



CENTRO UNIVERSITÁRIO 7 DE SETEMBRO - UNI7
CURSO DE ESPECIALIZAÇÃO EM ARQUITETURA, DESIGN E
IMPLEMENTAÇÃO DE SISTEMAS PARA INTERNET

RAFAEL ELIAS RIBEIRO BARBOSA

ANÁLISE DE DESEMPENHO DE APLICAÇÕES JAVA UTILIZANDO
PROGRAMAÇÃO REATIVA SEM BLOQUEIO E TRADICIONAL COM BLOQUEIO

FORTALEZA - 2018

RAFAEL ELIAS RIBEIRO BARBOSA

**ANÁLISE DE DESEMPENHO DE APLICAÇÕES JAVA UTILIZANDO
PROGRAMAÇÃO REATIVA SEM BLOQUEIO E TRADICIONAL COM BLOQUEIO**

Monografia apresentada ao Centro Universitário 7 de Setembro como requisito parcial para obtenção do título de Especialista em Arquitetura, Design e Implementação de Sistemas para Internet.

Orientador: D.Sc. Marum Simão Filho

Fortaleza - 2018

**ANÁLISE DE DESEMPENHO DE APLICAÇÕES JAVA UTILIZANDO
PROGRAMAÇÃO REATIVA SEM BLOQUEIO E TRADICIONAL COM BLOQUEIO**

Monografia apresentada ao Centro Universitário 7 de
Setembro como requisito parcial para obtenção do título
de Especialista em Arquitetura, Design e Implementação
de Sistemas para Internet.

Rafael Elias Ribeiro Barbosa

Monografia apresentada em ____/____/____

Prof. Marum Simão Filho, D.Sc.
Orientador

1º Examinador: _____
Prof. André Jackson Gomes Bessa, M.Sc.

2º Examinador: _____
Prof. Vitor Almeida dos Santos, D.Sc.

Prof. Marum Simão Filho, D.Sc.
Coordenador do Curso

DEDICATÓRIA

Dedico este trabalho, em especial, a meus pais, irmão e minha esposa que me deram força e coragem para seguir em frente. Aos colegas de trabalho, pela amizade e apoio. E ao Professor Marum por seus ensinamentos, paciência e confiança ao longo das supervisões das minhas atividades.

RESUMO

Programação reativa está presente, de forma nativa, em cada vez mais linguagens de programação como JavaScript, Java, .NET, etc. Isso pode ser explicado pelos benefícios trazidos por esse paradigma, como ter a execução assíncrona e sem bloqueio. Para a utilização desse paradigma no Java era necessário importar bibliotecas e *frameworks* de terceiros como Project Reactor, Akka e Vert.x e RxJava. Entretanto a versão 9 do Java, já veio com suporte nativo a esse paradigma. Nesse mesmo sentido, o *framework* Spring também passou a adotar esse paradigma. E esse trabalho faz uma comparação, em relação a performance, entre duas aplicações feita utilizando Spring, uma na forma tradicional, com bloqueio, e outra utilizando programação reativa, assíncrona e sem bloqueio. Além disso, há também uma revisão bibliográfica a respeito de programação reativa.

Palavras-chaves: Programação reativa, Bloqueio, Assíncrono

ABSTRACT

Reactive programming is present, natively, in more and more programming languages like JavaScript, Java, .NET, etc. This can be explained by the benefits brought by this paradigm, such as having asynchronous and non-blocking execution. To use this paradigm in Java, you needed to import third-party libraries and frameworks, such as Project Reactor, Akka and Vert.x and RxJava. However, Java's version 9, already comes with native support for this paradigm. In this same sense, the Spring framework also started adopting this paradigm. This work makes a comparison of performance between two applications made using Spring, one in traditional way, blocking, and another using reactive programming, asynchronous and non-blocking. In addition, there is also a literature review about reactive programming.

Keywords: Reactive programming, Blocking, Asynchronous

LISTA DE FIGURAS

Figura 1 - Ciclo de vida de uma <i>Thread</i>	17
Figura 2 - Gráfico de dependência em programação reativa	18
Figura 3 - Características de Sistemas Reativos.....	20
Figura 4 - Tempo de resposta com 1000 registros e 100 usuários	32
Figura 5 - Tempo de resposta com 1000 registros e 1000 usuários	34
Figura 6 - Tempo de resposta com 1000 registros e 2000 usuários	35
Figura 7 - Tempo de resposta com 2000 registros e 100 usuários	36
Figura 8 - Tempo de resposta com 2000 registros e 1000 usuários	37
Figura 9 - Tempo de resposta com 2000 registros e 2000 usuários	38

LISTA DE TABELAS

Tabela 1 - Resultado com 1000 registros e 100 usuários	32
Tabela 2 - Resultado com 1000 registros e 1000 usuários	33
Tabela 3 - Resultado com 1000 registro e 2000 usuários	34
Tabela 4 - Resultado com 2000 registros e 100 usuários	36
Tabela 5 - Resultado com 2000 registros e 1000 usuários	37
Tabela 6 - Resultado com 2000 registros e 2000 usuários	38
Tabela 7 - Resumo dos resultados.....	39

LISTAGEM DE CÓDIGOS

Listagem 1 - <i>Interface Publisher</i>	21
Listagem 2 - <i>Interface Subscriber</i>	22
Listagem 3 - <i>Interface Subscription</i>	22
Listagem 4 - <i>Interface Processor</i>	23
Listagem 5 - <i>Interface</i> que representa a camada Repository	26
Listagem 6 - Classe que representa a camada Controller	27
Listagem 7 - <i>Interface</i> que representa a camada Repository	27
Listagem 8 - Classe que representa a camada Controller	27
Listagem 9 - Classe que representa a tabela Users do MySQL.....	28
Listagem 10 - Classe que representa o Document Users no MongoDB	29
Listagem 11 - Classe de simulação do Gatling	30

LISTA DE ABREVIATURAS E SIGLAS

IO	Input and Output
CPU	Central Processing Unit
JDK	Java Development Kit
TCK	Technology Compatibility Kit
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation

SUMÁRIO

1 INTRODUÇÃO	13
1.1 MOTIVAÇÃO	13
1.2 OBJETIVOS	14
1.2.1 Gerais	14
1.2.2 Especificos	14
1.3 MÉTODO DE PESQUISA	14
1.4 ESTRUTURA DO TRABALHO	15
2 FUNDAMENTAÇÃO TEÓRICA	16
2.1 PARADIGMA IMPERATIVO	16
2.2 THREAD.....	16
2.3 PROGRAMAÇÃO REATIVA	17
2.3.1 Modelos de Avaliação (<i>Evaluation model</i>)	18
2.4 MANIFESTO REATIVO.....	19
2.5 REACTIVE STREAMS	21
2.6 SPRING 5.....	23
2.6.1 WebFlux	23
2.7 PROJECT REACTOR	24
3 DESENVOLVIMENTO DAS APLICAÇÕES PARA COMPARAÇÃO	26
3.1 API.....	26
3.1.1 Tradicional	26
3.1.2 Reativo	27
3.2 BASE DE DADOS	28
3.2.1 Tradicional	28
3.2.2 Reativo	28
3.3 SERVIDOR DE APLICAÇÃO	28
3.4 DOCKER	29
3.5 TESTES	29
3.6 AMBIENTE DE SIMULAÇÃO	30
4 RESULTADOS E DISCUSSÃO	31
4.1 RESULTADO COM 1000 REGISTROS	31
4.1.1 Resultados com 100 usuários.....	31
4.1.2 Resultados com 1000 usuários.....	32

4.1.3 Resultados com 2000 usuários	34
4.2 RESULTADOS COM 2000 REGISTROS	35
4.2.1 Resultados com 100 usuários	35
4.2.2 Resultados com 1000 usuários	36
4.2.3 Resultados com 2000 usuários	38
4.3 DISCUSSÃO	39
5 CONCLUSÃO	40
5.1 TRABALHOS FUTUROS	41
6 REFERÊNCIAS	42

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

No ano de 1965, o cofundador da Intel, Gordon Moore fez uma previsão de que o número de componentes por circuito integrado iria dobrar a cada ano. Mais tarde, em 1975, o próprio Moore revisou sua previsão e verificou que o número de componentes dobrava a cada dois anos. Isso ficou conhecido como lei de Moore (MOORE, 2006). Moore também previu que esse crescimento iria reduzir o preço dos processadores.

