UNIVERSIDADE DO VALE DO RIO DOS SINOS — UNISINOS
UNIDADE ACADÊMICA GRADUAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO ANSCHAU NEIS

# UMA ARQUITETURA BASEADA EM MICROSSERVIÇOS PARA APLICAÇÕES IOT EM FAZENDAS INTELIGENTES

São Leopoldo
2018

Rodrigo Anschau Neis

# UMA ARQUITETURA BASEADA EM MICROSSERVIÇOS PARA APLICAÇÕES IOT EM FAZENDAS INTELIGENTES

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação pela Universidade do Vale do Rio dos Sinos — UNISINOS

Orientador:
Prof. Dr. Kleinner Silva Farias de Oliveira

São Leopoldo
2018

## RESUMO

Como a difusão do conceito de IoT tornou-se tendência na indústria apoiada por grandes fornecedores de software e hardware, como Cisco e IBM, gerando novas áreas de pesquisa como cidades inteligentes e fazendas inteligentes. Desta forma, construir arquiteturas de software pensando nestes cenários gerou um novo desafio para os arquitetos de software, a elaboração de arquiteturas que possuem a capacidade para processar milhares de requisições e com baixo tempo de resposta. As pesquisas atuais que utilizam estes conceitos, são em maior parte genéricas, não sendo pensado em cenários especificas como fazendas inteligentes, este estudo terá por objetivo propor um modelo de arquitetura de software, cujo propósito será utilizar-se do conceito conhecido como microsserviços que emergiu como um padrão para construção de aplicações distribuídas a partir um conjunto de pequenos serviços. Desta forma, utilizar-se-á como estudo de caso o desenvolvimento de uma arquitetura responsável por capturar dados de diversos dispositivos, monitorando os dados e gerando ações a partir de regras pré-definidas, este experimento tem como objetivo de verificar a viabilidade da arquitetura proposta.



**Palavras-chave:** Microsserviços. Arquitetura de Software. IoT. Fazendas Inteligentes.

**SUMÁRIO**

# 1 INTRODUÇÃO

O número de dispositivos conectados está crescendo rapidamente. O instituto de pesquisas Gartner prevê que cerca de 21 bilhões de dispositivos até 2020 (GARTNER, 2017). No entanto, a pergunta que surge como esses dispositivos ou "coisas"podem interagir para agregar valor para o usuário. A partir do básico como o elaborar arquiteturas escaláveis que suportam grandes quantidades de requisições e com a capacidade de extrair informações, para este desafio existe um tipo de arquitetura que se encaixa perfeitamente neste contexto, a de microsserviços.

Por Martin Fowler microsserviços é um estilo de arquitetura, que aborda o desenvolvimento de uma aplicação a partir de pequenos serviços, cada um rodando em seu próprio contexto, comunicado-se através de meios de comunicação leves, provendo APIs com suporte a HTTP. Esses serviços são construídos em torno de uma única regra de negócio, sendo implantados de forma independente e totalmente automatizada (FOWLER; LEWIS, 2014).

Dado o crescente desenvolvimento de *frameworks* para o desenvolvimento de microsserviços, bem como a transformação da capacidade computacional na nossa atualidade, cujas possibilidades de acesso de dados gerados por sensores são diariamente amplificados, ademais tem-se ainda crescente necessidade de coletar, processar e extrair informações de valor. Diante do exposto acima, o presente trabalho tem como objetivo geral utilizar-se do conceito conhecido por *Microsserviços* para o desenvolvimento de um modelo de arquitetura de *software* que seja capaz de processar um grande volumes de requisições. Para isto destacam-se os seguintes objetivos específicos:

a) identificar e analisar os *frameworks* disponíveis no mercado para elaborar o modelo de arquitetura.

b) desenvolver um microsserviço para receber requisições de diferentes dispositivos.

c) desenvolver um microsserviço para guardar todos os dados recebidos.

d) desenvolver um microsserviço responsável pela cadastro e verificação de regras.

e) desenvolver um microsserviço responsável por executar ações a partir das regras.

f) analisar os resultados obtidos na utilização da arquitetura proposta, verificando a sua performance perante o número de requisições para avaliar a sua viabilidade.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta os conceitos que contribuem para o bom entendimento do texto. A Seção 3 refere-se aos trabalhos relacionados. Já a Seção 4 traz a arquitetura proposta. Por fim, a Seção 5 descreve os resultados da avaliação empírica, logo a Seção 6 apresenta as conclusões deste projeto.

# 2 FUNDAMENTAÇÃO TEÓRICA

Nesta Seção são descritos os principais conceitos que auxiliarão na compreensão do conteúdo a ser apresentado ao longo deste artigo. A Seção 2.1 introduz o conceito de software. A Seção 2.2 descreve a definição de arquitetura de software e descreve as suas variações, enquanto a Seção 2.3 traz uma visão geral do conceito de Spark. A Seção 2.4 explica a tecnologia Kafka, enquanto a Seção 2.5 explica o conceito de IoT.

## 2.1 Software

O software é a parte programável de um sistema de processamento de dados. Trata-se de elemento chave na criação de sistemas, tem como objetivo realizar instruções complexas e flexíveis que trazem funcionalidades, utilidades e valor ao sistema. Contudo, outros componentes também são indispensáveis no desenvolvimento de softwares como: as plataformas de hardware, os recursos de comunicação e transmissão de dados, os documentos de diversas naturezas, as bases de dados e até os procedimentos manuais que se integram aos automatizados (FILHO; PADUA, 2009).

## 2.2 Arquitetura de Software

A arquitetura de software é o processo de definir uma solução estruturada que atenda todos os requisitos técnicos e operacionais de um sistema, ao mesmo tempo que otimiza atributos comuns de qualidade como desempenho, segurança e capacidade de gerenciamento. Envolve uma série de decisões baseadas em uma ampla gama de fatores, e cada uma dessas decisões pode ter um considerável impacto na qualidade, desempenho, manutenção e sucesso geral da aplicação. Cabe ainda, a arquitetura de software buscar construir uma ponte entre os requisitos do negócio e os requisitos técnicos através do conhecimento dos casos de uso, encontrando assim maneiras de implementar esses casos de uso no software (SILVEIRA, 2012).

Uma boa arquitetura reduz os riscos comerciais associados à construção de uma solução técnica. Um bom design é suficientemente tolerante para lidar com a degradação natural que ocorrerá ao longo do tempo na tecnologia de hardware e software, bem como nos cenários e requisitos de usuários. Um arquiteto deve considerar o efeito geral das decisões do design, os compromissos inerentes entre atributos de qualidade (como desempenho e segurança) e as compensações necessárias para atender aos requisitos do usuário, sistema e negócios (MICROSOFT, 2009).

A arquitetura de software pode ser dividida entre os seguintes modelos: monolítica, orientada a serviços, microsserviços, orientada a eventos entre outras. Cada estilo de arquitetura tem suas peculiaridades e padrões. Nos tópicos a seguir será descrito as principais características de cada estilo das arquiteturas mais utilizadas pelo mercado de software.

### 2.2.1 Arquitetura Monolítica

Arquitetura monolítica é um modelo tradicional para o projeto de um software. Neste contexto, monolítico significa um sistema composto em uma única parte. Software monolítico é projetado para ser autônomo; os componentes do sistema são interligados e interdependentes em vez de fracamente acoplados, como é o caso dos softwares modulares.

Em uma arquitetura bem acoplada, todos os componentes da arquitetura devem estar presentes para que o código seja executado ou compilado. Nestas aplicações todos os componentes são desenvolvidos em um único repositório compartilhado por todos desenvolvedores, quando os desenvolvedores querem adicionar ou modificar um componente eles precisam garantir que todos os serviços estejam funcionando. A complexidade aumenta quando mais serviços são adicionados limitando as oportunidades das empresas em inovar com novas versões e funcionalidades. Além do que, quando novas versões de aplicações são implantadas em produção o conjunto completo de serviços são reiniciados provendo desagradável experiência para os usuários que estão utilizando os serviços. Uma implantação monolítica também representa um único ponto de insucesso, se aplicação falha todo o conjunto de serviços ficam fora de serviço (VILLAMIZAR et al., 2015).

### 2.2.2 Arquitetura Orientada a Serviços

A arquitetura orientada a serviços (SOA) permite que as funcionalidades de uma aplicação seja fornecida como um conjunto de serviços (*Web Services*), além de permitir o desenvolvimento de aplicações que utilizem serviços. Os serviços são fracamente acoplados porque o padrão utilizado é baseado no uso de interfaces que podem ser invocadas, publicadas e descobertas. Os serviços na arquitetura SOA estão focados em fornecer um esquema que contem o padrão aceito e a interação baseada em mensagens através de interfaces de escopo, e não através de componentes ou objetos. Um serviço SOA não deve ser tratado como um provedor de serviços baseado em componentes (ERL, 2016).

