CENTRO UNIVERSITÁRIO UNIVATES
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE ENGENHARIA DA COMPUTAÇÃO

UM ESTUDO COMPARATIVO ENTRE MODELOS DE CONCORRÊNCIA COM ÊNFASE EM ESCALABILIDADE PARA APLICAÇÕES WEB

Mauricio Colognese Concatto

Lajeado, novembro de 2016

Mauricio Colognese Concatto

UM ESTUDO COMPARATIVO ENTRE MODELOS DE CONCORRÊNCIA COM ÊNFASE EM ESCALABILIDADE PARA APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado ao Centro de Ciências Exatas e Tecnológicas do Centro Universitário UNIVATES, como parte dos requisitos para a obtenção do título de bacharel em Engenharia da Computação.

Orientador: Pablo Dall'Oglio

Lajeado, novembro de 2016

Mauricio Colognese Concatto

UM ESTUDO COMPARATIVO ENTRE MODELOS DE CONCORRÊNCIA COM ÊNFASE EM ESCALABILIDADE PARA APLICAÇÕES WEB

Este trabalho foi julgado adequado para a obtenção do título de bacharel em Engenharia da Computação do CETEC e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Banca Examinadora:

Prof. Pablo Dall'Oglio, UNIVATES - Orientador

Mestre pela Universidade do Vale do Rio dos Sinos – São Leopoldo, Brasil

Prof. Edson Funke, UNIVATES

Mestre pela Universidade Federal de Santa Maria – Santa Maria, Brasil

Prof. Willian Valmorbida, UNIVATES

Mestre pela Universidade do Vale do Rio dos Sinos – São Leopoldo, Brasil

Lajeado, novembro de 2016

RESUMO

Com a ascensão da internet, dos dispositivos móveis e a crescente demanda por aplicativos Web cada vez mais dinâmicos, os serviços que são disponibilizados na internet devem estar preparados para suportar um alto grau de acessos concorrentes sem enfrentar problemas como lentidão e instabilidade. A construção de aplicativos que suportem este alto grau de concorrência nem sempre é trivial, devido a isso nos últimos anos diversas abstrações de alto nível para implementar sistemas concorrentes vem ganhando destaque, como por exemplo o modelo de atores. Estas soluções são implementadas utilizando arquiteturas *multithread*, baseadas em eventos ou alguma solução hibrida. Entretanto, a área carece de estudos aprofundados que analisem o desempenho da utilização destas técnicas quando aplicadas nas diversas camadas de uma aplicação, como serviços e acesso a dados. Este trabalho visa realizar um estudo comparativo detalhado entre a utilização destes modelos nas várias camadas das aplicações Web, através da execução de testes de carga em um conjunto de cenários com diferentes requisitos de performance e concorrência.

Palavras-chave: Software, Concorrência, Web, Escalabilidade, Computação.

ABSTRACT

With the rise of the internet, mobile devices and the growing demand for increasingly dynamic Web applications, the services that are available on the Internet should be prepared to sustain a high degree of concurrent access without facing problems such as slowdowns and instability. Building applications that support this high level of concurrency is not always easy, because of that in recent years several high-level abstractions to implement concurrent systems is gaining prominence, such as the actor model. These solutions are implemented using multithread architecture, event-driven architecture or some hybrid solution. However, the area lacks depth studies to analyze the performance of the use of these techniques when applied in several layers of an application, such as services and data access. This study aims at a detailed comparative study of the use of models in the various layers of Web applications by running load tests on a set of scenarios with different performance and concurrency requirements.

Keywords: Software, Concurrency, Web, Scalability, Computing.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ACID	Atomicidade, consistência, isolamento e durabilidade
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CASE	Computer-aided software engineering
CRUD	Create, Read, Update and Delete
DMA	Direct Memory Access
DSL	Domain-specific Language
E/S	Entrada/Saída
GC	Garbage Collector
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAAS	Infrastructure as a service
JDBC	Java Database Connectivity
JIT	Just In Time Compiler
JMX	Java Management Extensions
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
IPC	Inter-process Communication
MVC	Model-View-Controller
ORDBMS	Object-Relational Database Management System
PAAS	Platform as a Service
RDBMS	Relational Database Management System
REST	Representational State Transfer

SAAS	Software as a Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SWEBOK	Software Engineering Body of Knowledge
TI	Tecnologia da informação
URI	Uniform Resource Identifier
WSDL	Web Services Description Language
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas vivenciamos uma grande transformação em todos os setores através da evolução da ciência e da tecnologia. Isto permitiu a criação de ferramentas de grande valia para a vida moderna. O constante avanço nas telecomunicações através do uso de fibras ópticas, satélites e rádios, junto com a popularização dos computadores pessoais acarretou a redução de custos no acesso à internet, e com isso, cada vez mais pessoas estão conectadas ao redor do mundo trocando informações.

Este processo inicia-se nos anos 1990 no Brasil, inicialmente restrito ao meio acadêmico e científico. Mais tarde em 1995, foi lançada a operação comercial no país, porém foi em 1996 que a internet começou a se expandir no pais, impulsionada pelos lançamentos de grandes portais e provedores de conexão. Nos primórdios, a internet era utilizada principalmente para disponibilização de conteúdo estático conectado através de *hyperlinks* (o que cunhou o termo *World Wide Web)* e troca de mensagens. Os primeiros navegadores eram bastante limitados, não possuíam nenhuma linguagem de programação embutida, e permitiam apenas alguns recursos multimídia, como imagens por exemplo. Este período posteriormente ganhou o nome de Web 1.0.

O termo Web 2.0 surgiu em meados de 1999 e teve uma maior visibilidade entre os anos 2002 e 2004. Este termo representa uma transição da internet em termos tecnológicos e finalidade de uso, partindo de uma natureza estática para algo dinâmico. A internet passa a ser vista como uma plataforma tecnológica e social, enfatizando usabilidade, interoperabilidade entre os sistemas, interação entre os usuários e foco no conteúdo gerado pelos mesmos. Termos como computação em nuvem, redes sociais, *Web Services*, *Software as a Service* (SAAS), *Asynchronous JavaScript and XML* (AJAX), dentre outros são usados em conjunto para

representar as inúmeras aplicações que se tornaram possíveis devido a evolução dos padrões da Web, tecnologias do lado do servidor e dos navegadores.

A computação em nuvem foi um termo muito representativo na transição para a Web 2.0. Aplicações que antes eram vistas isoladas em silos dentro das organizações, agora são repensadas para serem disponibilizadas nesta nova plataforma. Esta mudança de paradigma teve um impacto significante em um bom número destas aplicações, sendo que muitas tiveram de sofrer adaptações, algumas até serem reescritas (devido a alterações arquiteturais profundas), pois não estavam prontas para os altos níveis de concorrência e escalabilidade necessárias para a computação em nuvem.

Até a poucos anos atrás era normal a construção de aplicações que eram executadas em um único computador, utilizando um único processador. Muitas vezes, quando a aplicação não estava obtendo a performance desejada, os desenvolvedores aguardavam os processadores ficarem mais rápidos (através do aumento da velocidade de *clock*) ao invés de realizar otimizações no código do aplicativo (ROESTENBURG, BAKKER, & WILLIAMS, 2015). Já em 2005, Herb Sutter escreveu um artigo para o *Dr. Dobb's Journal* no qual cita que a velocidade de *clock* dos processadores estão chegando no limite, e comenta sobre a necessidade de codificar as aplicações de forma a aproveitar os recursos oferecidos pelos processadores recentes, como por exemplo os múltiplos núcleos, pois o único meio de obter a escalabilidade necessária é através da concorrência (SUTTER, 2005).

Hoje o acesso à internet é realizado de forma ininterrupta. Os componentes como microprocessadores, evoluem de forma rápida, tornando-se cada vez menores, mais rápidos e energeticamente eficientes. O melhor desempenho do hardware (menor tamanho e custo) impulsiona o aparecimento de aplicativos mais sofisticados e uma ampla gama de dispositivos inteligentes (como por exemplo os *smartphones* e mais recentemente os *werables*) (PRESSMAN, 2009). A maioria dos novos dispositivos possui algum tipo de conexão sem fio que permite o acesso à internet, e com isso estão livres para trocar informações com as aplicações Web que estão disponíveis na nuvem.

Com a ascensão da Web e da computação em nuvem, alguns dos limites antes estabelecidos pelas redes internas não existem mais, e uma aplicação que antes tinha um número limitado de usuários, agora pode ter milhares ou até milhões, da noite para o dia. Não é raro encontrar casos de aplicações portadas para um ambiente na nuvem que apresentaram

problemas de performance e disponibilidade. Portanto, as aplicações que irão utilizar um ambiente em nuvem precisam ser desenvolvidas ou adaptadas considerando este alto grau de concorrência para garantir a disponibilidade do serviço através da escalabilidade. Escalabilidade é o termo que indica a capacidade de estar preparado crescer. Ela é de suma importância na computação em nuvem, onde as aplicações podem ser submetidas à um alto nível de concorrência, que representa dois ou mais processos sendo executados simultaneamente fazendo uso de recursos compartilhados.

As aplicações Web focam cada vez mais na interatividade, experiência do usuário e integração. Uma boa parcela já disponibiliza alguma *Application Programming Interface* (API) para integração e consulta de suas informações. Com isso, além de um crescente número de usuários, começamos a ter a carga também de outros aplicativos que fazem requisições a esta API, e geralmente em quantidades muito maiores do que os próprios usuários. Para atender demandas com estes patamares de crescimento, é preciso adotar economia de escala. Permitir que os recursos computacionais sejam disponibilizados de maneira crescente e dinâmica ou pelo menos, geridos de uma forma mais eficiente para suportar a carga excedente e não afetar a estabilidade do sistema.

Algumas arquiteturas e técnicas utilizadas no desenvolvimento de software até os anos 2000 se tornaram ineficientes com a evolução das aplicações, dos processadores e do alto grau de concorrência exigido na computação em nuvem. Mesmo assim, estas técnicas continuam sendo empregadas em uma boa parcela das aplicações, que poderiam se beneficiar bastante através da utilização de outras estratégias.

O software amadurecera, ultrapassando o hardware como a chave para o sucesso de muitos sistemas baseados em computador e tornando-se um tema de preocupação administrativa, muitas vezes sendo o fator que diferencia as empresas (PRESSMAN, 2009). Com crescimento do mercado, a internet e os dispositivos móveis tornam-se cada vez mais presentes no mundo empresarial, e tem mudado a forma de fazer negócios de diversos setores, muitas vezes saindo de um segundo plano para se tornar foco principal destes. Algumas empresas estão optando pelo modelo de computação em nuvem que tem atraído muita atenção nos últimos tempos, sendo que através dela surgiram novos modelos para se disponibilizar e adquirir software e infraestrutura, alguns destes modelos são:

- **IAAS (Infraestrutura como Serviço)**: oferece uma estrutura computacional como serviço, geralmente uma máquina virtual com algum sistema operacional pré-instalado;

- **PAAS (Plataforma como Serviço)**: oferece uma plataforma focada no desenvolvimento e aplicações, permitindo desenvolver, executar e gerenciar a aplicação sem a complexidade de manter toda a infraestrutura necessária;

- **SAAS (Software como Serviço)**: é um modelo de comercialização de aplicações Web em que o software é hospedado de forma centralizada e é comercializado em modelo de assinatura.

O SAAS representa um modelo muito interessante de comercialização para as aplicações Web. Através da centralização da aplicação, é possível eliminar grande parte da complexidade gerada pelos processos relacionados a infraestrutura e disponibilização de recursos computacionais (tanto por parte do cliente quanto do fornecedor). Através da redução da complexidade também se abrem as portas para novos mercados, como empresas de pequeno porte que não possuem equipe de Tecnologia da informação (TI) por exemplo. Outro grande benefício deste modelo é permitir um modelo de vendas mais automatizado e com custo de aquisição bem abaixo do modelo tradicional.

1.1 Motivação

As aplicações que optam por trabalhar no modelo SAAS enfrentam alguns desafios. Um ponto bastante impactante quando se fala em computação em nuvem é em relação a conectividade, pois as estações de trabalho necessitam de acesso ininterrupto a internet, já que a falta desta irá acarretar a impossibilidade de acesso a aplicação. Com a centralização, a aplicação terá uma demanda muito maior e por isso é necessário que sejam tomadas precauções quanto a disponibilidade e escalabilidade.

Na computação em nuvem, os limites para o crescimento de uma aplicação são muitas vezes desconhecidos. Como a aplicação se torna mais acessível a novos usuários, a mesma pode sofrer uma variação de carga muito alta em um curto período de tempo. O suporte a estas variações de carga deve estar previsto na arquitetura da aplicação. Algumas aplicações que migraram para este modelo não estão preparadas para lidar com o número crescente de usuários

em uma plataforma centralizada e acabam por sofrer degradação de performance e até travamentos devido ao crescente número de acessos concorrentes.

Um padrão muito adotado atualmente é a construção de aplicações constituídas de vários *microservices* que disponibilizam uma API para integrar suas funcionalidades. Estas pequenas aplicações comunicam-se através de requisições baseadas no protocolo *Hypertext Transfer Protocol* (HTTP). As requisições são operações de Entrada e Saída (E/S) e sendo assim, podem ser realizadas de maneira síncrona ou assíncrona.

As estratégias mais utilizadas para a construção de servidores Web de alta performance são a utilização de *thread pools*[1] (servidores *multithread*) e arquiteturas baseadas em eventos. Para o servidor ter um bom tempo de resposta ao atender centenas de requisições simultâneas é necessário um alto grau de eficiência no tratamento das mesmas. Os servidores *multithread* funcionam de maneira que uma *thread* é alocada para atender a cada requisição. Já os servidores que adotam arquiteturas baseadas em eventos dividem o processamento das requisições em diferentes estágios e utilizam operações de E/S não bloqueantes (assíncronas) para atender simultaneamente as requisições de diferentes clientes utilizando uma mesma *thread* (BELTRAN, CARRERA, TORRES, & AYGUADÉ, 2004).

Apesar de ser um assunto de extrema importância, pois permeia todas as aplicações Web, a abordagem dada ao assunto em estudos comparativos geralmente tem foco somente no servidor Web (camada que trata a chegada da requisição do cliente), que apesar de ser de grande importância não é a única camada da aplicação que deve ser avaliada. Em qualquer aplicação real existem inúmeros outros fatores e processos que são desencadeados através da chegada de uma requisição. Neste aspecto, a maioria destes comparativos deixa uma lacuna ainda não totalmente explorada, que diz respeito à análise aprofundada destes diferentes modelos de concorrência de software quando aplicados a outras camadas da aplicação, como o acesso aos dados por exemplo.

É importante notar que cada caso de uso possui sua particularidade, porém ao identificarmos em qual ponto a aplicação está consumindo seu tempo, podemos explorar e aplicar técnicas mais efetivas para a solução do problema. Existem diversos modelos para se trabalhar com concorrência, sendo que podem ser utilizadas as primitivas mais básicas como

[1] Conjunto de *threads* pré-alocadas para a execução de tarefas.

processos e *threads* e realizar todo o controle de concorrência manualmente (o que não é uma tarefa trivial), ou utilizar um *framework* que possua abstrações de alto nível e gerencie a carga de forma automatizada.

1.2 Objetivos

1.2.1 Objetivo geral

Este trabalho visa realizar um estudo comparativo detalhado sobre o modelo *multithread* e o modelo de atores em aplicações Web através da realização de testes de carga em um conjunto de cenários com diferentes requisitos de performance e concorrência, levando em consideração todas as camadas da aplicação e não apenas de um servidor Web. Este estudo visa identificar as vantagens e desvantagens de cada um dos modelos na implementação de cada um dos cenários, visando uma maior eficiência.

1.2.2 Objetivos específicos

Dentre os objetivos específicos do presente trabalho, podemos citar:

- Revisão bibliográfica sobre os principais modelos de concorrência em aplicações Web;
- Definir um conjunto de cenários sobre os quais serão feitas as avaliações;
- Implementar protótipos dos cenários definidos em ambos os modelos selecionados;
- Realizar simulações de carga nos protótipos desenvolvidos;
- Coletar os dados resultantes das simulações;
- Analisar quantitativamente os dados coletados.

1.3 Organização do trabalho

Buscando facilitar o entendimento deste trabalho, o mesmo foi dividido em capítulos que estão organizados conforme descrito a seguir.

O capítulo introdutório contextualiza a evolução do cenário da Web desde seu surgimento até o momento atual e os problemas advindos desta evolução, que se tornaram na motivação para a realização deste trabalho. Além disso, define os objetivos da proposta, os objetivos específicos, bem como a estrutura do trabalho.

No segundo capítulo são explorados os referenciais teóricos existentes relacionados ao trabalho. São abordados assuntos como Engenharia de Software, Arquitetura de Software, Computação em Nuvem, Escalabilidade e Concorrência que dão o embasamento teórico necessário para o desenvolvimento da proposta.

No terceiro capítulo é abordada a metodologia utilizada na realização deste trabalho. São abordados neste capítulo os métodos científicos, procedimentos empregados e também as etapas que dão sequência ao estudo.

No quarto capítulo são apresentados os cenários de simulação com detalhes de seu funcionamento e objetivos, as métricas que serão coletadas durante a execução das simulações, os modelos de concorrência selecionados para avaliação, simulações de carga aplicadas nos protótipos, descrição das ferramentas utilizadas, apresentação das implementações realizadas e dos resultados obtidos.

No quinto capítulo são apresentadas as considerações finais obtidas através da realização deste estudo.

2 REFERENCIAL TEÓRICO

Este capítulo busca definir e esclarecer os conceitos relacionados ao trabalho utilizando informações obtidas através pesquisa bibliográfica realizada. Inicialmente será apresentada a grande área relacionada à temática do presente trabalho: A Engenharia de Software. Posteriormente será abordada a área de Arquitetura de Software e diferentes modelos de arquitetura de software como cliente-servidor, camadas e *Service Oriented Architecture* (SOA) uma vez que estes temas estão diretamente relacionados ao presente trabalho. Também será abordada a Computação em nuvem (SAAS, PAAS, IAAS), uma vez que é ela quem traz maiores desafios relativos à concorrência em aplicações Web. Por fim, assuntos como Escalabilidade, Computação paralela, e concorrência serão detalhados pois fundamentam os blocos de construção básicos para o entendimento e desenvolvimento do trabalho.

2.1 Engenharia de Software

Engenharia de software é a disciplina que engloba todas as partes da construção de software, desde sua especificação, desenvolvimento, entrega e manutenção (SOMMERVILLE, 2007). O *Software Engineering Body of Knowledge* (SWEBOK) define a Engenharia de Software como “a aplicação de uma abordagem sistemática e quantificável para o desenvolvimento, operação e manutenção de software, isto é, a aplicação da engenharia ao software”. A Engenharia de Software é uma tecnologia de camadas, conforme demonstrado na Figura 1, e a sua fundação tem base na camada de processos, que permite o desenvolvimento racional e oportuno de softwares.

As camadas são definidas da seguinte forma (PRESSMAN, 2009):

- A camada de *processos* precisa estar estabelecida para aplicarmos de forma eficiente a engenharia de software. O processo consolida a base para o gerenciamento dos projetos de software e estabelece o contexto no qual serão aplicados os métodos para garantir o sucesso do projeto;

- Os métodos proveem a técnica para construção de software e abrangem a comunicação, análise de requisitos, modelagem do sistema, construção do software, testes e suporte;

- As ferramentas proveem métodos automatizados ou semiautomatizados para auxiliar o processo. O termo *CASE (Computer-aided software engineering)* define a integração destas ferramentas ao processo de engenharia de software.

Figura 1 – Camadas da Engenharia de Software

Fonte: (PRESSMAN, 2009).

O (SWEBOK, 2014) organiza as áreas de conhecimento da Engenharia de Software da seguinte forma:

- **Requisitos de software**: busca a elicitação, análise, especificação e validação dos requisitos durante todo o ciclo de vida do software;

- **Projeto de software**: busca através da análise de requisitos construir e descrever a estrutura interna do software que servira de base para sua implementação;

- **Implementação de software**: refere-se à construção detalhada de um software através da codificação, verificação, testes e depuração;

- **Teste de software:** busca validar o funcionamento de um software através da análise de resultados esperados dentro de um número finito de casos de testes;
- **Manutenção de software:** trata da constante modificação e evolução que um software sofre no seu ciclo de vida;
- **Gerência de Configuração:** tem por objetivo gerenciar as mudanças ocorridas durante o processo de desenvolvimento do software;
- **Gerência de Engenharia:** tem foco nas atividades de gerenciamento como planejamento, coordenação, medidas, monitoramento, controle e geração de relatórios para garantir que o processo ocorra de forma eficiente e com foco no objetivo.
- **Processos de Engenharia:** estabelece uma sequência prática para auxiliar o desenvolvimento do software;
- **Modelos e Métodos:** busca impor uma certa estrutura a Engenharia de Software com o objetivo de fazer as atividades serem mais sistemáticas e automatizadas;
- **Qualidade:** busca definir processos para validar se o software implementado atende os requisitos especificados com os padrões de qualidade estabelecidos.

2.2 Arquitetura de Software

Arquitetura de software é uma representação geral de alto nível da estrutura de um software que define um modelo intelectualmente compreensível da estrutura do sistema e do funcionamento de seus componentes internos e de como essa estrutura provê as funcionalidades e mantêm a integridade do sistema. Representa a organização dos componentes utilizados pelo sistema e detalha a maneira como estes componentes interagem entre si e com as estruturas de dados existentes (PRESSMAN, 2009).

A representação da arquitetura permite que seja analisada, antes mesmo de iniciar o desenvolvimento, a efetividade do projeto arquitetural no atendimento dos requisitos do software, reduzindo assim os riscos associados à sua construção. Neste ponto também é possível analisar outros modelos arquiteturais e validar se a arquitetura desenvolvida realmente se

encaixa melhor no software em questão, já que ainda é uma etapa em que a mudança é relativamente fácil. As principais razões que levam a sua definição são (PRESSMAN, 2009):

- Melhoria na comunicação entre as os envolvidos no desenvolvimento de um software (cliente e equipe de desenvolvimento);
- Evidenciar decisões de projeto iniciais que tem grande impacto no trabalho de Engenharia de Software e sucesso do projeto.

2.2.1 O modelo cliente-servidor

Nas últimas décadas, muitas aplicações foram repensadas para suportar este modelo de arquitetura (aplicações Web são um exemplo). A essência deste modelo é a distribuição de recursos centralizados através de clientes. De modo geral está arquitetura tem as seguintes funcionalidades (PRESSMAN, 2009):

- Interfaces gráficas são implementadas no cliente, bem como as funcionalidades do sistema;
- Bases de dados são alocadas no servidor, que tem por tarefa manter os dados íntegros;
- Funcionalidades especializadas podem continuar rodando no servidor;
- Surgem novos requisitos de segurança e controle que precisam ser estabelecidos tanto do lado do cliente quanto do servidor.

2.2.2 O modelo em três camadas (Apresentação, negócio, dados)

A arquitetura Modelo-Visão-Controlador (*Model-View-Controller MVC)* é uma das várias arquiteturas sugeridas para a criação de aplicações, e prega a separação da aplicação em três camadas distintas (PRESSMAN, 2009):

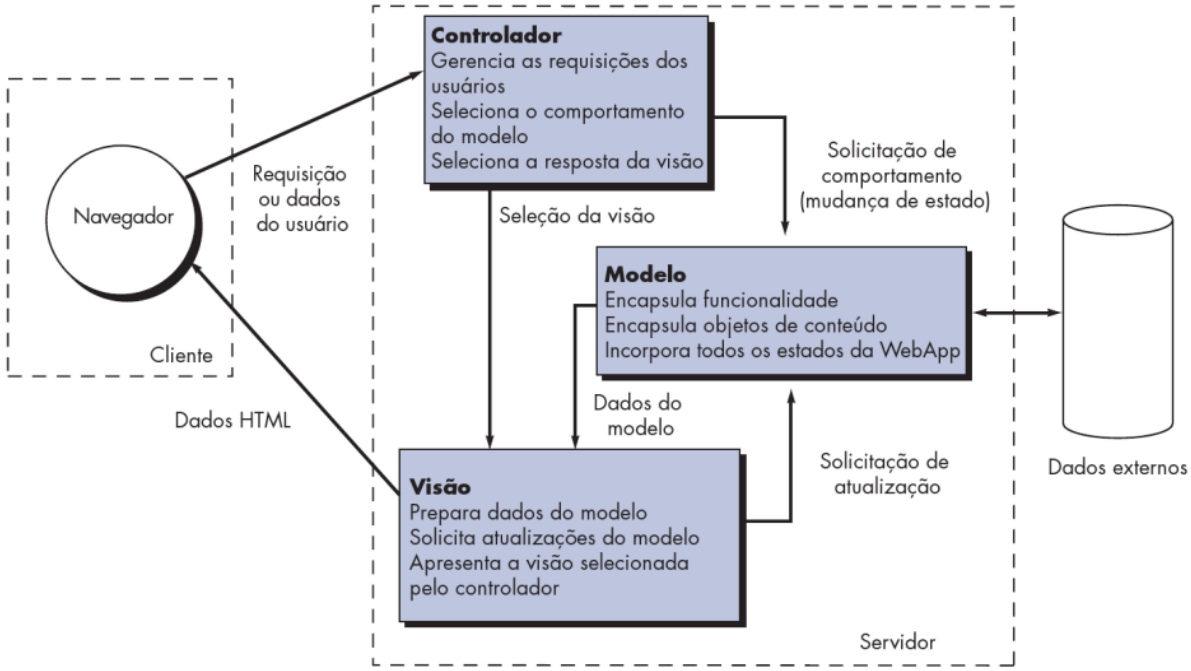
- ***Model***: camada que contém os dados e toda a lógica de funcionamento da aplicação, incluindo acesso a fontes de dados externas e processamentos específicos;

- ***View***: camada que contém as funcionalidades relacionadas com a interface, como apresentação de conteúdo e funções acessíveis para o usuário.

- ***Controller***: camada que gerencia o acesso ao modelo e a visão, controlando o fluxo de dados entre estas camadas.

A Figura 2 demonstra o processo de tratamento da requisição do usuário, o controlador é o componente que recebe a requisição e a partir dela acessa a camada de modelos para consultar ou alterar informações, após a realização das operações necessárias nesta camada o controlador seleciona uma visão e encaminha os dados necessários para seu processamento (PRESSMAN, 2009).

Figura 2 – Modelo de arquitetura MVC



Fonte: (PRESSMAN, 2009).

2.2.3 Arquiteturas orientadas a serviço (SOA)

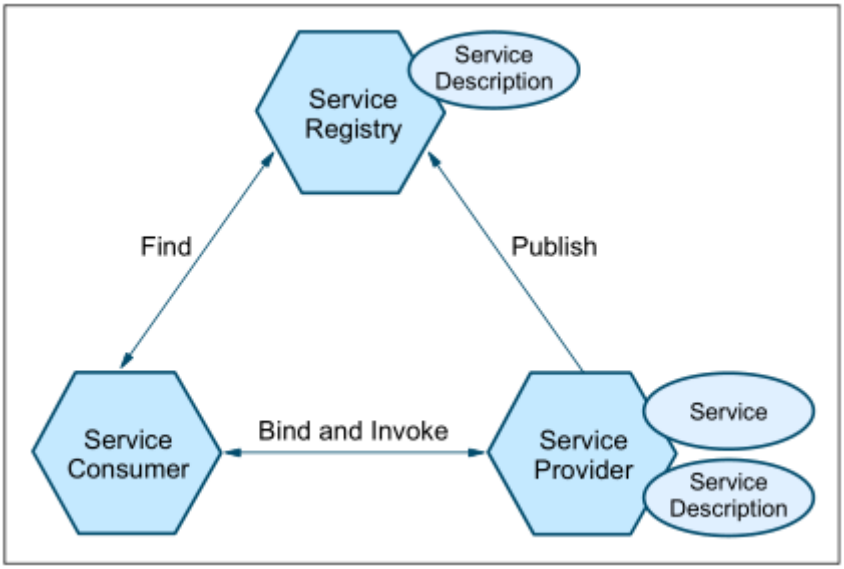
SOA é um padrão arquitetural no qual um componente prove serviços para outro através de um protocolo de comunicação qualquer, sendo que geralmente este processo ocorre através de uma rede. Cada serviço provê uma funcionalidade de forma isolada, ou seja, possuem baixo acoplamento e são independentes, e como comunicam-se através de algum protocolo, cada

serviço fica livre para ser implementado utilizando um conjunto de ferramentas diferentes que se adequem melhor aos requisitos (ERL, 2005).

Os serviços possuem uma interface bem definida e podem colaboram com outros serviços e aplicações para prover funcionalidades mais complexas. A Figura 3 ilustra um processo comum nas arquiteturas orientadas a serviços, que é a descoberta e utilização de serviços disponíveis através de um serviço de registro centralizado (ENDREI, et al., 2004).

A arquitetura orientada a serviços pode ser implementada utilizando diversas tecnologias e protocolos diferentes como CORBA, DCOM e J2EE, porém as implementações mais comuns são baseadas no protocolo de transporte HTTP e conhecidas como Web Services (ENDREI, et al., 2004).

Figura 3 – Colaboração entre serviços



Fonte: (ENDREI, et al., 2004).

2.2.3.1 SOAP

SOAP (*Simple Object Access Protocol* ou Protocolo Simples de Acesso a Objetos) é um protocolo de comunicação projetado para ser simples e extensível e prover as estruturas necessárias para descrever o conteúdo da mensagem e as instruções necessárias para seu processamento, sendo que toda a comunicação é feita no protocolo é codificada em *Extensible Markup Language* (XML). Além disso, o SOAP é totalmente independente de qualquer

protocolo de transporte, rede e linguagem de programação, sendo o único requisito para sua utilização o processamento de mensagens em formato XML (ENDREI, et al., 2004).

É muito comum utilizar SOAP em conjunto com WSDL (*Web Services Description Language* ou Linguagem de Descrição de Web Services) que é uma linguagem específica utilizada para descrever as características de um determinado Web Service. O principal objetivo desta linguagem é responder algumas questões básicas sobre um Web Service como do que se trata o serviço (qual funcionalidade), onde o serviço está localizado e como seus métodos podem ser chamados (ENDREI, et al., 2004).

2.2.3.2 REST

A arquitetura REST (*REpresentational State Transfer* ou Transferência de Estado Representacional) foi criada por Roy Thomas Fielding em sua tese de doutorado no ano 2000. O REST foi fortemente inspirado pela arquitetura da Web e define uma série de regras e restrições a serem seguidas para garantir que uma aplicação seja escalável, tolerante a falhas e extensível. Dentre as regras estão (FIELDING, 2000):

- Desacoplamento do cliente e do servidor (modelo cliente-servidor);
- Não manter estado no servidor (*Stateless*);
- Utilização de camadas;
- Utilização de cache;
- Utilizar identificadores únicos para recursos da aplicação.

O REST descreve a Web como um modelo para aplicações de *hypermedia* distribuídas em que os recursos relacionados comunicam-se através da troca de representações de seu estado. A implementação mais comum desta arquitetura é em cima do protocolo HTTP, porém sua definição é totalmente independente de protocolos específicos (WEBBER, PARASTATIDIS, & ROBINSON, 2010).

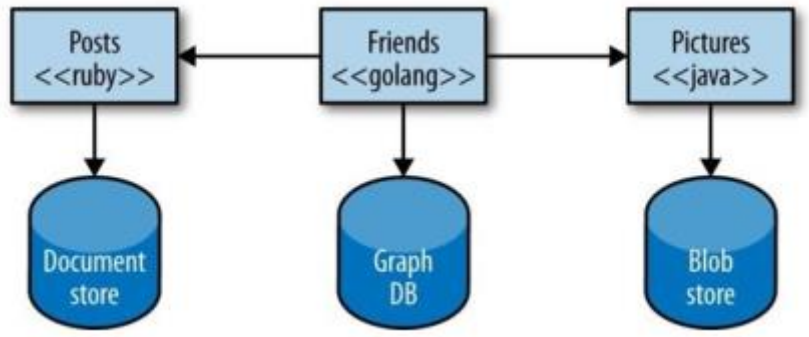
2.2.4 Microservices

A utilização *microservices* tem se tornado bastante popular nos últimos anos para a construção de sistemas distribuídos que promovem um alto grau de modularização de suas funcionalidades. Cada uma das funcionalidades é implementada por meio de uma pequena aplicação que atua de modo colaborativo com outros serviços para compor uma solução mais complexa, mantendo assim seu ciclo de desenvolvimento desacoplado dos outros serviços. A comunicação entre as aplicações é realizada por meio da API disponibilizada por cada aplicação através de algum protocolo independente de qualquer tecnologia específica (NEWMAN, 2015).

