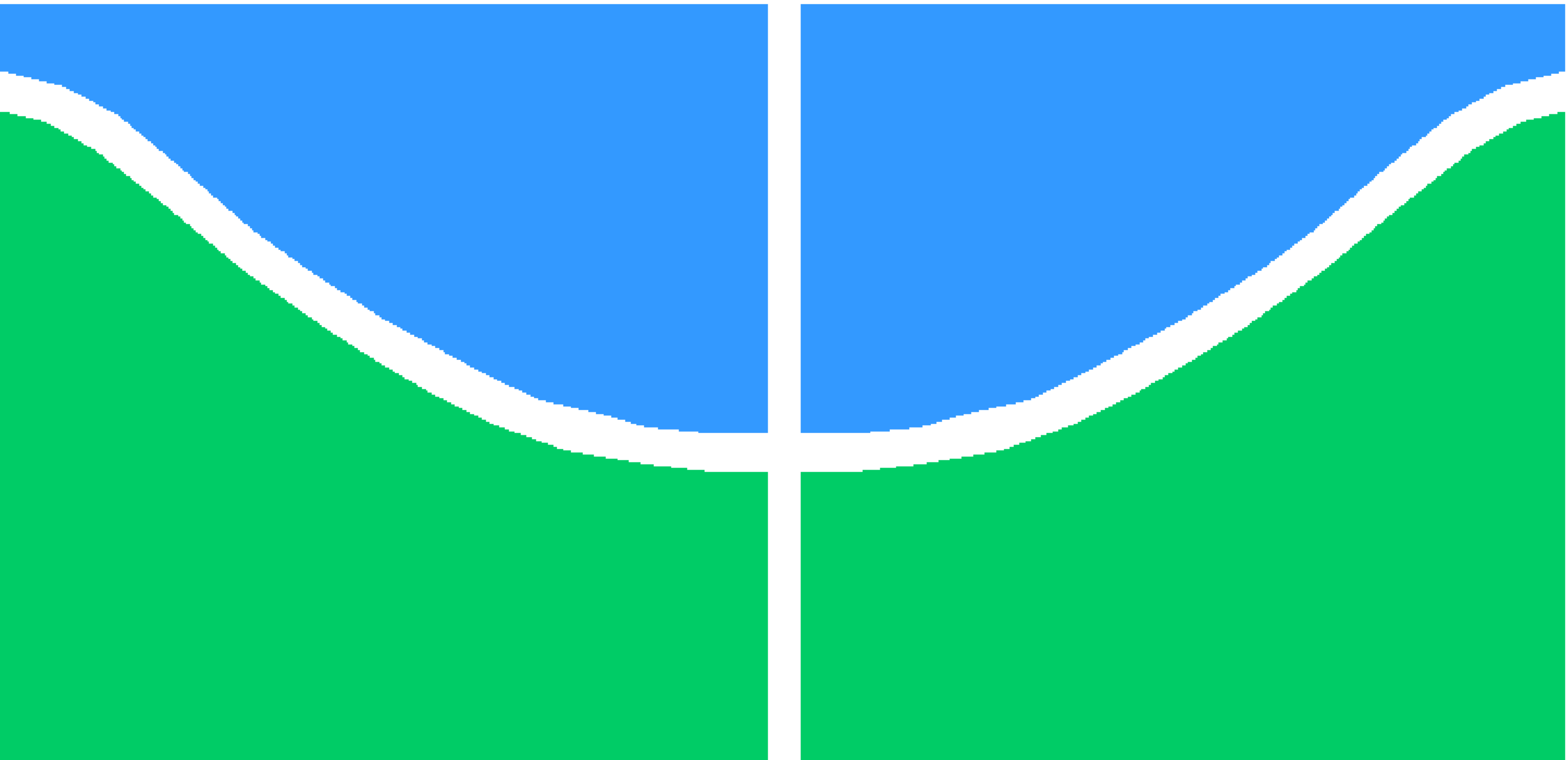

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Engenharia de Software

Processamento de dados em uma plataforma de cidades inteligentes

Autor: Dylan Jefferson Maurício Guimarães Guedes

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Brasília, DF

2017

Dylan Jefferson Maurício Guimarães Guedes

Processamento de dados em uma plataforma de cidades inteligentes

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Coorientador: Arthur de Moura Del Esposte

Brasília, DF

2017

Dylan Jefferson Maurício Guimarães Guedes

Processamento de dados em uma plataforma de cidades inteligentes/ Dylan Jefferson Maurício Guimarães Guedes. – Brasília, DF, 2017-

47 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Paulo Roberto Miranda Meirelles

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2017.

1. Cidades Inteligentes. 2. Big Data. I. Prof. Dr. Paulo Roberto Miranda Meirelles. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Processamento de dados em uma plataforma de cidades inteligentes

CDU 02:141:005.6

Dylan Jefferson Maurício Guimarães Guedes

Processamento de dados em uma plataforma de cidades inteligentes

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 18 de maio de 2017:

Prof. Dr. Paulo Roberto Miranda Meirelles
Orientador

Profa. Dra. Carla Silva Rocha Aguiar
Convidado 1

Prof. Dr. Fabio Kon
Convidado 2

Brasília, DF
2017

Resumo

O InterSCity é uma plataforma de cidades inteligentes baseado em uma arquitetura de microsserviços, e tem como objetivo suportar aplicações de cidades inteligentes oferecendo um conjunto de serviços reutilizáveis, interoperáveis e escaláveis. Contudo, o uso de ferramentas adequadas no processamento de seus dados ainda não se faz presente, sendo um obstáculo em cenários de maior massa de dados. Este trabalho tem como objetivos o projeto e desenvolvimento de um serviço de processamento de dados que permita ao InterSCity oferecer novos serviços a partir do uso de grande massas de dados, utilizando principalmente *Big Data*, tecnologia chave para cidades inteligentes. A partir da adoção do estado da arte em arquiteturas de processamento de dados em conjunto com tecnologias atuais de *Big Data*, esperamos que o InterSCity consiga suportar o desenvolvimento de aplicações mais sofisticadas para cidades inteligentes.

Palavras-chaves: Cidades Inteligentes, Big Data, Arquitetura Kappa.

Abstract

InterSCity is a smart cities platform based on a microservices architecture that aims at supporting smart cities applications, offering a set of reusable, interoperable, and scalable services. However, InterSCity does not use suitable tools to process its data, an obstacle in scenarios of larger data set. This work aims to design and to implement a data processing service that allows InterSCity to handle larger data set, using mainly Big Data, a key technology for smart cities. With the adoption of state-of-the-art data processing architectures and new Big Data technologies, we expect InterSCity to be able to support more sophisticated applications for smart cities.

Key-words: Smart Cities, Big Data, Kappa Architecture.

Lista de ilustrações

Figura 1 – Ciclo de vida de um recurso IoT no InterSCity. Baseado em: Esposte et al. (2017). 20
Figura 2 – Arquitetura completa do InterSCity. Fonte: Esposte et al. (2017). 21
Figura 3 – Ciclo de vida na Arquitetura Lambda. Baseado em: Marz e Warren (2015). 24
Figura 4 – Funcionamento da Arquitetura Kappa. Baseado em: Seyvet (2016). 26
Figura 5 – Funcionamento do *broker*. 27
Figura 6 – Pilha de tecnologias utilizadas - Apache Kafka e Apache Spark, e suas interações com o InterSCity. 31
Figura 7 – Novo ciclo de vida da plataforma, com relação ao novo serviço de processamento. 32
Figura 8 – Ciclo de vida do Shock dentro do InterSCity. 34

Lista de tabelas

Tabela 1 – Resultados da Sort Benchmark 2014, categoria GraySort 28
Tabela 2 – Cronograma com o planejamento das dívidas técnicas para o TCC 2. . 36
Tabela 3 – Atividades para serem feitas após o TCC. 37

Lista de abreviaturas e siglas

IoT	*Internet of Things*
MPLv2	Mozilla Public License Version 2.0
MSA	*Microservices Architecture*
TIC	Tecnologias da Informação e Comunicação

Sumário

1 INTRODUÇÃO . . . 17

2 INTERSCITY . . . 19
2.1 ARQUITETURA . . . 19
2.2 GERÊNCIA DE CONFIGURAÇÃO E DEPENDÊNCIAS . . . 21
2.3 PROPOSTA E METODOLOGIA PARA IMPLEMENTAÇÃO DO SERVIÇO DE PROCESSAMENTO DE DADOS . . . 22

3 PROCESSAMENTO DE DADOS . . . 23
3.1 ARQUITETURA LAMBDA . . . 23
3.2 ARQUITETURA KAPPA . . . 25
3.3 BROKER . . . 26
3.4 COMPARATIVO ENTRE TECNOLOGIAS . . . 27
3.4.1 Ferramentas de Processamento Batch . . . 27
3.4.2 Ferramentas de Processamento Streaming . . . 28
3.4.3 Broker . . . 29

