

Lucas Moura Barrozo

**Descoberta semântica de microservices em
contêineres**

Brasil

2016

Lucas Moura Barrozo

Descoberta semântica de microservices em contêineres

Trabalho de Conclusão de Curso submetido ao Curso de Ciência da Computação para obtenção do Grau Bacharel em Ciência da Computação.

Universidade Federal de Santa Catarina - UFSC

Departamento de Informática e Estatística

Curso de Ciência da Computação

Orientador: Frank Augusto Siqueira

Coorientador: Ivan Luiz Salvadori

Brasil

2016

Lucas Moura Barrozo
Descoberta semântica de microservices em contêineres/ Lucas Moura Barrozo.
– Brasil, 2016-
107 p. : il. (algumas color.) ; 30 cm.

Orientador: Frank Augusto Siqueira

Trabalho de conclusão de curso – Universidade Federal de Santa Catarina - UFSC
Departamento de Informática e Estatística
Curso de Ciência da Computação, 2016.

1. rdf. 2. web semântica. 3. service discovery. 4. linux container. I. Frank Augusto Siqueira. II. UFSC. III. Universidade Federal de Santa Catarina. IV. Título

CDU 02:141:005.7

Lucas Moura Barrozo

Descoberta semântica de microservices em contêineres

Trabalho de Conclusão de Curso submetido ao Curso de Ciência da Computação para obtenção do Grau Bacharel em Ciência da Computação.

Trabalho aprovado. Brasil, 25 de novembro de 2016:

Frank Augusto Siqueira
Orientador

Renato Fileto
Banca 1

Hylson Vescovi Netto
Banca 2

Brasil
2016

*Este trabalho é dedicado aos meus pais,
que apesar das dificuldades, nunca mediram
esforços para apoiar minha jornada até aqui.*

Agradecimentos

Agradeço ao professor Frank por ter sido um grande orientador e pela paciência na árdua caminhada até aqui, ao Ivan, meu coorientador, que apesar das dificuldades, cedeu suas horas valiosas a mim, à universidade e corpo docente por me fornecer o conhecimento, aos amigos pelo companheirismo e diversão e à minha namorada pelo incansável carinho e incentivo.

*"If everything seems under control,
you're not going fast enough."*

(Mario Andretti, Race driver)

Resumo

A Web vem aumentando cada vez mais suas proporções, tanto em tamanho quanto em importância. De natureza igual é o crescimento da quantidade de serviços disponíveis nela, que se evidencia gradualmente conforme a arquitetura de *microservices* é adotada. Isso torna indispensável saber onde esses serviços estão localizados, seja para monitorá-los ou mesmo para acessá-los, lembrando que é típico dessa arquitetura a volatilidade. Diversas tecnologias surgiram no mercado para solucionar esse problema, principalmente as que focam no registro de serviço dinâmicos. O que essas tecnologias fazem é apenas mapear um *host* a um nome, obrigando os desenvolvedores de software a conhecer em detalhes quais serviços um determinado endereço na rede disponibiliza. Este trabalho propõe a adaptação de um desses mecanismos de descoberta, o *Hyperbahn*, alterando a forma como os serviços são registrados e consultados. Ao invés de apenas um nome, serviços poderão ser identificados por várias entradas de registro. Tais modificações darão liberdade para a ferramenta modificada ter seu uso adaptado para dar mais significado à descoberta. Significado aos serviços será alcançado utilizando alguns conceitos da Web semântica. Será estabelecido que as entradas de registro de um *microservice* representem conceitos ou propriedades de uma ontologia. Por fim, com o propósito de comparação e validação, experimentos serão realizados na solução entregue e na implementação original. Com a utilização da solução proposta espera-se proporcionar um ambiente que reforce ainda mais as características empregadas pelo uso de *microservices*.

Palavras-chaves: rdf. web semântica. microservices. service discovery. linux container.

