Marlon Diego Casagranda França

Desempenho na comunicação entre serviços Web

Araquari – SC

Novembro de 2017

Marlon Diego Casagranda França

Desempenho na comunicação entre serviços Web

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do grau de bacharel em Sistemas de Informação do Instituto Federal Catarinense.

Instituto Federal Catarinense – IFC

Câmpus Araquari

Bacharelado em Sistemas de Informação

Orientador: Prof. Dr. Eduardo da Silva

Araquari – SC

Novembro de 2017

Marlon Diego Casagranda França
Desempenho na comunicação
entre serviços Web/ Marlon Diego Casagranda França. – Araquari – SC, Novembro de 2017-
63 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Eduardo da Silva

Monografia (Graduação) – Instituto Federal Catarinense – IFC
Câmpus Araquari
Bacharelado em Sistemas de Informação, Novembro de 2017.

1. Comunicação. 2. REST. 2. GraphQL. 3. Desempenho. I. Orientador. II. Instituto Federal Catarinense. III. Câmpus Araquari. IV. Título

Marlon Diego Casagranda França

Desempenho na comunicação entre serviços Web

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do grau de bacharel em Sistemas de Informação do Instituto Federal Catarinense.

Trabalho aprovado. Araquari – SC, 24 de novembro de 2016:

Prof. Dr. Eduardo da Silva
Orientador

Professor
Convidado 1

Professor
Convidado 2

Araquari – SC
Novembro de 2017

Dedicatória do trabalho de conclusão que deve ser algo breve e conciso.

Agradecimentos

Página com os agradecimentos do autor à pessoas importantes para a realização do trabalho.

"Colocar alguma frase importante que,
tenha motivado o autor no decorrer do desenvolvimento
do trabalho."
(Referência de local da frase)

Resumo

Com a ascensão de dispositivos móveis que reivindicam uma parcela cada vez maior do tráfego de internet, otimizar o desempenho da busca de dados torna-se muito importante. A arquitetura REST tem sido, durante muito tempo, a solução mais comum ao desenvolver APIs Web, mas o GraphQL tem se tornado, em tempos recentes, uma alternativa atrativa. O presente trabalho aborda as principais técnicas e conhecimentos sobre comunicação entre aplicações Web. Também é realizado um experimento, tendo como principal parâmetro de comparação o desempenho de APIs implementando REST e GraphQL quando realizado buscas de objetos aninhados. Protótipos de cada API foram implementados e usados para realizar medições do desempenho de cada técnica. O GraphQL apresentou um melhor desempenho em praticamente todos os cenários executados no experimento.

Palavras-chave: Comunicação. REST. GraphQL. Desempenho.

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

IoT	Internet of Things
API	Application Program Interface
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
URL	Uniform Resource Locator
IPC	Interprocess Communications
CPU	Central Processing Unit
SOA	Service Oriented Architecture
TI	Tecnologia da Informação
W3C	World Wide Web Consortium
XML	Extensible Markup Language
JSON	JavaScript Object Notation
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
HATEOAS	Hypermedia as the Engine of Application State
WMS	Warehouse Management System
ERP	Enterprise Resource Planning
UUID	Universally unique identifier

Lista de símbolos

Δ	Letra grega Delta

Sumário

1 Introdução

Em um mundo cada vez mais interconectado, usuários constantemente exigem maior disponibilidade de informações através da Web. Essas informações não devem apenas estar acessíveis, como também é crucial que seja rápido o tempo para acessá-las. Com a expansão da Internet para dispositivos dos mais diversos tipos, tais como *smartphones*, *tablets* e dispositivos de IoT, questões como tempo de carregamento e consumo de banda vem se tornando fatores cada mais mais debatidos.

O uso de dispositivos móveis para acessar a Internet cresceu 63% somente em 2016, como aponta o relatório de Previsão Global de Tráfego de Dados Móveis, elaborado pela Cisco. O tráfego de dados móveis passou de 4,4 exabytes [1] por mês em 2015, para uma média mensal de 7,2 exabytes em 2016. Esse aumento foi resultado de uma adição de cerca 429 milhões de novos dispositivos móveis à rede, sendo os *smartphones* responsáveis pela maior parte deste crescimento (Cisco, 2017).

Conforme Work (2011), usuários tendem a dar mais importância à velocidade em que recebem a informação do que a estética que ela é apresentada. O tempo de carregamento é um fator decisivo para a permanência em uma página Web, uma vez que a maioria dos usuários estão dispostos a aguardar de 6 a 10 segundos antes de abandonar qualquer página. Cada segundo de atraso pode resultar em uma redução de até 7% nas taxas de conversão, o que para um *e-commerce* que fatura mensalmente R$ 100.000,00 por exemplo, pode potencialmente custar R$ 84.000,00 em vendas não efetuadas em um ano.

Para atender estas novas demandas, nos últimos anos houve uma mudança para um modelo de computação chamado cliente/servidor, que aborda as falhas da computação centralizada. Claramente, o modelo de computação centralizado permanece válido em certos ambientes de negócios, no entanto, apesar de muitos benefícios, a computação centralizada é reconhecida como tendo promovido uma cultura de gerenciamento de informações que não conseguiu atender as necessidades de seus clientes.

Em 2010, ocorreu um grande avanço no número de APIs públicas impulsionado pela transição no modelo de comunicação entre aplicações distribuídas, em que estas passaram a utilizar amplamente o protocolo HTTP e o modelo cliente/servidor para a troca de informações através da Web (MASO, 2016). A adoção do REST (REpresentational State Transfer) como o método predominante para construir APIs públicas tem ofuscado qualquer outra tecnologia ou abordagem nos últimos anos. Embora várias alternativas (principalmente SOAP) ainda estejam presentes no mercado, adeptos do modelo SOA para construção de aplicações tomaram uma posição definitiva contra eles e optaram por REST

[1] Um exabyte é equivalente a um bilhão de gigabytes e mil petabytes.

como modelo de comunicação e JSON como seu formato de mensagem (LENSMAR, 2013).

Segundo Paliari (2012), REST é cada vez mais usado como alternativa ao "já antigo" SOAP em que a principal crítica é a burocracia, algo que o REST possui em uma escala muito menor. REST é baseado no *design* do protocolo HTTP, que já possui diversos mecanismos embutidos para representar recursos como código de *status*, representação de tipos de conteúdo, cabeçalhos, etc. O principal nesta arquitetura são as URLs do sistema e os *resources* [2], aproveitando os métodos HTTP para se comunicar.

Entretanto, com o aumento do uso do REST como modelo de comunicação das APIs, algumas limitações foram expostas, prejudicando o desempenho destas APIs em aspectos cruciais. Clientes com rotinas complexas necessitam realizar consultas complexas, buscando objetos aninhados com diversos relacionamentos. Como uma API REST expõe exclusivamente recursos, é necessário executar diversas buscas no servidor para que algumas rotinas possam ser processadas pelo cliente, uma vez que nem todas as informações necessárias estão presentes na resposta de uma única consulta.

Além disso, em tais cenários são necessárias diversas chamadas na API, pois uma só consulta pode não conter toda informação necessária. Ainda, grande parte destas chamadas irão retornar dados desnecessários ao contexto da rotina que a executou. Esta prática é conhecida como *over-fetching* e ocorre pois é responsabilidade do servidor da API montar o conteúdo da resposta, resultando no tráfego de dados desnecessários.

Foram propostas múltiplas soluções para aumentar a eficiência na busca de dados, algumas em relação aos formatos de consulta e resposta das requisições, enquanto outras estão otimizando o número de solicitações na rede. Uma tendência recente envolve um modelo de consultas declarativas de dados, em que as aplicações clientes especificam quais dados precisam, em vez de buscar tudo a partir de um local específico definido por um URL. Desta forma, estes modelos otimizaram a comunicação com os servidores para obter os dados de forma eficiente: esta é a proposta do GraphQL.

Construído pelos desenvolvedores do Facebook para atender as necessidades internas da rede social em 2012, o GraphQL foi lançado ao público em geral em 2015, e já vem ganhando diversos adeptos. Com a promessa de mitigar alguns problemas crônicos do *design* do REST, como versionamento de APIs, múltiplas viagens de ida e volta e excesso de dados trafegados na rede, a abordagem do Facebook já vem sendo usada por diversas empresas.

REST é de fato, o modelo mais utilizado para comunicação entre cliente e servidor nas aplicações atuais. Este trabalho foca em identificar as diferenças em termos de tempo de carregamento, quantidade de dados trafegados e consumo de recursos entre aplicações REST e o ainda pouco conhecido GraphQL. Para isso, serão criados dois protótipos de

[2] resource é um recurso, entidade

API, um implementando o *design* do REST e outro utilizando GraphQL como mecanismo para responder às consultas. O desempenho das duas APIs será então medido, com base em métricas quantitativas e será apresentado a análise dos resultados obtidos.

O restante deste trabalho está organizado da seguinte forma: o capítulo 2 apresenta a evolução das arquiteturas para construção de aplicações; o capítulo 3 aborda os modelos de comunicação entre aplicações, com informações sobre a evolução destes modelos e conceitos em geral, com ênfase nas duas tecnologias analisadas; o capítulo 4 apresenta o ambiente utilizado, as hipóteses levantadas, e detalhes das ferramentas e métricas utilizadas para a análise dos resultados; o capítulo 4 se resume na apresentação e análise dos resultados obtidos. Por fim, e não menos importante, será apresentada a conclusão.

2 Fundamentos

Neste capítulo serão apresentadas algumas das pricipais soluções utilizadas para a construção de aplicações. Ele mostra um pouco da evolução nas arquiteturas de aplicações, partindo de modelos simples como o IPC, passando pelas arquiteturas de duas e três camadas, até as arquiteturas distribuídas, como o SOA.

2.1 IPC - Comunicação entre processos

Nem sempre um programa sequencial é a melhor solução para um determinado problema. Muitas vezes, as implementações são estruturadas na forma de várias tarefas inter-dependentes que cooperam entre si para atingir os objetivos da aplicação (MAZIERO, 2013). Diversos sistemas operacionais fornecem mecanismos para viabilizar a comunicação e o compartilhamento de dados entre aplicações. Coletivamente, as atividades habilitadas por estes mecanismos são chamadas de Interprocess Communications (IPC) (Microsoft, 2008).

IPC consiste em mecanismos que garantem a comunicação entre processos concorrentes e os acessos aos recursos compartilhados. Algumas formas de IPC facilitam a divisão de trabalho entre diversos processos especialistas, enquanto outras facilitam esta divisão entre computadores dentro de uma rede.

Normalmente, os aplicativos que fazem parte de uma comunicação através de IPC são categorizados como clientes ou servidores. Um cliente é um aplicativo ou um processo que solicita um serviço de alguma outra aplicação ou processo. Por outro lado um servidor é um aplicativo ou um processo que responde a uma solicitação de cliente. Muitas aplicações agem como um cliente e servidor, dependendo da situação (Microsoft, 2008).

A figura 1 mostra como ocorre a comunicação entre processos (P1 P2, P3 e P4). Esta troca de informação pode acontecer de duas maneiras: em duas etapas, ou de forma direta. A comunicação em duas etapas, ilustrada na figura através de flechas, envolve um processo coordenador, que pode ser um interpretador em Python por exemplo, utilizado para dar início ao *workflow*. Na figura, a comunicação em duas etapas ocorre entre os processos P1 e P2, fazendo uso de um processo coordenador. A comunicação direta é ilustrada na figura ligada por linhas pontilhadas e não necessita de intermediação de nenhum processo. Esta maneira de comunicação pode ser executada através de diversas formas, entre elas: *Pipes*, *Shared Memory*, *Mapped Memory* e arquivos compartilhados.

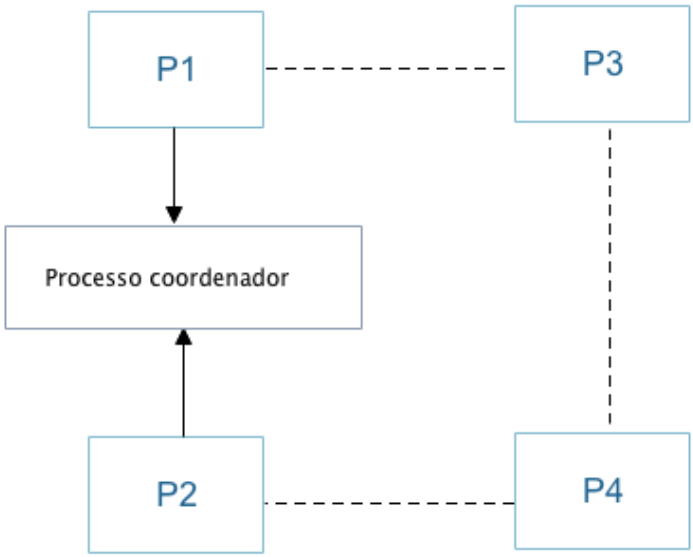

Figura 1 – Características dos mecanismos de comunicação

Fonte: Maziero (2013)

2.2 Cliente/Servidor

Também conhecido como arquitetura de duas camadas, o modelo cliente/servidor consiste em uma arquitetura em que a camada de apresentação se encontra no cliente e a camada de dados encontra-se no servidor. Esta separação se opõe ao modelo centralizado amplamente utilizado até seu surgimento.

O processamento dos dados é dividido em duas partes distintas. Uma parte é a requerente de dados (cliente), e a outra parte é a provedora dos dados (servidor). O cliente envia durante sua execução uma ou mais solicitações ao servidor para realizar alguma tarefa específica. É de responsabilidade do cliente tanto apresentar as informações para o usuário, quanto executar as regras de negócio necessárias à aplicação. O servidor é responsável por armazenar os dados e fornecer um meio para que o cliente os consulte (CLASS, 2013).

Desde a década de 1990, fornecedores de *software* desenvolvem e trazem ao mercado muitas ferramentas para simplificar o desenvolvimento de aplicativos para a arquitetura cliente/servidor de duas camadas. Algumas das mais conhecidas são: Microsoft Visual Basic, Delphi da Borland e PowerBuilder da Sybase. Estas ferramentas combinadas com uma comunidade ativa de desenvolvedores, fizeram da abordagem de duas camadas cliente/servidor uma solução econômica para criação diversos tipos de sistemas.

Desde então, aplicações *desktop* comunicando-se com o servidor de banco de dados

se tornaram um caso de uso normal. A maior parte da lógica de negócios foi incorporada dentro da aplicação *desktop*. Portanto, esse estilo de clientes na aquitetura de duas camadas também foi chamado de *fat clients*.

A figura 2 ilustra como a arquitetura em duas camadas funciona, mantendo uma comunicação direta entre cliente e servidor, sem intermediários entre as duas pontas. Visto que neste modelo as regras de negócio estão presentes na camada de aplicação, ele é frequentemente aplicado em ambientes homogêneos. Como a camada de banco de dados e a camada de aplicação estão fisicamente próximos, este modelo também oferece um bom desempenho para as aplicações.