Os *microservices,* hoje considerados um padrão arquitetural, surgiram de modo 'natural' nos meios de desenvolvimento devido a crescente complexidade de desenvolver e gerir bases de código de grandes aplicações monolíticas. O principal trunfo deste modelo é permitir que as aplicações sigam o princípio de responsabilidade única e através disso tornem-se mais fáceis de se gerir, manter e integrar novos membros as equipes de desenvolvimento responsáveis (NEWMAN, 2015).

Como as aplicações comunicam-se estritamente através de protocolos independentes de tecnologias, é possível utilizar diferentes tecnologias para compor as soluções, possibilitando a seleção das melhores ferramentas disponíveis para implementar os requisitos de cada aplicação. A Figura 4 exemplifica a integração entre serviços implementados utilizando diferentes tecnologias (NEWMAN, 2015).

Figura 4 – Comunicação entre serviços utilizando diversas tecnologias

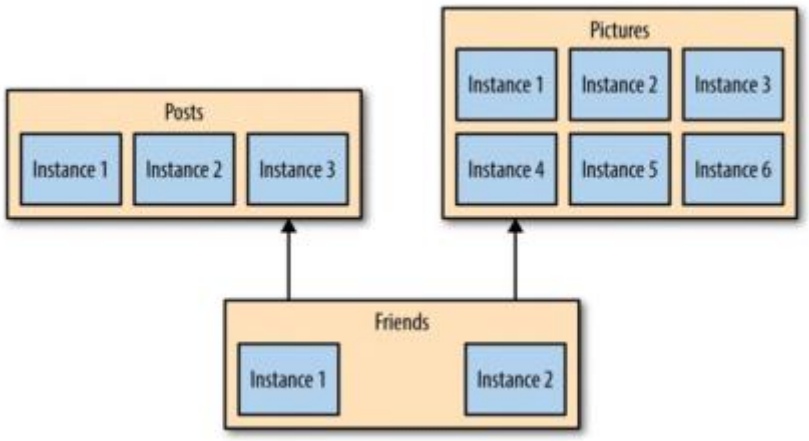


Fonte: (NEWMAN, 2015).

Outro benefício dos *microservices* diz respeito a escalabilidade. Enquanto em aplicações monolíticas é obrigatório escalar a aplicação como um todo, quando se trata de *microservices*

e aplicações independentes podemos escalá-los de forma individual conforme a necessidade. A Figura 5 demonstra este processo (NEWMAN, 2015).

Figura 5 – *Microservices* escalando de forma individual

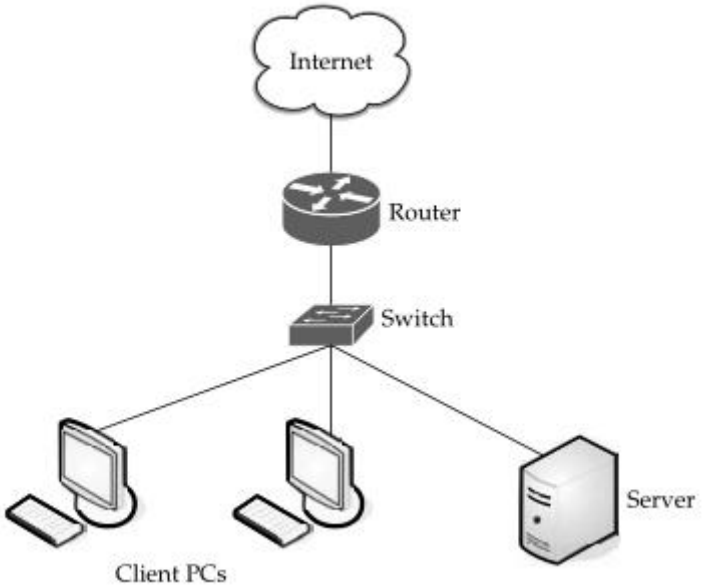


Fonte: (NEWMAN, 2015).

2.3 Computação em nuvem

O nome computação em nuvem teve origem em uma metáfora utilizada para representar a Internet em diagramas de rede. Nestes diagramas, existe o costume de ilustrar a Internet com uma nuvem, que representa uma abstração no diagrama, algo que é necessário para seu funcionamento, porém está fora do escopo do domínio da rede ou é de responsabilidade de outra entidade, como demonstrado na Figura 6. A essência da computação em nuvem é permitir o acesso a aplicações que não estão instaladas em um computador local, e sim rodando em qualquer lugar, como por exemplo em algum *datacenter* distante que hospeda várias aplicações (VELTE, VELTE, & ELSENPETER, 2009).

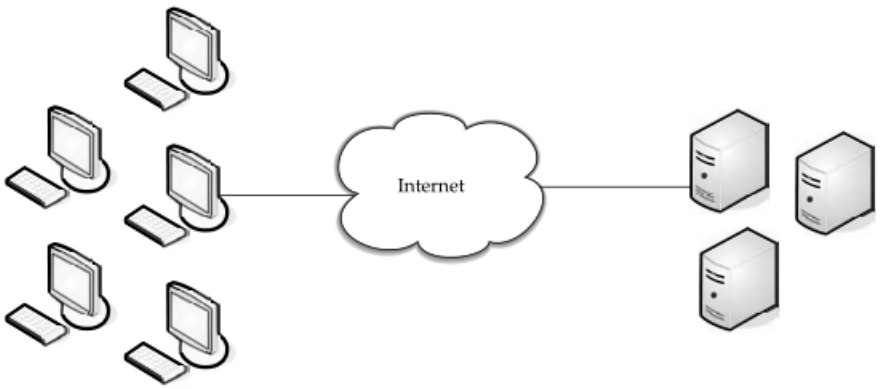
Figura 6 – Representação da Internet em um diagrama de rede



Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

Uma das promessas da computação em nuvem que mais chama a atenção é possibilidade de cortar custos operacionais e de investimentos em infraestrutura, permitindo assim que os departamentos de Tecnologia da Informação (TI) mantenham seu foco em projetos de interesse do negócio da empresa, ao invés de utilizar este tempo mantendo um *datacenter*. Como as aplicações e infraestrutura passam a ser responsabilidade da empresa prestadora de serviços os custos com energia, estrutura física alocada para os equipamentos, aquisição e atualização de equipamentos caem drasticamente. A maioria dos serviços disponibilizados na nuvem seguem um modelo sob demanda onde o prestador cobra do cliente somente o que for consumido, isso para algumas empresas pode representar um grande corte nos custos operacionais. A Figura 7 demonstra o funcionamento do acesso aos serviços disponibilizados na nuvem (VELTE, VELTE, & ELSENPETER, 2009).

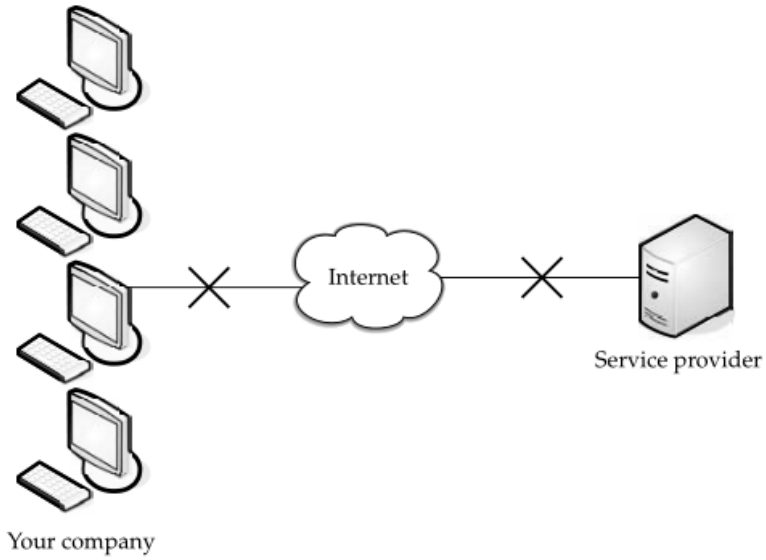
Figura 7 – Serviços disponibilizados na nuvem sendo acessados pelas estações de trabalho



Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

A computação em nuvem também traz alguns problemas em relação a garantia de disponibilidade dos serviços e gera dúvidas em relação a segurança dos dados. Como os serviços são disponibilizados e consumidos através da Internet, caso ocorra um problema com o provedor de acesso à Internet os sistemas ficarão indisponíveis neste período, o mesmo ocorre caso o prestador de serviços na nuvem passe por algum problema, conforme ilustrado na Figura 8. A segurança da informação é outro fator que algumas vezes impede uma migração para a computação em nuvem, visto que algumas aplicações envolvem o uso informações estratégicas ou altamente sigilosas o que acaba gerando uma certa desconfiança ao delegar estes dados para empresas terceiras (VELTE, VELTE, & ELSENPETER, 2009).

Figura 8 – Disponibilidade do serviço depende do acesso à Internet



Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

2.3.1 Componentes da computação em nuvem

As soluções de computação em nuvem são compostas de vários componentes com propósitos específicos como clientes, *datacenters* e servidores distribuídos, representados na Figura 9.

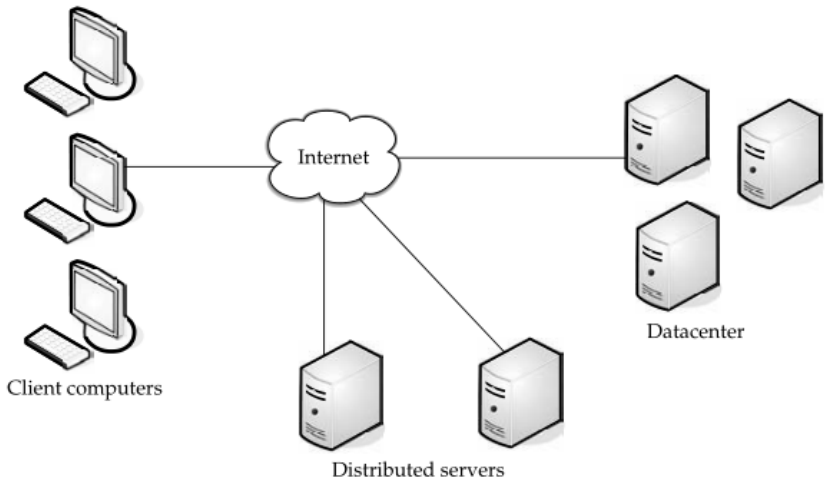
Os componentes são (VELTE, VELTE, & ELSENPETER, 2009):

- **Clientes**: os computadores, notebooks e dispositivos moveis são considerados os clientes na computação em nuvem. Inclusive os dispositivos móveis foram um dos grandes impulsionadores da computação em nuvem, pois a mobilidade requer que os serviços estejam disponíveis através da internet;

- ***Datacenters***: são aglomerados de servidores a partir dos quais as aplicações são servidas aos clientes. Os *datacenters* podem ser compostos tanto maquinas físicas, como maquinas virtuais. A virtualização é uma grande tendência na computação em nuvem pois permite um melhor aproveitamento de hardware e através disto uma redução de custos no licenciamento/alocação de servidores;

- **Servidores distribuídos**: os servidores não precisam obrigatoriamente estarem na mesma localização. A infraestrutura pode estar distribuída em várias regiões diferentes, o que permite uma maior tolerância a falhas (em casos de desastres naturais, por exemplo). Alguns provedores permitem que o usuário escolha a região na qual será inicializada a instancia do servidor selecionada.

Figura 9 – Componentes da computação em nuvem



Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

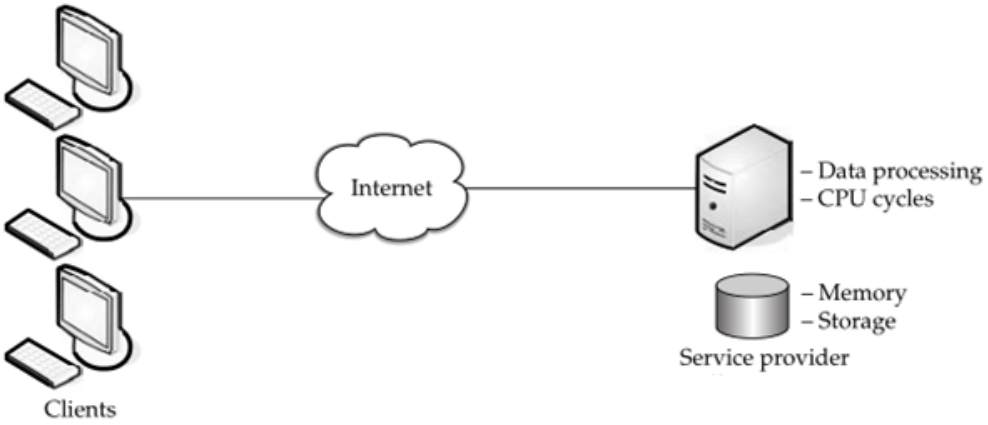
2.3.2 Serviços

O termo serviço na computação em nuvem é a capacidade de reutilizar componentes disponibilizados na rede do fornecedor, isto é amplamente conhecido como '*as a service'*. Os modelos de serviço permitem a redução de custos operacionais através do compartilhamento dos recursos entre diversos usuários, tornado o serviço mais acessível para pequenos negócios (VELTE, VELTE, & ELSENPETER, 2009).

2.3.2.1 IAAS

O termo *Infrastructure as a Service* (Infraestrutura como um serviço) define um modelo em que o provedor de serviços oferece a possibilidade de "alugar" recursos computacionais como por exemplo instancias de servidores, serviços de rede e armazenamento. Geralmente também é oferecia a possibilidade de escalar dinamicamente a capacidade dos serviços de forma automatizada, conforme a necessidade do cliente. A Figura 10 ilustra a operação deste modelo (VELTE, VELTE, & ELSENPETER, 2009).

Figura 10 – Modelo IAAS

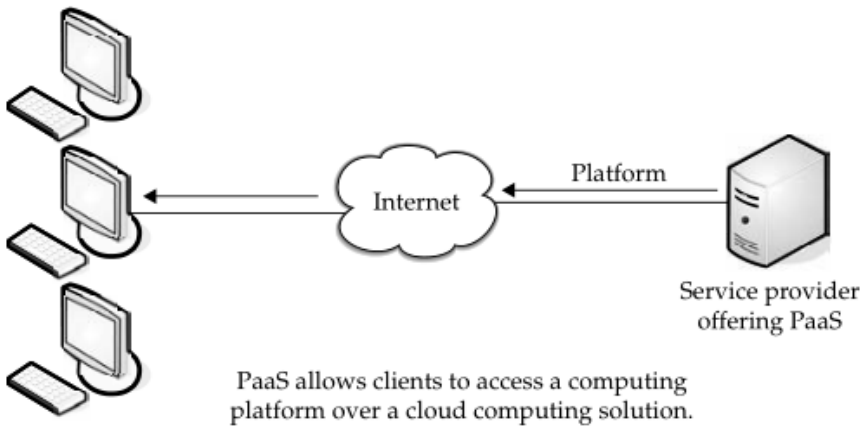


Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

2.3.2.2 PAAS

O modelo *Plataform as a Service* (Plataforma como um serviço) oferece aos desenvolvedores de aplicações os recursos necessários para a construção de suas aplicações diretamente da internet. Os serviços geralmente oferecem meios para projetar, desenvolver, testar e servir as aplicações desenvolvidas, e podem também oferecer serviços adicionais como por exemplo versionamento, integração com diversos bancos de dados, escalabilidade automatizada. A Figura 11 ilustra o uso deste modelo (VELTE, VELTE, & ELSENPETER, 2009).

Figura 11 – Modelo PAAS



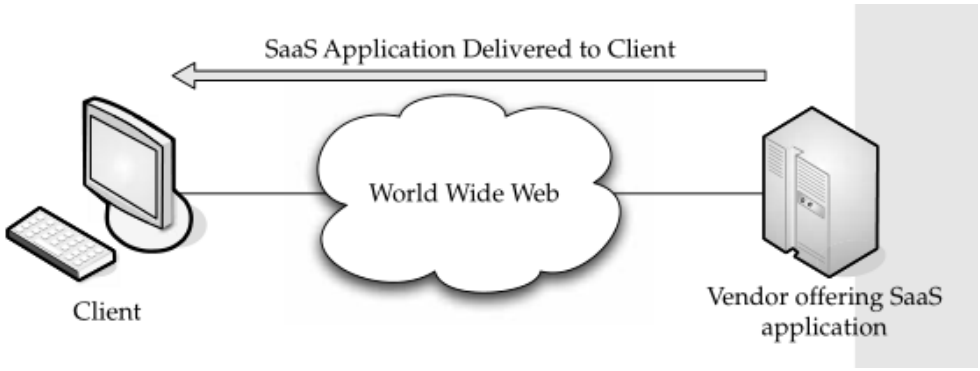
Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

2.3.2.3 SAAS

O *Software as a Service* (Software como um serviço) é o modelo no qual uma aplicação é disponibilizada ao cliente via internet, com isso o cliente não precisa se preocupar com questões como infraestrutura, manutenção ou suporte. Este modelo se difere dos modelos de computação distribuída pois as aplicações são desenvolvidas já pensando em um modelo em que serão compartilhadas entre vários clientes e disponibilizadas especificamente em um navegador Web via internet. A Figura 12 ilustra o funcionamento do modelo (VELTE, VELTE, & ELSENPETER, 2009).

A adoção de softwares neste modelo pode representar uma redução de custos significante para a empresa quando comparado ao modelo de licenciamento. Também possui a vantagem de estar disponível para acesso de qualquer lugar via internet, custos reduzidos com manutenção e em geral menos problemas com segurança. Um dos problemas para a adoção deste modelo é dependência criada com o provedor de software. Algumas aplicações não possuem nem um método para exportação dos dados, ou em algumas determinadas situações até cobram pelo serviço, o que acaba gerando certa desconfiança nas empresas interessadas em adotar o modelo (VELTE, VELTE, & ELSENPETER, 2009).

Figura 12 – Modelo SAAS



Fonte: (VELTE, VELTE, & ELSENPETER, 2009).

2.4 Escalabilidade

Os termos escalabilidade e performance estão intimamente relacionados, porém é importante ressaltar que representam coisas diferentes. A performance diz respeito a quão rápida e efetivamente uma operação pode ser completada pela aplicação, já escalabilidade diz

respeito a capacidade do sistema em acomodar um volume de trabalho de forma controlada (sem comprometer o funcionamento do sistema com quedas abruptas de performance e travamentos) e também a possibilidade de expansão de sua capacidade através da adição de mais recursos de hardware (LIU, 2009).

Se a performance de um sistema fica inaceitável ao atingir um certo grau de carga e não pode ser melhorada mesmo com a adição de hardware, então pode ser dito que a aplicação não é escalável, este problema de escalabilidade não pode ser resolvido sem mudanças arquiteturais na aplicação (LIU, 2009).

A aplicação pode escalar de duas formas: ou melhorando o hardware do servidor sobre qual ela roda, ou adicionando mais servidores para processamento. A primeira forma é conhecida como escalabilidade vertical e consiste em adicionar mais recursos a um servidor (memória e processadores por exemplo) para aumentar a performance da aplicação, este método geralmente não requer mudanças no código da aplicação. A segunda forma é conhecida como escalabilidade horizontal e requer que a aplicação esteja preparada para utilizar diversos servidores para melhorar sua performance (ERB, 2012).

2.4.1 Medidas

Na realização de testes de performance em aplicações de forma geral são consideradas métricas como tempo de resposta, latência e *throughput*, e em casos de testes mais aprofundados podem ser realizadas medidas também em elementos como escalabilidade e robustez. As métricas são definidas da seguinte forma (LIU, 2009):

- **Tempo de resposta**: é o tempo que o sistema leva para processar uma requisição de sua chamada até seu retorno;

- **Latência**: é tempo gasto com acesso a elementos remotos, por exemplo o tempo até uma requisição chegar efetivamente a um serviço Web;

- ***Throughput***: é o número de requisições por segundo que a aplicação consegue servir. Esta métrica é de extrema importância e deve ser cuidadosamente observada durante os testes de carga, e é importante também que os testes sejam construídos de forma que reflitam a carga esperada em produção;

- **Escalabilidade**: mede a resposta da aplicação quando são adicionados/removidos recursos de hardware;

- **Robustez**: mede o comportamento da aplicação em situações de estresse, por exemplo em sistemas de execução longa consiste em verificar se o sistema está liberando recursos (memória por exemplo) de forma correta

2.5 Computação Paralela

A computação paralela fica cada vez mais em evidencia devido a constante demanda por performance e a tendência atual da indústria de processadores em adicionar mais núcleos de processamento ao invés de simplesmente torná-los mais rápidos. Em aplicações que requisitam alta performance, como simulações por exemplo, é obrigatório o uso de técnicas de computação paralela para conseguir um tempo de processamento satisfatório. (PACHECO, 2011).

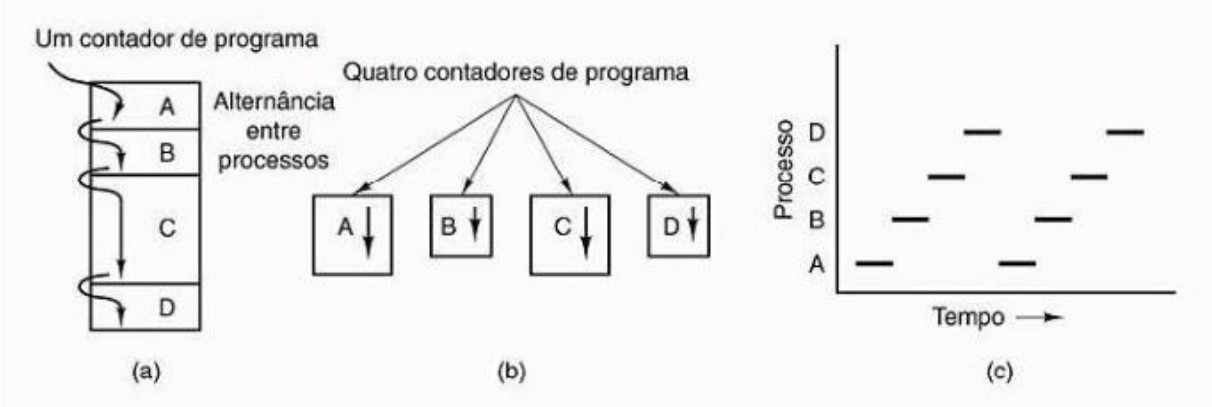
O problema é que programas escritos de forma serial não conseguem aproveitar os núcleos de processamento adicionais. Em geral não existe forma automática para se transformar um programa serial em paralelo. Essas alterações muitas vezes demandam modificações no código dos algoritmos ou substituição dos mesmos. Para poder explorar vários núcleos de processamento torna-se necessária a utilização de mecanismos como processos ou *threads* (PACHECO, 2011).

2.5.1 Processos

Um processo do ponto de vista do sistema operacional é simplesmente uma abstração de um programa em execução. Através desta abstração torna-se possível a realização de operações concorrentes (ou pseudo-concorrentes quando só existe um núcleo de processamento disponível). Em sistemas multiprogramados, o processador executa os processos de forma alternada, para que isso seja possível a cada troca de processo é necessário que sejam salvas algumas informações relativas à sua execução (registradores, variáveis, contador de programa, etc.) para poder voltar sua execução do ponto no qual havia parado, conforme demonstrado na Figura 13. Um único processador pode ser compartilhado entre vários processos e é utilizado

um algoritmo de escalonamento para definir quando parar um processo e iniciar outro (TANENBAUM, 2010).

Figura 13 – Processos rodando ao longo do tempo



Fonte: (TANENBAUM, 2010).

É perfeitamente possível que os processos criem outros processos, conhecidos como processos filhos. Cada processo filho é uma cópia idêntica do estado atual do processo pai, e como ele é totalmente isolado dos outros processos possui sua própria área de memória, registradores, etc. Através da criação de processos filhos conseguimos realizar a paralelização de tarefas de um programa, porém o uso de processos para este fim possui inúmeras desvantagens.

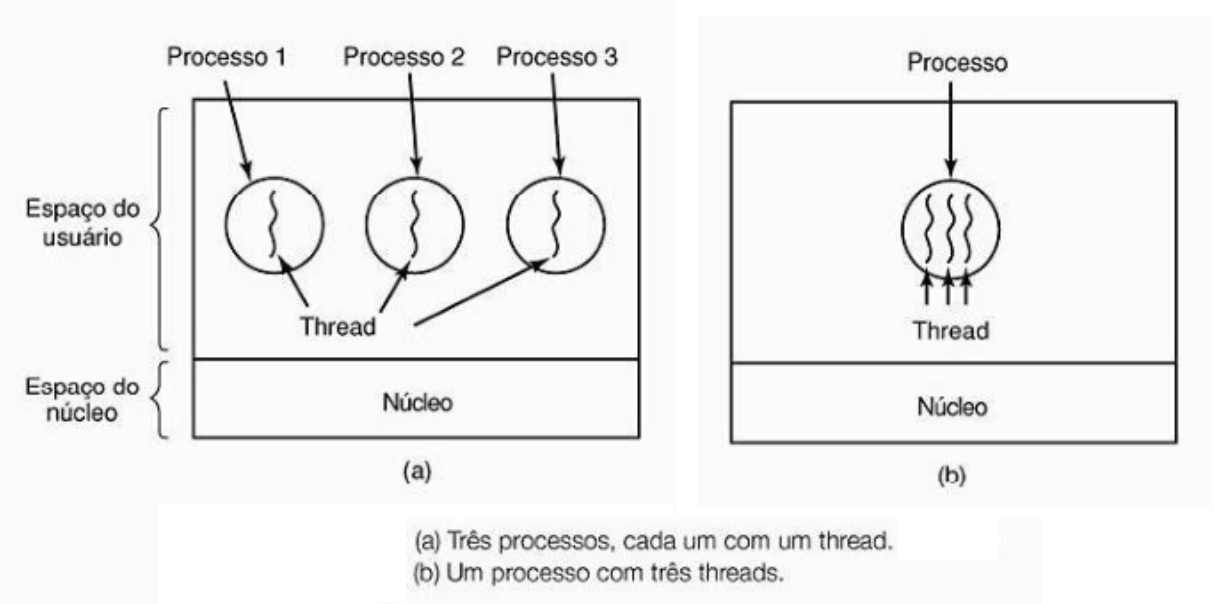
Como cada processo é totalmente isolado, é necessária a utilização de alguma técnica de comunicação entre processos (*Inter-process Communication* ou *IPC)* para realizar a troca das informações necessárias, o que não é tão eficiente como realizar um acesso direto à informação em memória por exemplo. Outro ponto negativo de se utilizar processos é relativo a seu custo de criação/destruição, ou seja, instanciar um novo fluxo de execução utilizando processos é custoso pois existem vários recursos que devem ser alocados e posteriormente desalocados pelo sistema operacional (TANENBAUM, 2010).

Com isso é perceptível que a utilização de processos diretamente para a realização de tarefas paralelas que requerem troca de informações locais não é recomendável. A utilização de processos se encaixa muito melhor quando necessário um isolamento total para evitar interferências entre os fluxos de execução de um programa (TANENBAUM, 2010).

2.5.2 Threads

As *threads* (também referenciadas como processos leves ou subtarefas) são fluxos de execução que residem e compartilham recursos já alocados por um processo, e devido a seu menor custo de alocação são mais indicadas para a paralelização de tarefas. As *threads* possuem algumas características que as diferem dos processos tradicionais. Primeiramente uma *thread* nunca pode existir de forma isolada, pois representam fluxos de execução individuais contidas dentro dos processos (que podem criar várias *threads*, conforme demonstra a Figura 14). Além disto elas compartilham o espaço de endereçamento do processo, podendo assim acessar estruturas de dados de todo o processo diretamente em memória (TANENBAUM, 2010).

Figura 14 – Processo com múltiplas *threads*



Fonte: (TANENBAUM, 2010).

O custo e complexidade de se instanciar uma *thread* é muito menor do que quando comparado a criação de um processo (em alguns sistemas operacionais o tempo criação pode ser até dez vezes maior), isto se deve ao compartilhamento dos recursos já alocados pelo processo. O uso de *threads* pode trazer ganhos significativos de desempenho em diversas situações.

Podemos citar como exemplo sistemas com grande quantidade de operações de E/S, nestes casos as *threads* permitem que o processador realize outras atividades enquanto a operação de E/S não é finalizada, reduzindo assim sua ociosidade. Também existem diversos

problemas que podem ser divididos em partes menores e processados em paralelo, em sistemas com múltiplos núcleos de processamento em que ocorre paralelismo real é possível obter ganhos expressivos de performance (TANENBAUM, 2010).

Devido ao compartilhamento de recursos são necessários alguns cuidados adicionais ao utilizar *threads*. Trechos de código que acessam recursos compartilhados em *threads* distintas estão sujeitos a condições de corrida e devem ser devidamente identificados e tratados com algum algoritmo de exclusão mutua ou alguma outra técnica. Ao aplicar estes métodos, é comum surgirem problemas de *deadlock*[2] no programa. Problemas que emergem devido concorrência em geral são difíceis de serem reproduzidos e corrigidos devido a sua natureza aleatória (TANENBAUM, 2010).

As *threads* podem ser criadas e geridas tanto no espaço do *kernel* do sistema operacional como no espaço do usuário. As *threads* em modo *kernel* possuem um maior custo de criação/destruição associado (por isso geralmente usam-se *thread pools* para aplicações de alta performance) e a troca de contexto se executada muito frequentemente gera uma grande sobrecarga (chamadas de sistema em geral são custosas), porém conseguem aproveitar os núcleos de processamento oferecidos e no caso de sua execução ser bloqueadas o escalonador do sistema operacional passa o controle para outra *thread* (TANENBAUM, 2010).

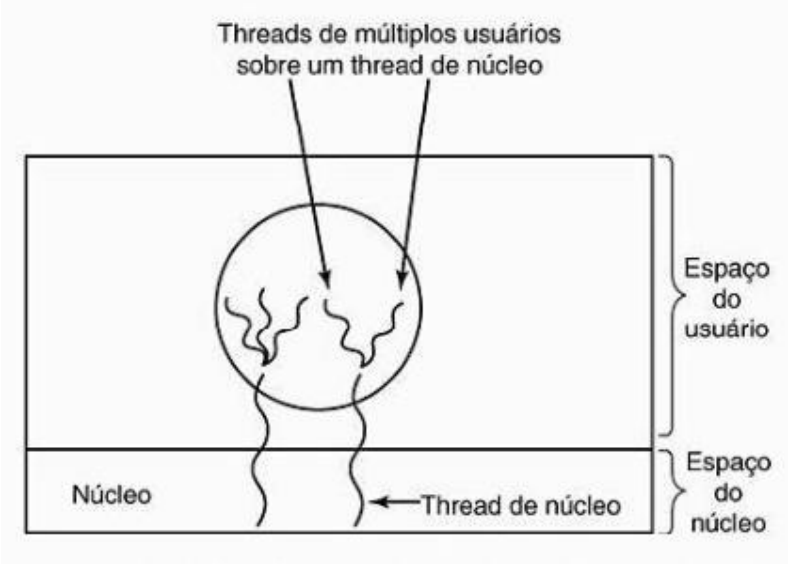
Threads em modo de usuário possuem diversos benefícios relacionados a performance. Como são rotinas locais ao programa, executam de forma eficiente pois não precisam ficar alternando entre o modo usuário e modo *kernel*, evitando assim a realização de trocas de contexto e limpeza dos caches de memória o tempo todo. Porém estas *threads* não possuem nenhum tipo de escalonador automático, ou seja, uma *thread* só será executada quando alguma outra *thread* liberar o processador, isso pode acabar gerando problemas sérios de performance no caso de execução de operações bloqueantes pois nada irá executar até que a *thread* execute a liberação (TANENBAUM, 2010).

