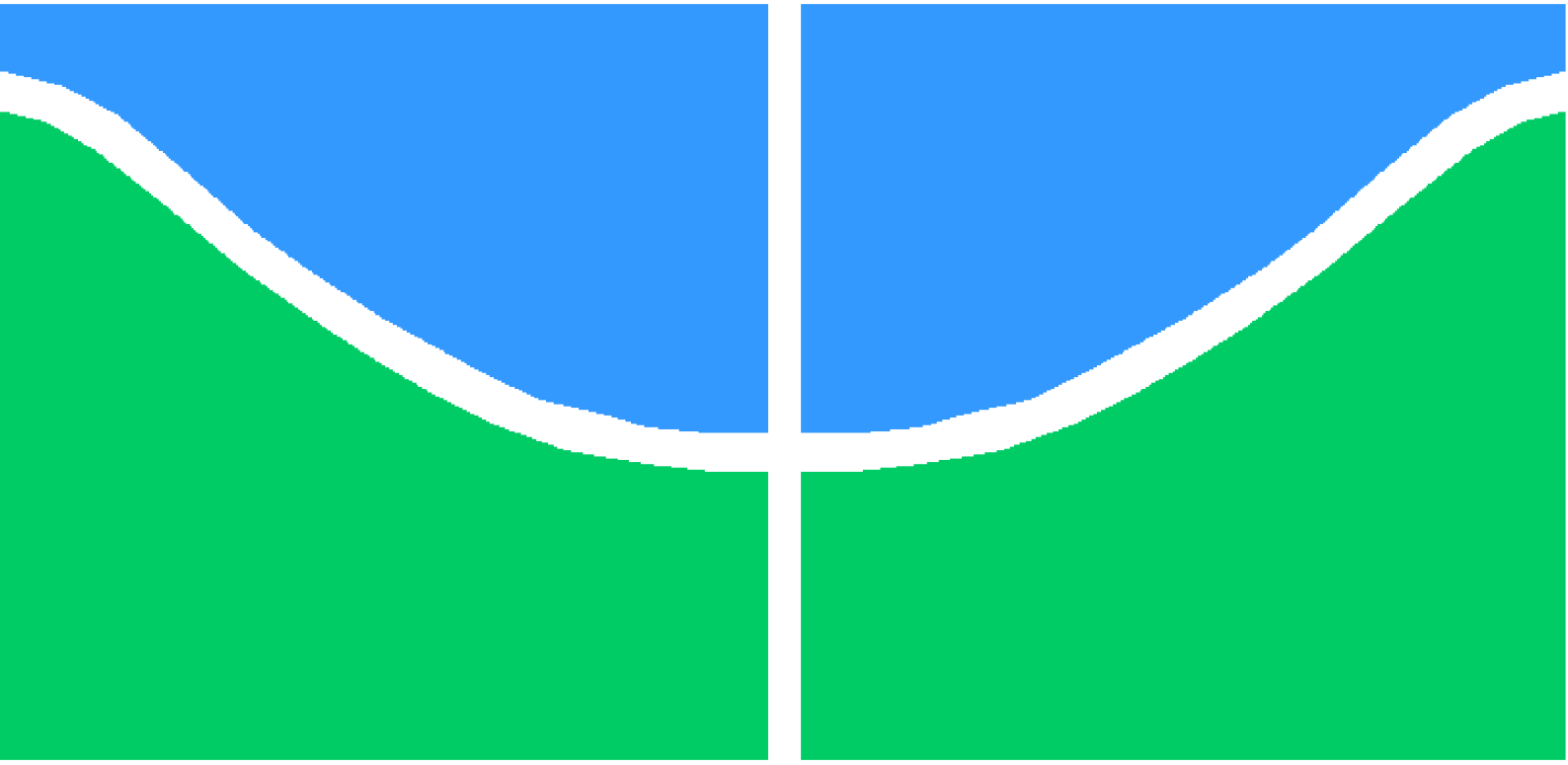

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

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-

51 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, 12 de Julho 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 através 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 through reusable, interoperable, and scalable services. However, InterSCity does not use suitable tools to process its data that is an obstacle in scenarios with 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

Lista de tabelas

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
API	*Application Programming Interface*

Sumário

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 alternativas para problemas graves e recorrentes das cidades atuais, como o mau uso de recursos, a burocracia, o transporte de má qualidade e a 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 e 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 viabilizar 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 definida e disponibilize 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).

Assim, este trabalho tem como objetivos o planejamento, análise e desenvolvimento do novo serviço de processamento de dados do InterSCity. Esses objetivos foram atingidos a partir das seguintes atividades, que serão descritas em maiores detalhes durante os capítulos que seguem:

---

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

- Análise de ferramentas de processamento de dados (TCC 1);
- Análise de padrões arquiteturais de *Big Data* para utilização no novo serviço (TCC 1);
- Desenho do novo serviço de processamento (TCC 1);
- Implementação do novo serviço de processamento (TCC 1 e 2);
- Desenvolvimento de uma aplicação que servisse de prova de conceito para o novo serviço desenvolvido (TCC 2);
- Definição e solução de casos de uso para o novo serviço (TCC 2);

Apresentamos no Capítulo 2 maiores detalhes sobre as características e o estado atual do InterSCity, trazendo também a abordagem que utilizamos para definir o novo serviço de processamento. No Capítulo 3 apresentamos 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. No Capítulo 5 apresentamos uma aplicação que exemplifica o uso do serviço de processamento, e por fim, no Capítulo 6, finalizamos o trabalho trazendo as considerações finais, as contribuições feitas e os trabalhos futuros.

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 (*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[2] da plataforma foram escritos em Ruby on Rails seguindo padrões que buscam extensibilidade e qualidade, e levando em conta princípios[3] importantes na busca por uma melhor arquitetura. A partir de experimentos feitos foi possível analisar o estado atual da performance e da escalabilidade do InterSCity, que se mostrou promissor (ESPOSTE et al., 2017). O projeto encontra-se hospedado no Gitlab[4], 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 segue uma arquitetura de microsserviços distribuída que possibilita armazenamento, análise, processamento, composição e integração de dados de recursos de cidades inteligentes (ESPOSTE et al., 2017). Os 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).

---

1 <www.mozilla.org/en-US/MPL/2.0/>

2 Os termos microsserviços, módulos e componentes serão utilizados intermitentemente, mas apresentando o mesmo significado.

3 Os princípios seguidos pelo InterSCity são apresentados no Apêndice A.

4 <https://gitlab.com/smart-city-software-platform>

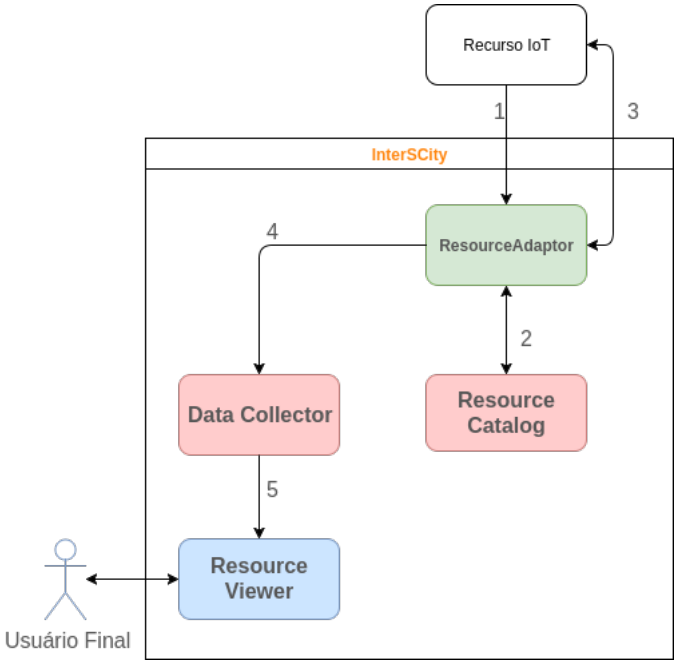

Figura 1 – Ciclo de vida de um recurso IoT no InterSCity. Baseado em: Esposte et al. (2017).

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).