Lista de ilustrações

Figura 1 – Modelo tradicional de descoberta de serviço	26
Figura 2 – Interpretações de uma página Web (humano x máquina)	30
Figura 3 – Classificação <i>Linked Data</i> por Berners-Lee (2006)	31
Figura 4 – Arquitetura da Web Semântica	31
Figura 5 – Exemplo de descrição RDF	32
Figura 6 – <i>Microservices</i> abrem a possibilidade de usar diferentes tecnologias	36
Figura 7 – Fluxo básico de automação de infraestrutura	37
Figura 8 – Pode-se direcionar o escalamento apenas nos serviços necessários	38
Figura 9 – Entrada DNS mapeada para um balanceador de carga	42
Figura 10 – Ringpop como uma camada da aplicação	47
Figura 11 – Mecanismo de <i>pings</i> do protocolo de <i>membership</i>	48
Figura 12 – Processo de disseminação de informação do protocolo <i>Gossip</i> SWIM	49
Figura 13 – Intervalo dos números inteiros em um <i>consistent hashing ring</i>	50
Figura 14 – Mapeamento das instâncias no <i>consistent hashing ring</i>	50
Figura 15 – Particionamento do <i>keyspace</i> entre as instâncias no <i>consistent hashing ring</i>	50
Figura 16 – Mapeamento de uma entidade no <i>consistent hashing ring</i>	51
Figura 17 – Rebalanceamento do <i>keyspace</i> em caso de falha no <i>consistent hashing ring</i>	51
Figura 18 – Exemplo de encaminhamento de requisições no Ringpop	52
Figura 19 – Fluxo de uso da solução proposta	55
Figura 20 – Ontologia de exemplo para o estudo de caso	62
Figura 21 – Estrutura de <i>microservices</i> presente no estudo de caso	62

Lista de tabelas

Tabela 1 – Estudo de caso - Entradas de registro no Hyperbahn	63
Tabela 2 – Média do tempo(ms) das requisições de registro	66
Tabela 3 – Média do tempo(ms) das requisições de 100 nomes	67
Tabela 4 – Média do tempo(ms) de resposta para obter todas as entradas de registro cadastradas	68

Listagens

Listagem 2.1 – Exemplo RDF/XML	33
Listagem 6.1 – Código executados por clientes para se registrar no mecanismo de descoberta de serviço	56
Listagem 6.2 – Código que lida com requisições de registro	57
Listagem 6.3 – <i>TagDiscovery</i> : parâmetros - <i>Thrift</i>	58
Listagem 6.4 – <i>TagDiscovery</i> : resposta - <i>Thrift</i>	58
Listagem 6.5 – <i>TagDiscovery</i> : serviço - <i>Thrift</i>	58
Listagem 6.6 – Processo de descoberta baseada em <i>tags</i>	59
Listagem 6.7 – Obtendo membros “vivos” do Ringpop	60
Listagem 6.8 – Obtendo entradas de registro cadastradas em todos os nodos pertencentes ao <i>cluster</i> do Hyperbahn	60
Listagem 6.9 – Obtendo entradas de registro cadastradas em um nodo	61

Lista de abreviaturas e siglas

API	<i>Application Programming Interface</i>
HTTP	<i>HyperText Transfer Protocol</i>
RDF	<i>Resource Description Framework</i>
TCP	<i>Transport Control Protocol</i>
SWIM	<i>Scalable Weakly-consistent Infection-style Process Group Membership Protocol</i>
DNS	<i>Domain Name System</i>
REST	<i>Representational State Transfer</i>
TTL	<i>Time to Live</i>

Sumário

1	INTRODUÇÃO	23
1.1	Objetivo geral	25
1.2	Objetivos específicos	25
1.3	Justificativa	26
1.4	Organização do texto	27
2	WEB SEMÂNTICA	29
2.1	Fundamentos	29
2.1.1	Linked data	30
2.2	Tecnologias para Web semântica	31
2.2.1	RDF	32
2.2.2	Outras tecnologias	33
3	MICROSERVICES	35
3.1	Componentização via serviços	35
3.2	Independência	36
3.3	Heterogeneidade tecnológica	36
3.4	Automação de infraestrutura	37
3.5	Resiliência	37
3.6	Escalabilidade	38
3.7	Alinhamento organizacional	38
4	DESCOBERTA DE SERVIÇO	41
4.1	DNS	41
4.2	Registro dinâmico de serviços	42
4.2.1	Zookeeper	42
4.2.2	Consul	43
4.2.3	Eureka	43
4.2.4	Hyperbahn	44
5	HYPERBAHN	45
5.1	Tecnologias	46
5.1.1	TChannel	46
5.1.2	Ringpop	47
5.1.2.1	Protocolo de Membership	48
5.1.2.2	Consistent hashing ring	49