Esse crescimento do poder computacional a um preço mais baixo permitiu a popularização dos computadores pessoais, *smartphones*, *tablets*. Acompanhando essa evolução na infraestrutura disponível, os programas computacionais também tiveram avanços e hoje temos aplicações que necessitam de vários servidores para serem executadas.

Contudo, mesmo com toda essa disponibilidade de infraestrutura, sempre há uma preocupação de utilizá-la da melhor forma. É nesse contexto que a programação reativa vem ganhando muita relevância nos últimos anos, principalmente, como uma forma de as aplicações otimizarem a utilização do *hardware*. Isso é mais evidente com a popularização dos *frameworks* Javascript, com suporte à programação reativa, como React.js e Angular (STACKOVERFLOW, 2018).

O interesse em produzir sistemas reativos levou um grupo de profissionais a criar um documento chamado *Reactive Manifesto*. Nesse documento, define-se que sistemas reativos são responsivos, resilientes, elásticos e dirigidos a eventos (BONÉR et al., 2018).

Esse crescimento também pode ser visto no lado do *back-end*. Em Java, por exemplo, já existem alguns *frameworks* reativos, como o Project Reactor, Akka e Vert.x. Há também o RxJava, que é uma biblioteca que fornece várias ferramentas reativas. Mas o que tornou mais evidente a importância da programação reativa com Java foi a inclusão desse paradigma tanto na versão 9 do Java Development Kit (JDK), com a Application Programming Interface (API) Flow (ORACLE, 2018), como na versão 5 do Spring Framework, com a API WebFlux (SPRING, 2018).

A vantagem que a programação reativa tem, para utilizar melhor a infraestrutura disponível, é que ela é assíncrona, sem bloqueio e, dessa forma, um processo que está sendo executado não fica bloqueado enquanto aguarda os resultados de outros processos (HOCHBERGS, 2017).

Nesse contexto, é inevitável o surgimento de questões sobre qual modelo de programação é mais performático, a programação reativa, sem bloqueio, ou a programação convencional, com bloqueio?

1.2 OBJETIVOS

1.2.1 Gerais

O objetivo geral deste trabalho é comparar dois paradigmas de programação em relação à performance: a programação reativa, que é assíncrona e sem bloqueio, com a programação tradicional, com bloqueio.

1.2.2 Específicos

Os objetivos específicos deste trabalho são:

- Discutir os conceitos de programação reativa;
- Fornecer exemplo de implementação de uma aplicação utilizando programação reativa, em Java; e
- Analisar a performance de aplicações utilizando programação reativa e programação tradicional, em Java.

1.3 MÉTODO DE PESQUISA

Inicialmente, para compor o referencial teórico, realizaremos uma revisão na literatura específica da área a respeito do tema em questão, ou seja, a programação reativa. Em seguida, serão desenvolvidas duas aplicações, uma adotando a programação tradicional e outra baseada na programação reativa. Por fim, será realizada uma comparação, com relação à performance, entre as duas aplicações, que irá considerar o tempo médio de resposta de cada requisição, em milissegundos, e a quantidade de requisições por segundo.

1.4 ESTRUTURA DO TRABALHO

O trabalho está organizado em 4 capítulos, além desta Introdução. O Capítulo 2, Fundamentação Teórica, introduz os conceitos necessários para compreender a programação reativa, o que o sistema precisa ter para ser reativo e aborda os princípios da especificação *Reactive Streams* para implementação de programação reativa no Java. No Capítulo 3, Desenvolvimento das Aplicações para Comparação, são apresentados os detalhes das implementações, o ambiente onde as aplicações foram executadas e a ferramenta utilizada para a aplicação dos testes. Já no Capítulo 4, Resultados e Discussão, são fornecidos todos os resultados obtidos nos testes e são feitas as comparações previstas. Por último, no Capítulo 5, Conclusão, são apresentadas as dificuldades para a realização do trabalho e as conclusões obtidas a partir dos resultados dos testes, além dos trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 PARADIGMA IMPERATIVO

Como explicado em Wegner (1990), o paradigma imperativo vê um programa como uma sequência de comandos que, progressivamente, transforma um estado inicial em um estado final. Eles são exemplificados pelo computador de von Neumann e a máquina de Turing. A linguagem Java utiliza paradigma imperativo orientado a objetos.

2.2 THREAD

De acordo com Rani e Dhingra (2014), uma *thread*, em Java, é uma unidade de trabalho despachável. Uma *thread* é um subconjunto de um processo. Um processo pode ser explicado como uma coleção de uma ou mais *threads* e os recursos de sistema associados. Por exemplo, se duas aplicações estão rodando em um computador, por exemplo, Paint e Outlook, dois processos são criados.

Ainda de acordo com Rani e Dhingra (2014), o ciclo de vida de uma *thread* é semelhante ao ciclo de vida de processos executando em um sistema operacional, e possui cinco estados: nova, executável, executando, não executável (bloqueada) e terminada.

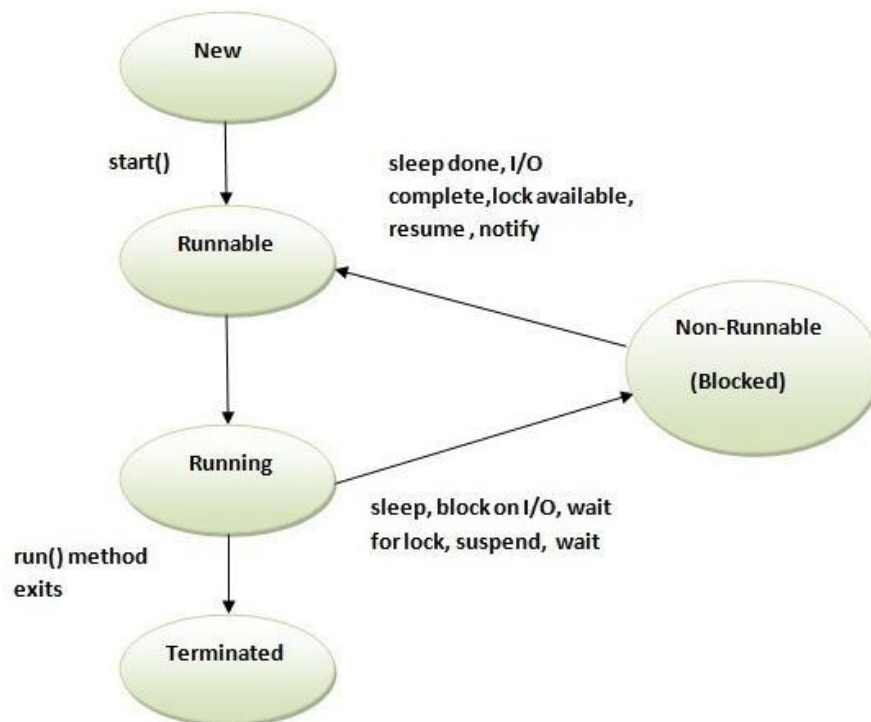
A passagem de um estado para outro se dá a partir de operações que são realizadas pela *thread* ou são realizadas nela, como se observa na Figura 1.

Uma explicação sobre os estados de uma *thread* é fornecida adiante:

- Nova: A *thread* está nesse estado quando ela é criada. Para criar uma *thread*, é necessário criar uma instância da classe *thread* ou uma instância de uma classe que é subclasse de *thread*.
- Executável: Nesse estado, a *thread* está apta para ser executada e está apenas aguardando o controle da CPU ser dado a ela.
- Executando: Nesse estado, a *thread* está no modo de execução; o controle da CPU foi dado a ela. Para isso ocorrer, o *scheduler* do sistema operacional selecionou uma *thread* específica do pool de *threads*.