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 recur-

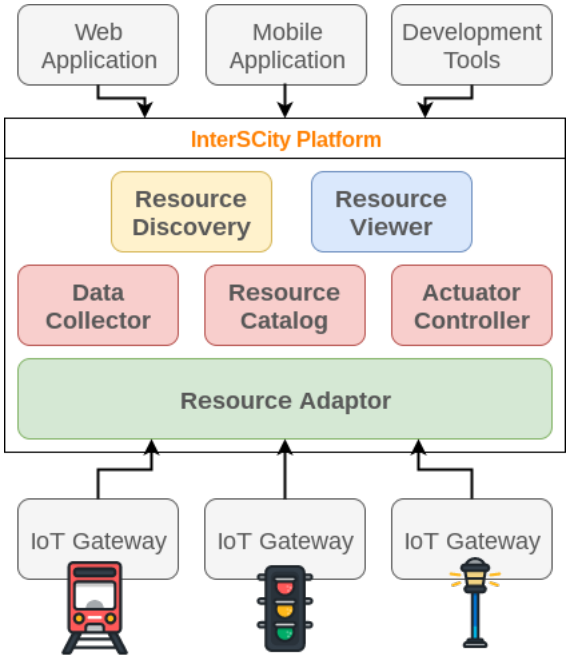

Figura 2 – Arquitetura do InterSCity. Fonte: Esposte et al. (2017).

sos, enquanto o Resource Discovery provê uma API (*Application Programming Interface*) 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, as aplicações clientes e os 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[5], onde cada microsserviço e dependência externa são executados em contêineres separados. O desenvolvimento da plataforma também faz extenso uso do Git[6], de modo que a configuração de um ambiente de teste do InterSCity tenha como pré-requisitos somente esses dois projetos: Docker e Git.

Cada microsserviço da plataforma contém seu próprio repositório, permitindo que evoluam de maneira distribuída. Com o propósito de ajudar no desenvolvimento, o In-

[5] <https://www.docker.com/>
[6] <https://git-scm.com/>

terSCity apresenta um repositório principal[7], que funciona como *repositório mestre*, sendo os repositórios dos microsserviços da plataforma submódulos.

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 pela equipe do InterSCity culminou em um serviço de processamento provisório, que pôde servir de base para o desenvolvimento da proposta do novo serviço de processamento. 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 dessa tecnologia. O Data Processor, contudo, apresenta uma solução bem específica, não podendo ser reaproveitado por outras aplicações.

Nesse contexto, neste trabalho, desenvolvemos um novo serviço de processamento de dados que permite que aplicações de cidades inteligentes utilizem o InterSCity no processamento e análise de seus dados. Através do novo serviço, um maior volume de dados poderá ser processado pela plataforma, possibilitando também o uso de algoritmos e operações complexas. O novo serviço faz uso de um padrão de projeto de *Big Data* adequado para o contexto de cidades inteligentes e compatível com a arquitetura atual do InterSCity. Fizemos um levantamento das arquiteturas de *Big Data* candidatas, assim como um estudo a respeito de possíveis ferramentas a serem utilizadas. Esse levantamento teve restrições, como: somente projetos software livre foram levados em conta e ferramentas que trouxessem grandes mudanças ao ecossistema do InterSCity teriam pouca prioridade.

A nova arquitetura de processamento de dados, apresentada neste trabalho, visa atender requisitos típicos de cidades inteligentes e possibilita a extensão para trabalhos futuros. A nova arquitetura permite, por exemplo, que um canal de dados possa ser utilizado, mesmo que faça uso de uma maior massa de dados, ou que uma aplicação processe seus dados através de operações específicas e customizadas, por chamadas diversas. Por fim, fornecemos um exemplo de aplicação que utiliza a arquitetura desenvolvida, como prova de conceito.

[7] <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 atingir esses 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.

Levantamos duas arquiteturas: a Lambda, que é mais difundida, e a Kappa, que escolhemos para utilização no novo serviço de processamento, e que é uma resposta a Arquitetura Lambda. Definimos também tecnologias que abranjam duas formas de processamento: processamento em lote e processamento de fluxos. No processamento em lote, 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 de fluxos, por outro lado, os dados são processados conforme estão disponíveis (ZHENG et al., 2015), permitindo consulta às 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 de processamento em lotes (*batch layer*), a camada de processamento de fluxo de dados (*speed layer*) e a camada de serviço (*serving layer*) (KIRAN; MURPHY; BAVEJA, 2015). Cada uma dessas camadas é implementada utilizando algoritmos e ferramentas específicas, de modo que certas ferramentas são mais apropriadas em certos contextos.

A **camada de processamento em lotes** 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 conjunto de dados mestre[1] (*master dataset*), que após processado, apresenta seus resultados condensados em visões de lotes (*batch views*), utilizados pela

[1] O conjunto de dados mestre é um lote histórico de informação que, por ser imutável, só possibilita **anexação** de informações.

camada de serviço. A camada de processamento em lotes é então, em sua essência, imutável, de 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 de processamento de fluxo de dados** tem como diferencial o processamento com baixa latência, que é obtido pelo uso de uma parcela menor da massa de dados[2]. Outra característica importante é que essa camada faz uso do processamento de fluxo de dados, estratégia que processa os dados conforme ficam disponíveis. Esse 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 de processamento em lote. Por esses motivos, a camada de processamento de fluxo de dados 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 da solução (MARZ; WARREN, 2015). Por fim, ela também é a camada que condensa os resultados de seu processamento em visões de tempo-real (*real-time views*), que serão fundidos com os resultados das visões de lote na apresentação do resultado. Ao final, os resultados da camada são dispensados após um lote terminar seu processamento (MARZ; WARREN, 2015).

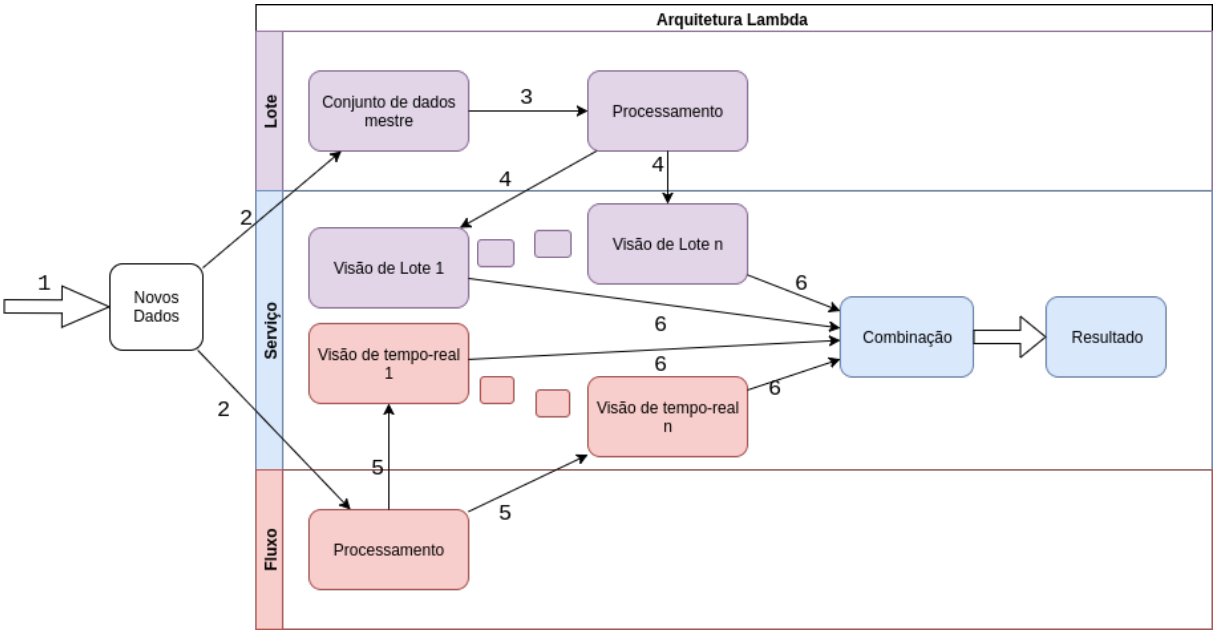

Figura 3 – Ciclo de vida na Arquitetura Lambda. Baseado em: Marz e Warren (2015).

[2] A camada de processamento de fluxo de dados só leva em conta dados que surgiram após a camada de processamento em lote 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 de processamento em lote quanto para a camada de processamento de fluxo de dados. A camada de processamento em lote (3) anexa os novos dados e os processa, (4) gerando assim visões de lotes que são enviadas para a camada de serviço. O mesmo dado que foi enviado para a camada de processamento em lote, no passo (1), também foi enviado para a camada de processamento de fluxo, onde (5) será processado com menor latência, por só levar em conta dados recentes, gerando visões de tempo-real. Caso uma consulta seja feita ocorrerá uma (6) soma entre os resultados dos visões de lote e visões de tempo-real, representando o resultado desejado para a consulta (MARZ; WARREN, 2015).