4 PROJETO E IMPLEMENTAÇÃO DO SERVIÇO DE PROCESSAMENTO . . . 31
4.1 IMPLEMENTAÇÃO . . . 32
4.2 SHOCK . . . 33

5 CONSIDERAÇÕES PRELIMINARES . . . 35

REFERÊNCIAS . . . 39

APÊNDICES 41

APÊNDICE A – PRINCÍPIOS SEGUIDOS PELO INTERSCITY . . . 43

APÊNDICE B – PSEUDO-IMPLEMENTAÇÃO - ARQUITETURA LAMBDA . . . 45
B.1 CÓDIGO DA CAMADA BATCH . . . 45
B.2 CÓDIGO DA CAMADA SPEED . . . 45
B.3 CÓDIGO DA CAMADA SERVING . . . 47

1 INTRODUÇÃO

O termo **cidades inteligentes** recebe cada vez mais atenção, e trata-se da utilização de tecnologias da informação e comunicação (TIC) para melhorar setores como segurança, transporte e saúde, aumentando a qualidade de vida da população (BATTY et al., 2012). As cidades inteligentes ganham força por atingirem soluções e mitigações concretas para problemas graves e recorrentes das cidades atuais, como o mau uso de recursos, a burocracia, transporte de má qualidade e falta de segurança (BATTY et al., 2012). Para ajudar no desenvolvimento de aplicações de cidades inteligentes são desenvolvidas plataformas que oferecem diversos requisitos funcionais e não-funcionais, facilitando assim o desenvolvimento de novas soluções (KON; SANTANA, 2016).

Diversas iniciativas de cidades inteligentes ocorrem atualmente. Em Santander, na Espanha, foi desenvolvida a plataforma SmartSantander[1], e utilizando-a, diversos aplicativos foram criados, como para apresentar informações diversas da cidade (sobre tráfego, temperatura, transporte público), ou para informar lugares livres para estacionar[2] (GUTIÉRREZ et al., 2013). Em Amsterdã, na Holanda, a plataforma Amsterdam Smart City[3] disponibiliza serviços para aplicações de cidades inteligentes. Apesar de existirem soluções e propostas, vários desafios técnicos em plataformas de cidades inteligentes ainda persistem. As soluções atuais costumam ser específicas, não promovendo interoperabilidade entre as ferramentas e não promovendo reuso dos projetos já desenvolvidos em outros contextos (ESPOSTE et al., 2017).

Com a finalidade de ser uma plataforma que em sua origem se atente aos problemas de interoperabilidade citados, surge o InterSCity, que visa suportar o desenvolvimento de novas aplicações, projetos e serviços em cidades inteligentes. A arquitetura da plataforma é baseada em microsserviços, e tem como foco a interoperabilidade, a padronização, a escalabilidade e a extensibilidade. O InterSCity está em desenvolvimento, e embora já tenha uma arquitetura bem definida e conte com diversas funcionalidades, ainda não dispõe de um serviço de processamento de dados adequado para contextos de larga escala, comum em cenários de cidades inteligentes (NUAIMI et al., 2015).

Este trabalho tem como principal contribuição o planejamento, desenho e implementação de um novo serviço de processamento de dados para o InterSCity, permitindo assim seu uso em cenários de grande massa de dados, e possibilitando a criação de um *pipeline de dados* sofisticado e customizável. Dessa forma, desenvolvemos o Shock, uma aplicação que abstrai o uso de tecnologias de *Big Data*, e que foi construído utilizando a

1 <www.smartsantander.eu/>
2 <www.smartsantander.eu/wiki/index.php/Mitos/Mitos>
3 <https://amsterdamsmartcity.com>

Arquitetura Kappa. É esperado também que o estudo que desenvolvemos e as implementações feitas sejam úteis para futuras plataformas de cidades inteligentes.

Apresentamos no Capítulo 2 maiores detalhes sobre as características e o estado atual do InterSCity, trazendo ainda a abordagem que utilizamos para definir o novo serviço de processamento. No Capítulo 3 trazemos um estudo sobre o estado da arte em arquiteturas e tecnologias de *Big Data*, bem como uma análise de diferentes ferramentas adequadas para o contexto da plataforma. No Capítulo 4 levantamos as decisões, justificativas e resultados atingidos quanto ao novo serviço de processamento, e por fim, no Capítulo 5, finalizamos o trabalho trazendo as considerações preliminares e os próximos passos que tomaremos.

2 INTERSCITY

O InterSCity é uma plataforma de cidades inteligentes nascida a partir de estudos científicos que buscam abordar os principais desafios encontrados no desenvolvimento de infraestruturas de cidades inteligentes (BATISTA et al., 2016). Está licenciado sob MPLv2[1] (Mozilla Public License Version 2.0), foi construído com a utilização da arquitetura MSA[2] (Microservices Architecture), e tem como principal objetivo prover os serviços e integrações necessárias para a construção de aplicações de cidades inteligentes complexas (ESPOSTE et al., 2017). Baseando-se no desenvolvimento colaborativo e na utilização de tecnologias software livre, o projeto é desenvolvido com a ajuda de diversos colaboradores, que utilizando práticas ágeis, atuam na manutenção e evolução da plataforma ao longo do tempo (ESPOSTE et al., 2017).

A maior parte dos microsserviços[3] da plataforma foram escritos em Ruby on Rails[4], seguindo padrões que priorizam extensibilidade e qualidade, e levando em conta princípios[5] que levaram a uma arquitetura madura, robusta e extensível. A partir de experimentos feitos foi possível conferir o quão promissor é a performance e a escalabilidade do InterSCity, provando-se capaz de lidar com cenários reais de cidades inteligentes (ESPOSTE et al., 2017). O projeto encontra-se hospedado no Gitlab[6], onde é possível ter acesso ao código fonte e documentação, bem como um exemplo de cliente que ilustra o uso da plataforma.

2.1 ARQUITETURA

O InterSCity é composto por uma arquitetura de microsserviços distribuída que possibilita armazenamento, análise, processamento, composição e integração de dados de recursos de cidades inteligentes (ESPOSTE et al., 2017). Esses microsserviços são desacoplados entre si, e se comunicam através de requisições REST (Representational State Transfer) e passagem de mensagem, modelo importante em contextos de concorrência por conta do isolamento provido (ARMSTRONG, 2003). No InterSCity a troca de mensagem é feita com auxílio do *broker* de mensagens RabbitMQ[7], através do padrão de projeto

1 <www.mozilla.org/en-US/MPL/2.0/>
2 <microservices.io/>
3 Os termos microsserviços, módulos, e componentes, serão utilizados intermitentemente, mas apresentando o mesmo significado.
4 <rubyonrails.org/>
5 Os princípios seguidos pelo InterSCity são apresentados no Apêndice A
6 <gitlab.com/smart-city-software-platform>
7 <www.rabbitmq.com/>

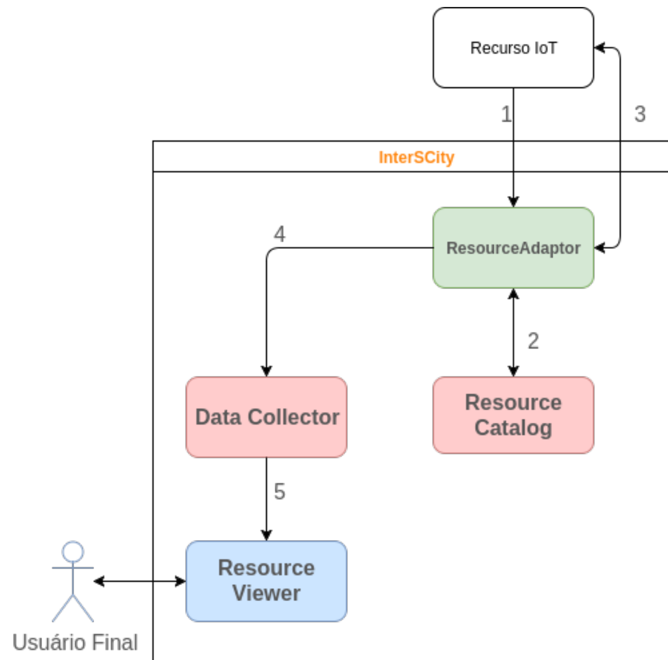

Figura 1 – Ciclo de vida de um recurso IoT no InterSCity. Baseado em: Esposte et al. (2017).

PubSub[8].

A Figura 1 ilustra o ciclo de vida típico de um recurso IoT (*Internet of Things*) na plataforma. Inicialmente um recurso IoT (1) faz um pedido de registro na plataforma via Resource Adaptor, que (2) cadastra o recurso no microsserviço Resource Catalog (3) e informa o UUID (identificador único) que será utilizado internamente desse passo em diante. Após, a comunicação entre o Resource Adaptor e o dispositivo IoT terá continuidade, mas (4) os dados terão como destino o módulo Data Collector, que armazenará as informações. Por fim, (5) as informações contidas no Data Collector são disponibilizadas, podendo ser apresentadas para um usuário final via Resource Viewer, ou consumidas por uma aplicação cliente.