- Bloqueada: Nesse estado, a *thread* não tem permissão de ser passada para os estados Executável e/ou Executando. Uma *thread* pode estar bloqueada porque pode estar suspensa, dormindo (*sleep*) ou aguardando.
- Terminada: A *thread* está nesse estado quando a execução do método `run` foi concluída, ou seja, o método retornou algo. Nesse estado, o ciclo de vida de uma *thread* termina.

Figura 1 - Ciclo de vida de uma *thread*



Fonte: Rani e Dhingra (2014)

2.3 PROGRAMAÇÃO REATIVA

De acordo com Hochbergs (2017) e Bainomugisha et al. (2013), programação reativa é um paradigma de programação que tem como base as ideias dos valores contínuos que variam no tempo (*time-varying values*) e na propagação de mudanças. Ela está incluída no paradigma de programação declarativo. A utilização desse paradigma facilita o desenvolvimento declarativo de aplicações *event-drive*, pois, segundo Bainomugisha et al. (2013), ela permite que os desenvolvedores expressem seus programas em termos de *o que fazer*, e, automaticamente, a linguagem gerencia o *quando fazer*. Essa abordagem é diferente do paradigma

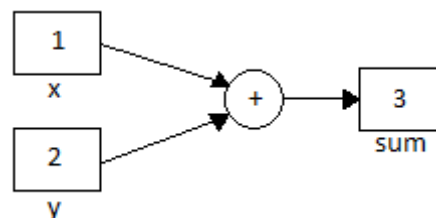
imperativo, onde o desenvolvedor altera o estado do programa através das instruções. Para explicar melhor essa diferença, vejamos o exemplo abaixo:

```
int x = 1;
int y = 2;
int sum = x + y;
```

O que esse exemplo mostra é uma simples soma na linguagem Java. Como o Java é uma linguagem imperativa, o programa vai ser executado em sequência. Assim, a variável `x` recebe o valor 1, depois a variável `y` recebe o valor 2 e, então, a variável `sum` recebe o valor 3. O valor da variável `sum` vai ser igual a 3 independente de alterações, posteriores, nas variáveis `x` ou `y`. Entretanto, no paradigma reativo, a variável `sum` ficará sempre dependendo dos valores de `x` e `y`, ou seja, a cada alteração nos valores de `x` ou `y`, o valor da variável `sum` também será atualizado. Na terminologia de programação reativa, diz-se que a variável `sum` é dependente das variáveis `x` e `y`.

Na Figura 2, temos o gráfico de dependência do programa anterior.

Figura 2 - Gráfico de dependência em programação reativa



Fonte: Bainomugisha et al. (2013)

Esse funcionamento é similar às fórmulas criadas em aplicativos de planilha, como Microsoft Excel, onde sempre que há uma alteração nos valores das variáveis que compõem a fórmula o resultado da mesma é executado novamente.

2.3.1 Modelos de Avaliação (*Evaluation model*)

Segundo Bainomugisha et al. (2013), em programação reativa, um modelo de avaliação descreve como as alterações são propagadas através do gráfico de dependência. Como já foi mencionado, a propagação dessas alterações é feita automaticamente, então, quando um valor é alterado todos os seus dependentes serão notificados. Nesse cenário, algumas questões vêm à tona. Quem será o

responsável pela propagação dessa alteração? O responsável pela alteração é quem notifica os seus dependentes ou os dependentes é que ficam responsáveis por perguntar se houve alguma alteração? Há dois modelos de avaliação em programação reativa, *push* e *pull* (HOCHBERGS, 2017), os quais serão descritos adiante.

No modelo *push*, quando há uma alteração em dado, ela é “empurrada” (*push*) para as computações que dependem dele. Esse modelo é similar ao padrão *publish-subscribe*. As linguagens que implementam esse modelo necessitam de uma solução eficiente para evitar computações desnecessárias, uma vez que computações serão feitas a cada alteração.

No modelo *pull*, pelo contrário, as computações que dependem de um dado precisam “puxá-lo” (*pull*) da fonte. Assim, o *puller* decide requisitar o novo dado quando for necessário. Uma crítica a esse modelo é a latência entre a ocorrência de um evento até a reação acontecer.

2.4 MANIFESTO REATIVO

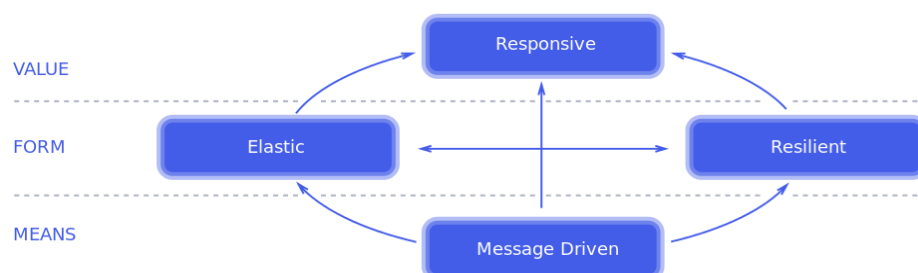
O manifesto reativo é um documento que teve sua versão 2 publicada em setembro de 2014. Ele tem o objetivo de propor uma arquitetura de software que seja capaz de atender as demandas dos sistemas atuais. Anteriormente, os sistemas tinham dezenas de servidores, segundos de tempo de resposta, horas fora do ar para manutenção e gigabytes de dados. Os sistemas atuais são executados tanto em dispositivos móveis como em *cluster* na nuvem, com milhares de processadores *multi-core*, os dados são medidos em Petabytes, os usuários esperam milissegundos de tempo de resposta e a aplicação estava sempre disponível (BONÉR et al., 2018). Assim, o autor acredita que é necessária uma arquitetura coerente para os sistemas feitos para atender as demandas atuais, onde os sistemas têm que ser resilientes, responsivos, elásticos e dirigido a mensagens (*event-driven*). Ele chama isso de sistemas reativos.

Sistemas reativos são flexíveis, têm baixo acoplamento e são escaláveis. Isso os torna fáceis de desenvolver e passíveis a mudanças. Eles são significativamente mais tolerantes a falhas e são altamente responsivos. A seguir, as características dos sistemas reativos são descritas:

- Responsivo: sistemas responsivos focam em prover tempos de resposta rápidos e consistentes. Responsividade é a base da usabilidade e utilidade de um sistema, mas, além disso, responsividade significa que os erros podem ser detectados rapidamente e tratados de forma eficaz. Isso passa mais confiança para o usuário, melhorando sua experiência ao usar o sistema.
- Resiliente: sistemas resilientes são capazes de se manter responsivos mesmo na ocorrência de uma falha. Para isso ser possível, é necessário ter replicação, contenção, isolamento e delegação. As falhas são contidas em cada componente, onde esse componente é isolado dos outros. A recuperação desse componente é delegada para um outro componente e a replicação é para manter a alta disponibilidade.
- Elástico: sistemas elásticos mantêm a responsividade quando submetidos a variações de carga. O sistema reage a essas variações aumentando ou diminuindo a alocação de recurso. Isso significa que esses sistemas são livres de gargalos.
- Dirigido a eventos: os sistemas reativos utilizam troca de mensagens assíncronas entre os componentes. Isso diminui o acoplamento, mantém o isolamento e a *location transparency* entre os componentes. *Location transparency* significa que, para o sistema, não importa se ele está executando um ou vários nós. Mensagens assíncronas permitem delegação de falhas, a elasticidade dos sistemas e o controle do fluxo das filas de mensagens, aplicando *back pressure* quando necessário. *Back pressure* tem seu funcionamento similar a uma torneira onde se consegue aumentar ou diminuir o fluxo de acordo com a necessidade.