5.1.2.3	Forwarding	52
6	PROPOSTA	53
6.1	Incorporação de semântica	54
6.2	Implementação	56
6.2.1	Registro de serviço	56
6.2.2	Consulta de serviço baseada em múltiplos nomes	58
6.2.3	Obtenção de todas as entradas de registro	60
6.3	Estudo de Caso	62
7	EXPERIMENTOS E RESULTADOS	65
7.1	Experimentos	65
7.2	Resultados	66
8	CONCLUSÃO	69
8.1	Trabalhos futuros	70
	REFERÊNCIAS	71
9	APÊNDICES	73
9.1	APÊNDICE A - Códigos fonte	73
9.2	APÊNDICE B – Artigo Descoberta Semântica de Microservices em Contêineres	91

1 Introdução

A importância e evolução da Web está cada vez mais inegável hoje. Passou de um extenso, quase que inacabável, repositório para textos e imagens para também se tornar uma grande provedora de serviços (MCILRAITH; SON; ZENG, 2001). Esses serviços, que foram originalmente criados para auxiliar na comunicação homem e máquina, têm como principal finalidade a comunicação de aplicações pela internet como meio de facilitar a interoperabilidade na troca de informações. De modo geral, esses serviços podem ser consumidos tanto por pessoas através de aplicações ou mesmo por outros serviços.

Juntamente com a busca incessante por melhores maneiras de se criar um software, os serviços web se acomodaram em diversas arquiteturas, sendo uma delas a arquitetura de micro serviços. Os *microservices* são pequenos serviços autônomos que trabalham em conjunto (NEWMAN, 2015).

Desde então, a adoção de micro serviços está em constante expansão, alterando a forma como as aplicações são construídas, transformando as tradicionais aplicações monolíticas em soluções que miram coesão e desacoplamento.

A última década trouxe grandes mudanças para a tecnologia da informação em diversas frentes. Dispositivos móveis, proliferação de dados, virtualização e *cloud computing* são apenas alguns dos grandes avanços. Como resultado temos a complexidade dos sistemas computacionais crescendo de forma rápida e quase que totalmente sem freios (STUBBS; MOREIRA; DOOLEY, 2015). O típico sistema Web agora mantém múltiplos componentes: um web *front-end*, um gerenciador de acesso, *workers* assíncronos, plataforma de *analytics*, sistema de logs, etc. A arquitetura de micro serviços substitui a monolítica por um sistema distribuído de serviços leves, independentes e de propósito único, o que torna os sistemas cada vez mais granulares, impactando diretamente na quantidade de serviços disponíveis.

Apesar da forma simples e objetiva com que conseguimos descrever a arquitetura de micro serviços, não conseguimos ter a mesma simplicidade na prática, que acaba se mostrando desafiadora. A simples tarefa de implantação, que pode ser realizada trivialmente na arquitetura monolítica, se torna extremamente complexa nesses novos moldes. Enquanto a implantação e distribuição de micro serviços pode ser simplificada utilizando *linux containers*, pouco eles fazem para solucionar o problema de comunicação entre serviços, o que se torna um problema bem evidente quando a aplicação começa a tomar grandes proporções. Em um ambiente regido pela arquitetura de micro serviços sabemos que uma instância redundante pode ser criada ou destruída sob demanda e que serviços podem não estar disponíveis em um determinado momento, portanto serviços consumidores precisam de um mecanismo para localizar serviços produtores em tempo real.

Diversas tecnologias surgiram no mercado para solucionar esse problema, principalmente as que focam no registro de serviço dinâmicos. Muitas dessas tecnologias apenas mapeiam um endereço de rede a um nome identificador, o que agrega muito pouco na hora de descrever suas características funcionais dos serviços disponibilizados.

