TESE DE MESTRADO 2024

Comparação de Concorrência Tecnologias em Java

Elias Gustafsson, Oliver Nederlund Persson

ISSN 1650-2884

LU-CS-EX: 2024-31

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

LTH | UNIVERSIDADE DE LUND

DISSERTAÇÃO DE MESTRADO

Ciência da Computação

LU-CS-EX: 2024-31

Comparação de tecnologias de concorrência em Java

Elias Gustafsson, Oliver Nederlund Persson

Comparação de tecnologias de concorrência em Java

(Testes estruturados em um ambiente de alta carga)

Elias Gustafsson
elias@gustafsson.at

Oliver Nederlund Persson
oliver.nederlund.persson@gmail.com

17 de junho de 2024

Trabalho de dissertação de mestrado realizado na Sinch AB.

Supervisores: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se Samuel Alberius Thomas Lundström

Examinador: Flavius Gruian, flavius.gruian@cs.lth.se

Resumo

Este estudo foi conduzido com o objetivo de testar o desempenho das threads virtuais do Java em comparação com threads de plataforma e sistemas reativos no contexto de um Sistema cliente/servidor de alta carga. Como as threads virtuais só foram introduzidas no Java. 21 (2023) eles ainda não foram testados completamente. Tomamos muito cuidado para garantir que as restrições do ambiente de teste não influenciaram indevidamente os resultados. Os experimentos foram projetados para testar o desempenho de diversas maneiras, envolvendo métodos com várias combinações de operações de E/S simuladas e computacionais. tarefas.

Em relação às operações puramente computacionais, nem as threads virtuais nem os fluxos reativos superaram as threads regulares da plataforma. No entanto, quando se tratava de para operações com uso intensivo de E/S, ambas as tecnologias de alta concorrência apresentaram bom desempenho. significativamente melhor do que threads de plataforma. Os fluxos reativos utilizaram significativamente menos memória e apresentaram menos flutuações, enquanto as threads virtuais apresentaram muito latências mais baixas. Por exemplo, as latências para threads virtuais chegaram a 44%. mais curto do que para reativo no percentil 99 para os testes IO, e teve o O Reactive apresentou o melhor desempenho em três dos quatro testes de benchmark. O Reactive teve cerca de metade dos resultados. uso de memória de threads virtuais em três de quatro benchmarks, e teve o Melhor desempenho de memória em todos os quatro testes de benchmark.

Devido às suas diferentes forças e desempenho que variam com o Com base na composição dos benchmarks, não podemos concluir que uma das tecnologias seja geralmente melhor que a outra. Trata-se, na verdade, de priorizar as métricas. Em nosso relatório, fornecemos todos os dados dos benchmarks e explicamos detalhadamente a metodologia de teste. Para oferecer algumas diretrizes, podemos afirmar: que threads virtuais e reativas tiveram um desempenho melhor do que threads de plataforma no Os testes de desempenho incluíram operações de bloqueio e mostraram que as threads virtuais eram geralmente mais rápidas do que as reativas. No entanto, as threads reativas se mostraram mais estáveis e eficientes em termos de memória do que as threads virtuais e as threads de plataforma.

Palavras-chave: Reativo, Threads virtuais, Java, Teste de carga, Concorrência

Agradecimentos

Primeiramente, gostaríamos de agradecer especialmente ao nosso orientador na LTH, Jonas Skepp-Stedt, que, além de nos orientar de forma clara e concisa nesta tese, nos inspirou de maneira verdadeiramente inspiradora.
palestras durante nosso período aqui. Em segundo lugar, gostaríamos de agradecer a Sinch e aos nossos orientadores.
Samuel Alberius e Thomas Lundström, que nos receberam calorosamente e nos proporcionaram uma visão interessante do funcionamento interno de uma empresa de TI. Em terceiro lugar, agradecemos a Dora Beroniÿ por ter respondido à nossa solicitação.
perguntas sobre seus estudos anteriores nesta área. Agradecemos também a Alexan-der Svarvare pela revisão do nosso relatório e a Richard Lundberg e outros por nos fornecerem um
Introdução à análise de rastreamento de pilha.

Conteúdo

Capítulo 1
Introdução

À medida que o mundo e todos os seus habitantes e intervenientes se tornam cada vez mais conectados, sistemas eficientes para comunicação e transferência de dados tornam-se mais importantes do que nunca. Em muitas aplicações, como servidores de mensagens, a eficiência implica lidar de forma adequada com chamadas de E/S bloqueantes e um alto nível de concorrência. Uma opção bem estabelecida para código escalável e não bloqueante são os sistemas reativos, que envolvem código assíncrono baseado em fluxos de dados com publicadores e assinantes. Por exemplo, vários backends na Sinch AB, onde este estudo foi conduzido, são implementados como sistemas reativos. Embora eficientes, os sistemas reativos são considerados difíceis e, portanto, dispendiosos de programar e manter, uma vez que os desenvolvedores muitas vezes não se sentem confortáveis em abandonar a programação imperativa (Brian Olson 2019).

As threads virtuais, um novo recurso do Java 21 (Ben Weidig 2023), visam resolver os mesmos problemas que os diversos frameworks reativos, sem a necessidade de sintaxe complexa e novos paradigmas de codificação. Por serem uma implementação da interface `java.lang.Thread`, podem ser usadas de forma intercambiável com threads de plataforma regulares (threads Java tradicionais, implementadas como wrappers simples sobre threads do kernel), embora essa nem sempre seja a melhor opção, como explicaremos na seção de contexto. As threads virtuais são desacopladas das threads do kernel do sistema operacional, sendo gerenciadas pela Máquina Virtual Java (JVM), que monta e desmonta as threads virtuais das threads de plataforma. Nesse contexto, montar uma thread virtual significa mover seus estados e variáveis locais para uma thread de plataforma, e desmontar significa mover os estados e variáveis locais da thread virtual para o heap, permitindo que outra thread virtual seja montada na thread de plataforma. Quando um método bloqueante é chamado a partir de uma thread virtual, ela é desmontada de sua thread de kernel e salva no heap durante o período de espera, tudo sem intervenção do programador.

Este estudo visa comparar o desempenho de sistemas reativos, threads virtuais e threads de plataforma, especificamente para um sistema com alta carga dinâmica, que é o principal caso de uso para ambas as metodologias. Como o Java 21 foi lançado apenas em 2023, o conjunto de pesquisas sobre suas threads virtuais ainda é muito escasso, o que justifica a realização deste estudo.
Existe alguma pesquisa limitada sobre threads virtuais que serve como parte da base teórica deste estudo e é apresentada na seção 1.3. Para tornar os testes o mais justos possível, os servidores reativos e baseados em threads virtuais utilizam o Spring Boot.

Os testes de desempenho, realizados por meio de frameworks HTTP (utilizado pela Sinch AB para testar seus servidores), implementam as mesmas funções. Eles são executados em um ambiente controlado, e as estatísticas do sistema são registradas com o VisualVM (Oracle Corporation 2024b). Diversos parâmetros são relevantes para medir e comparar o desempenho dessas tecnologias. Neste estudo, analisamos detalhadamente a latência, a taxa de transferência, o uso de memória e a utilização da CPU. Essas métricas são abordadas em grande parte das pesquisas anteriores citadas neste relatório. Consideramos também que elas são intuitivamente relevantes, visto que a latência e a taxa de transferência são importantes para a experiência do usuário, e o uso de memória e a utilização da CPU são importantes para o dimensionamento do hardware do servidor. É dada especial atenção para garantir que o software de teste e a conexão cliente-servidor tenham um impacto mínimo nas medições.

1.1 Questões de pesquisa

Pequenos ganhos de eficiência em softwares que lidam com altas cargas (como servidores) podem levar a um aumento na satisfação e fidelização do cliente (Kim et al., 2007), e acreditamos que softwares mais eficientes são geralmente desejáveis. Para otimizar sistemas de software com alta carga, o Java 21 introduziu threads virtuais em 2023. Este é um projeto promissor destinado a "aplicações concorrentes de alto desempenho, especialmente aquelas que consistem em um grande número de tarefas concorrentes que passam grande parte do tempo em espera" (Oracle Corporation, 2023). Se as threads virtuais apresentarem desempenho similar ou superior à abordagem anterior de programação assíncrona, isso será muito interessante. Mesmo que seu desempenho seja similar ao de outras tecnologias, elas também podem oferecer muitas outras vantagens, como softwares mais fáceis de ler, escrever e depurar.

Até o momento, existem poucos estudos comparando threads virtuais com outras abordagens de concorrência em Java devido à sua recente introdução. Além disso, acreditamos que muitos desses estudos comparativos são realizados por empresas e não são publicados. Existem estudos que comparam versões beta de threads virtuais em Java com threads de plataforma (Beroniÿ et al. 2022) (Beroniÿ et al. 2021) (Pufek et al. 2020) e um estudo que compara threads virtuais com uma abordagem reativa em Quarkus usando Java 21 (Navarro et al. 2023). No entanto, constatamos que esses estudos têm um escopo um tanto limitado. Por exemplo, alguns limitam o número máximo de threads de plataforma e a maioria não testa uma ampla gama de métodos (por exemplo, combinações de testes computacionais e de E/S que podem desfavorecer indevidamente algumas das tecnologias).
Outro aspecto que poderíamos adicionar a este tema de pesquisa são testes realizados em uma ampla variedade de cargas. Em resumo, **o objetivo deste estudo é comparar threads virtuais, threads de plataforma e sistemas reativos para aplicações de alta concorrência.** As questões de pesquisa desta tese são:

- **RQ1:** Quais são as diferenças entre essas diferentes técnicas de concorrência dentro da JVM em sistemas de alta carga?
- **RQ2:** Os threads virtuais são uma alternativa viável para substituir os threads de plataforma e os fluxos de dados assíncronos em aplicações de alta carga?

1.2 Justificativa do nosso estudo

Como as threads virtuais são uma adição muito recente à JVM, o conjunto de pesquisas sobre elas é relativamente escasso. É ainda mais escasso em comparações entre threads virtuais e programação assíncrona.

programação, que consideramos particularmente relevante, visto que a programação assíncrona tem casos de uso muito semelhantes aos das threads virtuais. Também acreditamos que podemos contribuir bastante com a metodologia de teste de desempenho para threads virtuais, uma vez que os estudos que citamos geralmente não divulgaram os detalhes de suas configurações de teste e não realizaram um conjunto de testes particularmente amplo e variado.

Reunimos uma grande quantidade de dados utilizando scripts que executam nossos casos de teste diversas vezes com diferentes parâmetros. Uma grande quantidade de dados é absolutamente necessária para que nossos testes sejam conclusivos e nossa pesquisa seja relevante. Este conjunto abrangente de dados representa uma importante contribuição para esta área.

1.3 Pesquisas Anteriores

O conjunto de pesquisas anteriores sobre comparações entre sistemas reativos e sistemas baseados em threads virtuais ainda é relativamente escasso; no entanto, nos beneficiamos muito de pesquisas que estudaram cada uma dessas tecnologias separadamente. Em especial, as metodologias de estudo e medição do desempenho de sistemas de alta concorrência têm sido muito aplicáveis a este estudo.

1.3.1 Construções de Concorrência Estruturada (2022)

No estudo “Comparação de Construções de Concorrência Estruturada em Java e Kotlin - Threads Virtuais e Corrotinas” (Beroniÿ et al. 2022), os autores comparam construções de concorrência em linguagens implementadas na JVM, incluindo Java. Os testes foram realizados em uma máquina com Ubuntu, 16 GB de memória e um processador Intel i7 (3,4 GHz), utilizando o JDK-19 do OpenJDK, uma versão de acesso antecipado com suporte a threads virtuais. O caso de teste consistia em um servidor HTTP aguardando requisições padrão. O estudo testou quatro conceitos diferentes, dois dos quais foram implementados em Java: threads virtuais e threads de plataforma.
As principais métricas utilizadas foram memória heap, latência e número de threads do kernel iniciadas. As ferramentas usadas para os testes foram o Vegeta (utilizado para testes de aplicações que podem realizar ataques em um servidor HTTP, conforme descrito na seção de contexto) e o VisualVM. O procedimento de teste consistiu em usar o Vegeta para realizar ataques com duas taxas diferentes (2000 e 8000 requisições por segundo) durante 10 segundos. Também foram realizados testes preliminares para aquecer o ambiente de teste. As latências foram calculadas pelo Vegeta e o VisualVM foi usado para coletar os dados restantes.
medidas.

O primeiro resultado apresentado foi o número total de threads iniciadas. Para 2000 ataques por segundo, a média de threads do kernel iniciadas foi de 101.017 para threads regulares da plataforma Java e 36 para threads virtuais Java. Para uma taxa de ataque de 8000, a média de threads iniciadas foi de 388.749 para threads regulares Java e 268 para threads virtuais Java.

O segundo resultado apresentado foi o uso de memória heap. Com uma taxa de ataque de 2000 requisições por segundo, as threads Java regulares utilizaram 128,28 MB, enquanto as threads virtuais utilizaram 16,22 MB. Com uma taxa de 8000 requisições por segundo, as threads Java regulares utilizaram 386,08 MB, enquanto as threads virtuais utilizaram 16,22 MB.
64,66 MB.

O terceiro e último teste mediu a latência em milissegundos. Para isso, os pesquisadores utilizaram a média, o 50º percentil e o 95º percentil. Para 8000 requisições por segundo, os resultados de latência para threads da plataforma foram 0,99, 0,16 e 0,83, respectivamente. Para threads virtuais, os valores foram 0,16, 0,10 e 0,13. Isso indica uma diferença significativa entre os percentis.

Os autores concluíram que as threads virtuais são promissoras, mas que testes mais extensivos precisam ser realizados na versão oficial. Alguns elementos deste estudo que decidimos incorporar foram a inclusão de métricas de uso de memória e latência, bem como a ferramenta de teste Vegeta. No entanto, coletamos um conjunto maior de métricas em um espectro mais amplo de cargas, além de realizar diversas verificações em nosso ambiente de teste.

1.3.2 Sobre a análise de threads virtuais (2021)

O estudo “On Analyzing Virtual Threads - a Structured Concurrency Model for Scaleable Applications on the JVM” (Beroniÿ et al. 2021) visa explorar a concorrência estruturada em Java com foco em threads virtuais. Os autores argumentam que operações de bloqueio, como E/S, causam problemas de eficiência devido à sua "política de acesso único". Essas threads dependem de threads do kernel, que requerem muitos recursos, levando a um uso ineficiente do hardware.
Eles testaram como os threads virtuais se comparavam aos threads da plataforma.

O ambiente de teste foi hospedado em uma máquina com Ubuntu, processador de 8 núcleos e 13 GB de memória. Foi utilizada uma versão inicial do JDK loom-17ea.2 (projeto loom), que suporta threads virtuais. Para comparar os resultados, foram utilizados os parâmetros de velocidade de criação, eficiência e nível de concorrência. Os testes consistiram em um algoritmo de ordenação por intercalação (merge sort), além de algoritmos genéricos de execução paralela e sequencial.

Para medir a velocidade de criação, eles usaram a latência medida em milissegundos. O teste simplesmente criou múltiplas threads com diferentes escalas. As quatro escalas diferentes foram 0,1, 0,5, 1 e 1,5 milhão. O resultado foi que a criação de threads regulares foi 1,7, 2,2 e 2,7 vezes mais lenta do que a criação de threads virtuais para 0,1, 1 e 1,5 milhão de threads criadas, respectivamente. Além disso, a criação de threads, tanto regulares quanto virtuais, pareceu escalar linearmente.

Threads regulares também foram testadas em comparação com threads virtuais em termos de desempenho. O teste foi realizado utilizando um algoritmo de ordenação por intercalação (merge sort), que ordenava arrays com 1000 a 512 000 elementos. Eles mediram quanto tempo threads regulares e threads virtuais levariam para executar o algoritmo, com ambas as abordagens tendo acesso a 16 threads do kernel. O resultado foi que, para escalas menores (até 60 mil elementos), as threads virtuais superaram as threads da plataforma em 80 vezes. A diferença diminuiu quando o array ficou maior, mas, em média, as threads virtuais apresentaram um desempenho 30% melhor.

Para testar o nível de concorrência, eles mediram a latência para a execução de uma tarefa não especificada, aumentando gradualmente o número de threads que representavam as requisições. O número inicial de threads era de cem mil e foi gradualmente aumentado para dez milhões devido ao aumento do número de requisições. Para ambos os tipos de threads, as latências aumentaram com o aumento do número de requisições. Com 2500 requisições, as threads virtuais superaram as threads da plataforma em 38%, mas, à medida que as requisições aumentaram para dez mil, a superação das threads da plataforma chegou a 591%. A partir disso, os autores concluíram que as threads virtuais deveriam ser capazes de lidar com uma carga maior, dada a mesma quantidade de recursos de CPU.

Alguns elementos deste estudo que influenciaram o nosso foram o benchmark de ordenação por intercalação (merge sort) e os dados de concorrência. O benchmark de ordenação por intercalação serviu de inspiração para um benchmark de multiplicação de matrizes que utilizamos, e o desempenho superior das threads virtuais serviu de justificativa para investigá-las mais a fundo.

1.3.3 Concorrência Estruturada Eficiente através de Fibras leves (2020)

O estudo “Alcançando Concorrência Estruturada Eficiente por meio de Fibras Leves na Máquina Virtual Java” (Pufek et al., 2020) explora threads de fibra Java (um nome provisório para threads virtuais). Os autores argumentam que sistemas que lidam com requisições de entrada tendem a usar threads do kernel de forma inadequada, desperdiçando muitos recursos. O estudo investiga técnicas de concorrência estruturada buscando métodos mais eficientes para lidar com sistemas de alta carga.

Os testes foram realizados em uma máquina virtual executando Ubuntu com 9 GB de memória.
Para as threads de fibra, eles usaram uma versão de acesso antecipado do OpenJDK. Assim como no estudo anterior, usaram o Vegeta para simular a carga. O lado do servidor da configuração de teste era uma implementação de servidor HTTP fornecida no JDK. O número de threads foi limitado a 64.

No teste, cada thread recebeu uma tarefa na qual foi adicionado um atraso (Thread.sleep) para simular artificialmente a carga. Em seguida, a latência foi medida para diferentes durações de atraso. Sob carga pesada, o resultado foi que a latência para threads virtuais foi 16% da latência para threads da plataforma. Os autores concluíram o relatório afirmando que as threads virtuais terão um papel importante no futuro, mas que são necessárias mais pesquisas devido ao seu caráter experimental.

A simulação de operações de E/S como atrasos simples, bem como com durações de atraso variáveis, realizada no estudo, foi algo que decidimos incluir também em nossos experimentos. Os argumentos apresentados por eles sobre como um alto nível de concorrência poderia exigir técnicas mais especializadas também serviram como justificativa para o nosso estudo.

1.3.4 Integração de threads virtuais em um framework Java (2023)

Um estudo recente sobre o assunto de threads virtuais em Java foi publicado recentemente (Navarro et al. (2023). O estudo visa investigar threads reativas e virtuais na JVM e integrá-las ao framework Quarkus, oferecendo assim uma abordagem para a integração de threads virtuais em frameworks reativos Java. O Quarkus foi escolhido por ser otimizado para ambientes de nuvem, apresentando baixo consumo de recursos em comparação com frameworks como o Spring Boot, devido à sua capacidade de compilação nativa e tempo de execução otimizado. Os autores também defenderam a escolha do Quarkus em detrimento do framework Spring, visto que a implementação de servidores de threads virtuais do Spring não permite a execução simultânea de threads virtuais e sistemas reativos (implementados no Spring Webflux) na mesma aplicação. O Quarkus permite a utilização das três tecnologias na mesma aplicação.

A parte experimental do estudo foi conduzida em uma máquina com um processador i7-10850H de doze núcleos (2,7 GHz) e 32 GB de memória, rodando Ubuntu. A aplicação foi dividida, com servidores e clientes executados em contêineres Docker com restrições de CPU e memória. O estudo implementou testes de carga para as três tecnologias diferentes, onde latência, throughput, uso de CPU e uso de heap foram medidos para diferentes taxas de requisição.

As funções do servidor utilizadas no estudo foram inspiradas no teste Fortune da Techempower.
O conjunto de testes de desempenho consiste em buscar um objeto em um banco de dados e adicioná-lo a uma lista, que em seguida é ordenada e um objeto JSON é retornado. Neste estudo, um atraso adicional foi adicionado para aumentar o tamanho do heap. Os autores do estudo também descreveram diferentes condições de operação para as diferentes funções do servidor, que denominaram: normal (taxa de transferência igual à taxa de requisições, uso da CPU inferior a 80%), crítica (taxa de transferência igual à taxa de requisições, mas com aumento da latência e

aumento do uso de memória) e sobrecarga (taxa de transferência inferior à taxa e alta utilização de recursos).

O resultado foi que, sem atraso adicional, a abordagem reativa apresentou a menor latência, a maior taxa de transferência e a menor utilização de recursos. Os threads da plataforma também superaram os threads virtuais. Com um atraso constante de 200 ms, a abordagem reativa ainda apresentou o melhor desempenho. A comparação entre threads bloqueantes e virtuais, no entanto, foi mais complexa.
Os threads virtuais apresentaram maior uso geral de CPU e memória, e maior taxa de transferência do que os threads de bloqueio até atingir um nível crítico e travar. As conclusões que tiraram disso foram:

- **i)** Que os threads virtuais no Quarkus atingiram rapidamente um platô em relação ao uso de memória.
- **ii)** Para operações de bloqueio mais curtas, os threads virtuais são computacionalmente mais dispendiosos do que os threads da plataforma.
- **iii)** Dentre essas tecnologias, as threads reativas apresentam a melhor utilização de recursos.

Após comparar as diferentes tecnologias, os autores utilizaram uma taxa de requisição fixa e compararam threads virtuais com a abordagem reativa. Eles levantaram a hipótese de que o coletor de lixo era o principal motivo da diferença entre as tecnologias.
Neste experimento, eles coletaram métricas do coletor de lixo, incluindo contagem, média de pausas, pausas de coleta e soma de pausas. Concluíram que as threads virtuais apresentavam um uso muito maior do coletor de lixo, o que era a causa principal de seu desempenho inferior em comparação com a abordagem reativa do Quarkus.