A Arquitetura Lambda garante sua resiliência através do *isolamento de complexidade*, que é obtido graças a separação entre camadas de processamento em lote e de fluxo de dados (MARZ; WARREN, 2015). A resiliência é obtida pois, uma vez que os resultados são processados na camada de processamento em lote, os resultados da camada de processamento de fluxo 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 lote a ser processado, possibilitando que os resultados incoerentes da camada de processamento de fluxo sejam descartados (MARZ; WARREN, 2015).

Do ponto de vista da implementação no InterSCity, a Arquitetura Lambda pode precisar de adaptações. Na literatura, a camada de serviço é a responsável pela junção de resultados das duas camadas que processam dados, sendo a separação entre as camadas 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[3] quanto a complexidade da Arquitetura Lambda. Kappa é uma arquitetura recente (2014) e simples, e se baseia somente no uso da **camada de processamento de fluxo de dados** (SEYVET, 2016). É guiada por quatro princípios: (1) tudo é um fluxo de dados; (2) os dados devem ser imutáveis; (3) somente um *framework* para processamento deve ser utilizado; (4) os dados devem ser armazenados, possibilitando a reprodução do estado (SEYVET, 2016).

Fazendo uso da observação que o *log* é um conjunto de informações imutáveis e com ordenação bem definida, a Arquitetura Kappa pode utilizá-lo para atingir os qua-

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

tro 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 ter começado, duas opções são possíveis: processar o *log* do início, tornando disponível os dados históricos; ou processar a partir do final, não tendo acesso aos dados históricos, mas tendo latência mínima (KREPS, 2014).

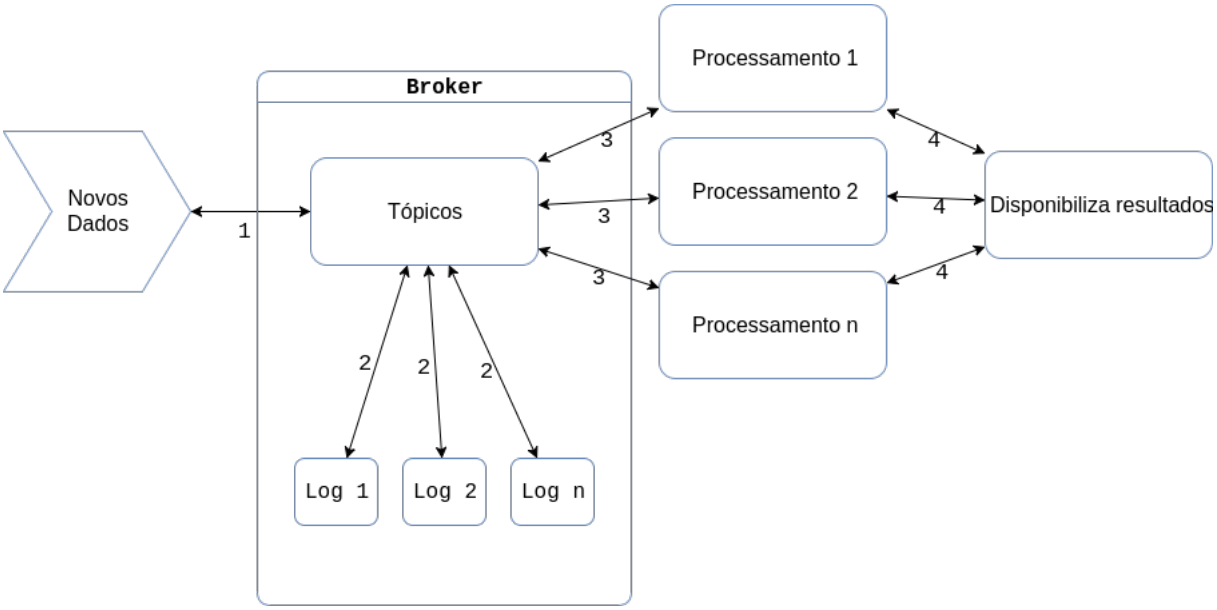

Figura 4 – Funcionamento da Arquitetura Kappa. Baseado em: Seyvet (2016).

O funcionamento da Arquitetura Kappa está apresentado na Figura 4, e começa com a chegada de novos dados, que são (1) publicados em tópicos do *broker*. O *broker* (2) anexa os novos dados em um *log* distribuído e (3) os transfere para ferramentas de processamento 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 formas diversas, e no InterSCity é feita via passagem de mensagem através do padrão PubSub (ESPOSTE et al., 2017). O ***broker*** atua como mediador

responsável por orquestrar as diferentes mensagens que são transmitidas (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 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.

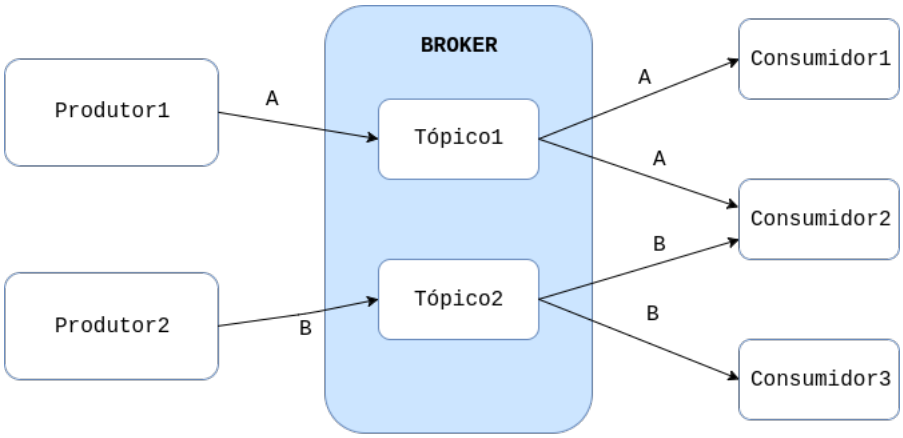

Figura 5 – Funcionamento do *broker*.

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. O 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 ARQUITETURAS

De maneira geral, as duas opções (Kappa e Lambda) parecem funcionar bem em cidades inteligentes. A Kappa pode ser vista como uma Arquitetura Lambda mais simples, que se sustenta em um menor conjunto de ferramentas e conceitos, enquanto a Lambda pode ser vista como uma opção segura, capaz de lidar com quase todos os problemas do ecossistema de cidades inteligentes. Decidimos pela Kappa, embora as duas arquiteturas apresentem vantagens, sendo o contexto um dos fatores primordiais.

A Arquitetura Kappa, sem a reprodução dos dados, permite somente o uso de algoritmos incrementais ou que não careçam analisar toda a massa de dados ao mesmo tempo. Em um algoritmo de aprendizagem de máquina, por exemplo, a Kappa pode funcionar, caso o treino do modelo seja incremental. Outro exemplo seria o de filtros, como a busca por um termo específico dentro de um texto - resultados anteriores não fazem diferença, e a Kappa mesmo sem a reprodução dos dados resolveria bem. Contudo, caso seja desejado uma operação em toda a massa de dados, como uma consulta em busca de um dado termo, a Kappa não funciona, enquanto a Lambda pode escalonar um *job* específico para essa tarefa sem maiores problemas. Dessa forma, pela Lambda suportar algoritmos e tarefas diferentes entre a camada de processamento em lote e processamento de fluxo, uma maior flexibilidade fica disponível, o que pode ser importante para certos contextos.

3.5 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 em lote, processamento de fluxos 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.

3.5.1 Ferramentas de Processamento em Lote

Analisamos duas ferramentas de processamento em lote: o **Apache Hadoop MapReduce**, precursor no ecossistema *Big Data*, e o **Apache Spark**, que optamos para compor o novo serviço de processamento. O Hadoop MapReduce é uma das ferramentas que compõe o ecossistema do Hadoop, e já foi utilizado, com excelência, em cenários de grande massa de dados[4] (ZAHARIA et al., 2008). Dispõe apenas de API nativa para linguagem Java, de modo que o uso do Hadoop MapReduce no InterSCity deva utilizar a saída e entrada padrão do Sistema Operacional para que seja possível o desenvolvimento em Ruby, linguagem de maior domínio pelo time do InterSCity.