Os microsserviços do InterSCity têm responsabilidades atômicas e bem definidas, princípio chave para que a plataforma contemple requisitos funcionais e não-funcionais. O microsserviço **Resource Adaptor** é o grande responsável pela comunicação entre os dispositivos IoT e a plataforma, funcionando como um mediador durante as requisições (ESPOSTE et al., 2017).

O **Data Collector** e o **Resource Catalog** tem papéis parecidos, mas enquanto o primeiro gerencia e armazena dados históricos de medições dos dispositivos, o segundo tem o papel de gerenciar e armazenar o registro dos dispositivos na plataforma (ESPOSTE et al., 2017).

[8] <xmpp.org/extensions/xep-0060.html>

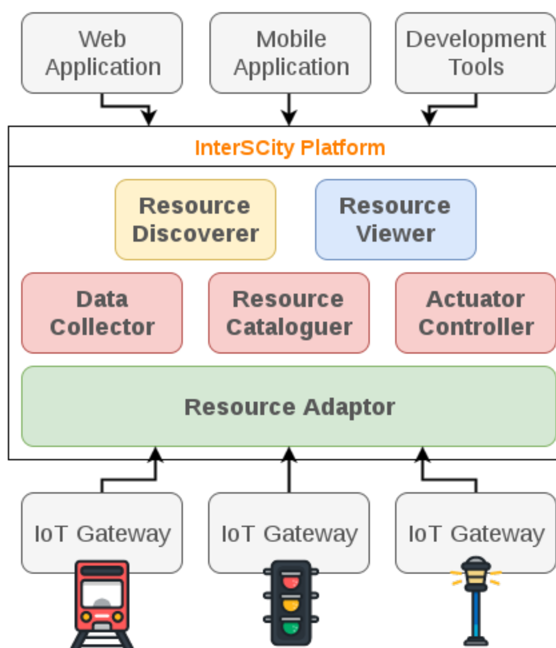

Figura 2 – Arquitetura completa do InterSCity. Fonte: Esposte et al. (2017).

O **Resource Viewer** e o **Resource Discovery**, por outro lado, são similares por manipularem e utilizarem o Data Collector e o Resource Catalog em sua execução. O Resource Viewer tem como objetivo apresentar ao usuário final os dados dos recursos, enquanto o Resource Discovery provê uma API de busca por dispositivos disponíveis, possibilitando o uso de filtros (ESPOSTE et al., 2017). Por fim, o **Actuator Controller** provê serviços para requisições nos recursos IoT atuadores registrados na plataforma, armazenando os dados e possibilitando auditoria (ESPOSTE et al., 2017). A Figura 2 trás uma visão geral da arquitetura, com todos os microsserviços reunidos, apresentando as fronteiras entre a plataforma, aplicações clientes e dispositivos.

2.2 GERÊNCIA DE CONFIGURAÇÃO E DEPENDÊNCIAS

Outro aspecto que recebe atenção no desenvolvimento do InterSCity é a gerência de configuração, que atualmente utiliza tecnologias que reduzem o esforço, promovem o isolamento entre a plataforma e o ambiente hospedeiro e aumentam a segurança no desenvolvimento. A gerência de configuração do InterSCity é guiada por contêineres do Docker[9], e cada microsserviço e dependência externa (como o RabbitMQ) são executados em contêineres separados. O desenvolvimento da plataforma também faz extenso uso do Git[10], de modo que a configuração de um ambiente para execução do InterSCity tenha como pré-requisitos somente estes dois projetos: Docker e Git.

[9] <https://www.docker.com/>
[10] <https://git-scm.com/>

Cada microsserviço da plataforma tem seu próprio repositório, permitindo que evoluam de maneira distribuída. Com o propósito de ajudar no desenvolvimento, o InterSCity apresenta um repositório principal chamado *dev-env*[11], que funciona como *repositório mestre*, sendo os repositórios dos microsserviços da plataforma submódulos pertencentes ao *dev-env*.

2.3 PROPOSTA E METODOLOGIA PARA IMPLEMENTAÇÃO DO SERVIÇO DE PROCESSAMENTO DE DADOS

Atualmente a plataforma está em desenvolvimento, e embora não conte com uma camada de processamento de dados ideal, certo esforço pelo time do InterSCity culminou em um serviço de processamento provisório, que pode servir de base para o desenvolvimento da proposta deste trabalho. Houve ainda a troca de tecnologia de banco de dados no microsserviço Data Collector, que passou do Postgres (tecnologia SQL) para o MongoDB (tecnologia NoSQL), troca importante na busca por uma maior elasticidade no volume de dados. A equipe do InterSCity nomeou a camada provisória de **Data Processor**, que conta com uma configuração pronta para uso do Apache Spark e *scripts* que ilustram situações de uso desta tecnologia. O Data Processor, contudo, apresenta uma solução bem específica, não podendo ser reaproveitado por outras aplicações na plataforma.

Este trabalho deve então produzir um novo serviço de processamento de dados que permita às aplicações de cidades inteligentes que utilizam o InterSCity processar e analisar seus dados de maneira eficaz. Através do novo serviço, grande volume de dados poderá ser processado pela plataforma, possibilitando ainda que algoritmos e operações complexas sejam aplicadas. O novo serviço fará uso de um padrão de projeto de *Big Data* adequado para o contexto de cidades inteligentes e compatível com a arquitetura atual do InterSCity. Um levantamento das arquiteturas de *Big Data* candidatas deve ser feito, assim como um estudo a respeito de possíveis ferramentas a serem utilizadas. Esse levantamento terá algumas restrições, como: somente projetos software livre deverão ser levados em conta, e ferramentas que necessitem grandes mudanças no ecossistema do InterSCity terão pouca prioridade.

É esperado uma nova arquitetura de processamento de dados que atenda requisitos típicos de cidades inteligentes, e que possibilite extensibilidade para trabalhos futuros. A nova arquitetura deve permitir, por exemplo, que um *pipeline de dados* possa ser utilizado, mesmo que faça uso de uma grande massa de dados, ou que uma aplicação processe seus dados através de operações específicas e customizadas, por chamadas diversas. Por fim, deverá ser fornecido um exemplo de aplicação que utilize a arquitetura desenvolvida, como prova de conceito.

[11] <https://gitlab.com/smart-city-software-platform/dev-env>

3 PROCESSAMENTO DE DADOS

A grande quantidade de geradores e consumidores de dados em cenários de cidades inteligentes trazem a necessidade dos *três V's* para as aplicações: *volume*, *velocidade* e *variedade* (NUAIMI et al., 2015). Uma forma de sanar estes pontos é através do uso de tecnologias de *Big Data*, que podem ser utilizadas para armazenar, processar e analisar os dados das aplicações de cidades inteligentes (NUAIMI et al., 2015). Como relatado, o InterSCity carece de um serviço de processamento de dados mais adequado, e a partir dos estudos que apresentamos neste capítulo, tomamos decisões a respeito do novo serviço de processamento.

Levantamos duas arquiteturas, Lambda e Kappa, e tecnologias a serem usadas que abrangem duas formas de processamento: *batch* e *streaming*. No processamento *batch* os dados são utilizados em conjunto, e armazenados de uma forma específica antes de serem escalonados para o processamento (ZHENG et al., 2015). No processamento *streaming*, por outro lado, os dados são processados conforme estão disponíveis (ZHENG et al., 2015), permitindo consulta a informações com menor latência. Descrevemos adaptações nas arquiteturas em relação ao apresentado na literatura, com o propósito de maior adequação ao contexto de cidades inteligentes, e maior compatibilidade com os princípios e arquitetura de microsserviços do InterSCity.

3.1 ARQUITETURA LAMBDA

A Arquitetura Lambda é um padrão de projeto para plataformas de processamento de dados que utilizam tecnologias de *Big Data* (KIRAN; MURPHY; BAVEJA, 2015), e surge como um caminho alternativo a outras arquiteturas mais antigas, como a incremental com *sharding* (MARZ; WARREN, 2015). É composta de três camadas: a camada *batch*[1], a camada *serving* e a camada *speed* (KIRAN; MURPHY; BAVEJA, 2015). Cada uma destas camadas é implementada utilizando algoritmos e ferramentas específicas, de modo que certas ferramentas são mais apropriadas em certos contextos.

A **camada *batch*** é responsável pelo processamento de uma grande massa de dados, e tem como ponto fraco a alta latência. Em sua execução, ela cria e gerencia um *master dataset*[2], que, após processado, têm seus resultados condensados em *batch views*, utilizados pela **camada *serving***. A camada *batch* é então, em sua essência, imutável, de

[1] No decorrer do texto, as camadas da Arquitetura Lambda e outros termos associados não serão traduzidos.

[2] O *master dataset* é um lote histórico de informação, que, por ser imutável, só possibilita **anexação** de informações.