A Figura 3 adiante ilustra as características de sistemas reativos e como elas se comunicam entre si.

Figura 3 - Características de Sistemas Reativos



Fonte: Bonér et al. (2018)

2.5 REACTIVE STREAMS

Reactive Streams é uma especificação que tem como objetivo prover um padrão para processamento de fluxos assíncronos com *back pressure non-blocking*. Isso engloba esforços direcionados a ambientes de execução (JVM e Javascript), assim como para protocolos de rede (CHRISTENSEN; MALDINI; KUHN, 2018).

O intuito dessa especificação é permitir a criação de implementações que estejam em conformidade com os princípios da programação reativa, preservando suas características e seus benefícios.

Segundo Christensen, Maldini e Kuhn (2018), o principal objetivo de *Reactive Streams* é gerenciar as trocas de fluxos de dados, de forma assíncrona, sem que o lado que recebe esse fluxo fique sobrecarregado com uma quantidade de dados maior do que ele pode manusear. A chave para isso está no *back pressure*.

Uma implementação da especificação *Reactive Streams* tem que estar certa de ser totalmente *non-blocking* e assíncrono, pois os benefícios de ter processamentos assíncronos podem ser anulados por uma comunicação síncrona do *back pressure*.

A especificação *Reactive Streams* é dividida em duas partes: a API e o TCK (*Technology Compatibility Kit*). A API especifica os tipos necessários para uma implementação de *Reactive Streams*. Já o TCK são testes padrões para aferir a conformidade das implementações (CHRISTENSEN; MALDINI; KUHN, 2018).

Na API de *Reactive Streams*, há quatro *interfaces*: *publisher*, *subscriber*, *subscription* e *processor*, as quais são descritas a seguir:

- *Publisher*: Um *publisher* é um provedor de um número ilimitado de elementos para seus *subscriber(s)*. Quem determina a velocidade da publicação desses dados é o *subscriber*. O único método dessa *interface* é o `subscribe`, que é responsável por cadastrar uma nova inscrição (*subscription*) para um *subscriber*. A Listagem 1 mostra a *interface publisher*.

Listagem 1 - Interface Publisher

```
1 public interface Publisher<T> {  
2     public void subscribe(Subscriber<? super T> s);  
3 }
```

Fonte: Christensen, Maldini e Kuhn (2018)

- *Subscribers*: *subscriber* é quem recebe os dados publicados por um *publisher*. Essa *interface* possui quatro métodos: `onSubscribe`, `onNext`, `onError` e `onComplete`. O método `onSubscribe` é sempre invocado como resposta à chamada do método `Publisher.subscribe(Subscriber)`. Para os dados serem emitidos do *Publisher* para um *subscriber*, é necessária a invocação do método `request` do *subscription*. Quando o dado é recebido pelo *subscriber* o método `onNext` é invocado para processar o dado. Se ocorrer algum erro no processamento do dado, o método `onError` será invocado para tratar o erro. O método `onComplete` é invocado quando não houver mais dados para ser publicado. A Listagem 2 mostra a *interface subscriber*.

Listagem 2 - Interface Subscriber

```
1 public interface Subscriber<T> {  
2     public void onSubscribe(Subscription s);  
3     public void onNext(T t);  
4     public void onError(Throwable t);  
5     public void onComplete();  
6 }  
7
```

Fonte: Christensen, Maldini e Kuhn (2018)

- *Subscription*: *subscription* representa a única relação entre um *subscriber* e um *publisher*. Essa *interface* possui dois métodos: `request` e `cancel`. Esses métodos só podem ser invocados dentro do contexto de um *subscriber*. Como já foi mencionado, nenhum dado será publicado enquanto o método `request` não for invocado. Já o método `cancel` é invocado para sinalizar que o *subscriber* não irá mais receber dados do *publisher*. A Listagem 3 mostra a *interface subscription*.

Listagem 3 - Interface Subscription

```
1 public interface Subscription {  
2     public void request(long n);  
3     public void cancel();  
4 }  
5
```

Fonte: Christensen, Maldini e Kuhn (2018)

- *Processor*: Um *processor* representa um estágio do processamento de um dado, no qual ele pode agir tanto como um *subscriber* quanto como um *publisher*. Para isso, ele tem que respeitar o contrato de ambos. Ou seja, quando *publisher*, ele só pode enviar dados para um *subscriber*, e quando *subscriber*, ele precisa se inscrever em um *publisher*. A Listagem 4 mostra a *interface processor*.

Listagem 4 - Interface Processor

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
2 }  
3
```

Fonte: Christensen, Maldini e Kuhn (2018)

A versão 9 do JDK possui a *interface* **java.util.concurrent.Flow** (ORACLE, 2018), que é semanticamente igual à especificação *Reactive Stream* (CHRISTENSEN; MALDINI; KUHN, 2018). A versão mais recente do Framework Spring, o Spring 5, também adota a especificação *Reactive Streams* (SPRING, 2018).

2.6 SPRING 5

O Spring é um dos *frameworks* mais utilizado em Java (HOTFRAMEWORKS, 2018), e sua versão mais recente, a versão 5, trouxe como uma das suas principais novidades a programação reativa, com sua *Reactive Stack*. Essa pilha (*stack*) são aplicações web baseadas na especificação *Reactive Streams* executando em servidores *non-blocking*, como o Netty.

2.6.1 WebFlux

O framework web reativo adicionado nesta versão é o WebFlux. Segundo a documentação Spring (2018), esse framework é totalmente *non-blocking*, com suporte a *Reactive Streams back pressure*, além de executar em servidores como Netty, Undertown e Servlet container 3.1+.

Há dois principais motivos para a criação do WebFlux. O primeiro é que a API Servlet 3.1 dá suporte a I/O *non-blocking*, entretanto, os contratos da API Servlet são síncronos ou *blocking*. Assim, fez-se necessária a criação de uma API que fosse capaz de executar em *runtime non-blocking*. O segundo motivo é a programação funcional, com a introdução dos *lambdas* no Java 8, que permite uma lógica assíncrona de forma declarativa.

O Spring WebFlux utiliza o Reactor como sua biblioteca reativa. Ele foi desenvolvido em colaboração com o Spring. O Reactor é uma biblioteca reativa baseada em *Reactive Streams*, portanto, todas as suas operações suportam *back pressure non blocking*.

2.7 PROJECT REACTOR

Como já foi mencionado, o Reactor é baseado na especificação *Reactive Streams*. O Reactor introduz tipos reativos compostos que implementam a *interface Publisher*, do *Reactive Streams*. Os mais notáveis são o **Flux** e o **Mono**. Além disso, fornecem uma rica opção de *operators*, onde cada *operator*, no Reactor, adiciona um comportamento ao *publisher* e encapsula o *publisher* da etapa anterior em uma nova instância.

Um objeto Flux representa uma sequência de 0...N itens reativos. Já um objeto Mono representa um valor simples ou vazio (0...1 item) (MALDINI; BASLÉ, 2018).

A diferença entre o uso dos dois tipos vai depender da semântica da informação que irá ser processada. Por exemplo, um HTTP *request* produz um único HTTP response, então, nesse caso, faz mais sentido utilizar um `Mono<HttpResponse>`, pois os *operators* disponíveis para esse caso fazem mais sentido em um contexto de zero ou um item. Uma breve descrição dos dois tipos é fornecida adiante:

- Flux: Um `Flux<T>` é um `Publisher<T>` padrão, que representa uma sequência assíncrona de 0...N itens emitidos, que pode, opcionalmente, ser terminado pela sinalização que o fluxo foi completado ou por um erro. Com isso, os possíveis valores de um Flux são: um valor, o sinal de completo ou

um erro. Esses possíveis valores representam os métodos `onNext`, `OnComplete` e `onError` da especificação *Reactive Streams*.

- **Mono:** um `Mono<T>` é um `Publisher<T>` especializado, que emite no máximo um item e, então, pode ser, opcionalmente, terminado por um sinal de que foi completado ou um erro. Ele possui apenas um subconjunto de *operators* de um `Flux`.

Uma característica importante de um `Mono` é que ele pode representar processos assíncronos sem valor, que tem apenas o conceito de completo, por exemplo, um `Runnable`. Para isso, basta criar um `Mono<void>`.

3 DESENVOLVIMENTO DAS APLICAÇÕES PARA COMPARAÇÃO

As implementações foram feitas utilizando o framework Spring, versão 5. Essa é a versão mais recente do framework e veio com suporte à programação reativa.

Para facilitar as implementações, foi utilizado o Spring Boot, versão 2. Nessa versão do Spring Boot, há facilidades tanto para programação tradicional como para programação reativa. Em ambas as implementações, foi utilizada a arquitetura em camadas: uma camada Controller, outra camada Repository e mais uma para o Domínio.

As aplicações têm, apenas, uma funcionalidade, que é retornar a lista, no formato JSON, com todos os usuários cadastrados no banco de dados. A lista é a mesma em ambas as aplicações.

3.1 API

A camada da API possui apenas um *endpoint*, o `getAllUsers`, que retorna todos os registros da tabela User.

3.1.1 Tradicional

Na aplicação síncrona, foram utilizados o Spring MVC e o Spring Data JPA. Nas Listagens 5 e 6 a seguir, são fornecidos os códigos das camadas Repository e Controller, respectivamente.

Listagem 5 - *Interface* que representa a camada Repository