Um dos questionamentos feitos ao Hadoop MapReduce é em relação ao constante acesso e uso do disco, que deve ser feito sempre que um *job* é finalizado. Por conta dessa característica, seu uso não é facilmente justificável em cenários em que a massa de dados não é grande o suficiente, e uma opção válida acaba sendo o Apache Spark, que troca

[4] Lista das empresas que utilizam ou já utilizaram o Hadoop: <https://wiki.apache.org/hadoop/PoweredBy>

o uso do disco pelo uso da memória, através da estratégia de micro-lotes (TAVAKOLI-SHIRAJI; DAS; WENDELL, 2014). Os micro-lotes têm tempo de processamento definidos em código, de modo que seja possível definir tempos de micro-lotes pequenos o suficiente para que seja considerado tempo-real[5].

Tabela 1 – Resultados da Sort Benchmark 2014, categoria GraySort. Fonte: Databricks, 2014[6].

	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**

Outras características importantes do Spark para o contexto do InterSCity são: API nativa em Python, Scala, Java e R; biblioteca de processamento de fluxos, permitindo que o Spark seja utilizado também na camada de processamento de fluxo da Arquitetura Lambda; e biblioteca de clusterização e aprendizagem de máquina, que pode ser útil para compor o futuro canal de processamento de dados do InterSCity. A Tabela 1 apresenta um comparativo entre o desempenho do Hadoop MapReduce e do Spark, pela competição SortBenchmark[7]. Na competição, o Spark apresentou um desempenho três vezes maior que o Hadoop MapReduce, utilizando dez vezes menos recursos.

3.5.2 Ferramentas de Processamento de Fluxo

Analisamos duas ferramentas de processamento de fluxo: O **Apache Storm**, escrito por Nathan Marz, criador da Arquitetura Lambda, e o **Apache Spark**, analisado anteriormente como ferramenta de processamento em lote, e que escolhemos para uso no novo serviço de processamento. 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 de processamento de fluxo de dados.

O Spark é utilizado como ferramenta de processamento de fluxos graças ao seu módulo Spark Streaming, que tem nativamente integração com outras 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 de fluxos via micro-lotes, ou seja, processa em intervalo de tempos pré-definidos, enquanto o Storm processa

[5] O tempo-real mencionado é sempre o *soft*.
[7] <http://sortbenchmark.org/>
[7] <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>

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 desempenho entre as duas ferramentas[8], contudo, o cenário aplicado utilizava pequena massa de dados, favorável ao Storm.

O Storm apresenta como vantagem em relação ao Spark o desempenho superior e a flexibilidade de uso da linguagem Ruby, já utilizada pelo time do InterSCity. Contudo, o Spark conta com a possibilidade de processamento em lote, 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.

3.5.3 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[9], 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*[10], que suportam, por exemplo, federação[11].

O Apache Kafka, ferramenta que decidimos para compor o novo serviço, é uma ferramenta mais nova[12], e também é utilizado em produção por diversas empresas[13]. Tendo como principais diferenciais o desempenho 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 a 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[14].

Ambas as opções são sólidas e adequadas para o uso do InterSCity. O desempenho entre os dois já foi comparado, e embora o Kafka tenha tido desempenho superior[15], o

[8] <http://xinhstechblog.blogspot.com.br/2014/06/storm-vs-spark-streaming-side-by-side.html>
[9] <https://www.rabbitmq.com/>
[10] <https://www.rabbitmq.com/plugins.html>
[11] <https://www.rabbitmq.com/federation.html>
[12] O RabbitMQ teve sua primeira versão estável em 2007, e o Apache Kafka em 2011.
[13] <https://cwiki.apache.org/confluence/display/KAFKA/Powered+By>
[14] <https://cwiki.apache.org/confluence/display/KAFKA/Clients>
[15] <http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/>

desempenho do RabbitMQ não compromete o uso no InterSCity. Assim, os pontos cruciais que levamos em conta na escolha de tecnologia foram: 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*. Assim, a escolha por qualquer uma dentre as duas tecnologias parece certa, mas ao final optamos pelo Kafka, pelos pontos citados.

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.

A decisão de uso da Arquitetura Kappa resulta na necessidade de escolha de uma tecnologia de processamento de fluxos e de um *broker* adequado. Ao final, o novo serviço de processamento trata-se do conjunto de serviços definidos (*broker* e ferramenta de processamento de fluxo) e aplicações desenvolvidas.

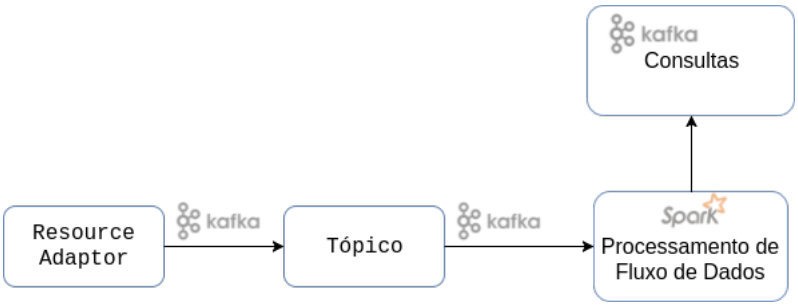

Figura 6 – Pilha de tecnologias utilizadas - Apache Kafka e Apache Spark, e suas interações com o InterSCity.

Escolhemos o **Apache Spark** como tecnologia de processamento de fluxos a ser usada, 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 em lote. 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 utilizamos 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.

4.1 IMPLEMENTAÇÃO

Dividimos a implementação da Arquitetura Kappa em três etapas: (1) configuração do ambiente, contemplando as ferramentas escolhidas; (2) ligações entre os diferentes serviços, tornando possível a publicação de mensagens no Kafka e sendo possível seu processamento no Spark; e (3) disponibilização de *hooks* que possam ser estendidos futuramente, possibilitando a criação de um canal de dados customizável.

Durante a implementação do novo serviço de processamento de dados para o InterSCity desenvolvemos o **Shock**, responsável por abstrair as comunicações entre as diferentes ferramentas e trazer a extensibilidade mencionada na terceira etapa da implementação. O Shock é uma aplicação que compõe o novo serviço de processamento, junto com outras aplicações utilizadas, e possibilita que usuários que não conhecem o Apache Spark consigam requisitar processamento e análise de fluxos de dados.

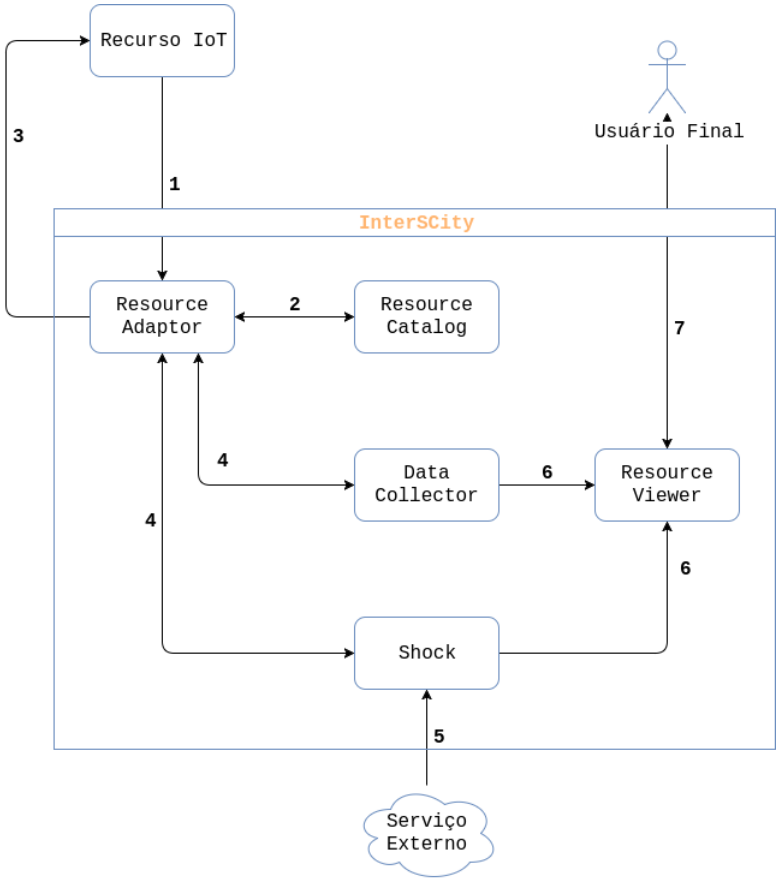

Figura 7 – Novo ciclo de vida da plataforma, com relação ao novo serviço de processamento.