Um aspecto deste estudo que pode ser facilmente aplicado ao nosso são os procedimentos para evitar a introdução de ruído nos dados de teste, o que causaria uma comparação injusta entre as tecnologias.
Os autores realizaram execuções de aquecimento antes de cada teste para garantir que as classes apropriadas tivessem sido carregadas e que o compilador just-in-time (JIT) tivesse otimizado os métodos mais acessados. Além disso, executaram cada teste três vezes. Mesmo tomando essas precauções, os testes ainda se mostraram um tanto instáveis e imprevisíveis. Eles atribuíram essa instabilidade ao "caos" introduzido pelo coletor de lixo e pelo compilador JIT. Por isso, optaram por escolher o melhor dos três testes em cada execução, em vez de calcular a média dos resultados. O uso de um banco de dados também pode ser uma desvantagem, segundo os autores. Os benchmarks relativamente complexos utilizados podem ter introduzido ruído adicional nos resultados, o que pode ser observado nos resultados apresentados, onde o desempenho dos servidores apresenta quedas inesperadas em alguns momentos. Além disso, o teste de benchmark utilizado neste estudo é um tanto injusto, pois força o servidor a usar muita memória, o que não é o caso de uso pretendido para threads virtuais (como claramente declarado no projeto Loom, nome do projeto para a implementação de threads virtuais do Java).

1.4 Distribuição do Trabalho

A distribuição do trabalho entre os autores foi praticamente igualitária. Cada etapa, tanto da metodologia experimental quanto da redação do relatório, foi objeto de intensos debates e revisões minuciosas. Bons resultados são, em geral, fruto de discussões participativas e diferentes perspectivas.

Existem, no entanto, algumas áreas em que uma pessoa contribuiu mais do que a outra. Por exemplo, Oliver Nederlund Persson contribuiu mais para a implementação do cliente e dos casos de teste automatizados, enquanto Elias Gustafsson contribuiu mais para a implementação de

o servidor e os testes na conexão de rede. Oliver também é mais responsável por encontrar e compilar pesquisas anteriores, enquanto Elias se concentrou mais na JVM.

Capítulo 2
Fundo

Neste capítulo, apresentamos alguns fundamentos teóricos sobre os conceitos centrais deste estudo.
Resumimos também alguns estudos selecionados sobre temas semelhantes.

2.1 Conceitos Centrais

Aqui apresentamos os conceitos importantes deste estudo, incluindo tudo o que o leitor precisa saber para aproveitar nossos resultados e conclusões. Introduzimos threads virtuais e programação reativa, com as quais o leitor pode não estar familiarizado, bem como o framework Spring, no qual os servidores foram implementados.

2.1.1 Threads Virtuais

O conceito de threads virtuais existe há muito tempo. (Vahalia 1996, p. 53-55) descreve um *processo leve* (LWP) como um processo mais desacoplado das threads do kernel do que, por exemplo, uma thread de plataforma. Os LWPs não bloqueiam as threads do kernel quando estão bloqueados devido a operações de E/S. A limitação dos LWPs é que eles exigem chamadas de sistema dispendiosas para sincronização, criação e destruição. No entanto, múltiplas *threads de usuário* (Vahalia 1996, p. 55-58) podem ser posteriormente montadas em LWPs ou processos regulares. *Threads de usuário* são uma abstração de thread de alto nível no nível do usuário, gerenciada por bibliotecas sem que o kernel tenha conhecimento delas. Para threads de usuário, a biblioteca gerencia agendamentos e trocas de contexto, além de salvar a pilha individual *da thread de usuário* . Contudo, o kernel ainda gerencia o agendamento entre os processos ou os LWPs. Se a *thread do usuário* estiver montada em um LWP (Processador de Linguagem de Trabalho), um bloqueio em nível de usuário (por exemplo, E/S) não bloqueará a thread do kernel (apenas o LWP). *Threads de usuário* que não bloqueiam o processo subjacente, as threads virtuais, foram implementadas na linguagem Go como Goroutines e agora no Java 21 como threads virtuais.

Os threads virtuais em Java foram propostos na Proposta de Aprimoramento 425 do JDK (JEP) e implementados como um recurso de pré-visualização no Java 19 e posteriormente no Java 20. Foram finalizados no Java 21. Os threads virtuais não se destinam a substituir os threads existentes nem os threads assíncronos existentes.

estilos cronológicos. Em vez disso, eles são concebidos como uma ferramenta para implementar a programação regular 'uma thread por tarefa', operando de forma não bloqueante.

Na JEP 444, os autores mencionaram como os aplicativos de servidor tendem a lidar com múltiplos usuários independentes entre si. Dedicar uma thread por usuário é uma maneira simples de lidar com a concorrência, mas essa abordagem não é escalável com threads de plataforma comuns. Isso ocorre porque essas threads Java tradicionais são implementadas pela JVM como wrappers em torno de threads do kernel. Portanto, o sistema subjacente (hardware e SO) dita quantas requisições paralelas podem ser processadas simultaneamente, o que geralmente impõe um limite baixo à concorrência, já que as requisições ficam presas aguardando operações de bloqueio. Um estilo de programação assíncrona pode ser usado para lidar com isso. No entanto, como afirmado na JEP 444, esse estilo de programação introduz alguns problemas e dificulta a depuração.

Em Java, as threads virtuais são instâncias de `java.lang.thread`, assim como as threads de plataforma. Uma thread de plataforma é executada em uma thread do kernel e a captura durante toda a duração do código, enquanto as threads virtuais não. Como as threads virtuais são implementações da interface `java.lang.thread`, elas podem ser consideradas uma maneira amigável de habilitar alta concorrência, já que a maioria dos desenvolvedores Java provavelmente está familiarizada com a interface. Toda a troca de contexto relacionada a threads virtuais e de plataforma é tratada automaticamente, e o programador não deve tratar as threads virtuais de forma diferente das threads de plataforma. Assim como a memória virtual simula recursos abundantes mapeando um grande número de endereços para uma memória menor, um grande número de threads virtuais é mapeado para um número menor de threads do kernel.

As threads de plataforma do Java dependem do agendador do sistema operacional, enquanto as threads virtuais são gerenciadas pela JVM, que as atribui às threads de plataforma. As threads virtuais não estão restritas a uma thread de plataforma específica durante seu ciclo de vida e podem ser atribuídas a diferentes threads de plataforma pelo agendador após cada troca de contexto. Uma thread virtual é desmontada de suas threads de plataforma quando está bloqueada (por exemplo, em operações de E/S), o que garante que as threads de plataforma subjacentes não sejam bloqueadas. No entanto, a JVM nem sempre desmonta as threads virtuais. Duas situações em que as threads virtuais não são desmontadas são: certas situações de bloqueio e fixação. A primeira situação deve-se a limitações da JVM e do sistema operacional, em que a JVM compensa aumentando temporariamente o paralelismo, ou seja, criando mais threads de plataforma. O segundo cenário, chamado de fixação, ocorre quando operações de bloqueio são realizadas dentro de um bloco sincronizado. Em ambas as situações, a thread de plataforma subjacente também é bloqueada (Oracle Corporation 2023).

Os frames de pilha das threads virtuais são armazenados no heap, que é coletado pelo coletor de lixo, entre as trocas de contexto. O heap cresce e diminui dinamicamente. Isso permite que o programa execute muitas threads virtuais. Há também suporte para variáveis locais de thread, mas recomenda-se evitar seu uso, pois o número de threads virtuais pode crescer muito, o que pode causar erros relacionados à memória à medida que o uso do heap aumenta. De acordo com a documentação do Java 21, uma única JVM pode potencialmente lidar com milhões de threads virtuais (Oracle Corporation 2024a) e os frameworks usados neste estudo não impõem limites ao número de threads virtuais criadas.

2.1.2 Sistemas Reativos

Existem diversas implementações de sistemas reativos, mas um denominador comum é que elas se baseiam em fluxos de dados assíncronos. Ao descrever sistemas reativos, o Manifesto Reativo (Jonas Bonér 2014) é geralmente citado. Escrito e assinado por diversos profissionais de TI, ele afirma que os sistemas reativos devem ser flexíveis, modulares e facilmente escaláveis. O manifesto enumera quatro propriedades distintas que os sistemas reativos devem possuir:

- **Orientado a mensagens:** A troca assíncrona de mensagens permite a compartimentalização e
  O monitoramento dos fluxos de mensagens facilita a aplicação de medidas como o gerenciamento de carga e...
  contrapressão. Os componentes podem ser editores ou assinantes nesses sistemas, e
  Os assinantes reagem aos dados fornecidos pelos editores.

- **Responsivo:** A capacidade de resposta de um sistema é fundamental para torná-lo fácil de usar e
  Portanto, utilizável, um sistema reativo deve se concentrar na capacidade de resposta. Como um adicional
  Além disso, isso também simplifica o tratamento de erros.

- **Elástico:** Um sistema reativo deve ser elástico, ou seja, responsivo e eficiente.
  não deve degradar-se com uma carga de trabalho variável. Vários métodos são empregados para atingir esse objetivo.
  isto, por exemplo, a contrapressão.

- **Resiliente:** Um sistema reativo deve permanecer responsivo mesmo em caso de falhas. Os componentes devem ser
  compartimentados e deve haver uma maneira bem definida de lidar com elas.
  Recuperação sem delegar o tratamento de erros aos clientes dos componentes com falha.

A programação assíncrona orientada a eventos é frequentemente considerada um processo difícil.
Implementar e manter, e o código resultante é frequentemente difícil de depurar e entender.
(Madsen et al. 2017), (Kambona et al. 2013). Código não imperativo é mais difícil de depurar, não.
não apenas por razões óbvias, como a dificuldade em inserir pontos de interrupção, mas também porque a maioria
Os recursos de depuração automatizada têm dificuldade em lidar com código assíncrono. O fenômeno
A programação assíncrona tem sido descrita como um "inferno de callbacks" (Edwards 2009), (Belson
et al. 2019), (Brodu et al. 2015). O código resultante foi comparado ao 'assíncrono'.
espaguete' e 'goto' moderno (Kambona et al. 2013).

Nas listagens 2.2 e 2.1, são ilustradas as diferenças de implementação entre serviços reativos e serviços síncronos
(mesmo código para threads virtuais e threads de plataforma).
Nestes exemplos, os serviços realizam uma chamada bloqueante para buscar um array em um banco de dados, converter os
valores em inteiros, classificá-los e remover todas as entradas com um determinado ID. Para o serviço reativo,
Para que um serviço funcione de forma assíncrona, a implementação geralmente deve ser feita na forma de callbacks.
resultando em longas linhas de código, como na programação funcional. Enquanto isso, a implementação de threads virtuais
e de plataforma segue uma abordagem mais imperativa que pode torná-la
significativamente mais fácil de implementar e de ler.

```
1 importar reator. núcleo. editor. Fluxo;
2 importar reator. núcleo. editor. Mono;
3 @Serviço
4 public class ReactiveService {
5         @Autowired
6         Repositório de dados reativo privado ;
7         public Flux < Data > fetchDataConvertSortAndFilter () {
8                 Flux < Data > dataFlux = reactiveDataRepository . findAll () ;
9                 retornar dataFlux.flatMap(dados -> {
10                                                 tentar {
11                                                         valor inteiro = Integer.parseInt(dados.
      getValue () ) ;
12                                                         dados.setValue(String.valueOf(valor)
      ) ;
13                                                         retornar Mono.just(dados);
14                                                 } catch ( NumberFormatException e ) {
15                                                         retornar Mono.vazio();
16                                                 }
17                                         })
```

```
                                        . filtro (dados -> dados. getId ()! = 31)
                                        .collectSortedList(Comparator.comparingInt(
        dados -> Inteiro.parseInt(dados.getValue())))
                                        . flatMapMany ( Flux :: fromIterable ) ;
        }
}
```

Listagem 2.1: Serviço reativo

```
@Serviço
public class RegularThreadService {
        @Autowired
        Repositório de dados privado dataRepository;
        public List<Data> fetchDataConvertSortAndFilter() {
                Lista<Dados> listaDeDados = repositórioDeDados.findAll();
                para ( Dados dados : listaDeDados ) {
                        tentar {
                                int value = Integer.parseInt(data.getValue());
                                dados.setValue(String.valueOf(valor));
                        } catch ( NumberFormatException e ) {
                                retornar nulo ;
                        }
                }
                Collections.sort(dataList, Comparator.comparingInt(data ->
        Inteiro.parseInt(dados.getValue())));
                dataList.removeIf(data->data.getId() == 31);
                retornar listaDeDados;
        }
}
```

Listagem 2.2: Roscas

2.2 Framework Spring

O framework Spring (Spring 2024) é usado para construir aplicações Java de nível empresarial.
O Spring oferece funcionalidades e recursos que facilitam o desenvolvimento de sistemas robustos e escaláveis.
e aplicações de fácil manutenção. O framework lida com injeção de dependência, o que promove
um acoplamento fraco entre os componentes da aplicação, externalizando suas dependências.
O Spring também oferece funcionalidades como acesso a dados (integração com bancos de dados), segurança,
Testando o padrão Model-View-Controller (MVC), WebFlux e Spring Boot. MVC é um framework.
que auxilia na construção de aplicações web e fornece componentes para gerenciar o estado da sessão.
e requisições HTTP. O Spring Boot é um framework que estende o framework Spring, fornecendo
autoconfiguração e algumas funções padrão que auxiliam na construção de aplicações prontas para produção.
O WebFlux é um framework de programação reativa que permite requisições não bloqueantes.
e concorrência assíncrona.

Em suma, o framework Spring permite o desenvolvimento de aplicações robustas e prontas para produção.
Ele é amplamente utilizado na indústria, inclusive pela empresa de hospedagem, e, portanto, será usado para
implementar os servidores neste projeto. A razão para isso é que o framework oferece uma boa base de recursos.
base para comparação entre as diferentes abordagens de concorrência, reduzindo as potenciais diferenças
entre elas em comparação com o que ocorreria se essas abordagens tivessem sido criadas inteiramente por
os autores.

2.2.1 Bota de mola

O Spring Boot (Spring 2024) é usado para simplificar o processo de configuração, implantação e construção de aplicações. Ele inclui aplicações capazes de lidar com requisições HTTP. Possui três características principais: autoconfiguração, a capacidade de criar aplicações independentes e uma abordagem opinativa para configuração. A autoconfiguração reduz a necessidade de o desenvolvedor realizar certas configurações relacionadas a dependências, embora seja possível sobrescrever essas autoconfigurações. A abordagem opinativa significa que o Spring Boot adiciona dependências e configurações iniciais com base nas necessidades do projeto. Aplicações independentes são aquelas que funcionam sem depender de um servidor web externo.

2.2.2 Spring WebFlux

Spring WebFlux é um framework reativo, não bloqueante e assíncrono (Spring 2024). O WebFlux é baseado no projeto Reactor e implementa fluxos reativos. O objetivo dos fluxos reativos é lidar com dados "ao vivo" com volume desconhecido. Isso é feito controlando a troca de dados através de limites assíncronos (dados entre threads e pools de threads) sem que o receptor precise armazenar em buffer uma quantidade desconhecida de dados. Um exemplo é a contrapressão, onde o consumidor do fluxo reativo (dados) controla a taxa na qual recebe dados, evitando assim uma possível sobrecarga (Reactive Streams 2024).

O Spring WebFlux oferece um cliente web reativo que lida com requisições HTTP de forma puramente assíncrona, sem bloqueio de E/S. Suas aplicações do lado do cliente permitem requisições de saída para serviços externos ou APIs sem bloqueio, graças ao aproveitamento de fluxos reativos e ao gerenciamento eficiente da contrapressão. Ele oferece uma variedade de operações HTTP (como POST e GET), além de suporte para tratamento de erros e integração com outras aplicações Spring.

2.2.3 Framework web MVC Spring

Este framework foi projetado para construir aplicações web seguindo o padrão MVC (Spring 2024). Ele inclui três componentes: modelo, visão e controlador. No contexto deste projeto, o controlador é o componente principal. Quando uma requisição é recebida, o servlet dispatcher do Spring intercepta a requisição e a encaminha para o controlador correto. O controlador lida com as requisições HTTP recebidas dos clientes. Dentro do controlador, existe um mapeamento de requisições que associa uma determinada requisição a uma função manipuladora específica, por exemplo, POST '/caminho'.

Este framework oferece um método simples para criar servidores capazes de lidar com requisições HTTP de forma eficiente. É compatível com Java 21 e oferece suporte tanto para threads de plataforma regulares quanto para threads virtuais através de um processo de configuração simples.

2.2.4 Resumo primavera

Em resumo, o framework Spring oferece uma maneira simples de criar aplicações prontas para produção, incluindo clientes web. O framework será utilizado neste projeto devido a esses principais benefícios. motivos:

- **i)** Trata-se de uma estrutura comercial, portanto, ao utilizá-la em vez de implementar algo por conta própria, reduzimos o risco de as implementações não serem igualmente boas para todas as tecnologias testadas.

- **ii)** É comumente utilizado em empresas, incluindo a empresa anfitriã.

2.3 Testes

Boas metodologias para testes de desempenho são absolutamente essenciais para este estudo. Como estamos interessados em comparar threads virtuais com sistemas reativos, é preciso ter cuidado para que outras partes da nossa configuração não influenciem indevidamente os resultados. Na primeira subseção ("Sobre Testes de Desempenho"), apresentamos alguns termos e conceitos úteis para os leitores do nosso estudo.

2.3.1 Sobre Testes de Desempenho

Para comparar diferentes abordagens, é necessário um sistema de comparação. No que diz respeito ao desempenho, existe uma infinidade de dados de teste possíveis que medem diferentes aspectos da aplicação. Além disso, a aplicação possui múltiplas áreas de operação e, portanto, a medição com diferentes cargas e tipos de carga é importante. Steven Haines aborda esse tema no livro "Quantifying Performance" (Haines, 2006), no contexto de uma empresa Java. Haines descreve três categorias principais de medição: tempo de resposta, taxa de transferência e utilização de recursos. Essas três categorias compartilham uma relação sob cargas crescentes (medidas por Haines como o número de usuários simultâneos). Quando a carga aumenta, a utilização de recursos aumenta juntamente com a taxa de transferência. Eventualmente, os recursos disponíveis se saturam devido à utilização ineficiente ou à limitação de hardware. Quando os recursos estão saturados, a taxa de transferência estagna e pode diminuir devido ao sistema ter que gastar mais recursos gerenciando a si mesmo (por exemplo, realizando trocas de contexto). Além disso, o tempo de resposta aumenta devido ao aumento da sobrecarga no sistema.

Haines menciona tanto desempenho quanto escalabilidade. Desempenho significa a capacidade de uma aplicação lidar com uma carga elevada, enquanto escalabilidade se refere a como o sistema lida com o aumento dessa carga. Embora desempenho e escalabilidade sejam geralmente vistos como similares, eles não são. O desempenho de um sistema é medido para uma determinada carga, enquanto a escalabilidade mede a capacidade de uma requisição manter o mesmo desempenho durante o aumento da carga. Uma maneira de testar a escalabilidade é testar o sistema aumentando gradualmente a carga e medindo métricas importantes. Quando a carga fica muito alta, o sistema experimenta uma diminuição na taxa de transferência e um aumento no tempo de resposta, o que marca o ponto em que o sistema atingiu seu potencial máximo de carga.

O teste de carga consiste em testar um sistema sob cargas elevadas, como a presença de muitos usuários simultâneos. O objetivo é detectar problemas relacionados à carga (Jiang & Hassan, 2015). Ele também pode ser estendido para identificar problemas de desempenho relacionados à carga, como tempo de resposta e taxa de transferência (Jiang et al., 2009). Recomenda-se que o teste de carga seja realizado por um período mais longo (Jiang et al., 2009) e que não se concentre apenas no resultado médio, mas também inclua fatores como o percentil 95 (Jiang & Hassan, 2015). A justificativa para testes mais longos reside nas pequenas flutuações de memória, uso da CPU etc. A análise de casos mais extremos, como o percentil 95, também fornece informações importantes, pois permite detectar desvios inaceitáveis no desempenho.

2.3.2 Testes em Java

Para responder às questões de pesquisa e atingir o objetivo da tese, foi utilizado um benchmark personalizado. Um benchmark pode ser visto como um teste para avaliar o desempenho de uma ferramenta ou técnica (Sim et al. 2003).

O objetivo dos benchmarks é testar as aplicações em diferentes áreas de operação, como operações intensivas de CPU e de E/S. Para que a comparação dos benchmarks seja justa, certos requisitos precisam ser atendidos em cada teste individual. De acordo com (Bull et al., 1999), algumas propriedades de um benchmark bem-sucedido são:

- **Representativo:** Se o benchmark pretende testar E/S, essas operações devem ser incluído.
- **Interpretável:** O resultado deve fornecer informações sobre por que aquele resultado específico foi alcançado.
- **Robustez:** O próprio teste não deve ser motivo de incerteza ao repetir os testes.
- **Portabilidade:** Ser capaz de recriá-lo.
- **Padronizado:** As métricas de desempenho devem significar a mesma coisa.
- **Transparente:** Deixe claro o que está sendo testado.

Em resumo, os testes de referência devem avaliar uma área específica onde as métricas e medições devem ter o mesmo significado entre as tecnologias. O teste também deve ser interpretável para explicar possíveis diferenças.

Realizar testes de desempenho em ambientes de execução pode ser muito difícil, e o desempenho de aplicações Java pode, por vezes, parecer imprevisível devido a vários fatores, como o coletor de lixo, o JRE e o JIT. O desempenho de uma aplicação Java pode depender da JVM em que está sendo executada ou do tempo de execução (Eeckhout et al., 2003). O ambiente de execução complexo e dinâmico das aplicações Java pode exigir testes mais bem elaborados durante a avaliação de desempenho, em comparação com linguagens compiladas como C e C++, que possuem um ambiente de execução previsível (Blackburn et al., 2006). O desempenho de aplicações Java pode ser difícil de testar com benchmarks devido à alta variabilidade dentro da máquina virtual; portanto, métodos quantitativos são recomendados para garantir um bom teste de benchmark (Gu et al., 2006).