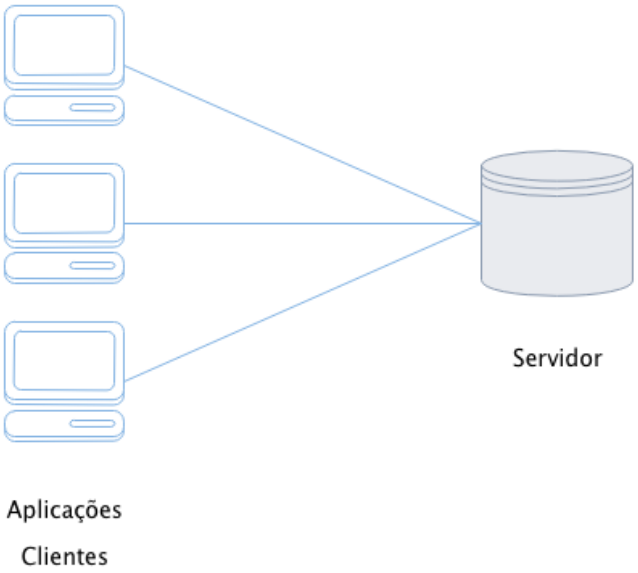

Figura 2 – Comunicação em duas camadas

Por outro lado, o modelo cliente servidor em duas cadamas possui grandes desafios de escalabilidade. Quando múltiplos usúarios executam requisições simultâneas, a aplicação perde muito desempenho, dado ao fato que cada cliente precisa de conexão de banco de dados separadas e processamento de CPU exclusivo para executar as requisições. Entretando, um dos maiores problemas na arquitetura em duas camadas ocorre quando há mudanças na estrutura do banco de dados. A maioria das aplicações cliente dependem da estrutura do banco de dados, o que impede qualquer remodelagem deste, criando um problema de estruturas legadas, e muitas vezes subutilizadas (THATIKONDA, 2011).

O modelo de aplicações cliente/servidor foi substituido pelo modelo de três camadas, muito mais eficiente, e que fornece uma maneira de dividir as funcionalidades envolvidas na manutenção e apresentação de uma aplicação.

2.3 Aplicações Monolíticas

Conforme Lewis (2014) explica, uma aplicação monolítica é construída como uma única unidade de *software*. Estas aplicações são construídas em três camadas: um banco de dados (que consiste em tabelas geralmente em um sistema de gerenciamento de banco de dados), uma interface de usuário do lado do cliente (consistindo de páginas executando em um navegador ou aplicativos móveis) e um lado do servidor de aplicação. Esta aplicação do lado do servidor trata as requisições HTTP, executa as regras de negócio, recupera e atualiza registros do banco de dados e constrói as respostas para serem enviadas às aplicações clientes.

Estas camadas representam uma aplicação monolítica - um único executável lógico, e para fazer alterações no sistema, é necessário criar e implantar uma versão atualizada do aplicativo do lado do servidor. Uma aplicação monolítica é autônoma e independente de outras aplicações. A filosofia do projeto consiste em um aplicativo que não é responsável apenas por uma determinada tarefa, mas que também pode executar todos os passos necessários para completar uma determinada função, como é mostrado na figura 3.

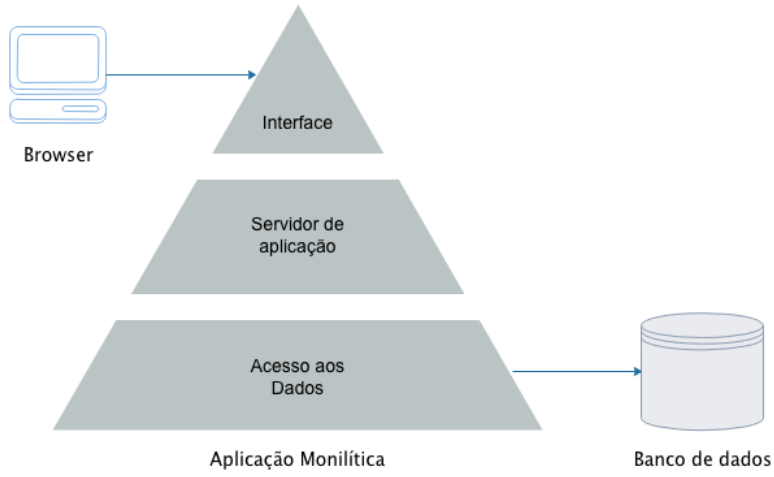

Figura 3 – Aplicação monolítica clássica em 3 camadas

Hicks (2017) afirma que o modelo monolítico é uma maneira natural para a evolução de uma aplicação. A maioria das aplicações começam com um único objetivo, ou um pequeno número de objetivos relacionados. Ao longo do tempo, novas funcionalidades são adicionados ao aplicativo para suportar as necessidades do negócio. Entretando, as aplicações monolíticas apresentam algumas desvantagens, sendo algumas delas:

a) Um único ponto de falha pode comprometer o funcionamento correto de todos os módulos do sistema.

b) Baixa estalabilidade dado a necessidade de copiar toda a *stack* [1] para o escalonamento horizontal.

c) Base de código volumosa, a medida que a aplicação cresce, uma vez que toda a regra de negócio se encontra em uma só base.

d) É necessária muita comunicação para que várias equipes de desenvolvedores trabalhem em paralelo. Esta sobrecarga diminui o ritmo de desenvolvimento.

Contudo, com a popularização dos *frameworks* Javascript como Angular e Backbone, houve um movimento de desacoplamento da camada de visualização nas aplicações monolíticas. Foram criadas então as chamadas aplicações *frontend*, responsáveis pela camada de apresentação, e sendo executadas em dispositivos móveis como *smartphones* e *tables* ou nos *browsers desktop*.

Esse movimento criou um modelo monolítico de duas camadas, ilustrado na figura 4. A remoção da interdependência entre as camadas de interface e o servidor de aplicação facilitou o escalonamento e a escalabilidade de cada uma das partes. Esta separação de conceitos foi o primeiro passo em direção a uma arquitetura mais orientada a serviços.

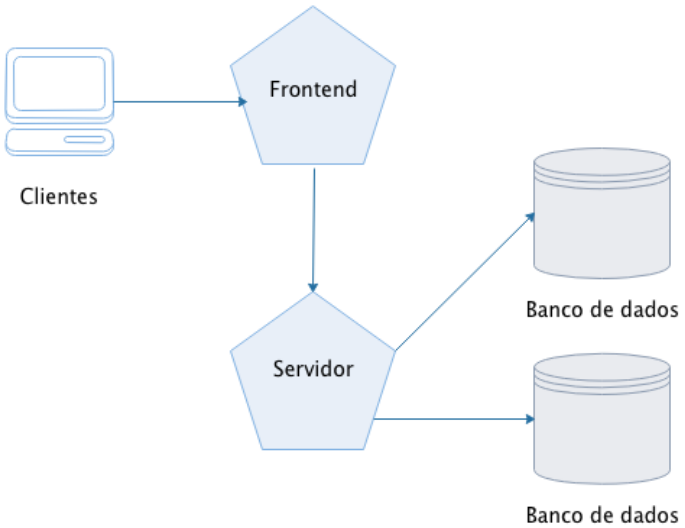

Figura 4 – Modelo monolítico de duas camadas

2.4 SOA - Service Oriented Architecture

Everware-CBDI (2014) define SOA como políticas, práticas e *frameworks* que permitem que rotinas de aplicações sejam fornecidas e consumidas como conjuntos de

[1] Conjunto de ferramentas que formam uma solução de software

serviços publicados em uma granularidade relevante para o consumidor do serviço. Estes serviços podem ser invocados, publicados e descobertos, e são abstraídos da implementação usando uma única forma de interface baseada em padrões.

Além de criar e expor serviços, Rouse (2014a) assinala que a SOA tem a capacidade de aproveitar tais serviços de forma recursiva em aplicações (conhecidos como aplicativos compostos). Para isto, ela pode atuar como uma arquitetura orquestradora de serviços, ou expor cada um deles individualmente. Um aspecto importante da SOA é a separação da interface de serviço (o que) da sua implementação (como). Esses serviços são consumidos por clientes que não estão preocupados com a forma como esses eles irão executar suas solicitações

A SOA permite a reutilização de ativos existentes, em que novos serviços podem ser criados a partir de uma infraestrutura de sistemas de TI existente. Em outras palavras, ela permite que as empresas reutilizem aplicativos existentes e possibilita a interoperabilidade entre aplicações e tecnologias heterogêneas. Ela fornece um nível de flexibilidade que não era possível antes, no sentido de que:

a) Os serviços são componentes de *software* com interfaces bem definidas e independentes da implementação.

b) Os serviços são autônomos (executam tarefas predeterminadas) e vagamente acoplados (para independência);

c) Serviços compostos podem ser construídos a partir de agregados de outros serviços;

Isso significa que a SOA é uma abordagem alinhada com o negócio, em que os aplicativos dependem dos serviços disponíveis para facilitar os processos de negócios. Um serviço é um componente de *software* reutilizável e autônomo, fornecido por um provedor de serviços e consumido pelos solicitantes. A SOA cria uma visão de flexibilidade de TI que facilita a agilidade do negócio. Sua implementação envolve principalmente componentes de aplicativos corporativos e/ou em desenvolvimento que usam serviços, disponibilizando aplicativos como serviços para outras aplicações (ZHANG; ZHANG; CAI, 2007).

Rouse (2014a) acrescenta que em um ambiente SOA típico, existe um provedor de serviços e um consumidor de serviços. Para que isso funcione, também é necessário um mecanismo para que eles possam se comunicar uns com os outros, como é ilustrado na figura 5. A W3C [2] definiu um padrão aberto para que *web services* possam implementar a SOA e habilitar a comunicação entre o provedor e o consumidor através de um protocolo baseado em XML: o *Simple Object Access Protocol* (SOAP). Outros padrões também são usados ou foram criados para realizar a comunicações entre os serviçoes SOA. Alguns

[2] World Wide Web Consortium

exemplos são o *REpresentation State Transfer* (REST), que utiliza tanto XML quando JSON para transportar os dados entre os serviços, ou mais recentemente o GraphQL, utilizando um modelo declaraivo de comunicação entre o cliente e o servidor, também baseado em JSON.

Figura 5 – Service Oriented Architecture

Em geral, há uma confusão entre o relacionamento entre SOA e *web services*. Natis (2003) faz uma distinção entre estes dois conceitos do seguinte modo: "Web services tratam-se de especificações de tecnologias, enquanto o SOA é um princípio de *design* de *software*. Notavelmente, Web services são um padrão de definição de interface adequado para a SOA e é neste ponto onde eles se conectam". Portanto, SOA é um padrão de arquitetura, enquanto os *Web services* são serviços implementados usando um conjunto de padrões. *Web service* é uma das maneiras em que é possível implementar a SOA, e por isso compartilha dos mesmo benefícios já citados relacionados a SOA.

As arquiteturas para construção de aplicações evoluiram de modelos centralizadas em direção a modelos mais distribuídos, com objetivo de dividir melhor a responsabilidade de cada um de seus componentes.

3 Evolução da comunicação entre aplicações

Neste capítulo serão apresentados alguns dos principais protocolos para a comunicação entre aplicações. Serão conduzidas breves abordagens dos protocolos RPC e SOAP, que hoje vêm perdendo espaço no mercado, mas tiveram um papel muito importante para a evolução dos sistemas baseados em SOA. Em seguida, será mostrada uma descrição mais detalhada dos modelos de comunicação REST e GraphQL, uma vez que estes são os dois modelos a serem seguidos no estudo de caso.

Serão apresentados as principais características desses protocolos, como também as suas limitações. Nenhum dos protocolos pode ser descardado na hora de tomada de decisão sobre qual modelo utilizar para a comunicação entre clientes e serviços remotos ou APIs. Todos eles possuem limitações, porém cumprem bem o papel para que foram desenhados e é possível notar como eles evoluíram à medida que novas demandas se tornaram necessárias no desenvolvimento de *software*.

3.1 Remote Procedure Call

Remote Procedure Call (RPC), ou Chamada de Procedimento Remoto, é um mecanismo em que uma aplicação solicita o serviço de uma outra aplicação que se encontra em um outro computador, geralmente conectados por uma rede. Uma RPC exige que uma aplicação X envie uma ou mais mensagens para outra aplicação Y a fim de invocar um procedimento da aplicação Y. A aplicação Y responde enviando uma ou mais mensagens de volta para a aplicação X. Este fluxo de informação pode ser observado na figura 6. O termo RPC identifica todo o processo de comunicação entre as aplicações (MERRICK; ALLEN; LAPP, 2006).

Conforme TANENBAUM e Steen (2007) assinala, a ideia fundamental da RPC é fazer com que a chamada de procedimeto remoto pareça o mais possível uma chamada local. Em outras palavras, é necessário que RPC seja transparente, fazendo com que o procedimento cliente não deva estar ciente que o procedimento chamado está executando em uma máquina diferente ou vice e versa.

Ainda, segundo (TANENBAUM; STEEN, 2007), há alguns desafios importantes a serem considerados ao utilizar as chamadas remotas em comparação as chamadas locais. Alguns destes desafios são:

a) o procedimento chamado e o procedimento cliente rodam em máquinas diferentes e sào executados em espaços de endereço diferentes, o que pode causar complicação ao

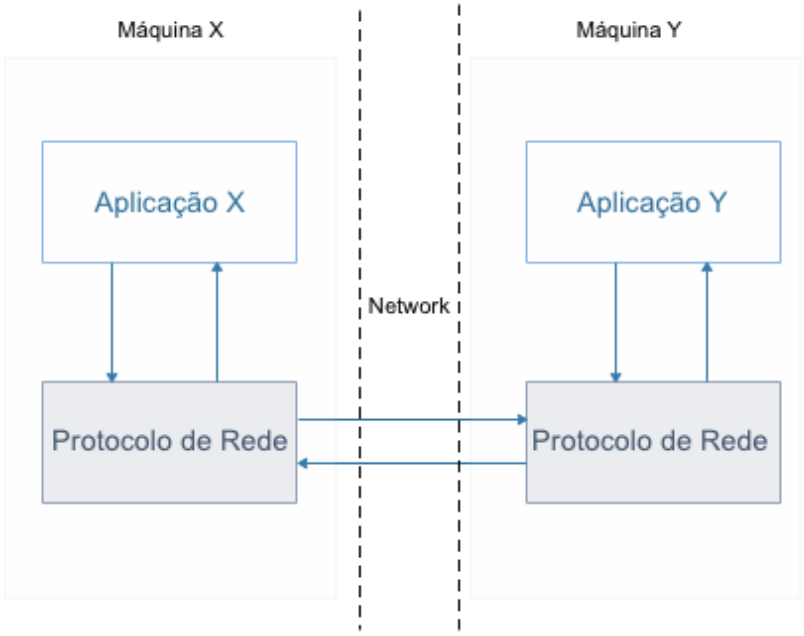

Figura 6 – Fluxo de informações RPC

implementar a RPC;

b) é necessários a passagem de parâmetros e resultados através da rede, o que pode ser complicado, em especial se as máquinas não forem idênticas;

c) qualuer uma das máquinas podem falhar e cada uma das possíveis falhas causa problemas diferentes;

Ainda assim, é possivel lidar com muitos destes desafios, e a RPC é uma técnica de ampla utilização subjacente a muitos sistemas distribuídos.

3.1.1 Limitações

Diferentes implementações de RPC são, geralmente, incompatíveis. Assim sendo, o uso de uma implementação específica, provavelmente resultará na dependência da implementação do fornecedor. Esse tipo de incompatibilidade entre diferentes implementações implica em um diverso número de funcionalidades, suportando inúmeros protocolos de rede e diferentes sistemas de arquivos.