O estilo SOA pode empacotar processos empresariais em serviços interoperáveis, usando uma variedade de protocolos de rede e formatos de dados para receber e transmitir informação. Os clientes e outros serviços podem acessar serviços locais executados na mesma camada ou acessar serviços remotos através de uma conexão de rede (KUMARI; RATH, 2015).

Por Erl et al. (2014) os principais princípios da arquitetura SOA são:

- **serviços são independente:** Cada serviço é mantido, desenvolvido, implantado e versionado de forma independente.
- **Serviços são distribuídos:** Os serviços podem estar implantados em qualquer lugar, podendo ser locais ou remotos se a rede suportar os protocolos de comunicação requeridos.
- **Serviços são fracamente acoplados:** Cada serviço é independente de outros, e podem

ser substituído ou atualizados sem impactar as aplacações quer estão utilizando enquanto a interface continua compatível.

- **Serviços compartilham o esquema e contrato não classe ou objeto:** Serviços compartilham contratos e esquemas na comunicação de dados ao invés de classes internas.
- **Compatibilidade e baseada na politica:** politica neste caso representa os aspectos como transporte, protocolos e segurança.

As aplicações que utilizam a arquitetura SOA geralmente possuíam características como compartilhamento de informações, tratamento de múltiplos processos, sistemas de reservas e lojas online, expondo dados ou serviços específicos do setor e combinar informações de várias fontes (MICROSOFT, 2009).

### 2.2.3 Arquitetura de Microsserviços

Nos últimos anos a arquitetura de microsserviços está conquistando grande popularidade na indústria. Essa arquitetura pode ser considerada uma simplificação e refinamento da arquitetura orientada a serviços (SOA). A ideia principal consiste ( em ao invés de arquitetar aplicações monolíticas), poder-se obter uma infinidade de benefícios ao criar-se vários serviços independentes que possam trabalhar em conjunto (AMARAL et al., 2015).

Os microsserviços são aplicações pequenas que possuem uma única responsabilidade e de forma independente podem ser implantadas, dimensionadas e testadas. O microsserviços refere-se a um estilo de desenvolvimento sob o qual um sistema é dividido em pequenos componentes onde esses componentes se comunicam através de uma interface de rede ponto a ponto (por exemplo, http) ou usando uma tecnologia orientada a eventos, como Apache Kafka ou RabbitMQ (O'CONNOR; ELGER; CLARKE, 2017).

Por Newman (2015) os principais benefícios da arquitetura de microsserviços são:

- Sistemas altamente modulares e desacoplados que são mais simples de se manter do que uma tradicional hierarquia de classes.
- A capacidade de implantar serviços rapidamente em produção pois os serviços podem ser implantados independentemente, sendo necessário testar apenas o serviço que foi alterado não impactando o resto da aplicação.
- Microsserviços são unidades de código altamente coesas que são fáceis de gerenciar isoladamente; Isso tende a reduzir o fardo para os desenvolvedores, se forem implementados de forma responsável, podera-se obter um código simples e com menos defeitos.

Deve-se levar em consideração que a utilização de uma arquitetura de microsserviços pode gerar custos, como a sobrecarga computacional ao executar diversos serviços em diferentes processos, custo da comunicação via rede e maior custo de manutenção pelas diferentes linguagens de programação empregadas nos microsserviços Fowler e Lewis (2014).

### 2.2.4 Arquitetura Orientada a Eventos

Arquitetura orientada a eventos é um estilo de arquitetura que se baseia em aspectos fundamentais de notificação de eventos para facilitar a propagação imediata da informação e também em execução reativa dos processos. Em uma arquitetura orientada a eventos, as informações podem ser propagadas praticamente em tempo real através de um ambiente altamente distribuído, permitindo que a organização responda proativamente às atividades comerciais. A arquitetura orientada a eventos permite as organizações comunicação com baixa latência e altamente reativa, melhorando as técnicas tradicionais de integração de dados, como replicação de dados feita através de lotes de mensagens ou relatórios de negócios. Desta forma, modela-se os processos de negócios em transações discretas (comparado aos fluxos de trabalho sequenciais) oferecendo uma grande flexibilidade nas mudanças de condições, além de criar uma forma apropriada para gerenciar as partes assíncronas de uma organização (CHOU, 2016).

A arquitetura orientada a eventos consegue lidar com fluxos de eventos complexos utilizando o modelo de processamento de eventos chamado de *Complex Event Processing* (CEP). Um dos principais conceitos do CEP é que os eventos não são independentes um do outro, mas correlacionados. Dados de sensores tendem ser extremamente correlacionados tanto em tempo e espaço. As leituras observadas em um instante são um indicio para as próximas leituras. A principal tarefa do CEP é descobrir padrões de eventos em um grande fluxo de dados, esses padrões podem gerar novos eventos que possam ser de suma importância para áreas de negócios (DUNKEL et al., 2010).

O trabalho de Chou (2016) descreve que a arquitetura orientada a eventos representa um novo paradigma, tendo como princípios:

- **Padrão de mensagens assíncronas:** ao invés de utilizar o padrão síncrono de requisição/resposta ou o padrão de interação cliente servidor, a arquitetura de eventos e construída sobre o modelo de publicar e subscrever para enviar notificações aos interessados em ouvir as mensagens, sem ficar bloqueando o processamento na espera de uma resposta.

- **Mensagens autônomas:** Os eventos são transmitidos no formato de mensagens autônomas, cada mensagem tem a informação necessária para o trabalho ser realizado. Cada evento não deve conter nenhuma dependência de hardware ou sessão de uma aplicação. Destina-se a comunicar mudanças de estado do negócio para cada aplicação, domínio ou grupo de uma empresa.

- **Sistemas distribuídos com alto desacoplamento:** O modelo de mensagens assíncronas encapsulo detalhes internos de sistemas distribuídos, não sendo necessário o desenvolvimento de padrões de interface ou definições de API. O único requisito necessário é definir mensagens em um formato semântico, que pode ser implementado como XML.

- **O Receptor possui o controle do fluxo de mensagens:** O receptor de um evento que esta escutando a fila de eventos tem o poder de controlar a carga que pode suportar. Os sistemas de controle de mensagens tem funcionalidades para verificar se os eventos estão sendo processados, na arquitetura orientada a eventos tem como característica a escalabilidade assim um único evento pode ser distribuído entre diversos nodos.

- **Propagação em tempo real:** Uma arquitetura de eventos só tem relevância quando os eventos são publicados, consumidos e propagados em tempo real. Se este padrão não é seguido a arquitetura terá a mesma efetividade da arquitetura tradicional orientada a lotes de mensagens.

## 2.3 Spark

Apache Spark é um mecanismo genérico e veloz para o processamento de dados em larga escala (SPARK, 2017). Surgiu como um mecanismo da próxima geração de processamento de dados, superando o Hadoop MapReduce, ajudou a acelerar a revolução do *big data*. Spark mantém a escalabilidade linear e a tolerância de falhas do MapReduce, mas estendendo em aspectos muito importantes como: velocidade de processamento (100 vezes mais rápido para determinadas aplicações), muito fácil de programar devido às suas APIs desenvolvidas em Python, Java, Scala, utiliza uma simples abstração de dados, estrutura distribuída de dados, e vai muito além das aplicações de lotes suportando uma variedade de tarefas intensivas da computação, incluindo consultas interativas, transmissão, aprendizado de máquina e processamento de gráficos (SHANAHAN; DAI, 2015).