2006). As metodologias predominantes para testar o desempenho de aplicações Java têm sido criticadas por levarem a resultados incorretos e enganosos (Lion et al. 2016). Algumas razões são que as aplicações Java podem apresentar resultados diferentes entre execuções devido a diferentes fontes de não determinismo, como a otimização just-in-time na máquina virtual, a coleta de lixo e o escalonamento de threads. Existem várias maneiras de apresentar os resultados de testes de benchmark em Java, tais como:

- **i)** Calcular a média.
- **ii)** Calcular a mediana.
- **iii)** Incluir apenas a melhor execução.
- **iv)** Incluindo apenas a pior execução.

Algumas maneiras de mitigar os efeitos aparentemente imprevisíveis na JVM incluem a coleta forçada de lixo entre diferentes iterações de teste para evitar que o coletor de lixo seja iniciado em momentos diferentes entre as execuções de teste. Também é possível realizar medições consecutivas ('cccddd' em vez de 'cdcdcd', sendo c e d testes específicos) na mesma instância da JVM. Execuções de aquecimento e o cuidado para não incluir o carregamento inicial de classes também podem fornecer dados melhores (Lion et al. 2016). Adotamos todas essas medidas em nosso estudo.

Estudos anteriores que utilizaram métodos semelhantes ao nosso projeto (teste de carga com Vegeta) não especificaram eventuais falhas do servidor ou uma alta variação entre as iterações de teste (Beroniÿ et al. 2022), (Beroniÿ et al. 2021), (Pufek et al. 2020). Uma participante desses três estudos foi questionada por nós (Beroniÿ 2024) sobre os resultados de seus testes. A pesquisadora consultada afirmou que, de fato, houve travamentos do servidor, mas que estes foram causados pelo 'Fibers', uma versão inicial de threads virtuais. Contudo, ela afirmou que os travamentos do servidor ocorreriam se os testes tivessem sido realizados com uma carga maior. Além disso, os pesquisadores afirmaram que não observaram diferenças significativas entre as iterações de teste, embora tenham notado picos aparentemente aleatórios de latência após análises adicionais, os quais atribuíram ao coletor de lixo. Por fim, ela enfatizou a importância das execuções de aquecimento, nas quais afirmou que seu grupo observou diferenças significativas entre as iterações de aquecimento. Isso nos inspirou a analisar mais detalhadamente a latência e o comportamento do servidor em taxas mais altas.

De modo geral, o manuseio cuidadoso dos dados de teste é especialmente importante ao testar aplicações Java. A natureza imprevisível do ambiente de execução Java pode levar a uma alta variância em medições repetidas, que, obviamente, devem ser analisadas. Por exemplo, esses dados podem ser concatenados utilizando alguma abordagem estatística (Lion et al. 2016).

2.3.3 Analisando o desempenho e os custos das reações-bibliotecas de programação ativa em Java

Ponge et al. comparam o desempenho de diferentes bibliotecas reativas. O estudo (Ponge et al., 2021) pode não estar relacionado a threads virtuais, mas oferece insights sobre como comparar sistemas similares. O estudo compara três bibliotecas Java comumente usadas para programação reativa: SmallRye Mutiny, RxJava e o projeto Reactor.

O estudo dividiu os testes em três domínios: operações individuais, operações com restrições de E/S e pipelines com múltiplos operadores. Esses domínios são semelhantes aos testes que utilizamos em nosso estudo. O objetivo era criar testes que se assemelhassem aos utilizados rotineiramente em programação reativa. Para operações individuais e múltiplas, comparou-se o desempenho na transformação de variáveis para as três bibliotecas reativas. Para as operações com restrições de E/S, foram realizados dois testes. O primeiro teste consistia no processamento de arquivos, onde leitura e escrita são operações bloqueantes. O segundo teste foi baseado em requisições de rede. Em vez de utilizar uma operação de atraso manual, optou-se por criar uma operação bloqueante realista. Nesse teste, o texto de um livro é obtido de um servidor e, posteriormente, operações são realizadas na resposta do servidor.

Um estudo testou servidores HTTP assíncronos versus servidores baseados em threads (executando Tomcat NIO e Tomcat BIO) usando o Apache Bench para simular altas cargas com usuários simultâneos (Fan & Wang 2015). A configuração de teste consistia em uma máquina servidora e uma máquina atacante, ambas com CPU Zeon de seis núcleos de 2,5 GHz e 16 GB de RAM, embora apenas uma CPU estivesse ativa para reduzir a complexidade. O uso de duas máquinas separadas é outro elemento que incluímos em nossos experimentos. O resultado foi que tanto os servidores assíncronos quanto os baseados em threads apresentaram tempos médios de resposta semelhantes com o aumento da carga de trabalho. No entanto, ao analisar os tempos de resposta da cauda (por exemplo, os percentis 95 e 99), o servidor assíncrono apresentou uma diferença significativa.

O tempo de resposta foi menor do que o das abordagens baseadas em threads, com uma diferença crescente em cargas mais altas. Os autores concluíram que o servidor baseado em threads não escalava bem devido ao tamanho limitado da fila. Levamos isso em consideração e decidimos analisar atentamente os parâmetros do Spring e da JVM que tínhamos a possibilidade de ajustar.

2.3.4 Exemplos de implementação de testes

O processo de comparação de diferentes abordagens concorrentes no contexto de operações de E/S apresenta muitas semelhanças com a comparação de diferentes aplicações de servidor. Tanto os testes de servidor quanto as abordagens concorrentes utilizam métodos como testes de carga e estresse. Além disso, compartilham muitas métricas valiosas. Pesquisas anteriores relacionadas à comparação de servidores podem ser úteis para a seleção de métricas e métodos de teste (por exemplo, como criar testes com operações de E/S).

Para o leitor interessado, seguem alguns estudos que nos inspiraram na elaboração de nossos testes:

- **i)** Comparação entre .NET e Java através da medição de latência e memória com diferentes cargas (Hamed & Kafri 2009).
- **ii)** Comparar diferentes arquiteturas de servidores web (por exemplo, threads por cliente) com testes de carga e medição de throughput e tempo de resposta (Pariag et al. 2007).
- **iii)** Comparação de servidores usando taxa de transferência e tempo de resposta para situações com uso intenso de E/S (usando um banco de dados MySQL) e situações com uso intenso de CPU (calculando Fibonacci) (Chitra & Satapathy 2017).
- iv) Comparação de arquiteturas de servidores web multi-core através de testes de carga usando a taxa de transferência como métrica principal, utilizando um servidor com dois discos rígidos e 21600 arquivos para simular E/S (Harji et al. 2012).
- **v)** Utilizando testes de carga (grande quantidade de solicitações HTTP) para comparar o desempenho de serviços baseados em nuvem, usando taxa de transferência, tempo de resposta e utilização da CPU como métricas primárias (Salah et al. 2017).

2.3.5 Análise de Custos Indiretos

As três tecnologias em teste fazem parte do ecossistema Java. Apesar de suas diferentes implementações, cada tecnologia depende do mecanismo de threads fundamental inerente à Máquina Virtual Java (JVM). Notavelmente, a JVM assume um papel importante no gerenciamento e agendamento dessas tecnologias, especialmente no gerenciamento de threads reativas e virtuais, que dependem menos do sistema operacional do que as threads da plataforma. Consequentemente, essa função de gerenciamento introduz uma sobrecarga, o que fica evidente ao examinar o código-fonte dessas tecnologias. Investigamos as bases de código desses métodos e encontramos indícios de que podem gerar extensas pilhas de chamadas, o que significa que analisar as pilhas de chamadas pode ser útil para explicar as diferenças de desempenho.

2.3.6 Métricas de hardware

O hardware geralmente é um fator limitante dos programas, tanto em termos de velocidade de processamento quanto de capacidade de carga. Do ponto de vista do desenvolvedor, não há muito que ele possa fazer para melhorar o hardware físico, mas ele tem controle sobre a criação de aplicativos que utilizem os recursos de hardware da maneira mais eficiente possível.

Em aplicações concorrentes com muitas instâncias de bloqueio, devido, por exemplo, a operações de E/S, existe o risco de os recursos da CPU permanecerem ociosos devido a estruturas de programa não otimizadas, onde certas operações bloqueiam threads do kernel. Isso pode causar condições semelhantes a filas, que têm o potencial de reduzir a taxa de transferência e aumentar a latência. Uma maneira de detectar esses bloqueios ineficientes é investigar a utilização da CPU e monitorar seu nível e estabilidade, por exemplo, oscilando entre baixa e alta utilização periodicamente durante cargas constantes.
Isso é algo que analisamos em nosso relatório.

utilização da CPU

Embora a utilização da CPU como métrica não forneça informações explícitas sobre métricas de eficiência como latência e taxa de transferência, ela ainda é valiosa para avaliar o desempenho geral e a utilização de recursos. A utilização da CPU pode fornecer informações valiosas sobre gargalos, uma vez que a métrica inclui a proporção entre o tempo em que a CPU está ociosa e se o sistema atinge ou não a saturação (utilização total dos recursos). O uso ideal da CPU para o nível de saturação e o desempenho ideal pode variar para diferentes arquiteturas e máquinas, mas cerca de 80% parece ser um bom limite para um servidor (Liu & Ding 2010).
(AWS) (Nuvem 2024). O uso da CPU também é considerado uma métrica econômica valiosa da perspectiva das partes interessadas em um ambiente de nuvem (Liu & Ding 2010). Medimos a utilização da CPU para todos os nossos benchmarks usando o VisualVM.

2.4 Ferramentas

As ferramentas descritas abaixo serão utilizadas ao longo do estudo.

2.4.1 Vegeta

Vegeta é uma ferramenta de teste de carga de código aberto que gera e transmite requisições HTTP e salva dados como latência e taxa de sucesso em um arquivo. O Vegeta é usado através da linha de comando e é fácil de integrar em scripts Python e Bash, por exemplo. A ferramenta permite realizar testes de carga simulando vários usuários simultâneos. Vários parâmetros podem ser ajustados manualmente, como duração, requisições por segundo, número de usuários simultâneos e outros. A ferramenta retorna um relatório contendo parâmetros importantes como latência (média, percentis 50, 90, 95 e 99), throughput, taxa real e respostas do servidor.

O Vegeta é escrito em Go, uma linguagem de alto desempenho com suporte nativo para concorrência. Isso permite testes com alta carga na máquina atacante (ou seja, o cliente enviando requisições para o servidor). Se o Vegeta não conseguir atender às especificações solicitadas, como o número de requisições por segundo, ele informará os usuários. Isso torna a ferramenta confiável.

Vegeta foi utilizado em estudos semelhantes nesta área (Beroniÿ et al. 2022), (Pufek et al. 2020).
Além disso, Vegeta tem sido usado em muitos estudos como ferramenta de teste de carga. Veja (Schuler et al.

2021), (Park et al. 2021), (Nor Sobri et al. 2022), (Choi et al. 2021) e (Zhang et al. 2023).
Outras ferramentas populares para testes de carga HTTP também foram avaliadas, como o Apache JMeter e o Apache Benchmark. No entanto, essas ferramentas não se mostraram tão confiáveis na simulação de um alto nível de concorrência e requisições por segundo, resultando em problemas de desempenho nas ferramentas do lado do cliente antes do servidor real (pelo menos durante nossos testes e avaliações).

2.4.2 VisualVM

O VisualVM foi utilizado para o perfilamento dos servidores em nosso estudo. Essa ferramenta permite o monitoramento em tempo real de um grande número de parâmetros em aplicações Java. Isso inclui parâmetros como uso de CPU, uso de memória, metadados de threads e uso do coletor de lixo. O VisualVM coleta dados sobre a aplicação JVM diretamente da JVM. Essa ferramenta foi escolhida devido à sua facilidade de uso e à sua capacidade de rastrear múltiplos parâmetros de desempenho e isolar o uso de recursos em aplicações Java individuais. Além disso, a ferramenta é considerada robusta por ser gerenciada pela Oracle e por ter sido utilizada em estudos anteriores, como (Beroniÿ et al. 2022).

2.4.3 Wireshark

Utilizamos o Wireshark para analisar as falhas de servidor que ocorreram durante nossos experimentos.
Quando o servidor estava sob alta carga, o remetente era bloqueado e relatava um erro.
Este erro não é relatado corretamente no Vegeta e ferramentas adicionais tiveram que ser usadas para analisar os erros e determinar se eles eram devidos ao servidor ou a fontes externas.

O Wireshark permite ao usuário analisar o tráfego de rede de forma útil. Ele pode, por exemplo, listar todo o tráfego de rede em uma conexão, exibindo URLs, protocolos utilizados, remetente, destinatário e registros de data e hora. Durante uma alta carga no servidor, este pode bloquear conexões adicionais, não conseguir enviar confirmações ou não ser capaz de recebê-las.
Isso pode ser observado no Wireshark quando o servidor envia comandos de reinicialização (fechando a conexão devido à sobrecarga) ou através de uma comunicação confusa relacionada a ACKs duplicados (pacotes podem ter sido perdidos ou recebidos fora de ordem) e retransmissões. Isso ocorre quando a capacidade de processamento do servidor é excedida e alguns pacotes podem ser processados fora de ordem ou descartados.

2.4.4 Gráficos de Chama

O Flame Graph (Gregg, 2016) é usado principalmente como uma ferramenta para visualizar os rastreamentos de pilha gerados por profilers como Perf, Dtrace, Jstack e Xtrace. O Flame Graph foi implementado para obter uma melhor compreensão dos rastreamentos de pilha gerados por profilers. Em aplicações multithread, várias threads podem executar o mesmo procedimento; assim, o Flame Graph agrega essas threads para mostrar os padrões gerais das aplicações relacionados a chamadas de função e frequência relativa.

Os flame graphs são relativamente fáceis de usar e são apresentados em duas dimensões. O eixo y representa a pilha de chamadas, enquanto o eixo x representa a frequência relativa e a porcentagem do tempo de CPU gasto em uma determinada função. A Figura 2.1 ilustra um exemplo de flame graph, e uma interpretação simples pode ser que as funções e(), g(), i() e a() são executadas na CPU após serem chamadas pelas funções ou procedimentos abaixo delas. No exemplo acima, o rastreamento da pilha para 'g' é: a -> b -> c -> d -> f -> g.

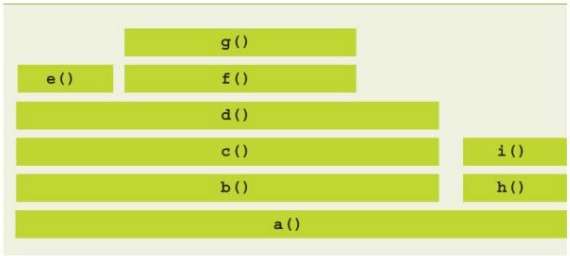


Figura 2.1: Inserir legenda

Em resumo, o gráfico de chamas é uma visualização do rastreamento de pilha agregado de uma aplicação. Ele pode ser usado para investigar o número de chamadas necessárias para alcançar a função que realmente é executada na CPU e também para identificar sobrecarga. Por exemplo, se a função g na figura 2.1 for a função principal, então as funções 'e' e 'i' podem ser funções que adicionam sobrecarga, como o coletor de lixo e o agendador de threads.

Capítulo 3
Método

3.1 Visão geral

Uma visão geral do método pode ser descrita pelas seguintes etapas:

1. **Projeto de benchmarks:** Desenvolva quatro benchmarks para testar E/S, computação e duas combinações. entre computação e entrada/saída.

2. **Criar servidores:** Crie três servidores usando threads de plataforma, threads virtuais e fluxos reativos. Todos os servidores implementarão os quatro benchmarks.

3. **Experimentos de Aumento Gradual de Carga:** Nos experimentos de aumento gradual de carga, uma máquina cliente enviará solicitações simultâneas a um servidor que hospeda uma das três tecnologias, para um dos quatro benchmarks. O nível de concorrência é determinado pelas solicitações por segundo e aumentará linearmente até que ocorra a saturação de recursos no servidor. Durante esses testes, as métricas do cliente e do servidor serão coletadas (como uso de CPU e latência).

4. **Experimentos com Carga Constante:** Semelhantes aos experimentos de aumento gradual de carga, exceto que agora a taxa de requisições é mantida constante em um nível correspondente a cerca de 70% de utilização de recursos (seja qual for o fator limitante para o benchmark em questão). Além disso, utilizamos uma única série de testes em vez de cinco, pois a variância foi muito menor do que para o aumento gradual de carga em nosso ambiente de teste. corre.

5. **Validação e Experimentos Adicionais:** Realize experimentos de validação e adicionais para explicar os resultados e validar o trabalho. Isso inclui a criação de perfis de pilha, a análise da conexão de rede e a avaliação do impacto do software de criação de perfis.

Nas seções seguintes, o método será detalhado em relação ao projeto de benchmark, ambiente de teste e outras considerações.

3.2 Configuração de teste

Nesta seção, apresentamos nossa configuração de teste, incluindo as implementações de benchmark.

3.2.1 Projeto de Referência

Em Java, existem muitos benchmarks comuns, como o conjunto de benchmarks Java Grande (Jils Matthew et al., 1999), que é uma coleção de benchmarks criada para avaliar o desempenho de determinadas implementações de Java. No entanto, se essas implementações de Java não puderem executar o mesmo código-fonte ou um código-fonte muito semelhante, por exemplo, concorrência assíncrona versus threads virtuais, esses testes podem introduzir incertezas. Por exemplo, a implementação do benchmark da tecnologia A e B em Java pelo programador pode levar a otimizações do compilador para uma das tecnologias, mas não para a outra. Isso, por sua vez, pode levar a testes injustos devido a erros na implementação do benchmark, em vez de erros na própria tecnologia. Para evitar possíveis diferenças na implementação de benchmarks entre diferentes tecnologias (por exemplo, reativa versus threads virtuais), uma boa abordagem é simplificar os benchmarks.
A simplicidade dos benchmarks individuais reduz o risco de que as potenciais diferenças entre as tecnologias testadas sejam devidas a erros de implementação. No entanto, vale ressaltar que esses benchmarks simples podem levar a otimizações específicas para cada benchmark.
Foram tomadas precauções para remover esses problemas, como a inclusão de variáveis de loop para evitar otimizações dentro dos loops.

Como esta tese se concentra em sistemas de alta concorrência, optamos por uma abordagem de servidor para implementar o ambiente de testes. O Spring foi utilizado para executar as aplicações do lado do servidor (benchmarks), visto que é uma biblioteca amplamente utilizada e, portanto, aumenta a relevância dos resultados potenciais.

Foram concebidos quatro benchmarks para testar as tecnologias em diferentes áreas; consulte o capítulo anterior para obter diretrizes sobre o projeto dos nossos benchmarks. O primeiro benchmark testa uma operação estritamente limitada por E/S, o segundo testa uma operação estritamente limitada por computação e os outros dois testam combinações de operações computacionalmente intensivas e operações com uso intensivo de E/S. Observe que todas as operações de E/S foram simuladas por meio de diferentes versões de 'thread.sleep' ou 'delay', em vez de usar, por exemplo, um banco de dados. A razão para isso foi eliminar possíveis gargalos, visto que o uso de um banco de dados como E/S foi apontado como um possível gargalo no estudo publicado por Navarro et al. (2023).

Todos os quatro benchmarks possuíam parâmetros diferentes que regulavam o comportamento do benchmark. Para E/S, o parâmetro era a duração do repouso/atraso, e para tarefas computacionais, o número de operações de ponto flutuante. Esses parâmetros foram ajustados manualmente para que o teste de benchmark representasse a área pretendida, por exemplo, CPU ou E/S. Os testes estão disponíveis no Apêndice C.

Teste de E/S

Este método simula uma operação de E/S bloqueante simples com operações computacionais mínimas ou nulas. Ele foi inspirado em Beroniÿ et al. (2022). Aqui, utilizamos um atraso fixo de 100 ms. O atraso pode parecer um pouco alto, mas não é irrazoável para operações bloqueantes como mensagens, busca de itens em servidores remotos ou bancos de dados. Apesar da natureza arbitrária da duração do atraso, ele ainda é eficaz para testar o desempenho dos diferentes servidores no gerenciamento de E/S e seu desempenho quando o número de requisições simultâneas aumenta.

Para threads de plataforma e threads virtuais, o atraso é uma simples operação de suspensão da thread.

Na abordagem reativa, o atraso é definido no próprio objeto de retorno, em vez da thread (consulte o Apêndice C), pois isso emula a forma como os aplicativos assíncronos lidam com operações de bloqueio.

O tempo de espera/atraso fixo foi definido em 100 ms, pois essa é uma latência razoável que pode simular condições realistas (OpenSignal 2020) para um servidor de mensagens.

Teste de Computação

Este método não contém nenhuma operação de entrada/saída além de receber uma solicitação para realizar um cálculo e retornar o resultado. Em vez disso, ele contém um cálculo com complexidade de tempo quadrática que visa carregar as diferentes aplicações. O objetivo disso é investigar possíveis diferenças entre as abordagens em relação a cálculos simples.

Teste Misto - computação

Como mencionado anteriormente, uma das principais preocupações com threads virtuais é que elas podem usar muito espaço no heap, já que seus frames de pilha são armazenados lá quando ocorrem trocas de contexto. Para testar isso (com um caso um tanto extremo), nossos servidores implementam um método de multiplicação de matrizes. A multiplicação de matrizes é um caso de teste muito comum em medições de desempenho e também uma operação comum em aplicações práticas (por exemplo, computação gráfica).
A multiplicação de matrizes é realizada em uma matriz 200x200, seguida por um período de repouso de 25 ms. As dimensões da matriz foram escolhidas experimentalmente para que o teste de aumento gradual da carga tivesse uma duração razoável.

O objetivo deste teste de desempenho era avaliar uma combinação de operações computacionais e de entrada/saída, com foco na computação. Portanto, as dimensões da matriz foram ajustadas para impor altas restrições à CPU, enquanto a duração do período de inatividade foi mantida curta.