De modo geral, na maioria das aplicações são utilizadas implementações hibridas destes modelos, *threads* de modo de usuário são multiplexadas em cima das *threads* em modo *kernel* conforme representado na Figura 15, assim é possível obter os benefícios oferecidos por ambos os métodos atenuando suas deficiências (TANENBAUM, 2010).

---

[2] Situação em que ocorre um impasse, e duas ou mais *threads* ficam bloqueados, esperando umas pelas outras.

Figura 15 – Multiplexação de *threads* em modo usuário

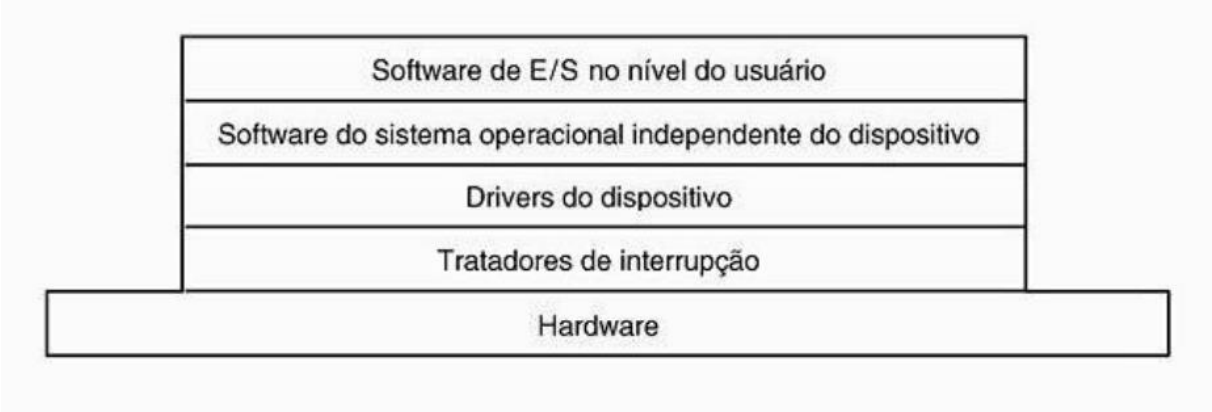


Fonte: (TANENBAUM, 2010).

2.5.3 E/S

Os processos de E/S geralmente possuem uma velocidade muito baixa quando comparada com a velocidade em que o processador pode operar e dependendo da aplicação podem se tornar um problema de performance. O tipo de transferência de uma operação de E/S pode ser síncrona (bloqueante) ou assíncrona (orientadas a interrupções ou eventos) e existem três meios de executá-la: de forma programada, utilizando interrupções ou utilizando mecanismos de *Direct Memory Access* (DMA). É importante ressaltar que fisicamente (a nível de hardware) a maioria destas operações é realizada de forma assíncrona, porém nos programas é muito comum que seja tratada de forma síncrona. Esta abordagem na maioria dos casos irá prover uma boa performance aliada a simplicidade no código, porém em ambientes de alta concorrência pode ser vantajoso o uso de métodos assíncronos para não desperdiçar recursos do sistema operacional com espera de E/S. A Figura 16 demonstra as partes envolvidas na execução de uma operação de E/S (TANENBAUM, 2010).

Figura 16 – Camadas do software de E/S



Fonte: (TANENBAUM, 2010).

2.5.3.1 E/S programada

O processo da E/S programada consiste no uso constante do processador para checar a disponibilidade do dispositivo de E/S e realizar a transferência dos valores desejados para ele, este processo é conhecido como espera ocupada (*busy waiting*) ou *polling*. Um ponto positivo é que a E/S programada é simples de se implementar e gerir, porém possui a desvantagem de manter o processador ocupado o tempo todo até que a operação de E/S seja concluída.

Este modelo de operação é aceitável em algumas situações, como por exemplo em sistemas embarcados onde o processador não possui outras tarefas, e em alguns casos pode ser até conveniente, como por exemplo quando o dispositivo de E/S em questão é rápido. Contudo, quando falamos de aplicações mais complexas a ineficiência deste modelo de operação torna-se mais evidente. A Listagem 1 exemplifica a implementação deste método (TANENBAUM, 2010).

Listagem 1 – Exemplo de um código realizando E/S programada

```
copy_from_user(buffer, p, cont);                /* p é o buffer do núcleo */
for (i=0; i < count; i++) {                     /* executa o laço para cada caractere */
    while (*printer_status_reg !=READY) ;       /* executa o laço até a impressora estar pronta*/
    *printer_data_register = p[i];              /* envia um caractere para a saída */
}
return_to_user();
```

Fonte: (TANENBAUM, 2010)

2.5.3.2 E/S usando interrupção

Uma forma de evitar o processo de *polling* é através do uso de interrupções nas operações de E/S, com isso após a realização de uma operação o escalonador elege outra tarefa para ser executada até o momento em que o dispositivo de E/S gere uma interrupção, sinalizando que está pronto para realizar outra operação, neste ponto o controle volta para o processo em questão que repete este procedimento até finalizar sua execução.

Em comparação com a E/S programada, o uso de interrupções permite um melhor aproveitamento do tempo o do processador, porém também existe degradação de performance devido ao alto número de trocas de contexto realizadas pelo escalonador. A degradação ocorre devido ao escalonador realizar uma troca de contexto para tratar cada interrupção que é gerada pelo dispositivo quando este finaliza uma operação e torna-se disponível (processo que ocorre inúmeras vezes durante uma escrita em disco por exemplo). A Listagem 2 exemplifica a implementação deste método (TANENBAUM, 2010).

Listagem 2 – Exemplo de um código realizando E/S usando interrupção

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Fonte: (TANENBAUM, 2010).

2.5.3.3 E/S usando DMA

Uma alternativa aos métodos anteriores é a utilização de um hardware especifico para realização de operações de E/S, este hardware é conhecido como controlador DMA. A sigla DMA referência a realização de acesso direto à memória, ou seja, o processador irá

disponibilizar as informações em memória e o controlador DMA irá se encarregar de alimentar o dispositivo de E/S com todas estas informações (TANENBAUM, 2010).

O processo é similar com a E/S programada, porém a grande vantagem é que o controlador DMA é quem realiza este procedimento e com isso o processador fica livre para executar outras tarefas neste tempo. A diferença da E/S usando interrupções e com DMA está no número de interrupções geradas, pois o DMA só gera uma interrupção ao final da execução de um bloco de informações e isso torna o processo muito mais performático (TANENBAUM, 2010).

Apesar do controlador DMA geralmente ser muito mais lento que o processador, em geral a sua utilização vale a pena. Em alguns casos específicos no qual ele não consegue operar o dispositivo de E/S em sua velocidade máxima pode ser vantajoso o uso de E/S programada ou orientada a interrupção para evitar degradação de performance. A Listagem 3 exemplifica a implementação deste método. (TANENBAUM, 2010).

Listagem 3 – Exemplo de um código realizando E/S usando DMA

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Fonte: (TANENBAUM, 2010).

2.6 Concorrência

A concorrência em aplicações refere-se à execução de múltiplas tarefas simultaneamente de forma não cooperativa, ou seja, diferente do paralelismo onde os fluxos cooperam para a execução de uma mesma tarefa, na concorrência os fluxos estão executando tarefas distintas (PACHECO, 2011).

Em geral, a forma como a concorrência é tratada nas aplicações pode se encaixar em dois grandes grupos (LAUER & NEEDHAM, 1978):

- **Orientadas a mensagens**: este modelo caracteriza-se pelo uso de poucos processos, filas e comunicação feita exclusivamente por troca de mensagens (também podem ser chamados de eventos, dependendo do contexto);

- **Orientadas a processos**: este modelo caracteriza-se pelo uso de muitos processos que utilizam memória compartilhada para comunicação e também pela necessidade de utilização estruturas de sincronização entre os processos.

2.6.1 Concorrência baseada em *threads*

Este é o modelo mais comum entre as aplicações e também um dos mais antigos. Surgiu com base na programação sequencial e pode explorar o paralelismo real através de múltiplos fluxos de execução (*threads*) simultâneos que são alocados pelo sistema operacional nos núcleos de processamento disponíveis. A utilização deste modelo geralmente implica na utilização de mecanismos de sincronização para evitar condições de corrida ao acessar/modificar estruturas de dados compartilhadas (ERB, 2012).

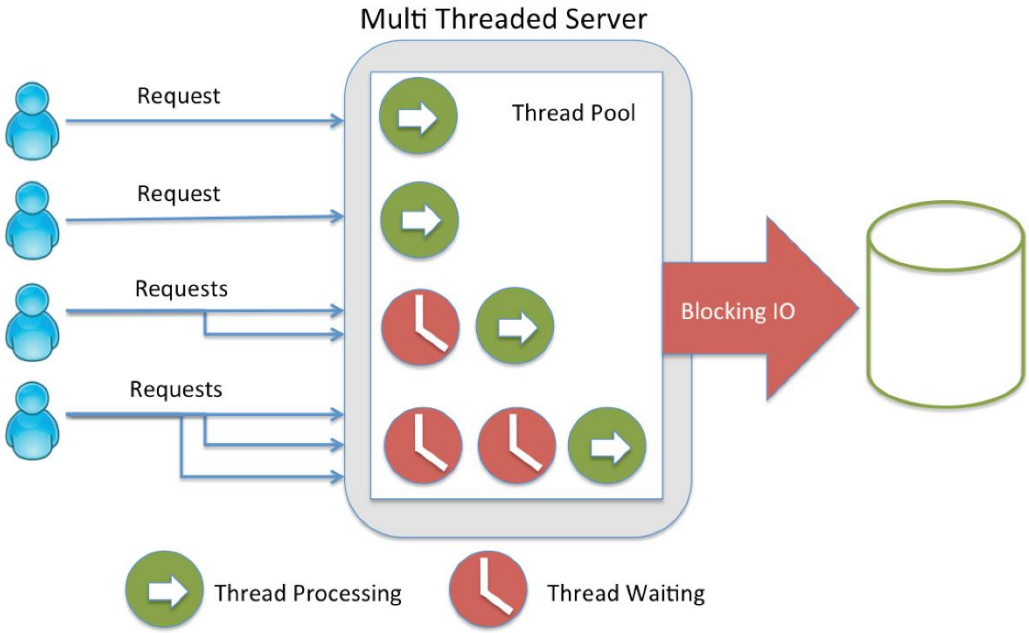
Apesar de ser performático em muitos casos, trabalhar com estruturas de baixo nível como *threads* e *locks*[3] é difícil e propenso a erros (SUTTER & LARUS, Sofware and the Concurrency Revolution, 2005). Quando duas ou mais *threads* precisam compartilhar determinadas informações o acesso realizado a elas deve ser controlado de alguma forma. Este controle geralmente é implementado utilizando algum mecanismo de exclusão mutua como um semáforo ou um monitor que permite gerir os acessos a uma seção crítica (ERB, 2012).

Porém a inclusão de *locks* no código, quando realizado de forma indevida, pode gerar inúmeros outros problemas como por exemplo os *deadlocks*, *livelocks* e *lock starvations* que podem ocorrer em casos em que o código possui uma dependência cíclica de dois ou mais *locks.* Outro problema que surge devido à má utilização de *locks* é quanto a performance, pois a serialização do acesso a uma sessão critica pode criar uma contenção onde várias *threads* ficam bloqueadas apenas aguardando a liberação do *lock*, para evitar isso a implementação dos locks deve estar no nível mais granular possível, o que nem sempre é fácil de identificar e tende a

[3] Estruturas utilizadas para limitar acesso a sessões críticas e recursos compartilhados nas aplicações.

gerar mais problemas de *deadlocks* (ERB, 2012). A Figura 17 ilustra um servidor atendendo múltiplos clientes através da alocação de uma *thread* para cada requisição.

Figura 17 – Utilização de *threads* para processar requisições de clientes a um servidor



Fonte: Retirado da página *Strongloop*[4].

2.6.1 Concorrência baseada em eventos

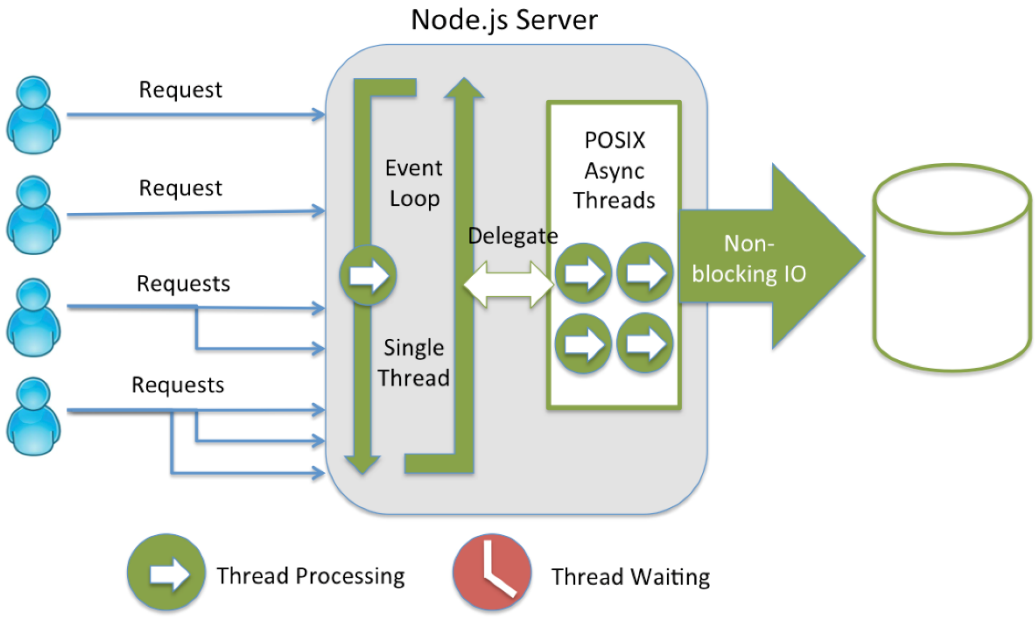
As arquiteturas baseadas em eventos funcionam em cima dos conceitos de *event-loop* e *event handlers*. Apesar da implementação deste modelo ser feita em cima das mesmas primitivas básicas, as *threads*, este modelo apresenta uma forma distinta de tratar a concorrência. Os *event-loops* tradicionalmente rodam com apenas uma *thread* e fazem o uso de filas para armazenar a entrada de eventos. Quando um novo evento é detectado seu processamento é direcionado para o *event handler* responsável pelo seu tratamento. Através deste modo de operação simplificado é possível reduzir muito o número de trocas de contexto e uso da pilha de chamada (ERB, 2012).

Devido ao fato dos *event-loops* executarem em cima de uma só *thread*, as arquiteturas baseadas em eventos fazem uso extensivo de meios de comunicação e operações de E/S assíncronas para evitar ao máximo bloqueio dela. Nos casos em que o bloqueio é inevitável,

[4] Disponível em: <https://strongloop.com/strongblog/node-js-is-faster-than-java>. Acesso em maio 2016.

recomenda-se que o processo seja executado em uma *thread* ou processo separado. A Figura 18 ilustra um servidor atendendo múltiplos clientes utilizando um *event-loop* para direcionar o evento para seu *event handler* responsável (ERB, 2012).

Figura 18 – Utilização de eventos para processar requisições de clientes a um servidor



Fonte: Retirado da página *Strongloop*[5].

2.6.1 Modelo de Atores

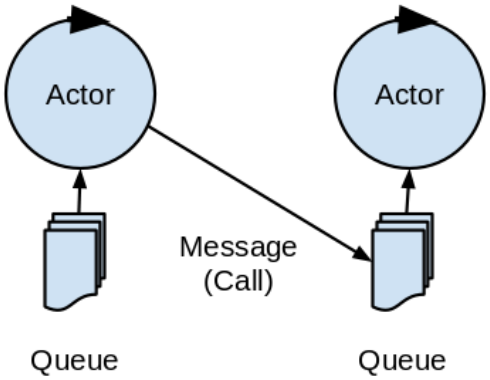
O modelo matemático de atores foi definido por Carl Hewitt na década de 70 como um modelo computacional não determinístico que faz uso extensivo de métodos de comunicação assíncrona. Neste modelo os atores são as primitivas universais para a computação e são utilizados para representar a lógica e paralelização da aplicação. Seu foco na troca de mensagens de forma assíncrona permite a criação de aplicações reativas que conseguem ser responsivas, resilientes e escaláveis. A Figura 19 exemplifica o processo de troca de mensagens e filas utilizados pelo modelo (VERNON, 2015).

O modelo de atores definido por Hewitt diz que todos os elementos dentro de um sistema devem ser atores, como por exemplo um objeto do tipo *String* seria um ator considerando esta modelagem. Porém não existe até hoje nenhuma linguagem que implemente o modelo de atores da forma como foi definido originalmente. O modelo de atores geralmente é implementado

[5] Disponível em: <https://strongloop.com/strongblog/node-js-is-faster-than-java>. Acesso em maio 2016.

como uma abstração em cima de uma linguagem que segue outro paradigma, como por exemplo orientação a objetos ou programação funcional. Esse modelo de construção permite agregar os benefícios de linguagens mais estruturadas com os conceitos mais importantes do modelo de atores (VERNON, 2015).

Figura 19 – Troca de mensagens entre atores



Fonte: Retirado da página *Java is the new C*[6].

Os atores nunca se apresentam de forma isolada, em geral uma funcionalidade é implementada através de um conjunto de atores que se comunicam para realizar uma determinada operação. Os atores seguem alguns princípios básicos, dentre os quais podemos citar (VERNON, 2015):

- Os atores devem seguir o princípio de responsabilidade única;
- Os atores suportam o conceito de máquina de estados finitos, a transição entre os estados é realizada através de mensagens;
- Os atores comunicam-se entre si através de troca de mensagens assíncronas;
- As mensagens utilizadas pelos atores devem ser imutáveis;

[6] Disponível em: <http://java-is-the-new-c.blogspot.com.br/2014/01/comparision-of-different-concurrency.html>. Acesso em maio 2016.

- Cada ator processa somente uma mensagem por vez, por isso é de extrema importância que caso o ator precise executar uma ação bloqueante o faça em uma *thread* separada;

- O estado interno de um ator jamais deve ser exposto para outros objetos, a única forma de alterar o comportamento de um ator é através de uma mensagem que irá ocasionar uma transição na máquina de estados finitos;

- O modelo de concorrência implementado pelos atores torna desnecessário o uso de qualquer estratégia de sincronização (como semáforos por exemplo), isso é possível devido a sua característica de processar somente uma mensagem por vez e possuir seu estado interno totalmente isolado;

- Um ator que possui filhos deve implementar uma estratégia de supervisão;

- Os atores comunicam-se exclusivamente através de endereços.

Se os atores da aplicação forem construídos seguindo estes princípios básicos, então não é necessária a utilização de estruturas de sincronização, como *locks* por exemplo, e é possível distribuir e escalar a aplicação simplesmente adicionando mais atores ao sistema ou criando novas instancias dos mesmos atores já existentes (VERNON, 2015).

3 METODOLOGIA

Os trabalhos de pesquisa que envolvem tópicos de computação têm por característica produzir alguma novidade, como um conhecimento (algoritmos, modelos, etc.) ou produto. Porém, para que a pesquisa seja realmente efetiva, deve ser adotada uma abordagem mais programática. Através do uso de métodos científicos é possível avaliar o produto do estudo de forma mais criteriosa (WAINER, 2007). O objetivo deste capítulo é apresentar o método científico empregado no desenvolvimento desta pesquisa com fim de atingir os objetivos estipulados para a mesma.

3.1 Delineamento de pesquisa

Esta pesquisa explora os possíveis benefícios de performance e escalabilidade que podem ser obtidos através da utilização de arquiteturas *event-driven* e hibridas na implementação de aplicações Web. Para atingir este objetivo, tornou-se necessária inicialmente a realização de um estudo buscando o entendimento dos modelos de concorrência existentes, bem como de seus componentes e blocos de construção. Neste contexto, esta pesquisa caracteriza-se segundo seu objetivo, como pesquisa exploratória.

A pesquisa exploratória consiste na realização de um estudo inicial sobre os temas relacionados ao projeto de pesquisa com o intuito de criar uma familiarização com os mesmos, e a partir disso permitir o desenvolvimento do trabalho. Este tipo de pesquisa sempre está relacionado com fontes já consolidadas de informação, como bibliografias e experiências práticas. (WAINER, 2007).

Os modelos explorados neste trabalho foram obtidos por meio de pesquisas em livros e artigos. Através da realização desta revisão da literatura foi obtida a base de conhecimento necessária para o entendimento dos modelos estudados. O uso deste método caracteriza o trabalho também como uma pesquisa bibliográfica.

A pesquisa bibliográfica refere-se ao levantamento e revisão da literatura existente sobre o um tema. Por intermédio da análise e discussão de várias contribuições cientificas, tanto teóricas quanto metodológicas, é construída a base do conhecimento sobre um determinado tópico. (PIZZANI, SILVA, BELLO, & HAYASHI, 2012).

Observar os possíveis benefícios de performance e escalabilidade associados a utilização de diferentes modelos de concorrência faz parte dos objetivos deste trabalho. Com isso, faz-se necessária a implementação de protótipos dos cenários a serem testados para permitir a realização de simulações em ambiente controlado com a finalidade de coleta de dados, caracterizando assim um experimento que se alinha com a natureza da pesquisa quantitativa.

A pesquisa quantitativa tem por objetivo garantir a precisão dos estudos realizados. Para isso o método tem base na comparação de resultados medidos e o uso intensivo de técnicas estatísticas. É muito comum o uso de dados sintéticos obtidos através da realização de experimentos, como por exemplo simulações em ambientes controlados ou artificiais (WAINER, 2007).

Esta pesquisa faz uso de um ambiente de laboratório no sentido de que as simulações de carga que serão executadas sobre os protótipos criados seguirão determinados padrões e condições pré-estabelecidas. A coleta de dados necessária para validação da proposta se dará através do uso de um ambiente artificial. Este ambiente irá permitir a simulação dos altos níveis de concorrência desejados, forçando assim situações onde os modelos selecionados para a implementação dos protótipos apresentem discrepâncias mais acentuadas.

A pesquisa de laboratório tem por objetivo buscar, descrever e analisar fenômenos em ambientes controlados (MARCONI & LAKATOS, 2003). Muitas vezes os fenômenos que precisam ser observados não apresentam padrões que podem ser controlados de forma adequada, ou não atingem o volume necessário para validação de um resultado. Devido a isso, para permitir a captação adequada de informações relevantes para a pesquisa, os fenômenos são reproduzidos artificialmente de forma controlada (SANTOS, 1999).

3.2 Próximas etapas

Nos próximos capítulos estão detalhadas atividades referentes a execução pratica e documentação deste estudo. Dentre as etapas estão:

- **Definição dos cenários**: Nesta etapa são definidos os cenários sob os quais serão gerados os protótipos para a execução de testes;
- **Definição das métricas**: Nesta etapa são definidas as métricas coletadas durante a execução dos testes;
- **Definição dos modelos de concorrência**: Esta etapa apresenta os modelos de concorrência que serão utilizados na implementação dos protótipos;
- **Definição e implementação das simulações de carga**: Nesta etapa estão especificadas as simulações que vão ser executadas em cada um dos cenários propostos;
- **Definição das ferramentas utilizadas**: Esta etapa consiste em definir as ferramentas utilizadas na implementação dos protótipos, dos cenários de simulação e na coleta dos resultados, detalhando sua funcionalidade e utilização dentro deste estudo;
- **Definição da arquitetura e implementação dos protótipos**: Nesta etapa estão descritas as arquiteturas que são utilizadas de base para a implementação de todos os protótipos em ambos os modelos de concorrência selecionados para avaliação. Os detalhes específicos de implementação de cada um dos protótipos também estão documentados nesta etapa;
- **Definição e configuração dos ambientes de teste:** Nesta etapa estão documentadas as configurações utilizadas nas maquinas virtuais no Google Cloud;
- **Definição dos procedimentos**: Nesta etapa estão definidos os procedimentos de execução e coleta dos resultados;

- **Execução das simulações e coleta de dados**: Esta etapa consiste na execução das simulações e coleta das métricas necessárias para o estudo;

- **Análise dos resultados**: Nesta etapa estão apresentados os resultados obtidos para cada um dos cenários em conjunto com uma análise comparativa entre os resultados obtidos em ambas as implementações;

- **Considerações finais**: Nesta etapa estão redigidas as conclusões obtidas através da realização deste estudo.

4 ESTUDO REALIZADO

Neste capítulo serão apresentados os modelos de concorrência selecionados para a implementação dos protótipos dos cenários de testes juntamente com a explicação do porquê de sua escolha para a realização deste estudo. Também serão demonstrados os cenários propostos e as métricas que serão coletadas nas simulações.

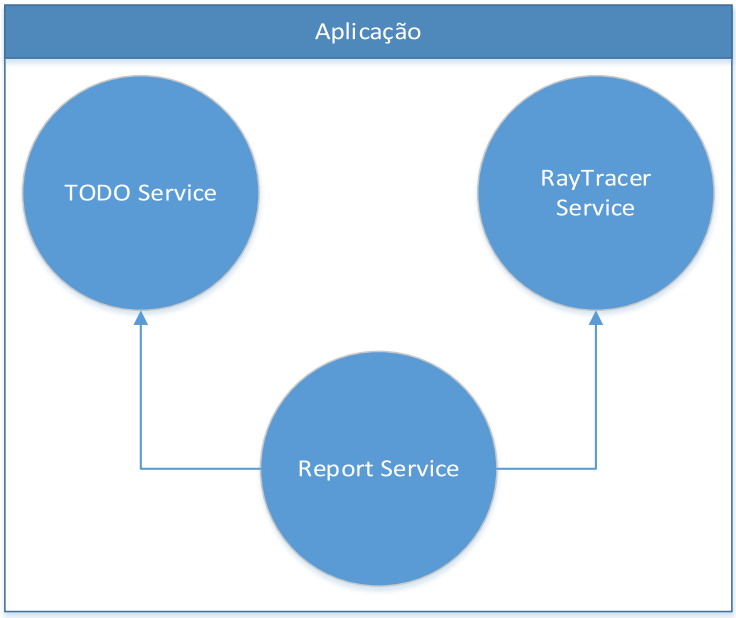
4.1 Cenário proposto

Como a proposta deste trabalho consiste em avaliar os modelos em aplicações Web, o cenário que será construído inclui arquiteturas que tem se popularizado atualmente na construção de sistemas altamente modulares e concorrentes.

A proposta é desenvolver uma pequena aplicação composta de três *microservices*, conforme demonstrado na Figura 20. Cada *microservice* vai disponibilizar uma API para comunicação via protocolo HTTP seguindo os princípios da arquitetura REST. O objetivo da utilização de *microservices* no cenário é a possibilidade de avaliar ambas as implementações tanto de forma individual como na cooperação com outros serviços.

Antes de explicar qual o papel de cada *microservice* dentro do cenário é importante ressaltar que estas aplicações não possuem nenhum objetivo quanto a requisitos de negócio, ou seja, são simplesmente implementações abstratas que simulam processos que ocorrem em aplicações reais.

Figura 20 – Arquitetura do cenário proposto



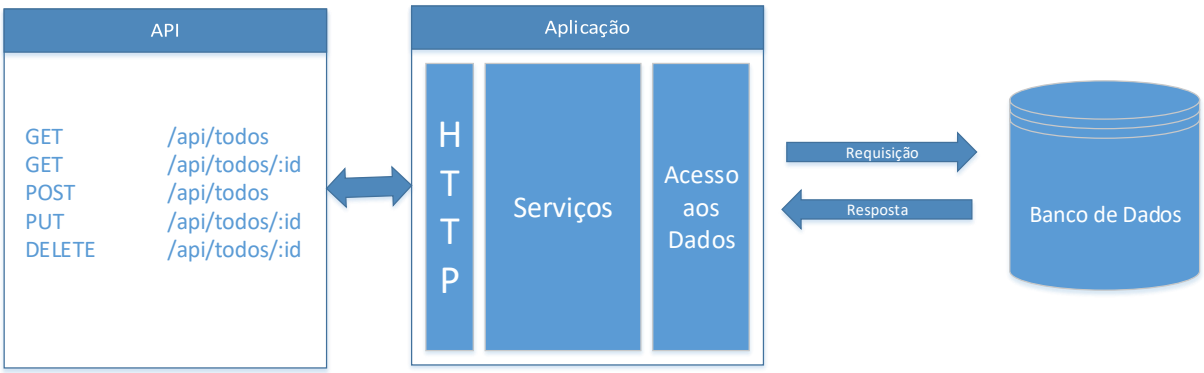
Fonte: Autor.

4.1.1 TODO Service

O **TODO Service** é uma aplicação que consiste em armazenar uma lista de tarefas a serem realizadas. As tarefas vão conter algumas informações básicas como título, descrição, data de criação e data de fechamento. O objetivo desta aplicação é a realização de operações básicas de cadastro, leitura, atualização e remoção (*Create, Read, Update and Delete,* CRUD) em um banco de dados relacional tradicional (*Relational Database Management System*, RDBMS) tradicional. A Figura 21 demonstra a arquitetura deste *microservice*.

Este modelo simula o tipo mais comum de aplicação Web. Este tipo de aplicação tem por característica um uso não tão intenso do processador e a necessidade de executar uma grande quantidade de operações de E/S para realizar a comunicação com o cliente, acesso ao banco de dados e sistema de arquivos.

Figura 21 – Arquitetura do TODO Service



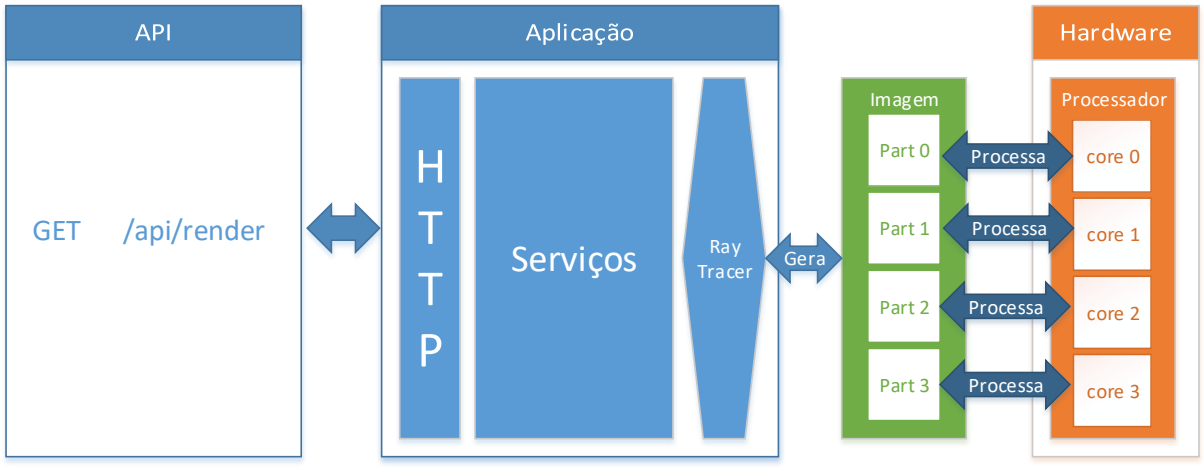
Fonte: Autor.

4.1.2 RayTracer Service

O **RayTracer Service** consiste em uma aplicação que renderiza uma imagem utilizando uma técnica chamada *Ray Tracing*. Este processo pode utilizar paralelismo cooperativo, ou seja, pode aproveitar os núcleos de processamento disponíveis para processar partes distintas da imagem. A Figura 22 demonstra a arquitetura deste *microservice* e a Figura 23 demonstra um resultado aproximado do processamento realizado.