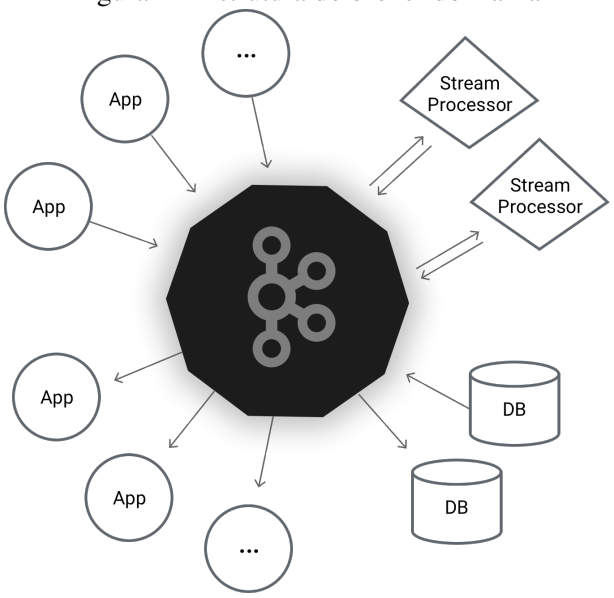
## 2.4 Kafka

Apache Kafka é uma plataforma de *streaming* de dados distribuída capaz de processar trilhões de eventos por dia. Projetada inicialmente para ser uma fila de mensagens, Kafka baseia-se em uma abstração distribuída *commit log*. Foi projetada desde o começo com um plataforma de código aberto, desenvolvida pelo LinkedIn em 2011, Kafka evoluiu rapidamente de uma simples fila de mensagens para uma plataforma completa de *streaming* (NARKHEDE; SHAPIRA; PALINO, 2017).

Como uma plataforma de *streaming*, Apache Kafka prove baixa latência, alto rendimento, tolerância a falhas, publicar e subscrever filas de mensagens, sendo capaz de processar eventos de *streams*. Kafka fornece de forma confiável respostas em milissegundos para suportar aplicações voltadas aos clientes e conectar sistemas de processamento de dados em tempo real. Kafka prove funcionalidades como : *Messaging System*, *Storage System*, *Stream Processing* (KAFKA, 2017)

O Apache Kafka apresenta uma sugestão de estrutura para sua utilização, no modelo proposto, existem aplicações que operam como fonte de dados, enquanto outros consomem as

Figura 1 – Estrutura do broker do Kafka




Fonte: Kafka (2017)

informações geradas. Verifica-se que todo o fluxo de informações passa pelo Broker, que realiza o controle de dados entre os serviços que processam as streams (NEWMAN, 2015). A Figura1 ilustra a estrutura do broker do Kafka.

## 2.5 IoT

O termo Internet of Things (IoT) em português Internet das Coisas, foi primeiro definido por Kevin Ashton em 1999 (MEMON et al., 2016). IoT se trata de um novo paradigma sobre a capacidade de conectar dispositivos com o objetivo de entender e coletar dados, e compartilhar estes dados utilizando a internet, para que os mesmos possam ser processados e utilizados para executar objetivos comuns.

IoT habilita que objetos físicos possam ver, escutar, pensar e realizar trabalhos a partir de conversas em grupo, para compartilhar informações e coordenar decisões. IoT transforma objetos tradicionais em inteligentes, explorando por baixo tecnologias, como computação ubíqua e pervasiva, dispositivos embarcados, tecnologias de comunicação, redes de sensores e protocolos de internet. Objetos inteligentes, juntamente com suas supostas tarefas, constituem um domínio de aplicações específicas (mercados verticais), enquanto que serviços ubíquos e serviços analíticos formam aplicações de serviços independentes de domínio (mercados horizontais) (AL-FUQAHA et al., 2015).

# 3 TRABALHOS RELACIONADOS

Nos últimos anos tem se notado um aumento exponencial nas pesquisas relacionadas a *big data* visto o seu grande potencial em diversas áreas de negocio. A literatura recente tem explorado bastante esse tema, dessa forma este capitulo apresenta a situação atual deste tema, explorando diversas pesquisas e uma comparação com a pesquisa proposta.

## 3.1 Seleção dos Trabalhos

O objetivo deste trabalho será propor a união de três ramos de pesquisa, arquitetura de microsserviços, IOT e fazendas inteligentes. Para descobrir a situação atual destes temas foi utilizado as seguintes palavras chaves: microservices, IOT, Smart Farming e variações destas palavras, além das palavras chaves foi utilizado os seguintes filtros: trabalhos publicados nos últimos 5 anos e publicado em inglês ou português.

Foram utilizadas as principais bibliotecas online de artigos técnicos e acadêmicos, entre elas, a IEEExplore Digital Library, a ACM Digital Library, a Editora Springer e a Editora Elsevier. A consulta e filtros utilizados na pesquisa, foram editados para se adequar aos padrões de consultas específicos de cada base de dados sem alterar os resultados obtidos.

A partir dos resultados obtidos pelas bases de dados, foi realizada heurísticas manuais para a seleção dos trabalhos relacionados, com intuito de descobrir os trabalhos que mais se aproximam com a pesquisa proposta.

## 3.2 Análise dos Trabalhos Selecionados

O estudo de Butzin, Golatowski e Timmermann (2016), apresenta um resumo sobre as melhores praticas e padrões surgidos com o conceito de microsserviços, descrevendo sobre os temas: serviços autocontidos, versionamento de APIs, monitoramento e tolerância a falhas. Este trabalho realizou um comparativo entre a utilização de containers para aplicações e as diferenças entre dispositivos IOT e microsserviços, chegando na conclusão que IOT e microsserviços possuem os mesmos objetivos de arquitetura, tento apenas diferenças e desafios, como os modestos hardwares dos dispositivos IOT.

O trabalho Bak et al. (2015) apresenta três microsserviços que podem acelerar o desenvolvimento de aplicações direcionadas a localização baseadas no contexto do usuário, demonstrando a utilização destes serviços em casos reais de uso como mapa de calor da localização dos usuários. Os microsserviços apresentados tem como função a visualização da localização do usuário, detecção de anomalias verificando os casos que os sensores não são capazes de gerar um dado com exatidão, gerando-se uma anomalia, este serviço detecta estás anomalias e as descarta, o ultimo microsserviço tem como o objetivo de analisar, tratar e entender problemas ligados a negócio, aplicação e anomalias. Os microsserviços apresentados neste trabalho disponibilizam

uma API REST(HTTP) que pode ser consumida por dispositivos IoT e aplicações móveis, foram desenvolvidos utilizando os conceitos de microsserviço, mas sem muitos detalhes de como a implementação foi feita.

O trabalho de Taneja et al. (2018) foca no tema de fazendas inteligentes, utilizando-se do conceito de microsserviço para implementar uma computação em névoa, tendo como principal objetivo de diminuir os custos com servidores em cloud e problemas ligados a conexão com a internet, sendo fazendas geralmente ficam distantes dos grandes centros. Este trabalho desenvolveu como experimento uma aplicação para monitorar o comportamento do gado leiteiro em fazendas, utilizando um sensor de contagem de passos fixados nas pernas frontais dos animais, os dados coletados são transmitidos via frequência de rádio, enviados para microsserviço responsável por analisar os dados na procura de comportamentos relacionados a doenças, ao encontrar tais comportamento é enviado um alerta para o fazendeiro responsável pelo gado leiteiro. Este microsserviço também é responsável por se conectar com servidores em nuvem transmitindo os dados coletados em certo período de tempo. A aplicação em nuvem é responsável por manter o histórico dos dados, analisar os dados e fazer processamentos de inteligencia artificial na procura de padrões.

O Trabalho Krylovskiy, Jahn e Patti (2015) está inserido no ramos de pesquisa de dispositivos IoT na construção de cidades inteligentes. O autor propõe a utilização de microsserviços, no desenvolvimento de uma arquitetura responsável por controlar as luzes de uma cidade, suportando a pesquisa de novos dispositivos através de consultas leves via HTTP. Arquitetura desenvolvida utiliza tecnologias como filas de mensagens e consultas semânticas.

## 3.3 Análise Comparativa e Oportunidades de Pesquisa

Muitos trabalhos já foram conduzidos nesta área, descrevendo as principais características, para o desenvolvimento de arquiteturas de software genéricas ou estudos da bibliografia. A comparação entre os projetos abrange quesitos de vital importância para o trabalho proposto, tais como o tipo de trabalho, a fonte de dados, o modelo de solução, a utilização de containers de aplicação, filas de mensagens.

Cada critério descrito anteriormente foi avaliado em separado para cada um dos trabalhos relacionados. Os resultados desta comparação podem ser observados na 1.