Teste Misto - Entrada/Saída

Este método é semelhante ao método anterior, mas as dimensões da matriz foram reduzidas para 150x150 e o atraso aumentado para 100 milissegundos. Isso direciona este teste mais para entrada/saída, embora ainda contenha elementos significativos de computação e requisitos de memória.

O objetivo deste benchmark é semelhante ao anterior, mas com foco adicional em E/S e menor utilização da CPU. As dimensões da matriz foram reduzidas para diminuir as operações de ponto flutuante, enquanto a duração do período de espera/atraso foi aumentada.

3.3 Testes na JVM

Como mencionado nos capítulos anteriores, o teste de aplicações Java pode levar a resultados diferentes entre iterações, mesmo que as variáveis e o ambiente de teste sejam aparentemente os mesmos. Para minimizar esse problema, as seguintes precauções foram tomadas. Consideramos que elas são eficazes e estão dentro do escopo razoável desta tese.

Coleta manual de lixo entre cada iteração de teste. Para permitir que todas as iterações de teste tenham o mesmo ponto de partida (Lion et al. 2016). Antes de cada iteração, a máquina atacante enviará uma solicitação HTTP que força o servidor a realizar uma coleta de lixo.

Aquecer a JVM antes dos testes (Lion et al. 2016), (Lion et al. 2016), (Navarro et al. 2023), (Pufek et al. 2020), (Beroniÿ et al. 2021), (Beroniÿ et al. 2022). Isso permite que o aplicativo carregue as classes necessárias e que o JIT realize otimizações de aquecimento. Isso permite uma

resultados mais consistentes. O aquecimento é realizado executando um teste de carga prolongado com o mesmo método de teste antes de cada iteração. O aquecimento também visa forçar a compilação JIT. A máquina que executa o servidor possui um contador de limite para compilação JIT igual a 10.000 (acessado pelo comando `java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version`), o que significa que um método pode ser compilado após ser chamado esse número de vezes. Assim, as especificidades do aquecimento (número de requisições) serão projetadas para chamar os métodos mais de 10.000 vezes com margem de segurança. Outra opção potencial seria desabilitar o compilador JIT. No entanto, o compilador JIT é parte integrante do Java e não é recomendável desabilitá-lo. Além disso, o compilador JIT provavelmente estará sempre em execução durante condições semelhantes às deste teste experimental (servidores), tornando o estudo mais realista com o compilador JIT habilitado.

Medições consecutivas também serão realizadas, onde uma JVM executando uma determinada abordagem será utilizada. permanecerá ativo entre as medições (Lion et al. 2016).

Além disso, os testes serão realizados várias vezes. Como visto em pesquisas anteriores, existem diversas maneiras de processar os dados. Uma delas é calcular a média de todas as execuções (Pufek et al. 2020), (Beroniÿ et al. 2022), mas isso pode levar a uma alta variância, resultando em conclusões enganosas. Outra abordagem é considerar o melhor resultado de cada iteração (Navarro et al. 2023). A ideia por trás disso é comparar o melhor resultado para as três tecnologias em um determinado método, resultando em uma comparação justa. A terceira abordagem é adotar uma abordagem estatística (Lion et al. 2016) usando intervalos de confiança. Todas essas opções são viáveis e oferecem diferentes vantagens e desvantagens. Calcular a média pode reduzir parte da variância entre as execuções dos testes. No entanto, se uma tecnologia for muito volátil, por exemplo, se apresentar bom desempenho em algumas execuções e baixo desempenho em outras, os resultados podem ser enganosos. Selecionar a melhor versão permite uma comparação justa entre as tecnologias, considerando que todas são comparadas durante seu desempenho máximo.

Todas essas abordagens foram consideradas no planejamento do experimento. No entanto, optou-se por calcular a média dos testes, similarmente a estudos anteriores (Pufek et al. 2020), (Beroniÿ et al. 2022). Este estudo considerou a média de cinco testes, em vez dos três utilizados em estudos anteriores. É claro que calcular a média de um número maior de testes poderia levar a um resultado mais preciso. Devido às restrições de tempo, cinco testes foram a opção mais razoável e ainda representam um número maior do que o utilizado em estudos anteriores (Pufek et al. 2020), (Beroniÿ et al. 2022), (Navarro et al. 2023).

Para reduzir o viés, as tecnologias serão testadas com diferentes tipos de benchmarks. Por exemplo, não é recomendável o uso de threads virtuais para tarefas que exigem muita memória (Oracle Corporation, 2023). Portanto, devem existir testes que também as avaliem em outras áreas, como tarefas de computação, tarefas de E/S e, naturalmente, tarefas que consomem mais memória.

Em resumo, pesquisas e críticas anteriores parecem indicar que os testes de desempenho de aplicações Java podem levar a resultados imprevisíveis em algumas execuções. Para reduzir essa volatilidade, foram tomadas medidas inspiradas em estudos anteriores, conforme descrito nas seções de Introdução e Contexto. Além disso, outras medidas foram implementadas, como o monitoramento da rede entre as máquinas, a manutenção de um ambiente de teste consistente e estável e a padronização dos testes. Por exemplo, se a tecnologia A for testada por meio de um ataque (um lote de requisições HTTP) a cada dois minutos, a tecnologia B também será testada dessa forma.

Para garantir a consistência entre os testes, eles serão automatizados para assegurar tempos iguais (por exemplo, o tempo entre lotes de requisições para o teste de aumento gradual de carga), coleta de lixo e incremento de requisições.

3.3.1 Configuração de hardware

Nossos experimentos foram realizados em duas máquinas idênticas (MacBook Pro 2019) com as seguintes especificações: processador Intel Core i7 de 6 núcleos a 2,6 GHz, memória de 16 GB DDR4 de 2667 MHz e macOS 14.2.1.
Uma máquina hospedava um servidor e coletava dados de perfil, enquanto a outra realizava testes de carga por meio do Vegeta. As máquinas estavam conectadas por um cabo Ethernet de um gigabit. O OpenJDK versão 21.0.2+13-58 foi utilizado tanto para o ambiente de execução quanto para a JVM (máquina virtual de servidor de 64 bits). O Spring Boot versão 3.3.2 também foi utilizado.

3.3.2 Métricas coletadas

Tanto a máquina cliente quanto a máquina servidora coletam métricas durante os ataques. A máquina cliente coleta as seguintes métricas: latência (mínima, máxima, média, percentis 50, 90, 95 e 99), taxa de transferência, taxa de amostragem, respostas do servidor (para taxa de sucesso), bytes de entrada e bytes de saída. Isso é feito através do Vegeta.

O servidor coleta métricas referentes à utilização da CPU, número de threads (ativas e iniciadas) e heap (tamanho total e tamanho utilizado). A justificativa para a inclusão dessas métricas foi explicada nas seções anteriores. Também coletamos dados sobre inflações e deflações de pontos de sincronização, bem como pontos de segurança. Pontos de sincronização e pontos de segurança estão relacionados à concorrência e serão explicados brevemente quando ocorrerem no relatório. O estudo desses dados pode ser útil para a interpretação dos nossos resultados.
Em seguida, todos os dados são concatenados em arquivos e processados.

3.3.3 Tratamento de erros

Tanto a máquina atacante quanto o servidor foram monitorados em busca de erros (impressos no terminal ou nos arquivos de saída), que ocorreram apenas sob cargas elevadas ou em momentos em que os recursos (CPU, heap ou threads) estavam muito sobrecarregados. Os erros não foram incluídos na comparação, mas ainda podem ser importantes para o estudo, especialmente ao monitorar tecnologias em zonas críticas com cargas elevadas.

Do lado do receptor, os erros ocorreram principalmente devido à falta de resposta do servidor às requisições. Esses erros foram exibidos por meio de um token 'EOF' no Vegeta ou por uma mensagem como 'Conexão TCP reiniciada' no Wireshark. Para confirmar que os erros se originavam no servidor, o Wireshark foi utilizado na máquina do servidor para monitorar seu tráfego durante períodos de alta carga. Observe que o Wireshark não estava presente durante os testes em si, mas sim incorporado após a sua conclusão. Isso confirmou que as falhas do servidor eram devidas à sobrecarga e eram exibidas por meio de uma mensagem clara 'reset' ou por um diálogo caótico entre o receptor e o remetente referente a retransmissões e ACK-DUP. Esses cenários geralmente ocorrem devido à sobrecarga do servidor e não estavam presentes durante os testes sem erros. Obviamente, isso não prova com certeza que não possa haver problemas de rede, embora durante os testes a conexão tenha sido monitorada juntamente com outras variáveis relacionadas à sobrecarga do servidor, como alta utilização de recursos, alta latência e baixa taxa de transferência.

3.4 Parâmetros

Tivemos que escolher vários parâmetros para nossos experimentos, explicados brevemente a seguir.

- Um teste foi interrompido após três requisições consecutivas com falha. Após esse evento, as latências foram de vários minutos, o que está fora do escopo deste estudo, pois estamos interessados em casos de uso práticos.

- Realizamos cada série de testes cinco vezes e calculamos as médias. À medida que cada série de testes era repetida, obtínhamos uma média de cinco repetições. Levou várias horas, fazer mais não era viável.

- O comprimento e as taxas de requisição usados no aquecimento foram calculados para chamar o servidor mais de dez mil vezes com alguma margem de erro, já que até esse número a JVM poderia executar compilações JIT (como explicado no capítulo de contexto).

- Atrasos de 60 segundos antes da coleta de lixo e 20 segundos depois. Essa configuração foi definida empiricamente por meio do estudo de séries temporais do VisualVM, escolhendo valores que permitissem que a utilização da CPU e o uso de memória retornassem aos valores nominais entre os testes. iterações.

3.5 Experimento de Rampa de Carga

Os quatro benchmarks diferentes descritos acima (computação, E/S, E/S mista e computação mista) foram executados em nossos três servidores distintos. Eles foram automatizados por meio de um script e consistiram em ataques Vegeta escalonados com um atraso e uma coleta de lixo forçada entre cada execução. Essa série de testes foi realizada cinco vezes para cada configuração de servidor-método, e todas as séries de testes foram precedidas por ataques de aquecimento.

Os testes são realizados por um script automatizado para garantir tempos iguais (tempo de aquecimento, tempo para coleta de lixo etc.) entre as diferentes abordagens e métodos. Este script inclui um ataque de aquecimento entre cada iteração do teste. Após cada teste individual, o cliente envia uma solicitação ao servidor para realizar a coleta de lixo manualmente e, em seguida, a máquina cliente tem um atraso constante antes de iniciar o próximo ataque HTTP.

A variável que muda entre os ataques individuais é o número de requisições por segundo (nível de concorrência), com incrementos de 50 requisições por segundo (50 foi escolhido para que os experimentos pudessem ser conduzidos em um tempo razoável, mantendo ainda um certo grau de granularidade). Isso continuará até que o servidor falhe três vezes consecutivas ou que a taxa de transferência se torne muito baixa em comparação com a taxa de ataque, levando a uma falha na máquina atacante devido ao tempo limite de espera pelos resultados (70 segundos). Apenas os testes com 100% de sucesso serão considerados neste experimento; no entanto, todos os dados dos testes serão incluídos e discutidos.

Durante este experimento, diversas métricas foram coletadas. A máquina que enviava as requisições HTTP coletava dados como latência (mínima, máxima, média, percentis 50/90/95/99), taxa de transferência, taxa de requisições por segundo (taxa real e taxa pretendida), total de requisições enviadas, total de requisições com falha e bem-sucedidas, e eventuais códigos de erro. A máquina servidora coletava dados como uso da CPU, uso do heap, número de threads, pontos seguros, pontos de inflação, tentativas de bloqueio em disputa e pontos de deflação.

Os testes foram realizados cinco vezes cada, o que significa que um total de sessenta testes foram realizados durante este experimento (quatro métodos de teste, três tecnologias testadas, cinco execuções de teste).

Cada iteração de teste (por exemplo, teste de E/S de threads virtuais) foi conduzida conforme as listas abaixo.

Aquecimento:

1. 60 segundos com 300 solicitações por segundo

2. Coleta manual de lixo

3. Operação em modo de espera por 20 segundos

4. Repita três vezes com taxas ajustadas para o método específico.

As três taxas de aquecimento posteriores são ajustadas para 70% da taxa máxima do método (conforme explicado no capítulo de contexto, esta é uma boa faixa de operação para um servidor de mensagens).
Após o aquecimento, os testes propriamente ditos são realizados de acordo com a lista abaixo.

Ataques:

1. Envie solicitações HTTP com a taxa 'RATE' durante 10 segundos. Salve os resultados.

2. Operação em modo de repouso por 60 segundos

3. Coleta manual de lixo

4. Durma por 20 segundos

5. Aumente a 'TAXA' em 25/50 e repita.

O procedimento acima é repetido N vezes até que ocorram três falhas consecutivas, por exemplo, a falha do servidor. Consulte o Apêndice B para obter o script automatizado.

3.6 Teste de Carga Constante e Pro- Adicional petição

3.6.1 Teste de Carga Constante

Para verificar os resultados dos testes primários, foram realizadas execuções de testes mais longas para investigar se a diferença entre as tecnologias ainda estava presente e se essa diferença se mantinha ao longo do tempo. De acordo com nosso supervisor na Sinch, os servidores geralmente são escalados quando cerca de 80% dos recursos de hardware são utilizados. Assim, esses testes mais específicos foram conduzidos com uma taxa de requisição que corresponde a cerca de 70% de carga nas CPUs (testes de computação) ou na memória (testes de E/S). Como 80% de utilização de recursos significa escalar o servidor e considerando que há alguma variação no uso de recursos, um modo de operação razoável seria um pouco menor que 80% (por exemplo, 70%). As execuções de longa duração foram realizadas com apenas uma taxa específica, selecionada para simular condições de trabalho realistas. As taxas para os benchmarks foram diferentes, mas as taxas foram as mesmas para todas as tecnologias; por exemplo, o método de computação para threads virtuais e fluxos reativos teve a mesma taxa de requisição.

Cada teste foi realizado de acordo com o seguinte procedimento:

1) 3 séries de aquecimento de 60 segundos com 300 solicitações por segundo cada.

2) Corrida de aquecimento de 2 minutos com a cadência pretendida.

3) Coleta manual de lixo.

4) Aguarde 60 segundos.

5) Iniciar teste. 10 minutos.

Para esses testes, foram utilizadas as ferramentas Vegeta e VisualVM para coletar amostras dos resultados.

3.6.2 Perfilamento de Pilha

Para investigar as pilhas de chamadas e obter uma visão geral da sobrecarga nessas três tecnologias, foram utilizados FlameGraphs. Esses gráficos são uma abstração da saída de profilers, como perf e Dtrace. Utilizamos o Dtrace. Os gráficos mostram visualmente a extensão da pilha de chamadas e também aproximam o tempo de CPU gasto em cada método, apresentado como uma fração do tempo de amostragem.

O procedimento de teste seguiu os mesmos protocolos apresentados na seção anterior, incluindo extensos aquecimentos. O teste em si, no entanto, foi mais curto (com 240 segundos), pois não observamos nenhuma diferença nos resultados em comparação com testes mais longos em nossos ensaios. Cada iteração do teste foi realizada de acordo com o seguinte procedimento:

1) Aquecimento (3 ataques de Vegeta de 60 segundos com as mesmas taxas que seriam usadas no teste real. Incluindo coleta manual de lixo)

2) Coleta manual de lixo

3) Atraso manual de 60 segundos

4) Início do teste. 240 segundos com taxa constante.

Este procedimento foi realizado para todas as três tecnologias com todos os quatro métodos. As taxas foram diferentes entre os métodos, mas foram mantidas iguais para todas as três tecnologias.
As taxas foram ajustadas manualmente com o objetivo de não sobrecarregar os servidores durante os testes, mantendo a utilização do hardware dentro de condições de trabalho realistas (ou seja, 70% de utilização do fator limitante). Isso se mostrou difícil devido a alguns métodos apresentarem falhas antes de outros.
As taxas finais foram:

- 1200 solicitações por segundo para computação
- 1600 solicitações por segundo para E/S
- 300 solicitações por segundo para computação de multiplicação de matrizes
- 600 solicitações por segundo para o modo de espera matmul

Após a conclusão das medições, o resultado do perfilamento foi processado usando os scripts Flamegraph de Brendan Gregg para consolidar as pilhas e convertê-las em FlameGraphs. A implementação atual do script de processamento Jstack (Oracle Help Center 2024) não foi atualizada para lidar com threads virtuais, mas alguns ajustes de nossa parte resolveram esse problema.

3.7 Validação

Seria inadequado realizar esse tipo de teste sem a devida validação para garantir que as restrições e configurações do nosso ambiente de teste não influenciassem indevidamente os resultados. Abaixo, apresentamos as principais validações e testes da nossa configuração.

3.7.1 Conexão de Rede

Quando os testes de carga foram realizados neste estudo, eles foram executados até que o servidor falhasse ou a conexão de rede fosse interrompida. Naturalmente, surgiram dúvidas sobre se era realmente a sobrecarga do servidor que levava ao desligamento da rede. Para testar isso, dois experimentos foram realizados. Primeiro, um endpoint vazio foi configurado em cada um dos servidores (vazio no sentido de que simplesmente retornava um reconhecimento e não forçava o computador do servidor a executar nenhuma outra ação). Ao chamar esse método, deveríamos ser capazes de atingir taxas muito altas se a conexão de rede não impusesse um limite baixo de capacidade. Em segundo lugar, uma seleção de nossos testes usuais foi realizada novamente e monitorada no Wireshark (descrito no capítulo anterior). Os resultados das medições são apresentados no próximo capítulo.

3.7.2 Impacto do Software de Criação de Perfil

É uma lei fundamental da natureza que, ao observar algo, você o afeta (Heisenberg, 1927). Isso é especialmente verdadeiro para softwares de perfilamento, pois algum tipo de sonda precisa ser incorporado a eles e os dados precisam ser registrados pelo mesmo hardware que está sendo testado. Para garantir que nosso software de perfilamento tivesse um impacto insignificante em nossos experimentos, executamos nossos testes uma vez sem o perfilamento. Por meio de medições no lado do cliente, verificou-se que qualquer desvio no desempenho estava bem dentro das margens de erro normais, ou seja, o desempenho medido a partir do cliente (latência e taxa de transferência) não foi afetado de forma perceptível pela ausência do software de perfilamento.

Capítulo 4

Resultados

Este estudo consiste em dois experimentos principais, um de carga constante e outro de carga crescente, conforme descrito no capítulo 2. Os resultados desses experimentos serão apresentados aqui.
Ao medirmos a latência, apresentamos diferentes percentis, conforme descrito na seção teórica.
Segue abaixo um breve resumo do que é um percentil.

O percentil *x* da função de distribuição de probabilidade P, doravante denotado por *Px* (ou por percentil x), indica o percentil x. Notavelmente, o percentil 50, P50 , é definido pela função de distribuição de probabilidade como o valor mediano, que é o mesmo valor para o qual a média *(A)* de N amostras converge se *N* crescer (no caso de P simétrica). O percentil 50 pode ser definido da seguinte forma.

$$50\% = P\,(X \ddot{y} a) = \int_{\ddot{y}\ddot{y}}^{um} f\,(t)dt$$

De forma mais geral, o percentil *Px* ocorre quando a função de distribuição cumulativa atinge x/100. Portanto, os percentis são úteis para descrever os piores cenários.

4.1 Experimento de Carga Gradual

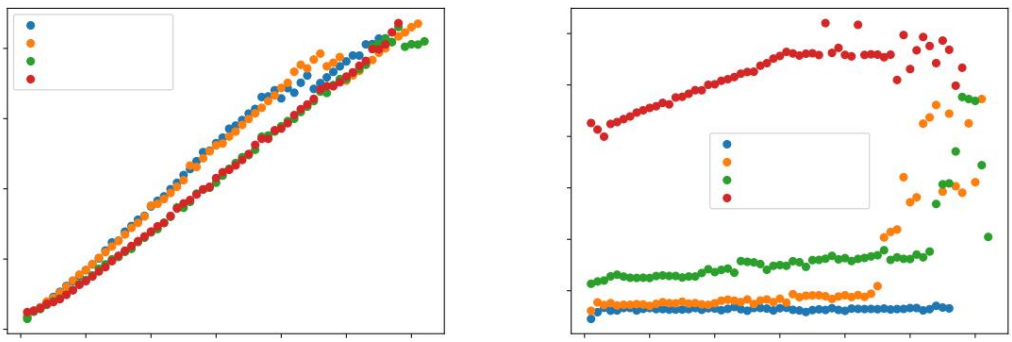
Neste experimento, threads virtuais, threads de plataforma e um sistema reativo foram testados em nossos métodos de benchmark personalizados. Os dados desses testes serão apresentados a seguir.

Conforme mencionado na seção teórica, os resultados entre testes idênticos podem variar bastante devido ao funcionamento interno da JVM. Diversas precauções foram tomadas para mitigar esse problema, e elas são explicadas em detalhes no capítulo 2. Cada teste individual resultou em entre vinte e oitenta séries de amostras, dependendo do teste (as amostras são concatenadas pelo VisualVM).

A taxa descrita nas seções seguintes refere-se a solicitações por segundo. Nesta seção, também serão exibidos diferentes percentis de latência. O percentil 99 para latência representa a latência (tempo) que 99% das solicitações superam. Isso significa que ele descreve, em certa medida, os piores cenários. Devido à instabilidade dos testes, nos gráficos da seção seguinte, cada amostra representa a média de cinco tentativas bem-sucedidas.

4.1.1 Uso Intenso da CPU

Este método consiste em cálculos vagamente inspirados no cálculo de uma sequência de Fibonacci com alguns elementos de aleatoriedade introduzidos (ver Apêndice). Não contém atrasos.
ou chamadas de servidor, e, portanto, não é o caso de uso pretendido para threads virtuais ou reativas.
fluxos. No entanto, é uma medida interessante de desempenho e, portanto, está incluída neste.
relatório.



(a) Utilização média da CPU em porcentagem para threads reativas, de plataforma e virtuais.

(b) Utilização média de heap em bytes para reativo, plataforma e virtual fios.