Kukreja (2014) aponta diversas limitações técnicas e de desempenho encontradas em implementações de RPC's. Dentre estas limitações, se destacam:

a) **Organização dos parâmetros**: para organizar os parâmetros, o cliente precisa conhecer a quantidade e a tipagem exata dos parâmetros. Como não existe um

padrão bem definido, a estruturação dos parâmetros podem variar entre diferentes implementações.

b) **Falta de paralelismo**: com RPC, em um determinado momento, ou o servidor ou o cliente está ativo. Como cliente e servidor são co-rotinas, a prática de paralelismo presente em outros modelos de comunicação não é possível de ser implementada.

c) **Ferramentas de padronização**: as ferramentas padronizadas tornam-se muito mais difíceis de construir, devido ao grande número de implementações diferentes. Construir estas ferramentas geralmente requer o redesenho de convenções de tipo HTTP baseado sistema RPC escolhido, por consequencia da especialidade de cada sistema RPC.

3.2 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) é um protocolo para troca de mensagens em ambientes descentralizados e distribuídos. Box et al. (2000) definem o SOAP como um protocolo baseado em XML que consiste em três partes: um envelope que defina um *framework* para descrever o conteúdo das mensagens trafegadas e como processar este conteúdo, um conjunto de regras de codificação para expressar instâncias de tipos de dados definidos por uma aplicação, e uma convenção para representar chamadas remotas e suas respostas.

Rouse (2014b) argumenta que o SOAP pode ser considerado parecido com as Chamadas de Procedimento Remoto (RPC), porém eliminando algumas das complexidades frequentemente encontradas nas implementações de RPCs. Utilizando o SOAP, é possível chamar serviços de outras aplicações que estão sendo executadas em qualquer *hardware*, independente do sistemas operacional ou da linguagem de programação.

Embora a especificação do SOAP tenha evoluído para longe da necessidade de acessar objetos, como ocorre com as Chamadas de Procedimentos Remoto, existem ainda convenções para o encapsulamento e o envio de chamadas RPC utilizando uma representação uniforme de chamadas e repostas. Definindo um padrão para o mapeamento das chamadas RPC para SOAP, torna-se possível para infraestrutura traduzir as invocações de serviços em mensagens com o protocolo SOAP em tempo de execução, sem a necessidade de redesenhar todo o serviço Web presente na plataforma (SKONNARD, 2003).

A figura 7 ilustra como é realizada a troca de mensagens entre uma aplicação cliente e o servidor utilizando o SOAP. Como é possível observar, a aplicação cliente primeiramente serializa a mensagem e envia. Ao receber a requisição, o servidor precisa deserializar a mensagem, executar a consulta e serializar a mensagem de resposta. A aplicação cliente então, deserializa a resposta para poder manipulá-la.

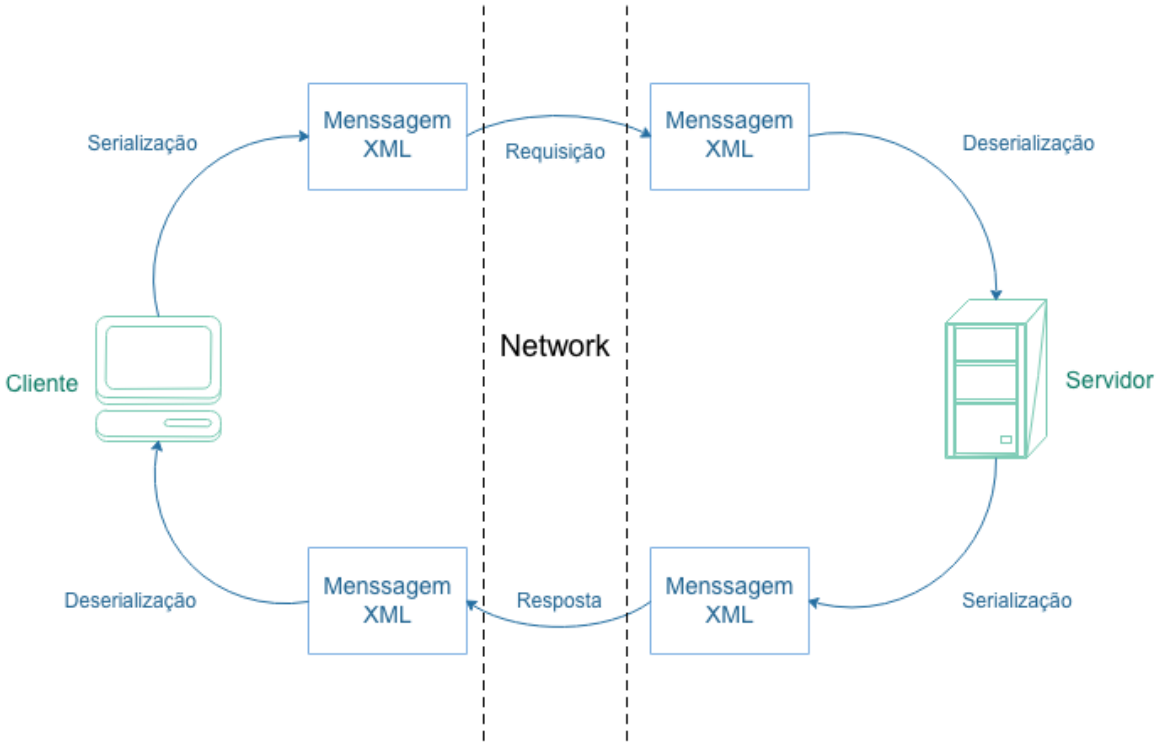

Figura 7 – SOAP fluxo de dados.

Leopoldo (2007) assinala que SOAP é um protocolo que define uma gramática XML especializada, que padroniza o formato das estruturas das mensagens. As mensagens são, por outro lado, o método fundamental de troca de informações entre os serviços Web e os seus consumidores. Ao utilizar XML para codificar mensagens, o SOAP apresenta alguns benefícios:

a) permite a comunicação entre sistemas protegidos por *firewalls*, sem precisar abrir portas adicionais e, possivelmente, não seguras;

b) é mais fácil de entender e eliminar erros pois o formato XML pode ser mais facilmente lido e entendido;

c) as mensagens podem ser compreendidas por quase todas as plataformas de hardware, sistemas operacionais e linguagens de programação, uma vez que os dados do SOAP são estruturados usando XML;

d) pode ser usado, potencialmente, em combinação com vários protocolos de transporte de dados, como HTTP, SMTP e FTP;

e) O SOAP mapeia satisfatoriamente para o padrão de solicitação / resposta do HTTP;

3.2.1 Limitações

Kohlhoff (2003) explica que tecnologias de metadados abertas, como o XML, podem fornecer um grande ganho de usabilidade, mas o sucesso dessas tecnologias exige que seu uso não degrade o desempenho de forma irracional. O XML é extremamente robusto, no entanto, o seu uso pode afetar negativamente o desempenho do SOAP nas seguintes áreas:

a) **Velocidade de codificação e decodificação**: a conversão de dados de binário para ASCII e vice-versa é o principal custo de desempenho do XML. O uso de um protocolo baseado em texto também impede a aplicação de otimizações disponíveis para protocolos binários quando a comunicação ocorre entre sistemas homogêneos;

b) **Tamanho da mensagem**: para XML, um fator de expansão de 6-8 vezes em relação aos dados binários originais não é incomum. O tamanho da representação de dados do SOAP é tipicamente cerca de 10 vezes o tamanho da representação binária equivalente. Estes fatores resultam em maiores custos de transmissão de rede e latência aumentada;

Como é possível notar, as maiores limitações encontradas na utilização do SOAP como modelo de comunicação estão relacionadas a estruturação usando XML. Embora esse formato padrão de mensagens tenha trazido vantagens em relação às tecnologias anteriores, ele também trouxe uma série de limitações conforme sua popularidade foi crescendo. A necessidade de buscar alternativas para superar esses problemas foi crescente, e novos formatos foram aparecendo, tendo como destaque o JSON.

3.3 REpresentational State Transfer

REpresentational State Transfer (REST) é um *design* de arquitetura baseado em um conjunto de princípios que descrevem como os recursos em rede são definidos e abordados. Estes princípios foram descritos pela primeira vez em 2000 por Roy Fielding, como parte de sua dissertação de doutorado (BARRY, 2003).

Lensmar (2013) assinala que a adoção do REST como o método predominante para construir APIs públicas tem ofuscado qualquer outra tecnologia ou abordagem nos últimos anos. Embora várias alternativas (principalmente SOAP) ainda estejam presentes no mercado, adeptos do modelo SOA, descrito da seção 2.4, para construção de aplicações tomaram uma posição definitiva contra eles e optaram por REST como sua abordagem e JSON como seu formato de mensagem .

Segundo Battle (2008), REST é um padrão de operações de recursos que emergiu como a principal alternativa ao SOAP para o *design* de serviços em aplicativos Web 2.0. Considerando que a abordagem tradicional baseada em SOAP para Web Services usa

objetos remotos completos com invocação de método remoto e funcionalidade encapsulada, o REST trata apenas de estruturas de dados e da transferência de seu estado. A simplicidade do REST, juntamente com seu ajuste natural sobre o HTTP, contribuiu para o seu *status* como um método de escolha para aplicativos da Web 2.0 expondo seus dados.

3.3.1 Restrições

As restrições do REST são regras desenhadas para estabelecer as características distintas da arquitetura REST. Cada restrição é uma decisão de projeto pré-determinada que pode ter impactos positivos e negativos. A intenção é que os aspectos positivos de cada restrição equilibrem os negativos para produzir uma arquitetura geral que se assemelhe à Web.

Uma arquitetura que elimina uma das restrições de **cliente/servidor**, **stateless**, **cache**, **interface uniforme**, **sistemas em camadas** ou **código por demanda**, geralmente é considerada como não mais conforme ao REST. Isso exige que as decisões sejam tomadas para entender os potenciais *trade-offs* quando se desviam deliberadamente da aplicação de restrições REST.

A primeira restrição estabelecida é a da arquitetura **cliente-servidor**, descrita na Seção 2.2. A separação de responsabilidades é o princípio por trás das restrições cliente-servidor. Ao separar as responsabilidades da interface do usuário com as responsabilidades de armazenamento de dados, é melhorada a portabilidade da interface do usuário em várias plataformas e a escalabilidade, simplificando os componentes do servidor. Talvez, o mais importante para a Web, no entanto, é que a separação permite que os componentes evoluam de forma independente, apoiando assim o requisito de escalabilidade da Internet de múltiplos domínios organizacionais (FIELDING, 2000)

Em seguida, foi adicionada ao REST uma restrição à interação cliente-servidor: a comunicação deve ser totalmente **stateless**, ou seja, independente de estado, de modo que cada solicitação do cliente deve conter todas as informações necessárias para entender o pedido e não pode tirar proveito de nenhum contexto armazenado no servidor. O estado da sessão é, portanto, mantido inteiramente no cliente (FIELDING, 2000).

Essa restrição induz as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é melhorada porque um sistema de monitoramento não precisa olhar além de um único dado de solicitação para determinar a natureza completa da solicitação. A confiabilidade é melhorada porque facilita a tarefa de recuperação de falhas parciais. A escalabilidade é melhorada porque não precisar armazenar o estado das solicitações permite que o servidor rapidamente libere recursos, simplificando a implementação.

A restrição de **cache** foi criada para melhorar a eficiência da rede. Ela exige que os dados dentro de uma resposta a uma solicitação sejam rotulados de forma implícita ou

explícita como cacheáveis ou não armazenáveis em cache. Se uma resposta for armazenada em cache, um cache do cliente terá o direito de reutilizar esses dados de resposta para solicitações equivalentes posteriores.

A vantagem de adicionar restrições de cache é que eles têm o potencial de eliminar parcial ou totalmente algumas interações, melhorar a eficiência, escalabilidade e desempenho percebido pelo usuário, reduzindo a latência média de uma série de interações. O *trade-off*, no entanto, é que um cache pode diminuir a confiabilidade se os dados obsoletos dentro do cache diferirem significativamente dos dados que teriam sido obtidos se a solicitação fosse enviada diretamente para o servidor.

A característica central que distingue a arquitetura REST de outros estilos baseados em rede é a ênfase em uma **interface uniforme** entre os componentes. Ao aplicar o princípio de engenharia de software de especialização/generalização para a interface do componente, a arquitetura geral do sistema é simplificada e a visibilidade das interações é melhorada. As implementações são dissociadas dos serviços que eles fornecem, o que incentiva a evolução independente. O *trade-off*, no entanto, é que uma interface uniforme degrada a eficiência, uma vez que a informação é transferida em uma forma padronizada e não específica para as necessidades de uma aplicação.

Para obter esta interface uniforme, são necessárias várias restrições arquitetônicas para orientar o comportamento dos componentes. REST é definido por quatro restrições de interface: identificação de recursos; manipulação de recursos através de representações; mensagens auto-descritivas; e, hipermídia como motor do estado da aplicação.

A fim de melhorar o comportamento dos requisitos de escalonamento, o REST também implementa restrições de **sistema em camadas**. O sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas ao restringir o comportamento dos componentes, de modo que cada componente não possa acessar além da camada imediata com a qual eles estão interagindo. Ao restringir o conhecimento do sistema a uma única camada, colocamos um limite na complexidade geral do sistema e promovemos a independência do componentes. As camadas podem ser usadas para encapsular serviços legados e para proteger novos serviços de clientes legados, simplificando componentes e movendo funcionalidades raramente usadas para um intermediário compartilhado. Os intermediários também podem ser usados para melhorar a escalabilidade do sistema ao permitir o balanceamento de carga de serviços em várias redes e processadores.

A principal desvantagem dos sistemas em camadas é que eles adicionam sobrecarga e latência ao processamento de dados, reduzindo o desempenho perceptível pelo usuário. Para um sistema baseado em rede que suporta restrições de cache, isso pode ser compensado pelos benefícios do armazenamento em cache compartilhado em intermediários.

A última restrição que o REST implementa é a restrição de **código por demanda**.

O REST permite que a funcionalidade do cliente seja estendida baixando e executando o código na forma de *applets* ou *scripts*. Isso simplifica os clientes, reduzindo o número de recursos necessários para serem pré-implementados. Permitir que os recursos sejam baixados após a implantação melhora a extensibilidade do sistema, no entanto, reduz a visibilidade e, portanto, é apenas uma restrição opcional dentro do REST.

3.3.2 Modelo de Maturidade de Richardson

O Modelo de Maturidade de Richardson é um modelo criado por Leonard Richardson que descreve os requisitos necessários para o desenvolvimento de uma API REST bem estruturada e compatível com as restrições definidas pela arquitetura REST. Quanto melhor a API adere às restrições citadas na sessão anterior, melhor será pontuada. O modelo de Richardson, ilustrado na figura 8, descreve 4 níveis (0-3), onde o nível 3 designa uma API verdadeiramente *RESTful*. Richardson usou três fatores para decidir a maturidade de uma API: URI (*Uniform Resource Identifiers*), métodos HTTP e HATEOAS (Hypermedia). Quanto mais uma API emprega essas tecnologias, mais madura ela é categorizada.