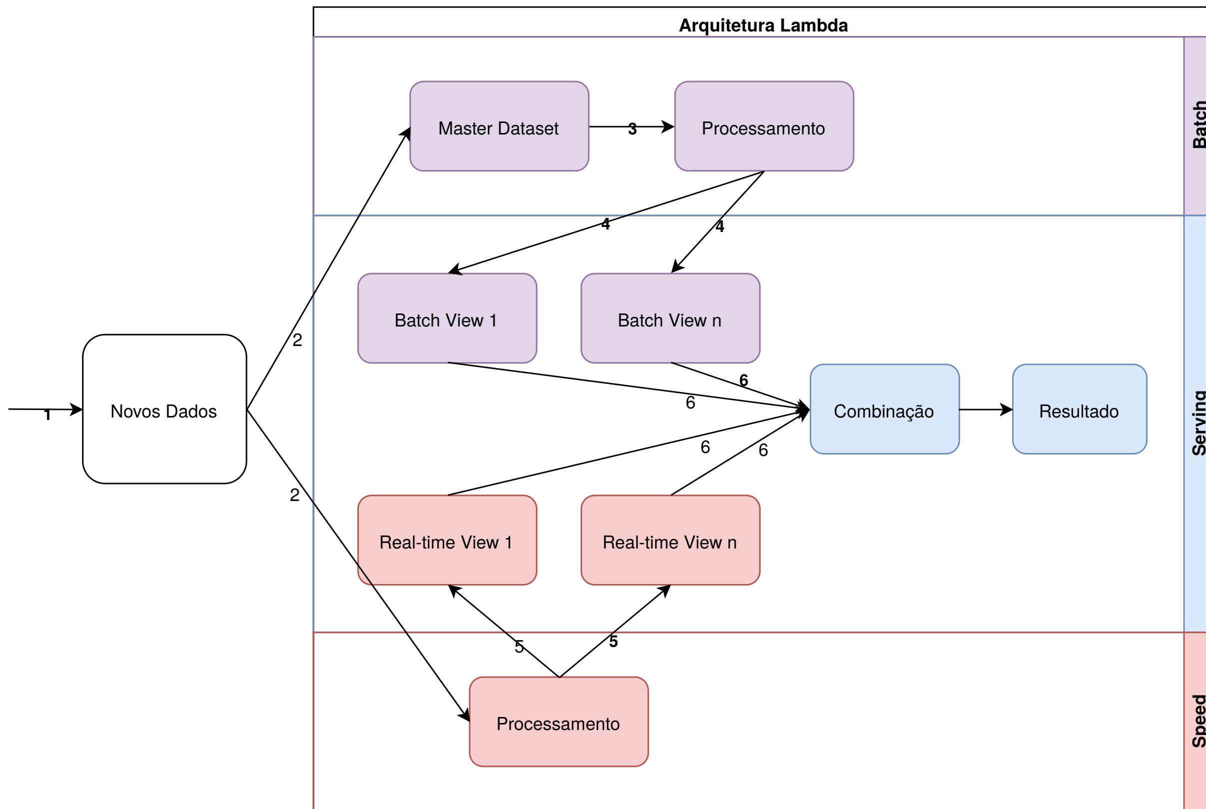

Figura 3 – Ciclo de vida na Arquitetura Lambda. Baseado em: Marz e Warren (2015).

modo que caso uma mudança seja necessária, uma abordagem diferente deve ser seguida: o dado que carece alteração não sofre transformações, permanecendo inalterado, mas um novo dado com as alterações é inserido no lote (MARZ; WARREN, 2015).

A **camada *speed*** tem como diferencial o processamento com baixa latência, que é obtido pelo uso de uma parcela menor da massa de dados[3]. Outra característica importante é que esta camada faz uso do processamento *streaming*, que ocorre quando os dados são processados conforme estão disponíveis. Este tipo de processamento funciona bem com mecanismos de passagem de mensagem (MARZ; WARREN, 2015), e permite que as consultas feitas levem em conta dados recentes, ignorados temporariamente pela camada *batch*. Por estes motivos, a camada *speed* também é conhecida como a camada que faz *processamento incremental* (MARZ; WARREN, 2015), e que por aceitar a mutação de dados, força o uso de um banco de dados que suporte escrita aleatória, aumentando substancialmente a complexidade desta camada (MARZ; WARREN, 2015). Por fim, a camada *speed* condensa os resultados de seu processamento em *real-time views*, que serão fundidos com os resultados das *batch views* na apresentação do resultado final. Os resultados da camada *speed* são então dispensados sempre que um *batch* terminar seu processamento (MARZ; WARREN, 2015).

[3] A camada *speed* só leva em conta dados que surgiram após a camada *batch* ter começado seu processamento.

Um ciclo de vida típico da Arquitetura Lambda pode ser acompanhado na Figura 3, e tem seu início com a (1) chegada de novos dados, que são (2) transmitidos tanto para a camada *batch* quanto para a camada *speed*. A camada *batch* (3) anexa os novos dados e os processa, (4) gerando assim *batch views* que são enviadas para a camada *serving*. O mesmo dado que foi enviado para a camada *batch*, no passo (1), também foi enviado para a camada *speed*, onde (5) será processado com menor latência, por só levar em conta dados recentes, gerando *real-time views*. Caso uma consulta seja feita ocorrerá uma (6) soma entre os resultados dos *batch views* e *real-time views*, sendo esta soma o resultado desejado para a consulta (MARZ; WARREN, 2015).

A Arquitetura Lambda garante sua resiliência através da *isolação de complexidade*, que é obtida graças a separação entre camadas *batch* e *speed* (MARZ; WARREN, 2015). A resiliência é obtida pois, uma vez que os resultados são processados na camada *batch*, os resultados da camada *speed* podem ser descartados. Essa técnica é essencial, já que o processamento em tempo-real pode criar inconsistências, por conta da baixa precisão que é ocasionada por usar somente dados recentes; contudo, essa inconsistência é corrigida no próximo *batch* a ser processado, possibilitando que os resultados incoerentes da camada *speed* sejam descartados (MARZ; WARREN, 2015).

Quanto à sua implementação no InterSCity, a Arquitetura Lambda pode precisar de adaptações. Na literatura, a camada *serving* é a responsável pela junção entre o resultado do processamento feito pelas camadas *speed* e *batch*, e a separação entre as camadas *batch* e *serving* é bem definida (MARZ; WARREN, 2015), contudo, isso depende das ferramentas definidas para uso.

3.2 ARQUITETURA KAPPA

A Arquitetura Kappa é um padrão de projeto de *software*, e surgiu após questionamentos[4] quanto a complexidade da Arquitetura Lambda. Kappa é uma arquitetura recente (2014), simples, e se baseia somente no uso da **camada *speed*** (SEYVET, 2016). É guiada por quatro princípios: (i) tudo é um *stream*; (ii) os dados devem ser imutáveis; (iii) somente um *framework* para processamento deve ser utilizado; (iv) a reprodução histórica deve ser disponível (SEYVET, 2016).

Fazendo uso da observação de que o *log* é um conjunto de informações imutáveis e com ordenação bem definida, a Arquitetura Kappa pode utilizá-lo para atingir os quatro princípios citados anteriormente (KREPS, 2014). O *log* é processado em tempo real, permitindo consultas em baixa latência, com a possibilidade de acesso de dados atuais e históricos (FORGEAT, 2015). Caso o processamento seja iniciado após o fluxo de dados já ter começado, duas opções são possíveis: processar o *log* do início, tornando disponível

[4] <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>

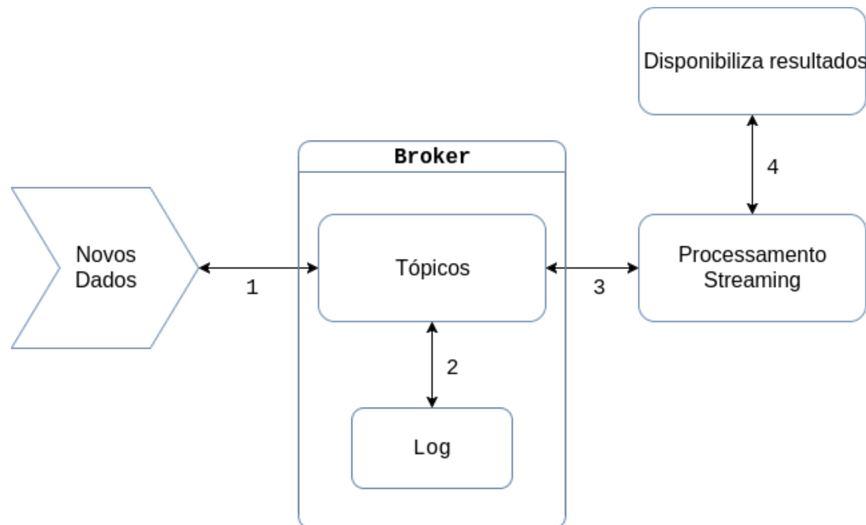

Figura 4 – Funcionamento da Arquitetura Kappa. Baseado em: Seyvet (2016).

os dados históricos, ou processar a partir do final, não tendo acesso aos dados históricos, mas tendo latência mínima (KREPS, 2014).

O funcionamento da Arquitetura Kappa está apresentado na Figura 4, e começa a chegada de novos dados, que são (1) publicados em tópicos do *broker*. O *broker* anexa (2) os novos dados em um *log* distribuído e (3) os transfere para ferramentas de processamento *streaming* que estejam inscritas no tópico relacionado. Por fim, (4) os resultados do processamento são disponibilizados para serem consumidos.

