *thread*; iniciar o servidor, executar o disparo de requisições (5000 requisições/segundo, durante 10 segundos seguidos) sete vezes, esperando 70 segundos entre cada execução para que a JVM descarte as *threads* em cache, em cada um dos três cenários.

4. Análise dos dados e discussão dos resultados

Para visualização das informações, gráficos foram gerados com a biblioteca *Matplotlib* da linguagem Python, com base nos dados gerados pelas ferramentas *VisualVM* e *Vegeta*. Dos sete disparos de requisições realizados, os dados gerados pelas duas primeiras execuções são desconsiderados para descartar os efeitos da compilação *just-in-time* e do carregamento de classes da JVM.

Dos dados gerados pelo cliente (Vegeta), todas as requisições obtiveram retorno (código HTTP 200). No gráfico da Figura 7 se observa pouca variação na latência (média de tempo de resposta das 50.000 requisições) em cada execução por tipo de *thread*, mas na média, as execuções com *threads* virtuais apresentaram melhor performance (menor valor).

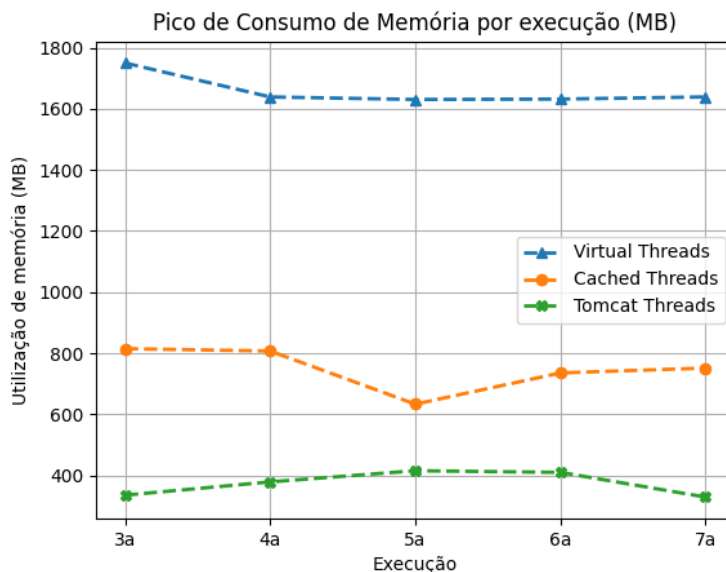
Figura 7: Latência média das requisições disparadas, para cada execução.



Fonte: Primária (2023).

Quanto aos indicadores do servidor: na Figura 8 é representado o pico de consumo de memória pelo servidor Tomcat por tipo de *thread* em cada execução, onde observa-se que as *threads* virtuais chegaram a utilizar 5,2 vezes mais memória do que o modelo tradicional de *threads* do Tomcat em uma das execuções.

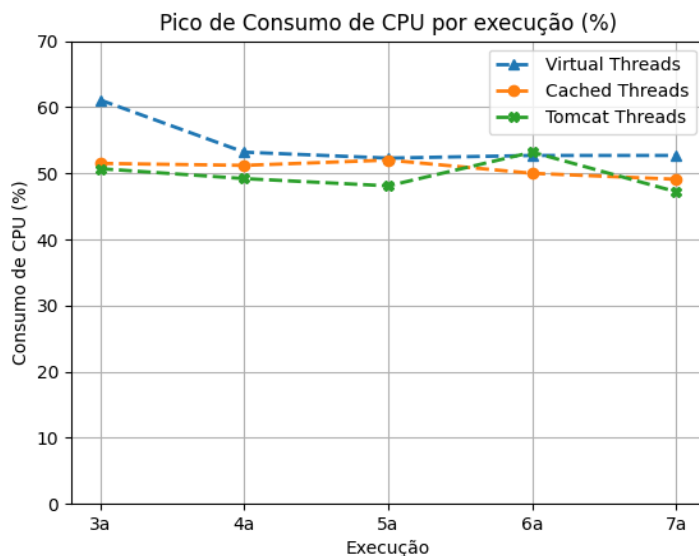
Figura 8: Pico de consumo de memória por tipo de *thread*, em cada execução.



Fonte: Primária (2023).

Quanto ao uso de CPU do dispositivo pelo servidor Tomcat, não se observou variação expressiva entre os modelos de *thread*, conforme ilustrado na Figura 9.

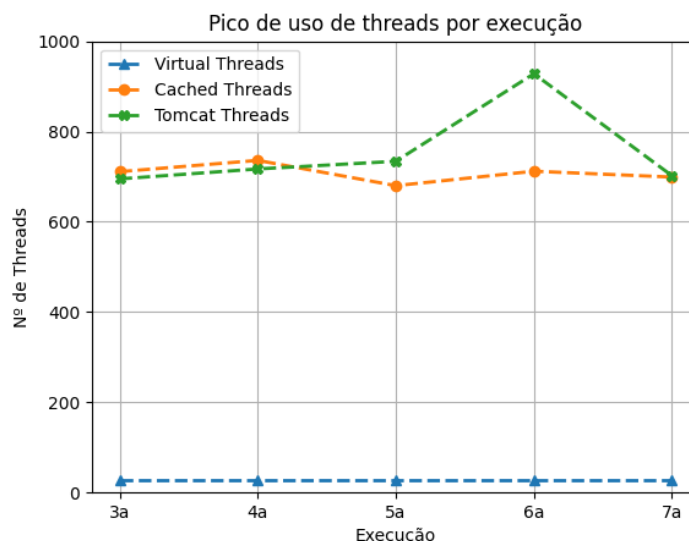
Figura 9: Pico de consumo de CPU por tipo de *thread*, em cada execução.