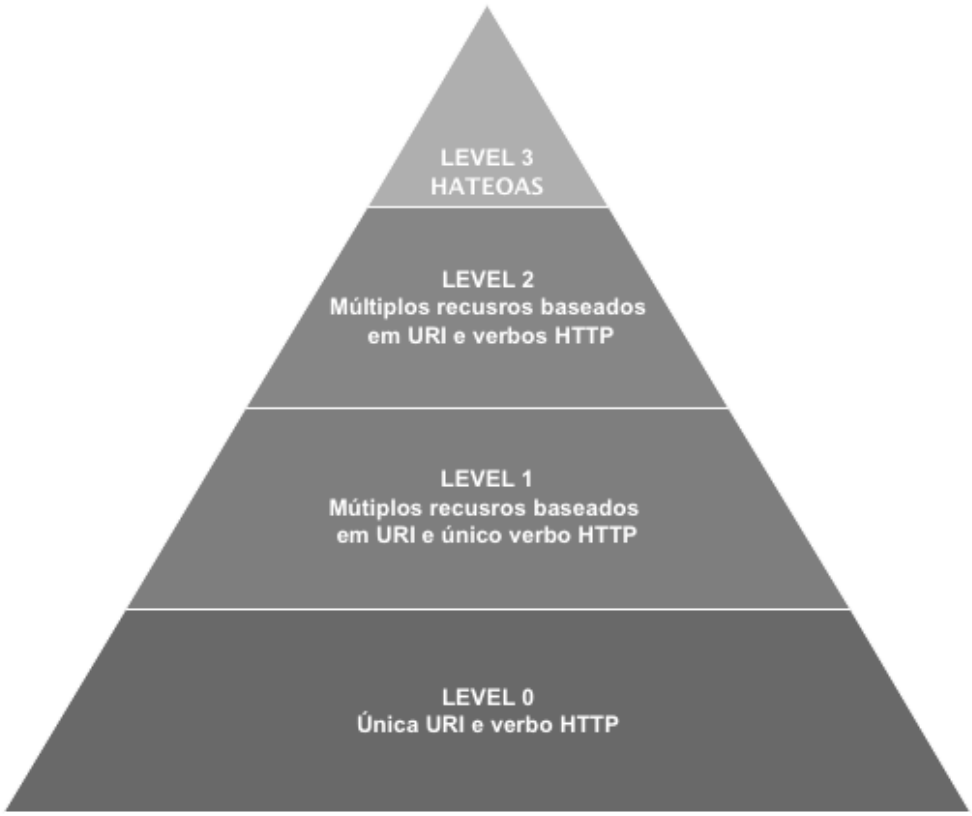

Figura 8 – Modelo de maturidade de Richardson.

Os níveis do modelo de maturidade de Richardson são:

a) Nível 0: O ponto de partida para o modelo é usar HTTP como um sistema de transporte para iterações remotas, mas sem usar nenhum dos mecanismos da web. Essencialmente, o que está sendo feito é utilizar o HTTP como um mecanismo de tunelamento para seu próprio mecanismo de interação remota, geralmente com base em Chamadas de Procedimento Remotas.

   É como se estivesse sendo chamado funções, porém através do uso do HTTP. Todos os serviços são centralizados em um único *endpoint*, ou seja, todas as solicitações são feitas em uma única URI.

b) Nível 1: Quando uma API faz a distinção entre recursos diferentes, pode se considerar que atingiu o nível 1. Esse nível usa vários URIs, em que cada URI é o ponto de entrada para um recurso específico. Ainda assim, esse nível usa apenas um único método, como o POST.

c) Nível 2: Esse nível indica que a API deve usar as propriedades do protocolo para lidar com escalabilidade e falhas. Não é recomendado que se use um único método POST para todas as chamadas, mas faça uso do GET quando estiver solicitando recursos e use o método DELETE quando desejar excluir recursos. Além disso, o uso dos códigos de resposta do protocolo de aplicação também é recomendado. Não deve ser usado, por exemplo, o código 200 (OK) quando algo der errado.

d) Nível 3: O nível três de maturidade faz uso de todos os três fatores, isto é, URIs, métodos HTTP e HATEOAS. Esse é o nível em que a maioria das APIs menos implementam e pois não seguem o princípio de HATEOAS.

HATEOAS (Hypermedia as the Engine of Application State) é uma abordagem para a construção de serviços REST, em que o cliente pode descobrir dinamicamente as ações disponíveis em tempo de execução. Todo cliente deve exigir uma URI inicial e um conjunto de tipos de mídia padronizados para começar a troca de mensagens. Uma vez que carregou o URI inicial, todas as futuras transições de estado da aplicação serão conduzidas pelo cliente selecionando as escolhas fornecidas pelo servidor.

Não seguir essa abordagem não é necessariamente ruim. Há alguns bons argumentos a serem feitos a favor e contra a utilização de HATEOAS. Enquanto, por um lado, torna as APIs fáceis de descobrir e usar, por outro, eleva o tempo e o esforço de desenvolvimento das aplicações.

3.3.3 Limitações

Contudo, com a evolução e o aumento da complexidade das APIs, a comunicação via o protocolo REST vem se mostrando muitas vezes inviável pois sua implementação trás como consequências alguns problemas estruturais como:

a) a necessidade de executar múltiplas requisições entre o cliente e o servidor a fim de obter objetos complexos e com atributos aninhados Para aplicações móveis operando em condições variáveis de rede, essas múltiplas viagens de ida e volta são indesejáveis, pois geram atrasos e maior tráfego na rede (SCHROCK, 2015);

b) a prática de *over-fetching*, ou seja, quando o cliente busca alguma informação do servidor e a resposta contém mais informação que o cliente precisa. Esse tipo de problema acarreta no uso desnecessário de recursos de comunicação (GUSTAVSSON, 2016);

c) o versionamento da API, que ocorre quando há alterações significativas na API, sujeitas a quebra de código nos clientes consumidores. Essa prática pode ser extremamente penosa se a API é usada por uma grande massa de clientes que não são facilmente atualizados (KROHN, 2013);

3.4 Arquiteturas baseadas em JSON/Graphs

Recentemente, um novo design de arquitetura vem ganhando espaço, preenchendo algumas lacunas que arquiteturas anteriores deixaram. Lideradas pelo Facebook com seu GraphQL e Netflix com o Falcor, esta nova arquitetura da um passo para trás comparando-se ao REST, atingindo apenas o nivel 0 no Modelo de Maturidade de Richardson.

Os principais problemas que essa arquitetura ajuda a resolver são:

a) a dependência das aplicações cliente nos servidores, eliminando a necessidade do servidor de ter que manipular as informações ou o tamanho da resposta;

b) a necessidade de executar diversas requisições para acessar os dados exigidos por uma *view*;

c) melhorar a experiência de desenvolvimento *frontend*, pois existe uma relação próxima entre os dados necessários à interface da aplicação e a forma como um desenvolvedor pode expressar uma descrição desses dados para a API.

O presente trabalho irá apenas focar no GraphQL como representante das arquiteturas baseadas em JSON/Graph.

GraphQL é uma linguagem de consulta, criada pelo Facebook em 2012, que fornece uma interface comum entre o cliente e o servidor para manipulação e busca de dados. GraphQL utiliza de um sistemas chamado *cliente-specified queries*, onde o formato de resposta de uma requisição é definida pelo cliente. Schrock (2015) afirma que uma vez que a estrutura de dados não é codificada, como nas APIs tradicionais, a consulta de dados do servidor se torna mais eficiente para o cliente.

Consultas utilizando GraphQL sempre retornam apenas o que foi pré definido pela requisição, fazendo suas respostas sempre serem previsíveis. As consultas com GraphQL acessam não apenas as propriedades de um único recurso, mas também seguem as referências entre eles. Enquanto as APIs REST típicas exigem o carregamento de múltiplas URLs, as APIs GraphQL obtêm todos os dados que precisam em uma única requisição.

Adicionar novos campos ou tipos à uma API GraphQL não afeta nenhuma consulta ou funcionalidade já existente. Campos não mais utilizados podem ser obsoletos e ocultos de ferramentas de mapeamento. Ao usar uma única versão em evolução, as APIs GraphQL dão às aplicações acesso contínuo a novos recursos e encorajam um código de servidor mais limpo e mais sustentável.

3.4.1 Operações

Existem três tipos de operações modeláveis com GraphQL:

a) **Query**: Uma consulta de somente leitura;

b) **Mutation**: Uma escrita seguida por uma consulta;

c) **Subscription**: Uma requisição de longa duração que obtém dados em resposta a eventos disparados pelo servidor;

Uma Query em GraphQL é uma maneira de obter dados de uma maneira somente de leitura em uma API GraphQL. De uma maneira geral, GraphQL se baseia em consultar campos específicos em objetos. Isso significa que a consulta tem exatamente o mesmo formato que a resposta. Isso é essencial para GraphQL porque a resposta é sempre previsível, e o servidor sabe exatamente quais os campos que o cliente está pedindo.

Como GraphQL não se limita apenas em consultas de dados, as APIs também podem implementar operações para criar, atualizar e destruir dados. Para esses tipos de operações, o GraphQL usa o termo Mutations. As Mutations são uma maneira de alterar dados em seu servidor. É importante notar que as *mutations* consistem em uma alteração seguida de uma busca do dado que acabou de ser alterado, tudo em uma única operação.

Os aplicativos em tempo real precisam de uma maneira de enviar dados do servidor. As *subscriptions* permitem que aplicações publiquem eventos em tempo real através de um servidor de assinaturas GraphQL. Com o modelo de subscriptions baseado em eventos no GraphQL - muito parecido com as *queries* e *mutations* - um cliente pode dizer ao servidor exatamente quais dados devem ser enviados e como esses dados devem ser encontrados. Isso leva a menos eventos rastreados no servidor e notificados para o cliente.

3.4.2 Schemas e Tipos

O sistema de tipos do GraphQL descreve as capacidades de um servidor GraphQL e é usado para determinar se uma consulta é válida. O sistema de tipos também descreve os formatos de entrada de variáveis de consulta para determinar se os valores fornecidos em tempo de execução são válidos. As capacidades do servidor GraphQL são referidas como *schema* do servidor. Um *schema* é definido baseado nos tipos que ele suporta.

Cada servidor GraphQL define um conjunto de tipos que descrevem completamente o conjunto de dados possíveis que você pode consultar nesse servidor. Então, quando as consultas chegam, elas são validadas e executadas contra os *schemas*. O *schema* descreverá quais campos o servidor pode responder e quais tipos de objetos estarão contidos nas respostas. A informação de tipo é muito importante para o GraphQL e o cliente pode assumir de forma segura que o servidor retornará tipos consistentes de objetos para o mesmo campo.

A unidade fundamental de qualquer *schema* GraphQL é o tipo. Existem oito formatos de tipos suportados no GraphQL. O tipo mais básico é o ***Scalar***. Um campo do tipo *Scalar* representa um valor primitivo, como uma *string* ou um número inteiro. Muitas vezes, as respostas possíveis para um campo do tipo *Scalar* são enumeráveis. O GraphQL oferece um tipo ***Enum*** nesses casos, onde o tipo especifica o espaço de respostas válidas. Campos do tipo ***Object*** definem um conjunto de campos, onde cada campo é outro formato, permitindo a definição de hierarquias de tipos arbitrários. O GraphQL suporta dois tipos abstratos: ***Interfaces*** e ***Unions***.

Todos os tipos até agora são considerados nulos e singulares, por exemplo, uma *string* retorna um valor nulo ou singular. O sistema de tipo pode querer definir que ele retorna uma lista de outros tipos. O tipo ***List*** é fornecido por esse motivo e envolve outro tipo. Da mesma forma, o tipo ***Non-Null*** envolve outro tipo e indica que o resultado nunca será nulo.

Finalmente, muitas vezes é útil fornecer estruturas complexas como entradas para consultas GraphQL; o tipo ***Input Object*** permite que o *schema* defina exatamente quais dados são esperados do cliente nessas consultas.

3.4.3 Caching

De acordo com Grigorik (2017), consultas de dados através da internet podem ser lentas e custosas, e por esse motivo, a capacidade de armazenar em cache e reutilizar os recursos obtidos anteriormente é um aspecto crítico da otimização para o desempenho.

Em uma API baseada em *end-points*, os clientes podem usar o armazenamento em cache do protocolo HTTP, ilustrado na figura 9 para evitar buscas desnecessárias identificando quando dois recursos são iguais. A URL nessas APIs é o identificador

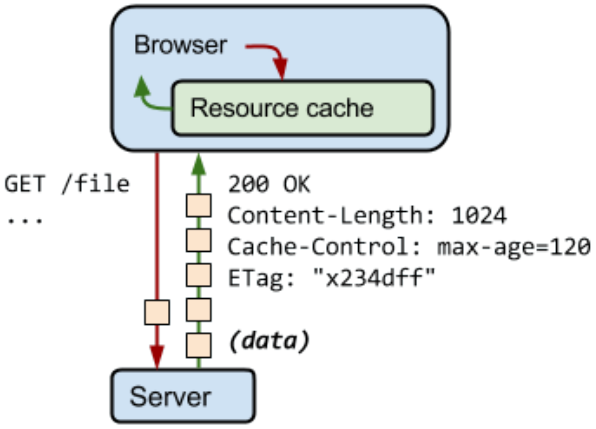

Figura 9 – HTTP Caching.

Fonte: Grigorik (2017)

globalmente exclusivo que o cliente pode aproveitar para criar um cache. No GraphQL, porém, não existe uma primitiva semelhante a uma URL que forneça esse identificador globalmente exclusivo para um determinado objeto. Entretanto, existem várias práticas que podem ser aplicadas a um servidor GraphQL para atingir o mesmo objetivo.

Um padrão possível para isso é reservar um campo, como id, para ser o identificador globalmente exclusivo. Da mesma forma que os URLs de uma API baseada em recursos fornecem uma chave globalmente exclusiva, o campo id neste sistema fornece uma chave globalmente única. Outra abordagem possível funcionaria de forma semelhante ao padrão usada nas APIs baseadas em *end-points*. O texto da consulta em si pode ser usado como identificador globalmente exclusivo.

3.4.4 Limitações

Segundo aponta Buna (2017), uma ameaça importante que o GraphQL *facilita* são os ataques de negação de serviço. Um servidor GraphQL pode ser atacado com consultas excessivamente complexas que irão consumir todos os recursos do servidor. Esse tipo de ataque não é específico do GraphQL, mas é preciso um cuidado redobrado para evitá-los.

Há, entretanto, alguns procedimento que, se implementados, podem mitigar a ameaça de negação de serviço. É possível fazer uma análise de custos na consulta com antecedência e impor *um certo tipo* de limites na quantidade de dados que uma requisição pode consumir. Também é possível implementar um tempo limite para que requisições que levam muito tempo para serem resolvidas, sejam excluídas da fila de execução .

4 Estudo de Caso

Para obter um exemplo do mundo real, um estudo de caso foi projetado com base em uma aplicação de WMS (*Warehouse Management System*). Rouse (2009) explica que um sistema de WMS é um software que auxilia as operações do dia-a-dia em um armazém. Os sistemas WMS permitem o gerenciamento centralizado de tarefas, como o rastreamento de níveis de inventário e locais de estoque. Tais sistemas WMS podem ser um aplicativo independente ou parte de um sistema de ERP.

Com o intuito de realizar consultas relevantes, foi modelado um esquema contendo seis entidades capazes de representar consultas reais, e ainda produzirem informações pertinentes para comparar o desempenho dos protótipos. A construção das APIs seguiram os princípios do ciclo tradicional do desenvolvimento de *software*, portanto a modelagem foi elaborada antes do início da implementação das APIs. Isso assegura que nenhuma decisão de tecnologia tenha afetado a modelagem das entidades.

A modelagem da figura 10, representa o gerenciamento de um armazém com capacidade de armazenar diversos itens. Os itens são dispostos em *pallets*, e alocados em endereços dentro do armazém. Como pode ser observado na figura 11, os endereços do armazém são formados a partir de uma combinação de três dimensões: **Prateleira**, **Linha** e **Nível**. Cada combinação dessas três propriedades é capaz de alocar *pallets*, que por sua vez podem conter diversas unidades de um mesmo item.