Assim como levantado a respeito da Arquitetura Lambda, a implementação da Arquitetura Kappa também pode precisar de adaptações para ser implantada no InterSCity. Um casamento entre as tecnologias escolhidas deve ocorrer para que a arquitetura seja implementada como sugerido na literatura, o que pode não ser possível dadas as restrições da plataforma.

3.3 BROKER

A comunicação entre diferentes módulos em plataformas de processamento de dados pode ocorrer de diversas formas, e uma destas é através da passagem de mensagem via PubSub, já utilizado pelo InterSCity (ESPOSTE et al., 2017). O ***broker*** é o mediador responsável por orquestrar as diferentes mensagens que são transmitidas via PubSub (MARZ; WARREN, 2015), sendo primordial no funcionamento das duas arquiteturas de *Big Data* mencionadas anteriormente.

Uma abstração que facilita a compreensão do *broker* é pensá-lo como o mediador de um sistema de notícias. Uma entidade, desejando transmitir uma notícia, a publica em um **tópico**, agindo como o produtor do conteúdo. Outras entidades que desejam ser

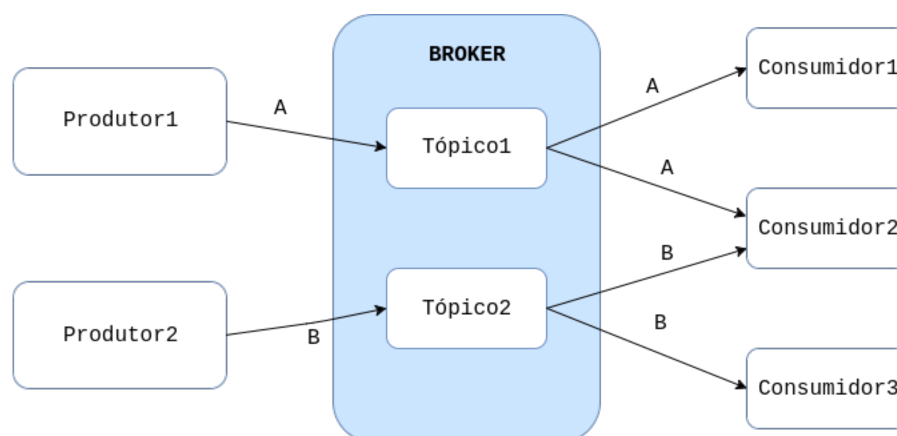

Figura 5 – Funcionamento do *broker*.

notificadas sobre aquele tema inscrevem-se no tópico associado, e serão notificadas quando uma notícia referente for publicada, agindo como consumidores. O *broker* gerencia esse processo, e tem a tarefa de transferir as mensagens de um emissor para receptores.

A Figura 5 ilustra o funcionamento do *broker*, onde é apresentado um cenário típico de atuação. No exemplo mostrado, a mensagem A é transmitida pelo Produtor1 no Tópico1, e a mensagem B é transmitida pelo Produtor2 no Tópico2. O Consumidor1 e o Consumidor2 recebem a mensagem A, por estarem inscritos no Tópico1, e os consumidores Consumidor2 e Consumidor3 recebem a mensagem B, por estarem registrados no Tópico2. Este funcionamento pode receber variações, a depender das tecnologias utilizadas, e é útil na implementação das arquiteturas Kappa e Lambda, sendo o *broker* o responsável por recuperar novas informações e as disponibilizar para as ferramentas de processamento.

3.4 COMPARATIVO ENTRE TECNOLOGIAS

Nesta seção apresentamos diferentes alternativas para compor a arquitetura do novo serviço de processamentos do InterSCity. Separamos as ferramentas nas categorias *processamento batch*, *processamento streaming* e *broker*. Não analisamos ferramentas de banco de dados pois uma troca seria prejudicial e não interessante para a equipe do InterSCity. Analisamos um *broker* diferente do utilizado pelo InterSCity considerando sua atuação somente com o serviço de processamento, não alterando os microsserviços existentes da plataforma.

3.4.1 Ferramentas de Processamento Batch

Analisamos duas ferramentas de processamento *batch*: o **Apache Hadoop MapReduce**, precursor no ecossistema *Big Data*, e o **Apache Spark**, ferramenta mais recente. O MapReduce é uma das ferramentas que compõe o ecossistema do Hadoop, e já foi

utilizado em contextos extremos[5], atuando com excelência em cenários de grande massa de dados (ZAHARIA et al., 2008), sendo uma escolha segura para compor a camada *batch* da Arquitetura Lambda. Só dispõe de API nativa para linguagem Java, de modo que a implementação do MapReduce no InterSCity deva utilizar o uso da saída e entrada padrão do sistema operacional para que seja possível o uso da linguagem Ruby, linguagem de maior domínio pelo time do InterSCity.

Um dos questionamentos feitos ao MapReduce é em relação ao constante acesso e uso do disco, que deve ser feito sempre que um *job* é finalizado. Por conta desta característica, seu uso não é facilmente justificável em cenários em que a massa de dados não é grande o suficiente, e uma opção válida nestes casos acaba sendo o Apache Spark, que evita o uso do disco através de *micro-batches* (TAVAKOLI-SHIRAJI; DAS; WENDELL, 2014). Esses *micro-batches* têm seu tempo de processamento definidos em código, de modo que seja possível definir tempos de *micro-batches* pequenos o suficiente para que seja considerado tempo-real[6].

Outras características importantes do Spark para o contexto do InterSCity são: API nativa em Python, Scala, Java e R; biblioteca de *streaming*, permitindo que o Spark seja utilizado também na camada *speed* da Arquitetura Lambda; e biblioteca de clusterização e aprendizagem de máquina, que pode ser útil para compor um futuro *pipeline de dados* do InterSCity. A Tabela 1 apresenta um comparativo entre a performance do MapReduce e do Spark, feita em uma competição que ocorreu em 2014, chamada Sort-Benchmark[7]. Na competição, o Spark apresentou uma performance três vezes maior que o Hadoop MapReduce, utilizando dez vezes menos recursos.

Tabela 1 – Resultados da Sort Benchmark 2014, categoria GraySort. Fonte: Databricks, 2014[8].

	Hadoop MRRecord	**SparkRecord**	**Spark1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s(est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

3.4.2 Ferramentas de Processamento Streaming

Analisamos duas ferramentas de processamento *streaming*: O **Apache Storm**, escrito por Nathan Marz, criador da Arquitetura Lambda, e o **Apache Spark**, analisado

[5] Lista das empresas que utilizam ou já utilizaram o Hadoop: <https://wiki.apache.org/hadoop/PoweredBy>
[6] O tempo-real mencionado durante este trabalho se refere ao *soft real-time*.
[7] <http://sortbenchmark.org/>
[8] <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

anteriormente como ferramenta de processamento *batch*. O Apache Storm permite o uso de qualquer linguagem de programação que interaja com a saída e entrada padrão do sistema operacional, e sendo sugerido por Nathan Marz em seu livro sobre a Arquitetura Lambda (MARZ; WARREN, 2015), se faz uma opção segura para compor a camada *speed*.

O Spark é utilizado como ferramenta de *streaming* graças ao seu módulo Spark Streaming, que tem nativamente integração com várias ferramentas que podem servir de produtores, como o Kinesis, Kafka, HDFS, dentre outras. O grande diferencial entre o Spark e o Storm é que o Spark faz processamento *streaming* via *micro-batches*, ou seja, processa em intervalo de tempos pré-definidos, enquanto o Storm processa em tempo-real, conforme chegam novos dados. Essa diferença resulta em um tempo de latência notável entre os dois, onde o Spark apresenta tempo de latência com pouca variância conforme o volume de dados aumenta, enquanto o Storm apresenta latência mínima com baixo volume de dados, mas que vai aumentando conforme o volume aumenta. Um estudo feito relatou a diferença de performance entre as duas ferramentas[9], contudo, o cenário aplicado utilizava pequena massa de dados, favorável ao Storm.

O Storm apresenta como vantagem em relação ao Spark a performance superior e a flexibilidade de uso da linguagem Ruby, já utilizada pelo time do InterSCity, porém o Spark conta com a possibilidade de processamento *batch*, integração nativa com ferramentas produtoras, e o acesso a ferramentas de clusterização e aprendizagem de máquina, que por serem interessantes para o InterSCity, equilibram o *trade-off* entre as duas ferramentas.

3.4.3 Broker

Analisamos dois *brokers*: o **RabbitMQ**, utilizado extensivamente pelo InterSCity, e o **Apache Kafka**, constantemente referenciado para implantação da Arquitetura Kappa. O RabbitMQ é um *broker* bem difundido, utilizado por empresas como a Cisco, Instagram, New York Times, dentre outros[10], e apresenta suporte para diversas linguagens (ZAITSEV, 2014), o que facilitou sua adoção pelo time do InterSCity. Podem ser destacados como diferenciais do RabbitMQ em relação a outros *brokers*: tolerância a falhas, processamento distribuído, alto desempenho, filas (e tópicos) com composições mais complexas e a possibilidade de uso de *plugins*[11], que suportam, por exemplo, federação[12].