É possível verificar que, apesar da grande quantidade de trabalhos relacionados ao tema proposto, existe uma deficiência na questão de processamento em tempo real e suporte a diferentes tipos de dispositivos, por este motivo a arquitetura proposta tem como principal objetivo processar uma grande quantidade de dados, ser genérica para qualquer aparelho que tenha a capacidade de fazer requisições HTTP, fornece uma estrutura dividida em módulos e ser reaproveitável. A arquitetura proposta faz o uso dos conceitos de mensagens e coreografia na responsabilidade dos componentes de serviços para construir módulos com baixo acoplamento, mesclando características das arquiteturas de microsserviços e orientadas a eventos, este mo-

Tabela 1 – Tabela 1 – Análise Comparativa

| | Tipo de Trabalho | | | | Uso de Tecnologias | | | Tipo de modelo | | | | Tipo de Composição | | Visualização | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Revisão da Literatura | Experimento Controlado | Ferramenta | Arquitetura | Containers | Fila de Mensagens | Chamadas Reativas | Arquitetura | Algoritmo | Ferramenta | Modelo Conceitual | Orquestração | Coreografia | Painel de Controle | Dados Estruturados |
| (BUTZIN; GOLATOWSKI; TIMMERMANN, 2016) | P | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| (BAK et al., 2015) | ← | ← | ← | P | - | P | - | P | - | P | - | P | - | P | - |
| (TANEJA et al., 2018) | - | P | - | P | P | P | - | P | - | - | - | - | P | P | - |
| (KRYLOVSKIY; JAHN; PATTI, 2015) | ← | P | - | P | P | P | - | P | - | - | - | - | P | P | - |
| Arquitetura Proposta | - | P | - | P | P | P | P | P | - | P | - | - | P | - | P |

Legenda: Suporta (P) Não suporta (-) Suporta parcialmente (←) Não se aplica (?)

delo implementação favorece a construção de estruturas expansíveis, reaproveitáveis, escaláveis e com alta elasticidade, contribuindo para o processamento de grandes quantidades de dados.

# 4 ARQUITETURA PROPOSTA

A presente seção é dedicada à descrição da arquitetura desenvolvida. Explicando a integração e comunicação entre os componentes propostos. Para isso, a seção 4.1 apresenta uma visão geral da arquitetura, detalhando as tecnologias empregadas, a seção 4.2 descreve com maiores detalhes o funcionamento de cada componentes elabora, descrevendo suas funcionalidades e objetivos, sendo que a seção 4.3 elenca as principais características da arquitetura proposta, já a Seção 4.4 demonstra os aspectos de implementação da arquitetura proposta, elucidando as tecnologias e ferramentas empregadas no desenvolvimento deste protótipo, a Seção 4.5 apresenta os requisitos da arquitetura.

## 4.1 Visão Geral

A arquitetura desenvolvida tem como objetivo receber uma grande volume de dados, além de ser capaz de realizar processamentos em tempo real, com foco também de coletar dados gerados por dispositivos *IOT*, concentrando-se principalmente nos dados coletados por sensores. Está arquitetura foi projetada para suportar diversos formatos de dados, assim sendo genérica.

A figura 2 representa uma visão geral da arquitetura, neste diagrama é possível verificar o fluxo básico de como é realizado o processamento de dados. O tipo de arquitetura escolhida para lidar com este desafio é uma mistura dos conceitos de arquitetura orientada a eventos e arquitetura de Microsserviços. No diagrama apresentado pode verificar-se a existência de um *broker* de mensagens, sendo este o ponto chave desta arquitetura. Nota-se ainda que de que todos os eventos transmitidos através do *broker* este componente é responsável pela coordenação da mensagens transmitidas ente os produtores e consumidores.

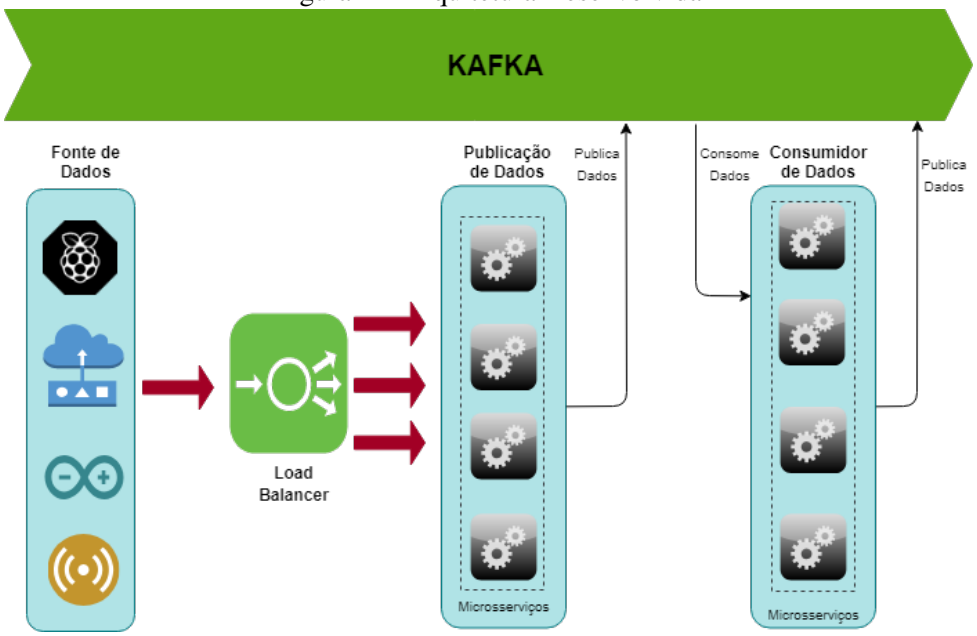
Os dados serão transmitidos via chamada REST, essas chamadas podem ser feitas a partir de qualquer dispositivo que implementa o método POST do HTTP. A carga destas requisições são distribuídas entre os nodos disponíveis do microsserviço produtor, utilizando uma aplicação denominada de *Load Balancer*. Está aplicação é responsável por descobrir e balancear a carga entre as instâncias ativas dos microsserviços produtor de mensagens.

O modulo produtor de mensagens é responsável por receber as chamadas de serviços, para transmitir o evento no determinado tópico correspondentes no *broker* de mensagem.

Os microsserviços do tipo consumidor tem a finalidade de receber as mensagens do *broker* para fazer o seu processamento, neste microsserviços serão feitos processamentos em tempo real, com o intuito de gerar dados analíticos e realizar ações, como, por exemplo, o envio de um e-mail contendo um alerta. Essas ações podem ser configuradas via cadastro, sendo que cada uma dessas ações pode ser um novo tópico no *broker* de mensagens tendo um microsserviço responsável por realizar este processamento.

Os detalhes de implementação e estrutura dos microsserviços serão abortados com maior profundidade nas próximas seções.

Figura 2 – Arquitetura Desenvolvida




Fonte: elaborado pelo autor

## 4.2 Projeto Arquitetural

A arquitetura proposta é dividida entre seis microsserviços, cada um desempenha uma função e tem uma finalidade dentro da arquitetura geral.

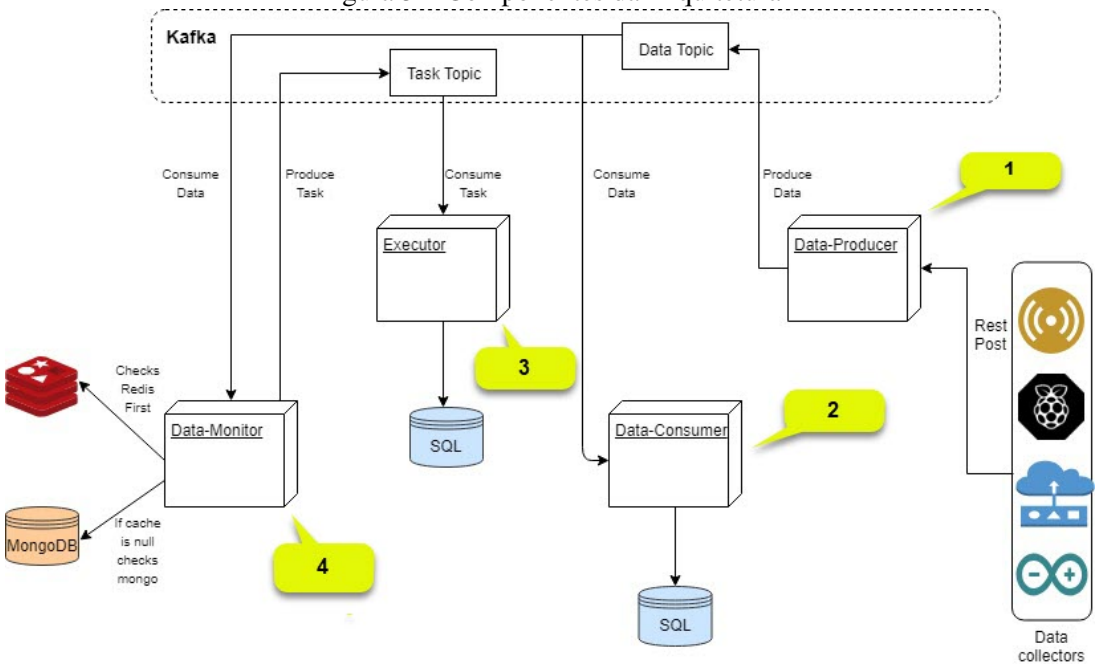
A figura 3 demonstra os componentes desenvolvidos, e como é feito o fluxo de dados.