```
1 public interface UserRepository extends CrudRepository<User, Integer> {  
2  
3 }
```

Fonte: Elaborado pelo autor

Listagem 6 - Classe que representa a camada Controller

```

1 @RestController
2 @RequestMapping("/users")
3 public class UserController {
4
5     @Autowired
6     private UserRepository userRepository;
7
8     @GetMapping
9     public @ResponseBody Iterable<User> getAllUser() {
10         return userRepository.findAll();
11     }
12 }

```

Fonte: Elaborado pelo autor

3.1.2 Reativo

Na aplicação reativa, foi utilizado o Spring WebFlux e Spring Data Mongo Reactive. Nas Listagens 7 e 8 a seguir, são fornecidos os códigos das camadas Repository e Controller, respectivamente.

Listagem 7 - Interface que representa a camada Repository

```

1 public interface UserRepository extends ReactiveMongoRepository<User, Integer> {}

```

Fonte: Elaborado pelo autor

Listagem 8 - Classe que representa a camada Controller

```

1 @RestController
2 @RequestMapping("/users")
3 public class UserController {
4
5     @Autowired
6     private UserRepository userRepository;
7
8     @GetMapping
9     public Flux<User> getAllUser() {
10         return userRepository.findAll();
11     }
12 }

```

Fonte: Elaborado pelo autor

3.2 BASE DE DADOS

Atualmente, não há, pelo menos de fornecedor oficial, driver reativo de conexão com o banco de dados MySQL. Por isso, para a aplicação reativa, foi utilizado o banco NoSQL MongoDB, que possui driver reativo.

Em ambos os bancos, há apenas uma tabela, coleção no caso do MongoDB, que é a tabela User com três colunas: id, name e age.

3.2.1 Tradicional

O banco de dados para a aplicação síncrona foi o MySQL. Abaixo, na Listagem 9, segue o código da Entity que representa a tabela Users.

Listagem 9 - Classe que representa a tabela Users do MySQL

```
1 @Data
2 @Entity(name="Users")
3 public class User implements Serializable {
4
5     private static final long serialVersionUID = 1L;
6
7     @Id
8     @GeneratedValue(strategy=GenerationType.AUTO)
9     private Long id;
10    private String name;
11    private Integer age;
12
13 }
```

Fonte: Elaborado pelo autor

3.2.2 Reativo

Como mencionado, o banco de dados para a aplicação reativa foi o MongoDB. A Listagem 10 adiante exibe o Document que representa a coleção Users.

3.3 SERVIDOR DE APLICAÇÃO

Como ambas as aplicações utilizam o modelo cliente-servidor, é necessário um servidor para executá-las. As aplicações que utilizam o Spring Boot vêm com um

servidor padrão. No caso da aplicação síncrona, onde é utilizado o Spring MVC, o servidor é o Apache Tomcat. Já na aplicação reativa, onde é utilizado o Spring WebFlux, o servidor é o Netty.

Listagem 10 - Classe que representa o Document Users no MongoDB

```
1 @Data
2 @Document(collection = "Users")
3 public class User implements Serializable {
4
5     private static final long serialVersionUID = 1L;
6
7     @Id
8     private Long id;
9     private String name;
10    private Integer age;
11
12 }
```

Fonte: Elaborado pelo autor

3.4 DOCKER

Para simular o mesmo ambiente, foram utilizados containers Dockers com a mesma configuração para ambas as aplicações. Os bancos de dados também executaram em container Docker.

3.5 TESTES

O objetivo deste trabalho é aferir a performance de cada aplicação com relação ao tempo de resposta. E para isso foi utilizado testes de estresse com a ferramenta Gatling.

Os testes simulam um determinado número de usuários acessando as aplicações e então é medido o tempo de resposta da aplicação com essa carga de usuários. Após os testes foi comparado a média aritmética dos tempos de resposta.

Na Listagem 11 tem um exemplo de configuração do Gatling utilizado nos testes.

3.6 AMBIENTE DE SIMULAÇÃO

O computador utilizado para realizar os testes tem um processador Intel Core i7-4790 com 3.60GHz. E a configuração do Docker é a seguinte:

- Sistema operacional: Boot2Docker 18.06.0-ce
- Tipo do sistema operacional: Linux
- Arquitetura: x86_64
- CPUs: 4
- Memória: 1.955GiB

Listagem 11 - Classe de simulação do Gatling