O Apache Kafka é uma ferramenta mais nova[13], e também é utilizado em contextos extremos por várias empresas[14]. Tendo como um dos principais diferenciais a performance

[9] <http://xinhstechblog.blogspot.com.br/2014/06/storm-vs-spark-streaming-side-by-side.html>
[10] <https://www.rabbitmq.com/>
[11] <https://www.rabbitmq.com/plugins.html>
[12] <https://www.rabbitmq.com/federation.html>
[13] O RabbitMQ teve sua primeira *release* em 2007, e o Apache Kafka em 2011.
[14] <https://cwiki.apache.org/confluence/display/KAFKA/Powered+By>

e a escalabilidade, recentemente foi capaz de lidar com mais de 1.4 trilhões de mensagens diárias, distribuídas sobre 1400 nós (KOSHY, 2016). Um outro diferencial relevante entre o Kafka e os outros concorrentes é o suporte padrão à integração entre *logs* e tópicos, importante na implementação da Arquitetura Kappa. Ainda, o Kafka conta com sistema de tolerância a falhas e suporte a várias linguagens de programação[15].

Ambas as opções são sólidas e adequadas para o uso do InterSCity. A performance entre os dois já foi comparada, e embora o Kafka tenha tido performance superior[16], a performance do RabbitMQ não compromete o uso no InterSCity. Assim, os pontos cruciais que levaremos em conta na escolha de tecnologia são: o fato do Kafka ter suporte a *log* e *handler* nativo para o Spark, e o RabbitMQ já ser utilizado pelo InterSCity e ter o uso estendido via *plugins*.

15 <https://cwiki.apache.org/confluence/display/KAFKA/Clients>
16 <http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/>

4 PROJETO E IMPLEMENTAÇÃO DO SERVIÇO DE PROCESSAMENTO

Escolhemos a **Arquitetura Kappa** como padrão de projeto para servir de base para o novo serviço de processamento de dados do InterSCity. A Arquitetura Lambda não justifica a maior complexidade no contexto atual da plataforma, de modo que essa escolha facilita a manutenibilidade e adoção do serviço pelo time atual do InterSCity. A Arquitetura Kappa permitirá a análise em tempo-real sem que ocorra perda de informações relevantes, o que é importante no contexto de cidades inteligentes, ao passo em que permite a análise de dados históricos, desde que tenha ocorrido o pré-processamento. Essa decisão resultou na necessidade de escolha de uma tecnologia de processamento *streaming* e de um *broker* adequado.

Escolhemos o **Apache Spark** como tecnologia de *streaming* a ser usada, principalmente por dispor nativamente de biblioteca de clusterização e aprendizagem de máquina. O Spark ainda facilita, caso necessário, a troca para a Arquitetura Lambda, por dispor de processamento *batch*. Escolhemos o **Apache Kafka** como o *broker* do novo serviço de processamento, sendo essa uma escolha menos óbvia que a anterior. Embora o RabbitMQ já seja utilizado pelo InterSCity e tenha vantagens em certos aspectos em relação ao Kafka, tomamos essa decisão pelo RabbitMQ não dispor de uma interface nativa que o conecte ao Spark. Outro fator que levamos em conta é o fato do Kafka ter gerenciamento nativo de *log*, que ajuda na implantação da Arquitetura Kappa. Contudo, só utilizaremos o Kafka no serviço de processamento de dados, não forçando mudanças no ecossistema de microsserviços do InterSCity.

A Figura 6 ilustra a pilha de tecnologias que deve compor a Arquitetura Kappa no InterSCity, e as principais relações entre os diferentes serviços.

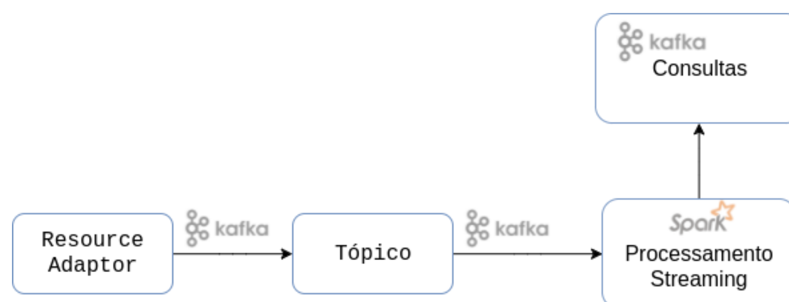

Figura 6 – Pilha de tecnologias utilizadas - Apache Kafka e Apache Spark, e suas interações com o InterSCity.

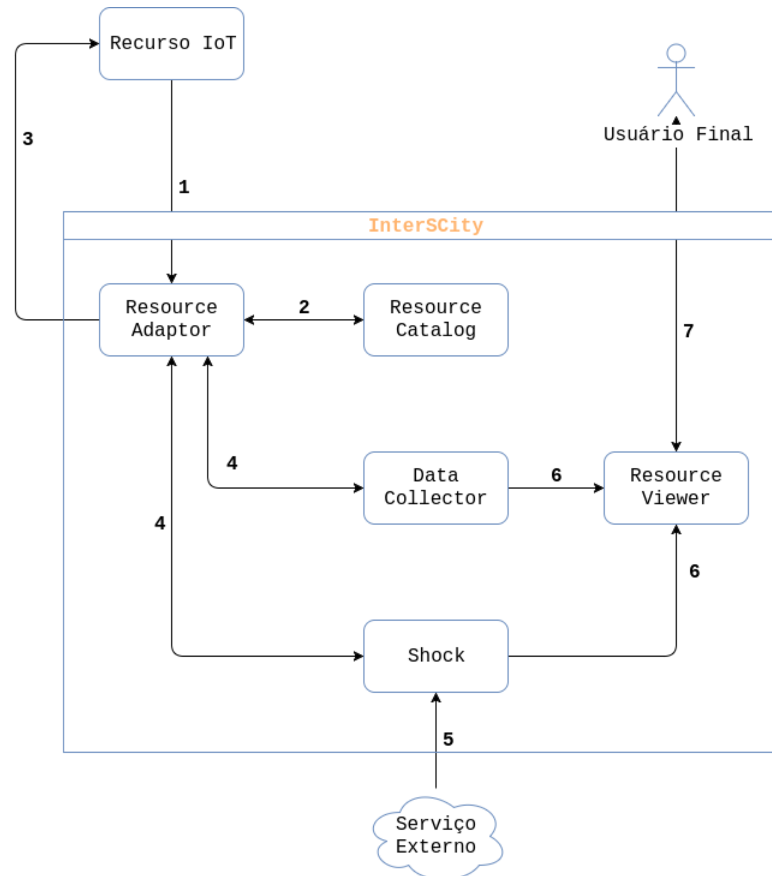

Figura 7 – Novo ciclo de vida da plataforma, com relação ao novo serviço de processamento.

4.1 IMPLEMENTAÇÃO

Dividimos a implementação da Arquitetura Kappa em três etapas: (i) configuração do ambiente, contemplando as ferramentas escolhidas; (ii) ligações entre os diferentes serviços, tornando possível a publicação de mensagens no Kafka e sendo possível seu processamento no Spark; e (iii) disponibilização de *hooks* que possam ser estendidos futuramente, possibilitando a criação de um *pipeline de dados* customizável.

Durante a implementação do novo serviço de processamento de dados para o InterSCity desenvolvemos o **Shock**, um componente responsável por abstrair as comunicações entre as diferentes ferramentas e trazer a extensibilidade mencionada na terceira etapa da implementação. Apresentamos no Apêndice B pseudo algoritmos que ilustram o uso das ferramentas escolhidas na implementação da Arquitetura Lambda[1].

A Figura 7 ilustra o novo fluxo completo do InterSCity com a adição do novo serviço de processamento de dados. O início do fluxo (passos 1 ao 3) continua o mesmo, e foi detalhado na Seção 2.1. As mudanças começam quando o (4) Resource Adaptor passa

[1] Pseudo códigos para a Arquitetura Lambda foram disponibilizados como forma de complemento, pois o Shock já contempla a Arquitetura Kappa.

a publicar a chegada de novos dados também no Shock através do Kafka, algo que não trouxe grandes mudanças, mas que estendeu o InterSCity. Serviços externos (5) anunciam no Shock quais operações devem ser executadas no *stream* de dados, de modo que os novos dados serão processados com esse conjunto de operações. Por fim, os resultados do Shock e do Data Collector (6) serão disponibilizados, podendo ser consumidos por aplicações como o microsserviço Resource Viewer, que trata os dados para (7) disponibilizá-los ao usuário final.

Assim como seguido pelo time do InterSCity, utilizamos o Docker na gerência de configuração do novo serviço. Reutilizamos uma configuração do Spark que já havia sido desenvolvida pelo time do InterSCity, configuramos um contêiner com o Kafka, e o ligamos aos contêineres do Spark e do microsserviço Resource Adaptor.