Fonte: Primária (2023).

Já na utilização de *threads* do sistema operacional pelo processo do servidor Tomcat, as execuções com *threads* virtuais utilizaram um número expressivamente menor quando comparado aos demais modelos de *threads*, conforme Figura 10. O modelo de *threads* virtuais manteve baixa utilização de *threads* do SO (pico de 27 *threads*) em uso pelo processo do servidor, enquanto a execução com as *threads* tradicionais do servidor Tomcat foi a que demandou maior número (pico de 928 *threads*).

Figura 10: Pico de uso de *threads* do SO por modelo de *threads* da JVM, em cada execução.



Fonte: Primária (2023).

5. Conclusão

O objetivo deste trabalho foi aplicar as *threads* virtuais no servidor Tomcat, amplamente utilizado, e experimentar se traria melhorias significativas na ferramenta quando comparada às demais formas já existentes e ao padrão utilizado pelo servidor.

Apesar de no servidor Tomcat terem sido efetuadas alterações para possibilitar o uso de *threads* virtuais na versão 10, com base no cenário executado em cada um dos casos de teste, verificou-se que as *threads* virtuais não trouxeram melhoras expressivas quando utilizadas no componente *Executor* do servidor Tomcat e comparadas ao modelo de *threads* em cache e ao já utilizado pelo servidor, seja por limitações do servidor Tomcat ou pela sobrecarga de gerenciamento das *threads* virtuais, que pode ter impactado negativamente o desempenho geral e gerado um ganho limitado ou até diminuído a eficiência.

Este estudo buscou contribuir no entendimento da aplicabilidade das *threads* virtuais em ferramentas com ampla utilização do mundo Java, como o servidor Tomcat, em contraste de outras implementações já existentes. Como a versão da linguagem com *threads* virtuais ainda está em desenvolvimento, é possível que melhoras sejam implementadas e a aplicação nas ferramentas tenha melhor resultado.

Para trabalhos futuros, abordagens executando novos cenários de teste com outras configurações do servidor Tomcat; utilizando uma versão atualizada com suporte de *threads* virtuais em outros componentes do servidor ou ainda com a versão da linguagem Java com suporte de longo prazo (LTS) contendo as *threads* virtuais, podem ser úteis para esclarecimento das vantagens e limitações do novo modelo e das demais abordagens.

Referências

APACHE TOMCAT. **Apache Tomcat 11 Documentation Index**. Disponível em: <https://tomcat.apache.org/tomcat-11.0-doc/index.html>. Acesso em 25 jun. 2023.

BERONIĆ, D.; PUFEK, B.; MIHALJEVIĆ, B.; RADOVAN, A. **On Analyzing Virtual Threads – a Structured Concurrency Model for Scalable Applications on the JVMs**, MIPRO 2021. 01 out 2021.

BERONIĆ, D.; MODRIĆ, L.; MIHALJEVIĆ, B.; RADOVAN, A. **Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines**, MIPRO 2022. 23 maio 2022.

BLOCH, Joshua. **Effective Java: Best practices for**. 3ª edição. Pearson Education Inc., 2018.

GOETZ, Brian. **Java Concurrency in Practice**. 1ª edição. Addison-Wesley 2006.

GOETZ, Brian. **Virtual Threads: New Foundations for High-Scale Java Applications**. Disponível em: <https://www.infoq.com/articles/java-virtual-threads/>. Acesso em 25 jun. 2023.

GOMES, Francisco M. **Pré-cálculo: Operações, equações, funções e trigonometria**. Disponível em: Minha Biblioteca, Cengage Learning Brasil, 2018.

HORSTMANN, Cay. **Conceitos de Computação com Java**. Grupo A, 2009. E-book. ISBN 9788577804078. Disponível em: Biblioteca Digital.

HORSTMANN, Cay. **Core Java Volume I: Fundamentals**. 12ª edição. Pearson Education Inc., 2021.

JREBEL. **JREBEL 2022 Java Developer Productivity Report**. Disponível em: <https://www.jrebel.com/resources/java-developer-productivity-report-2022>. Acesso em 25 jun. 2023.

Matplotlib. **Matplotlib: Visualization with Python**. Disponível em: <https://matplotlib.org/>. Acesso em 25 jun. 2023.

PRESSLER, Ron; BATEMAN, Alan. **JEP 425: Virtual Threads (Preview)**. Disponível em: <https://openjdk.org/jeps/425>. Acesso em 25 jun. 2023.

PRESSLER, Ron; BATEMAN, Alan. **JEP 444: Virtual Threads**. Disponível em: <https://openjdk.org/jeps/425>. Acesso em 25 jun. 2023.

SCHILDT, Herbert. **Java para iniciantes**. 6ª edição. Oracle Press, 2015.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Sistemas Operacionais com Java**. 8ª edição. Elsevier Editora, 2016.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 4ª edição. Pearson, 2016.

TIOBE INDEX. **TIOBE Index for October 2023**. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em 25 jun. 2023.

VEGETA. **Vegeta: HTTP load testing tool and library**. Disponível em: <https://github.com/tsenart/vegeta/>. Acesso em 25 jun. 2023.

VISUALVM. **VisualVM: All-in-one Java troubleshooting Tool**. Disponível em: <https://visualvm.github.io/>. Acesso em 25 jun. 2023.