1. **Microsserviço Data-Producer:** Este microsserviço tem a responsabilidade de receber dados através de chamadas REST, estás chamadas recebe como *Body* um JSON. As chamadas de serviço podem ser enviadas por diversos dispositivos por exemplo: sensores de temperatura e umidade, Arduino, Raspberry pi ou qualquer aparelho que suporta o protocolo de rede HTTP. Depois de receber a chamada REST este microsserviço publica O JSON recebido no tópico "data"do Kafka.

2. **Microsserviço Data-Consumer:** O Data-Consumer é responsável por consumir o tópico "Data"do Kafka e persistir a mensagem no banco de dados relacional, esse microsserviço disponibiliza uma API REST para listar os dados persistidos na base. Este serviço contém filtros para retornar os dados, se nenhum filtro for passado todos os dados serão retornados.

3. **Microsserviço Data-Monitor:** Este microsserviço consome o tópico "Data"do Kafka analisando as mensagens, para verificar se há existência de um monitor correspondente, quando uma mensagem possui um monitor correspondente é executada uma tarefa previamente registrada no monitor. As tarefas são publicadas no tópico "Task"do Kafka para serem processadas pelo microsserviço Executor.

Figura 3 – Componentes da Arquitetura




Fonte: elaborado pelo autor

Este microsserviço disponibiliza uma API para criar, editar, excluir e listar todos os monitores. O Objeto monitor é composto por um conjunto de regras e uma lista de ações.

**4. Microsserviço Executor:** o Executor consome a mensagem "Task"do Kafka, sendo responsável por processar a tarefa, as tarefas suportadas neste microsserviço, são: envio de e-mail e envio de mensagem no Slack. Se a tarefa recebida do tópico não for suportada é salva uma mensagem de erro na base de dados.

Este Microsserviço disponibiliza um API REST, para consultar as ações processadas, e também as que não foram processadas.

## 4.3 Principais características

- **Plugabilidade de novos componentes:** A partir da utilização de eventos na comunicação entre os componentes da arquitetura, torna-se pratico o desenvolvimento de novos componentes, basta consumir e produzir mensagens no broker para estar ligado a arquitetura.

- **Baixo acoplamento entre os componentes:** Diferente da arquitetura monolítica que todos os módulos eram implantados no mesmo pacote, sendo a principal forma de comunicação entre os módulos as interfaces. Na arquitetura de eventos o acoplamento é baixo, a partir de que cada modulo pode ser implantado separadamente. Sendo a comunicação feita através de mensagens, não gerando dependências de objetos e contratos entre os microsserviços.

- **Escalabilidade dos componentes:** Como cada modulo da arquitetura desenvolvida tem um única responsabilidade, é possível verificar se algum módulo terá uma maior carga de trabalho. Por exemplo se o modulo Data-Producer receber muitas requisições é possível subir mais instancias desse modulo para aguentar a carga de trabalho.

## 4.4 Aspectos de implementação

Os microsserviços foram desenvolvidos utilizando a linguagem de programação Java 8. Sendo os microsserviços Data-Producer Data-monitor utilizam os *frameworks* Spring Boot 2.0 e Spring Webflux. Utilizando como servidor de aplicação Netty. Essa *stack* foi escolhida tendo em vista o suporte a chamadas reativas não bloqueantes, com o intuito de aumentar a performance do sistema e diminuindo o uso de hardware, para manter o suporte a chamadas não bloqueantes o banco de dados escolhido para o microsserviço Data-Monitor foi o MonngoDB que é um banco não relacional, ainda para esse ser capaz de suportar mensagens em tempo real foi adicionado um cache em memória para armazenar os monitores cadastrados. O cache escolhido foi Redis, tendo em visto o suporte as transações do MongoDB, não sendo necessário a preocupação de atualizar e editar o cache, sendo que o *framework* realiza estás operações automaticamente.

Como o microsserviço Data-Consumer, não tem a necessidade de realizar processamento em tempo real, neste caso a Stack escolhida é a combinação de Spring Boot 1.5, Spring MVC, e servidor de aplicação Apache Tomcat.

O Apache Kafka foi instalado via imagem Docker, sendo executado via Docker Compose. A configuração do Kafka utilizada para suportar essa arquitetura foi:

Tabela 2 – Configuração Apache Kafka Utilizada

| Nº de instâncias | Componente | configuração |
|---|---|---|
| 1 | Apache Zookeeper | Padrão |
| 1 | Cluster Kafka | Padrão |
| 2 | Topico kafka | Personalizada |

Fonte: elaborado pelo autor.

Como é visto na tabela 2 a configuração feita nos tópicos foi personaliza, foram criados dois tópicos um com o nome Data, o segundo com o nome de task, como o tópico Data é consumido por dois microsserviços foi escolhido uma fator de replicação 2, assim cada microsserviço consome de uma replica distinta evitando que mensagens sejam consumidas mais de uma vez, além de um maior controle no consumo de mensagens.

# 5 AVALIAÇÃO DE DESEMPENHO DA ARQUITETURA

Esta Seção apresenta avaliações praticas realizadas na arquitetura/protótipo desenvolvido. A Seção 5.1 apresenta as especificações do ambiente de teste, enquanto a Seção 5.2 apresenta a visão geral da avaliação. Já as seções 5.3, 5.4 e 5.5 demonstra os resultados das avaliações realizadas.

## 5.1 Especificações do Ambiente de Teste

A avaliação de desempenho da arquitetura foi executada em uma máquina virtual, rodando a partir do Windows 10 com o programa Oracle VirtualBox sendo disponibilizado os seguintes propriedades de hardware:

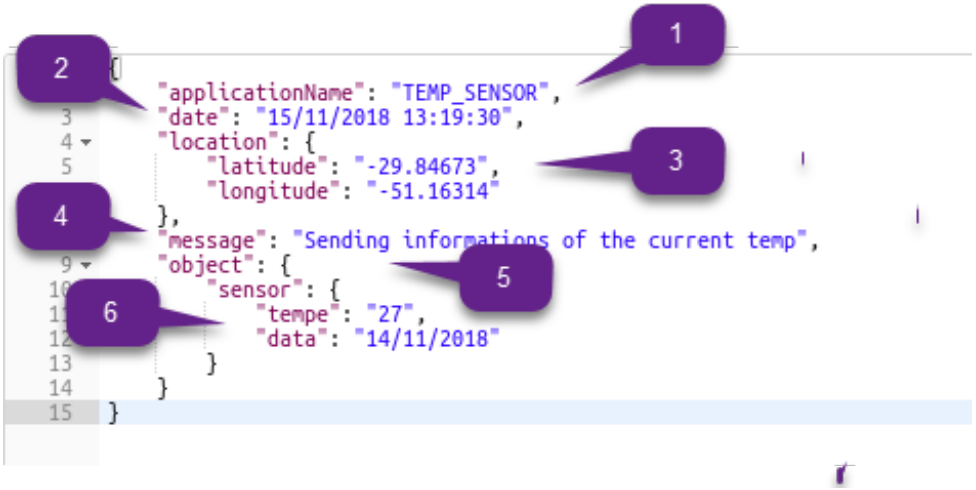
- Processador AMD Ryzen 1600X @4.0GHz com 6 threads disponíveis
- Total de 16 GB memória RAM DDR4 @2400 MHz sendo disponibilizado nos testes entre 8 e 12 Gigabytes
- 30 GB de SSD Samsung Evo 840 (Leitura: 540MB/s. Escrita: 520MB/s)

Para realizar os testes da arquitetura, com o intuito de executar todos os microsserviços para verificar a performance, a máquina virtual foi construída possuindo as seguintes propriedades de software:

- Sistema operacional Ubuntu 18.04.1
- Java 8 Update 181
- Kafka 2.0.1
- Docker CE 18.09
- Docker Compose 1.23.1
- MySql Server 8.0.13
- MongoDB Community Server 4.0.4
- Gradle 4.10.2
- IDE IntelliJ IDEA Community 2018.2
- Apache JMeter 5.0

As avaliações de performance realizadas tem o intuito de verificar a capacidade da arquitetura em lidar simultaneamente com diversos dispositivos conectados, e a capacidade de lidar com o processamento em tempo real das informações capturadas. Para isso, foi utilizado um arquivo JSon com os atributos simulando os dados gerados por um sensor de temperatura, este arquivo foi utilizado como *Body* da chamada de serviço POST para o microsserviço Data-Consumer. O Json conta com 333 bytes de dados. A figura 4 exibe o objeto JSon utilizado bem com sua estrutura. A seguir será elencado brevemente cada um dos seus atributos:

Figura 4 – Objeto JSon utilizado




Fonte: elaborado pelo autor

**1. Application Name:** Neste atributo é passado o nome da aplicação que está enviando os dados, sendo neste exemplo utilizado o valor "TEMP_SENSOR"demonstrando que está chamada é realiza por algum dispositivo que está conectado com um sensor de temperatura.