Figura 4.1: Utilização da CPU e do Heap

As Figuras 4.1a e 4.1b ilustram a utilização de hardware atribuível às três tecnologias durante uma operação intensiva de CPU. A utilização da CPU para os métodos difere significativamente. A abordagem reativa apresenta a maior utilização de CPU no geral, seguida de perto por
threads virtuais. Os threads de plataforma de bloqueio têm a menor utilização de CPU. A utilização de memória para as três abordagens de concorrência também é diferente (como pode ser visto em
figura 4.1b). Em taxas mais baixas, os threads da plataforma têm a maior utilização de memória, enquanto
Os modelos virtuais e reativos têm menor utilização de memória. No entanto, à medida que a taxa de requisições aumenta, esse consumo de memória também aumenta.
Os threads virtuais começam a consumir mais memória.

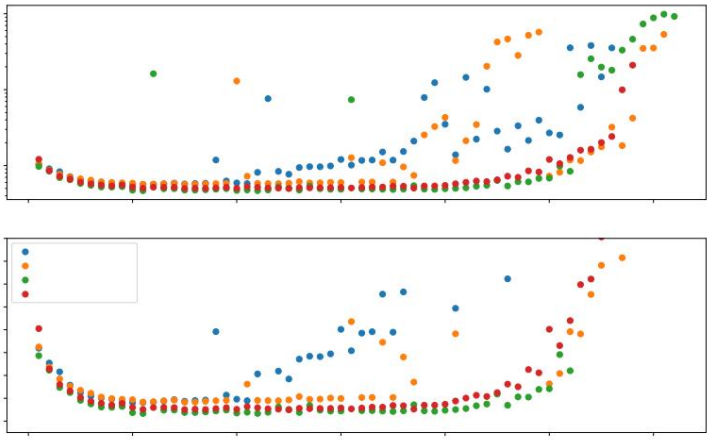


Figura 4.2: Média da latência do 99º percentil em milissegundos para threads reativas, de plataforma e virtuais.

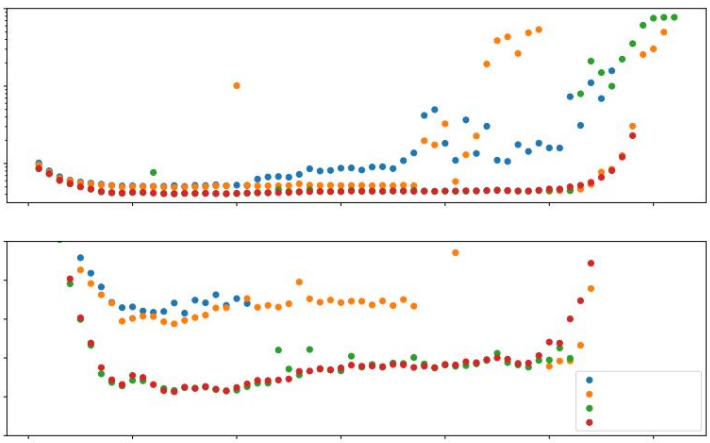


Figura 4.3: Latência média do 90º percentil em milissegundos para threads reativas, de plataforma e virtuais.

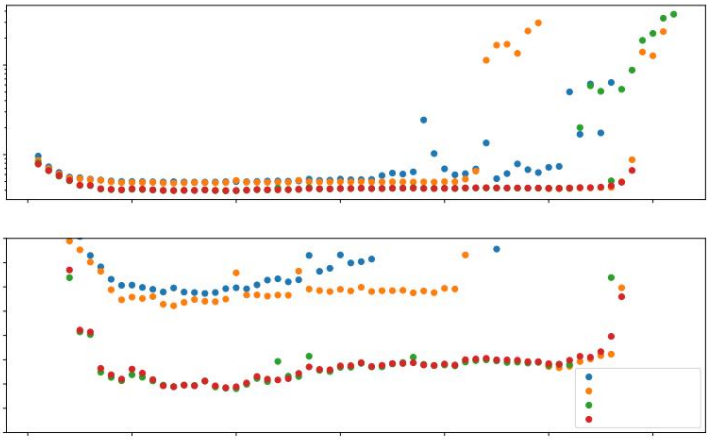


Figura 4.4: Latência média do 50º percentil em milissegundos para threads reativas, de plataforma e virtuais.

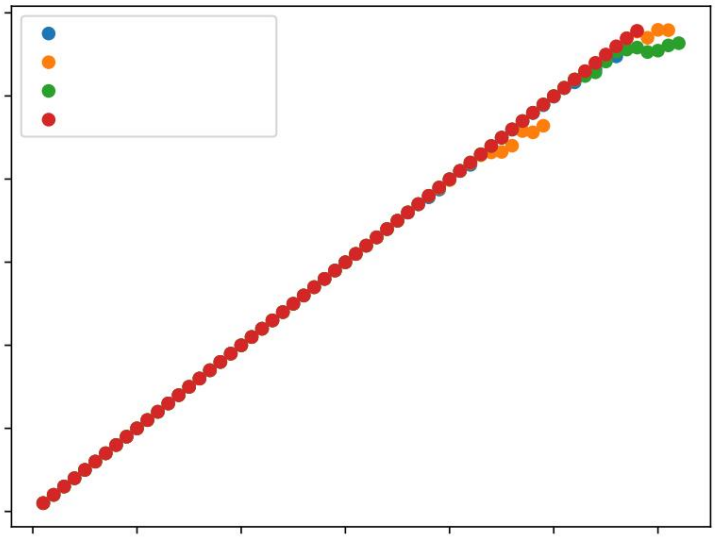
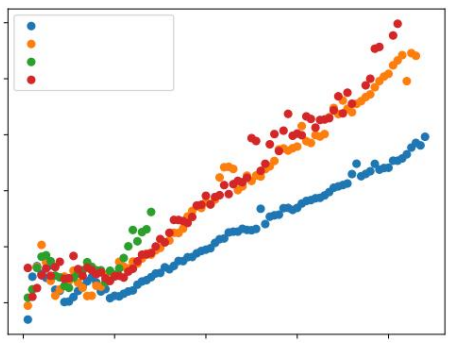


Figura 4.5: Taxa de transferência média (mensagens por segundo) para threads reativas, de plataforma e virtuais.

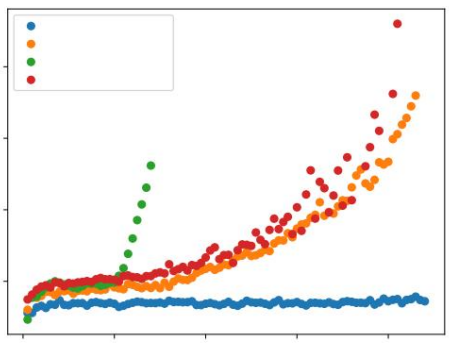
As figuras 4.2, 4.3, 4.4 e 4.5 ilustram as métricas de desempenho para as diferentes tecnologias. Os threads de plataforma apresentam o melhor desempenho em termos de latência em todos os percentis, com uma latência quase 25% menor do que os threads virtuais e os fluxos reativos no 99º percentil. As threads virtuais apresentam uma latência ligeiramente menor do que a abordagem reativa para taxas inferiores a 2200 requisições por segundo. No entanto, para taxas de requisição superiores a esse valor, a latência das threads virtuais aumenta e, consequentemente, torna-se muito instável, resultando em um desempenho irregular.

4.1.2 E/S intensa

Os resultados para as três tecnologias em operação com uso intensivo de E/S serão apresentados aqui. Primeiramente, serão apresentados gráficos comparativos das diferentes abordagens. Em seguida, serão ilustrados gráficos individuais para cada uma das três abordagens. O objetivo desses gráficos individuais é visualizar a relação entre a utilização de recursos de hardware e as métricas de desempenho, o que pode fornecer informações sobre o desempenho das tecnologias.



(a) Utilização média da CPU em porcentagem para threads reativas, de plataforma e virtuais.



(b) Utilização média de heap em bytes para threads reativas, de plataforma e virtuais.

Figura 4.6: Método de E/S, uso de CPU e Heap

As Figuras 4.6a e 4.6b ilustram a utilização de hardware das três tecnologias durante uma operação de E/S. Todas as três tecnologias apresentam uma utilização de CPU relativamente baixa (como pode ser visto na Figura 4.6a), mas na comparação entre threads virtuais e a abordagem reativa, as threads reativas apresentam claramente uma utilização de CPU menor. Observe que a utilização de CPU para esse método é baixa, o que significa que diversos fatores podem afetá-la significativamente, como a coleta de lixo. Tanto as threads virtuais quanto as threads de plataforma utilizam muita memória heap em comparação com a abordagem reativa (como pode ser visto na Figura 4.6b). Quando os servidores estão sobrecarregados, o uso da memória heap aumenta drasticamente para threads de plataforma e virtuais.

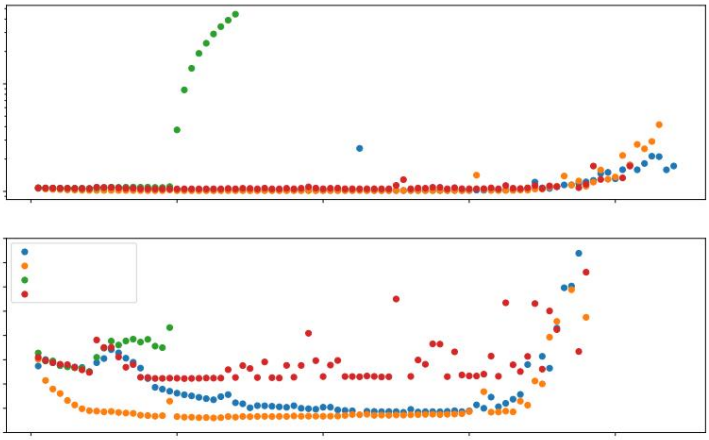


Figura 4.7: Média da latência do 99º percentil em milissegundos para threads reativas, de plataforma e virtuais.

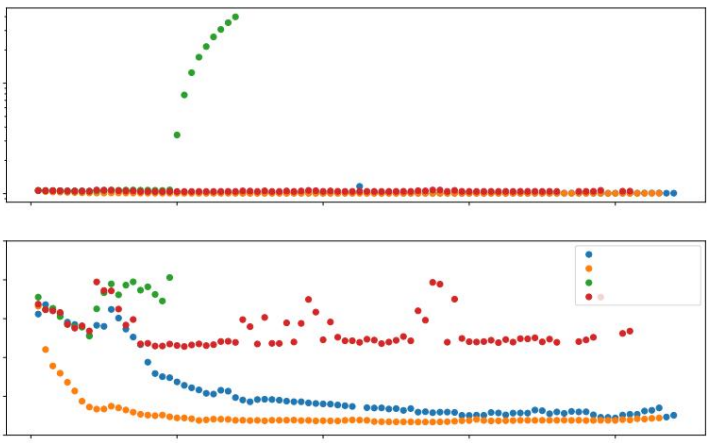


Figura 4.8: Média da latência do 90º percentil em milissegundos para threads reativas, de plataforma e virtuais. Cada amostra representa a média de 5 tentativas bem-sucedidas.

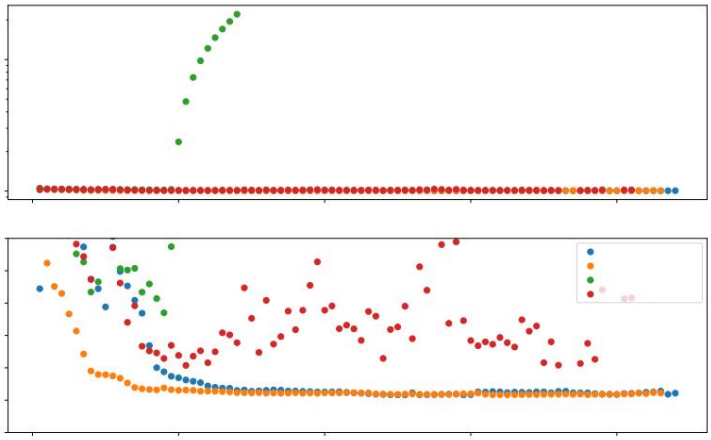


Figura 4.9: Latência média do 50º percentil em milissegundos para threads reativas, de plataforma e virtuais.

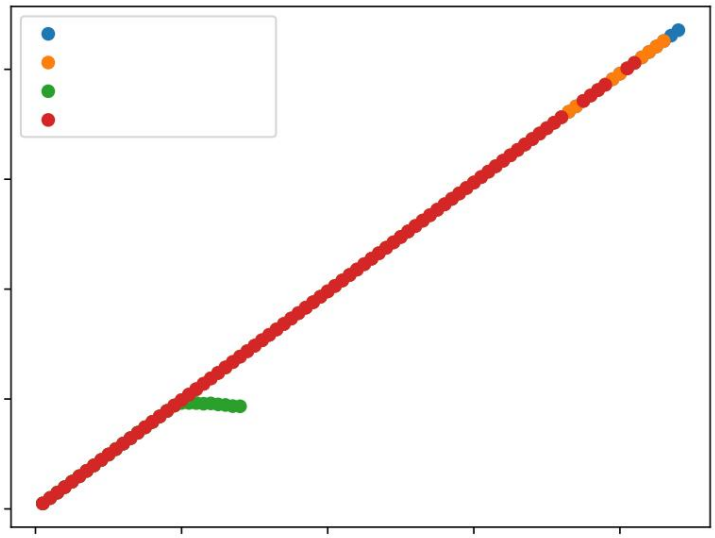
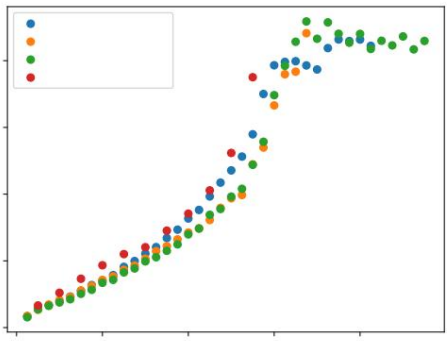


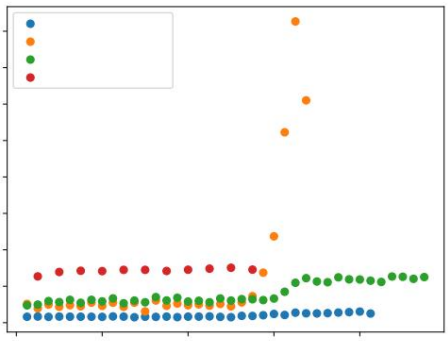
Figura 4.10: Taxa de transferência média (mensagens por segundo) para threads reativas, de plataforma e virtuais.

As figuras 4.7, 4.8, 4.9 e 4.10 ilustram as métricas de desempenho para as três tecnologias. Threads virtuais e fluxos reativos são relativamente equivalentes em termos de latência, mas as threads virtuais apresentam uma latência ligeiramente melhor no 90º percentil. Threads de plataforma têm um desempenho notavelmente pior em relação à latência, comparado a threads reativas e virtuais, tanto na mediana quanto no 90º percentil. Threads reativas conseguem lidar com cargas de concorrência mais altas, apresentando desempenho superior ao das threads virtuais no 99º percentil para cargas acima de 4000 requisições por segundo.

4.1.3 Matmul



(a) Utilização média da CPU em porcentagem para threads reativas, de plataforma e virtuais.



(b) Utilização média de heap em bytes para threads reativas, de plataforma e virtuais.

Figura 4.11: Uso de CPU e Heap na multiplicação de matrizes

As Figuras 4.11a e 4.11b ilustram a utilização de hardware das três tecnologias durante uma operação mista. A utilização da CPU para os métodos é relativamente similar (como pode ser visto na Figura 4.11a). A abordagem reativa apresenta a maior utilização de CPU no geral, seguida de perto por threads de plataforma (ilimitadas) e threads virtuais. A utilização de memória para as três tecnologias também difere (como pode ser visto na Figura 4.11b). Em taxas mais baixas, as três abordagens apresentam utilização de memória similar, mas à medida que a taxa aumenta, o uso de memória tanto para threads virtuais quanto para threads de plataforma aumenta exponencialmente.

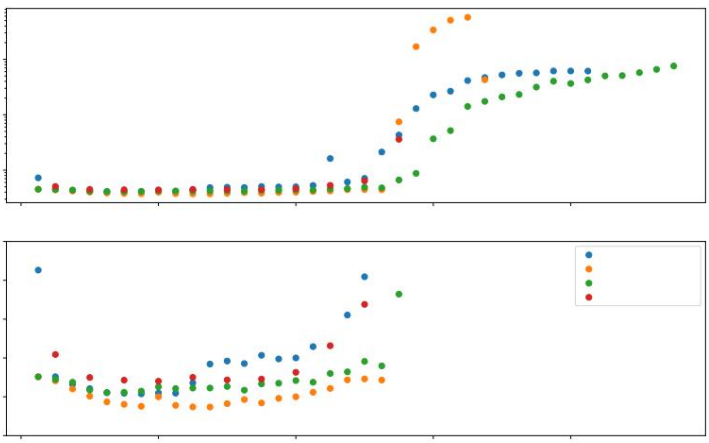


Figura 4.12: Média da latência do 99º percentil em milissegundos para threads reativas, de plataforma e virtuais.

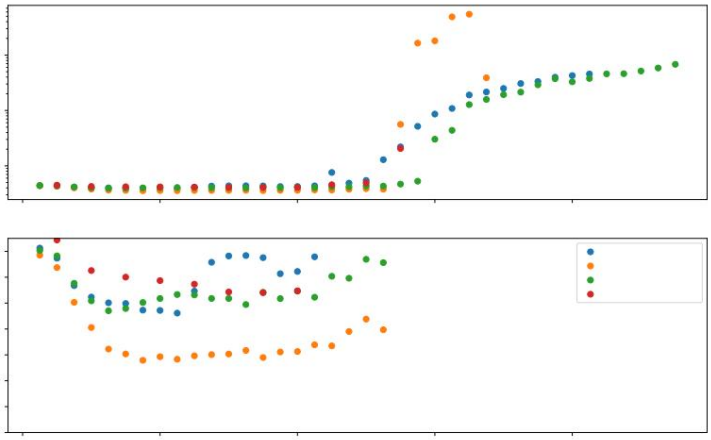


Figura 4.13: Média da latência do 90º percentil em milissegundos para threads reativas, de plataforma e virtuais.

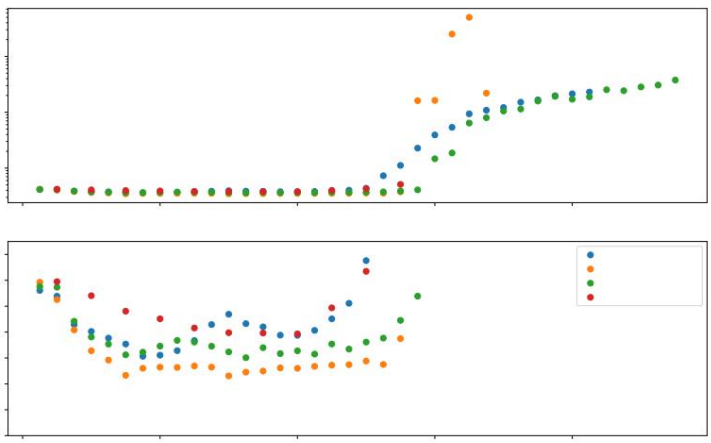


Figura 4.14: Latência média do 50º percentil em milissegundos para threads reativas, de plataforma e virtuais.

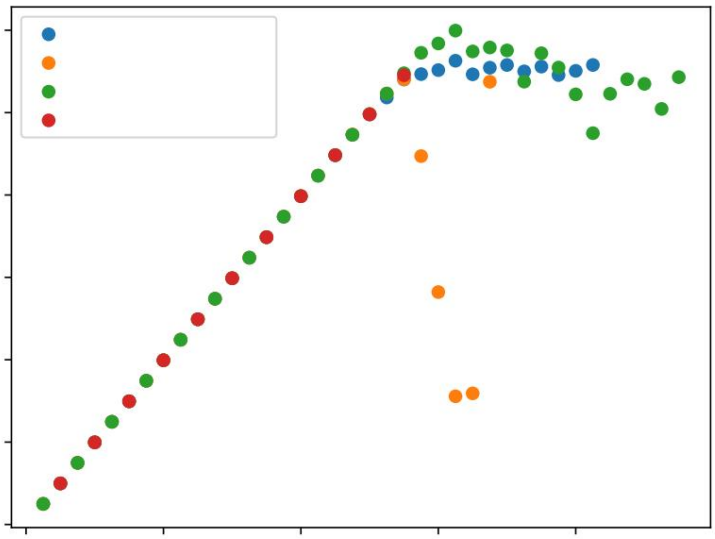
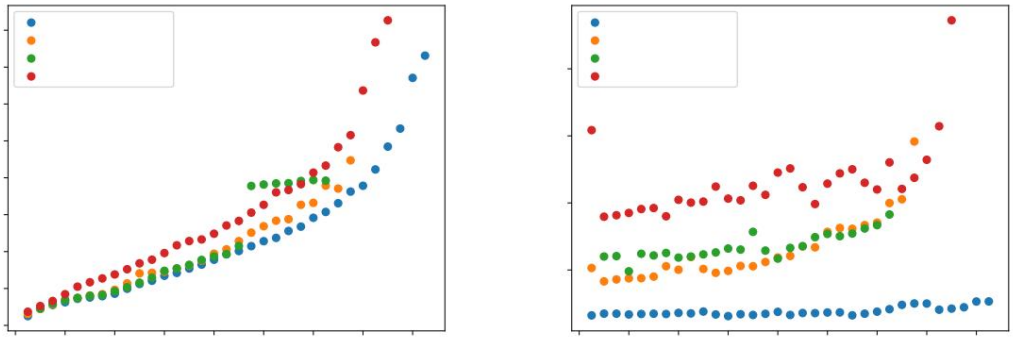


Figura 4.15: Taxa de transferência média (mensagens por segundo) para threads reativas, de plataforma e virtuais.