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 começa com (1) um recurso IoT se registrando na plataforma através da interação com o Resource Adaptor. O Resource Adaptor (2) cadastra o recurso no microsserviço Resource

Catalog (3) e informa o UUID que deve ser utilizado pelo recurso. O recurso passa a enviar dados ao Resource Adaptor, que (4) publica a chegada dos novos dados no Shock através do Kafka, promovendo extensão ao InterSCity. Aplicações (5) interagem com o Shock via Kafka, construindo o canal de fluxo de dados, definindo as operações a serem executadas. Por fim, os resultados do Shock e do Data Collector (6) serão disponibilizados, podendo ser consumidos por aplicações como o microsserviço Resource Viewer, que (7) apresenta dados ao usuário final.

Assim como seguido pela equipe do InterSCity, utilizamos o Docker na gerência de configuração do novo serviço. A configuração do Spark que havia sido feita pelo time do InterSCity na criação do DataProcessor foi reutilizada, e configuramos um contêiner com o Kafka. Por fim, ligamos os contêineres configurados com o microsserviço Resource Adaptor, permitindo assim a interação entre o InterSCity e as ferramentas definidas para uso.

Iniciamos a ligação entre os projetos, etapa 1, com uma adaptação no microsserviço Resource Adaptor, que com a mudança, passou a publicar em um tópico específico no Kafka a chegada de novos dados. Essa adaptação não trouxe mudanças significativas no InterSCity, não afetando o fluxo usual da plataforma. Após, solucionamos as etapas 2 e 3 através do desenvolvimento do Shock, responsável por receber mensagens em tópicos específicos do Kafka e passá-los ao Spark Streaming. O Shock gerencia a execução de fluxo de dados do Spark, permitindo que usuários terceiros configurem fluxos de dados no Spark sem ter conhecimento técnico da ferramenta.

4.2 SHOCK

O Shock encontra-se disponível em um repositório no Gitlab[1] e o novo microsserviço de processamento de dados pode ser encontrado em um *fork* do serviço original[2]. O Shock abstrai o uso das diferentes ferramentas e pode ser customizado por serviços externos que definem e configuram fluxos para serem executados. A arquitetura do Shock apresenta pontos de estensão e possui um *handler* para a Arquitetura Kappa desenvolvido[3], mas caso seja desejado o uso de outra estratégia, basta implementar um novo *handler*.

[1] <https://gitlab.com/DGuedes/shock>
[2] <https://gitlab.com/DGuedes/data-processor>
[3] <https://gitlab.com/DGuedes/shock/blob/master/shock/handlers.py>

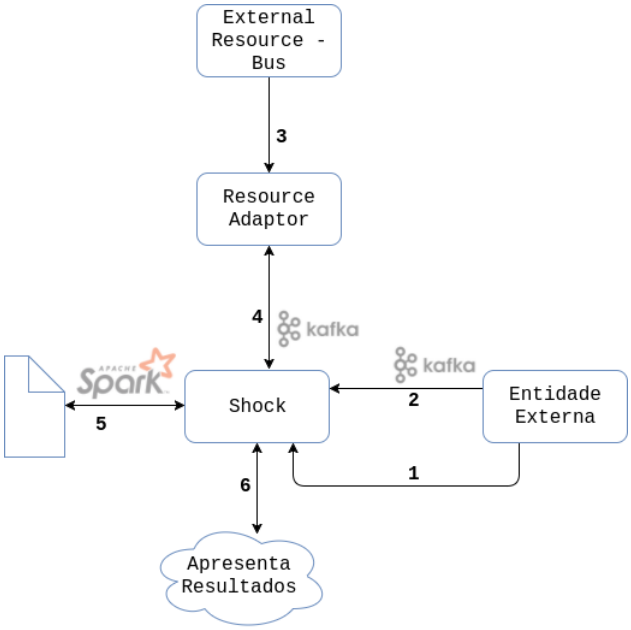

Figura 8 – Ciclo de vida do Shock dentro do InterSCity.

A Figura 8 ilustra o uso do Shock, utilizando como exemplo uma aplicação de cidades inteligentes desenvolvida pelo time do InterSCity[4], que disponibiliza dados como as coordenadas, o tempo, o índice e a linha de alguns ônibus de São Paulo. Imaginando que seja desejado a criação de uma nova informação a partir dos dados dos ônibus, como a *velocidade*, o fluxo de uso com o Shock poderia ser: (1) um cliente que deseje utilizar o serviço de processamentos do InterSCity analisa as funções que o Shock disponibiliza para criação do fluxo, e (2) constrói o fluxo desejado através dessas funções, via Kafka. Os recursos IoT (3) publicam no Resource Adaptor a chegada de novos dados de ônibus, para que esses dados sejam (4) publicados em um tópico específico do Kafka pelo Resource Adaptor. Após serem disponibilizados no Kafka, os dados ficam disponíveis para serem utilizados pelo Shock, que os (5) recebe nos fluxos de dados configurados para ingerir dados, e os processa através do canal construído no passo 2. Por fim, após o processamento dos dados, o Shock pode (6) disponibilizar os resultados do processamento, que pode ser reaproveitado por aplicações terceiras.

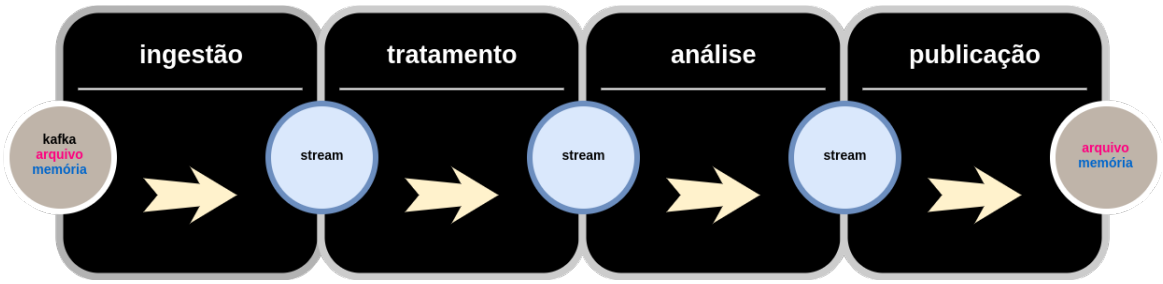

Figura 9 – Padrão *ingestão, preparo, análise e publicação.*

[4] <https://gitlab.com/smart-city-software-platform/external-resources/tree/master/bus>

Do ponto de vista da implementação, utilizamos no Shock a arquitetura de camadas *ingestão, tratamento, análise e publicação*, cujas traduções em inglês são respectivamente *ingest, store, analyze* e *publish*. Essa arquitetura é apresentada na Figura 9, e a criamos para o Shock utilizando como base arquiteturas mais difundidas, como a *ingest, store, analyze* e *visualize*, utilizada na plataforma Google Cloud Platform[5], e a arquitetura *collecton tier, message queuing tier, analysis tier, in-memory data store* e *data access tier*, explicada por Psaltis (2017). No Shock, cada uma dessas camadas são classificadas como estágios de um fluxo de dados, podendo um fluxo de dados ter no máximo 4 estágios.

Diferente dos outros serviços da plataforma, as aplicações clientes podem interagir com o Shock através de uma API fornecida via o Kafka. Assim, as aplicações enviam mensagens ao Kafka utilizando um formato semelhante ao JSON, com o padrão "arg1;{arg2}", onde o primeiro *token*, "arg1", representa o nome da ação desejada, e o segundo, "arg2", os argumentos da ação. Um *handler* no Shock receberá a ação requisitada no método *handle*, e deve tratar a requisição recebida. Por exemplo, uma mensagem **ingestion;{"stream": "mystream", "shock_action": "socketIngestion"}**, deve ser enviada caso queira-se o registro de um estágio de ingestão no fluxo de dados *mystream* utilizando a estratégia *socketIngestion*.