**2. Date:** Este atributo recebe a data atual no formato (dd/MM/yyyy HH:mm:SS). Este valor pode ser utilizado para verificar se a mensagem já foi processado além de servir como histórico.

**3. Objeto Location:** Neste objeto é recebido os dados das coordenadas do dispositivo.

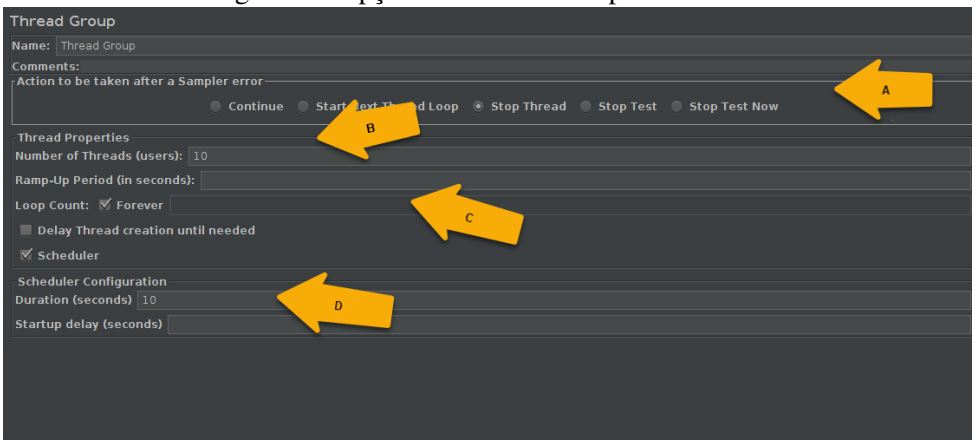
**4. Message:** O atributo message é utilizado para passar mensagens, este campo pode ser utilizado para realizar algum processamento no microsserviço Data-Monitor, por exemplo se é adicionado um monitor tendo como regra o campo message, para verificar se existe a palavra Alerta no texto. Pode ser adicionado uma ação de enviar e-mail para os interessados.

**5. Object:** Este campo pode receber tipo de objeto, contendo chave e valor, podendo o campo valor ser um objeto ou tipo simples. Ë possivel enviar mais de um item dentro do campo Object, sendo apenas necessário ter chaves distintas.

**6. Objeto Sensor:** Neste caso como a chamada é do tipo "TEMP_SENSOR"é enviado o objeto Sensor que contem os atributos tempe o mesmo contendo a temperatura que o sensor coletou, já o campo data pode ser enviado qualquer tipo de dados. Se fosse algum outro tipo de aplicação poderia-ser enviado um outro tipo de objeto.

Como o objetivo verificar a performance da arquitetura, assim verificando a quantidade de chamadas suportadas por segundo, em dispositivos conectados em paralelo. Para realizar este teste de stress, utilizou-se a aplicação Apache Jmeter, sendo uma ferramenta bem intuitiva de fácil utilização. Abaixo a figura 4 demonstra a tela com as opções de teste utilizadas neste experimento. Elencando cada uma das opções empregadas:

Figura 5 – Opções utilizadas no Apache JMeter




Fonte: elaborado pelo autor

**A. Qual ação será feita quando for encontrado uma amostra com erro :** Foi selecionada a opção desligar a thread quando algum erro for encontrado, porque em testes de stress quando uma aplicação para de responder, o limite de requisições foi alcançado, o servidor não sendo capaz de responder.

**B. Número de threads:** está opção serve para escolher o numero de requisições que serão realizas em paralelo.

**C. Quantidade de chamadas:** foi escolhido a opção de fazer um laço infinito de chamadas, este campo também suporta a opção de determinar o numero de chamadas, chegando neste limite o teste é encerado.

**D. Tempo de execução do teste em segundos:** Neste campo suporta a quantidade de tempo que o teste deve ser executado, caso nenhum valor seja colocado, o teste ficara rodando até que a quantidade de chamadas seja atingida ou o teste seja encerrado por alguma ação manual.

## 5.2 Teste de stress por tempo de execução e número de threads

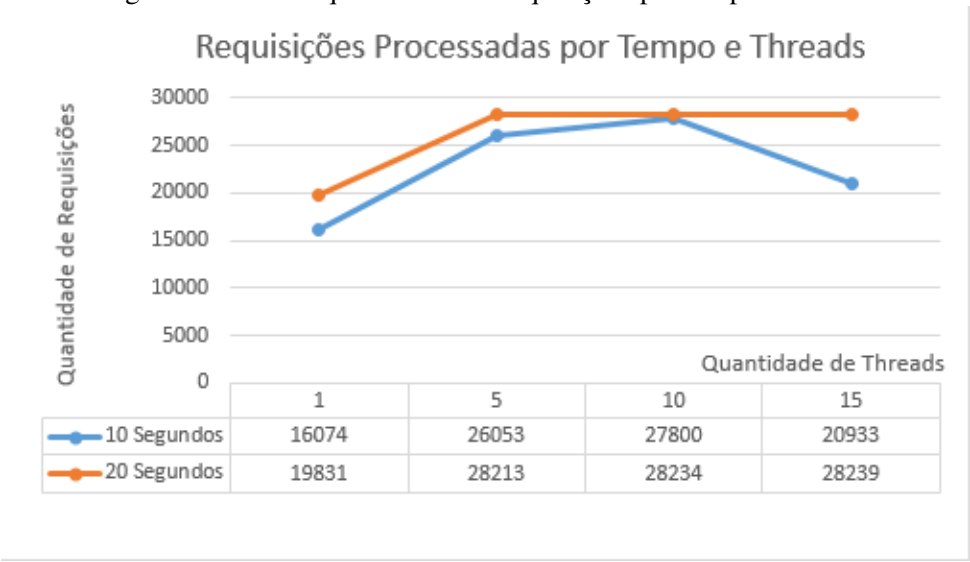
Neste teste foi utilizado a configuração padrão da vm, sendo disponibilizado 8 GB de memória RAM, os microsserviços, banco de dados foram executados a partir de imagens Docker rodando via compose. Docker Compose permite que mais de uma imagem Docker rode em simultaneo na mesma máquina/VM. Na tabela 3 mostra os resultados obtidos:

Tabela 3 – Resultados Obtidos no 1º Cenário de Avaliação

| Número de Threads | Tempo de Execução em segundos | Número de requisições | Requisições por segundo | KB/seg enviados | KB/seg recebidos |
|---|---|---|---|---|---|
| 1 | 10 | 16074 | 1607,4 | 916,72 | 59,65 |
| 5 | 10 | 26053 | 2605,3 | 1485,39 | 96,65 |
| 10 | 10 | 27800 | 2780 | 1584,83 | 103,12 |
| 15 | 10 | 20933 | 2093,3 | 1193,36 | 77,65 |
| 1 | 20 | 19831 | 991,55 | 565,49 | 36,80 |
| 5 | 20 | 28213 | 1410,65 | 804,78 | 354,34 |
| 10 | 20 | 28234 | 1411,7 | 1069,11 | 71,12 |
| 15 | 20 | 28239 | 1411,95 | 1143,54 | 76,91 |

Fonte: elaborado pelo autor.

Figura 6 – Gráfico quantidade de Requisições por tempo e threads




Fonte: elaborado pelo autor

Estes resultados mostram a quantidade de chamadas suportada pela arquitetura, verifica-se que o maior número de threads não resulta em uma significativa melhora nos resultados. Neste ambiente de teste existe uma limitação de hardware, o processamento é limitado em 6 threads, outra limitação é a de software, como a arquitetura foi testada em uma VM, o escalonador de

processos do Windows, pode em algum momento não ter priorizado os processos da VM, está limitações podem gerar variações nos resultados coletados.