As figuras 4.12, 4.13, 4.14 e 4.15 ilustram as métricas de desempenho para as três tecnologias. Para taxas mais baixas, correspondentes a uma utilização de CPU inferior a 60%, as threads virtuais apresentam o melhor desempenho em relação às latências nos percentis 99, 90 e 50, com uma taxa de transferência semelhante à das outras duas tecnologias. No entanto, quando a carga aumenta para cerca de 550, a latência das threads virtuais aumenta drasticamente, juntamente com uma redução na taxa de transferência e um aumento na utilização de memória. Isso resulta na falha do servidor. As threads de plataforma e os fluxos reativos também apresentam aumentos na latência e reduções na taxa de transferência, porém não o suficiente para causar uma falha no servidor. As threads de plataforma conseguiram atingir a taxa mais alta.

4.1.4 Método misto

O "método misto" consiste na multiplicação de matrizes com um atraso de 100 ms, o que implica em cargas significativas em termos de computação, operações de atraso e uso de memória.



(a) Utilização média da CPU em porcentagem para threads reativas, de plataforma e virtuais.

(b) Utilização média de heap em bytes para reativo, plataforma e virtual fios.

Figura 4.16: Uso de CPU e Heap pelo método misto

As Figuras 4.16a e 4.16b ilustram a utilização de recursos para as três tecnologias. Os threads da plataforma apresentam a maior utilização de CPU e heap, enquanto os threads reativos...
A tecnologia tem o nível mais baixo.

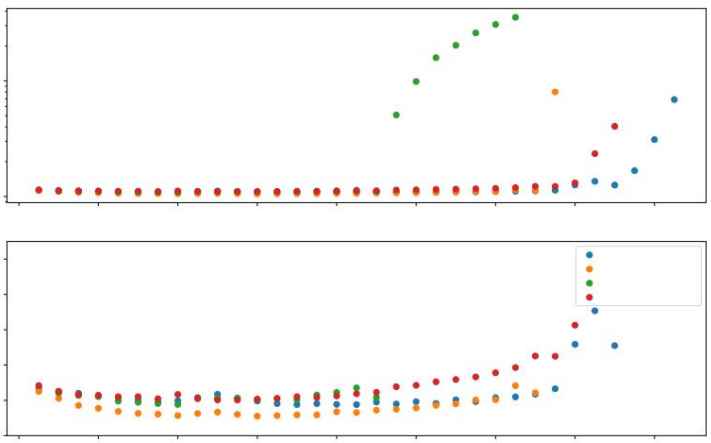


Figura 4.17: Média da latência do 99º percentil em milissegundos para threads reativas, de plataforma e virtuais.

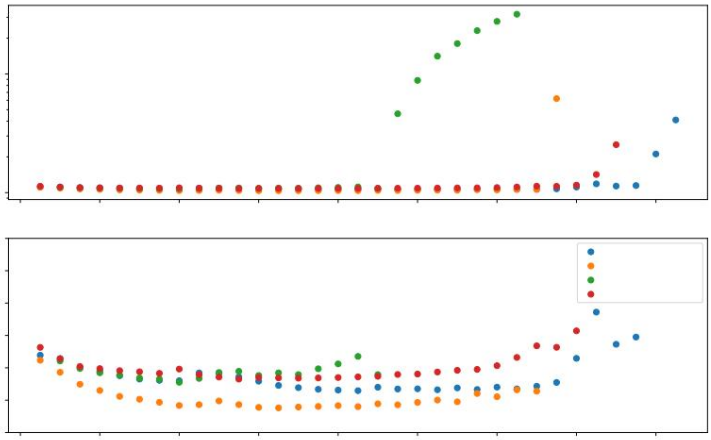


Figura 4.18: Média da latência do 90º percentil em milissegundos para threads reativas, de plataforma e virtuais.

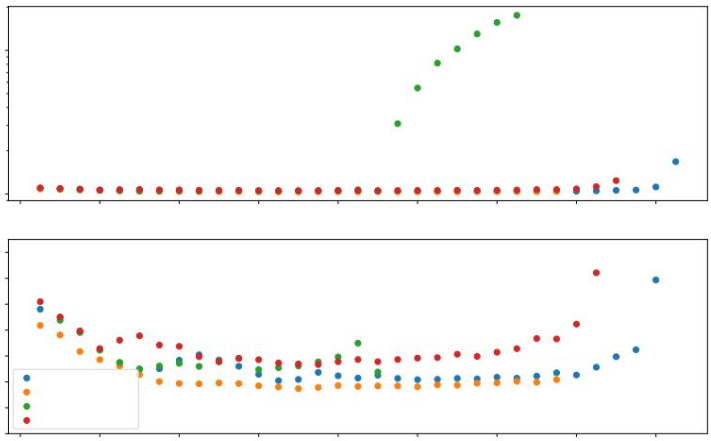


Figura 4.19: Latência mediana em milissegundos para threads reativas, de plataforma e virtuais.

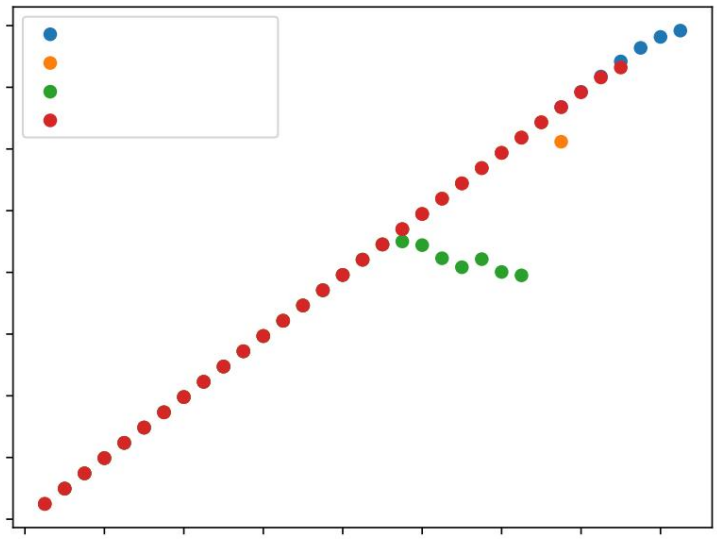


Figura 4.20: Taxa de transferência média (mensagens por segundo) para reativo, plataforma e threads virtuais.

As figuras 4.17, 4.18, 4.19 e 4.20 ilustram as métricas de desempenho. Os threads virtuais têm
A menor latência em todos os percentis foi seguida de perto pela abordagem reativa. No entanto,
A abordagem reativa consegue lidar com mais solicitações por segundo. A taxa de transferência para
As tecnologias são relativamente semelhantes, embora ambas exijam threads de bloqueio ilimitadas e limitadas.
Atinge um patamar de produtividade antes de entrar em colapso.

De forma geral, as threads virtuais apresentam o melhor desempenho com taxas inferiores a 1300 requisições por thread.
Em segundo lugar, a abordagem reativa parece ser a mais estável, sendo capaz de lidar com mais concorrência
antes de falhar. Os threads da plataforma apresentam a maior utilização de recursos de hardware.
onde a abordagem ilimitada utiliza mais CPU e memória, devido ao maior número de threads.

4.1.5 Criação de Tópicos

As tabelas 4.2 e 4.1 mostram o número de threads criadas para os diferentes métodos no início.
e fim dos testes. Observe que a contagem de threads para plataformas limitadas é superior a 100, mas
Isso ocorre devido ao limite de threads definido para as threads que processam as solicitações recebidas. Para o
Em servidores reativos e virtuais, o número de threads não aumenta muito com a carga.
aumenta. No entanto, as threads da plataforma aumentaram a contagem de threads à medida que a carga aumenta.

Tabela 4.1: Média de threads ativas (início/fim) para diferentes métodos

Método	**Calcular**	**E/**	**Multiplicação de matrizes Método misto**
Reativo	**S** 29 / 30 40 / 41		40 / 41 40 / 41
Thread virtual 30 / 32 30 / 34			33 / 34 33 / 33
Plataforma - Limitada 120 / 121 51 / 120			98 / 121 120 / 121
Plataforma - Ilimitada 814 / 1049 98 / 716			875 / 1076 332 / 1170

Tabela 4.2: Média de threads Daemon (início/fim) para diferentes métodos

Método	Calcular	E/	Multiplicação de matrizes	Método misto
Reativo	**S** 27 / 28 38 / 39		38 / 39 38 / 39	
Fio virtual	27 / 28 25 / 30		29/30 28/29	
Plataforma - Limitada	115 / 117 47 / 116		94 / 117 115 / 117	
Plataforma - Ilimitada	810 / 1045 94 / 712		870 / 1072 328 / 1165	

4.1.6 Estabilidade

Na figura 4.21, é apresentada a porcentagem de falhas para cada parâmetro de avaliação. Observe que...
A porcentagem de falhas não se baseia em cinco testes para cargas mais elevadas. Isso ocorre porque os testes são
A operação foi interrompida devido a múltiplos erros consecutivos; alguns desses erros não estão presentes no gráfico acima.
devido aos testes terem sido interrompidos por tempo limite excedido pela máquina atacante, o que significa que estes são
erros do lado do servidor. A partir dos resultados exibidos na figura 4.21, pode-se deduzir que o reativo
Os servidores são mais estáveis. Além disso, para fins de validação, outra observação é que
Para taxas mais altas, a importância do resultado para threads virtuais e de plataforma é reduzida.
devido aos valores médios conterem menos amostras.

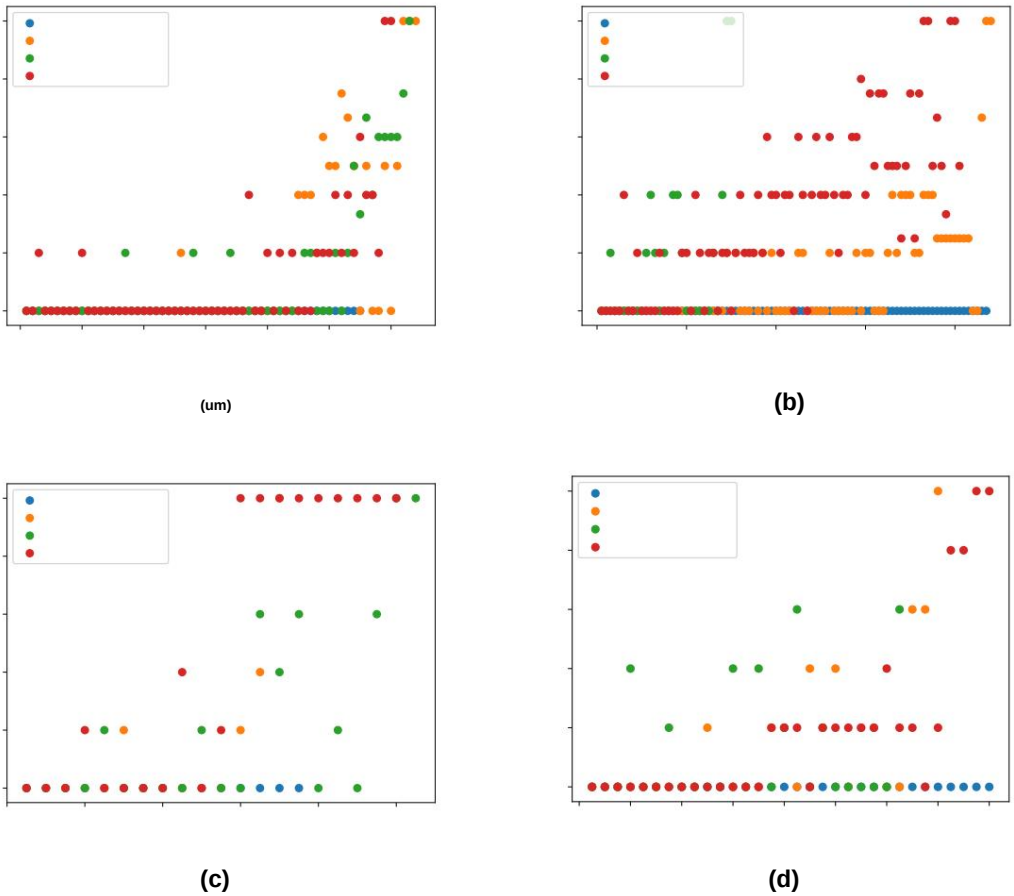


Figura 4.21: Percentagem de falhas para as diferentes tecnologias em diferentes parâmetros de referência. Um teste é considerado reprovado se apresentar um ou mais solicitações com falha.

4.2 Experimento de Carga Constante

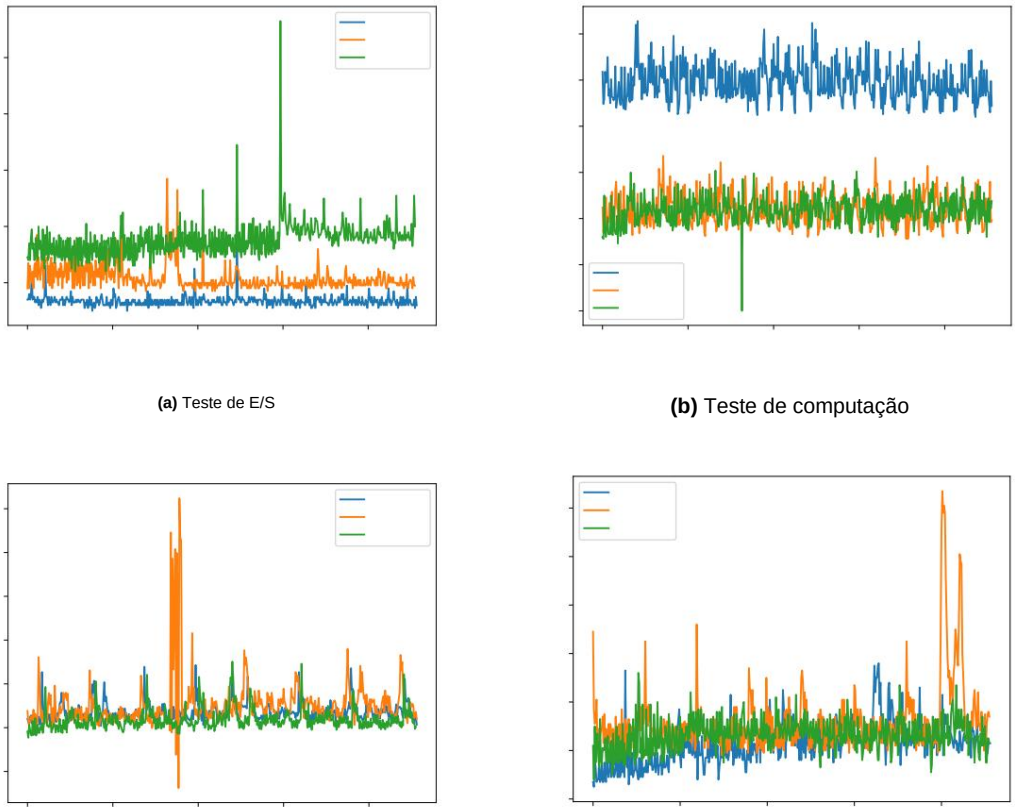
Além de testar o quão bem as diferentes tecnologias se adaptavam à taxa de requisições, nós
Também foram testados sob carga constante. Os resultados são apresentados como no experimento anterior, com as medições das diferentes tecnologias apresentadas lado a lado.
e divididos pelos métodos testados e pelas medições realizadas. Para obter uma compreensão mais aprofundada.
Para compreender a diferença entre as tecnologias, os resultados também contêm dados sobre
pontos seguros e tentativas de bloqueio contestadas. Abaixo, na Tabela 4.3, estão as latências dos diferentes tecnologias conforme medidas neste experimento, com atrasos embutidos subtraídos antes da normalização:

Tabela 4.3: Latências médias entre benchmarks para diferentes percentis

Latências médias de E/S (ms)					
Tecnologia	Significar	50º 90º		99º	99º Normalizado
Reativo	100,79	100,62	101,33	102,45	0,36
Fios virtuais	100,61	100,58	100,71	101,08	0,16
Tópicos da plataforma	102,28	101,26	104,93	106,80	1,00
Calcular latências médias (ms)					
Tecnologia	Significar	50º 90º 99º			99º Normalizado
Reativo	7,88	6.17	11:00	22,50	1,00
Fios virtuais	4,87	4,85	5.15	5,54	0,25
Tópicos da plataforma	4,96	4,80	5.01	6,08	0,27
Latências médias de Matmul (ms)					
Tecnologia	Significar	50º 90º 99º			99º Normalizado
Reativo	40,26	38,83	45,32	52,13	0,05
Fios virtuais	50,85	37,78	40,83	538,44	1,00
Tópicos da plataforma	38,29	37,81	41,05	45,88	0,04
Latências médias mistas (ms)					
Tecnologia	Significar	50º 90º 99º			99º Normalizado
Reativo	105,99	105,36	109,38	113,17	1,00
Fios virtuais	104,17	104,00	105,13	106,61	0,50
Tópicos da plataforma	105,05	104,14	107,52	108,89	0,68

4.2.1 Medições da CPU

A utilização da CPU foi semelhante para as três abordagens de concorrência nos dois testes mistos.
sendo que as threads virtuais são ligeiramente mais instáveis do que as outras duas abordagens. No entanto,
No teste de E/S, houve diferenças claras, e no teste de computação, o desempenho das threads reativas foi significativamente pior, enquanto as threads virtuais e de plataforma tiveram desempenho semelhante.



(a) Teste de E/S

(b) Teste de computação

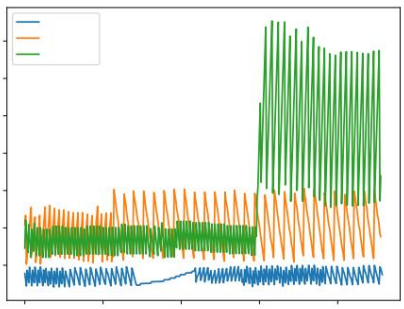
(c) Teste Matmul

(d) Teste misto

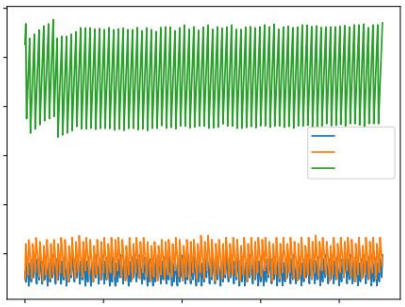
Figura 4.22: Utilização da CPU (%) versus tempo (s) para diferentes testes.

4.2.2 Medidas de memória

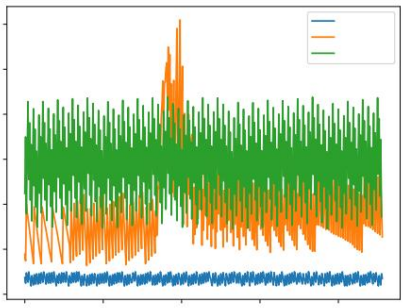
As medições de memória foram semelhantes nos quatro testes, com o desempenho sendo classificado da mesma forma. O servidor reativo apresentou uso de memória consistentemente baixo e estável, os threads da plataforma apresentaram uso de memória alto e volátil, e os threads virtuais tiveram desempenho variável entre o reativo e o da plataforma, com a volatilidade variando de acordo com o método de teste.



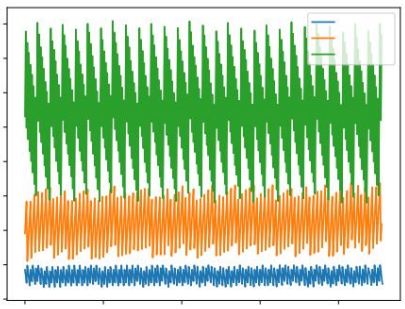
(a) Teste de E/S



(b) Cálculo
teste



(c) Teste de
multiplicação de matrizes

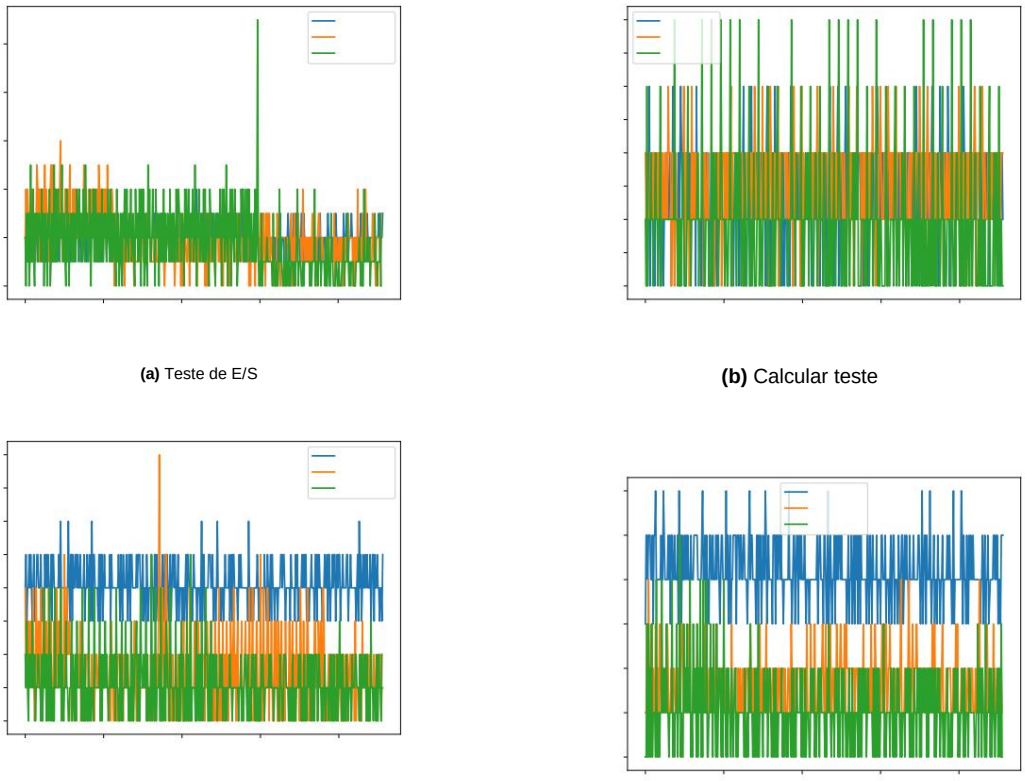


(d) Teste misto

Figura 4.23: Utilização de memória heap (bytes) versus tempo (s) para diferentes testes.