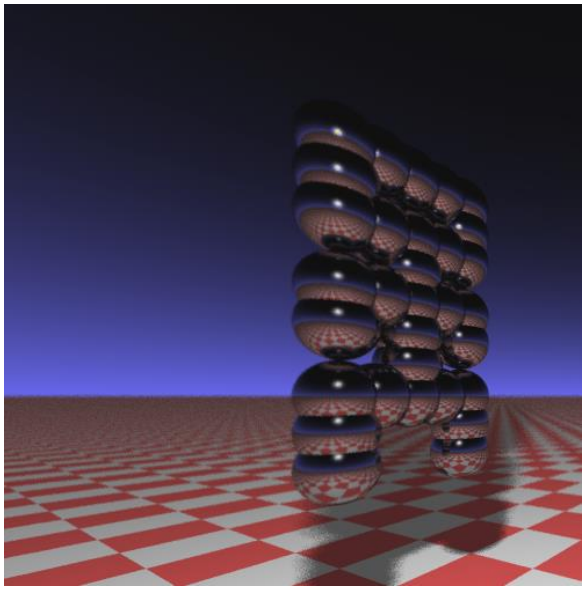
O objetivo com isto é simular serviços que possuam naturezas altamente bloqueantes devido a necessidade de computação intensiva, ou seja, dependem puramente de processamento bruto e não de operações de E/S.

Figura 22 – Arquitetura do RayTracer Service



Fonte: Autor.

Figura 23 – Imagem resultante do processo do RayTracer Service



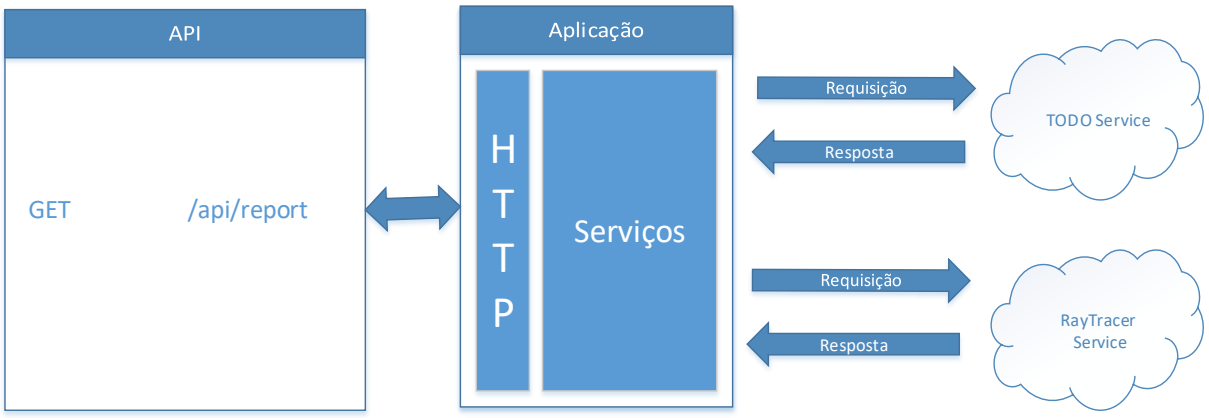
Fonte: Autor.

4.1.3 Report Service

O **Report Service** é uma aplicação que consome a API disponibilizada pelos outros dois *microservices* e produz um relatório contendo as informações obtidas dos mesmos. A Figura 24 demonstra a arquitetura deste *microservice.*

O objetivo é simular aplicações que realizam pouco processamento, porém que possuem uma alta dependência de processos de E/S, como *sockets*, manipulação de arquivos e acesso a outras aplicações.

Figura 24 – Arquitetura do Report Service



Fonte: Autor.

4.2 Métricas coletadas

Como o estudo trata especificamente de aplicações Web, todas as métricas estão associadas a requisições dos clientes para as aplicações. As métricas de interesse que serão coletadas para a realização do estudo estão relacionadas com:

- Tempo de resposta;
- *Throughput;*
- Escalabilidade.

É importante destacar que a escalabilidade que será observada no estudo é a capacidade da aplicação em acomodar um alto volume de trabalho, ou seja, não envolve alteração do hardware e conceitos de escalabilidade horizontal ou vertical. Esta métrica será avaliada baseado na capacidade da aplicação em manter o *throughput* conforme a carga dos testes for aumentando.

Durante a execução das simulações vão ser coletadas várias métricas, algumas relacionadas ao desempenho da aplicação em si e outras relacionadas ao consumo de recursos do sistema. Mais especificamente, serão coletadas as seguintes métricas:

- Tempo de resposta mínimo;
- Tempo de resposta máximo;
- Tempo de resposta médio;
- Uso do processador pela aplicação;
- Memória utilizada pela aplicação;
- Número de *threads* alocadas durante a simulação.

4.3 Modelos de concorrência selecionados

Foram selecionados para a implementação dos protótipos deste trabalho o modelo *multithread* e o modelo de atores. Além dos motivos que serão apresentados a seguir, estes dois modelos foram escolhidos por uma razão particularmente interessante: eles representam níveis de abstração distintos.

Na implementação utilizando modelo *multithread* quase sempre entramos em contato com primitivas de baixo nível como *threads* e *locks.* Já no modelo de atores trabalhamos simplesmente com a troca de mensagens, delegando a responsabilidade pelo gerenciamento de *threads* e trocas de contexto para alguma biblioteca que implementa o modelo em questão.

4.3.1 Multithread

O modelo *multithread* foi escolhido por ser o mais simples e comumente encontrado nas aplicações Web existentes. Portanto, representa a linha a ser batida, tanto em performance e escalabilidade quanto em complexidade e manutenibilidade. A complexidade é um ponto importante a ser considerado, pois coordenar um processo utilizando várias *threads* é uma tarefa difícil e altamente propensa a erros.

4.3.2 Modelo de atores

O modelo de atores foi escolhido pois representa, nas implementações de bibliotecas existentes atualmente, um modelo hibrido entre o uso de *threads* e o uso de arquiteturas *event-driven* baseadas em uma única *thread*. Outro ponto positivo deste modelo é o foco na troca de mensagens que fornece uma abstração de alto nível para se trabalhar com concorrência. A comunicação estritamente assíncrona entre os atores assemelha-se bastante ao modelo *event-driven*, porém é trivial delegar tarefas que exijam operações bloqueantes para *thread pools* dedicados. Outra razão para sua escolha é devido a sua crescente popularidade no desenvolvimento de aplicações que requisitam alto grau de responsividade, como jogos online por exemplo.

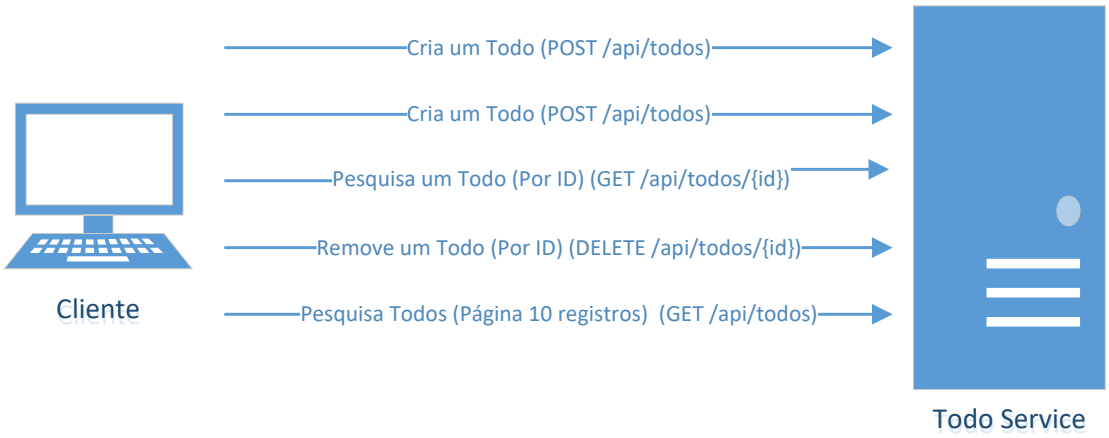
4.4 Simulações de carga

Neste subcapítulo serão apresentados os cenários de simulação que serão aplicados em cada um dos protótipos, bem como a carga a qual serão submetidas as aplicações sob teste.

4.4.1 TODO Service

A simulação aplicada no **TODO Service** consiste em um fluxo de uso do usuário dentro do sistema. Cada usuário realiza um total de cinco operações, sendo duas operações de inserção, uma consulta buscando um único registro por seu identificador, uma remoção e uma pesquisa por uma página de resultados, estas ações estão exemplificadas na Figura 25.

Figura 25 – Simulação TODO Service



Fonte: Autor.

Para esta simulação foi configurada uma carga de 1000 usuários simultâneos, resultando em um total de 5.000 requisições para a aplicação.

4.4.2 RayTracer Service

Na simulação aplicada no **RayTracer Service** cada usuário realiza uma chamada para a *Uniform Resource Identifier*[7](URI) **/api/render** utilizando o método HTTP **GET**. Para esta

[7] URI refere-se a um conjunto de caracteres utilizados para representar um recurso.

simulação foi configurada uma carga de 50 usuários simultâneos, resultando em um total de 50 requisições para a aplicação.

4.4.3 Report Service

Na simulação aplicada no **Report Service** cada usuário realiza uma chamada para a URI **/api/report** utilizando o método HTTP **GET**. Para esta simulação foi configurada uma carga de 50 usuários simultâneos, resultando em um total de 50 requisições para a aplicação.

4.5 Ferramentas utilizadas

Neste subcapítulo serão apresentadas as tecnologias e ferramentas utilizadas no desenvolvimento dos protótipos, criação dos ambientes e cenários de teste.

4.5.1 Google Cloud

O Google Cloud é uma plataforma de computação em nuvem oferecida pelo Google que dispõem de inúmeras ferramentas e serviços que vão desde de servidores até aplicações como banco de dados e inteligência artificial. A infraestrutura disponível é a mesma na qual o Google roda seus próprios serviços como o por exemplo o Google Search e o YouTube (GOOGLE, 2016).

No escopo deste trabalho o Google Cloud Compute Engine será utilizado para criar as maquinas virtuais necessárias para a execução das simulações. A plataforma permite a escolha de várias configurações de hardware predefinidas e também algum nível de customização de processamento e memória caso necessário.

4.5.2 PostgreSQL

Surgido nos anos 90 como uma evolução do projeto Ingres na Universidade de Berkeley (Califórnia), o PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (ORDBMS) de código aberto com foco na extensibilidade e compatibilidade com os padrões

SQL. Possui suporte a vários recursos avançados como por exemplo herança de tabelas, replicação síncrona e assíncrona e *window functions* (POSTGRESQL, 2016).

É um software robusto e escalável que fornece as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID). O gerenciamento da concorrência no banco é realizado através de um sistema chamado *Multiversion Concurrency Control* (MVCC) que provê uma imagem do último estado valido para operações de consulta, eliminando a necessidade de *locks* para operações de leitura, permitindo assim que o banco mantenha as propriedades ACID de maneira mais eficiente (POSTGRESQL, 2016).

Neste trabalho o PostgreSQL será utilizado para armazenamento de dados do protótipo **TODO Service**.

4.5.3 Java

Criado por James Gosling, Mike Sheridan e Patrick Naughton nos anos 90, o Java é uma linguagem de programação compilada, orientada a objetos, fortemente tipada e interpretada. A compilação de um código em Java gera um código intermediário (*bytecode)* que é executado em uma máquina virtual chamada *Java Virtual Machine* (JVM), devido a isso, o *bytecode* gerado pelo processo de compilação pode ser executado em qualquer plataforma suportada por uma JVM (SCHILDT, 2005).

Em suas versões iniciais o Java sofria com problemas de performance devido a ser uma linguagem interpretada. Para melhorar a performance de execução do *bytecode* foram criados mecanismos para detecção de pontos críticos na execução dos programas (*hotspots)* e transformação do *bytecode* para código nativo da plataforma, este mecanismo é conhecido como *Just In Time Compiler* ou JIT. Além disto, durante a execução de um programa as maquinas virtuais podem aplicar diversas otimizações como por exemplo *method inlining* e *loop unrolling* (SCHILDT, 2005).

O Java será utilizado para implementar todos os protótipos propostos no trabalho. Também serão utilizadas ferramentas disponíveis no *Software Development Kit* (SDK) da plataforma para realizar o monitoramento do uso de processador e memória durante a execução dos testes.

4.5.4 Scala

Criada por Martin Odersky nos anos 2000, Scala é uma linguagem fortemente tipada com suporte a múltiplos paradigmas (orientação a objetos e programação funcional). O processo de compilação da linguagem gera o mesmo código intermediário que o Java (*bytecode*), isso permite que seja executado pela JVM e também possibilita a interoperação com o Java e outras linguagens que rodam sobre a JVM. Com isso a linguagem beneficia da infraestrutura e da enorme gama de bibliotecas já existentes que rodam sobre a JVM (ODERSKY, 2014).

O Scala tem ganho mercado devido a sua expressividade aliada a boa performance (muito simular ao Java) e bons recursos para se trabalhar com programação concorrente e assíncrona. A linguagem já vem sendo adotada por algumas grandes empresas como o Twitter e é utilizada em alguns projetos de código aberto de grande porte como o Apache Spark e o Akka (ODERSKY, 2014).

Dentro do contexto deste trabalho, o Scala é utilizado em algumas dependências dos protótipos e também nas implementações dos *scripts* de simulação.

4.5.5 Spring Framework

Spring é um *framework* de injeção de dependências e integração para Java. Através dos anos foram integradas diversas ferramentas e também desenvolvidos diversos módulos próprios, como por exemplo Spring MVC para desenvolvimento Web e o Spring Data para facilitar o acesso a dados (WALLS, 2015).

Neste trabalho o Spring será utilizado na implementação dos protótipos no modelo *multithread*. O *framework* será utilizado em conjunto com algumas tecnologias padrão do Java, como as *Servlets* para HTTP e o *Java Database Connectivity* (JDBC) para acesso a banco de dados.

4.5.6 Akka

O Akka é um *framework* escrito em Scala visando simplificar a criação de aplicações altamente concorrentes e distribuídas na JVM. Suporta vários modelos de programação concorrente, porém seu foco é a utilização do modelo de atores como primitivas para concorrência. Deste modo, a concorrência no *framework* é baseada em comunicação por troca de mensagens imutáveis através de canais assíncronos (VERNON, 2015).

A comunicação entre os componentes é realizada através de rotas (como um sistema de arquivos por exemplo) e, devido a isso, detalhes de implementação como *threads* e *clusters* são abstraídos em configurações, possibilitando assim escalar a aplicação de forma transparente conforme a necessidade e capacidade dos servidores utilizados (VERNON, 2015).

O Akka é utilizado no trabalho na implementação dos protótipos utilizando o modelo de atores. Também é utilizado seu modulo experimental para trabalhar com HTTP substituindo assim o padrão de *Servlets* amplamente utilizado nos *frameworks* construídos com Java.

4.5.7 Gatling

Lançada em 2011, o Gatling é uma ferramenta de código aberto destinada a criação e execução de testes de performance em aplicações Web. A ferramenta é escrita em Scala e possui uma API bem simples e abrangente para executar chamadas via HTTP, o que permite a criação de cenários de teste para aplicações Web de maneira muito rápida (TOLLEDO, 2014).

Neste trabalho o Gatling será utilizado para implementar e executar todos os cenários de teste escritos para os protótipos, e também para realizar a coleta de informações relacionadas a performance das aplicações.

4.6 Arquitetura

Todos os protótipos implementados compartilham da mesma arquitetura base. A linguagem escolhida para realizar a implementação foi o Java devido a sua popularidade e a ampla gama de opções em relação a ferramentas para a implementação dos cenários propostos.

Para os protótipos que necessitam de persistência de dados foi utilizado como banco de dados o PostgreSQL

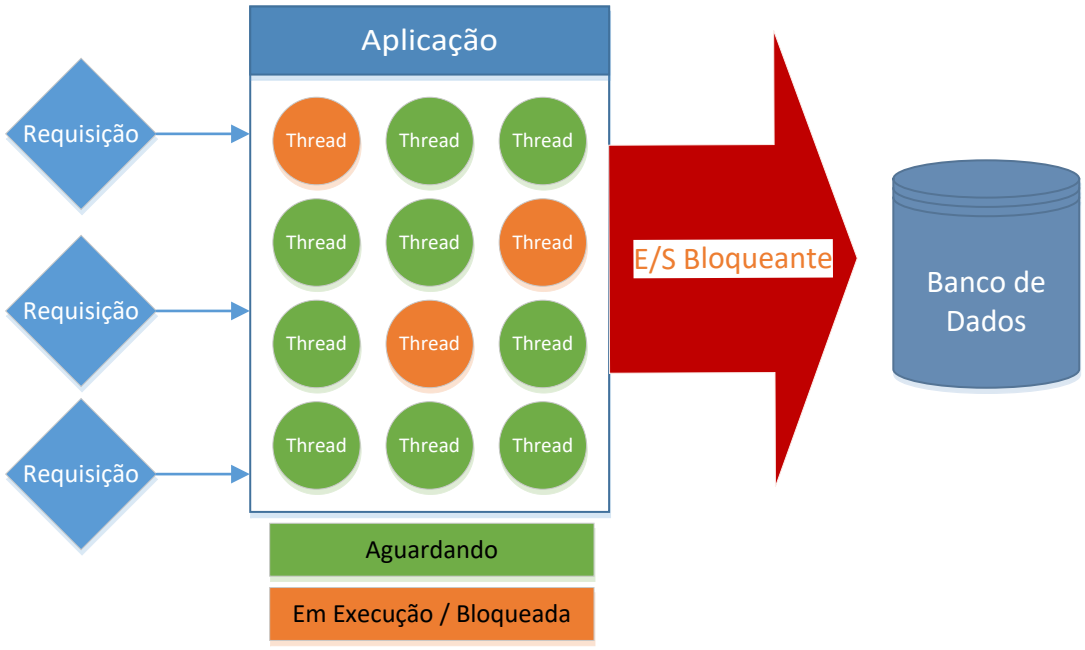
O modelo *multithread* utiliza como base as tecnologias que estão disponíveis por padrão na plataforma do Java. Para comunicação HTTP é utilizado o padrão *Servlet* e o acesso a banco de dados faz uso do *Java Database Connectivity* (JDBC), que é o padrão para acesso a bancos de dados relacionais no Java. Todas estas tecnologias são abstraídas através da utilização do Spring Framework, mais especificamente dos módulos Spring MVC e Spring JDBC.

O Spring MVC, através da classe **DispatcherServlet**, é responsável por receber as requisições, identificar a URI e o método HTTP definidos, e rotear a requisição para o método da aplicação responsável por trata-la, extraindo as informações solicitadas pelo método caso necessário. Estas informações podem ser parâmetros, objetos codificados em algum formato (como por exemplo *JavaScript Object Notation* (JSON), XML, etc.) e informações de contexto (como por exemplo o usuário que disparou a requisição). Após a execução do método e obtenção do resultado, o *framework* também é responsável por codificar a resposta no formato solicitado na requisição.

Para realizar a integração com o Spring MVC, as classes da aplicação que precisam trabalhar com requisições feitas via Web definem junto a seus métodos algumas anotações com o objetivo de definir a URI e método HTTP sob os quais vão atuar. Nos protótipos implementados, todas as classes que fazem uso destas anotações receberam em seu nome o sufixo **Resource** para facilitar sua identificação.

A utilização das tecnologias padrões da plataforma Java implica em seguir um modelo de programação sequencial, pois o retorno do resultado das operações está acoplado diretamente na chamada dos métodos disponibilizados pelas APIs. Devido à ausência de abstrações na obtenção dos resultados, torna-se impossível a realização de chamadas assíncronas para estas APIs sem a alocação de uma *thread* que fique bloqueada aguardando o resultado do processamento. A Figura 26 demonstra o fluxo de processamento de uma requisição através da arquitetura.

Figura 26 – Fluxo de execução na arquitetura *multithread*



Fonte: Autor.

Em contraste, a arquitetura baseada no modelo de atores é altamente focada em troca de mensagens, uso de filas e operações de E/S assíncronas. A arquitetura implementada neste modelo é baseada na ferramenta Akka e alguns de seus módulos. Para comunicação HTTP é utilizado um módulo ainda experimental do Akka, o Akka HTTP, e o acesso ao banco de dados é realizado utilizando um *driver* de código aberto chamado Postgres Async Driver[8] que suporta comunicação assíncrona com o PostgreSQL.

A integração da aplicação com o Akka HTTP é realizada de forma mais programática. Os pontos de integração são realizados através da definição de instancias da interface funcional **Route** (interface com um único método responsável por processar a requisição) para as URIs e métodos HTTP desejados. Um dos pontos positivos do Akka HTTP é o seu extensivo suporte para geração de respostas através de computações assíncrona e também sua flexibilidade, já que basta gerar uma instancia que implemente a interface **Future** para integrar qualquer com qualquer modelo de computação.

Neste modelo de programação é comum encontrar no retorno dos métodos abstrações representando resultados assíncronos ao invés de um resultado direto como no modelo anterior.

[8] Disponível em: <https://github.com/alaisi/postgres-async-driver>. Acesso em outubro 2016.

Algumas das abstrações mais comuns e que serão vistas e utilizadas na implementação dos protótipos neste modelo são:

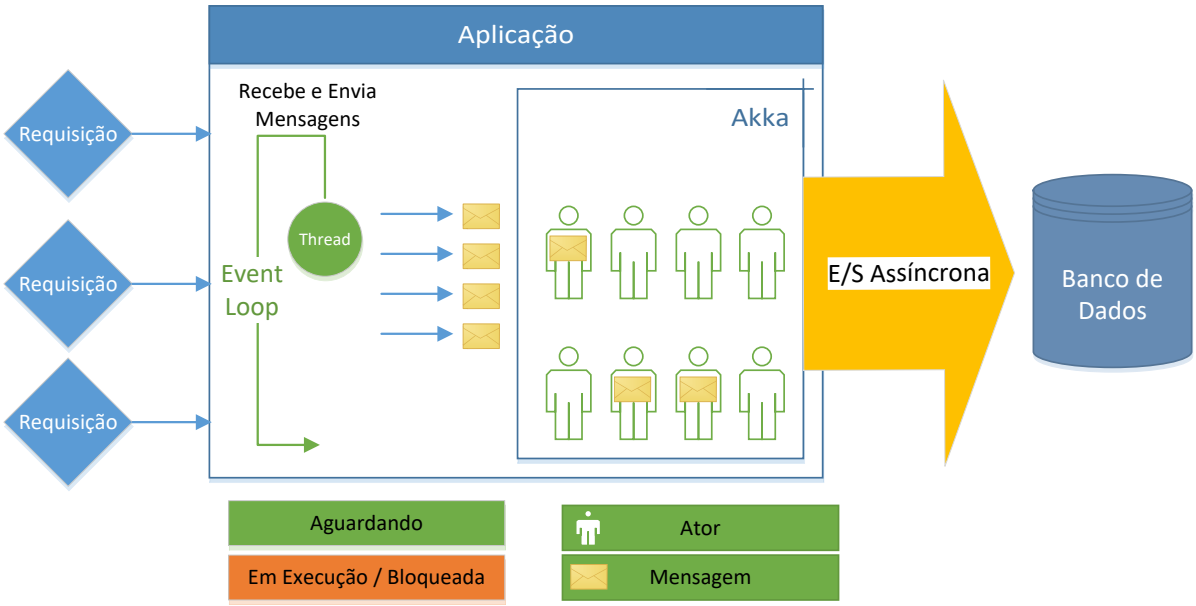
- **Future**: Representa o resultado de uma computação assíncrona e disponibiliza inúmeros métodos para consultar, consumir, compor e transformar o resultado gerado;

- **Observable**: Representa uma fonte de dados observável que pode emitir um ou mais elementos. Esta abstração permite que o cliente se inscreva para receber notificações quando alguma informação nova estiver disponível para ser processada.

Estas abstrações formam a base para a definição de um fluxo de processamento assíncrono, porém é importante observar que elas são totalmente independentes do contexto de execução, ou seja, também poderiam ser utilizadas para realizar um processamento síncrono.

O Akka faz uso exclusivamente de troca de mensagens, ou seja, não depende de nenhuma abstração especifica, porém elas são muito uteis para interagir com códigos e bibliotecas que não seguem o mesmo modelo de programação e que, devido a isso, não podem ser integradas diretamente com o sistema de mensagem dos atores. A Figura 27 demonstra o fluxo de processamento de uma requisição através deste modelo de arquitetura.

Figura 27 – Fluxo de execução na arquitetura baseada em atores



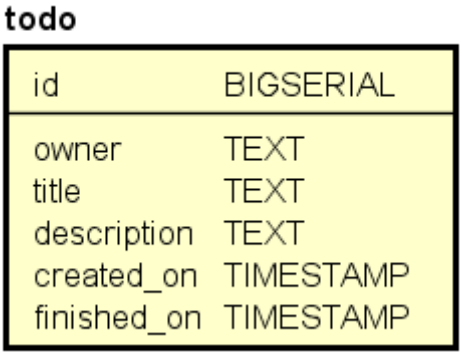
Fonte: Autor.

4.6.1 TODO Service

Como descrito anteriormente, este *microservice* tem por objetivo simular um ambiente onde ocorram operações de CRUD e é o único dos protótipos que realiza a persistência dos dados em um banco de dados relacional. Na Figura 28 está ilustrado o modelo Entidade Relacionamento (ER) da estrutura de dados criada no banco de dados PostgreSQL com o objetivo de armazenar as informações mantidas pela aplicação, as colunas representam:

- **Id**: Chave primaria da tabela, utilizada para identificar cada registro;
- **Owner**: Campo utilizado para identificar o nome do usuário que está vinculado ao item;
- **Description**: A descrição da tarefa a ser realizada;
- **Created_on**: Campo utilizado para identificar a data de criação do registro no sistema;
- **Finished_on**: Campo utilizado para identificar a data de finalização de uma tarefa.

Figura 28 – Modelo relacional do TODO Service



Fonte: Autor.

O uso de um banco de dados externo implica na necessidade de criação de conexões com este serviço, na implementação de ambos os modelos são utilizados *connection pools*[9] para trabalhar com o banco de dados, mantendo sempre um total de 8 conexões abertas para a realização de operações relacionadas a manipulação e consulta dos dados.

4.6.1.1 Modelo multithread

Nesta implementação todas as chamadas que dependem de operações de E/S são realizadas de maneira síncrona, ou seja, a *thread* que executa operação fica bloqueada até a finalização da mesma.

O domínio da aplicação é representado pela classe **Todo** que armazena informações referentes a tarefas a serem executadas, bem como se elas foram executadas e a data de sua execução. Ela representa uma relação direta com o modelo ER apresentado anteriormente.

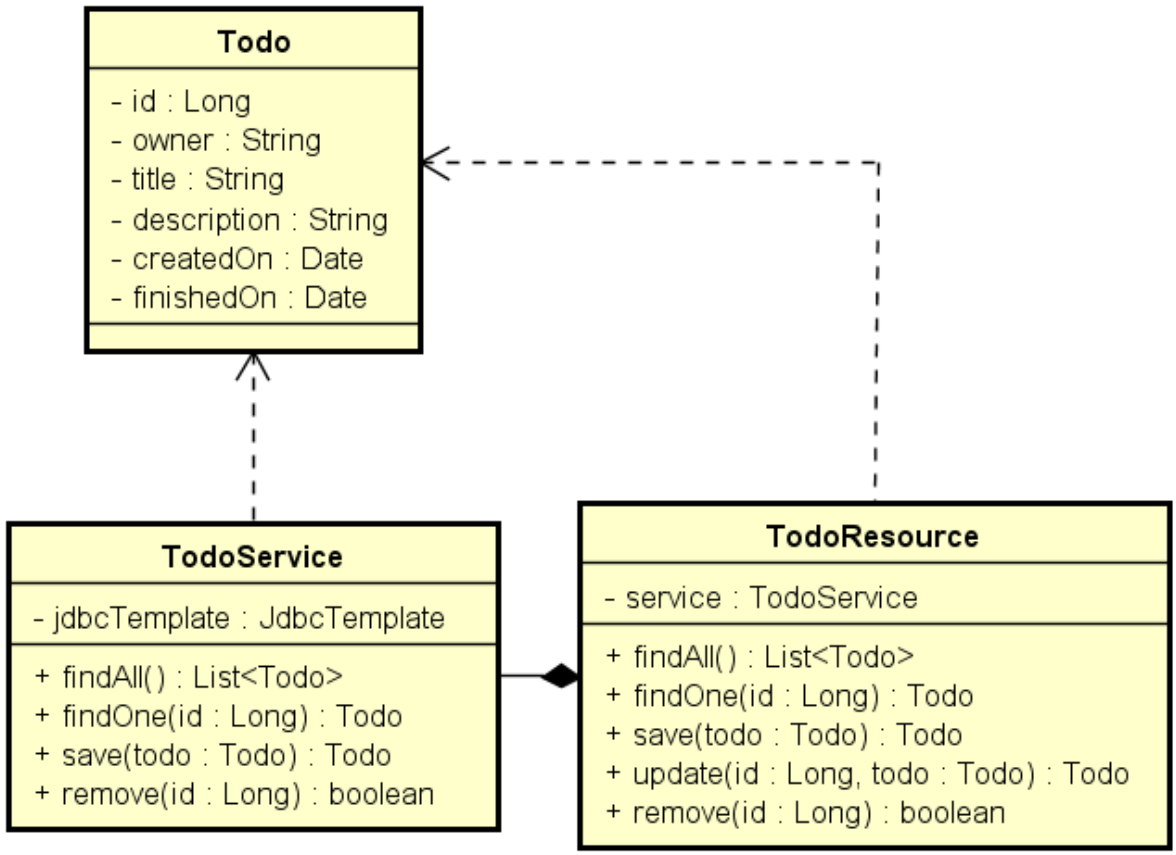
A classe **TodoService** é responsável por prover e conectar os serviços necessários para permitir o gerenciamento da classe de negócio **Todo**. Esta classe mantém encapsulados detalhes de implementação relacionados a persistência dos dados. Neste caso, a persistência é realizada no banco de dados relacional PostgreSQL fazendo o uso de um driver que segue o padrão JDBC através da classe **JdbcTemplate**.

O **JdbcTemplate** é uma classe de conveniência disponibilizada no módulo Spring JDBC do *framework* Spring que trabalha em cima do padrão JDBC. Suas principais funcionalidades são o gerenciamento de recursos (conexões com o banco de dados) de maneira automática, controle transacional integrado com o *framework* e uma API mais concisa.

A classe **TodoResource** é responsável por expor as operações definidas na classe de serviços **TodoService**, bem como por validar os dados enviados nas requisições efetuada pelos clientes. Ela expõe seus métodos para a Web através de anotações onde cada um dos métodos define a URI e método HTTP pelo qual é responsável. Na Figura 29 estão representadas as classes e relações que compõem o funcionamento do serviço no modelo *multithread.*

[9] Conjunto de conexões com o banco de dados mantidas abertas por uma aplicação com o objetivo de melhorar a performance através da reutilização das mesmas.

Figura 29 – Diagrama de classes – TODO Service – Modelo *multithread*

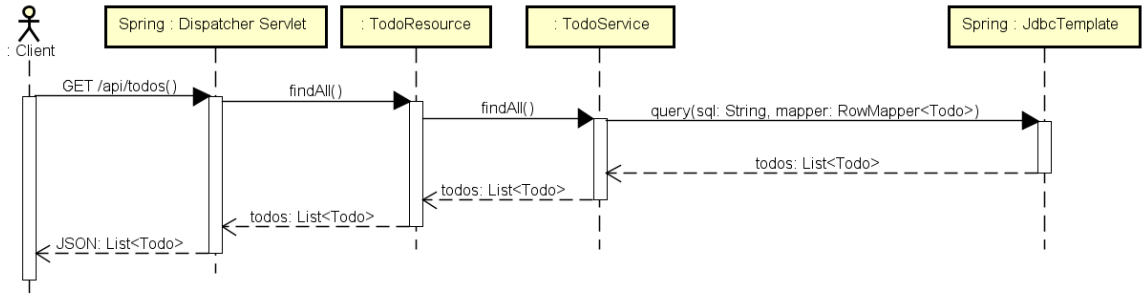


Fonte: Autor.