Iniciamos a ligação entre os projetos com uma adaptação no microsserviço Resource Adaptor, que com a mudança, passou a publicar em um tópico específico no Kafka a atualização de novos dados. Essa adaptação não trouxe mudanças significativas no InterSCity, não afetando o fluxo usual da plataforma. Desenvolvemos o Shock, responsável por receber mensagens em tópicos específicos do Kafka e passá-los ao Spark Streaming. O Shock gerencia a execução de *micro-batches* do Spark em intervalos de tempos definidos previamente, e em cada um destes processamentos, *hooks* extensíveis são executados. Caso queira-se anexar mais uma tarefa no *pipeline de dados*, basta então registrá-la no Shock e fornecer uma prioridade, que define se uma tarefa deve ser executada antes ou depois.

4.2 SHOCK

O Shock encontra-se disponível em um repositório no Gitlab[2], e o novo microsserviço de processamento de dados pode ser encontrado em um *fork* do serviço original[3]. O Shock abstrai as configurações entre as diferentes ferramentas e pode ser customizado por serviços externos que definem quais operações devem ser executadas. É configurável, possuindo um *handler* para a Arquitetura Kappa desenvolvido[4], mas caso seja desejado a implementação de um outro *handler*, basta implementar as funções obrigatórias e fornecer o novo módulo como parâmetro para o Shock.

A Figura 8 ilustra o uso do Shock, utilizando como exemplo uma aplicação de cidades inteligentes desenvolvida pelo time do InterSCity[5]. Essa aplicação disponibiliza as coordenadas, tempo, o índice e a linha de alguns ônibus de São Paulo. Imaginando que seja desejado a criação de uma nova informação a partir destes dados, como a *velocidade*,

[2] <https://gitlab.com/DGuedes/shock>
[3] <https://gitlab.com/DGuedes/data-processor>
[4] <https://gitlab.com/DGuedes/shock/blob/master/shock/handlers.py>
[5] <https://gitlab.com/smart-city-software-platform/external-resources/tree/master/bus>

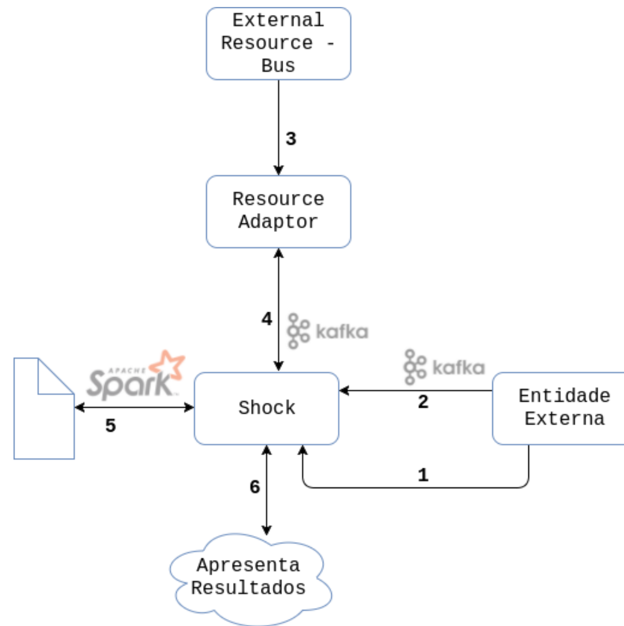

Figura 8 – Ciclo de vida do Shock dentro do InterSCity.

o fluxo de uso com o Shock poderia ser: (1) uma entidade[6] fornece ao Shock um arquivo contendo um conjunto de funções necessárias para compor o *pipeline de dados*. Essa etapa é opcional, e o controle entre quem pode ou não enviar os arquivos desejados não cabe ao Shock, e sim a quem tem acesso ao servidor. A entidade externa então (2) constrói o *pipeline de dados* através de publicações no Kafka, que serão recebidos pelo Shock, transformando os *streams* de processamento. Os recursos IoT (3) publicam no Resource Adaptor a chegada de novos dados, neste caso, sobre os ônibus. O Resource Adaptor (4) publica os novos dados em um tópico específico no Kafka para que sejam recebidos pelo Shock, que os (5) processa através do Spark no próximo *micro-batch* escalonado, já utilizando as funções registradas no passo 2. Por fim, após o processamento dos dados, o Shock (6) disponibiliza os resultados do processamento.

[6] Entidade neste cenário se trata de qualquer serviço externo que deseje contribuir com o InterSCity.

5 CONSIDERAÇÕES PRELIMINARES

As contribuições deste trabalho propiciaram ao InterSCity a possibilidade de atuação em cenários mais extremos e abrangentes, por meio de um novo serviço para processamento de seus dados. Juntamente com o novo serviço, desenvolvemos uma aplicação que abstrai as ferramentas de *Big Data* utilizadas pelo projeto, e que fornece meios para que outras aplicações de cidades inteligentes requisitem tarefas para serem processadas pela plataforma. Com a incorporação dos resultados que desenvolvemos o InterSCity passa então: a poder atuar em operações com grande massa de dados; a fornecer processamento de dados para terceiros através da plataforma; e a possibilitar o uso de algoritmos mais complexos, como de aprendizado de máquina.

Começamos esta primeira etapa do trabalho em março/2017, e estivemos envolvidos com a revisão bibliográfica e das técnicas sobre arquiteturas e tecnologias de *Big Data*, assim como na concepção do serviço de processamento de dados para o InterSCity. Desta forma, ainda existem várias melhorias a serem feitas na implementação inicial do Shock, como a resolução de dívidas técnicas e apuração de maior referencial científico. Portanto, planejamos as seguintes atividades para a continuação do trabalho:

- **Segunda rodada de revisão na bibliografia**: para termos esta proposta e os primeiros resultados em 2 meses, fizemos uma primeira rodada de revisão bibliográfica, e entendemos que as tecnologias e técnicas utilizadas são recentes, o que não nos trouxe muitas referências acadêmicas. Com uma segunda revisão bibliográfica, poderemos gerar mais insumos para justificativas e adaptações a serem implantadas no InterSCity;
- **Desacoplar o núcleo do Shock do Kafka**: o Shock neste momento está acoplado ao Kafka em diversos aspectos, de modo que a escrita de testes seja difícil, e impossibilitando o uso de outros *streams* no Spark, como o de MQTT;
- **Testar o núcleo do Shock**: por conta do pouco tempo e do acoplamento entre o Kafka e o Shock, não desenvolvemos os testes da aplicação;
- **Documentar a API de serviços**: por conta da alta volatilidade da arquitetura e da API do Shock, não produzimos uma documentação completa. Uma melhor documentação ajudará o time do InterSCity a prosseguir com o serviço desenvolvido;
- **Utilizar outras estruturas de dados**: atualmente o Shock só conta com RDD's, que são estruturas elementares do Spark. A troca por outras estruturas, como os Data Frames, pode melhorar a performance e a manutenibilidade do serviço;

- **Customização de janelas de *micro-batches***: o Shock utiliza o mesmo tempo de janela de *micro-batch*, independente do contexto. Uma customização ajudaria na adaptação do Shock em um maior número de contextos, sendo útil inclusive no desenvolvimento de testes;
- **Múltiplos *streams***: o Shock só utiliza os *streams* do Kafka, e sempre um *stream* único. Com múltiplos *streams* seria possível ter múltiplos *pipeline de dados*, interessante para as aplicações que utilizam a plataforma;
- **Controle de *check-point***: o controle de *check-points* adicionará maior tolerância ao Shock em caso de falhas graves, diminuindo os pontos de ruptura;
- **Utilizar *log* do Kafka**: a utilização do *log* do Kafka possibilitará a recuperação de dados históricos;
- **Introduzir o Shock ao *core* do InterSCity**: o Shock não está em um estado estável o suficiente para fazer parte do InterSCity. Quando o Shock estiver nesse nível de estabilidade, ele poderá finalmente fazer parte do núcleo da plataforma.

Planejamos algumas das pendências do novo serviço de processamento para serem resolvidas no TCC 2, conforme o cronograma apresentado na Tabela 2.

Tabela 2 – Cronograma com o planejamento das dívidas técnicas para o TCC 2.

Mês	**Atividade**	**Esforço**
Maio	Revisão mais profunda na bibliografia a respeito das tecnologias e arquiteturas no contexto de cidades inteligentes.	Alto
Maio	Desacoplar o núcleo do Shock do Kafka.	Médio
Maio e Junho	Testar o núcleo do Shock.	Alto
Maio e Junho	Documentar a API de serviços.	Baixo
Junho	Disponibilizar customização de janelas de *micro-batch*.	Baixo
Julho	Utilizar recuperação de dados históricos através dos *logs* do Kafka.	Médio
Julho	Agregar o novo serviço de processamento e as mudanças necessárias para o núcleo do InterSCity.	Médio
Julho	Apresentar segunda etapa do trabalho.	-

Definimos pendências de menor prioridade para serem resolvidas após o TCC, e estão apresentadas na Tabela 3.

Tabela 3 – Atividades para serem feitas após o TCC.