Imagine um item de código 22B12, por exemplo, que representa um dado produto X. Este produto é disposto em um *pallet* com capacidade de armazenar 30 unidades do item 22B12. O armazém é composto por 26 prateleiras, sequenciadas de 'A' a 'Z'. Cada prateleira possui 2 linhas de profundidade e 3 níveis de altura. Um *pallet* com o código 001 contém 30 unidade do item 22B12, e precisa ser alocado dentro armazém, para isso é utilizado um sistema de códigos envolvendo as 3 dimensões do armazém.

Desta forma, 60 unidades do item 22B12 estão dispostos em dois *pallets*(001, 002) existentes no armazém, e cada um dos *pallets* será destinado a um endereço. O *pallet* 001 será alocado na terceira prateleira, no segundo nível e na primeira linha. Após a formação do endereço, o *pallet* 001 estará localizado no endereço C0201, prateleira C, nível 02 e linha 01.

Baseado em um sistema de WMS, dois protótipos de APIs serão implementados para uma análise. A primeira API será implementada seguindo as melhores práticas do *design* do REST, descritas na sessão 3.3. A segunda será desenvolvida utilizando o GraphQL, e o seu desempenho será comparado com a API REST.

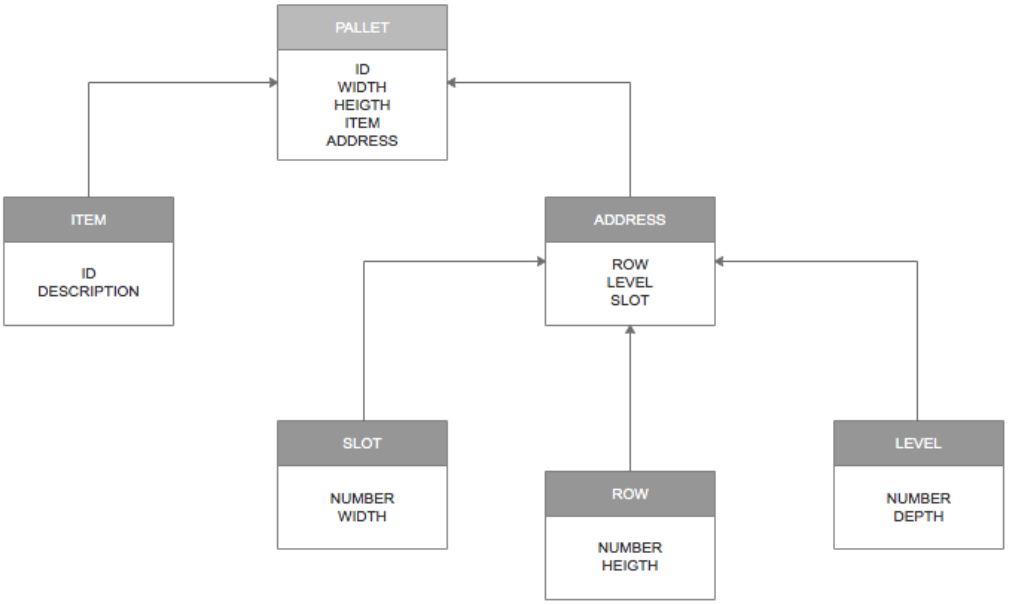

Figura 10 – Modelagem WMS

Fonte: Autor

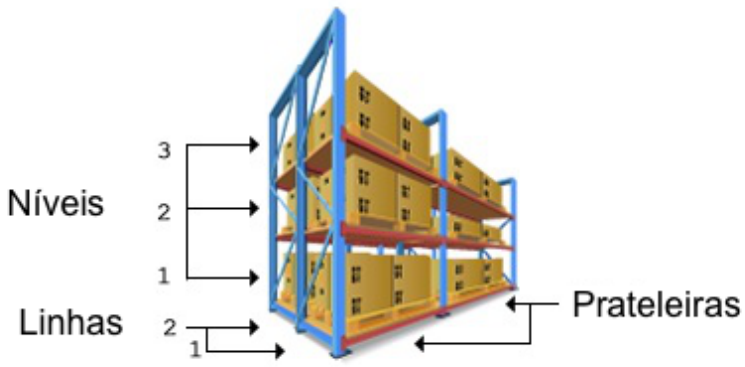

Figura 11 – Diagrama de dimensões do armazém

Fonte: NECS (2010)

4.1 Hipóteses

As hipóteses na diferença de desempenho entre as APIs derivaram dos fundamentos teóricos. Elas baseiam-se no entendimento de que os protocolos utilizados possibilitam a implementação de uma combinação de técnicas afim de afetar positivamento o desempenho da API. Portanto, ambas as implementações devem ter as mesmas propriedades, seguindo suas melhores práticas, modelos de maturidade e documentação.

As hipóteses deste trabalho são:

a) O tamanho da resposta será menor utilizando GraphQL;

b) O tempo de resposta será menor utilizando GraphQL;

c) O tempo de utilização da CPU será menor utilizando REST;

d) O consumo de memória será menor Utilizando REST;

Com o propósito de validar as hipóteses definidas, foram determinadas duas perguntas que envolvessem todas as entidades. Para cada pergunta existe apenas uma resposta correta e sua lógica é baseada em campos das estruturas de dados de retorno.

Questão 1: Qual item ocupa a maior quantidade de *pallets* alocados no armazém?

Questão 2: O item com o código 22B12 está armazenado em quais endereços?

Desta forma, buscas serão realizadas tanto na API REST quanto na API GraphQL afim de recuperar as informações necessárias para a formulação das respostas. Para realizar estas buscas, medir o desempenho das APIs processando suas respostas e compara-los, algumas ferramentas foram utilizadas durante este trabalho. Estas ferramentas estão descritas na sessão 4.2.

4.2 Ferramentas utilizadas

A escolha das ferramentas a serem usadas na implementação foi uma das partes mais importantes no planejamento do estudo de caso. Foi necessário pensar em uma especificação que apresentasse uma curva rápida de aprendizagem, um fluxo de execução replicável e independente de plataforma, e uma documentação madura em relação à implementação das APIs.

As APIs foram escritas na linguagem de programação JavaScript, utilizando a especificação do ECMAScript 5. Foi escolhida esta linguagem devido ao fácil acesso a boas ferramentas para construção de serviços Web, como o Node.js. Node.js é uma plataforma para desenvolvimento de servidores Web utilizando JavaScript e o V8 JavaScript Engine.

Assim, com Node.js pode-se criar uma variedade de aplicações Web utilizando código em JavaScript (LOPES, 2014).

4.2.1 Ferramentas servidor

Node.js dispõe de inúmeros recursos e ferramentas que possibilitam a construção de APIs. Mesmo assim, o ecosistema de bibliotecas no Node.js conta com ferramentas que simplificam ainda mais a construção de aplicações para servidores Web. Abaixo são citadas algumas bibliotecas que foram utilizadas para desenvolver tanto a API REST quando a API GraphQL.

Expess.js

Express.js é um *framework* para Node.js extremamente flexível, que fornece um conjunto robusto de ferramentas para a construção de aplicações Web e Móveis. Conta com um robusto sistema de roteamento, facilitando o desenvolvimento de APIs. O Express.js fornece uma fina camada de abstração nas principais funcionalidades do Node.js, sem sobrepor seus recursos.

Nos protótipos desenvolvidos para executar o experimento, o Express.js atua como um *middleware*, gerenciando as rotas e delegando a responsabilidade de interpretação das requisições para os *Controllers* na API REST e para os *Resolvers* na API GraphQL.

O código 1 ilustra como os *Controllers* enviam as informações da requisição para o modelo, que é o responsável na API REST em executar as consultas no banco de dados. No exemplo abaixo, o trecho de código é o responsável por retornar um *Item* baseado no id recebido como parâmetro da requisição. É possível observar que na linha 2 o modelo da entidade *Item* é importado para ser utilizado no *Controller*, e na linha 8, o método get, cujo a lógica está implementada dentro do modelo, é invocado passando como parâmetro o id.

```
1 //item.controller.js
2 import Item from '../models/item.model';
3
4 /**
5  * Load item and append to req.
6  */
7 function load(req, res, next, id) {
8   Item.get(id)
9     .then((item) => {
10       req.item = item;
11       return next();
12     })
13     .finally(e => next(e));
14 }
```

Código 1 – Controller para carregar um item

Já o código 2 mostra como o Express.js gerencia a requisição recebida via método GET, e delega a responsabilidade para o schema.js, onde está contida a lógica para a interpretação dos parâmetros da requisição, e retorna um objeto JSON com a devida resposta.

```
1 //server.js
2 import express from 'express';
3 let app = express();
4
5 import schema from './schema.js';
6
7 app.get('/', (req, res) => {
8     graphql(schema, req.query.query)
9     .then((result) => {
10         res.send(result);
11     });
12 });
```

Código 2 – Expess gerenciando rotas para GraphQL

MongoDB

MongoDB é um banco de dados *open source* que utilizada um modelo de dados orientado a objetos. Ele tem como característica conter todas as informações importantes em um único documento, possuir identificadores únicos universais (UUID), possibilitar a consulta de documentos através de métodos avançados de agrupamento e filtragem, também conhecido como *MapReducers*. Ao invés de usar tabelas e linhas, como os bancos de dados relacionais, o MongoDB usa uma arquitetura baseada em coleções e documentos.

Ferramentas clientes

Algumas ferramentas foram utlizadas para se comportaram como clientes das APIs construídas, e foram responsáveis por executar as consultas nas APIs. O foco deste trabalho não é em cima das aplicação clientes, entretanto é importante conhecê-las pois são elas que invocarão as consultas como também é através delas que algumas métricas serão extraídas. O software Postman será usado fundamentalmente para a execução das buscas nas APIs.

O Postman é uma cadeia completa de ferramentas para desenvolvedores de APIs. Ele é uma ferramenta elegante e flexível usada para construir softwares conectados via APIs, de forma rápida, fácil e precisa.

Ele funciona como uma emulador para execução de consultas em APIs. Com ele é possível executar buscas utilizando qualquer dos métodos do protocolo HTTP, possibilitando personalização tando do corpo quanto dos *headers* das requisições, se preciso. Junto com a resposta da consulta, o Postman traz também informações extremamente relevantes, como tempo que a API levou para responder e o tamanho em bytes contido na resposta. Estas funcionalidades, aliadas com a opção de executar um número personalizável de iterações para cada requisição, será a base para avaliar o desempenho das APIs REST e GraphQL.

4.2.2 Ambiente

A configuração da máquina utilizada para o teste de desempenho local é descrita na Tabela 1. As APIs REST e GraphQL foram construídas para responder às requisições recebidas, retornando respostas no formato JSON, a fim de gerar um fator de medição de desempenho da execução dos testes. Para a execução do experimento, apenas os softwares necessários estavam ativos. Portanto, ao executar as buscas, o servidor REST ou GraphQL estará ativo, além do servidor de banco de dados MongoDB, e a aplicação cliente Postman.

Item	**Computador 1**
Marca	Apple
Modelo	Macbook Air 13"
Processador	Intel Core i5 1.6 GHz
Memória	4 GB 1600 MHz DDR3
Disco rígido	128 GB SSD
Sistema	macOS Sierra 10.12.6

Tabela 1 – Especificação da máquina utilizada

4.2.3 Detalhes implementação REST

O servidor REST consiste em um servidor HTTP escrito em Node.js que recebe as requisições HTTP. Dependendo do método e URL da requisição, roteia-o para o *controller* correspondente. O *controller* faz a consulta na base de dados do MongoDB. Ao realizar o gerenciamento das rotas, o servidor realiza os devidos registros de latência. Após a consulta, a resposta desejada é enviada de volta para o cliente. Esse fluxo acontece em qualquer cenário, independente do número de requisições desejadas.

No presente trabalho, como o objetivo é apenas medir o desempenho de consultas utilizando REST, todas as requisições utilizarão o método HTTP GET. Para as medições da API REST, a aplicação cliente enviará, por exemplo, uma requisição para recuperar todos os itens cadastrados. Como pode ser observado da tabela 2, é necessário executar uma busca do tipo /item, que será interpretada pelo servidor REST, identificando qual é

a rota responsável por essa requisição. O servidor consultará a base de dados retornando uma resposta no formato JSON, com todos os itens cadastrados na API. Esse fluxo é ilustrado na figura 12

URI	**Descrição**
/item	Consulta lista de itens
/item/:id	Consulta item pelo id
/pallet	Consulta lista de pallets
/pallet/:id	Consulta pallet pelo id
/address	Consulta lista de endereços
/address/:id	Consulta endereço pelo id
/slot	Consulta lista de prateleiras
/slot/:id	Consulta prateleira pelo id
/row	Consulta lista de linhas
/row/:id	Consulta linha pelo id
/level	Consulta lista de nível
/level/:id	Consulta nível pelo id

Tabela 2 – Servidor REST

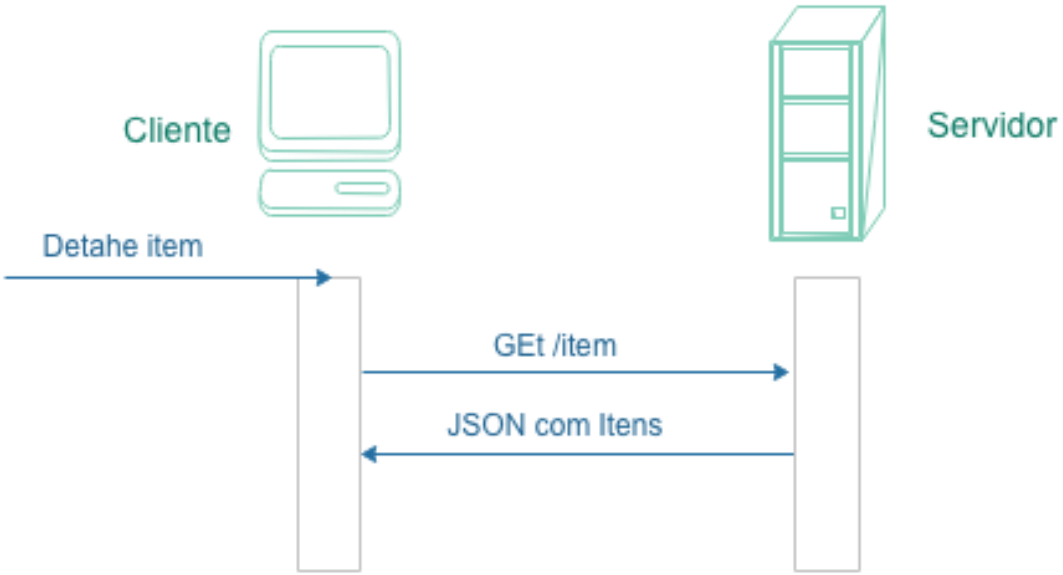

Figura 12 – Fluxo REST

4.2.4 Detalhes implementação GraphQL

O servidor GraphQL foi implementado também usando Node.js. A diferença em comparação com a implementação do servidor REST é que o servidor GraphQL envia todos os pedidos ao seu núcleo, ao invés de rotear as requisições recebidas para vários

controllers diferentes. O GraphQL analisa a consulta e envia os parâmetros para os *resolvers* responsáveis, localizados nos *schemas*. *Resolvers* são funções definidas para todos os campos no *schema*, cada um retorna os dados para o campo específico. Estas funções são executadas quando os campos correspondentes são consultados e os resultados são retornados na resposta.

```
query RootQuery {
  items {
      id
      description
    }
}
```