O diagrama de sequência representado na Figura 30 ilustra uma interação com o método **findAll** da classe **TodoResource**, que tem por objetivo de retornar os dez primeiros itens contidos na tabela **todo**. Este método é invocado quando é recebida uma requisição utilizando o método HTTP **GET** para a URI **/api/todos**. Esta chamada então é delegada para o método **findAll** da classe **TodoService** que invoca o método **query** da classe **JdbcTemplate**, este último será responsável por buscar registros através de uma consulta em *Structured Query Language*[10] (SQL) e, posteriormente, transformar os resultados para objetos do tipo **Todo** que serão serializados para o formato JSON e enviados como resposta.

[10] Linguagem de programação utilizada para gerenciamento de dados em bancos de dados relacionais.

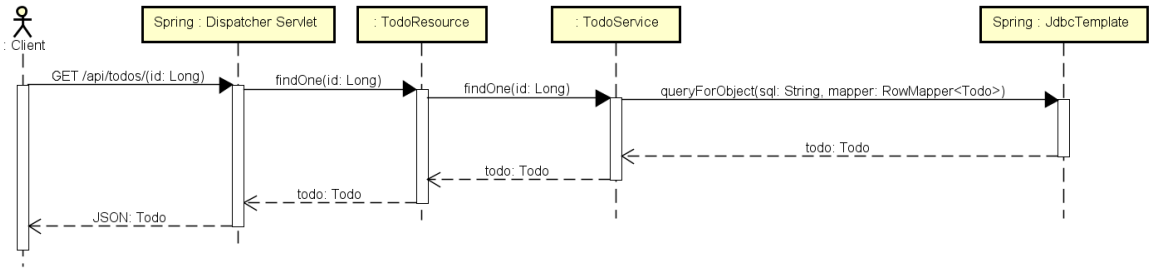
Figura 30 – Diagrama de sequência – TODO Service – Modelo *multithread* – Método findAll



Fonte: Autor.

O diagrama de sequência representado na Figura 31 ilustra uma interação com o método **findOne** da classe **TodoResource**, que tem por objetivo retornar um único item da tabela **todo** identificado por sua chave primaria (campo **id** neste caso). Este método é invocado quando é recebida uma requisição utilizando o método HTTP **GET** para a URI **/api/todos/{id}**. Esta chamada então é delegada para o método **findOne** da classe **TodoService** que utiliza o método **queryForObject** da classe **JdbcTemplate** para busca o registro através de uma consulta em SQL. Após a obtenção do resultado, o mesmo é transformado um objeto do tipo **Todo**, serializado para o formato JSON e enviado como resposta (caso o mesmo seja encontrado).

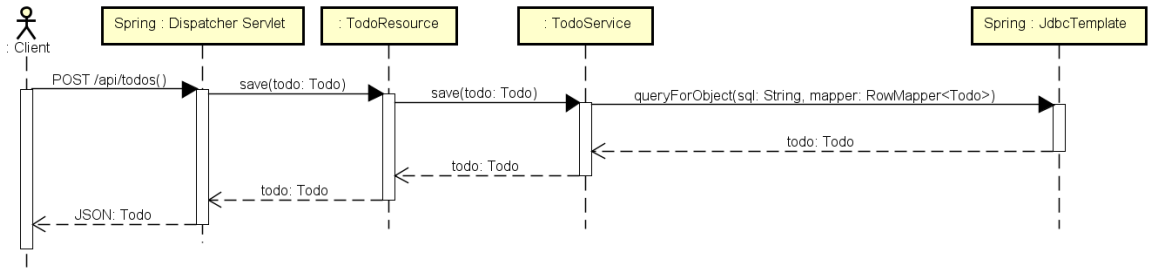
Figura 31 – Diagrama de sequência – TODO Service – Modelo *multithread* – Método findOne



Fonte: Autor.

O diagrama de sequência representado na Figura 32 ilustra uma interação com o método **save** da classe **TodoResource**, que tem por objetivo criar um novo item na tabela **todo**. Este método é invocado quando é recebida uma requisição utilizando o método HTTP **POST** para a URI **/api/todos** trazendo o conteúdo do objeto **Todo** em formato JSON no corpo da requisição. Esta chamada então é delegada para o método **save** da classe **TodoService** que utiliza o método **queryForObject** da classe **JdbcTemplate** para executar a inserção dos dados em SQL e retornar o identificador gerado para o novo registro. Após a persistência e atualização do identificador no objeto, o mesmo é serializado para o formato JSON e enviado como resposta.

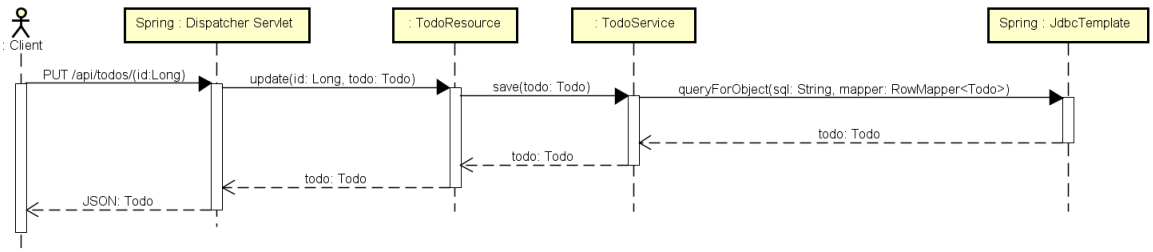
Figura 32 – Diagrama de sequência – TODO Service – Modelo *multithread* – Método save



Fonte: Autor.

O diagrama de sequência representado na Figura 33 ilustra uma interação com o método **update** da classe **TodoResource**, que tem por objetivo atualizar um item na tabela **todo**. Este método é invocado quando é recebida uma requisição utilizando o método HTTP **PUT** para a URI **/api/todos/{id}** trazendo o conteúdo do objeto **Todo** em formato JSON no corpo da requisição. O método verifica a existência do objeto e invoca método **save** da classe **TodoService** que utiliza o método **queryForObject** da classe **JdbcTemplate** para executar a atualização dos dados via SQL. Após a persistência, o mesmo é serializado para o formato JSON e enviado como resposta ao cliente.

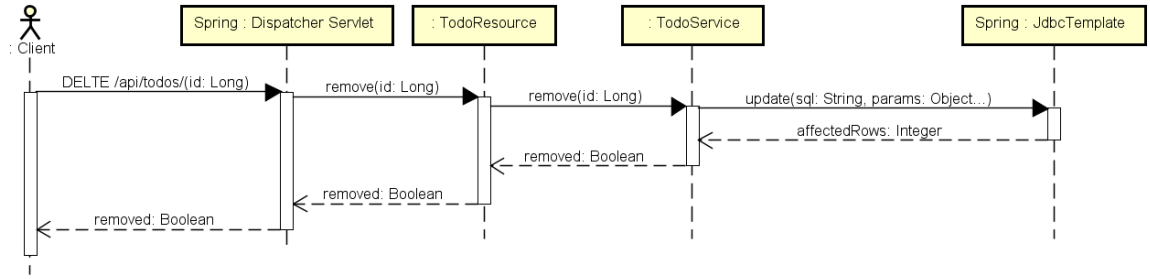
Figura 33 – Diagrama de sequência – TODO Service – Modelo *multithread* – Método update



Fonte: Autor.

O diagrama de sequência representado na Figura 34 ilustra uma interação com o método **remove** da classe **TodoResource**, que tem por objetivo remover um item da tabela **todo** identificado por sua chave primaria. Este método é invocado quando é recebida uma requisição utilizando o método HTTP **DELTE** para a URI **/api/todos/{id}**. Esta chamada é delegada para o método **remove** da classe **TodoService** que utiliza o método **update** da classe **JdbcTemplate** para executar a remoção do item através de um comando em SQL. O resultado é retornado para o cliente indicando se o registro foi removido ou não.

Figura 34 – Diagrama de sequência – TODO Service – Modelo *multithread* – método remove



Fonte: Autor.

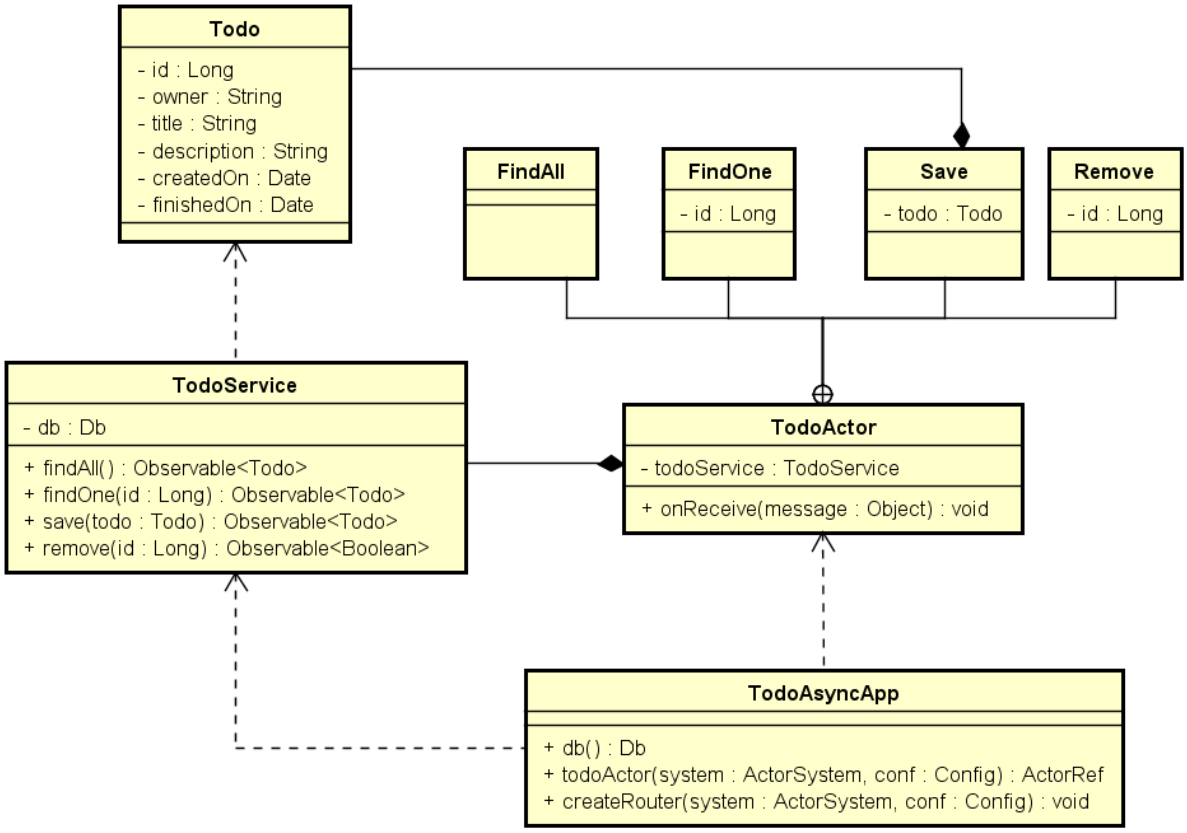
4.6.1.2 Modelo de atores

Na Figura 35 estão representadas as classes que compõem o funcionamento do serviço no modelo de atores. Nesta implementação todas as chamadas que dependem de operações de E/S são realizadas de maneira assíncrona e não bloqueante, ou seja, uma mesma *thread* pode multiplexar diversas operações de E/S simultaneamente.

A classe **Todo** tem exatamente a mesma função descrita no modelo anterior, porém, neste modelo ela é tratada como uma classe imutável, ou seja, os valores contidos nas propriedades da instancia são definidos no momento de sua construção e não podem ser mais alterados. Caso seja necessária a alteração de algum valor, é necessária a criação de uma nova instancia contendo a alteração.

À primeira vista a imutabilidade pode parecer uma complexidade e custo de processamento desnecessário, porém sua utilização traz diversos benefícios (além de ser imprescindível para o uso do modelo de atores) como por exemplo a simplificação na criação de mecanismos de *cache*, operações de comparações simplificadas (para garantir que uma instancia é igual a outra, basta comparar a referência em memória, não sendo necessária a comparação de suas propriedades) e principalmente por não serem necessárias a utilização de mecanismos de sincronização (como *locks* por exemplo).

Figura 35 – Diagrama de classes – TODO Service – Modelo de atores



Fonte: Autor.

A classe **TodoService** mantém a mesma função descrita no modelo anterior, porém a diferença neste modelo é a forma como as operações são executadas. Esta classe possui uma instancia da classe **Db** que representa o acesso ao banco de dados utilizando o *driver* de código aberto descrito anteriormente. As operações realizadas por esta classe retornam instancias do tipo **Observable** que, conforme descrito anteriormente, representam computações assíncronas.

A classe **TodoActor** representa um ator dentro do sistema, ela possui uma instancia da classe **TodoService** para a qual delega o processamento necessário que é recebido através das mensagens. Toda a lógica de processamento das mensagens inicia-se no método **onReceive**, que pode receber uma mensagem de qualquer tipo. As mensagens suportadas pelo ator são definidas junto ao mesmo como classes internas, tornando assim mais visível os tipos de mensagem suportadas pelo ator em questão. Caso seja recebida uma mensagem que não é suportada no contrato do ator, a mesma é destinada para uma caixa de mensagens especifica do Akka para mensagens não entregues e pode receber um tratamento posteriormente. As mensagens suportadas pelo ator são:

- **FindAll:** Ao receber esta mensagem o ator executa o método **findAll** da classe **TodoService** e envia o resultado obtido para o remetente;

- **FindOne:** Ao receber esta mensagem o ator executa o método **findOne** da classe **TodoService** passando o parâmetro **id** contido na mensagem e envia o resultado obtido para o remetente;

- **Save:** Ao receber esta mensagem o ator executa o método **save** da classe **TodoService** passando como parâmetro o objeto **todo** contido na mensagem. Após a conclusão da persistência no banco de dados e resultado obtido através do retorno da inserção é enviado para o remetente;

- **Remove:** Ao receber esta mensagem o ator executa o método **remove** da classe **TodoService**. O retorno do método indicando se a exclusão realmente ocorreu é enviado para o remetente;

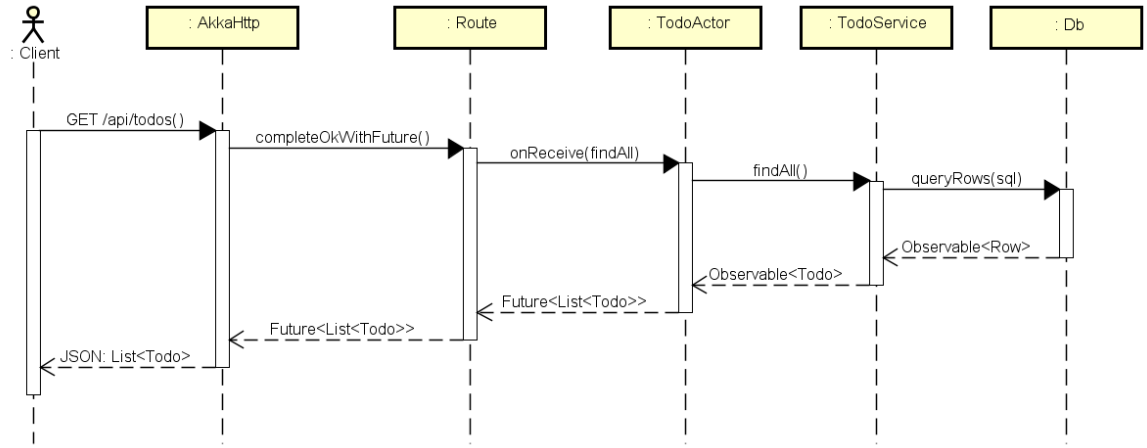
A classe **TodoAsyncApp** é a responsável por inicializar a aplicação, isso inclui instanciar as dependências necessárias e também realizar integração com o Akka HTTP, definindo as funções responsáveis para tratar as URIs.

O diagrama de sequência representado na Figura 36 ilustra uma interação com o ator **TodoActor** através do envio de uma mensagem do tipo **FindAll**. A funcionalidade é a mesma descrita no modelo anterior, porém a execução ocorre de maneira distinta. O processamento inicia-se quando é recebida uma requisição utilizando o método HTTP **GET** para a URI **/api/todos**, neste momento o **Akka HTTP** invoca o método (instancia da interface **Route)** responsável pelo processamento. O método implementado irá criar uma instancia da mensagem **FindAll** e enviar para o **TodoActor** utilizando o *ask pattern*[11] e, com isso, imediatamente é retornado um resultado do tipo **Future** que é repassado para o **Akka HTTP**, este por sua vez irá aguardar a finalização da computação assíncrona para dar continuidade ao processamento da resposta. O **TodoActor** ao receber a mensagem e invoca o método **findAll** da classe **TodoService**, imediatamente é retornada uma instancia da classe **Observable**, o ator então se inscreve no **Observable** para receber uma notificação quando o resultado estiver disponível e encaminha o resultado para o remetente da mensagem. O **Akka HTTP** ao receber a notificação

[11] Padrão utilizado para receber a resposta da mensagem utilizando uma abstração do tipo **Future.**

de conclusão do processamento, serializa o resultado para o formato JSON e encaminha o resultado para o cliente.

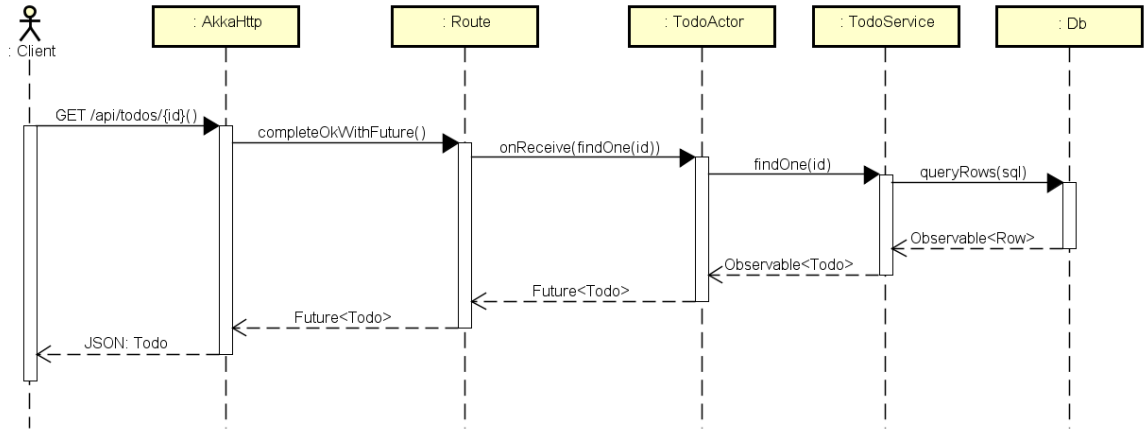
Figura 36 – Diagrama de sequência – TODO Service – Modelo de atores – Método findAll



Fonte: Autor.

O diagrama de sequência representado na Figura 37 ilustra uma interação com o ator **TodoActor** através do envio de uma mensagem do tipo **FindOne**. O processamento se inicia quando é recebida uma requisição utilizando o método HTTP **GET** para a URI **/api/todos/{id}**, neste momento será criada uma mensagem do tipo **FindOne** contendo o **id** extraído da URI que será enviada para o **TodoActor**. O ator então e invoca o método **findOne** da classe **TodoService**, imediatamente é retornada uma instancia da classe **Observable**, o ator então se inscreve no **Observable** para receber uma notificação quando o resultado estiver disponível e encaminha o resultado para o remetente da mensagem. O **Akka HTTP** ao receber a notificação de conclusão do processamento, serializa o resultado para o formato JSON e encaminha o resultado para o cliente.

Figura 37 – Diagrama de sequência – TODO Service – Modelo de atores – Método findOne

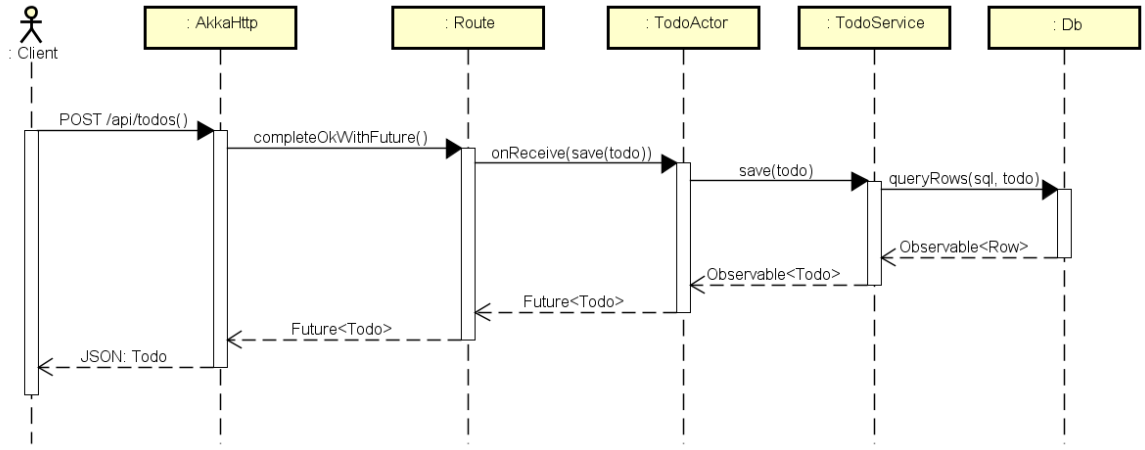


Fonte: Autor.

O diagrama de sequência representado na Figura 38 ilustra uma interação com o ator **TodoActor** através do envio de uma mensagem do tipo **Save**. Ao receber uma requisição utilizando o método HTTP **POST** para a URI **/api/todos** é criada uma mensagem do tipo **Save** contendo um objeto do tipo **Todo** com dados extraídos da requisição, e esta mensagem é encaminhada para o **TodoActor**.

O ator, ao receber a mensagem, identifica seu tipo e invoca o método **save** da classe **TodoService**, esta, por sua vez, através da classe **Db**, executa uma operação de inserção de dados no banco de dados retorna de imediato uma instancia da classe **Observable** representando a computação assíncrona**.** O ator então se inscreve no **Observable** para ser notificado assim que o resultado estiver disponível, e após isso, encaminha o resultado para o remetente da mensagem. O **Akka HTTP** ao receber a notificação de conclusão do processamento, serializa o resultado para o formato JSON e encaminha o resultado para o cliente.

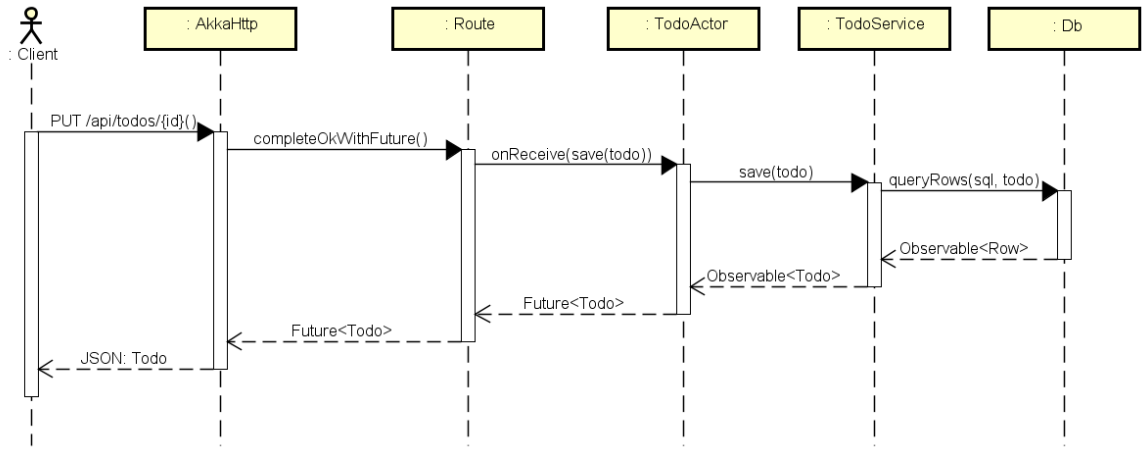
Figura 38 – Diagrama de sequência – TODO Service – Modelo de atores – Método save



Fonte: Autor.

O diagrama de sequência representado na Figura 39 ilustra uma interação com o ator **TodoActor** para realizar a atualização de um registro através do envio de uma mensagem do tipo **Save** que é disparada ao receber uma requisição utilizando o método HTTP **PUT** para a URI **/api/todos/{id}**. O processo ocorre exatamente como descrito no método anterior, sendo que a única diferença é a execução de um comando de atualização no banco de dados ao invés de uma inserção.

Figura 39 – Diagrama de sequência – TODO Service – Modelo de atores – Método update



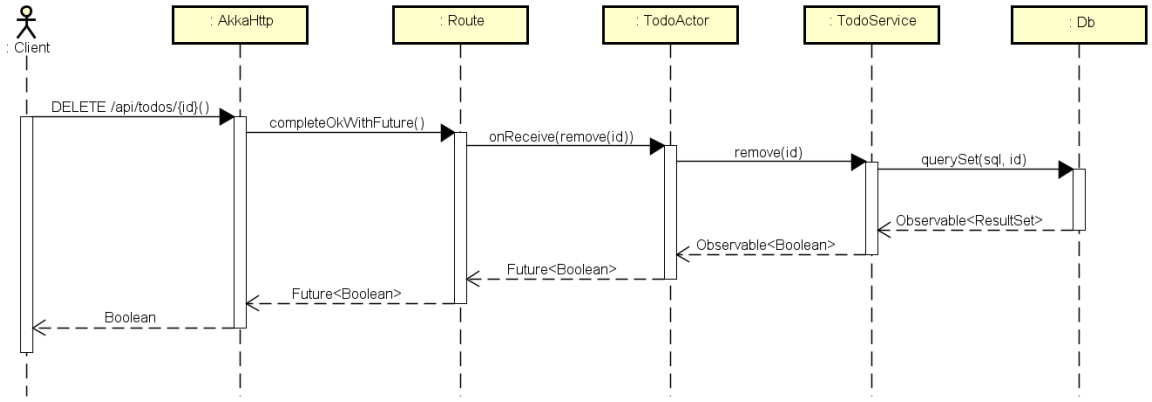
Fonte: Autor.

O diagrama de sequência representado na Figura 40 ilustra uma interação com o ator **TodoActor** para a remoção de um registro através do envio de uma mensagem do tipo **Remove**. Ao receber uma requisição utilizando o método HTTP **DELTE** para a URI **/api/todos/{id}** é

criada e encaminhada para o **TodoActor** uma mensagem do tipo **Remove** contendo o **id** do objeto a ser removido.

O ator, ao receber a mensagem, identifica seu tipo e invoca o método **remove** da classe **TodoService**, esta, por sua vez, através da classe **Db**, executa uma operação de remoção no banco de dados e retorna de imediato uma instancia da classe **Observable** representando a computação assíncrona**.** O ator então se inscreve no **Observable** para ser notificado assim que o registro for excluído, e após isso, encaminha o resultado da exclusão para o remetente da mensagem. O **Akka HTTP** ao receber a notificação de conclusão da exclusão envia um resultado sinalizando se a exclusão foi concluída com sucesso para o cliente.

Figura 40 – Diagrama de sequência – TODO Service – Modelo de atores – Método remove



Fonte: Autor.

4.6.2 RayTracer Service

Este *microservice*, conforme descrito anteriormente, tem por objetivo gerar uma imagem através do processo de renderização conhecido como *Ray Tracing*. O código para geração de imagem com o *Ray Tracing* foi implementado em um módulo separado e é utilizado em ambas as implementações deste *microservice*. O código contido neste módulo foi adaptado de um projeto de código aberto chamado JFXRay[12].

[12] Disponível em: <https://github.com/chriswhocodes/JFXRay>. Acesso em outubro 2016.

4.6.2.1 Modelo multithread

A funcionalidade da aplicação é toda implementada em um módulo comum a ambos os modelos, chamado **raytracer-common**, portanto o detalhamento do funcionamento deste modulo neste subcapitulo continua sendo valido para o próximo modelo. Este módulo é composto pelas classes **Vector3f, RenderConfig, JFXRay e RayTracer**.

A classe **Vector3f** encapsula um conjunto de coordenadas representando um ponto em um espaço tridimensional e contém algumas operações uteis para trabalhar com vetores.

A classe **RenderConfig** define todos os parâmetros que serão utilizados na geração da imagem. Através dela é realizada a configuração do tamanho desejado (altura e largura), número de *threads* a serem utilizadas na renderização, posição da câmera e definição dos objetos (esferas) a serem renderizados.

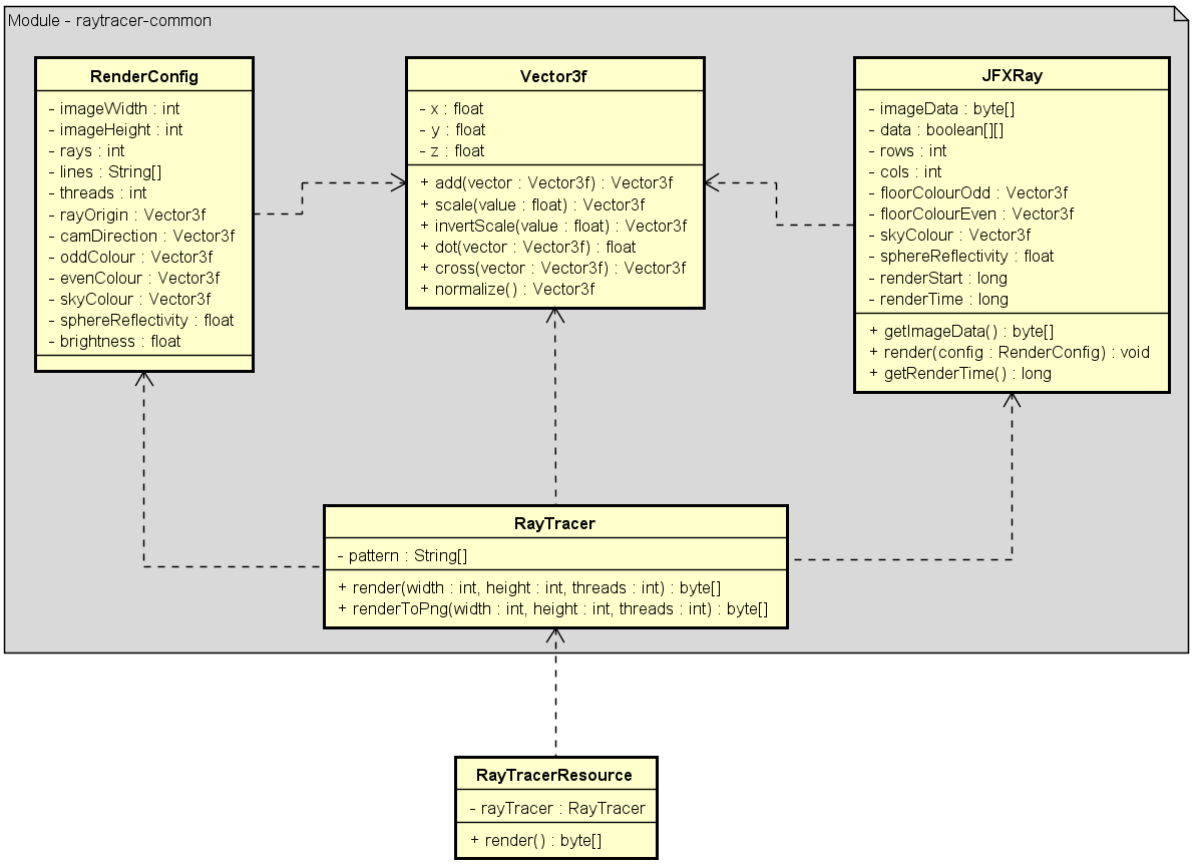
A classe **JFXRay** é responsável por implementar o algoritmo de Ray Tracing e gerar a imagem com base nos parâmetros recebidos através de um objeto do tipo **RenderConfig**. O processo suporta o uso de múltiplas *threads* para renderizar a imagem, tornando possível aproveitar todos os núcleos de processamento disponíveis. O processamento da imagem inicia-se no método **render** e após sua conclusão o resultado pode ser coletado através do método **getImageData**.