Neste cenário de teste a melhor opção encontrada de número de threads é 10, está configuração sendo capaz de processar 27800 requisições 10 segundos, e em média processar entre 2780 requisições por segundo. Tento em vista o universo dos dispositivos IOT, se cada aparelho transmite uma chamada por segundo, está configuração facilmente suportaria mais de 2600 aparelhos simultaneamente.

Nota-se a partir do gráfico contido na imagem 6 que a maior quantidade de tempo e threads, o número total de chamadas não teve um significativo ganho, chegando no limiar de 28000 requisições.

## 5.3 Teste de stress por tempo de execução e número de threads - 12 GB de RAM

Neste teste é submetida a mesma bateira de testes executados na primeira avaliação, tendo como diferença a configuração de hardware disponibilizado para VM. O total de memoria RAM do sistema é 16 GB, neste teste 12 GB são disponibilizados para VM. Está mudança tem o intuito de verificar se diferentes configurações de hardware tem influencia no desempenho da arquitetura.

Tabela 4 – Resultados Obtidos no 2º Cenário de Avaliação

| Número de Threads | Tempo de Execução em segundos | Número de requisições | Requisições por segundo | KB/seg enviados | KB/seg recebidos |
|---|---|---|---|---|---|
| 1 | 10 | 10197 | 1019,7 | 581,72 | 37,85 |
| 5 | 10 | 22052 | 2205,2 | 1258,66 | 81,90 |
| 10 | 10 | 21778 | 2177,8 | 1240,66 | 80,73 |
| 15 | 10 | 21561 | 2156,1 | 1228,79 | 79,96 |
| 1 | 20 | 19844 | 992,2 | 565,84 | 36,82 |
| 5 | 20 | 28230 | 1411,5 | 959,82 | 63,15 |
| 10 | 20 | 28235 | 1411,75 | 1109,45 | 73,81 |
| 15 | 20 | 28240 | 1412 | 949,23 | 63,84 |

Fonte: elaborado pelo autor.

Verifica-se que nos resultados obtidos na tabela 4, não existe um ganho significativo no número máximo de requisições, em alguns momentos há inclusive perda de performance comparando com o primeiro cenário, devido as limitações de software já comentadas.

## 5.4 Comparativo de performance com duas instâncias do Kafka

Com o intuito de analisar o impacto do Kafka na arquitetura, este cenário de teste foi desenvolvido para verificar esse impacto. Para isso foi executada duas instancias do Kafka ligadas no mesmo Zookeeper.

Tabela 5 – Resultados Obtidos no 3º Cenário de Avaliação

| Número de Threads | Tempo de Execução em segundos | Número de requisições | Requisições por segundo | KB/seg enviados | KB/seg recebidos |
|---|---|---|---|---|---|
| 5 | 20 | 28229 | 1411,45 | 819,45 | 53,92 |
| 10 | 20 | 28234 | 1411,7 | 1117,81 | 74,36 |
| 15 | 20 | 28239 | 1411,95 | 1114,87 | 74,98 |

Fonte: elaborado pelo autor.

Através dos resultados obtidos na tabela 5 nota-se que as duas instancias do Kafka não trouxeram um diferença de performance. Gerando maior gasto de hardware sem ganho de performance. Neste cenário é visto que uma instância do Kafka é capaz de suprir a demanda da aplicação.

## 5.5 Comparativo de Performance Entre os Servidores de Aplicação

O Spring boot 2.0 suporta 3 diferentes servidores de aplicações embarcados no framework, sendo o servidor padrão desta versão o Jetty, além deste servidor de aplicação o Spring suporta: Tomcat e Undertow. Com o intuído de verificar qual o melhor servidor de aplicação para arquitetura, foi desenvolvido este cenário de avaliação, que consiste em editar todos os microsserviços desenvolvidos, para utilizar todos os servidores de aplicação disponíveis no Spring.

Para cada um dos servidores de aplicação foi rodada a mesma bateria de testes realizadas nas seções anteriores.

Tabela 6 – Resultados Obtidos no 4º Cenário de Avaliação

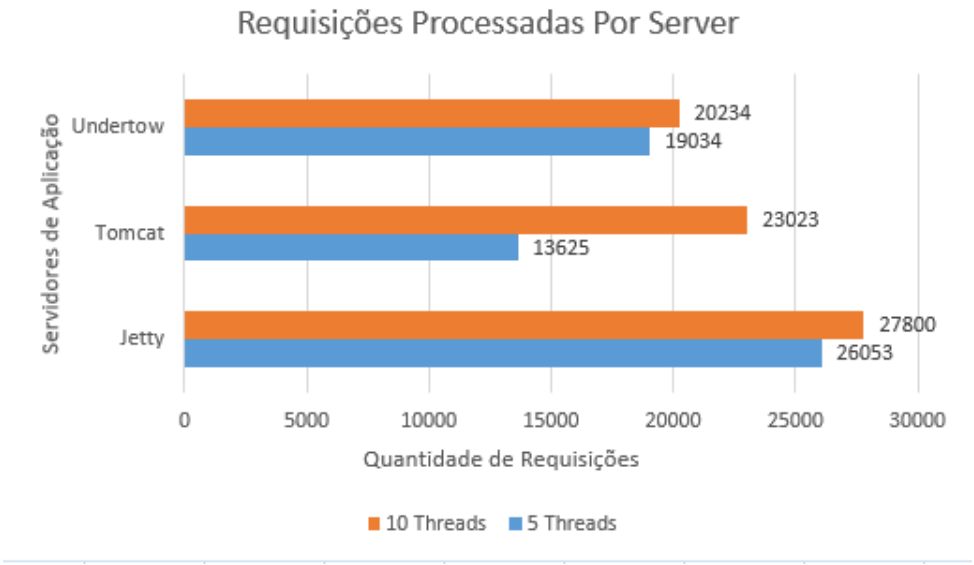
| Application Server | Threads | Número de requisições | Requisições por segundo |
|---|---|---|---|
| Jetty | 5 | 26053 | 2605,3 |
| Jetty | 10 | 27800 | 2780 |
| Tomcat | 5 | 13625 | 1362,5 |
| Tomcat | 10 | 23023 | 2302,3 |
| Undertow | 5 | 19034 | 1902,4 |
| Undertow | 10 | 20234 | 2023,4 |

Fonte: elaborado pelo autor.

A partir dos resultados obtidos é visto que o melhor servidor, sendo capaz de processar um maior número de requisições é o Jetty com 26053 chamadas com 5 threads e 27800 com 10 threads. Sendo capaz de processar 2780 requisições por segundo.

O Segundo melhor servidor de aplicação para está arquitetura é o Undertow, este é capaz de processar 19034 chamadas com 5 threads abertas em simultâneo, com 10 threads 20334

Figura 7 – Gráfico com a quantidade de requisições processadas




Fonte: elaborado pelo autor

chamadas tendo como média neste caso 2023 chamadas por segundo.

Em ultimo o Tomcat com 5 threads abertas em simultâneo foi capaz de responder 13625 requisições, mas com 10 threads foi capaz de responder 20234 tendo um desempenho melhor que o Undertow. O Tomcat está mais tempo no mercado, sendo utilizado por grandes companhias, além de que é provido por empresas um suporte. Por estes motivos que grandes empresas acabam por escolher o tomcat.

A partir destes resultados foi escolhido o servidor de aplicação Jetty para arquitetura, sendo que é capaz de processar mais requisições comparados com os outros estudados.

## 5.6 Pesquisa dissertativa com desenvolvedores

Para verificar a aceitação da arquitetura por desenvolvedores de software, foi feita uma pesquisa contendo 4 perguntas sobre a arquitetura proposta.

A pesquisa foi realiza por 4 funcionários da empresa Agibank localizada em Porto Alegre. Os desenvolvedores que responderam essa pesquisa tem em média 27 anos de idade entre 3 há 7 anos de experiencia com programação, sendo todos residentes da cidade Porto Alegre, ainda em torno 50% são graduados e os demais cursando a graduação. Cursos de Ciência da Computação, Sistemas da Informação e Engenharia de Software. Os entrevistados tem experiencia com microsserviços, trabalhando diariamente com as tecnólogas utilizadas na arquitetura.

Abaixo o questionário utilizado na coleta de dados:

**1)** Comente sobre as facilidades de implementar a arquitetura proposta?

**2)** Comente sobre as dificuldades de implementar a arquitetura proposta?

**3)** Quais sugestões de melhoria que você tem em relação arquitetura proposta?

**4)** Qual sua percepção da arquitetura proposta em relação as tecnologias atuais?

Como os desenvolvedores entrevistados optaram por ficar anônimos na pesquisa, as respostas de cada um entrevistado será identificada por letras entre (a - d)

### 5.6.1 Respostas entrevistado a:

1. A arquitetura parece ser fácil de implementar, tendo essa utilização de microsserviços, esses podem ser rodados a partir de docker.
2. A dificuldade maior seria a instalação dos bancos de dados e configurações manuais para instalação destes.
3. A utilização de scripts de implantação para plataformas cloud, como a Amazon WS.
4. A arquitetura utiliza tecnologias bem atuais, como Spring Webflux, suportando chamadas reativas de serviço. Outro ponto positivo é utilização de banco não relacional.