Código 3 – Consulta de itens

A consulta ilustrada do trecho de código 3 solicita ao servidor GraphQL a lista a de todos os itens cadastrados na base de dados. Note no trecho de código 4 que a resposta contém apenas os atributos que a consulta requisitou, ou seja, o id e a descrição dos itens:

```
{
    "data": [
        {
          "id": 22B12,
            "description": "Flat screen"
        },
        {
          "id": 21C44,
            "description": "Computer screen"
        },
        {
          "id": 43F12,
            "description": "Smartphone screen"
        },
    ]
}
```

Código 4 – Listagem dos itens (Validar)

Para recuperar alguma informação na API GraphQL, é necessária realizar uma requisição utilizando o método HTTP GET, passando como parâmetro a consulta desejada. A API irá então processar a consulta, e retornar um objeto JSON como a resposta da requisição. Detalhes deste fluxo esta ilustrado na figura 13

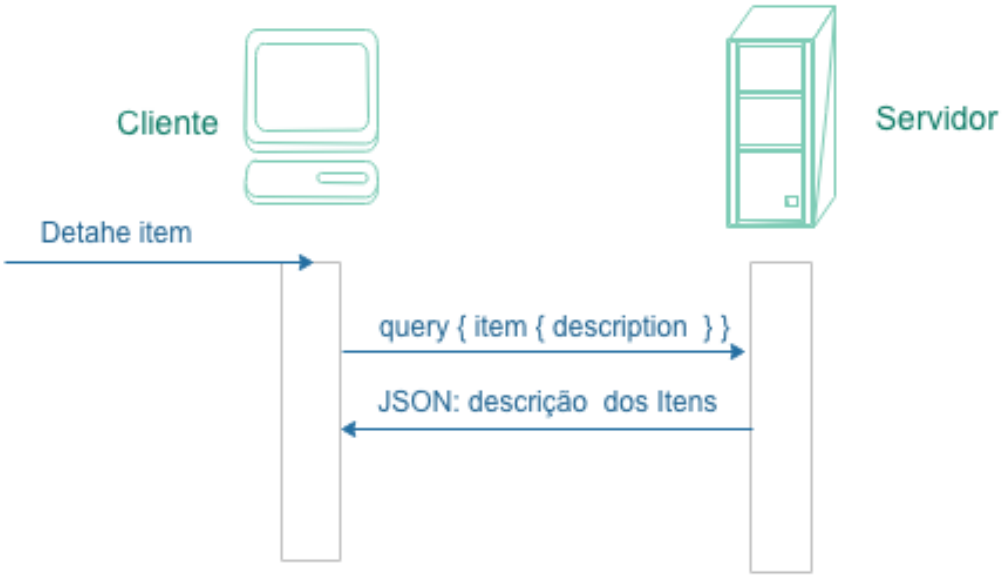

Figura 13 – Fluxo GraphQL

4.3 Métricas

A fim de comparar as medidas de desempenho de APIs desenvolvidas em REST com APIs desenvolvidas em GraphQL, algumas métricas precisam ser definidas. O desempenho de cada API vai depender de sua implementação, porém, escolhendo as métricas corretas, o efeito da implementação pode ser reduzido. Focando em medições corretas, as diferenças relevantes das APIs podem ser destacadas e melhor ponderadas.

Para o presente trabalho, serão utilizadas quatro métricas diferentes: O tempo de utilização da CPU, Consumo de memória, Tempo de resposta e o Tamanho da resposta. É importante ressaltar que cada métrica foi medida separadamente, para que *logs* e *outputs* pertinentes a uma métrica específica não interfira no resultado das demais.

Tempo de utilização da CPU

O tempo de utilização da CPU é a quantidade de tempo para o qual uma unidade de processamento central (CPU) foi usada para processar instruções de um programa de computador ou sistema operacional

Esta métrica será extraída através do módulo *process* presente no *core* do Node.Js. A função utilizada será a *process.cpuUsage()* que trás como resultado o tempo gasto por um processo dentro da CPU, calculado pelo sistema operacional. A seguinte fórmula será usada para calcular esta métrica:

$$CPU = \frac{\sum_{1}^{n} \Delta cpu}{n}$$

Onde n é o número de CPUs e Δcpu é o tempo de uso de CPU pela aplicação, sendo $\Delta cpu = T1 - T2$, em que $T1$ é o tempo final da execução e $T2$ é o tempo inicial. Nos resultados, esta métrica será apresentada na unidade de milissegundos (ms).

Consumo de memória

O consumo de memória, junto com a utilização da CPU, é uma das medidas mais importantes pois são nesses pontos que observamos o verdadeiro custo computacional por trás da escolha da ferramenta. O consumo de memória é a quantidade de em bytes utilizada pela API, e será medido através da soma da utilização de memória em cada consulta necessária para atender os cenários propostos. A seguinte fórmula será usada para calcular esta métrica:

$$Mem = \frac{m}{M} * 100$$

Onde m é a quantidade de memória usada pela aplicação e M é o total de memória. Nos resultados, esta métrica será apresentada na unidade de megabytes.

Tempo de resposta

É o tempo que cada requisição levou para realizar a consulta. Esse tempo é calculado a partir do início da requisição, até o retorno da resposta completa. No caso da API REST, esta métrica será cumulativa, ou seja, a soma do tempo de todas as requisições necessárias. A seguinte fórmula será usada para calcular esta métrica:

$$\Delta t = T1 - T2$$

Onde $T1$ representa o tempo que a requisição respondeu a consulta, e $T2$ o tempo que foi solicitado a requisição. Nos resultados, esta métrica também será apresentada na unidade de milissegundos (ms).

Tamanho da resposta

O tamanho da resposta será calculada baseado no tamanho em *bytes* da resposta. Novamente, para a API REST será usada a média das somas de todas as consultas necessárias para obter a resposta desejada. A seguinte fórmula será usada para calcular esta métrica:

$$Tam = \sum_{1}^{n} ti$$

Onde ti é o tamanha da resposta de uma requisição, d n é o número total de requisições. Nos resultados, esta métrica será apresentada na unidade de kilobytes.

A figura 14 mostra como as métricas serão extraídas. A utlização do CPU e o consumo de memória serão medidas utilizando ferramentas do próprio Node.js. Ambas serão medidas via *logs* no código fonte dos protótipos implementados. O tamanho da resposta e o tempo de resposta serão extraídos utilizando a ferramenta Postman, ao final de cada consulta.

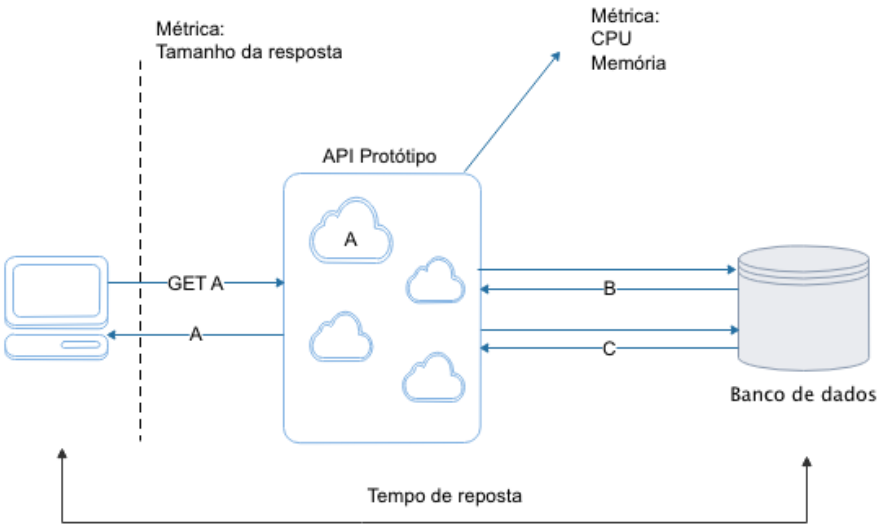

Figura 14 – Arquitetura das APIs e diferentes pontos de medidas

Afim de obter resultados mais precisos, serão feitas diversas iterações para medição de cada umas das métricas. A média destas medições será calculada através da segunte fórmula:

$$Avg = \frac{\sum_{1}^{n} X}{n}$$

Onde n é numero de iterações realizadas para cada métrica, e X é o resultado de cada uma destas iterações.

Após a realização dos testes, os dados provenientes de cada uma das métricas foram coletados, analisados e os resultados serão apresentados no próximo capítulo.

5 Resultados

Neste capítulo, os resultados e a análise dos experimentos realizados durante o trabalho são reportados. Os resultados obtidos nos testes de validação apresentam a média dos valores coletados após diversas execuções dos cenários de forma sequencial.

Foram executadas 30 iterações para cada requisição a fim de obter as respostas das questões levantadas no capítulo 4. Os dados necessários puderam ser buscados em uma única requisição à API GraphQL, enquanto foram necessário duas requisições à API REST para chegar a resposta da primeira questão, e vinte e oito para obter a resposta da segunda. Os detalhes podem ser constatados nas tabelas 3 e 4.

Requisição	**Resultado**	**Número de requisições**
/pallets	Todos os pallets	1
/items/:id	Detalhes do item mais presente	1

Tabela 3 – Fluxo de dados para responder Questão 1

Requisição	**Resultado**	**Número de requisições**
/items	ID do item 22B12	1
/items/:id	Detalhes do item 22B12	1
/pallets	Pallets contendo item 22B12	1
/pallets/:id	Detalhes do Pallet contendo o item 22B12	5
/addresses/:id	Detalhes do Endereço contendo o item 22B12	5
/levels/:id	Nível contendo o item 22B12	5
/slots/:id	Prateleira contendo o item 22B12	5
/rows/:id	Linha contendo o item 22B12	5

Tabela 4 – Fluxo de dados para responder Questão 2

Além das 30 iterações feitas para cada requisição, foram montados também três cenários para melhor análise dos resultados. Estes cenários possuem quantidade de registros diferentes para as entidades Item e Pallet, pois são estas que afetam de maneira mais significativa o quão eficiente será o desempenho das APIs. Estes três cenários estão descritos na tabela 5 com suas respectivas quantidades.

5.1 Questão 1

Embora necessite de buscas mais simples, através da Questão 1 já é possível observar as diferenças em termos de desempenho entre a aplicação REST e a aplicação GraphQL. A Questão 1 busca o item com a maior quantidade de *pallets* alocados no armazém, e para responde-la é necessário duas etapas. A primeira busca todos os *pallets* registrados

Recurso	Cenário 1 (C1)	Cenário 2 (C2)	Cenário 3 (C3)
Item	1000	10000	30000
Pallet	1000	10000	30000
Address	156	156	156
Slot	26	26	26
Row	2	2	2
Level	3	3	3

Tabela 5 – Cenários analisados

no sistema, e após identificado qual o item mais comum presente nos *pallets* registrados, a segunda etapa detalha este item, extraindo a descrição dele por exemplo.

5.1.1 Utilização da CPU

Ao analisar a utilização da CPU, ilustrada na figura 15, percebe-se que as APIs REST e GraphQL tem desempenho similares nos cenários C1 e C2. Por outro lado, as consultas para o cenário C3, mostram a API REST menos eficiente ao utilizar a CPU.

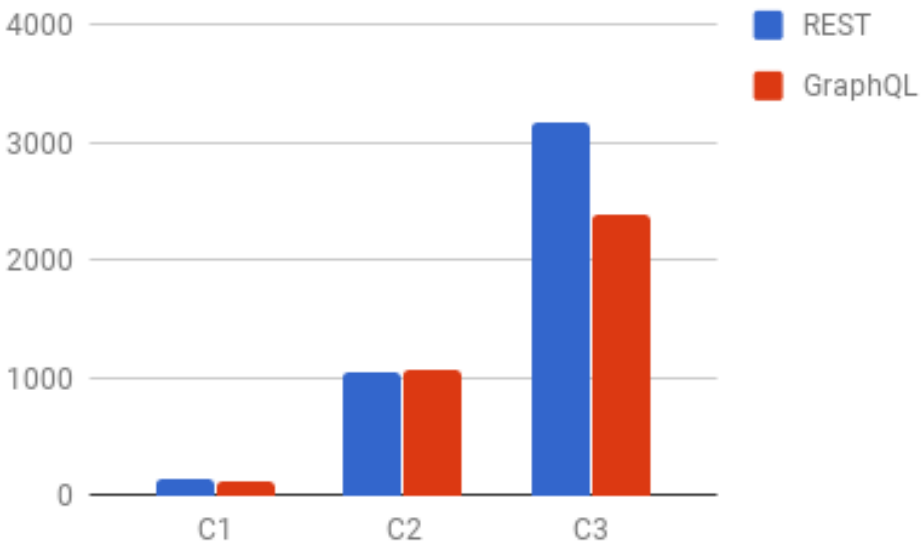

Figura 15 – Comparação da Utilização de CPU

As consultas de C1 exigiram 133,80 ms de CPU na API REST e apenas 111,10 ms na API GraphQL. Executando o cenário C2, a API REST foi processada em 1042,52 ms enquanto a API GraphQL levou 1065,45 ms, e é neste cenário que nota-se a maior similaridade no tempo de utilização da CPU entre as APIs. No entanto, quando analisados os resultados de C3, nota-se uma grande desvantagem para API REST, que utilizou 3177,60 ms da CPU enquanto a API GraphQL usou apenas 2382,10 ms: uma diferença de cerca de 25%.

5.1.2 Consumo de memória

Os dados do resultado da comparação do consumo de memória expõem que a API REST também se mostrou menos eficiente neste quesito em relação à API GraphQL. Os resultados podem ser obervados na figura 16

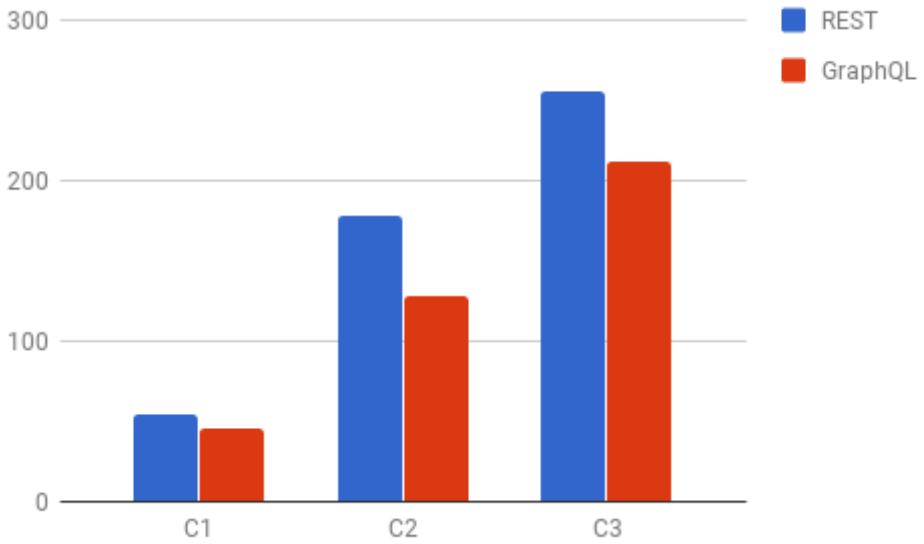

Figura 16 – Comparação do Consumo de Memória

Ao compararmos os resultados de C1 e C3, a API REST se mostrou cerca de 15% menos eficiente que a API GraphQL. O que se destaca é quando compara-ses os resultados de C3, em que a API GraphQL consumiu 127,71 megabytes de memória e a API REST consumiu 178,01 megabytes, uma diferença de aproximadamente 30%.