A classe **RayTracer** encapsula o processo de geração de uma imagem especifica que é utilizada nos testes, através dela é gerada uma instancia pré-definida da classe **RenderConfig** que é renderizada através de uma instancia da classe **JFXRay**, retornando assim os *bytes* que compõem a imagem final. Esta classe também é responsável por codificar a informação da imagem bruta gerada pelo processo de renderização no formato *Portable Network Graphics* (PNG).

A única classe especifica da implementação deste modelo é o **RayTracerResource.** Esta classe contém um instancia da classe **RayTracer** e é responsável por expor a funcionalidade da mesma para a Web através de seu método **render**. Ela não realiza nenhum tipo de processamento, servindo somente como uma ponte entre a aplicação e o Spring MVC. As classes e relações que compõem esta implementação podem ser vistas na Figura 41.

.

Figura 41 – Diagrama de classes – RayTracer Service – Modelo *multithread*

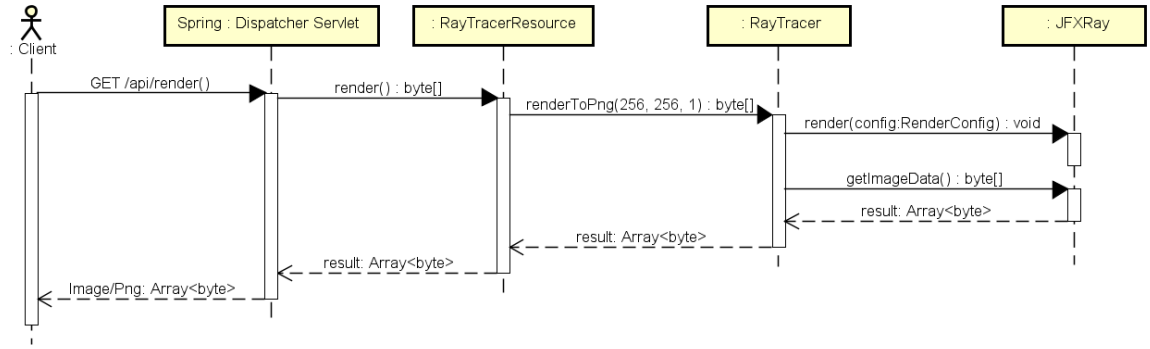


Fonte: Autor.

O diagrama de sequência representado na Figura 42 ilustra uma interação com o método **render** da classe **RayTracerResource** para obter uma imagem. Ao receber uma requisição utilizando o método HTTP **GET** para a URI **/api/render** a classe **RayTracerResource** invoca o método **renderToPng** da classe **RayTracer** solicitando uma imagem de 256 *pixels* de largura por 256 *pixels* de altura utilizando somente uma *thread* para processamento da imagem.

A classe **RayTracer** então cria uma instancia da classe **RenderConfig** contendo alguns parâmetros padronizados e customiza a mesma com os parâmetros recebidos. Após a configuração é gerada uma instancia da classe **JFXRay** e invocado o método **render** para iniciar o processamento, passando como parâmetro a instancia da classe **RenderConfig** criada anteriormente. Após a conclusão do processamento, a classe **RayTracer** invoca método **getImageData** para obter os *bytes* da imagem resultante, realiza a codificação para o formato PNG e retorna o resultado. O **RayTracerResource** então envia este resultado para o cliente, sinalizado que o conteúdo representa uma imagem codificada em PNG.

Figura 42 – Diagrama de sequência – RayTracer Service – Modelo *multithread*



Fonte: Autor.

4.6.2.2 Modelo de Atores

Conforme descrito anteriormente, ambas as implementações compartilham código através do módulo **raytracer-common**. A implementação no modelo de atores é composta das classes **RayTracerApp**, **CoordinatorActor** e **ExecutorActor.** Esta aplicação é altamente dependente de processamento e, portanto, é altamente bloqueante. Devido a isto, a parte do código que realiza o processamento deve ser executada fora das *threads* utilizadas pelo Akka para coordenar a execução dos atores, caso contrário todo o sistema todo sofrerá perda de performance.

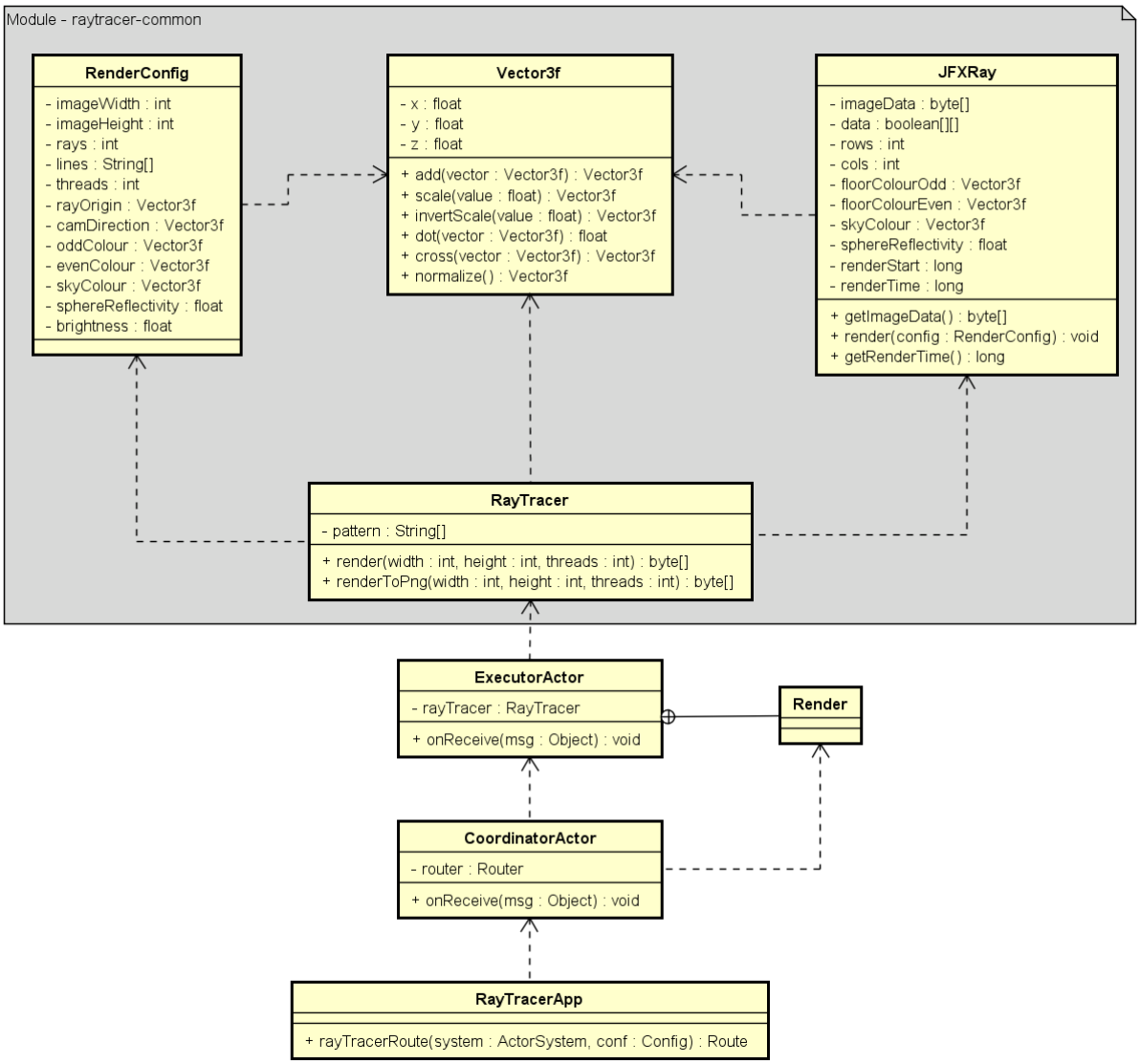
O ator **ExecutorActor** possui uma instancia da classe **RayTracer** e é responsável por gerar uma imagem quando solicitado. A solicitação é realizada através da mensagem do tipo **Render** (definida como uma classe interna do **ExecutorActor**) e é a única suportada pelo ator. Ao receber esta mensagem o ator inicia a execução do processamento da imagem através da classe **RayTracer**. Os atores deste tipo, por executarem operações altamente bloqueantes, trabalham com um *thread pool* separado do principal, evitando assim que o sistema todo seja afetado por sua contenção.

O ator **CoordinatorActor** é responsável pela coordenação e supervisão do processo de geração de imagem executado pelos atores do tipo **ExecutorActor**. Este ator pode gerir várias instancias de atores deste tipo e funciona como uma interface de roteamento para os mesmos, os atores coordenados ficam encapsulados dentro da classe **Router** do Akka. Ao receber uma mensagem do tipo **Render** este ator direciona a mesma para um de seus atores supervisionados

para prosseguir com a execução. Por padrão este ator instancia um **ExecutorActor** para cada núcleo de processamento disponível.

A classe **RayTracerApp** é responsável por inicializar toda a aplicação, instanciando os componentes necessários e registrando os atores responsáveis pelo tratamento das URIs. O diagrama de classes da Figura 43 ilustra a estrutura da aplicação.

Figura 43 – Diagrama de classes – RayTracer Service – Modelo de atores



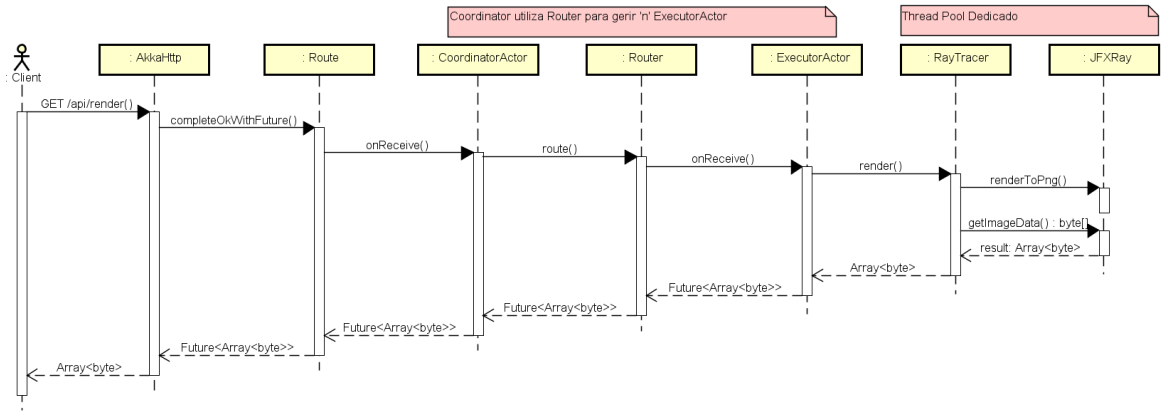
Fonte: Autor.

O diagrama de sequência da Figura 44 ilustra uma interação com o método de renderização implementado no modelo de atores para a geração de uma imagem. Quando uma requisição é enviada para servidor utilizando método HTTP **GET** para a URI **/api/render**, o **Route** responsável pela URI envia uma mensagem do tipo **Render** para o **CoordinatorActor.**

Este por sua vez delega a mensagem para um dos atores do tipo **ExecutorActor** através da do método **route** contido na interface **Router**. O **Router**, através de uma estratégia pré-configurada (*Round-robin*[13] por padrão), seleciona alguma das instâncias da classe **ExecutorActor** existentes para a qual irá delegar a mensagem.

O **ExecutorActor** ao receber a mensagem inicia o processamento da imagem executando o método **renderToPng** da classe **RayTracer** solicitando uma imagem de 256 *pixels* de largura por 256 *pixels* de altura utilizando somente uma *thread* para processamento da imagem. Ao finalizar a operação, o resultado é enviado diretamente para o remetente da mensagem, neste caso o Route integrado com o Akka, que irá completar a resposta.

Figura 44 – Diagrama de sequência – RayTracer Service – Modelo de atores



Fonte: Autor.

4.6.3 Report Service

Este *microservice*, conforme descrito anteriormente, tem por objetivo consultar os dois outros *microservices* propostos e criar um relatório a partir dos resultados obtidos dos mesmos.

[13] Algoritmo de escalonamento que distribui as tarefas de maneira igualitária e circular através de seus participantes.

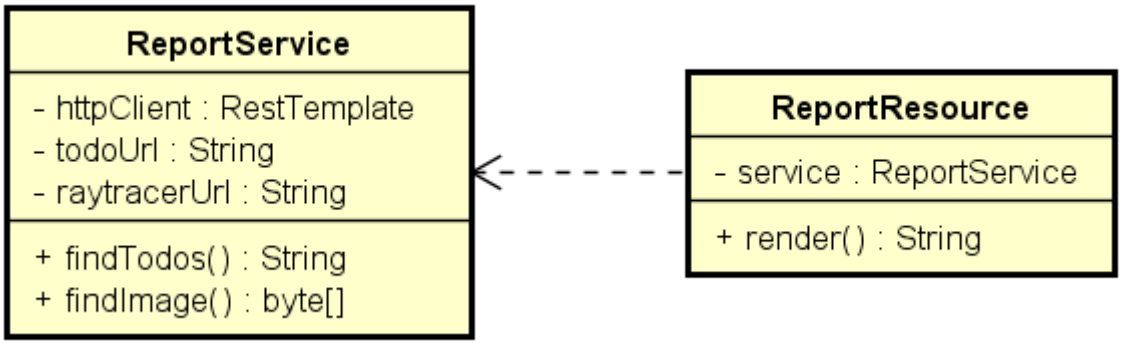
4.6.3.1 Modelo multithread

Na implementação deste modelo todas as operações de E/S são executadas de maneira síncrona. O protótipo é composto de duas classes, o **ReportService** e o **ReportResource**.

A classe **ReportService** é responsável por implementar a comunicação com os serviços externos (**TODO Service** e **RayTracer Service**). Para isto ela mantém os atributos **todoUrl** e **raytracerUrl**, que são os caminhos em que estão disponíveis os serviços a serem consultados. Também possui uma instancia da classe **RestTemplate**, que é uma classe do *framework* Spring para facilitar o consumo de *webservices*. A classe também disponibiliza um método para consultar cada um dos serviços.

A classe **ReportResource** possui uma instancia da classe **ReportService**, a qual utiliza para buscar informações necessárias para compor o resultado do seu método **render** que é disponibilizado através da URI **/api/render**. O diagrama de classes representando a estrutura da aplicação pode ser visto na Figura 45.

Figura 45 – Diagrama de classes – Report Service – Modelo *multithread*



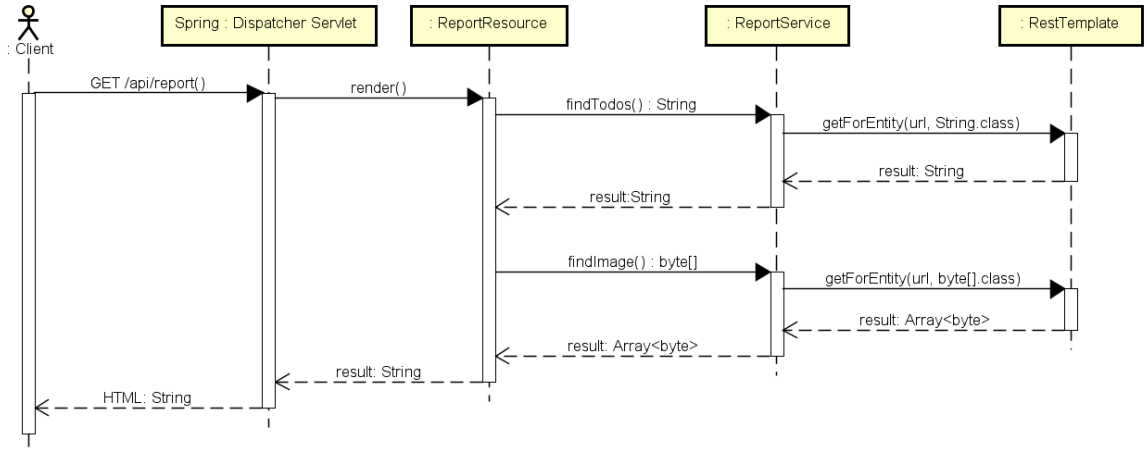
Fonte: Autor.

O diagrama de sequência apresentado na Figura 46 demonstra uma interação com o método de geração do relatório. Ao receber uma requisição utilizando o método HTTP **GET** para a URI **/api/report** a classe **DispatcherServlet** do Spring MVC invoca o método **render** da classe **ReportResource**.

O **ReportResource**, primeiramente, invoca o método **findTodos** da classe **ReportService**, e esta por sua vez, faz uma chamada para o método **getForEntity** da classe **RestTemplate** passando a URI da aplicação **TODO Service**, que é responsável por

disponibilizar uma listagem dos objetos do tipo **Todo** existentes em seu banco de dados. Após completar esta chamada, é invocado o método **findImage** da classe **ReportService** que utiliza o método **getForEntity** da classe **RestTemplate** passando a URI da aplicação **RayTracer Service** responsável por gerar a imagem. Após a conclusão destas operações, seus resultados são apresentados em uma página em *Hypertext Markup Language* (HTML).

Figura 46 – Diagrama de sequência – Report Service – Modelo *multithread*



Fonte: Autor.

4.6.3.2 Modelo de atores

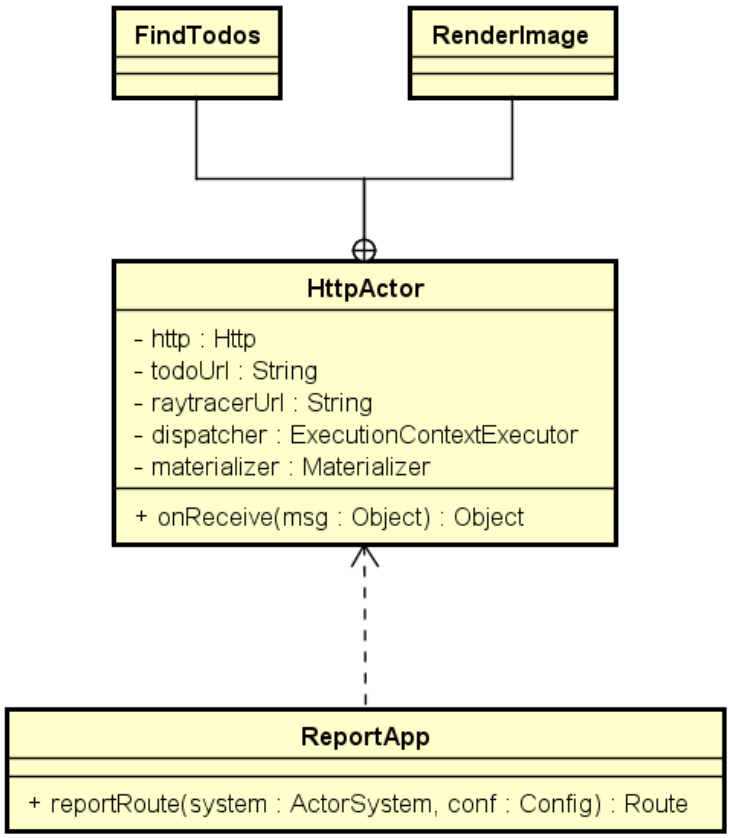
Na implementação deste modelo, todos os processos de E/S utilizados neste modelo são executados de maneira assíncrona. A aplicação é composta pelas classes **ReportApp** e **HttpActor**.

A classe **HttpActor** é a responsável por realizar a comunicação com os *webservices* **TODO Service** e **RayTracer Service**. Os atributos **todoUrl** e **raytracerUrl** armazenam o caminho utilizado para acessar os *webservices***.** Além disto, a classe ainda faz uso de algumas classes especificas do Akka, que são utilizadas para realizar requisições HTTP para os *webservices* de maneira assíncrona. Este ator suporta dois tipos de mensagem:

- **FindTodos**: Solicita ao ator o conteúdo existente no *webservice* **TODO Service**;
- **RenderImage**: Solicita ao ator uma imagem a ser renderizada pelo *webservice* **RayTracer Service**.

A classe **ReportApp** é responsável por inicializar toda a aplicação, instanciando os componentes necessários e registrando os atores responsáveis pelo tratamento das URIs. O diagrama de classes da Figura 47 ilustra a estrutura da aplicação.

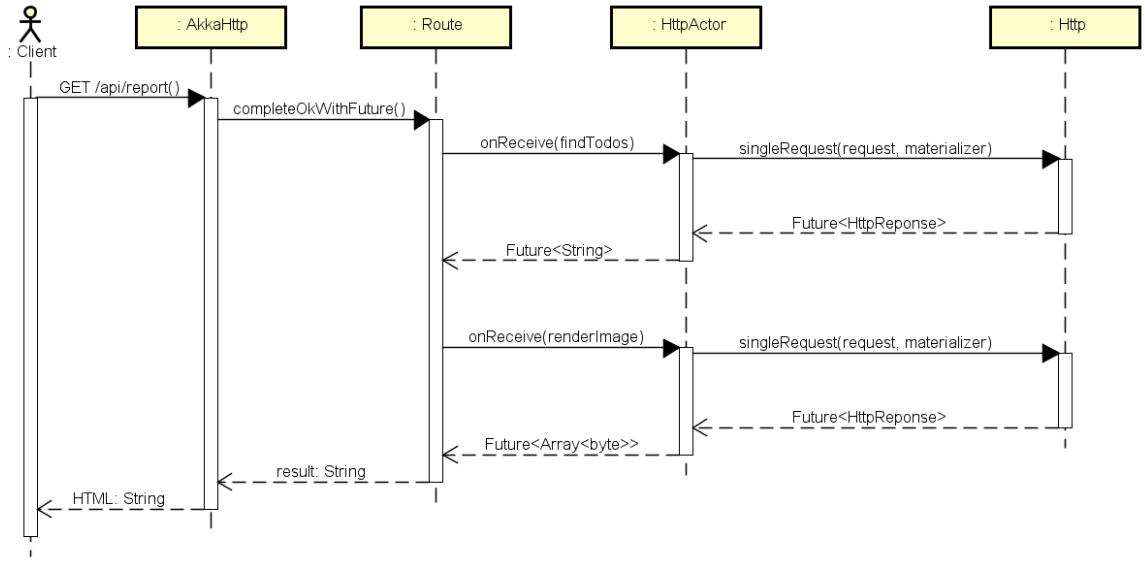
Figura 47 – Diagrama de classes – Report Service – Modelo de atores



Fonte: Autor.

O diagrama de sequência da Figura 48 ilustra uma interação com o método de geração de relatório. O processo inicia-se quando é recebida uma requisição utilizando o método HTTP **GET** para a URI **/api/report**. O **Akka HTTP** ao identificar a chamada, encaminha a requisição para o **Route** responsável, e este ao receber a requisição envia uma mensagem do tipo **FindTodos** para o **HttpActor** e logo após o retorno (que é imediato, pois é uma operação assíncrona) envia uma mensagem do tipo **RenderImage** para o mesmo ator. O **Route** então aguarda a finalização de ambas as computações assíncronas, gera o conteúdo HTML e envia o resultado ao cliente.

Figura 48 – Diagrama de sequência – Report Service – Modelo de atores



Fonte: Autor.

4.7 Testes

Neste subcapitulo serão apresentados todos os elementos relacionados a execução dos testes como os ambientes utilizados, configurações aplicadas, distribuição das aplicações, procedimentos e softwares utilizados.

4.7.1 Ambiente

Todos os testes foram executados no ambiente do Google Cloud. Para a execução foram criadas quatro maquinas virtuais do tipo **n1-highcpu-2**, que possuem a seguinte especificação:

- Processador: Intel Xeon E5-2697v3 2,30GHz (dois núcleos);
- Memória: 1,8GB 1600MHz;
- Disco: SSD 10GB sem especificação;
- Sistema Operacional: Ubuntu Server 16.04.

Para permitir a execução dos testes de maior escala, foi necessária realizar uma configuração no sistema operacional para aumentar o limite máximo de descritores de arquivos

abertos por usuário. A configuração visível na Listagem 4 foi aplicada no arquivo **/etc/security/limits.conf**:

Listagem 4 – Configuração do limite de descritores de arquivos abertos no Ubuntu Server

```
* hard nofile 99999
* soft nofile 99999
* hard noproc 99999
* soft noproc 99999
```

Fonte: Autor.

Cada máquina virtual ficou a cargo da execução de um dos quatro serviços necessários para a execução das simulações:

- **PostgreSQL**
- **TODO Service**
- **RayTracer Service**
- **Report Service**

Algumas configurações adicionais (visíveis na Listagem 5) foram realizadas no PostgreSQL para permitir uma melhor utilização da memória disponível na máquina virtual:

Listagem 5 – Configuração do PostgreSQL

```
max_connections = 10
shared_buffers = 378MB
effective_cache_size = 1134MB
work_mem = 38707kB
maintenance_work_mem = 96768kB
min_wal_size = 1GB
max_wal_size = 2GB
checkpoint_completion_target = 0.7
wal_buffers = 11612kB
default_statistics_target = 100
```

Fonte: Autor.

A configuração da JVM também foi padronizada em todas as maquinas, utilizando o parâmetro **-Xmx1024m** para instruir um limite máximo de 1GB de memória a ser alocado pela aplicação durante sua execução.

As configurações das ferramentas utilizadas na implementação dos modelos foram mantidas em seus valores padrão. A implementação realizada no modelo *multithread* faz uso de um servidor HTTP Apache Tomcat que, em sua configuração padrão, disponibiliza no máximo 200 *threads* para processamento de requisições. Já a implementação do modelo de atores faz uso do Akka que por padrão utiliza um *thread pool* contendo uma *thread* por núcleo de processamento disponível no hardware e mais uma *thread* adicional, neste caso utilizando um total de 3 *threads*.

As execuções dos *scripts* de simulação foram realizadas em uma máquina local, fora da rede do Google Cloud. Com isso foi possível adicionar as simulações um certo nível de latência de rede. As configurações da máquina são:

- Processador: AMD Phenom II X4 955 3.2GHz (quatro núcleos);
- Memoria: 8GB 1600MHz;
- Disco: SSD 250GB Samsung EVO 850;
- Sistema Operacional: Windows 10;
- Conexão: 15Mb de *download* e 3Mb de *upload*.

4.7.2 Instrumentação

Para realizar a implementação dos cenários propostos, foi utilizada a ferramenta Gatling e a linguagem de programação Scala. O Gatling disponibiliza uma *Domain-Specific Language* (DSL) em Scala para a criação e execução de simulações sob o protocolo HTTP de maneira fácil e eficiente.

O Gatling executa os *scripts* de simulação e, durante a execução, coleta diversas métricas úteis como por exemplo o tempo de resposta, status das respostas e erros durante a execução. A partir destes dados, ele gera um relatório contendo diversas informações como médias de tempo de resposta para cada ação executada dentro da simulação e número de respostas por segundo.

Para realizar as medições referentes a utilização do processador e uso de memória, foi utilizada uma ferramenta que é distribuída junto ao Java, o VisualVM. Esta ferramenta permite o monitorar aplicações remotas que executam na JVM utilizando o *Java Management Extensions* (JMX).

4.7.3 Procedimentos

Para a execução dos testes foram criados *scripts* para realizar a limpeza do ambiente e do banco de dados (quando necessário) antes da inicialização das aplicações. Todas as simulações são executadas três vezes, sendo que a última execução é considerada como resultado final. A repetição das execuções tem a finalidade de remover variáveis como tempo *warmup* da JVM (devido ao JIT) do resultado final.

As simulações são executadas através dos seguintes passos:

- Inicialização da aplicação e de suas dependências através dos *scripts*;
- Inicialização do VisualVM e conexão com a aplicação para realização do monitoramento;
- Realização das três execuções do *script* de simulação da aplicação em questão utilizando o Gatling;
- Coleta dos resultados.

4.8 Análise dos resultados

Neste subcapitulo estão apresentados os resultados coletados durante execução dos testes. Para cada um dos *microservices* estão descritos os resultados obtidos em ambos os modelos implementados e também uma análise comparativa entre os mesmos, explicando as causas das diferenças entre os resultados.

4.8.1 TODO Service

Neste subcapitulo serão apresentados os resultados coletados nas simulações para o *microservice* **TODO Service**.

4.8.1.1 Modelo multithread

Este modelo levou um total de **166 segundos** para completar a execução da simulação. Na Tabela 1 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **30,12 requisições por segundo** além de um tempo médio de resposta de **19,63 segundos** por requisição. Um total de **569** requisições resultaram em erro devido ao alto tempo de resposta, isto representa **11,38%** do número total de requisições enviadas para a aplicação.

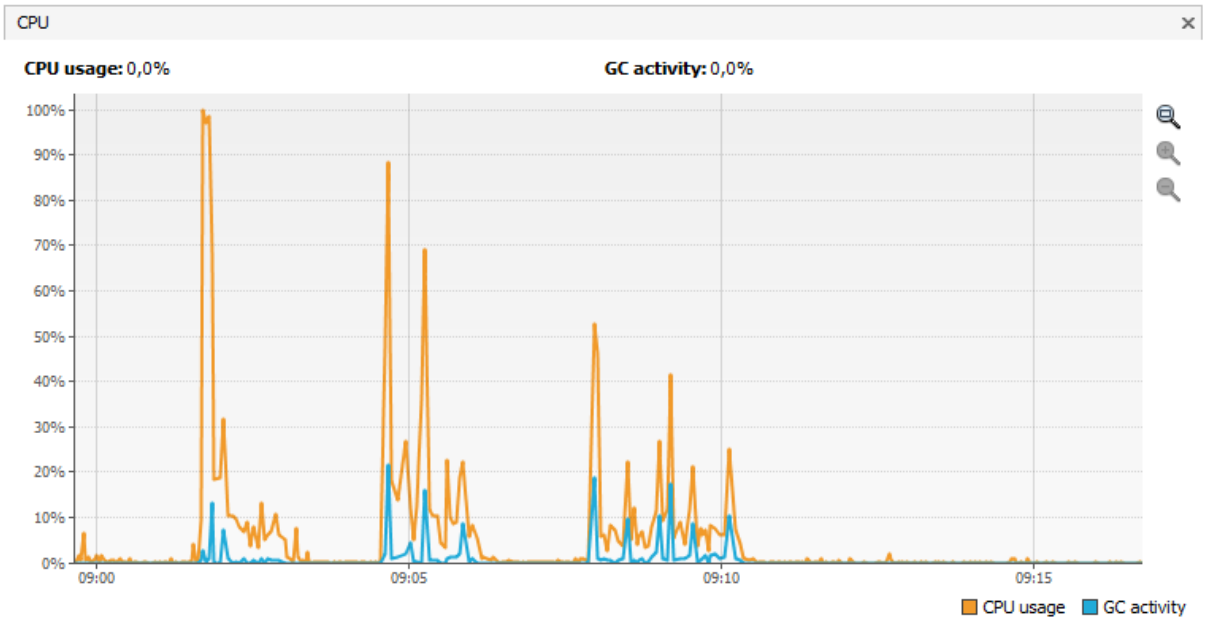
Tabela 1 – TODO Service – Tempos de reposta – Modelo *multithread*

Tipo	**Execuções**				**Tempo de Resposta (ms)**		
	Total	**OK**	**Com Erro**	**Req./s**	**Min.**	**Max.**	**Media**
Create	2.000	1.992	8	12,05	179	50.026	8.251
FindOne	1.000	1.000	0	6,02	179	45.689	12.202
Delete	1.000	1.000	0	6,02	179	39.137	15.615
FindAll	1.000	439	561	6,02	14.920	60.049	53.810
Global	5.000	4.431	569	30,12	179	60.049	19.626

Fonte: Autor.

Na Figura 49 podemos acompanhar o uso do processador durante a execução dos testes. Na última execução, considerada como resultado final, o uso do processador manteve-se em média abaixo dos **20%**, podendo ser observado um pico de aproximadamente **50%** de uso em alguns momentos. O uso do processador por parte do *Garbage Collector* (GC) também foi relativamente alto, chegando a picos de quase **20%** em alguns momentos.