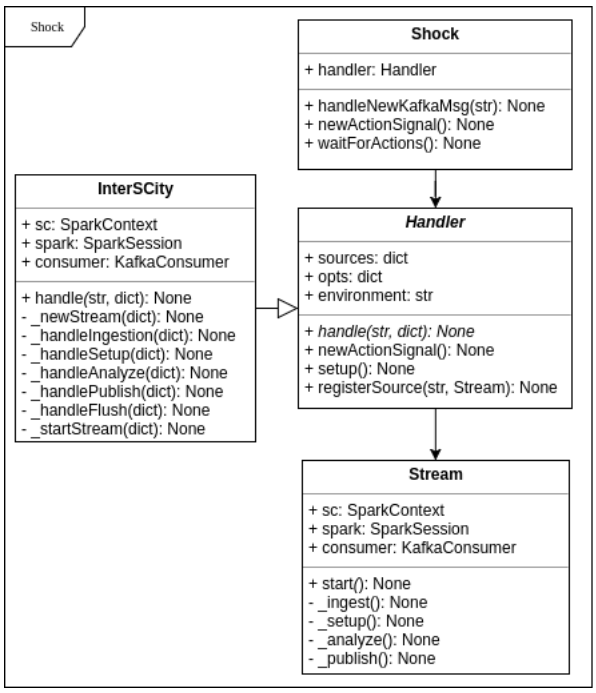

Figura 10 – Diagrama de classes do Shock.

A Figura 10 apresenta o diagrama das principais classes do Shock. O núcleo do Shock é representado pela classe Shock, que sustenta-se no uso de um *handler* que implemente o método *handle*. Atualmente, um *handler* para o InterSCity encontra-se implementado, contudo, caso seja desejado o uso de um *handler* que utilize a Arquitetura Lambda, por exemplo, basta a criação de uma nova classe que herde da classe *Handler* e que implemente o método *handle* para

[5] <https://cloud.google.com/solutions/data-lifecycle-cloud-platform>

as diferentes ações requisitadas. Por fim, uma classe *Stream* utiliza a arquitetura de camadas citada anteriormente, onde cabe ao *handler* gerenciar e interagir com os diferentes fluxos de dados criados.

```
# source: https://gitlab.com/DGuedes/shock/blob/master/shock/ingestion.py
def socketIngestion(args: dict) -> SparkDataFrame:
    spark = args.get("spark")
    host = args.get("host")
    port = args.get("port")
    return spark.readStream.format("socket").option("host", host) \
            .option("port", port).load()


def kafkaIngestion(args: dict) -> SparkDataFrame:
    spark = args.get("spark")
    topic = args.get("topic")
    brokers = args.get("brokers")
    return spark.readStream.format("kafka") \
        .option("kafka.bootstrap.servers", brokers) \
        .option("subscribe", topic).load() \
        .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Listagem 4.1 – Ingestão de dados no Shock via *socket* e Kafka.

O primeiro estágio de um fluxo de dados, a ***ingestão***, trata-se da ingestão dos dados a partir de alguma fonte, como algum tópico do Kafka, ou algum arquivo novo. Atualmente o Shock fornece três tipos diferentes de ingestão: ingestão via tópico do Kafka, ingestão via arquivo (Parquet e JSON) e ingestão via *socket*. A depender da estratégia de ingestão, alguns parâmetros são necessários na configuração - numa ingestão via Kafka, por exemplo, é necessário configurar o endereço do *broker* e os tópicos que serão utilizados. A Listagem 4.1 apresenta o código-fonte de duas estratégias de ingestão disponíveis no Shock.

```
# source: https://gitlab.com/DGuedes/shock/blob/master/shock/processing.py
def castentity(stream: DataStreamReader, args: dict) -> DataStreamReader:
    json_objects = []
    for u in ["uuid", "capability", "timestamp", "value"]:
        json_objects.append(get_json_object(stream.value, '$.'+u).alias(u))
    return stream.select(json_objects)
```

Listagem 4.2 – Tratamento de dados no Shock via *cast*.

O segundo estágio de um fluxo de dados é chamado ***preparo***, e tem o papel de ajuste e limpeza dos dados para que sejam utilizados pelos fluxos sem maiores complicações. Um exemplo típico é o uso de um estágio de preparo que faça *cast* (mudanças nos tipos de dados) de dados, quando valores estão como *string* quando são necessários valores *double*. Atualmente, o Shock apresenta somente preparo de dados via *cast*, que são essenciais no uso do InterSCity e de consumo do Kafka, apresentados na Listagem 4.2.

```
# source: https://gitlab.com/DGuedes/shock/blob/master/shock/processing.py
def streamFilter(stream: SparkDataFrame, args: dict) -> SparkDataFrame:
    query = args.get("query")
    if (query):
        return stream.where(query)
    else:
        raise('Missing required param "query"')


def mean(stream: SparkDataFrame, args: dict) -> SparkDataFrame:
    df1 = stream.selectExpr('cast(value as double) value',
            'capability', 'uuid', 'timestamp')
    df2 = df1.select(avg("value"))
    return df2
```

Listagem 4.3 – Operações de análise de dados no Shock.

O terceiro estágio de um fluxo, a ***análise***, é o estágio principal do processamento, e permite filtros, agregações e cálculos. No Shock, a etapa de análise dos dados recebe como entrada um um fluxo de dados de dados e retorna um fluxo de dados transformado. Uma aplicação que deseje utilizar a operação de filtro no fluxo *mystream* com a finalidade de filtrar os valores de *air_quality* iguais a 31, deve fazer uma requisição no Kafka com o conteúdo **processing;{"stream": "mystream", "shock_action": "streamFilter", "query": "select * from air_quality where 'value' == 31"}**. Algumas possibilidades de estágios de análise estão contidos na Listagem 4.3.