```

1 class ServiceSimulation extends Simulation {
2   var name="async"
3   var users=1
4   var rampupTime=30 seconds
5   var delay=""
6   var repeatTimes=1
7   var url="http://localhost:8080"
8
9   def startup() {
10    val httpProtocol = http.baseURL(url)
11    val page = repeat(repeatTimes, "n") {
12      exec(http(name).get("/users/" + delay))
13    }
14    val scn = scenario("page").exec(page)
15    setUp(scn.inject(rampUsers(users).over(rampupTime))).protocols(httpProtocol)
16  }
17 }
18
19 class AsyncSimulation extends ServiceSimulation {
20   name="async"
21   url="http://192.168.99.100:8080"
22 }
23
24 class SyncSimulation extends ServiceSimulation {
25   name="sync"
26   url="http://192.168.99.100:8080"
27 }
28
29 class Async100u extends AsyncSimulation {
30   users=100
31   repeatTimes=100
32   startup()
33 }
34
35 class Sync100u extends SyncSimulation {
36   users=100
37   repeatTimes=100
38   startup()
39 }

```

Fonte: Elaborado pelo autor

4 RESULTADOS E DISCUSSÃO

Os resultados dos testes foram separados em 2 etapas, considerando o volume de registros:

- A primeira etapa contou com 1000 registros;
- Na segunda etapa, foram 2000 registros.

Em cada uma das etapas, foram realizadas três cargas de testes, simulando quantidades diferentes de usuários acessando a aplicação, conforme descrito a seguir:

- O primeiro teste rodou com 100 usuários;
- No segundo teste, foram 1000 usuários;
- E, por último, foram 2000 usuários.

Nas próximas subseções, são fornecidos os resultados obtidos em cada um dos testes.

4.1 RESULTADO COM 1000 REGISTROS

Nessa primeira etapa, os testes foram feitos com um volume de dados de 1000 registros. Cada um dos testes adiante foi repetido 100 vezes.

4.1.1 Resultados com 100 usuários

O primeiro teste, com 1000 registro, foi feito com 100 usuários acessando cada aplicação. Os resultados estão na Tabela 1.

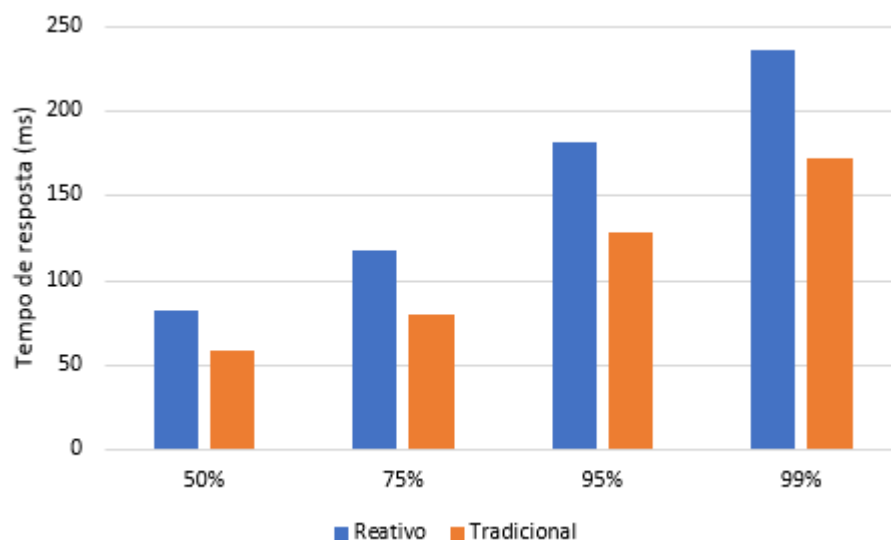
O gráfico da Figura 4 adiante mostra a comparação entre os tempos de resposta das duas aplicações com um volume de 1000 registros e 100 acessos simultâneos. É possível verificar que a aplicação reativa teve um desempenho um pouco pior do que a aplicação tradicional, tanto no tempo de resposta como na média de requisições por segundo.

Tabela 1 - Resultado com 1000 registros e 100 usuários

1000 registros - 100 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	9	8
50%	82	58
75%	118	80
95%	182	129
99%	237	173
Máximo	366	298
Média	90	64
Desvio Padrão	49	34
Requisições por segundo		
Média Req/s	270.27	285.714

Fonte: Elaborado pelo autor

Figura 4 - Tempo de resposta com 1000 registros e 100 usuários



Fonte: Elaborado pelo autor

4.1.2 Resultados com 1000 usuários

O segundo teste, com 1000 registros, foi feito com 1000 usuários acessando cada aplicação. Os resultados estão na Tabela 2.

Na aplicação tradicional, algumas requisições resultaram em erro de *timeout*: **j.u.c.TimeoutException: Request timeout to /192.168.99.100:8080 after 60000 ms**. Um motivo para esse problema pode ter sido a não limitação do número de threads do Tomcat, assim, toda requisição que chegava ao servidor era colocada em

uma nova *thread*. Com isso, algumas requisições ficaram muito tempo esperando por uma conexão com o banco de dados.

Já na aplicação reativa ocorreu o seguinte erro: **Too many threads are already waiting for a connection. Max number of threads (maxWaitQueueSize) of 500 has been exceeded.** Esse erro é devido à configuração de duas variáveis de configuração da biblioteca de conexão com o MongoDB: *maxPoolSize* e *waitQueueMultiple*. Como a configuração padrão é 50 e 10, respectivamente, têm-se, no máximo, 500 threads aguardando conexão. Esse número, ou seja, 500 aguardando conexão mais 50 conectados, não é o suficiente já que, nesse teste, são 1000 usuários acessando a aplicação, simultaneamente. Assim, a configuração utilizada foi 20 e 100, respectivamente.

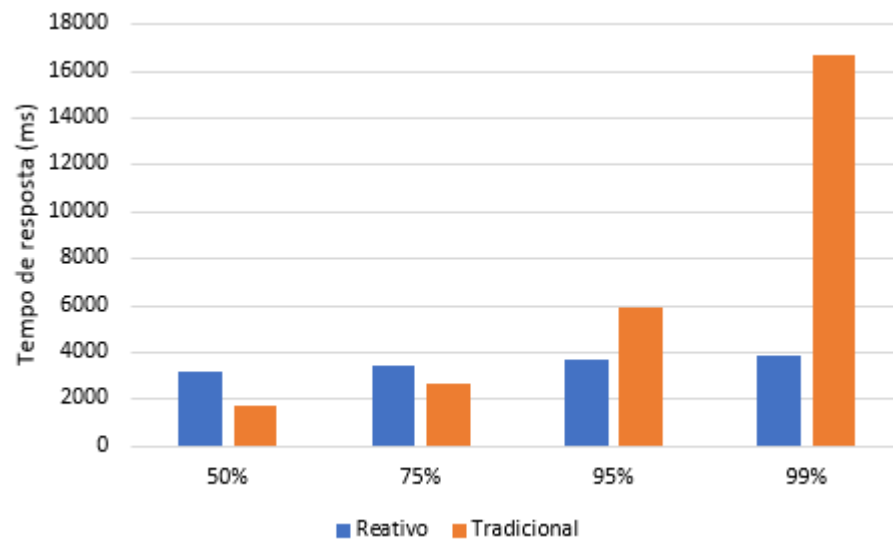
O gráfico da Figura 5 mostra o resultado comparativo desse segundo teste, com 1000 usuários acessando as aplicações. Na aplicação tradicional, foram consideradas apenas as requisições que tiveram sucesso. Pode-se constatar um desempenho melhor da aplicação reativa com relação ao tempo de resposta, se forem analisados 95% das requisições que obtiveram sucesso. Porém, considerando o número de requisições por segundo, a aplicação tradicional saiu-se melhor.

Tabela 2 - Resultado com 1000 registros e 1000 usuários

1000 registros - 1000 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	13	9
50%	3184	1759
75%	3434	2680
95%	3699	5961
99%	3830	16657
Máximo	4180	59768
Média	2984	2465
Desvio Padrão	782	3353
Requisições por segundo		
Média Req/s	304.878	324.967

Fonte: Elaborado pelo autor

Figura 5 - Tempo de resposta com 1000 registros e 1000 usuários



Fonte: Elaborado pelo autor

4.1.3 Resultados com 2000 usuários

O último teste, dessa vez com 1000 registros, foi feito com 2000 usuários acessando as aplicações. Os resultados são fornecidos na Tabela 3.

Tabela 3 - Resultado com 1000 registro e 2000 usuários

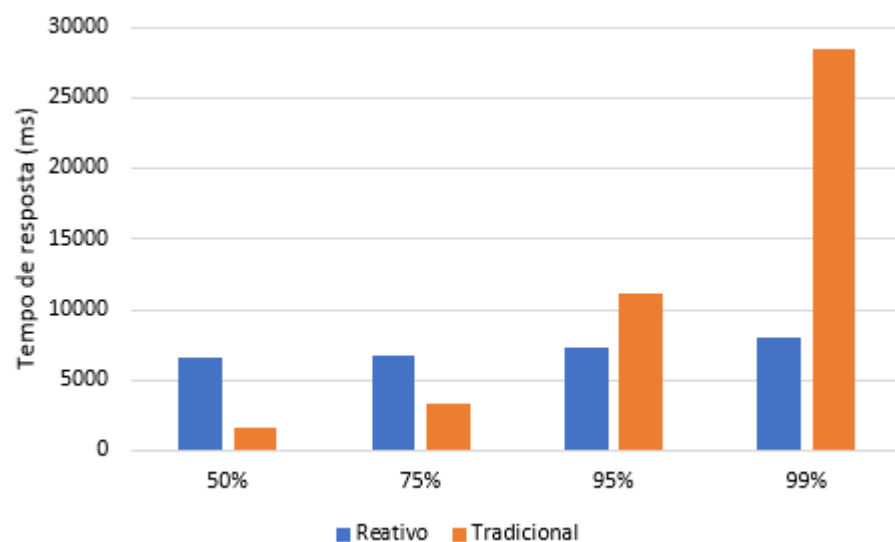
1000 registros - 2000 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	17	9
50%	6614	1556
75%	6754	3345
95%	7263	11145
99%	8080	28493
Máximo	8667	59990
Média	6322	3231
Desvio Padrão	1279	5126
Requisições por segundo		