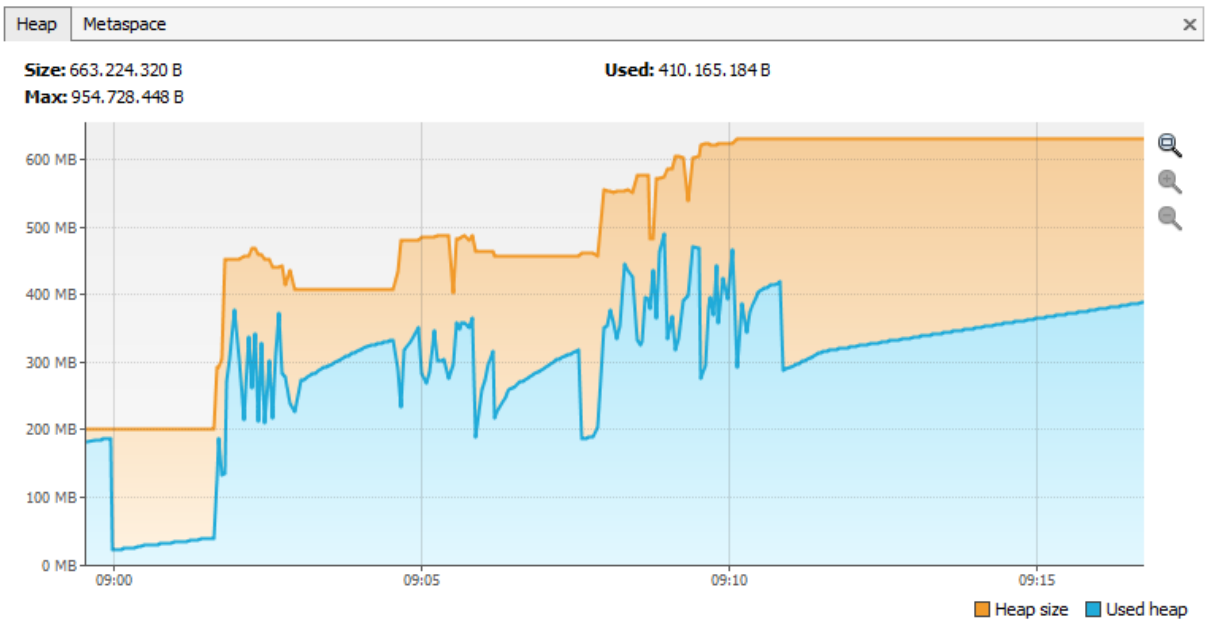
Figura 49 – TODO Service – Uso do processador – Modelo *multithread*



Fonte: Autor.

O uso de memória durante a simulação manteve-se em uma média de aproximadamente **350MB**, atingindo picos de quase **500MB** durante alguns momentos, como pode ser visto na Figura 50.

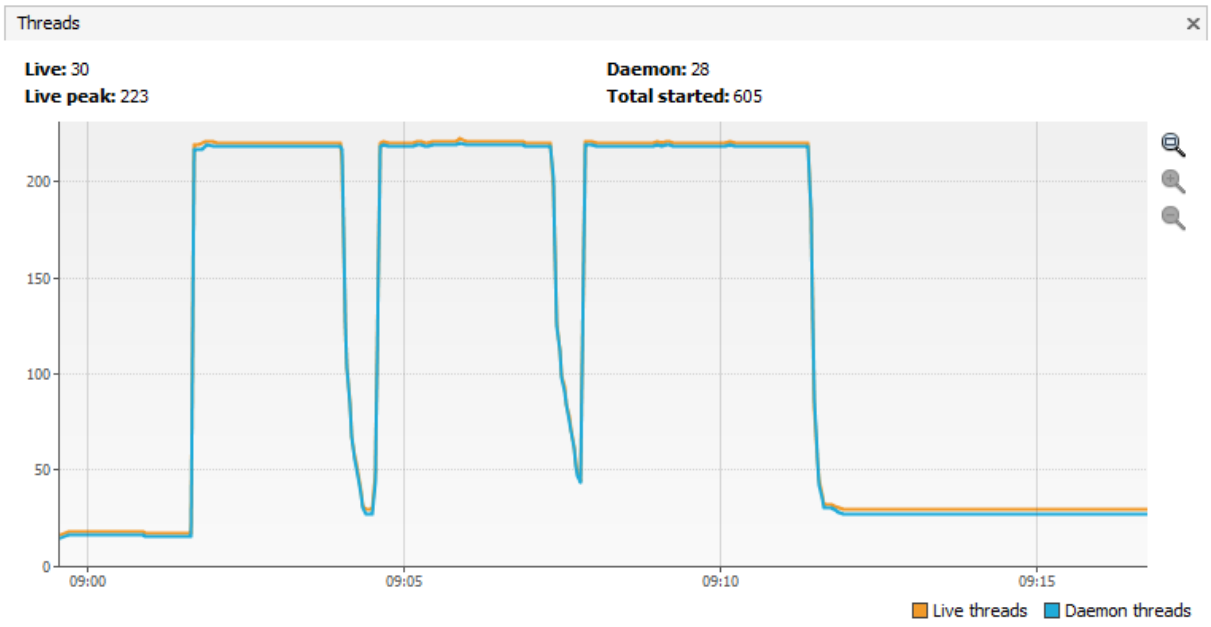
Figura 50 – TODO Service – Uso de memória – Modelo *multithread*



Fonte: Autor.

O número de *threads* alocadas atingiu o limite máximo de 200 definido por padrão no servidor Tomcat durante a execução das simulações. Durante todo o tempo da simulação estiveram ativas **223** *threads*, como pode ser visto na Figura 51.

Figura 51 – TODO Service – Alocação de t*hreads* – Modelo *multithread*



Fonte: Autor.

4.8.1.2 Modelo de atores

Este modelo levou um total de **6 segundos** para completar a execução da simulação. Na Tabela 2 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **833,33 requisições por segundo** além de um tempo médio de resposta de **0,5 segundo** por requisição. Todas as requisições executadas foram finalizadas com sucesso.

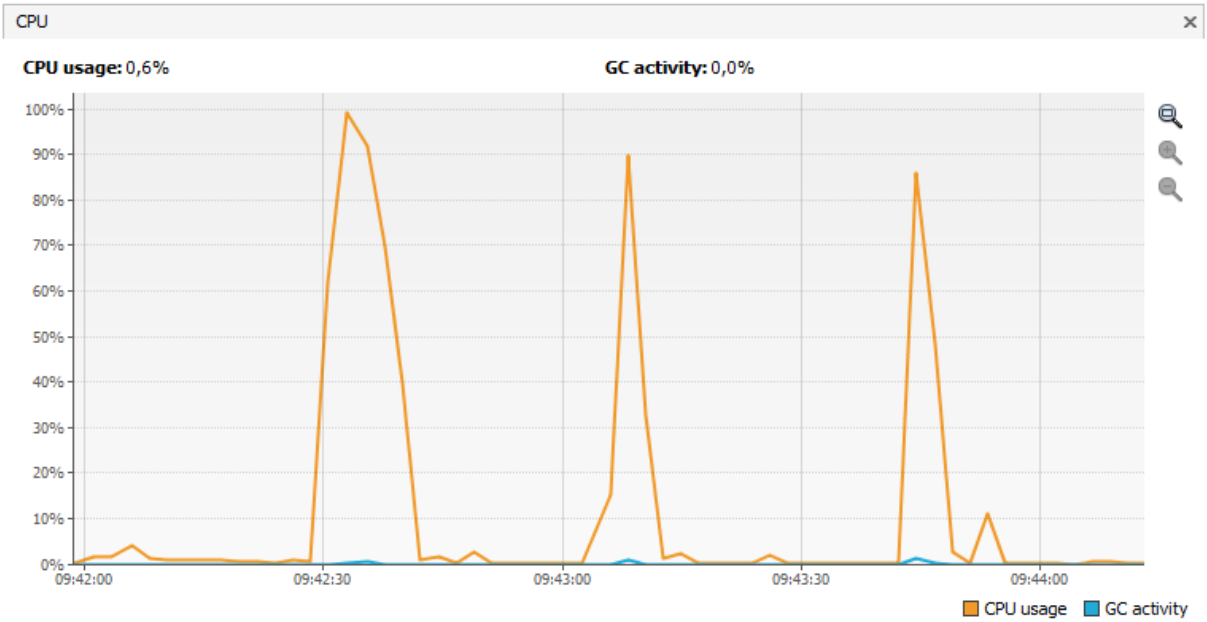
Tabela 2 – TODO Service – Tempos de reposta – Modelo de atores

Tipo	Execuções				Tempo de Resposta (ms)		
	Total	OK	Com Erro	Req./s	Min.	Max.	Media
Create	2.000	1.000	0	333,33	181	3.547	784
FindOne	1.000	1.000	0	166,67	182	1.376	344
Delete	1.000	1.000	0	166,67	181	1.214	302
FindAll	1.000	1.000	0	166,67	181	1.240	278
Global	5.000	5.000	0	833,33	181	3.429	499

Fonte: Autor.

Na Figura 52 podemos acompanhar o uso do processador durante a execução dos testes. O uso do processador ocorre em picos que alcançam quase **90%** durante alguns instantes e caem de forma abrupta logo em seguida. O uso de processador por parte do GC neste modelo é praticamente nulo.

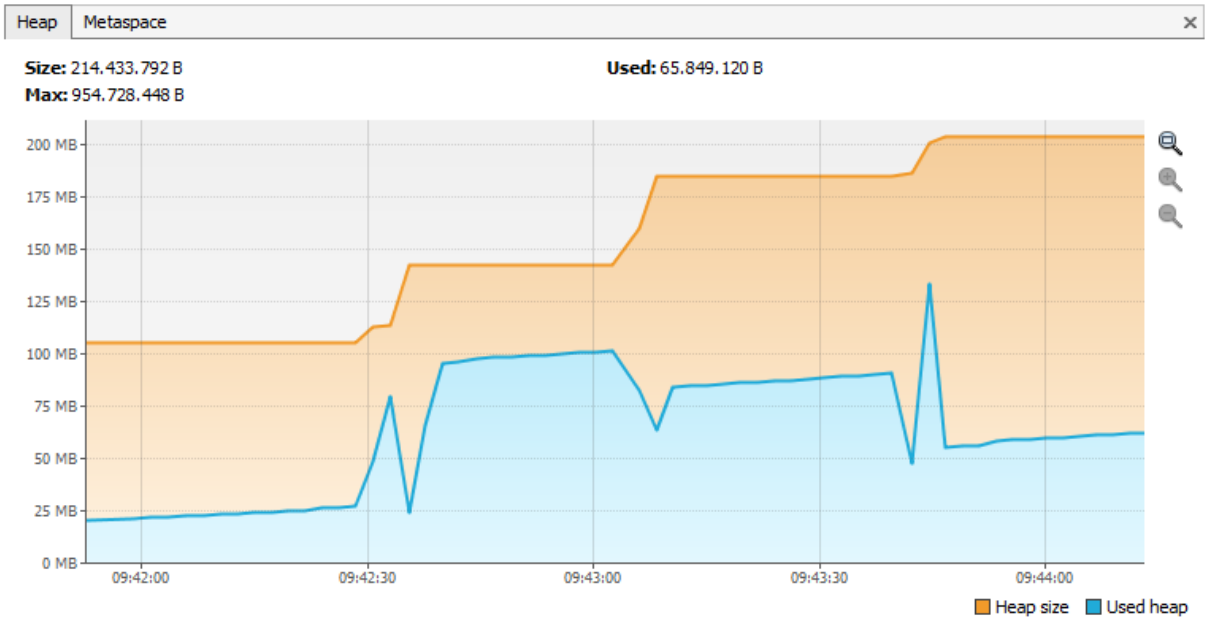
Figura 52 – TODO Service – Uso do processador – Modelo de atores



Fonte: Autor.

O uso de memória durante a simulação manteve-se em uma média de aproximadamente **80MB**, atingindo picos de quase **125MB** durante alguns momentos, como pode ser visto na Figura 53.

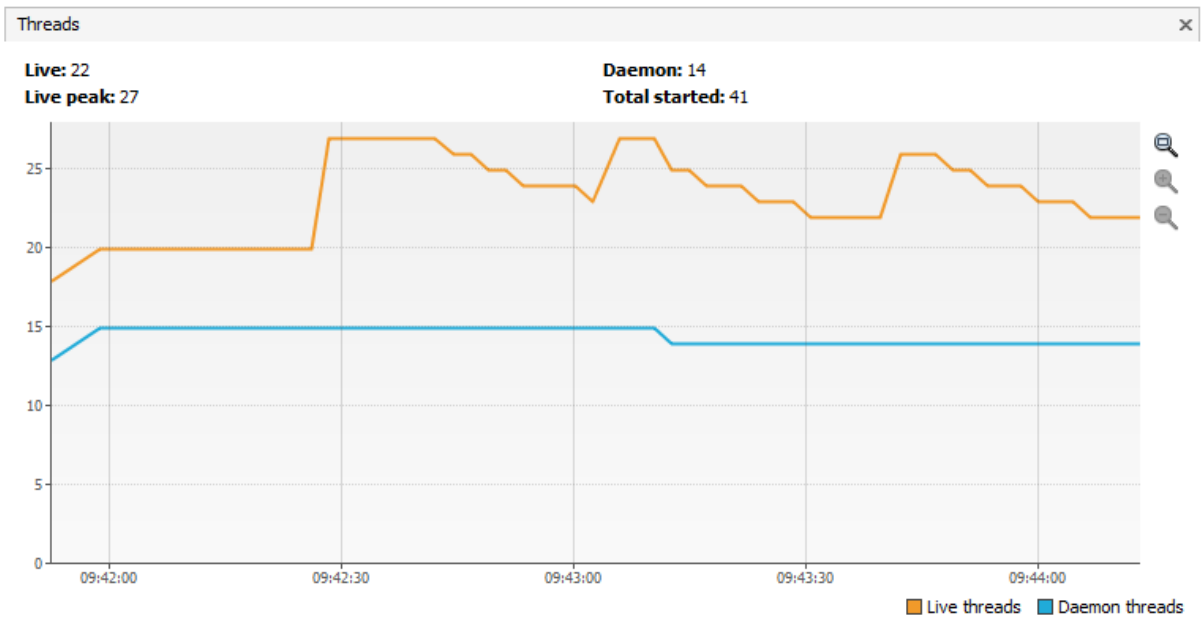
Figura 53 – TODO Service – Uso de memória – Modelo de atores



Fonte: Autor.

Durante a execução das simulações o número de *threads* alocadas variou de **20** até **27**, mantendo um valor relativamente estável durante toda a simulação, como pode ser visto na Figura 54.

Figura 54 – TODO Service – Alocação de *threads* – Modelo de atores



Fonte: Autor.

4.8.1.3 Comparativo dos resultados

Neste *microservice* o modelo de atores apresentou um resultado muito superior ao modelo *multithread*. A principal razão disto é devido a forma como as operações são executadas pela aplicação. O modelo de atores executa muito menos operações de *context switch* do que a implementação no modelo *multithread* devido a multiplexação de *threads*, uso de filas e operações de E/S assíncronas, além de não gerar nenhum tipo de contenção através do uso de estruturas de sincronização. Com estas ações, o modelo de atores consegue realizar um uso mais eficiente do processador, permitindo que o mesmo execute suas tarefas com um menor número de interrupções e, portanto, as finalize em um menor tempo. Podemos observar este fenômeno através do uso do processador e tempo total do teste nas simulações executadas sob este modelo.

No modelo *multithread* o uso do processador se mantem relativamente baixo mesmo com inúmeras requisições pendentes. Isto ocorre devido ao esgotamento do *thread pool* utilizado para atender as requisições. Além disto o alto número de *threads* utilizadas gera um consumo de memória bem elevado quando comparado com o modelo de atores, isto ocorre por que cada *thread* possui um custo de alocação de memória associado. Outro aspecto que podemos observar é o uso relativamente alto do processador por parte do GC, o que indica que o Java pode estar tendo dificuldade em recuperar a memória alocada nesta implementação.

No entanto, o modelo *multithread* apresenta um tempo de resposta mínimo levemente inferior ao modelo de atores indicando que para cenários de baixa concorrência ambos os modelos devem apresentar resultados similares.

4.8.2 RayTracer Service

Neste subcapitulo serão apresentados os resultados coletados nas simulações para o *microservice* **RayTracer Service**.

4.8.2.1 Modelo multithread

Este modelo levou um total de **61 segundos** para completar a execução da simulação. Na Tabela 3 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **0,82 requisições por segundo** além de um tempo médio de resposta de **60 segundos** por requisição. Todas as requisições enviadas para a aplicação falharam devido ao alto tempo de resposta.

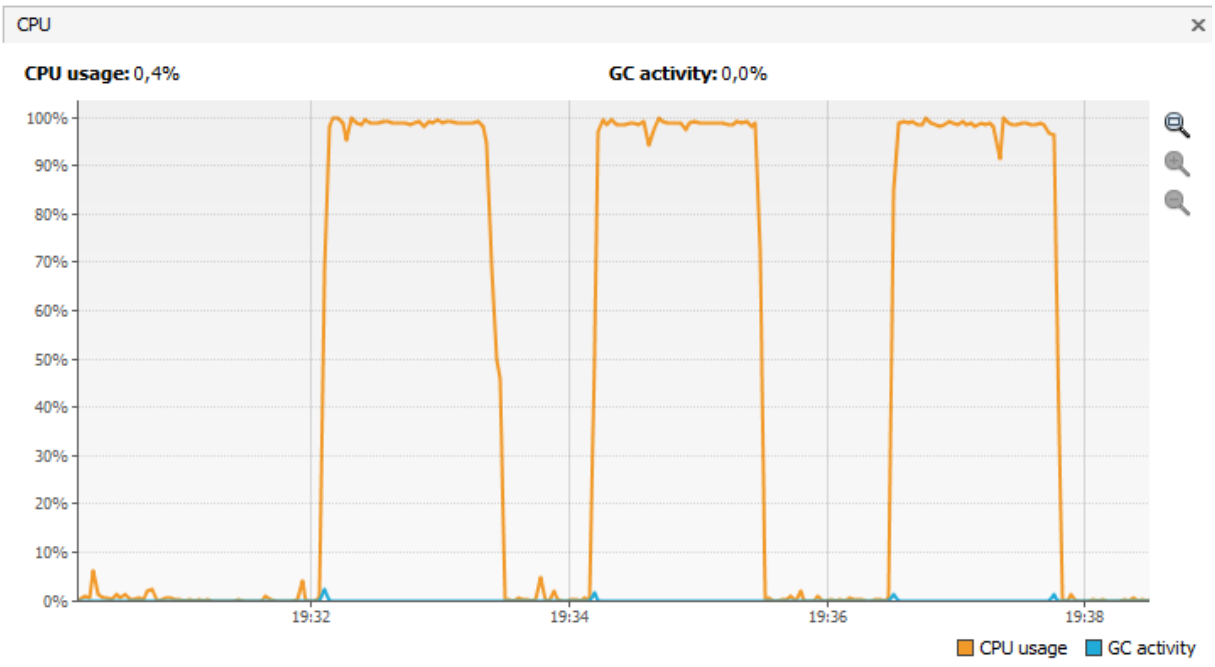
Tabela 3 – RayTracer Service – Tempos de reposta – Modelo *multithread*

Tipo	**Execuções**				**Tempo de Resposta (ms)**		
	Total	**OK**	**Com Erro**	**Req./s**	**Min.**	**Max.**	**Media**
Render	50	0	50	0,82	60.004	60.017	60.010

Fonte: Autor.

Na Figura 55 podemos acompanhar o uso do processador durante a execução dos testes que de maneira geral se mantem sempre em **100%** devido ao tipo de processamento desta aplicação.

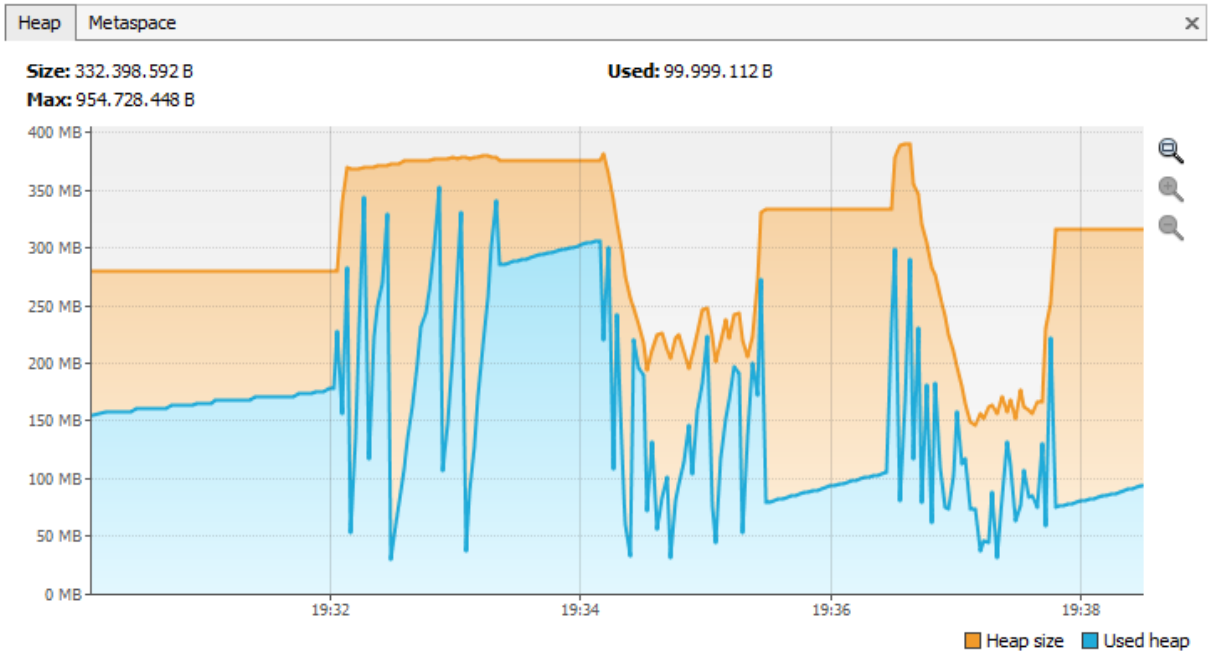
Figura 55 – RayTracer Service - Uso do processador - Modelo *multithread*



Fonte: Autor.

O uso de memória foi variou bastante durante a execução das simulações. Os picos de consumo chegaram até **350MB,** porém durante boa parte do tempo o uso manteve-se abaixo dos **200MB**, como pode ser visto na Figura 56.

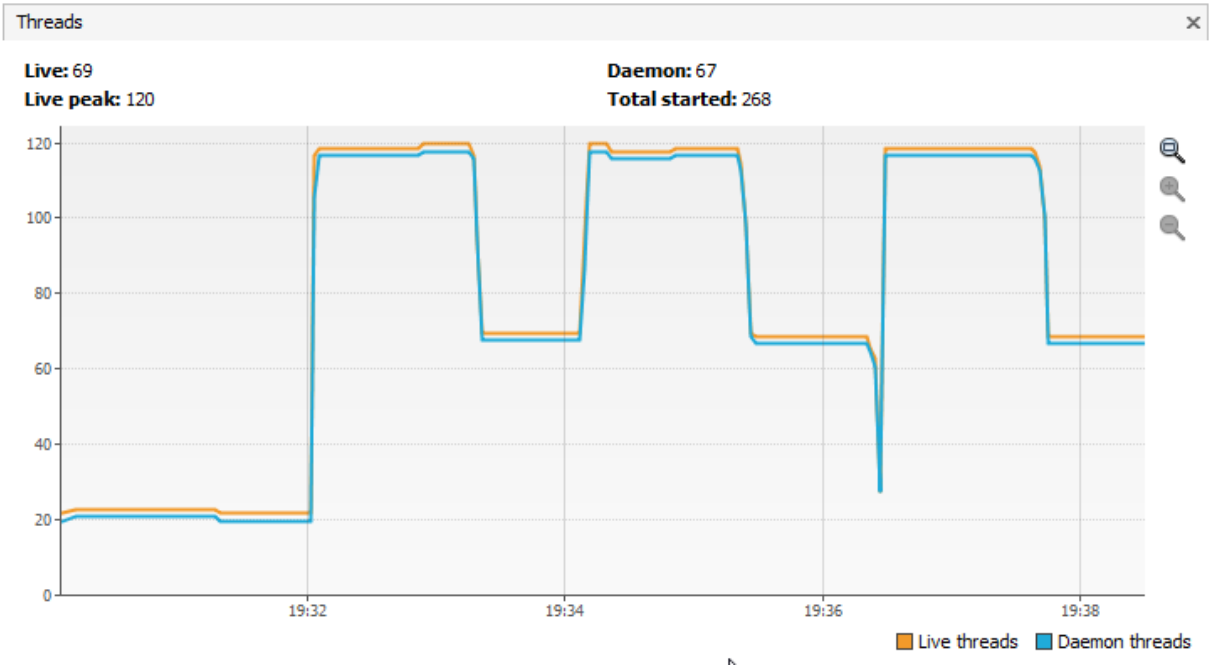
Figura 56 – RayTracer Service – Uso de memória – Modelo *multithread*



Fonte: Autor.

Durante a execução das simulações foram alocadas **120** *threads* e após seu termino são liberadas aproximadamente 50 *threads*, como pode ser visto na Figura 57.

Figura 57 – RayTracer Service – Alocação de *threads* – Modelo *multithread*



Fonte: Autor.

4.8.2.2 Modelo de atores

Este modelo levou um total de **21 segundos** para completar a execução da simulação. Na Tabela 4 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **2,38 requisições por segundo** além de um tempo médio de resposta de **18,1 segundos** por requisição. De um total de 50 requisições enviadas para a aplicação **38** finalizaram com erro devido ao alto tempo de resposta, isto representa **76%** do total de requisições.

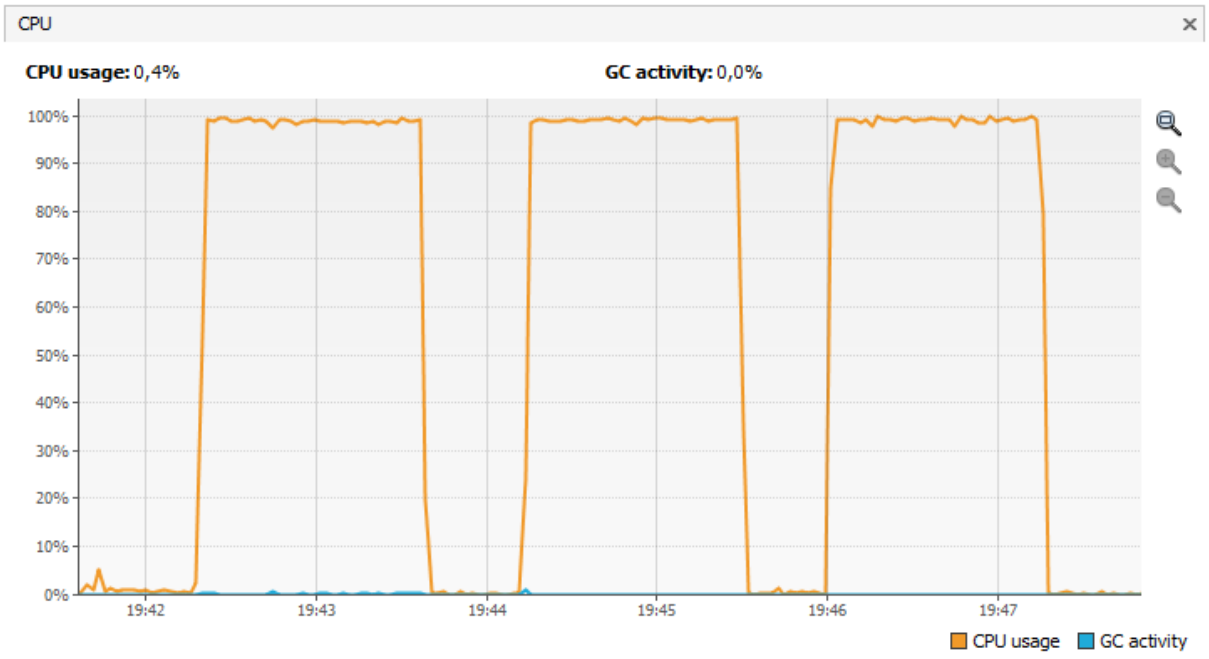
Tabela 4 – RayTracer Service – Tempos de reposta – Modelo de atores

Tipo	**Execuções**				**Tempo de Resposta (ms)**		
	Total	**OK**	**Com Erro**	**Req./s**	**Min.**	**Max.**	**Media**
Render	50	12	38	2,38	3.162	20.471	18.136

Fonte: Autor.

Na Figura 58 podemos acompanhar o uso do processador durante a execução dos testes, que como o modelo anterior, manteve-se em **100%** durante todo o tempo de simulação.

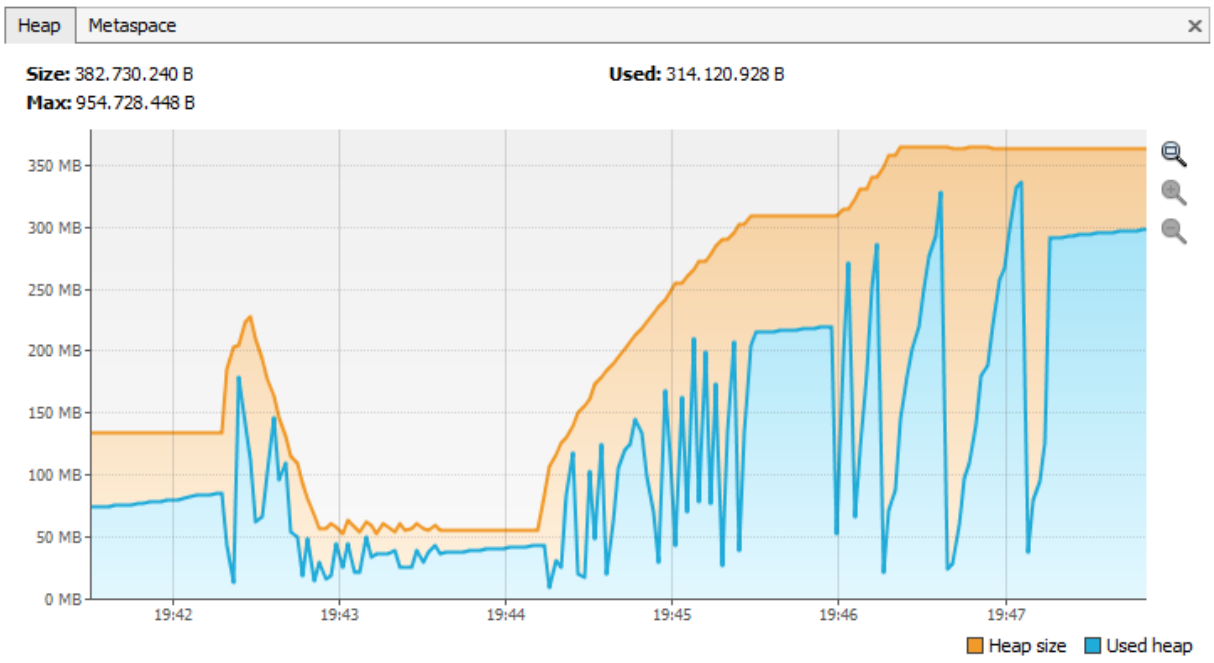
Figura 58 – RayTracer Service – Uso do processador – Modelo de atores



Fonte: Autor.

O uso de memória durante a simulação também teve uma alta variação e manteve-se abaixo de **200MB** em boa parte do tempo, atingindo picos próximos a **350MB** durante alguns momentos, como pode ser visto na Figura 59.