5.1.3 Tempo de resposta

Como esperado, a API implementada com GraphQL realmente respondeu as consultas em um tempo menor do que a API REST. A figura 17 mostra a diferença do tempo de respota das APIs para executar as consultas da primeira questão.

Nas requisições de C1, a API REST teve como resultado um tempo de resposta de 147,23 ms, enquanto a API GraphQL respondeu a consulta em 115,63 ms, representando uma diferença de 21%. Ao analizar as consultas de C2, a API REST respondeu as consultas em 1108,13 ms e a API GraphQL devolveu os resultados em 925,63 ms, uma diferença de 16%. Por último, as consultas de C3 foram respondidas em 2261,10 ms na API REST e 1725,70 ms na API GraphQL, o que representa uma diferença de 23%.

Para o tempo de respota foi extraído mais um gráfico, que pode ser observado na figura 18, que ilustra o tempo de resposta para cada umas das 30 requisições nas duas APIs. Esse gráfico é baseados nas consultas para o cenário C1, e ajuda a explicar

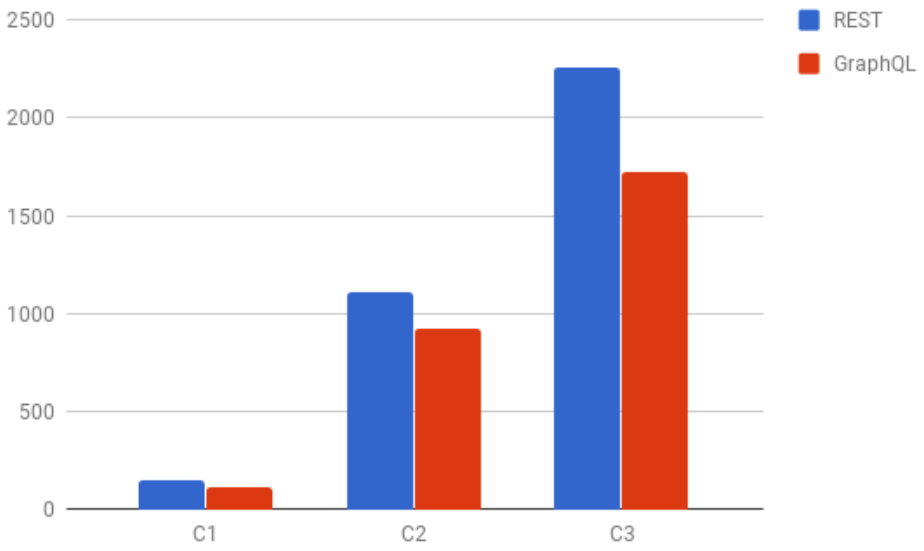

Figura 17 – Comparação do tempo de resposta

o comportamento de tempo resposta de cada uma das APIs. É possivel observar em ambos os protótipos que a primeira requisição leva um tempo bem superior que a média do protótipo. Isso se explica pois na primeira requisição a API precisa de um tempo de *warmup*, necessário para estar em pleno desempenho. Esse período de *warmup* ocorre apenas a primeira requisição e as próximas requisições já se mostram muito mais rápidas.

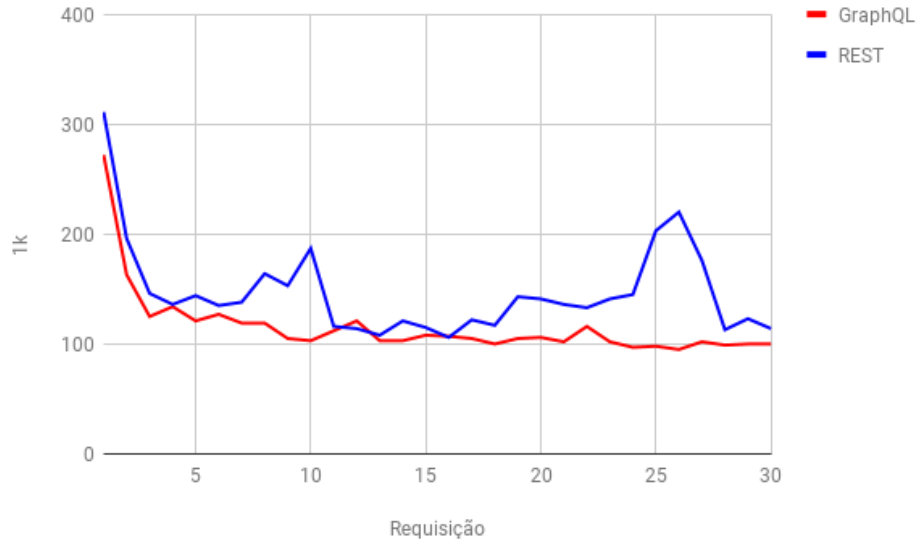

Figura 18 – Tempo de resposta

5.1.4 Tamanho da resposta

Outro resultado esperado era que o tamanho da resposta da API GraphQL fosse menor do que o tamanho da resposta da API REST. Essa hipótese se confirmou como pode ser visto da figura 19.

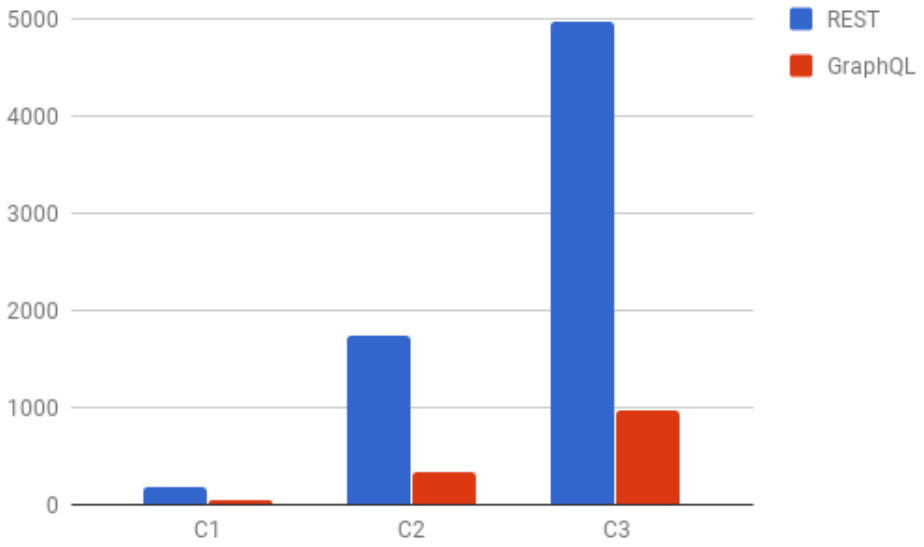

Figura 19 – Comparação do tamanho de resposta

A API REST respondeu as requisições da Questão 1 com um tamanho de resposta de 174,04 kilobytes, 1740,17 kilobytes e 4980 kilobytes para C1, C2, C3 respectivamente. Da mesma maneira, a API GraphQL teve como resultado respostas com 31,68 kilobytes, 322,53 kilobytes e 967,35 kilobytes. Comparando os três cenários é constatado uma diferença constante de cerca de 80% entre a API REST e a API GraphQL.

5.2 Questão 2

Para as repostas da questão 2, as consultas foram mais complexas na API GraphQL, e mais numerosas na API REST. Estas consultas visaram responder quais são os endereços que contém o Item 22B12, e foi analisando os resultados que é possível identificar a maior diferença de desempenho das APIs, com destaque ao tamanho da resposta obtida.

5.2.1 Utilização da CPU

Analisando os resultados da utilização de CPU, podemos concluir que a API GraphQL utiliza este recurso de uma maneira mais eficiente do que a API REST. A figura 20 ilustra os resultados obtidos.

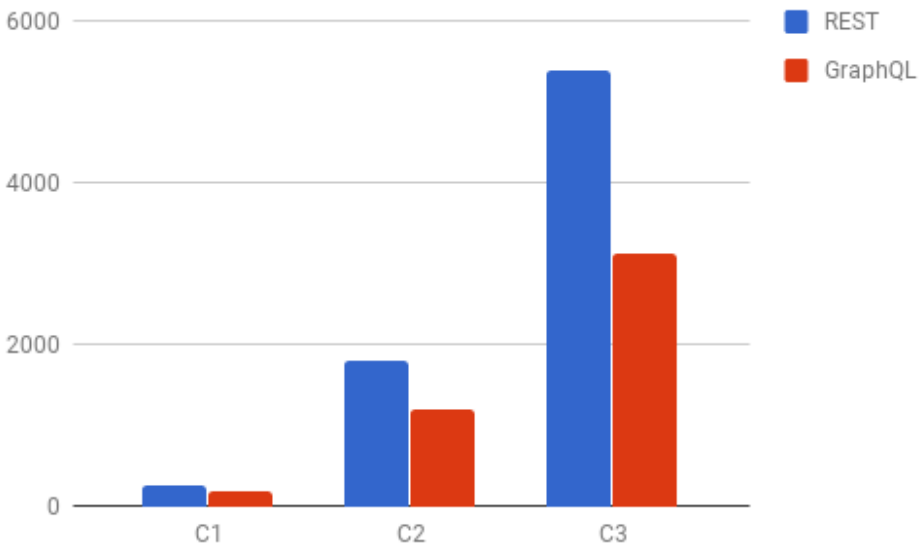

Figura 20 – Comparação da Utilização da CPU - Q2

Aqui a diferença nos resultados obtidos é notada com mais clareza, à medida em que se aumenta o número de registros. Analizando C1, a API REST exigiu 244,41 ms da CPU para ser processada, ao mesmo tempo que a API GraphQL exigiu somente 178,22 ms. Nas consultas de C2, a API REST demandou 1787,74 ms para ser processada, e a API GraphQL demandou 1199,53 ms. A maior diferença encontra-se em C3, onde a API REST levou 5383,40 ms para ser completamente processada pela CPU e a API GraphQL levou apenas 3132,98 ms. No cenário C3 observamos uma diferença de mais de 40% entre o desempenho das APIs.

5.2.2 Consumo de memória

Os dados referentes ao consumo de memória estão ilustrados na figura 21, e apontam que a API REST faz uso menos eficiente deste recurso, comparando com o consumo de memória da API GraphQL. Foi analisando esta métrica que se encontrou o único cenário em que a API REST se mostrou mais eficiente que a API GraphQL.

O consumo de memória não se mostrou tão diferente comparando as APIs nos cenários C1 e C2. Nas requisições de C1, a API REST demonstra ser mais eficiente, mesmo que a diferença seja de apenas 9,88 megabytes (ou 12%), em relação a API GraphQL. A API GraphQL consumiu 76,19 megabytes de memória, contra 66,27 consumidos pela API REST. Essa melhor eficiencia já não é mais identificada em C2, onde a API REST consumiu 181,02 megabytes de meória, e a API GraphQL consumiu 26% mais, totalizando 133,52 megabytes. Nas consultas de C3, observamos uma diferença relevante no consumo de mémoria, com a API REST consumindo 300,30 megabytes de memória enquanto a API GraphQL consumiu 206,02 megabytes, uma impressionante diferença de quase 100

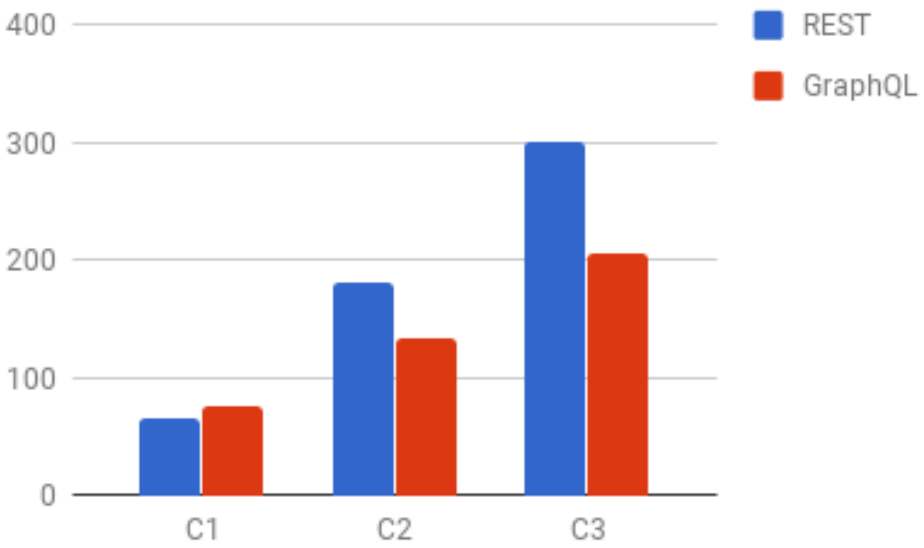

Figura 21 – Comparação do Consumo de memória Q2

megabytes ou 30%.

5.2.3 Tempo de resposta

Como mostra a figura 22, é possivel identificar com clareza que a API REST leva um tempo consideravelmente maior para responder todas as requisições comparando com o tempo levado pelas consultas na API GraphQL.

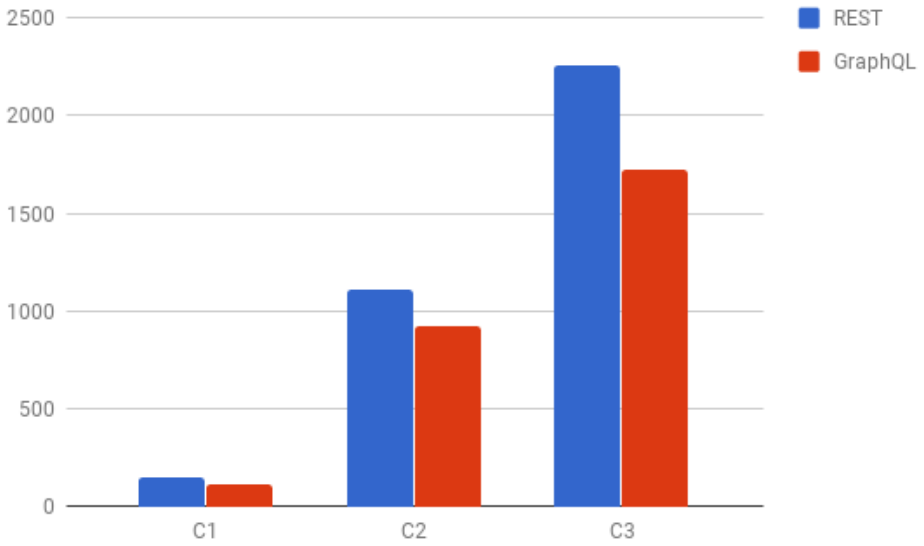

Figura 22 – Comparação do tempo de resposta Q@

Considerando as consultas de C1, a API REST teve como resultado um tempo de resposta de 254,56 ms, enquanto a API GraphQL respondeu a consulta em 148,50 ms,

representando uma diferença de pouco mais de 40%. Quando analisado as requisições de C2, a API REST respondeu as consultas em 2072,03 ms e a API GraphQL devolveu o resultado em 1201,30 ms, uma diferença de 42%, muito semelhante ao C1. Por último, as consultas de C3 foram respondidas em 4770,20 ms na API REST e 2291,10 ms na API GraphQL, o que representa uma diferença de pouco mais de 50%.