4.2.3 Medições do Safepoint

Na JVM, um ponto de segurança representa um estado no qual todas as referências na pilha estão mapeadas e podem ser contabilizadas. A maior diferença entre as tecnologias testadas pode ser observada nos métodos mistos, onde o método reativo apresenta o maior número de pontos de segurança. Isso permite que o servidor reativo gerencie a memória de forma mais eficiente devido à coleta de lixo mais frequente.



(a) Teste de E/S

(b) Calcular teste

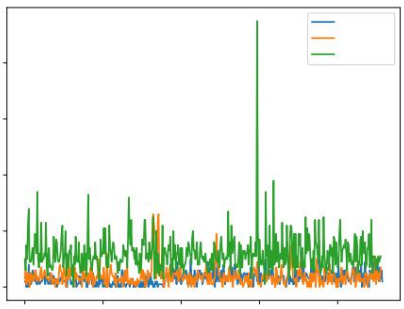
(c) Teste de multiplicação de matrizes

(d) Teste misto

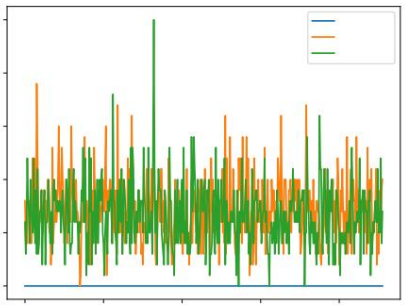
Figura 4.24: Número de pontos seguros versus tempo (s) para diferentes testes.

4.2.4 Medições de Sincronização

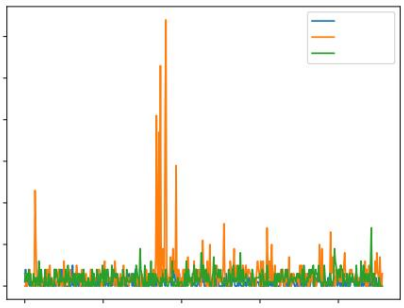
As tentativas de bloqueio disputado são incluídas para melhor compreender as diferenças de desempenho apresentadas acima. Elas são uma boa medida de quão bem as tecnologias lidam com concorrência e sincronização, já que o tempo gasto esperando por um bloqueio ocupado é tempo gasto em espera (e, para threads virtuais, isso pode fazer com que sejam desmontadas de sua thread do kernel e armazenadas no heap). A principal conclusão dos resultados abaixo é que o servidor reativo apresentou um número consistentemente menor de tentativas de bloqueio disputado.



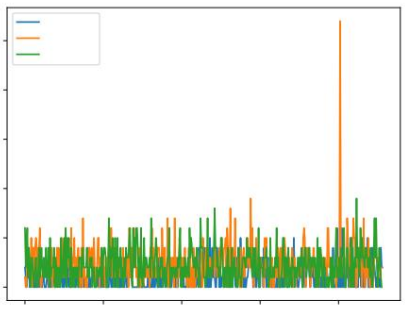
(a) Teste de E/S



(b) Cálculo
teste



(c) Teste de
multiplicação de matrizes



(d) Teste misto

Figura 4.25: Número de tentativas de bloqueio disputadas versus tempo (s) para diferentes testes.

4.2.5 Análise da Pilha de Chamadas

A pilha de chamadas foi analisada com a ferramenta Flame Graph de Brendan Gregg, com dados coletados usando o JStack (Oracle Help Center 2024). A análise da pilha de chamadas foi incluída neste relatório para melhor compreender a influência da sobrecarga na diferença de desempenho entre as três tecnologias de concorrência testadas. Consideramos duas métricas de especial importância: a altura da pilha de chamadas e a largura das chamadas aos métodos executados nos servidores (ou seja, a porcentagem do desempenho dos programas que foi utilizada para realizar a tarefa desejada). Isso se deve ao fato de que a largura da pilha de chamadas permite entender quanto tempo de CPU o servidor gasta na execução do benchmark em si, em comparação com o gerenciamento do servidor e da concorrência. A altura da pilha de chamadas indica, simplesmente, quantas chamadas de função determinadas tecnologias exigem para gerenciar a concorrência.

As alturas das pilhas de chamadas foram:

- 56 para threads de plataforma
- 56 para threads virtuais
- 85 para reativos

Taxas: 1200 solicitações por segundo para computação, 1600 para E/S, 300 para computação de multiplicação de matrizes e 600 matmul dormir.

A porcentagem de uso da CPU dedicada aos métodos implementados é apresentada em
A tabela abaixo deve ser lida como o complemento da porcentagem de uso da CPU.
gastos com despesas gerais.

Tabela 4.4: Percentual de utilização da CPU gasto no benchmark real.

CPU utilizada pelo método (%)				
Tipo de concorrência E/S		Calcular matriz		Sono da Matriz
Reativo 29 21	0			10
Fios virtuais 43 28	3			20
Tópicos da plataforma	0	35	27	20

A partir dos dados da tabela 4.4, podemos deduzir que Java Reactive (concorrência assíncrona)
Possui uma sobrecarga significativa em comparação com threads de plataforma e virtuais. Isso é
deduzido pelo fato de que todo o tempo gasto em funções fora do benchmark principal é contabilizado.
pode ser visto como um custo logístico adicional relacionado ao gerenciamento do servidor, da conexão de rede e, principalmente, da logística.
a concorrência. Mesmo que a altura das pilhas de chamadas fosse a mesma para virtual- e
threads da plataforma, é razoável concluir que o agendador para threads virtuais (eles
são tratados em um ForkJoinPool) é mais eficiente do que para threads de plataforma.

4.2.6 Análise de Conexão de Rede

Conforme descrito no capítulo de metodologia, realizamos dois testes da própria rede. Primeiro, nós...
tentamos chamar nosso método "vazio" com taxas crescentes. Ao fazer isso, taxas muito altas foram obtidas.
alcançado. O WireShark foi então usado para analisar o potencial da rede como um gargalo durante
Execuções de alguns de nossos testes de aumento gradual de carga. A ferramenta forneceu informações interessantes sobre a anatomia.
das falhas do servidor. O cenário mais comum era o cliente começar a solicitar retransmissões por estar sobrecarregado
e não conseguir acompanhar. Isso geralmente era seguido logo em seguida por, por exemplo,
confirmações duplicadas e outros sinais de ineficiência. Após esse ponto de ruptura, a maioria dos
O tráfego de rede estava "sobrecarregado" até que o servidor enviasse uma solicitação para fechar a conexão.
Os resultados desses experimentos indicam que a conexão de rede não é um fator limitante.
deste estudo, pois conseguimos transmitir a taxas muito mais altas do que as utilizadas em nossos experimentos.
No teste a seco, o WireShark mostrou que os erros se originaram no processo do servidor.

4.3 Resumo dos Resultados

Nos testes de E/S, os threads virtuais apresentaram o melhor desempenho geral em relação à latência.
Percentis 99 e 90. Os threads virtuais apresentaram latência até 1 ms menor do que os threads reativos.
fios no percentil 90 (figura 4.8), sendo a diferença maior em taxas mais baixas.
Os threads da plataforma, tanto limitados quanto ilimitados, apresentaram um desempenho visivelmente pior do que
threads reativas e virtuais para latência em todos os percentis com desempenho mediano que
Foi de 1 a 3 milissegundos mais lento para a maioria das taxas de ataque e apresentou um desempenho no 90º percentil.
Isso representou uma lentidão de cerca de 4 a 5 milissegundos para a maioria das taxas. O limite de threads da plataforma era de 100.
Os threads alcançaram uma latência muito maior (cerca de vinte vezes maior em médias distâncias) do que
todas as outras tecnologias. Isso confirma o resultado de Pufek et al. (2020), Beroniÿ et al.

(2022) e Beroniÿ et al. (2021), onde threads de plataforma limitadas apresentaram desempenho significativamente pior do que threads virtuais. As diferenças entre threads de plataforma ilimitadas e threads virtuais não foram tão óbvias; no entanto, as threads de plataforma ilimitadas utilizaram significativamente mais memória. A abordagem reativa apresentou a menor utilização de recursos em termos de memória e CPU. O heap para threads de plataforma limitadas aumentou drasticamente quando a taxa atingiu 1000 requisições por segundo, pouco antes da abordagem falhar. A razão para a falha foi simplesmente a falta de threads suficientes para um nível mais alto de concorrência. Um atraso de 100 ms significa que uma thread bloqueada pode lidar com um máximo de dez requisições por segundo, enquanto 100 threads bloqueadas podem lidar com, no máximo, 1000 requisições por segundo. Depois disso, as requisições são colocadas em uma fila, levando a um aumento na latência. Threads virtuais e de plataforma ilimitada apresentaram utilização de CPU semelhante, que aumenta linearmente, enquanto a utilização do heap aumentou exponencialmente com a taxa de requisições.

Para o método computacionalmente intensivo, as threads de plataforma apresentaram as melhores latências gerais nos percentis 50, 90 e 99. Elas foram cerca de 0,75 ms mais rápidas que as threads virtuais na mediana. Para taxas inferiores a 2200, as threads virtuais apresentaram um desempenho ligeiramente melhor que os fluxos reativos em termos de latência em todos os percentis. Após esse ponto, os resultados tornaram-se instáveis, provavelmente devido à alta utilização de recursos. Em relação à utilização de recursos, as threads de plataforma apresentaram a menor utilização de CPU, seguidas pelas threads virtuais e, por último, pelos fluxos reativos. Quanto à utilização de memória, os fluxos reativos são os mais eficientes, seguidos de perto pelas threads virtuais, embora a utilização de memória para as threads virtuais aumente exponencialmente.

As threads da plataforma apresentam, em geral, uma utilização de memória pior do que as threads reativas e virtuais.

Para a multiplicação de matrizes com 25 milissegundos de atraso, as threads virtuais apresentaram latências menores em todos os percentis, seguidas pelas threads de plataforma e, por último, pelas threads reativas. Para taxas mais altas, as threads de plataforma têm um desempenho melhor do que as reativas. A utilização da CPU para todas as tecnologias é relativamente semelhante, embora as threads de plataforma com número limitado de threads apresentem menor utilização da CPU do que as threads de plataforma com número ilimitado de threads (provavelmente devido aos requisitos adicionais de CPU criados pelo gerenciamento de mais threads de plataforma). As threads reativas apresentam a menor utilização de memória em geral. Ambos os tipos de threads de plataforma apresentam uma utilização de memória relativamente constante e estável, enquanto a das threads virtuais aumenta exponencialmente em taxas mais altas.

Para o método misto, as threads virtuais apresentaram latências menores do que as outras tecnologias em todas as amostras coletadas, nos percentis 50, 90 e 99. Em seguida, vieram as threads reativas, com uma diferença decrescente à medida que a taxa aumenta. As threads de plataforma limitadas apresentam uma latência que aumenta exponencialmente, pois não podem criar mais threads. As threads virtuais falham antes das threads de plataforma (ilimitadas) e das threads reativas. O servidor reativo apresentou o melhor desempenho após a falha das threads virtuais. Para taxas mais baixas, as threads reativas, virtuais e de plataforma limitada apresentaram utilização de CPU semelhante, enquanto as threads de plataforma ilimitadas apresentaram uma utilização de CPU maior. As threads reativas apresentaram a menor utilização de memória, seguidas pelas threads virtuais e pelas threads de plataforma limitada.

Os testes de carga constante levam a conclusões semelhantes às dos testes de carga variável. O servidor reativo funciona de forma mais estável e utiliza menos memória do que os servidores baseados em plataforma e em threads virtuais. No entanto, apresenta latências significativamente maiores do que o servidor baseado em threads virtuais.

Capítulo 5
Conclusões

5.1 Abordando as questões de pesquisa

Com base nos dados coletados e na metodologia rigorosa, sentimos-nos à vontade para responder às questões de pesquisa no contexto do Spring e do hardware utilizado. **A RQ1** indaga se há diferença entre as diferentes técnicas de concorrência na JVM para sistemas de alta carga. Os resultados desta tese indicam que, de fato, existem diferenças. As diferenças de desempenho entre as tecnologias podem ser resumidas da seguinte forma:

i) Para tarefas puramente computacionais, os threads da plataforma têm um desempenho melhor do que os threads virtuais. e reativo em termos de latência.

ii) Para tarefas limitadas por E/S, threads virtuais e reativas têm um desempenho melhor do que threads de plataforma, sendo que as threads virtuais apresentam uma latência ligeiramente melhor no 90º e 99º percentil.

iii) Para um método que utiliza tanto E/S quanto computação (multiplicação de matrizes), as threads virtuais apresentam o melhor desempenho em termos de latência.

iv) Para um método que utiliza tanto E/S quanto computação (com mais E/S), as threads virtuais apresentam o melhor desempenho. No entanto, elas são seguidas de perto pelos fluxos reativos.

As diferenças de utilização de hardware entre as tecnologias podem ser resumidas da seguinte forma:

a) Para tarefas puramente computacionais, os threads da plataforma têm a melhor utilização de CPU, embora tenham a pior utilização de memória (usando mais memória), enquanto os fluxos reativos têm a melhor utilização de memória.

b) Para tarefas puramente de E/S, os fluxos reativos apresentam a melhor utilização de CPU e memória. Threads virtuais e threads de plataforma ilimitadas têm utilização de memória e CPU semelhante, embora as threads de plataforma criem 20 vezes mais threads do kernel.

c) Para um método que utiliza tanto E/S quanto computação (multiplicação de matrizes), todas as tecnologias apresentam utilização de CPU semelhante. Fluxos reativos têm a melhor utilização de memória, enquanto threads virtuais e threads de plataforma ilimitadas têm uma utilização de memória ruim.
Threads de plataforma ilimitadas têm uma utilização de memória geral maior, mas esse consumo aumenta exponencialmente para threads virtuais.

d) Para um método que utiliza tanto E/S quanto computação (com mais E/S), os fluxos reativos apresentam a melhor utilização de CPU e memória. Threads de plataforma ilimitadas apresentam a pior utilização de memória e CPU.

Outra observação é que os fluxos reativos são menos propensos a falhar nas solicitações em comparação com as outras abordagens.

A razão pela qual os threads de plataforma têm o melhor desempenho em tarefas puramente computacionais (i) pode ser explicada pela abordagem ser mais eficaz para gerenciar recursos de CPU.
A razão pela qual os fluxos reativos e os threads virtuais têm um desempenho melhor do que os threads de plataforma para E/S pura (ii) pode ser parcialmente explicada pelo fato de que os threads de plataforma são limitados aos threads do kernel, onde threads adicionais do kernel levam a um aumento do uso da CPU devido à sobrecarga.
A razão pela qual os threads virtuais têm o melhor desempenho em uma combinação de E/S e computação (iii) e (iv) pode ser provavelmente descrita por menor sobrecarga (em comparação com fluxos reativos) e menos recursos gastos no gerenciamento de threads.

Para responder explicitamente **à RQ1,** existe uma diferença entre as tecnologias. De modo geral, threads virtuais oferecem a melhor latência para todos os métodos que contêm elementos de E/S, enquanto fluxos reativos apresentam a melhor utilização de hardware. Além disso, fluxos reativos são superiores às outras tecnologias em relação às taxas de falha. É importante ressaltar que essa conclusão se baseia em um experimento no qual os servidores não foram sobrecarregados devido à saturação de hardware (uso irrealisticamente alto de CPU e memória). A razão pela qual essas observações são descartadas é que os servidores tendem a se comportar de forma imprevisível quando ficam sobrecarregados. Além disso, a taxa de aceitação de falhas igual a zero significa que a significância dos resultados diminui à medida que a taxa de falhas aumenta, devido ao aumento das taxas de erro para todas as tecnologias, exceto fluxos reativos.

A segunda questão de pesquisa **(RQ2)** investiga se threads virtuais são uma alternativa viável a sistemas reativos. Com base na resposta da primeira questão de pesquisa **(RQ1) ,** a resposta é sim. No entanto, é importante ressaltar que sistemas reativos ainda são um método bastante viável. Ao escolher entre threads virtuais ou fluxos reativos, é preciso considerar fatores importantes como desempenho (latência), utilização de hardware e estabilidade. Os resultados desta tese indicam que fluxos reativos apresentam latências maiores do que threads virtuais, mas possuem uma utilização de hardware superior (especialmente em relação à memória) e são, no geral, mais estáveis. Portanto, a introdução de threads virtuais pode levar a custos de hardware mais elevados e a um maior número de requisições com falha, mesmo que, em alguns casos de uso, possam ser mais rápidas.

5.2 Contribuição e Pesquisa Futura

Nosso estudo confirma os resultados de (Beroniÿ et al. 2021), (Beroniÿ et al. 2022) e (Beroniÿ et al. 2022) em relação ao melhor desempenho de threads virtuais em operações de E/S em comparação com threads de plataforma limitadas. Além disso, o estudo amplia parcialmente o de Navarro et al. (2023) ao comparar threads virtuais e fluxos reativos em aplicações Spring. Nosso estudo alcançou conclusões semelhantes às de Navarro et al. (2023) quanto à melhor utilização de recursos dos fluxos reativos. No entanto, os resultados de desempenho foram diferentes, visto que Navarro et al. (2023) concluíram que os fluxos reativos apresentaram melhor desempenho, embora seja muito importante ressaltar que

Navarro et al. (2023) compararam o desempenho em cargas mais elevadas, quando a utilização da CPU estava entre 50% e 100%, enquanto nesta tese a comparação baseou-se principalmente em resultados para cargas mais baixas. Isso se deu porque os resultados em cargas que levam a uma alta utilização do hardware foram considerados inadequados e menos precisos. Assim, os estudos se complementam, avaliando o desempenho das tecnologias em todo o espectro de níveis de concorrência.
Os resultados do nosso estudo corroboram os resultados deles, visto que a latência para threads virtuais tende a ser pior do que para fluxos reativos em alguns métodos sob cargas elevadas. Outras diferenças potenciais podem ser atribuídas a diferentes metodologias de teste e diferentes benchmarks.

Nosso estudo contribui para a base de conhecimento atual ao fornecer uma comparação detalhada entre threads de plataforma, threads virtuais e fluxos reativos em um contexto de alta carga. Os resultados indicam que existem diferenças entre as tecnologias e que threads virtuais são uma opção válida para lidar com a concorrência no servidor, visto que apresentaram um desempenho muito bom em relação à latência. O resultado que apoia o uso de threads virtuais sugere que pesquisas adicionais devem ser conduzidas sobre o assunto. Áreas de particular importância para pesquisas futuras incluem estabilidade, escalabilidade e aplicações maiores (com o risco de introduzir efeitos de caixa-preta nos testes). Possíveis questões de pesquisa incluem:

Q1. Os threads virtuais, os fluxos reativos e os threads de plataforma escalam de forma diferente com a alteração dos recursos de hardware?

Q2. Como os fluxos reativos e as threads virtuais diferem em termos de estabilidade?

5.3 Limitações e Considerações

Em nossa metodologia, diversas precauções foram tomadas para estabilizar o ambiente de teste e mitigar o comportamento um tanto imprevisível da JVM. Após essas precauções, a variância entre as diferentes iterações de teste foi reduzida. No entanto, ainda existe alguma variância dentro dos testes, o que pode afetar os resultados. Isso é muito prevalente na extremidade superior do espectro de carga, onde a alta utilização do hardware levou a um aumento no número de testes com falha. Embora os testes com falha não tenham sido considerados em nosso estudo, eles resultaram em um número menor de amostras para calcular a média, o que reduz a significância de algumas partes de nossos testes. Isso, juntamente com o comportamento relativamente imprevisível em cargas mais altas, dificultou a obtenção de conclusões sobre os resultados com alta utilização de recursos. Como uma utilização de recursos muito alta não é comum em servidores reais, esses resultados não foram considerados em nossas conclusões. Outra limitação do nosso estudo é o hardware. Os resultados podem ser diferentes em diferentes configurações de hardware e sistemas operacionais. Nossa configuração de hardware também não nos permitiu controlar a frequência da CPU de maneira adequada, o que torna nossos testes com uso intensivo de CPU menos confiáveis. No entanto, acreditamos que isso seja atenuado pelo fato de o ambiente de hardware e as variações de temperatura observadas terem sido praticamente iguais para as diferentes tecnologias.

5.4 Conclusões em resumo

Esta tese teve como objetivo investigar três técnicas de concorrência: threads virtuais, fluxos reativos e threads de plataforma bloqueantes. O objetivo geral foi investigar se as threads virtuais são uma opção viável para aplicações altamente concorrentes. Duas questões de pesquisa foram:

A primeira questão de pesquisa **(RQ1)** foi formulada para investigar se existe alguma diferença de desempenho entre as diferentes tecnologias. A resposta à **RQ1** foi afirmativa, com as principais conclusões indicando que os threads virtuais apresentam um desempenho geral melhor do que os outros sistemas durante as operações de E/S. No entanto, os fluxos reativos têm melhor utilização de recursos e são muito menos propensos a falhas nas solicitações do cliente. A segunda questão de pesquisa **(RQ2)** questiona se os threads virtuais são uma alternativa viável aos fluxos reativos. A resposta à **RQ2** foi **afirmativa,** mas ao selecionar a tecnologia, é preciso considerar tempos de resposta rápidos versus tolerância a falhas nas solicitações e recursos de hardware. Se um tempo de resposta rápido for essencial e houver alguma tolerância a falhas nas solicitações, os threads virtuais podem ser uma opção melhor do que os fluxos reativos. Mas se houver tolerância zero a falhas nas solicitações e restrições de hardware, os fluxos reativos podem ser a melhor opção.

Em resumo, esta tese encontrou diferenças distintas entre as tecnologias. Os resultados indicam que threads virtuais podem oferecer vantagens em termos de desempenho. Isso, aliado ao fato de que a implementação de threads virtuais é mais fácil em comparação com fluxos reativos, indica que mais pesquisas devem ser feitas sobre o assunto. Áreas de interesse para pesquisas futuras estão relacionadas à escalabilidade e à estabilidade. Também encorajamos o leitor a tirar suas próprias conclusões a partir da riqueza de dados aqui apresentada.