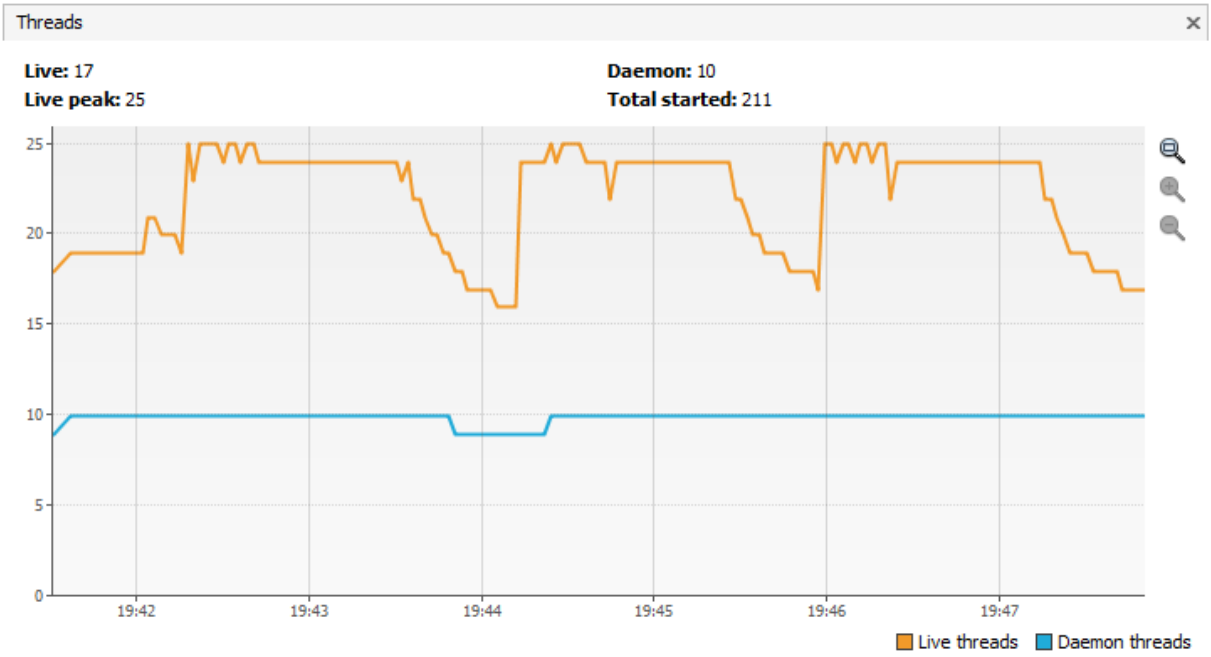
Figura 59 – RayTracer Service – Uso de memória – Modelo de atores



Fonte: Autor.

Durante a execução das simulações o número de *threads* alocadas variou de **16** até **25**, mantendo um valor relativamente estável durante toda a simulação, como pode ser visto na Figura 60.

Figura 60 – RayTracer Service – Alocação de *threads* – Modelo de atores



Fonte: Autor.

4.8.2.3 Comparativo dos resultados

Os testes realizados nesta aplicação apresentaram resultados favoráveis a utilização do modelo de atores mesmo em um cenário com baixa demanda de operações de E/S e com a presença de processamento altamente bloqueante. O modelo de atores conseguiu atender um maior número de requisições mesmo utilizando menos *threads* do que a implementação no modelo *multithread*. Isto ocorre pois no modelo de atores são utilizadas filas para limitar o número de tarefas concorrentes executando no processador, desta forma reduzindo a sobrecarga do processador na realização de operações de *context switch* por parte do sistema operacional e permitindo uma execução mais continua das tarefas.

Neste teste não houve nenhuma grande diferença em relação a consumo de recursos. Isto se deve ao fato de que a carga aplicada (número de usuários simultâneos) neste teste foi baixa em comparação ao teste realizado no *microservice* **TODO Service**, demonstrando assim

que grande parte do consumo de memória advém do número de *threads* executando simultaneamente.

4.8.3 Report Service

Neste subcapitulo serão apresentados os resultados coletados nas simulações para o *microservice* **Report Service**.

4.8.3.1 Modelo multithread

Este modelo levou um total de **61 segundos** para completar a execução da simulação. Na Tabela 5 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **0,82 requisições por segundo** além de um tempo médio de resposta de **60 segundos** por requisição. Todas as requisições enviadas para a aplicação falharam devido ao alto tempo de resposta.

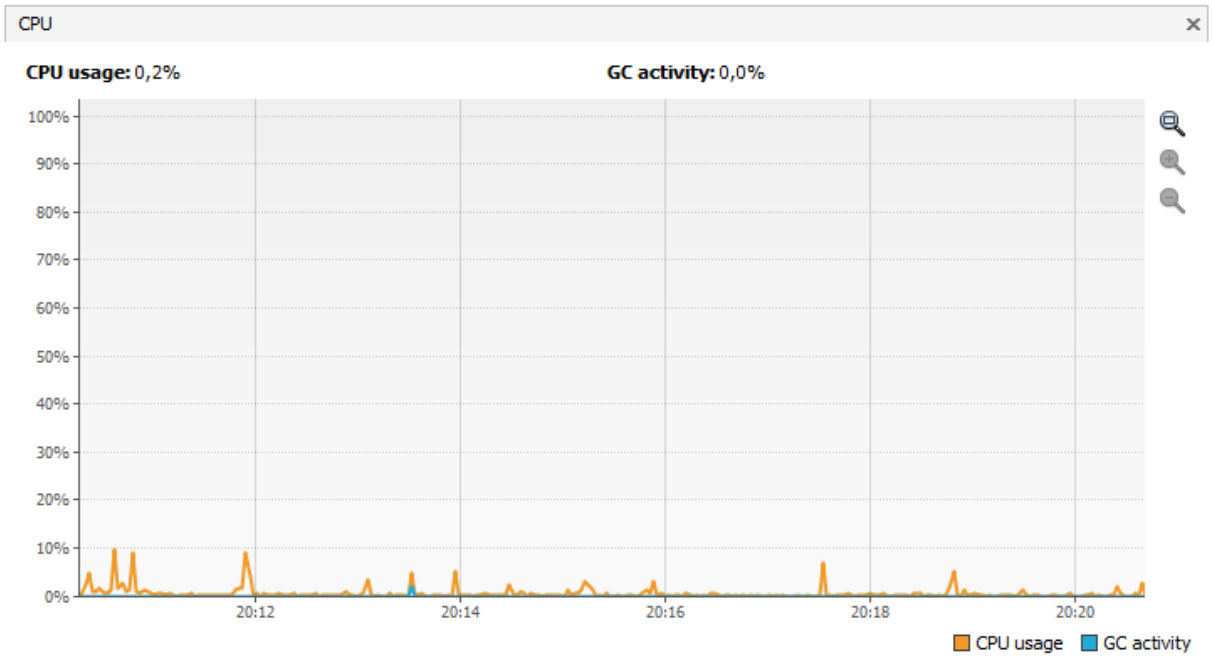
Tabela 5 – Report Service – Tempos de reposta – Modelo *multithread*

Tipo	**Execuções**				**Tempo de Resposta (ms)**		
	Total	**OK**	**Com Erro**	**Req./s**	**Min.**	**Max.**	**Media**
Report	50	0	50	0,82	60.003	60.018	60.008

Fonte: Autor.

Na Figura 61 podemos acompanhar o uso do processador durante a execução dos testes que chegou a um máximo de **10%**. O baixo uso do processador neste teste deve-se ao fato da aplicação trabalhar quase que somente com operações de E/S.

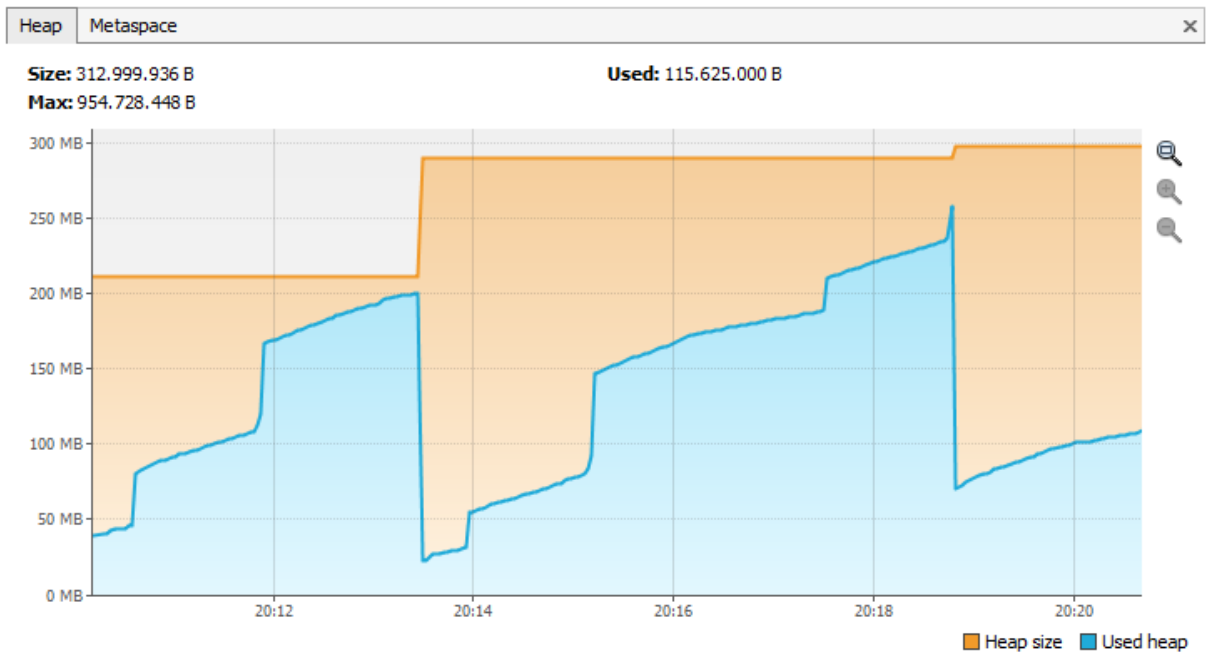
Figura 61– Report Service – Uso do processador – Modelo *multithread*



Fonte: Autor.

O uso de memória durante a execução das simulações chegou a até 2**50MB** em alguns picos, porém durante boa parte do tempo manteve-se abaixo dos **200MB**, como pode ser visto na Figura 62.

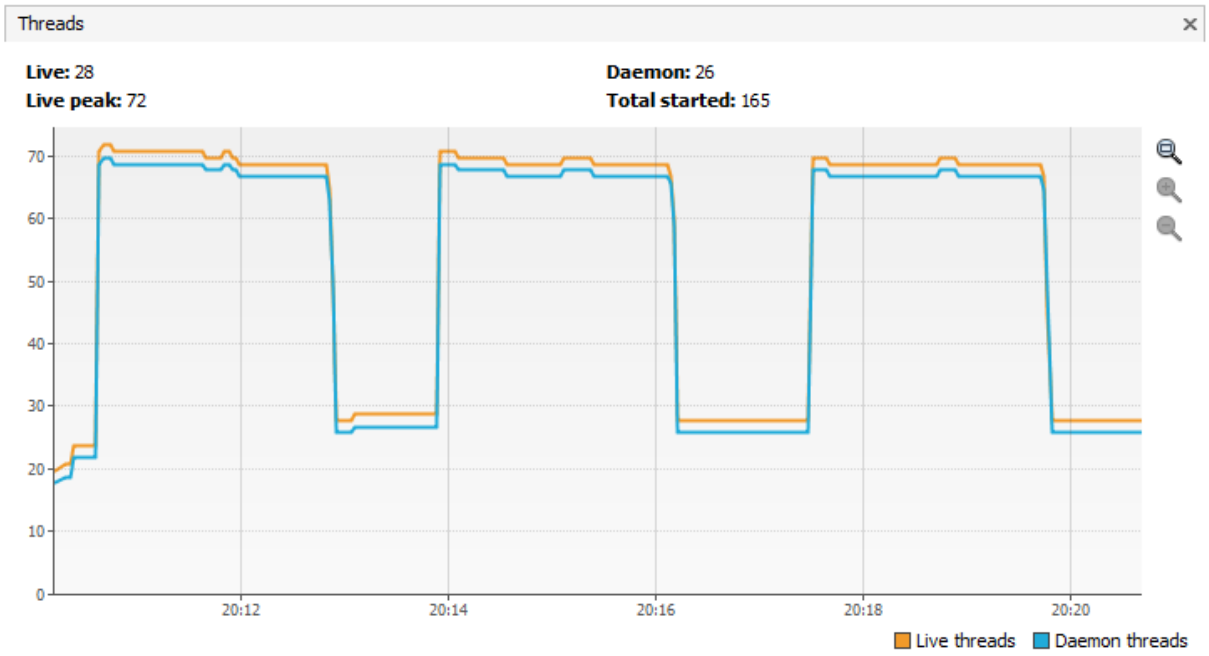
Figura 62 – Report Service – Uso de memória – Modelo *multithread*



Fonte: Autor.

Durante a execução das simulações foram alocadas **72** *threads*, como pode ser visto na Figura 63.

Figura 63 – Report Service – Alocação de *threads* – Modelo *multithread*



Fonte: Autor.

4.8.3.2 Modelo de atores

Este modelo levou um total de **21 segundos** para completar a execução da simulação. Na Tabela 6 estão representados os dados coletados pelo Gatling durante a execução da simulação. Podemos observar um *throughput* de **2,38 requisições por segundo** além de um tempo médio de resposta de **17,4 segundos** por requisição. De um total de 50 requisições enviadas para a aplicação **38** finalizaram com erro devido ao alto tempo de resposta, isto representa **76%** do total de requisições.

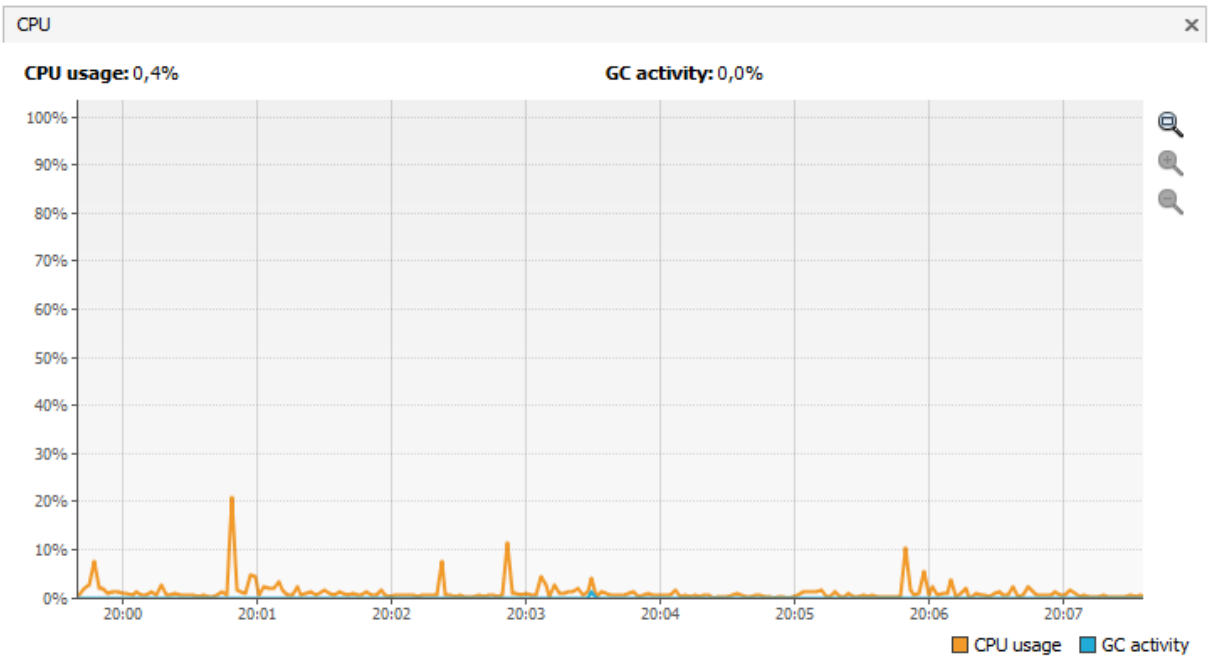
Tabela 6 – Report Service – Tempos de reposta – Modelo de atores

Tipo	Execuções				Tempo de Resposta (ms)		
	Total	OK	Com Erro	Req./s	Min.	Max.	Media
Render	50	12	38	2,38	507	20.480	17.388

Fonte: Autor.

Na Figura 64 podemos acompanhar o uso do processador durante a execução dos testes que se manteve baixo o tempo todo, chegando ao máximo de **20%** em alguns instantes.

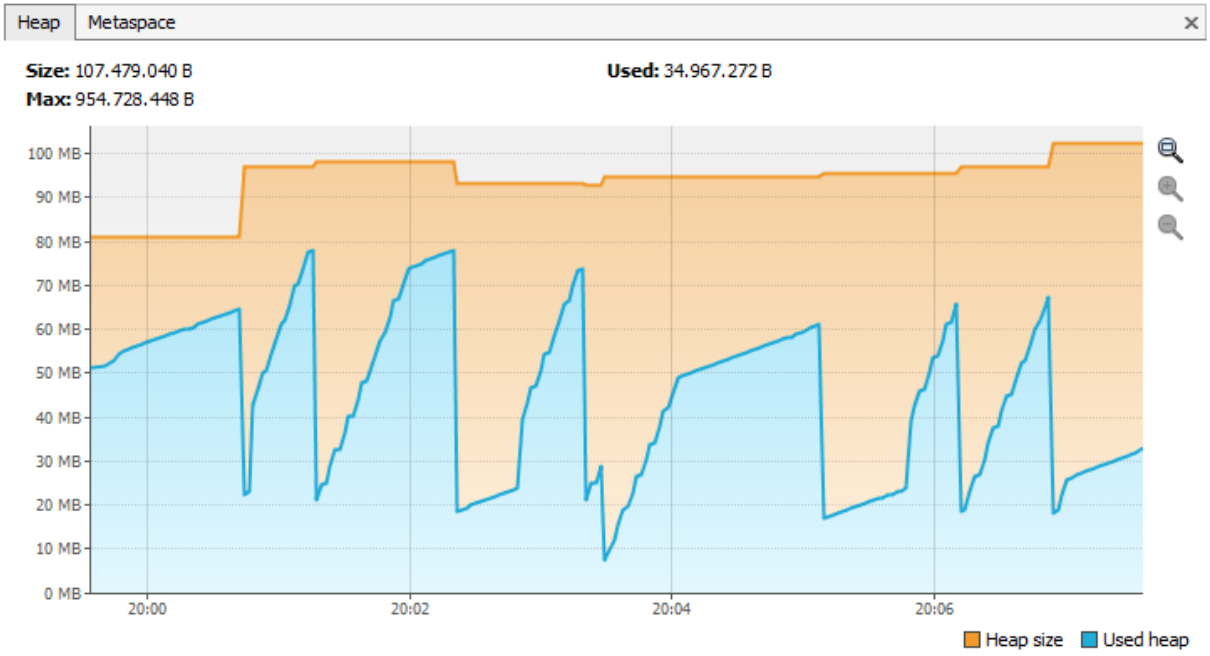
Figura 64 – Report Service – Uso do processador – Modelo de atores



Fonte: Autor.

O uso de memória durante a simulação manteve-se abaixo dos **80MB** o tempo todo como pode ser visto na Figura 65.

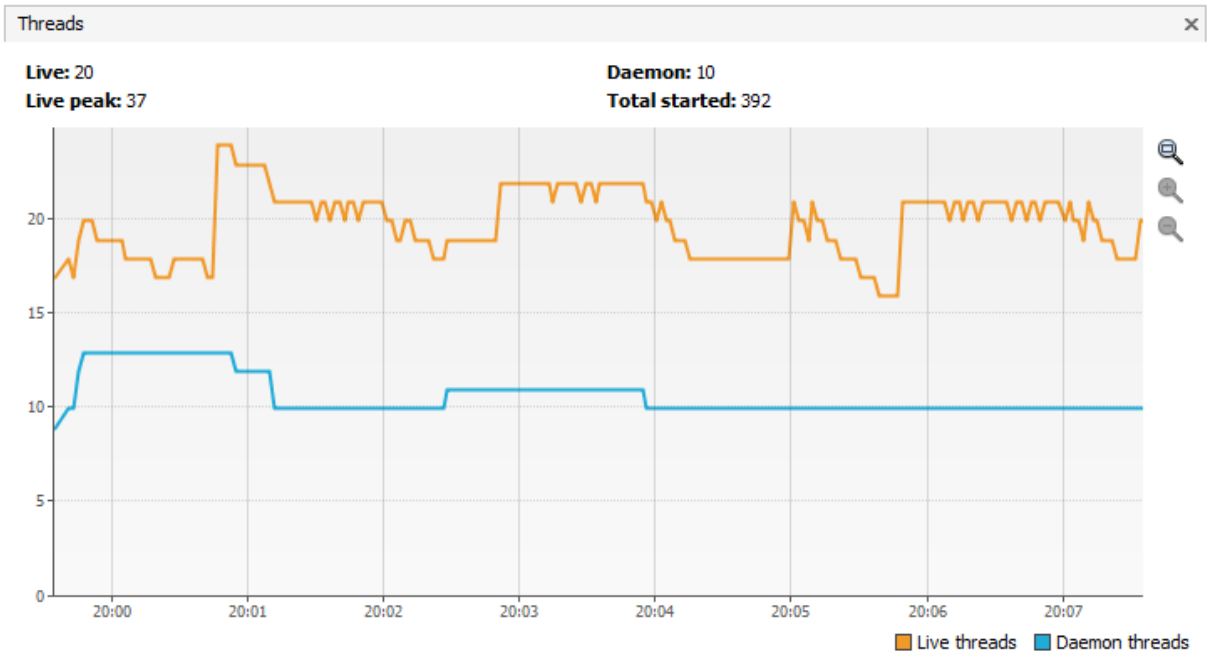
Figura 65 – Report Service – Uso de memória – Modelo de atores



Fonte: Autor.

Na Figura 66 pode ser vista a alocação de *threads* durante a execução das simulações. O número de *threads* alocadas permaneceu baixo, variando de **16** até **24** *threads* ativas.

Figura 66 – Report Service – Alocação de *threads* – Modelo de atores



Fonte: Autor.

4.8.3.3 Comparativo dos resultados

O resultado obtido no teste desta aplicação tornou-se altamente dependente do tempo de resposta dos outros dois *microservices* envolvidos, praticamente replicando os resultados obtidos nos testes do **RayTracer Service** (que era o serviço mais lento). Deste modo não foi possível realizar uma comparação direta entre a performance das duas soluções, sendo que o resultado deste teste foi influenciado pelos resultados superiores obtidos pelo modelo de atores nas outras implementações.

Em termos de consumo de recursos, o modelo de atores consegue trabalhar de maneira mais eficiente e com um menor consumo de memória devido ao baixo número de *threads* alocadas. Essa diferença de consumo tende a aumentar conforme a aplicação sofra um maior número de acessos concorrentes, pois o uso de operações de E/S assíncronas permite manter um número reduzido de *threads* operando.

5 CONSIDERAÇÕES FINAIS

Durante a pesquisa bibliográfica realizada neste trabalho foi possível identificar vários aspectos que são fundamentais para a implementação de aplicações de alta performance. Observar os requisitos de cada aplicação é fundamental para possibilitar o desenvolvimento de uma arquitetura que faça uso dos recursos existentes nas plataformas atuais de maneira eficiente. É importante estar atento a detalhes de implementação como por exemplo uso de processos ou *threads*, operações de E/S síncronas ou assíncronas e também, sempre que possível, preferir a implementação de algoritmos que permitam escalar de maneira distribuída, seja através da paralelização ou execução de vários fluxos de trabalho concorrentes.

Além da pesquisa bibliográfica sobre o tema, também foram previstos nos objetivos deste trabalho a definição e implementação de um conjunto de cenários com diferentes tipos de processamento, fazendo o uso diferentes modelos de concorrência e arquitetura. A implementação destes protótipos teve o objetivo de permitir a execução de simulações de carga para realizar a coleta de dados relativos a performance e uso de recursos disponíveis no hardware. Os dados provenientes das simulações permitiram que fosse realizada uma comparação quantitativa entre os resultados obtidos em ambas as implementações de cada um dos cenários.

Os comparativos permitiram identificar o potencial do modelo de atores em conjunto com operações de E/S assíncronas para a criação de aplicativos que requisitem alta performance, responsividade e baixa latência. Para as aplicações que são disponibilizadas na nuvem, o uso dos recursos computacionais disponíveis de maneira eficiente e controlada pode representar uma redução de custos significativa com servidores e manutenção dos mesmos.

Foi constatada de maneira pratica a eficiência do modelo de atores e do Akka em ambientes de alta concorrência, principalmente em aplicações com alta dependência de operações de E/S. É importante ressaltar que algumas ferramentas utilizadas na implementação dos protótipos no modelo de atores, como por exemplo o Akka HTTP, ainda estão em estágio experimental e possivelmente irão receber ajustes relacionados a performance até o lançamento de sua versão estável.

Os resultados obtidos na realização deste estudo permitiram identificar a importância da arquitetura e da qualidade da implementação dos componentes envolvidos no desenvolvimento de aplicações, principalmente as que irão atuar na nuvem, aonde o nível de acessos concorrentes tende a ser maior. Um ponto importante a ser observado é que este trabalho fez uso das configurações padrão em todas as ferramentas utilizadas, portanto existe margem para otimização em ambas as implementações através do ajuste dos parâmetros de configuração.

Durante a implementação dos protótipos foram realizadas pesquisas em diversas fontes em busca de exemplos de códigos, como por exemplo na documentação das ferramentas e também na internet de maneira geral. Um ponto interessante observado na documentação oficial do Akka é que em diversas seções são exemplificadas boas e más práticas na implementação dos atores e seus componentes, deixando visíveis ao programador questões que podem vir a gerar problemas.

O modelo de programação imposto pelo Akka foge bastante do que é considerado comum em uma linguagem como o Java e isto pode representar um grande obstáculo em sua adoção, pelo menos em empresas mais conservadores. Apesar disto ele pode ser inserido de maneira gradativa em uma aplicação já existente, realizando a integração de seu código através do uso de padrões como o *ask pattern* descrito anteriormente. Porém para poder explorar todo o potencial que a ferramenta oferece o ideal é que todo ferramental envolvido possa trabalhar com computações assíncronas.

Isso nem sempre é possível pois muitas bibliotecas também não possuem nenhuma implementação similar no modelo de atores ou pelo menos que operem de maneira assíncrona para serem utilizadas diretamente junto com o Akka. Na maioria dos casos é necessário encapsular a funcionalidade de bibliotecas existentes em atores, sempre tomando cuidado para não quebrar as garantias que cada ator deve oferecer em relação ao encapsulamento, imutabilidade e operações bloqueantes.

Um exemplo disto seria a utilização de *drivers* que implementam o padrão JDBC em conjunto com o Akka. Para isto seria necessário executar as operações do *driver* através de um *thread pool* dedicado para não bloquear as *threads* utilizadas pelo Akka na execução do sistema de atores. Caso isso não seja feito e as operações sejam executadas diretamente em um ator, a performance de toda a aplicação ira degradar visto que o Akka trabalha com um baixo número de *threads* alocadas.

O padrão *Servlet*, sob o qual o Spring MVC executa, disponibilizou suporte para a utilização de E/S assíncrona somente em sua última versão (3.1) lançada por volta de maio de 2013, muito antes disto, diversas ferramentas já disponibilizavam este recurso. As tecnologias padronizadas pela plataforma Java (como as *Servlets* e o JDBC) demoram muito para serem atualizadas, e quando são atualizados, na grande maioria das vezes já não possuem muita relevância pois outras ferramentas já dominaram o mercado ou são tecnicamente superiores as disponibilizadas pelo padrão. Apesar disto, estas evoluções permitem que as aplicações que fazem uso de alguma destas tecnologias utilizem os novos recursos, na maioria das vezes sem abrir mão da compatibilidade com o código já existente.

A realização deste estudo permitiu preencher as lacunas existentes em diversos testes similares encontrados na internet, aonde os comparativos entre os modelos citados neste trabalho não abrangiam o uso de elementos comuns a quase todas as aplicações, como por exemplo o uso de algum banco de dados ou realização de comunicação com um serviço externo.

Por fim, através das pesquisas e práticas realizadas, este trabalho conseguiu atingir todos os objetivos definidos em sua proposta. Todos os códigos-fonte criados durante a implementação dos protótipos e documentos gerados durante realização deste trabalho estão disponíveis publicamente no GitHub[14].

[14] Disponível em <https://github.com/mauriciocc/tcc>.

REFERÊNCIAS

BELTRAN, Vicenc, David CARRERA, Jordi TORRES, e Eduard AYGUADÉ. **Evaluating the Scalability of Java Event-Driven Web Servers**. Barcelona: Computer Architecture Department, Technical University of Catalonia, 2004.

CHEMIN, Beatris Francisca. **Manual da Univates para trabalhos acadêmicos: planejamento, elaboração e apresentação**. 3. ed. Lajeado: Univates, 2015.

ENDREI, Mark, et al. **Patterns: Service-Oriented Architecture and Web**. New York: IBM Redbooks, 2004.

ERB, Benjamin. **Concurrent Programming for Scalable Web Architectures**. Ulm: Ulm University, 2012.

ERL, Thomas. **Service-Oriented Architecture: Concepts, Technology, and Design**. Upper Saddle River: Prentice Hall, 2005.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. IRVINE: UNIVERSITY OF CALIFORNIA, 2000.

GOOGLE. **Google Cloud**. 2016. Disponível em <https://cloud.google.com/products>. Acesso em: 3 set. 2016.

LAUER, Hugh C., e Roger M. NEEDHAM. **On the Duality of Operating System Structures**. California: Palo Alto, 1978.

LIU, Henry H. **Software Performance and Scalability: A Quantitative Approach**. Hoboken, New Jersey: John Wiley & Sons, 2009.

MARCONI, Marina de Andrade, e Eva Maria LAKATOS. **Fundamentos de Metodologia Científica**. 5. ed. São Paulo: Editora Atlas, 2003.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. Sebastopol: O'Reilly Media, 2015.

ODERSKY, Martin. **Scala By Example**. Lausanne: EPFL, 2014.

PACHECO, Peter S. **An Introduction to Parallel Programming**. Burlington: Elsevier, 2011.

PIZZANI, Luciana, Rosemary Cristina da SILVA, Suzelei Faria BELLO, e Maria Cristina Piumbato Innocentini HAYASHI. A ARTE DA PESQUISA BIBLIOGRÁFICA NA BUSCA DO CONHECIMENTO. **Revista Digital de Biblioteconomia e Ciência da Informação**, São Carlos, v. 10, n.1, p. 53-66, jul. 2012.

POSTGRESQL. **POSTGRESQL**. 2016. Disponível em <https://www.postgresql.org/about>. Acesso em: 3 set. 2016.

PRESSMAN, Roger S. **Software Engineering Practitioner's Approach**. New York: McGraw-Hill, 2009.

ROESTENBURG, Raymond, Rob BAKKER, e and Rob WILLIAMS. **Akka in Action**. Greenwich: Manning Publications, 2015.

SANTOS, Antônio Raimundo dos. **Metodologia científica: a construção do conhecimento**. Rio de Janeiro: DP & A, 1999.

SCHILDT, Herbert. **Java: A Beginner's Guide**. New York: McGraw-Hill/Osborne, 2005.

SOMMERVILLE, Ian. **Engenharia de Software**. Upper Saddle River: PEARSON EDUCATION, 2007.

SUTTER, Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. **Dr. Dobb's Journal**, 2005.

SUTTER, Herb, e James LARUS. Sofware and the Concurrency Revolution. **Queue**, p. 54-62, 2005.

SWEBOK. **Guide to the Software Engineering**. Pierre Bourque, École de technologie supérieure (ÉTS), 2014.

TANENBAUM, ANDREW S. **Sistemas Operacionais Modernos**. 3. ed. São Paulo: Pearson Education do Brasil, 2010.

TOLLEDO, Rodrigo. **Thoughtworks**. 2014. Disponível em <https://www.thoughtworks.com/insights/blog/gatling-take-your-performance-tests-next-level>. Acesso em: 5 set. 2016.

VELTE, Anthony T., Toby J. VELTE, e Robert ELSENPETER. **Cloud Computing, A Practical Approach**. New York: McGraw-Hill, 2009.

VERNON, Vaughn. **Reactive Messaging Patterns with the Actor Model**. New York: Addison-Wesley, 2015.

WAINER, Jacques. **Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação**. Campinas: Unicamp, 2007.

WALLS, CRAIG. **Spring in Action**. Shelter Island: MANNING, 2015.

WEBBER, Jim, Savas PARASTATIDIS, e Ian ROBINSON. **REST in Practice**. Sebastopol: O'Reilly, 2010.