Atividade	**Esforço**
Utilizar Data Frames e outras estruturas de dados do Spark, e não só RDD's.	Médio
Permitir múltiplos *streams* através da mesma API (possibilitando o uso do Kinesis, HDFS, Sockets, entre outros).	Alto
Adicionar o controle de *check-points*.	Médio

Referências

ARMSTRONG, J. Making reliable distributed systems in the presence of software errors. 2003. Citado na página 19.

BATISTA, D. M. et al. Interscity: Addressing future internet research challenges for smart cities. In: *Online Proceedings*. IEEE, 2016. ISBN 978-1-5090-4671-3. Disponível em: <http://ieeexplore.ieee.org/document/7810114/>. Citado na página 19.

BATTY, M. et al. Smart cities of the future. *The European Physical Journal Special Topics*, Springer-Verlag, v. 214, n. 1, p. 481–518, 2012. Citado na página 17.

ESPOSTE, A. de M. D. et al. Interscity: A scalable microservice-based open source platform for smart cities. In: *Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems*. [S.l.: s.n.], 2017. Citado 7 vezes nas páginas 9, 17, 19, 20, 21, 26 e 43.

FORGEAT, J. Data processing architectures – lambda and kappa. 2015. Disponível em: <https://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa>. Citado na página 25.

GUTIÉRREZ, V. et al. Smartsantander: Internet of things research and innovation through citizen participation. In: ______. *The Future Internet: Future Internet Assembly 2013: Validated Results and New Horizons*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 173–186. ISBN 978-3-642-38082-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-38082-2_15>. Citado na página 17.

KIRAN, M.; MURPHY, P.; BAVEJA, S. S. Lambda architecture for cost-effective batch and speed big data processing. In: *First Workshop on Data-Centric Infrastructure for Big Data Science (DIBS)*. [S.l.: s.n.], 2015. Citado na página 23.

KON, F.; SANTANA, E. F. Z. Cidades inteligentes: Conceitos, plataformas e desafios. In: ______. *Jornadas de Atualização em Informática 2016 — JAI*. [S.l.]: SBC, 2016. ISBN 978-85-7669-326-0. Citado na página 17.

KOSHY, J. Kafka ecosystem at linkedin. 2016. Disponível em: <https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin>. Citado na página 30.

KREPS, J. Questioning the lambda architecture. 2014. Disponível em: <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>. Citado 2 vezes nas páginas 25 e 26.

MARZ, N.; WARREN, J. Big data - principles and best practices of scalable real-time data systems. 2015. Citado 6 vezes nas páginas 9, 23, 24, 25, 26 e 29.

NUAIMI, E. A. et al. Applications of big data to smart cities. *Journal of Internet Services and Applications*, v. 6, n. 1, p. 25, 2015. ISSN 1869-0238. Disponível em: <http://dx.doi.org/10.1186/s13174-015-0041-5>. Citado 2 vezes nas páginas 17 e 23.

SEYVET, N. Applying the kappa architecture in the telco industry. 2016. Disponível em: <https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry>. Citado 3 vezes nas páginas 9, 25 e 26.

TAVAKOLI-SHIRAJI, A.; DAS, T.; WENDELL, P. Apache spark 1.1: The state of spark streaming. 2014. Disponível em: <https://databricks.com/blog/2014/09/16/spark-1-1-the-state-of-spark-streaming.html>. Citado na página 28.

ZAHARIA, M. et al. Improving mapreduce performance in heterogeneous environments. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 29–42. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855744>. Citado na página 28.

ZAITSEV, P. Exploring message brokers: Rabbitmq, kafka, activemq, and kestrel. 2014. Disponível em: <https://dzone.com/articles/exploring-message-brokers>. Citado na página 29.

ZHENG, Z. et al. Real-time big data processing framework: challenges and solutions. *Applied Mathematics & Information Sciences*, Natural Sciences Publishing Corp, v. 9, n. 6, p. 3169, 2015. Citado na página 23.

Apêndices

APÊNDICE A – PRINCÍPIOS SEGUIDOS PELO INTERSCITY

O InterSCity foi desenvolvido utilizando princípios de *design*, e, assim, busca atender critérios estabelecidos. Os princípios são:

- **Modularidade através de serviços**: O InterSCity se torna mais modular através da criação de mais microsserviços, que buscam ter responsabilidades atômicas e bem definidas (ESPOSTE et al., 2017).
- **Modelos e Dados Distribuídos**: O InterSCity melhora sua escalabilidade através da distribuição dos dados e dos modelos. Com essa prática, cada microsserviço pode evoluir separadamente, por contar com seu próprio banco de dados (ESPOSTE et al., 2017). Contudo, esse princípio apresenta o ponto negativo de aumentar a complexidade.
- **Evolução Descentralizada**: Por conta do não-acoplamento, é possível que módulos do InterSCity evoluam e sofram manutenção independentemente, sem afetar outros microsserviços da plataforma (ESPOSTE et al., 2017).
- **Reuso de Projetos de Código Aberto**: O InterSCity preferencia projetos robustos já desenvolvidos, ao invés de desenvolver soluções do zero (ESPOSTE et al., 2017). Contudo, esta escolha é feita com cuidado, e somente projetos com colaboradores e mantenedores ativos, e com documentação apropriada, são utilizadas na plataforma (ESPOSTE et al., 2017).
- **Adoção de Padrões Abertos**: O InterSCity adota padrões já difundidos, para que seja provida maior interoperabilidade entre a plataforma e outros projetos (ESPOSTE et al., 2017).
- **Assíncrono contra Síncrono**: O InterSCity busca prover serviços e atividades assíncronas sempre que possível, com a finalidade de evitar que eventos blocantes ocorram. Isto é atingido principalmente através do padrão PubSub, e de estratégias baseadas em eventos (ESPOSTE et al., 2017).
- **Serviços sem Estado**: Os microsserviços do InterSCity evitam, sempre que possível, ter um estado específico (ESPOSTE et al., 2017). Com isso, os microsserviços podem responder a qualquer requisição a qualquer momento, ao contrário do que ocorreria caso tivessem estados específicos, pois só conseguiriam caso certas transições ocorressem.

APÊNDICE B – PSEUDO-IMPLEMENTAÇÃO - ARQUITETURA LAMBDA

O Shock só apresenta a implementação da Arquitetura Kappa, e, como forma de complementação, deixamos disponível uma pseudo-implementação com a utilização da Arquitetura Lambda.

B.1 CÓDIGO DA CAMADA BATCH

```
class BatchLayer(threading.Thread):
    daemon = True
    ...
    def run(self):
        heappush(hooks.actions, (10, Handler1.AcquireData))
        heappush(hooks.actions, (12, Handler2.AdjustDataModels))
        heappush(hooks.actions, (20, Handler3.ProcessData))
        heappush(hooks.actions, (21, Handler4.FindAnomalies))
        # extensible , new hooks can be added

        self.batch_view = sc.emptyRDD()

        while (True):
            # continuously processing
            rdd = spark.sql("SELECT * FROM sensor_data").cache()
                for op in hooks.actions:
                rdd = op(rdd).cache()
            self.batch_view = rdd # updates batch view
            rdd.unpersist()
            self.finished_batch = True # notifies speed layer
```

B.2 CÓDIGO DA CAMADA SPEED

```
class SpeedLayer(threading.Thread):
```

```
daemon = True
...
def run(self):
    self.reset_realtime_view = False
    self.new_realtime_view = False

    def reset_realtime_view(stream):
        # checks if the batch layer finished its processing
        if (batch_view.finished_batch):
            batch_view.finished_batch = False
            stream.foreachRDD(lambda rdd: rdd.unpersist())
            self.reset_realtime_views = True
            return stream.filter(lambda a: false)
        else:
            return stream

    def notify_serving(stream):
        self.new_realtime_view = True

    # connects to Kafka
    kvs = KafkaUtils.createDirectStream(ssc, [topic], \\
        {"metadata.broker.list": brokers})

    heappush(hooks.actions, (10, Handler1.AcquireData))
    heappush(hooks.actions, (12, Handler2.AdjustDataModels))
    heappush(hooks.actions, (20, Handler3.ProcessData))
    heappush(hooks.actions, (21, Handler4.FindAnomalies))
    heappush(hooks.actions, (0xfffff, notify_serving))
    heappush(hook.actions, (0xffffff, lambda realtime_view: \
        reset_realtime_view(realtime_view)))

    self.realtime_view = kvs
    for op in hook.actions:
          # register hooks to be executed by the stream
          self.realtime_view = op(self.realtime_view)

    ssc.start() # start the stream
    ssc.awaitTermination()
```

B.3 CÓDIGO DA CAMADA SERVING

```
class ServingLayer:
    ...
    def start(self):
        merged_data = sc.emptyRDD()
        speed_layer = SpeedLayer()
        batch_layer = BatchLayer()
        while (True):
            if (speed_layer.new_realtime_view):
                # updates results with available realtime views
                speed_layer.new_realtime_view = False
                merged_data = merged_data.union(\
                    speed_layer.realtime_view)
            if (speed_layer.reset_realtime_view):
                # reset realtime views when batches finishes
                merged_data = batch_layer.batch_view
                speed_layer.reset_realtime_view = False
            if (query):
                query(merged_data)
```