Como feito na Questão 1, também foi extraído um gráfigo ilustrando o tempo de resposta para cada umas das 30 requisições, no cenário C1. Entretanto, diferente do que ocorreu na Questão 1, onde grande parte das requisições levaram tempos muito similares, para a Questão 2 obervamos uma diferença muito mais significativa. Os resultados podem ser vistos na figura 23

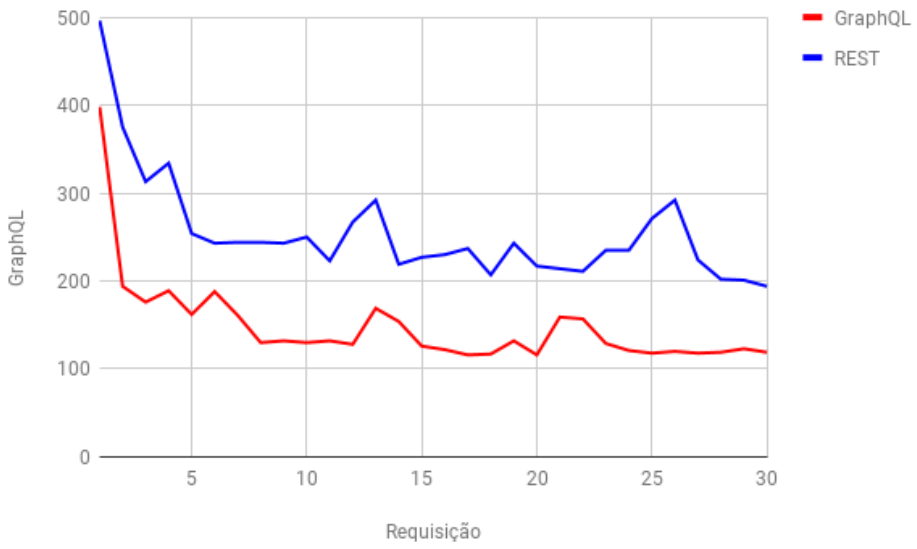

Figura 23 – Tempo de resposta

O período de *warmup* é também facilmente oberservado na Questão 2. A API REST responde a primeira requisição em quase 500 ms, enquanto a API GraphQL responde a primeira requisição em pouco menos de 400 ms. Essa diferença de quase 100ms se mantém ao decorer das requisições, sendo que a API GraphQL se mostra um pouco mais estável do que a API GraphQL, sem grandes ocilações.

5.2.4 Tamanho da resposta

A API GraphQL também mostrou-se mais vantajosa em termos de tamanho da resposta na Questão 2. A figura 24 mostra a comparação entre os protótipos levando em consideração o tempo da resposta.

A API REST respondeu as requisições da questão 1 com um tamanho de resposta de 259,53 kilobytes, 2522.17 kilobytes e 7221,00 kilobytes para C1, C2 e C3 respectivamente. Da mesma maneira, a API GraphQL teve como resultado respostas com 101,73

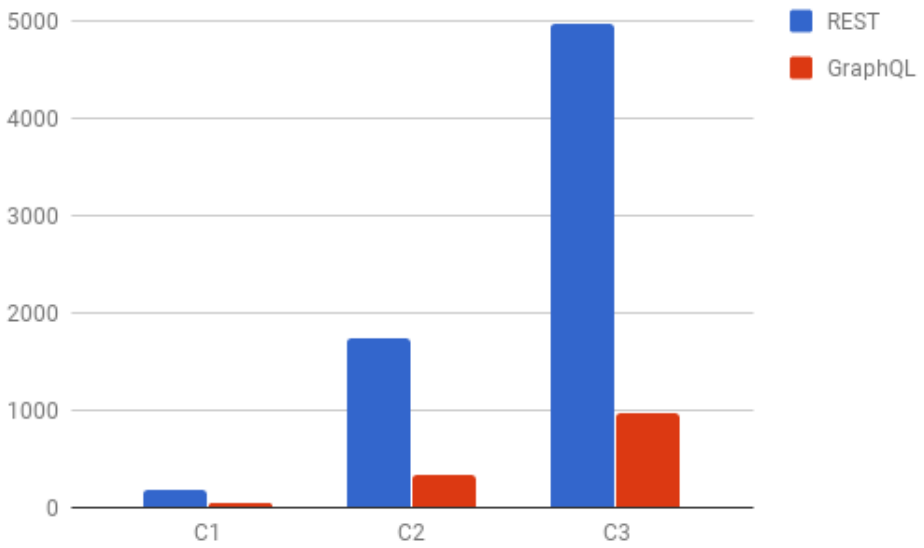

Figura 24 – Comparação do tamanho da resposta

kilobytes, 1005,23 kilobytes e 2850,00 kilobytes. Como ocorrido da Questão 1, a diferença de desempenho entre as APIs se manteve em torno de 60% nos três cenários, deixando claro a melhor eficiencia da API GraphQL neste quesito.

O GraphQL mostrou-se muito estável nos testes realizados levando em consideração todas as métricas analisadas. Em 7 dos 8 cenários propostos, a API GraphQL apresentou um desempenho superior em relação à API REST.

6 Conclusão

Com cada vez mais dispositivos interconectados, a eficiência na comunicação entre aplicações é um tema que está em constante debate e evolução. Recursos computacionais como memórias e processadores, embora gradativamente mais acessíveis, ainda demandam preocupação em relação ao seu uso sustentável, principalmente levando em consideração serviços *on demand* em nuvem.

Este trabalho apresentou conceitos e modelos de comunicação entre aplicações, abrangendo algumas de suas vantagens e desvantagens. Além disso, foram levantadas quatro hipóteses referentes ao desempenho de APIs implementando protótipos de dois dos modelos apresentados: REST e GraphQL. Com a utilização dos modelos, um experimento foi realizado para compará-los. A realização do experimento mensurou o desempenho das APIs em termos de tempo de resposta, quantidade de banda utilizada e consumo de recursos computacionais.

A primeira hipótese sugeria que o tamanho da resposta seria menor para as consultas à API GraphQL. Os resultados do experimento consolidaram esta hipótese como verdadeira, mostrando que a API GraphQL foi em todos os cenários propostos mais eficiente do que a API REST, sendo que em alguns cenários a diferença entre as duas APIs foi de quase 80%. Este resultado já era esperado, pois como princípio, APIs REST expõem recursos, e a consulta destes recursos sempre retornam a representação completa dos objetos. Ainda, APIs GraphQL respondem apenas os campos solicitados pela consulta, resultando em respostas mais concisas.

A segunda hipótese declarava que o tempo de resposta também seria menor para as consultas à API GraphQL. Novamente os resultados confirmaram esta hipótese, e em nenhum dos cenários a API REST se mostrou mais eficiente do que a API GraphQL neste quesito. Em um dos cenários, a API GraphQL respondeu a consulta na metade do tempo necessário para a API REST, sendo que em nenhum cenário a diferença foi menor do que 15%. Os resultados da primeira hipótese podem ajudar um pouco a explicar por que os tempos de resposta da API GraphQL são menores do que os tempos da API REST. Uma vez que é preciso trafegar menos informações pela rede, as consultas são naturalmente respondidas mais rapidamente. Também, outro fator que influencia o tempo de resposta, é quanto tempo a consulta leva para ser processada.

A terceira hipótese afirmava que o tempo de utilização da CPU para as requisições REST seria maior do que o tempo para as requisições GraphQL. Os resultados mostraram que embora em um dos cenários a utilização da CPU foi praticamente igual entre as duas APIs, a medida que é necessário realizar consultas em objetos maiores, o tempo de

utilização da CPU da API REST aumenta signitivamente, alcançando uma diferença de mais de 40% em relação a API GraphQL.

A última hipótese alegava que o consumo de memória da API REST seria menor do que o consumo de memória da API GraphQL. Outra vez, os resultados refutaram esta hipótese. Foi ao avaliar este quesito que o único cenário em que a API REST teve uma eficiência maior do que a API GraphQL foi encontrado. Entretanto, enquanto a diferença de desempenho entre as API foi de apenas 12% no cenário favorável ao REST, no cenário em que o GraphQL foi mais eficiente, a diferença foi de 30%.

Ao fim do trabalho conclui-se que ambas as tecnologias oferecem uma solução prática e eficiente para o mesmo problema, entretanto o GraphQL apresenta-se como uma ótima alternativa como mecanismo de comunicação entre aplicações. A facilidade de uso, menor consumo de recursos computacionais e de banda contribuem para que o GraphQL seja uma ferramenta muito utilizada futuramente pelas organizações e desenvolvedores.

Por outro lado, os critérios de desempenho podem não ser os únicos benefícios trazidos com a utilização do GraphQL. Posto que, os desenvolvedores das aplicações clientes são encarregados de definir cada requisição, eles podem ser bastante seletivos em relação os dados trafegados. Desde modo, o GraphQL ajuda a maximizar a produtividade do desenvolvimento de aplicações, resultando em uma melhor *Developer Experience* (DX), uma disciplina que vem ganhando cada vez mais dedicação e importância dentro das empresas.

Contudo, a maturidade o ecosistema GraphQL ainda é uma questão muito discutida pela comunidade. Embora ele tenha sido usado internamente pelo Facebook por anos, o GraphQL só foi apresentado a um público mais amplo em 2015, e foi apenas em 2016 que o Facebook transferiu seu *status* para *production-ready*.

Trabalhos futuros poderiam abordar como a utilização de arquiteturas baseadas em JSON/Graphs (GraphQL, Falcor), podem otimizar o desenvolvimento de *softwares*, não apenas em uma maneira quantitativa, mas também de maneiras qualitativas, levando em consideração a *Developer Experience*. Outros trabalhos também poderiam abordar as vantagens de ter uma comunicação fortemente tipada, como ocorre com o GraphQL. Ou ainda, se a teoria das categorias pode ser aplicada através do GraphQL.

Referências

BARRY, D. K. *Service Architecture - Representational State Transfer (REST)*. 2003. Disponível em: <https://goo.gl/KB2CGw>. Acesso em: 17 mai 2017.

BATTLE, E. B. R. *Web Semantics: Science, Services and Agents on the World Wide Web*. 1300 North 17th Street, Suite 400 Arlington, VA 22209, United States: BBN Technologies, 2008.

BOX, D. et al. Simple Object Access Protocol (SOAP) 1.1. Http://www.w3.org/TR/soap. 2000.

BUNA, S. *REST APIs are REST-in-Peace APIs. Long Live GraphQL*. 2017. Disponível em: <https://goo.gl/3NDGrq>. Acesso em: 15 abr 2017.

Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. [S.l.], 2017.

CLASS, S. T. *What is Difference Between Two-Tier and Three-Tier Architecture?* 2013. Disponível em: <https://goo.gl/7RMiie>. Acesso em: 15 abr 2017.

EVERWARE-CBDI. *CBDI-SAE Knowledgebase for SOA. Best Practices and Resources for SOA*. 2014. Disponível em: <https://goo.gl/Cy8dWp>. Acesso em: 15 abr 2017.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. AAI9980887.

GRIGORIK, I. *GraphQL Overview*. 2017. Disponível em: <https://goo.gl/nGfQ4x>. Acesso em: 15 abr 2017.

GUSTAVSSON, K. *Efficient data communication between a webclient and a cloud environment*. Dissertação (Mestrado) — Faculty of Engineering (LTH), Lund University, jun 2016.

HICKS, G. *Are Monolithic Software Applications Doomed for Extinction?* 2017. Disponível em: <https://goo.gl/rgCHny>. Acesso em: 15 abr 2017.

KOHLHOFF, R. S. C. *Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems*. 2003. Disponível em: <https://goo.gl/2w8a41>. Acesso em: 17 mai 2017.

KROHN, A. *Do you really need API Versioning?* 2013. Disponível em: <https://goo.gl/lhvEph>. Acesso em: 15 abr 2017.

KUKREJA, N. G. R. Remote procedure call: Limitations and drawbacks. *International Journal of Research (IJR)*, v. 1, n. 10, p. 914–917, 11 2014.

LENSMAR, O. *Is REST losing its flair - REST API Alternatives*. 2013. Disponível em: <https://goo.gl/bPm1ez>. Acesso em: 15 abr 2017.

LEOPOLDO, M. R. B. *Simple Object Access Protocol Entendendo o Simple Object Access Protocol (SOAP)*. 2007. Disponível em: <https://goo.gl/ATS7QB>. Acesso em: 17 mai 2017.

LEWIS, J. *Microservices, a definition of this new architectural term*. 2014. Disponível em: <https://goo.gl/zKZmd8>. Acesso em: 15 abr 2017.

LOPES, C. *O que é Node.js e saiba os primeiros passos*. 2014. Disponível em: <https://goo.gl/SGQuT2>. Acesso em: 15 abr 2017.

MASO, M. *Um Modelo de Comunicação para Automação na Execução de Consultas de Dados sobre APIs Web*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2016. AAI9980887.

MAZIERO, C. A. *Sistemas Operacionais: Conceitos e Mecanismos*. [S.l.], 2013. Acesso em: 01 mai 2017.

MERRICK, P.; ALLEN, S.; LAPP, J. *XML remote procedure call (XML-RPC)*. Google Patents, 2006. US Patent 7,028,312. Disponível em: <https://www.google.com/patents/US7028312>.

Microsoft. *Interprocess Communications*. [S.l.], 2008.

NATIS, Y. V. *Service-Oriented Architecture Scenario*. 2003. Disponível em: <https://goo.gl/NZ9aoW>. Acesso em: 15 abr 2017.

NECS. *NECS entrée V4 SQL and RELATED ADD-ON MODULES SOFTWARE END USER LICENSE AGREEMENT*. 2010. Disponível em: <https://goo.gl/X6zg9p>. Acesso em: 15 abr 2017.

PALIARI, M. *COMO FUNCIONA UM WEBSERVICE REST*. 2012. Disponível em: <https://goo.gl/lJgyrd>. Acesso em: 15 abr 2017.

ROUSE, M. *warehouse management system (WMS)*. 2009. Disponível em: <https://goo.gl/sgh88Y>. Acesso em: 15 abr 2017.

ROUSE, M. *service-oriented architecture (SOA)*. 2014. Disponível em: <https://goo.gl/VeAeTw>. Acesso em: 15 abr 2017.

ROUSE, M. *SOAP (Simple Object Access Protocol)*. 2014. Disponível em: <https://goo.gl/CYyXtD>. Acesso em: 17 mai 2017.

SCHROCK, N. *GraphQL Overview*. 2015. Disponível em: <https://goo.gl/wX2VJ8>. Acesso em: 15 abr 2017.

SKONNARD, A. *Understanding SOAP*. 2003. Disponível em: <https://goo.gl/1G2yiz>. Acesso em: 17 mai 2017.

TANENBAUM, A. V. S.; STEEN, M. V. *Sistemas distribuídos: princípios e paradgmas*. Pearson Educación, 2007. ISBN 9788576051428. Disponível em: <https://books.google.com.br/books?id=r2SGPgAACAAJ>.

THATIKONDA, S. K. *Difference Between Two Tier Architecture and Three Tier Architecture*. 2011. Disponível em: <https://goo.gl/9cSbBg>. Acesso em: 15 abr 2017.

WORK, S. *How Loading Time Affects Your Bottom Line*. 2011. Disponível em: <https://goo.gl/xUpdJ8>. Acesso em: 15 out 2017.

ZHANG, L.-J.; ZHANG, J.; CAI, H. *Service-Oriented Architecture. In: Services Computing*. [S.l.]: Springer, Berlin, Heidelberg, 2007.