```
# source: https://gitlab.com/DGuedes/shock/blob/master/shock/sinks.py
def consoleSink(stream: StructuredStream, args: dict) -> OutputStream:
    streamName = args.get("stream")
    return stream.writeStream.outputMode('append').format('console') \
            .start()
def parquetCompleteSink(stream: StructuredStream, args: dict) ->
    OutputStream:
    streamName = args.get("stream")
    path = args.get("path") or "/analysis"
    return stream.writeStream.outputMode('complete').format('memory') \
            .queryName('analysis').start()
def memorySink(stream: StructuredStream, args: dict) -> OutputStream:
    table = args.get('table')
    if not table:
        table = 'analysis'
    stream.writeStream.outputMode('complete').format('memory')\
            .queryName(table).start()
```

Listagem 4.4 – Estratégias de publicação de resultados presentes no Shock.

O último estágio de um fluxo de dados, conhecido como estágio de ***publicação***, é o estágio de apuração do processamento, retornando os dados necessários para o cliente. O Shock atualmente permite a publicação via arquivo, memória e *console*, e após o lançamento da versão 2.2 do Spark, permitirá a publicação via Kafka. A publicação via arquivo é limitada, pois não permite a publicação em um servidor externo. A publicação via memória também não permite, mas é interessante pois o conteúdo passa a estar disponível para ser requisitado via SparkSession, permitindo consultas com sintaxe SQL. Por fim, a publicação via *console* está presente somente para fins de desenvolvimento, mas é possível que ocorra uma combinação com outras ferramentas que leiam da saída padrão do Sistema Operacional. Categorizamos no Shock o nome *sink* para as diferentes estratégias de publicação, que é o nome utilizado pelo Spark. A Listagem 4.4 apresenta as estratégias de publicação em *console*, Parquet e memória.

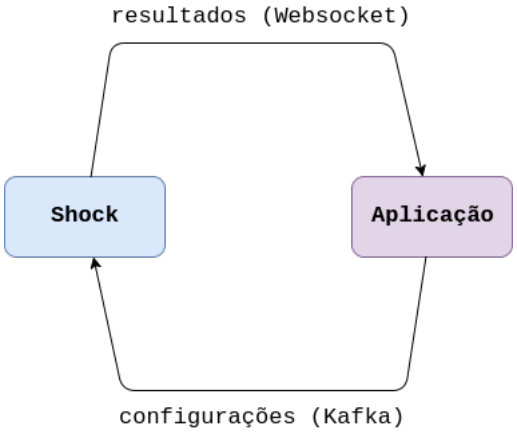

Figura 11 – Comunicação entre o Shock e aplicações.

Como alternativa às limitações de publicação dos dados, o Shock disponibiliza na API os *flushes*, que atuam como *jobs* adicionais que ingerem dados de alguma fonte (que não seja um fluxo) e publicam em outra. Essa alternativa não precisaria existir caso a publicação via Kafka já estivesse disponível no Spark, mas enquanto o lançamento da versão 2.2 não ocorre, acaba sendo uma opção válida. Nesse sentido, uma aplicação cliente do InterSCity poderia usar o Shock para realizar processamento de seus dados seguindo a interação apresentada na Figura 11. Nessa interação, a aplicação faz definições no Shock via Kafka, e o Shock retorna os resultados pelos estágios de publicação ou por WebSocket.

5 EXEMPLO DE USO

Como forma de ilustração do uso do serviço que desenvolvemos, elaboramos uma aplicação chamada Forensic[1], que interage com o Shock via Kafka, respeitando os padrões utilizados pela API. A aplicação foi feita em Elixir, uma linguagem funcional que utiliza a máquina virtual do Erlang, utilizando o *framework* Phoenix, que facilita no desenvolvimento de aplicações Web.

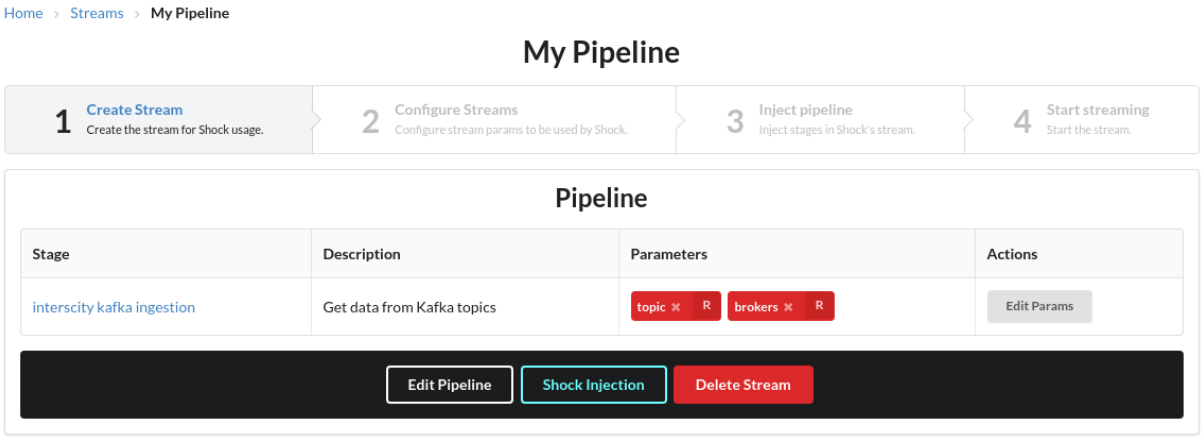

Figura 12 – Página de configuração de um fluxo. Os parâmetros obrigatórios (*topic* e *brokers*) foram configurados.

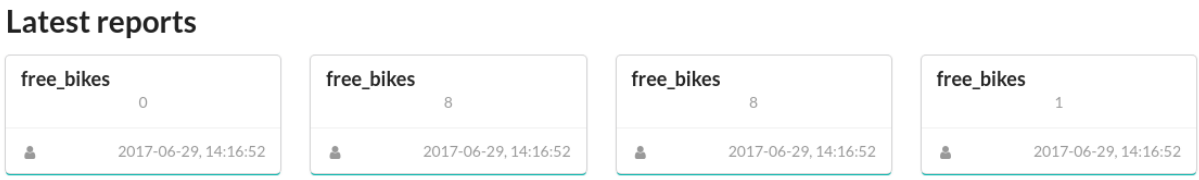

Figura 13 – Página de visualização de resultados.

O Forensic tem como principais objetivos abstrair o Shock do usuário final, e consequentemente, as ferramentas de *Big Data* (como o Spark), ao passo em que fornece um conjunto de funcionalidades que permitem ao usuário final configurar a atuação dos fluxos, semelhante a outros serviços existentes, como o Amazon AWS[2]. Um usuário que deseje interagir com o Shock e o InterSCity deve (1) configurar um fluxo novo com os parâmetros desejados; (2) criar esse fluxo no Shock; (3) injetar esse fluxo no Shock; (4) e iniciar o processamento do fluxo. O Forensic abstrai esses 4 passos, facilitando o uso das ferramentas e da plataforma, aproximando usuários finais que não tenham grande conhecimento do código-fonte do Shock. A Figura 12 apresenta a página de visualização e edição de um fluxo, e a Figura 13 a página de visualização de resultados

[1] <https://gitlab.com/DGuedes/forensic>
[2] <https://aws.amazon.com/kinesis/analytics/>

processados. Os resultados de cada dado são mostrados separadamente, onde o título representa a capacidade do recurso, e o texto centralizado o valor da respectiva capacidade.

Com o propósito de aproximar as soluções que desenvolvemos aos cenários mais reais, separamos dois casos de uso para serem resolvidos com uso do Forensic e do Shock. Em cada um desses casos ocorre uma interação entre o InterSCity, Shock e Forensic, utilizando como produtor de dados alguns coletores de São Paulo[3]. De maneira geral, os casos de uso abrangem os conceitos citados anteriormente, como o *cast* de dados, filtros, dentre outros.

5.1 CASO DE USO 1 - REGIÕES COM QUALIDADE DO AR INSATISFATÓRIA

O primeiro caso que separamos é o de regiões que apresentam qualidade do ar insatisfatória, utilizando como base os dados da CETESB[4] (Companhia Ambiental do Estado de São Paulo). A solução desse caso de uso pode ser divida em três etapas:

1. Coletar os dados;
2. Publicar os dados no InterSCity;
3. Configurar um fluxo que filtre os dados, removendo da lista os dados com boa qualidade do ar (pois queremos as regiões com qualidade insatisfatória).

O primeiro passo, de coleta dos dados, pode ser resolvido via *crawler*, para extrair e normalizar os dados coletados. Um *script* que utiliza o Mechanize já havia sido desenvolvido por contribuidores do InterSCity[5], e pôde ser reaproveitado.

O segundo passo, de publicação dos dados no InterSCity, se baseia na normalização dos dados e das requisições, para que aja uma interação com o Resource Adaptor. Adaptamos o *script* mencionado no passo anterior para que passasse a fazer requisições REST ao Resource Adaptor do InterSCity, possibilitando o registro de recursos e o envio dos dados.