Média Req/s	300.752	300.48

Fonte: Elaborado pelo autor

Para esse teste, na aplicação tradicional, ocorreu o mesmo erro de *timeout* do teste anterior. Também foram consideradas, para fins de comparação, apenas as requisições que obtiveram sucesso.

Já o problema com o número de threads, verificado na aplicação reativa, não ocorreu nesse caso uma vez que foram utilizadas as configurações do teste anterior. Os resultados, apresentados no gráfico da Figura 6, foram muito parecidos com os do teste anterior, onde a aplicação reativa teve um resultado melhor no tempo de resposta, se for analisado 95% das requisições que obtiveram sucesso. Considerando o número de requisições por segundo, o resultado foi igual para ambas.

Figura 6 - Tempo de resposta com 1000 registros e 2000 usuários



Fonte: Elaborado pelo autor

4.2 RESULTADOS COM 2000 REGISTROS

O objetivo deste teste foi observar o comportamento da aplicação com um volume maior de dados, considerando o dobro do volume dos testes anteriores. Cada um dos testes abaixo também foi repetido 100 vezes.

4.2.1 Resultados com 100 usuários

O primeiro teste com esse volume maior foi feito com 100 usuários acessando as aplicações. Os resultados estão na Tabela 4.

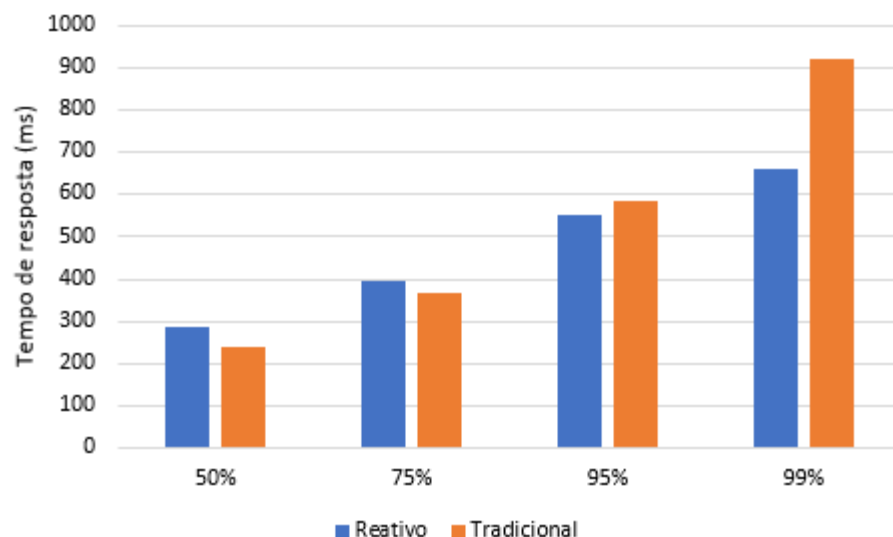
Tabela 4 - Resultado com 2000 registros e 100 usuários

2000 registros - 100 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	17	13
50%	285	239
75%	394	368
95%	554	584
99%	661	921
Máximo	921	1923
Média	294	271
Desvio Padrão	149	187
Requisições por segundo		
Média Req/s	175.439	185.185

Fonte: Elaborado pelo autor

Os números mostram que as aplicações tiveram resultado similar no tempo de resposta, se forem observados 95% das requisições que obtiveram sucesso, como é mostrado no gráfico da Figura 7. Já com relação ao número de requisições por segundo, a aplicação tradicional obteve um resultado melhor.

Figura 7 - Tempo de resposta com 2000 registros e 100 usuários



Fonte: Elaborado pelo autor

4.2.2 Resultados com 1000 usuários

O segundo teste também foi realizado com 1000 usuários e os resultados são exibidos na Tabela 5.

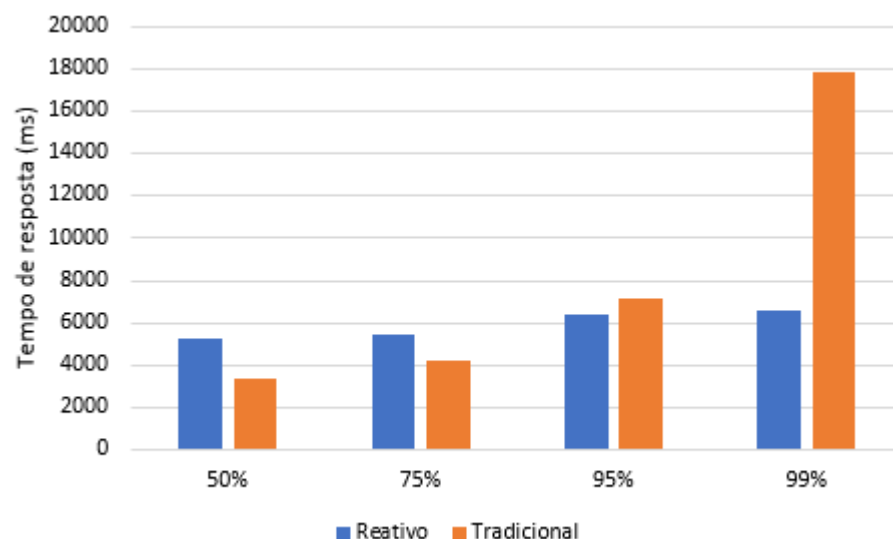
Tabela 5 - Resultado com 2000 registros e 1000 usuários

2000 registros - 1000 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	26	11
50%	5243	3327
75%	5476	4177
95%	6422	7145
99%	6618	17905
Máximo	10681	59923
Média	5074	3665
Desvio Padrão	1031	3490
Requisições por segundo		
Média Req/s	185.529	216.412

Fonte: Elaborado pelo autor

Nesse teste, também ocorreram os erros de *timeout* na aplicação tradicional, pelos mesmos motivos já explanados anteriormente. Como se observa no gráfico da Figura 8, a aplicação reativa teve um resultado melhor no tempo de resposta, se considerados 95% das requisições com sucesso. Já no número de requisições por segundo, a aplicação tradicional obteve um desempenho melhor.

Figura 8 - Tempo de resposta com 2000 registros e 1000 usuários



Fonte: Elaborado pelo autor

4.2.3 Resultados com 2000 usuários

Esse último teste foi realizado com 2000 usuários acessando as aplicações. Os resultados podem ser vistos na Tabela 6.

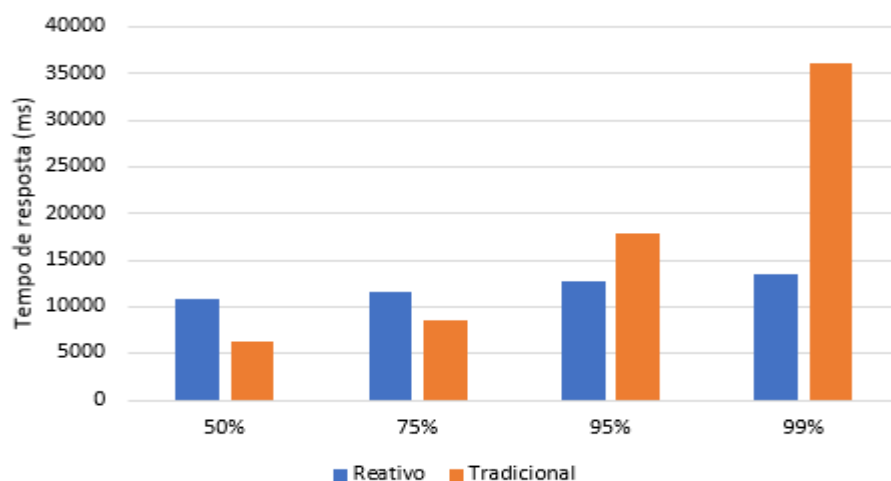
Tabela 6 - Resultado com 2000 registros e 2000 usuários

2000 registros - 2000 usuários		
	Reativo	Tradicional
Tempo de resposta (ms)		
Mínimo	23	13
50%	10870	6298
75%	11543	8563
95%	12689	17986
99%	13484	36176
Máximo	29769	63622
Média	10746	7506
Desvio Padrão	1824	6235
Requisições por segundo		
Média Req/s	180.18	178.171

Fonte: Elaborado pelo autor

O erro de *timeout* na aplicação tradicional também ocorreu nesse teste. E como pode ser visto no gráfico abaixo, a aplicação reativa teve um desempenho melhor no tempo de resposta, se considerados 95% das requisições, e teve um resultado melhor no número de requisições por segundo.

Figura 9 - Tempo de resposta com 2000 registros e 2000 usuários



Fonte: Elaborado pelo autor

4.3 DISCUSSÃO













Antes de discutir sobre os resultados, é importante ressaltar que tais resultados valem para o ambiente no qual eles foram executados. Uma variação desse ambiente de execução pode levar a achados diferentes.

É importante notar também que a pilha de execução das aplicações foi diferente, com relação aos servidores, Tomcat e Netty, e aos bancos de dados, MySQL e MongoDB, que podem ter influência no resultado. Porém, como essa é a configuração padrão do Spring Boot, a mesma foi mantida.

No caso dos servidores, até poderia ter sido utilizado o Tomcat para ambas as aplicações, já que o Tomcat mais recente dá suporte a aplicações reativas, sem bloqueio. Entretanto, o MySQL não poderia ter sido utilizado para as duas aplicações já que não há *driver* reativo, pelo menos de fornecedores oficiais, para bancos relacionais.

A Tabela 7 mostra um resumo dos resultados dos testes discutidos acima. É possível notar que a aplicação tradicional tem um desempenho bom, em relação à aplicação reativa, quando o número de usuários é menor. Com o aumento do número de usuários, a aplicação reativa passa a ter um desempenho igual ou melhor ao da aplicação tradicional.

Tabela 7 - Resumo dos resultados

Critério	Aplicação tradicional	Aplicação reativa
Com 1000 registros		
100 usuários		
1000 usuários		
2000 usuários		
Com 2000 registros		
100 usuários		
1000 usuários		
2000 usuários		