Bibliografia

(AWS), AWS (2024), 'Melhores práticas do Amazon ECS - provedores de capacidade e escalonamento automático'.
URL: *https:// docs.aws.amazon.com/ AmazonECS/ latest/ bestpracticesguide/ capacity-dimensionamento automático.html*

Belson, B., Holdsworth, J., Xiang, W. & Philippa, B. (2019), 'Um levantamento de programação assíncrona usando corrotinas na internet das coisas e sistemas embarcados', *ACM Transactions on Embedded Computing Systems (TECS)* 18(3), 1–21.

Ben Weidig (2023), 'Olhando para java 21: Threads virtuais'.
URL: *https:// belief-driven-design.com/ looking-at-java-21-virtual-threads-bd181/*

Beroniÿ, D. (2024), 'Diálogo com pesquisador sobre sua pesquisa anterior', Comunicação pessoal. Correspondência por e-mail enviada em 15/03/2024.

Beroniÿ, D., Modriÿ, L., Mihaljeviÿ, B. & Radovan, A. (2022), Comparação de estruturas Construções de concorrência em Java e Kotlin – threads virtuais e corrotinas, *em* '2022 45º Convenção Internacional Jubileu sobre Tecnologia da Informação, Comunicação e Eletrônica (MIPRO)', IEEE, pp. 1466–1471.

Beroniÿ, D., Pufek, P., Mihaljeviÿ, B. & Radovan, A. (2021), Sobre a análise de threads virtuais– Um modelo de concorrência estruturado para aplicações escaláveis na JVM, *na* 44ª Convenção Internacional sobre Tecnologia da Informação, Comunicação e Eletrônica de 2021. (MIPRO)', IEEE, pp.

Blackburn, SM, Garner, R., Hoffmann, C., Khang, AM, McKinley, KS, Bentzur, R., Di-wan, A., Feinberg, D., Frampton, D., Guyer, SZ et al. (2006), Os benchmarks dacapo: Desenvolvimento e análise de benchmarks em Java, *em* 'Anais da 21ª Conferência Anual da ACM'. Conferência SIGPLAN sobre sistemas, linguagens e aplicações de programação orientada a objetos', pp. 169–190.

Brian Olson (2019), 'A programação assíncrona é realmente difícil'.
URL: *https:// devblabs.medium.com/ asynchronous-programming-is-really-really-hard-8f7d97d7cddf*

Brodu, E., Frénot, S. & Oblé, F. (2015), Em direção à atualização automática de callbacks para promises, *em* 'Anais do 1º Workshop sobre Sistemas de Tempo Real Totalmente Web', pp. 1–8.

Bull, JM, Smith, L., Westhead, MD, Henty, D. & Davey, R. (1999), Uma metodologia para avaliação comparativa de aplicações Java Grande, *em* 'Anais da conferência ACM 1999 sobre Java Grande', pp. 81–88.

Chitra, LP & Satapathy, R. (2017), Comparação e avaliação de desempenho do node.js e do servidor web tradicional (iis), *em* '2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)', IEEE, pp. 1–4.

Choi, B., Park, J., Lee, C. & Han, D. (2021), phpa: Uma estrutura de autoescalonamento proativo para cadeia de microsserviços, *em* '5º Workshop Ásia-Pacífico sobre Redes (APNet 2021)', pp. 65– 71.

Cloud, G. (2024), 'Google Cloud Bigtable - monitorando uma instância'.
URL: *https:// cloud.google.com/ bigtable/ docs/ monitoring-instance*

Edwards, J. (2009), Reação coerente, *em* 'Anais da 24ª conferência ACM SIGPLAN sobre linguagens e aplicações de sistemas de programação orientada a objetos', pp. 925–932.

Eeckhout, L., Georges, A. & De Bosschere, K. (2003), Como os programas Java interagem com máquinas virtuais no nível microarquitetural, *em* 'Anais da 18ª conferência anual ACM SIG-PLAN sobre programação orientada a objetos, sistemas, linguagens e aplicações', pp. 169–186.

Fan, Q. & Wang, Q. (2015), Comparação de desempenho de servidores web com arquiteturas diferentes: um estudo de caso usando carga de trabalho de alta concorrência, *em* '2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)', IEEE, pp. 37–42.

Gregg, B. (2016), 'O gráfico de chama', *Communications of the ACM* 59(6), 48–57.

Gu, D., Verbrugge, C. & Gagnon, EM (2006), Fatores relativos na análise de desempenho de máquinas virtuais Java, *em* 'Anais da 2ª Conferência Internacional sobre Ambientes de Execução Virtual', pp. 111–121.

Haines, S. (2006), *Pro Java EE* 5, Springer.

Hamed, O. & Kafri, N. (2009), Teste de desempenho para arquiteturas de aplicativos baseados na web (.net vs. java ee), *em* '2009 First International Conference on Networked Digital Technologies', IEEE, pp. 218–224.

Harji, AS, Buhr, PA & Brecht, T. (2012), Comparando arquiteturas de servidores web multi-core de alto desempenho, *em* 'Anais da 5ª Conferência Internacional Anual de Sistemas e Armazenamento', pp. 1–12.

Heisenberg, W. (1927), 'Über den anschaulichen inhalt der quantentheoretischen kinematik e mecânica', *Zeitschrift für Physik* 43(3-4), 172–198.

Jiang, ZM & Hassan, AE (2015), 'Uma pesquisa sobre testes de carga de sistemas de software de grande escala', *IEEE Transactions on Software Engineering* 41(11), 1091–1118.

Jiang, ZM, Hassan, AE, Hamann, G. & Flora, P. (2009), Análise automatizada de desempenho de testes de carga, *em* '2009 IEEE International Conference on Software Maintenance', IEEE, pp. 125–134.

Jils Matthew et al. (1999), 'Análise e desenvolvimento de benchmarks java grande'.
URL: *https:// www.researchgate.net/ publication/ 2610597AnalysisandDevelopmento fJavaGrandeBenchmarks*

Jonas Bonér, ea (2014), *O Manifesto Reativo.*

Kambona, K., Boix, EG & De Meuter, W. (2013), Uma avaliação da programação reativa e promessas para estruturar aplicações web colaborativas, *em* 'Anais do 7º Workshop sobre Linguagens Dinâmicas e Aplicações', pp. 1–9.

Kim, K.-J., Jeong, I.-J., Park, J.-C., Park, Y.-J., Kim, C.-G. & Kim, T.-H. (2007), 'O impacto do desempenho do serviço de rede na satisfação e lealdade do cliente: caso do serviço de internet de alta velocidade na Coreia', *Expert Systems with Applications* 32(3), 822–831.
URL: *https:// www.sciencedirect.com/ science/ article/ pii/ S0957417406000388*

Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N. & Yuan, D. (2016), Não seja pego no frio, aqueça sua JVM: Entenda e elimine a sobrecarga de aquecimento da JVM em sistemas de dados paralelos, *em* '12º Simpósio USENIX sobre Projeto e Implementação de Sistemas Operacionais (OSDI 16)', pp. 383–400.

Liu, M. & Ding, X. (2010), Sobre a confiabilidade da medição e contabilização do uso da CPU, *em* '2010 IEEE 30th International Conference on Distributed Computing Systems Workshops', pp. 82–91.

Madsen, M., Lhoták, O. & Tip, F. (2017), 'Um modelo para raciocinar sobre promessas javascript', *Anais da ACM sobre Linguagens de Programação* 1 (OOPSLA), 1–24.

Navarro, A., Ponge, J., Le Mouël, F. & Escoffier, C. (2023), Considerações para a integração de threads virtuais em um framework Java: um exemplo Quarkus em um ambiente com recursos limitados, *em* 'DEBS'2023-17ª Conferência Internacional ACM sobre Sistemas Distribuídos e Baseados em Eventos'.

Nor Sobri, NA, Abas, MAH, Mohd Yassin, AI, Megat Ali, MSA, Md Tahir, N. & Zabidi, A. (2022), 'Pool de conexões de banco de dados em arquitetura de microsserviços', *Journal of Electrical and Electronic Systems Research (JEESR)* **20,** 29–33.

OpenSignal (2020), 'Latência média de rede 4G e 3G por provedor nos Estados Unidos em 2019 (em milissegundos).
URL: *https:// www-statista-com.ludwig.lub.lu.se/ statistics/ 818205/4g-and-3g-network-latency-in-the-united-states-2017-by-provider/*

Oracle Corporation (2023), 'Documentação do Java 21'.
URL: *https:// docs.oracle.com/ en/ java/ javase/ 21/ core/ virtual-threads.html#GUID-2BCFC2DD-7D84-4B0C-9222-97F9C7C6C521*

Oracle Corporation (2024a), 'Documentação do Java 21'.
URL: *https:// docs.oracle.com/ en/ java/ javase/ 21/ core/ virtual-threads.html#GUID-8AEDDBE6-F783-4D77-8786-AC5A79F517C0*

Oracle Corporation (2024b), 'Visualvm'.
URL: *visualvm.github.io*

Centro de Ajuda da Oracle (2024), 'Jstack'.
URL: *https:// docs.oracle.com/ javase/ 8/ docs/ technotes/ tools/ unix/ jstack.html*

Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A. & Cheriton, DR (2007), 'Comparando o desempenho de arquiteturas de servidores web', *ACM SIGOPS Operating Systems Review* 41(3), 231–243.

Park, J., Choi, B., Lee, C. & Han, D. (2021), Graf: Uma estrutura de alocação de recursos proativa baseada em rede neural gráfica para microsserviços orientados a lentidão, *em* 'Anais da 17ª Conferência Internacional sobre Experimentos e Tecnologias de Redes Emergentes', pp. 154–167.

Ponge, J., Navarro, A., Escoffier, C. & Le Mouël, F. (2021), Analisando o desempenho e os custos de bibliotecas de programação reativa em Java, *em* 'Anais do 8º Workshop Internacional ACM SIGPLAN sobre Linguagens e Sistemas Reativos e Baseados em Eventos', pp. 51–60.

Pufek, P., Beroniÿ, D., Mihaljeviÿ, B. & Radovan, A. (2020), Alcançando concorrência estruturada eficiente por meio de fibras leves na máquina virtual Java, *em* '2020 43ª Convenção Internacional sobre Tecnologia da Informação, Comunicação e Eletrônica (MIPRO)', IEEE, pp. 1752–1757.

Fluxos reativos (2024), Fluxos reativos.
URL: *https:// www.reactive-streams.org/*

Salah, T., Zemerly, MJ, Yeun, CY, Al-Qutayri, M. & Al-Hammadi, Y. (2017), Comparação de desempenho entre serviços baseados em contêineres e baseados em máquinas virtuais, *em* '2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)', IEEE, pp. 185–190.

Schuler, L., Jamil, S. & Kühl, N. (2021), Alocação de recursos baseada em IA: Aprendizado por reforço para escalonamento automático adaptativo em ambientes sem servidor, *em* '2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)', IEEE, pp. 804–811.

Sim, SE, Easterbrook, S. & Holt, RC (2003), Usando benchmarking para avançar a pesquisa: um desafio para a engenharia de software, *em* '25ª Conferência Internacional sobre Engenharia de Software, 2003. Anais.', IEEE, pp. 74–83.

Primavera (2024), Spring web reativo.
URL: *https:// docs.spring.io/ spring-boot/ docs/ current/ reference/ htmlsingle/ #web.reactive*

Primavera (2024), 'Framework Spring'.
URL: *https:// spring.io/ projects/ spring-framework*

Vahalia, U. (1996), *UNIX internals : the new frontiers.,* An Alan R. Apt book, Prentice Hall.

Zhang, Y., Li, M. & Tong, F. (2023), 'Balanceamento de carga com eficiência energética para tarefas divisíveis em clusters heterogêneos', *Transactions on Emerging Telecommunications Technologies* 34(10), e4829.

Apêndices

Apêndice A

UM

A.1 Roteiros

A.1.1 Script para envio de requisições HTTP

Este script foi usado no cliente para enviar solicitações HTTP com Vegeta e salvar os resultados de forma organizada em um arquivo.

#!/bin/bash #

Chamado com: bash bashScript "testSleep" "B" "S" "output" "final" "10s" ulimit -u 4000

ulimit -n 20000 ulimit
-s 20000
ulimit ilimitado

eco de
aquecimento "POST http://169.254.249.75:8080/$1" | ataque vegeta -duração=60s -taxa=3
| tee results.bin | vegeta report curl -X POST
"http://169.254.249.75:8080/gc"
Durma 20

Ajuste a taxa para cerca de 60 de uso da CPU
echo "POST http://169.254.249.75:8080/$1" | vegeta attack -duration=60s -rate=1 | tee results.bin | vegeta
report curl -X POST "http://169.254.249.75:8080/
gc"
Durma 20

echo "POST http://169.254.249.75:8080/$1" | ataque vegeta -duração=60s -taxa=1
| tee results.bin | relatório vegeta

```
curl -X POST "http://169.254.249.75:8080/gc"
Durma 20

para i em {1..150}
fazer
  taxa de ataque : echo "POST http://                    :" $(( 50 * $i )) echo "iniciar
  169.254.249.75:8080/$1" | vegeta attack -duration="$6" -timeout=70s -rate=$(( 50 * $i )) -timeout=0 -max-workers 100000 | tee
      results.bin | vegeta report --type json | python3 script.py -rate=$(( 50 * $i )) -type="$2" -method="$3" -fileName="$4"
       -durat="$6" echo "tempo de espera para reiniciar"


  Sleep 60
  curl -X POST "http://169.254.249.75:8080/gc"
  Durma 20


python3 processAttack.py -fileName="$4" -resultFile="$5"
```

Apêndice B

Dados

B.1 Testes de iteração

Método	Calcular multiplicação de matrizes de entrada/saída, método misto, soma.				
Reativo	247	423 165 995	160		
Fio virtual	251	373 115 848	109		
Plataforma - Limitada	276	123	139	105	643
Plataforma - Ilimitada	265	257	51	125	698
Soma	1039	1176	459	510	3184

Tabela B.1: Número de observações separadas por benchmark e tecnologia.

Método	Calcular multiplicação de matrizes de entrada/saída, método misto, soma.				
Reativo	10868	18612 7040 7260 43780			
Tópico virtual	11044	16412 4796 5060 37312			
Plataforma - Limitada	12144	5412	6116	4620	28292
Plataforma - Ilimitada	11660	11308	2244	5500	30712
Soma	45716	51744	20196	22440	140096

Tabela B.2: Número total de amostras (linhas vezes colunas) separadas por meio de benchmarks e tecnologia.

Apêndice C
Métodos de referência

C.1 Imperativo

C.1.1 E/S

```
public void io() throws InterruptedException{ Thread.sleep(DELAY);

}
```

C.1.2 Calcular

```
public double compute() { double
    result = 0.0; double rand =
    System.currentTimeMillis(); int N = 1000; for (int i = 3; i <
    N; i += 1) {

        para (int j = 1; j < N; j += 1) { resultado +=
            rand / 3; rand -= j; }


    }

    retornar resultado;
}
```

C.1.3 Matmul e Misto

Os parâmetros dim1, dim2 e delay foram alterados entre os testes matmul e mixed para alcançar as propriedades desejadas.

```
    public long matmul() throws InterruptedException{
    Random rand = new Random(); long[]
    [] m = new long[dim1][dim2];

    para (int i = 0; i < dim1; ++i) {

        m[rand.nextInt(dim1)][rand.nextInt(dim1)] = System.currentTimeMillis();

        para (int j = 0; j < dim1; ++j) {
            m[i][j] = i * j; para (int k
            = 0; k < dim2; ++k) m[i][j] += a[i][k] * b[k][j];
        }
    }

    Thread.sleep(DELAY);

    retornar m[rand.nextInt(dim1)][rand.nextInt(dim1)];
}
```

C.2 Reativo

C.2.1 E/S

```
    public Mono<ServerResponse> io(ServerRequest request) {

    return


        ServerResponse.ok() .contentType(MediaType.APPLICATION_JSON) .body(BodyInserters.fromValue(("success
}
```

C.2.2 Calcular

```
    // função auxiliar para teste de computação privada
double calc_res() { double result = 0.0;
    double rand =
    System.currentTimeMillis(); int N = 1000; for (int i = 3; i < N;
    i += 1) {

        para (int j = 1; j < N; j += 1) { resultado +=
            rand / 3; rand -= j * i;

        }
    }

    retornar resultado;
```

```
}

// Método de teste de concorrência com uso intensivo
de CPU public Mono<ServerResponse> compute(ServerRequest request)
    { double result = calc_res();
    return ServerResponse.ok()

            .contentType(MediaType.APPLICATION_JSON) .body(BodyInserters.fromValue(result));
}
```

C.2.3 Matmul e Misto

Os parâmetros dim1, dim2 e d foram alterados entre os testes matmul e mixed para atingir as propriedades desejadas.

```
public Mono<ServerResponse> matmul(ServerRequest r) { long[][] m = new
    long[dim1][dim2];
    Random rand = novo Random();

    para (int i = 0; i < dim1; ++i) {
        m[rand.nextInt(dim1 - 1)][rand.nextInt(dim1 - 1)] = System.currentTimeMill for (int j = 0; j < dim1; ++j) {

            m[i][j] = i * j; para (int
            k = 0; k < dim2; ++k) m[i][j] += a[i][k] * b[k][j];
        }
    }
    return

            ServerResponse.ok() .contentType(MediaType.APPLICATION_JSON) .body(BodyInserters.fromValue(
            - 1)][rand.nextInt(dim .delayElement(d);
}
```

Instituições para Datavetenskap | Lunds Tekniska H¨ogskola | Redovisas 31 de maio de 2024

Trabalho de exame: Estudo de serviços reativos versus bloqueio regular versus threads virtuais para um serviço de alta carga

Alunos: Oliver Nederlund Persson, Elias Gustafsson

Handledare: Jonas Skeppstedt (LTH), Samuel Alberius (Sinch AB)
Examinador: Flavius Gruian

Compreendendo a programação assíncrona, bloqueie·o co+mércio e o comércio de virtude para o sistema de alta qualidade.

POPULARVETENSKAPLIG SAMMANFATTNING AV **Oliver Nederlund Persson, Elias Gustafsson**

JAVA AR ETT AV V ARLDENS MEST ANV PROGRAMAÇÃO ANDA SSPR AK SOM UTG OU BASEADO F OU ETT FLERTAL PO PROGRAMA DE DADOS PULARA , ATÉ EXEMPEL Minecraft OCH GLOBALA PLATTFORMAR SASOM . VIRTUELA Netflix OCH Spotify. 2023 INTRODUÇÕES JAVA 21 SOM INNEHALLER ETT NYTT KONCEPT F OU SPR AKET , VIRTUELLA TRADAR TRADAR AVSES KUNNA FORB ATTRA PROGRAMAS DIVERSOS F ORM AGA ATT HANTERA M ANGA ANV ANDARE PARALLELLT . DENNA STUDIE VISAR ATT VIRTUELLA TRADAR HAR GOD POTENCIAL PA ETT EFFEKTIVT S ATT .

Det ¨ar inte gÿangbart for mÿanga applicationer att han-tera e anv¨andare i taget. Ifall det até o exemplo tar 0,1 sekunder att skicka ett meddelande ¨pela internet para um anv¨andare, sÿa hade det kunnat ta en timme for mot-tagaren att fÿa meddelandet om 36 000 anv¨andare hade skickat samtidigt. Detta ¨ar helt ekelt inte acceptabelt para a maioria dos aplicativos e sÿaledes serão program-rare att allt detta skall ske parallellt, det vill s¨aga att tiden det tar f¨or anv¨andare att skicka ett meddelande i ovanstÿaende exempel ¨ar n¨ara 0.1 sekunder oberoende av antalet anv¨andare. Detta kallas para flertrÿadad programa-vara e det bygger pÿa programas f¨ormÿaga att anv¨anda hÿardvaruresurser pÿa ett effektivt s¨att. Det Finns Flertal técnico para att g¨ora detta, inom denna studye unders¨oker vi virtudella trÿadar, plattformstrÿadar e programação assíncrona.

Platformstrÿadar ¨ar kraftiga trÿadar como kan hantera flera uppgifter, mas de kan blockera hÿardvara n¨ar de v¨antar pÿa att en uppgift skall slutf¨oras. Programação asynkron l¨oser plattformstrÿadarnas problema genom att inte v¨anta pÿa att en uppgift skall vagabunda innan n¨asta uppgift pÿab¨orjas (eftersom de arbetar icke-sekventiellt). Virtuella trÿadar ¨ar l¨attviktiga trÿadar som ej blockerar datorer n¨ar de v¨antar pÿa att uppgifter skall l¨osas. Detta eftersom de monteras nedëoch sparas i minnet medan de inte kan arbeta.

Denna estudou testar esse genom de três técnicas para implementar tre olika servrar como drivers de tre oli-ka teknikerna. D¨arefter konstrueras fyra tester como teste de olika teknologierna e olika arbetsomrÿaden sÿasom operador ber¨akningstunga, operador minnestunga e framf¨or todo o operador de inicialização de E/S, d¨ar I/O in-neb¨ar att datorn tar emot e skickar ut data. Slutli-gen skickar e annan dator f¨orfrÿagningar to servrarna com vários cenários que simulam flera samtidiga anvandarare. Ett flertal steg togs para att s¨akerst¨alla en bra testmilj¨o, vilket ¨ar A ech O f¨or den h¨ar typen av studier.

Resultaten av desse tester visar attvirtualla trÿadar i de flesta fall erbjuder b¨ast prestanda (dvs ¨ar snabbare) ¨uma alternativa. D¨aremot anv¨ander en server som drivs av asynkron programação minst hÿardvaruresurser och har den l¨agsta felfrekvensen. Eu testei como in-neh¨oll nÿagra blockerande operador pré-estade de van-liga platformtrÿadarna b¨ast.

Implicações deste ¨ar att valet mellan att anv¨anda programação assíncrona ou virtudella trÿadar para um gerenciamento de servidor para uma vantagem de mellan prestanda, stabilitet e hÿardvarukostnader.