```
# web/channels/alert_channel.ex
  alias Forensic.Report, as: R
  def handle_in("new_report", payload, socket) do
    params = payload |> Map.merge(%{"timestamps" => payload[:timestamps]})
             |> Map.drop([:timestamp])
    changeset = R.changeset(%R{}, params) |> Repo.insert
  end
```

Listagem 5.1 – Consumo dos dados via *websocket* caso o evento seja *new_report*.

[3] <https://github.com/lucaskanashiro/collect_sp_data>
[4] <http://sistemasinter.cetesb.sp.gov.br/Ar/php/ar_resumo_hora.php>
[5] <https://github.com/lucaskanashiro/collect_sp_data/blob/master/air_quality.rb>

O terceiro passo, de configuração do fluxo, é feito através do Forensic. Um fluxo deve ser criado na interface, e configurado para uso dos estágios *Kafka Ingestion*, *Kafka Cast* e *Filter*. O estágio de publicação é mais flexível, e qualquer estratégia disponível no Forensic serve para esse caso de uso. O Forensic apresenta uma página de alertas, que é populada via um *flush* específico apresentado na Listagem 5.1. Quando esse *flush* é acionado, o Shock organiza um *job* que lê de um repositório específico no formato Parquet, e publica no Forensic via *websocket*.

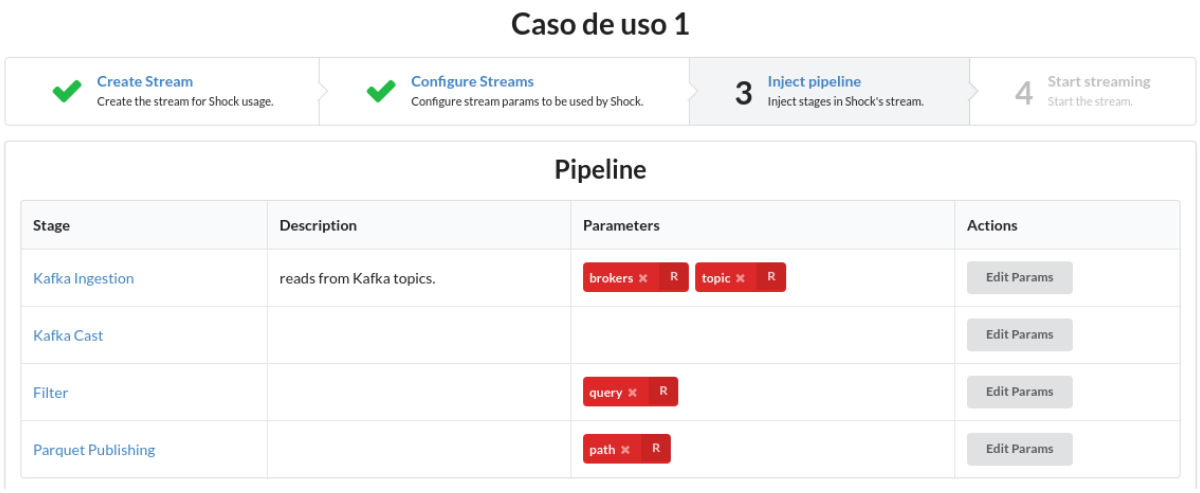

Figura 14 – Parâmetros do primeiro caso.

Tabela 2 – Parâmetros do fluxo utilizado no primeiro caso de uso.

Estágio	**Parâmetro**	**Valor**
Kafka Ingestion	*topic*	*interscity*
	brokers	*kafka:9092*
Kafka Cast	-	-
Filter	*query*	*capability == air_quality AND value != 'boa'*
Parquet Publishing	*path*	*/analysis*

Após o fluxo ser criado e os estágios selecionados, basta configurar os parâmetros e transferir as definições para o Shock. A Figura 14 apresenta uma tela do Forensic com o fluxo já configurado, utilizando os valores presentes na Tabela 2.

Por fim, basta seguir os passos de transferência que o Forensic sugere (*Create Stream, Configure Stream, Inject Pipeline e Start Streaming*), presentes no topo da Figura 14. O Shock então começará o processamento dos dados, e a próxima vez que o usuário visitar a página de alertas e requisitar atualização, aparecerão os dados insatisfatórios da qualidade do ar da cidade de São Paulo.

5.2 CASO DE USO 2 - MÉDIA DE BICICLETAS LIVRES NA REGIÃO COM COMBINAÇÃO DE FLUXOS

O segundo caso que separamos visa calcular a média de bicicletas livres em estações de bicicletas de São Paulo, utilizando como base os dados do CityBik[6]. Esse caso pode ser visto como uma versão mais complexa do caso anterior, pois deve ser feito o filtro de capacidades de bicicletas e o cálculo da média dos valores. Ainda, como mencionado, o Shock só disponibiliza 4 estágios por fluxos, e como esses requisitos carecem ao menos 5, devem ser criados dois fluxos, onde o primeiro deve ingerir dados do InterSCity, e o segundo deve ingerir os resultados do primeiro.

Utilizando os mesmos passos definidos no caso de uso anterior, para o Passo 1 desenvolvemos um *script*[7] que requisita dados à API do CityBik, via REST. Após a ingestão desses dados, para o Passo 2, o *script* se comunica com o Resource Adaptor, publicando os dados no InterSCity.

Tabela 3 – Parâmetros do primeiro fluxo.

Estágio	**Parâmetro**	**Valor**
Kafka Ingestion	*topic*	*interscity*
	brokers	*kafka:9092*
Kafka Cast	-	-
Filter	*query*	*capability == free_bikes*
Parquet Publishing	*path*	*/interscity-data/freebikes*

Tabela 4 – Parâmetros do segundo fluxo.

Estágio	**Parâmetro**	**Valor**
Parquet Ingestion	*path*	*/interscity-data/freebikes*
Mean	-	-
Memory Publishing	*table*	*avg*

Para o Passo 3, devem ser configurados dois fluxos. O primeiro deve conter os estágios *Kafka Ingestion*, *Kafka Cast*, *Filter* e *Parquet Publishing*, de modo a ingerir novos dados do InterSCity, filtrar dados não desejados e disponibilizar os dados em formato Parquet. O segundo fluxo deve conter os estágios *InterSCity Parquet Ingestion*, *Mean* e alguma forma de publicação. A ligação entre os dois fluxos ocorre quando o primeiro publica resultados em um arquivo Parquet, e o segundo ingere esses resultados como entrada. O valor dos parâmetros utilizados no primeiro fluxo estão apresentados na Tabela 3, e os parâmetros do segundo fluxo na Tabela 4. Após a configuração dos parâmetros e transferência das configurações para o Shock, semelhante ao ocorrido no caso de uso anterior, os dois fluxos começam o processamento dos dados, retornando para o Forensic a média de bicicletas disponíveis na região.

[6] <https://citybik.es/>
[7] <https://github.com/lucaskanashiro/collect_sp_data/blob/interscity_integration/citybik.rb>

6 CONSIDERAÇÕES FINAIS

As contribuições deste trabalho propiciaram ao InterSCity a possibilidade de atuação em cenários mais abrangentes, por meio de um novo serviço para processamento de seus dados. Juntamente com o novo serviço desenvolvemos o Shock, uma aplicação que abstrai e gerencia o uso de ferramentas de *Big Data*, e o Forensic, que visa demonstrar a utilização do Shock. O Forensic é necessário pois sua interface aproxima os usuários finais, que passam a poder gerenciar processamento de fluxos de dados complexos sem conhecimento profundo das tecnologias. Com a incorporação dos resultados que desenvolvemos, o InterSCity passa a poder atuar em operações com maior massa de dados e a fornecer processamento de dados para terceiros através da plataforma.

Na primeira etapa do trabalho, focamos em na concepção, desenho e planejamento do novo serviço de processamento para a plataforma. Mas nesta segunda etapa do trabalho, que começou em maio/2017, focamos em amadurecer o Shock e em definir casos de uso que pudessem ser resolvidos com a sua utilização - o que nos levou ao desenvolvimento do Forensic, que apresenta uma interface mais amigável ao usuário final. Com relação ao planejado, embora tenhamos dado foco em amadurecer o Shock e a solucionar cenários de uso, ainda assim implementamos parte das atividades planejadas relacionadas a evolução e funcionalidades, por serem importantes em cenários de uso mais reais. Foram elas:

- **Documentar a API de serviços:** Conseguimos dar maior ênfase a documentação da API do serviço desenvolvido, que podem ser conferidas num servidor específico para isso[1], desenvolvido em Sphinx;
- **Utilização dos *data frames*:** A utilização dos *data frames* possibilitou o uso de síntaxe SQL, de menor complexidade que a manipulação de *RDDs*;
- **Múltiplos fluxos de dados:** Mudamos a API de fluxos de dados para que fosse feito o uso de fluxo estruturado, que possibilitam o uso de fluxos múltiplos, adicionando ao InterSCity a possibilidade de, com apenas uma instância do ecossistema da plataforma, servir processamento de dados para vários usuários.

Além das novas funcionalidades, a partir do desenvolvimento deste TCC foram feitas outras contribuições, como (i) a criação de um novo *script* de coleta de dados, que consome a API de estações de bicicleta; e (ii) melhoria de um *script* que ingere dados sobre a qualidade do ar.

As limitações das aplicações desenvolvidas se devem principalmente aos cenários alternativos, como edição e visualização de *log* dos fluxos, que não foram desenvolvidos. Atualmente não é possível, por exemplo, a edição de um fluxo que já está em estado de processamento, a

[1] <http://shock.readthedocs.io/en/latest/index.html>

abortagem do processamento de um fluxo ou a visualização do *log* do estado de um fluxo pelo Forensic. Com base nessas limitações, definimos como os próximos passos para o Shock e Forensic a adição de cenários alternativos para os fluxos de processamento, um teste em cenário real de cidades inteligentes e a incorporação do serviço de processamento ao núcleo do InterSCity.

Referências

ARMSTRONG, J. *Making reliable distributed systems in the presence of software errors*. Tese (Doutorado) — Mikroelektronik och informationsteknik, 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 51.

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 26.

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 na página 26.

MARZ, N.; WARREN, J. *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. Manning, 2015. ISBN 9781617290343. Disponível em: <https://books.google.com.br/books?id=HW-kMQEACAAJ>. Citado 6 vezes nas páginas 9, 23, 24, 25, 27 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.

PSALTIS, A. *Streaming Data: Understanding the Real-time Pipeline*. Manning Publications Company, 2017. ISBN 9781617292286. Disponível em: <https://books.google.com.br/books?id=1yCxDAEACAAJ>. Citado na página 37.

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 29.

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 30.

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, essa 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. Isso 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.