### 5.6.2 Respostas entrevistado b:

1. Muito simples de utilizar a arquitetura, só fazer chamadas de serviço para aproveitar a capacidade de extrair informações.
2. Acredito que poderia ter um frontEnd para o cadastro dos monitores.
3. Para a quantidade de chamadas suportadas, pode-se trocar o Kafka por alguma ferramenta mais leve para envio de mensagens. Essa capacidade computacional pode ser utilizada para supir mais uma instancia do Microsserviço que recebe as mensagens e manda pra fila, está funcionalidade acaba tendo mais carga que as outras.
4. Utiliza tecnologias atuais com o suporte a chamadas reativas e containers de aplicação.

### 5.6.3 Respostas entrevistado c:

1. Arquitetura boa para utilização em qualquer dispositivo, o payload é bem pequeno utilizando pucos dados transmitidos por rede.
2. Não vejo muita importância nas dificuldades encontradas ao ponto que possam ser relacionadas.
3. Processar via streamming, no microsserviço Data-Monitor em vez de transformar a mensagem em Json, utilizando ferramentas como Apache Spark ou Kafka Streams para o processamento em tempo real destes fluxos.

4. Acho bem relevante as tecnologias utilizadas, penso que essas poderiam ser bem empregadas nas corporações e os microsserviços estão em alta no mercado.

### 5.6.4 Respostas entrevistado d:

1. A Arquitetura é um pouco complexa, utilizando diversos frameworks, mas em compensação tudo pode ser instalada via containers com isso facilitaria muito o trabalho do desenvolvedor. API bem facil de utilizar.
2. Dificuldade na instalação dos componentes.
3. Utilização de ferramentas automáticas para documentar as APIs disponíveis.
4. Boa proposta de arquitetura, podemos notar a utilização de tecnologias modernas, e isso proporciona alta capacidade de processamento e baixa utilização de hardware, sendo uma opção interessante para a proposta de suportar IOT.

### 5.6.5 Analise das Respostas

Verifica-se que com esta pesquisa que a arquitetura utiliza-se de tecnologias atuais perante aos entrevistados e que seria fácil de implementa-la para utilização de chamadas de serviço e utilização de containers. Já como pontos de melhorias, foram levantados a criação de fontend, documentação da API, implantação da arquitetura na Cloud e o uso de frameworks para o processamento de fluxos. Em relação as principais dificuldades encontradas pelos desenvolvedores, foram relacionadas a complexidade de instalação de todos os componentes e banco de dados.

Estes dados coletados são fundamentais para evolução da arquitetura, elencando os principais prioridades para as próximas versões e atualizações, tais melhorias irão deixar a arquitetura mais completa para o processamento em tempo real, facilitando a implantação pelo uso de scripts e provisionamento na cloud.

# 6 CONCLUSÃO

Este estudo desenvolveu modelo de arquitetura de software e um sistema conceitual para suportar dispositivos IoT, utilizando-se dos conceitos de microsserviços para possibilitar o desenvolvimento de uma arquitetura que consiga lidar com diversos tipos de dados e grande quantidade de requisições.

A avaliação realizada mostrou que a arquitetura proposta é capaz de lidar com grande quantidade de requisições, além de conseguir extrair Informações e gerar ações, mesmo utilizando uma configuração modesta de hardware. A arquitetura teve ótima aceitação entre os desenvolvedores entrevistados com experiência em microsserviços, sendo como principal fonte de sucesso o emprego de tecnologias atuais.

Como trabalhos futuros sugere-se a atualização da arquitetura para suportar o processamento de streams para realizar consultas em tempo real, criando assim um novo microsserviço para substituir o data-Monitor e o desenvolvimento de um frontend para analisar os dados recebidos e criar atualizar os monitores. Outro hipótese de trabalho futuro é o emprego arquitetura proposta em um cenário real de utilização, como extrair informações de sensores em lavouras, assim sendo necessário implantar a arquitetura em um ambiente com maior disponibilidade de hardware, preferencialmente em nuvem.

# REFERÊNCIAS

AL-FUQAHA, A. et al. Internet of things: a survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys Tutorials**, [S.l.], v. 17, n. 4, p. 2347–2376, Fourthquarter 2015.

AMARAL, M. et al. Performance evaluation of microservices architectures using containers. **Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015**, [S.l.], p. 27–34, 2015.

BAK, P. et al. Location and context-based microservices for mobile and internet of things workloads. In: IEEE INTERNATIONAL CONFERENCE ON MOBILE SERVICES, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 1–8.

BUTZIN, B.; GOLATOWSKI, F.; TIMMERMANN, D. Microservices approach for the internet of things. In: IEEE 21ST INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 1–6.

CHOU, D. **Using events in highly distributed architectures**. Disponível em: <https://msdn.microsoft.com/en-us/library/dd129913.aspx>. Acesso em: 21 novembro 2017.

DUNKEL, J. et al. Expert Systems with Applications Event-driven architecture for decision support in traffic management systems. **Expert Systems**, [S.l.], p. 7–13, 2010.

ERL, T. **Service-oriented architecture (paperback)**: concepts, technology, and design (the prentice hall service technology series from thomas erl). [S.l.]: Prentice Hall, 2016.

ERL, T. et al. **Soa with java**: realizing service-orientation with java technologies (the prentice hall service technology series from thomas erl). [S.l.]: Prentice Hall, 2014.

FILHO, P.; PADUA, W. de. **Engenharia de software** : fundamentos, métodos e padrões. terceira. ed. Rio de Janeiro: LTC, 2009.

FOWLER, M.; LEWIS, J. **Microservices**. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 21 novembro 2017.

GARTNER. **Gartner says 8.4 billion connected**. Disponível em: <https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016>. Acesso em: 20 outubro 2018.

KAFKA. **Introduction**. Disponível em: <https://kafka.apache.org/intro>. Acesso em: 22 novembro 2017.

KRYLOVSKIY, A.; JAHN, M.; PATTI, E. Designing a smart city internet of things platform with microservice architecture. In: INTERNATIONAL CONFERENCE ON FUTURE INTERNET OF THINGS AND CLOUD, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 25–30.

KUMARI, S.; RATH, S. K. Performance comparison of soap and rest based web services for enterprise application integration. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 1656–1660.

MEMON, M. H. et al. Internet of things (iot) enabled smart animal farm. In: INTERNATIONAL CONFERENCE ON COMPUTING FOR SUSTAINABLE GLOBAL DEVELOPMENT (INDIACOM), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 2067–2072.

MICROSOFT. **Microsoft application architecture guide, 2nd edition**. Disponível em: <https://msdn.microsoft.com/en-us/library/ff650706.aspx>. Acesso em: 02 novembro 2017.

NARKHEDE, N.; SHAPIRA, G.; PALINO, T. **Kafka**: the definitive guide: real-time data and stream processing at scale. [S.l.]: O'Reilly Media, 2017.

NEWMAN, S. **Building microservices**: designing fine-grained systems. [S.l.]: O'Reilly Media, 2015.

O'CONNOR, R. V.; ELGER, P.; CLARKE, P. M. Continuous software engineering—A microservices architecture perspective. **Journal of Software: Evolution and Process**, [S.l.], v. 29, n. 11, p. 1–12, 2017.

SHANAHAN, J. G.; DAI, L. Large scale distributed data science using apache spark. In: ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, 21., 2015, New York, NY, USA. **Proceedings...** ACM, 2015. p. 2323–2324. (KDD '15).

SILVEIRA, P. **Introdução à arquitetura e design de software** : uma visão sobre a plataforma java. Rio de Janeiro, Brasil: Elsevier, 2012.

SPARK. **Apache spark**. Disponível em: <https://spark.apache.org/>. Acesso em: 22 novembro 2017.

TANEJA, M. et al. Fog assisted application support for animal behaviour analysis and health monitoring in dairy farming. In: IEEE 4TH WORLD FORUM ON INTERNET OF THINGS (WF-IOT), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p. 819–824.

VILLAMIZAR, M. et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. **10th Computing Colombian Conference**, [S.l.], p. 583–590, 2015.