Fonte: Elaborado pelo autor

5 CONCLUSÃO

Programação reativa é um paradigma de programação declarativo e se baseia nos conceitos das *time-varying values* e na propagação de mudanças. Por ser declarativo, as expressões só precisam identificar o *que fazer*, já o *quando fazer* fica como responsabilidade da linguagem, diferentemente do paradigma imperativo, onde ambas as responsabilidades ficam com o desenvolvedor. Esse tipo de programação reativa tem se tornado cada vez mais comum nas aplicações disponíveis no mercado.

Diante desse cenário, esse trabalho teve como objetivo comparar dois paradigmas de programação quanto à performance: a programação reativa, que é assíncrona e sem bloqueio, com a programação tradicional, com bloqueio.

O Manifesto Reativo afirma que uma arquitetura reativa também deve envolver questões importantes, como escalabilidade e resiliência, entre outros. A comparação da performance levou em consideração somente o tempo de resposta das aplicações, não sendo analisadas outras questões, como consumo de memória, utilização da CPU e utilização da rede.

Com base nos resultados dos tempos de resposta, pode-se observar que a aplicação tradicional teve um resultado melhor do que a reativa, para o caso de menos usuários acessando a aplicação, como é visto nos resultados com 100 usuários. Aumentando o número de usuários, inicialmente para 1000 e depois para 2000 usuários, a aplicação reativa teve um desempenho melhor que a tradicional, observando 95% das requisições que obtiveram sucesso.

Ainda com base nos resultados, é possível ver que o número de requisições por segundo seguiu um padrão parecido com o do tempo de resposta, já que, com menos usuários, a aplicação tradicional teve um resultado melhor. Mas, ao se aumentar o número de usuários, a aplicação reativa a superou. O aumento da quantidade de registros pouco alterou os resultados e o padrão observado para 1000 registros foi similar ao padrão com 2000 registros.

É importante citar que, com o aumento da quantidade de usuários, a aplicação tradicional apresentou problemas de *timeout*. Uma possível explicação para isso está no número de threads criadas pelo servidor, que não é suportado pelo banco de dados, assim, algumas requisições ficaram muito tempo aguardando uma conexão, o que disparou o erro de *timeout*. Há, no Spring Boot, uma configuração

para limitar o número de threads do Tomcat: **server.tomcat.max-threads**. Porém, essa configuração não foi utilizada neste trabalho.

Outro problema que se verificou nos testes foi em relação ao *pool* de conexões e à fila de threads aguardando conexão, na aplicação reativa. Foi escolhido o valor 20 para o *pool* de conexões e 2000 para a fila. Esses valores foram escolhidos para poder suportar o ambiente de testes e alterações neles podem melhorar ou piorar o desempenho da aplicação. Da mesma forma, é possível alterar as configurações do *pool* de conexões da aplicação tradicional, mas não foi objetivo deste trabalho analisar quais as melhores configurações para cada aplicação.

5.1 TRABALHOS FUTUROS

Como trabalhos futuros, sugerimos os seguintes:

- Realizar outras comparações, dessa vez, considerando aspectos como o consumo de memória, a utilização da CPU e a utilização da rede;
- Realizar as mesmas comparações, mas com o mesmo banco de dados e mesmo servidor, para ambas as aplicações;
- Verificar o desempenho em relação a aplicações feitas em outras linguagens que também têm suporte à programação reativa, como Javascript.

6 REFERÊNCIAS

WEGNER, Peter. Concepts and Paradigms of Object-Oriented Programming. **Acm Sigplan OOps Messenger**. Nova Iorque, p. 7-87. ago. 1990.

RANI, Sangeeta; DHINGRA, Neetu. Java Special Feature: Multithreading. **International Journal Of Current Engineering And Technology**. Bahadurgarh, p. 4034-4037. dez. 2014.

HOCHBERGS, Gustav. **Reactive programming and its effect on performance and the development process**. 2017. 66 f. Dissertação (Mestrado) - Curso de Computer Science, Lund University, Lund, 2017.

BAINOMUGISHA, Engineer et al. A survey on reactive programming. **Acm Computing Surveys**. Nova Iorque, p. 52:1-52:35. ago. 2013.

BONÉR, Jonas et al. **The Reactive Manifesto**. Disponível em: <<https://www.reactivemanifesto.org/>>. Acesso em: 26 ago. 2018.

CHRISTENSEN, Ben; MALDINI, Stephane; KUHN, Roland. **Reactive Streams**. Disponível em: <<http://www.reactive-streams.org/>>. Acesso em: 26 ago. 2018.

ORACLE. **Class Flow**. Disponível em: <<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html>>. Acesso em: 26 ago. 2018.

SPRING. **Web on Reactive Stack**. Disponível em: <<https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#spring-webflux>>. Acesso em: 26 ago. 2018.

HOTFRAMEWORKS. **Java**. Disponível em: <<https://hotframeworks.com/languages/java>>. Acesso em: 26 ago. 2018.

MALDINI, Stephane; BASLÉ, Simon. **Reactor 3 Reference Guide**. Disponível em: <<https://projectreactor.io/docs/core/release/reference/>>. Acesso em: 26 ago. 2018.

MOORE, Gordon E.. Chapter 7: Moore's law at 40. In: BROCK, David C.. **Understanding Moore's Law: Four Decades of Innovation**. Chemical Heritage Foundation, 2006. p. 67-84.

STACKOVERFLOW. **Most Popular Technologies**. Disponível em: <<https://insights.stackoverflow.com/survey/2018/#most-popular-technologies>>. Acesso em: 26 ago. 2018.