1.1 Objetivo geral

Este trabalho consiste em realizar o estudo e aprimoramento de uma ferramenta já existente de descoberta de serviços e roteamento para sistemas distribuídos de larga escala, habilitando a descoberta de serviço de forma semântica ao invés da tradicional.

Existem diversas ferramentas presentes no mercado que implementam a solução para a descoberta de serviço tradicional. A proposta é, ao invés de criar mais uma ferramenta concorrente, aproveitar toda a estrutura já existente, que já foi validada no ambiente de produção, e aprimorá-la alterando a forma com que serviços são consultados e como se registram.

Tradicionalmente a consulta e registro de um serviço é baseada em um nome identificador, que por sua vez é uma abordagem um tanto quanto simples para se descrever o que um determinado *microservice* faz. O intuito é dar significado à descoberta de serviço e retirar a responsabilidade do desenvolvedor de ter que conhecer as funcionalidades que *microservice* entrega.

1.2 Objetivos específicos

Os objetivos gerais do trabalho podem ser alcançados através dos seguintes objetivos específicos:

- Possibilitar a descoberta de serviço não somente por um nome identificador.
- Aprimorar uma ferramenta já existente de modo a habilitar o seu uso de forma semântica;
- Estabelecer uma forma de usar o registro para descrever as características funcionais de um *microservice*;
- Reduzir o acoplamento entre serviços consumidores e produtores, diminuindo o impacto gerado pela evolução dos sistemas;

1.3 Justificativa

A quantidade de serviços disponíveis na *Web* está aumentando cada vez mais conforme as empresas optam por esse tipo de tecnologia (ELGAZZAR; HASSAN; MARTIN, 2010). Esse comportamento se evidencia ainda mais com a adoção da arquitetura de *microservices*, que reforça a modularidade e independência. Com isso, se torna indispensável saber onde esses serviços estão localizados, seja para monitorá-los ou mesmo para acessá-los.

No Capítulo 4, seção 4.2 serão apresentadas algumas tecnologias em evidência, no mercado. O que elas fazem, na verdade, é descobrir instâncias mapeadas para nomes predefinidos pelo desenvolvedor, como na Figura 1.

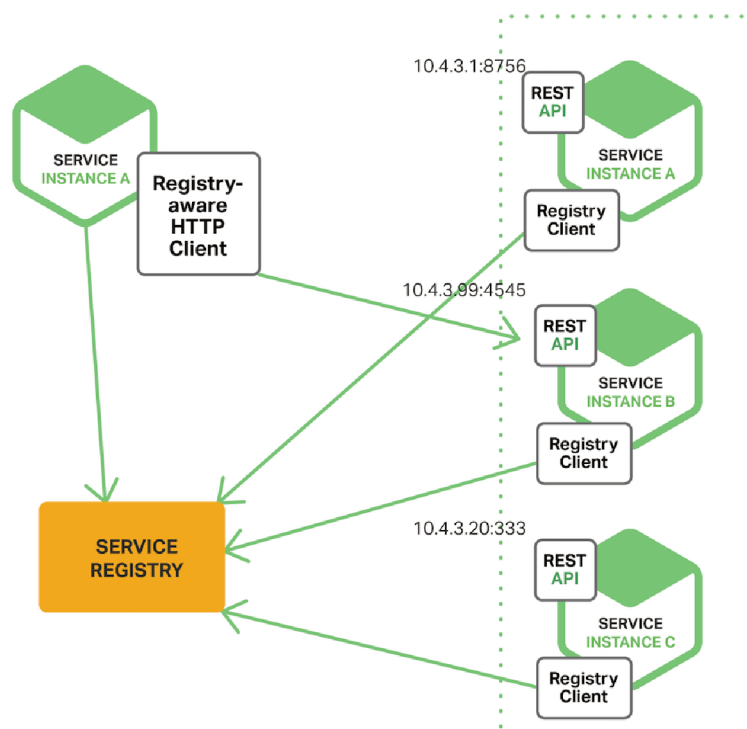


Figura 1 – Modelo tradicional de descoberta de serviço

Fonte: (RICHARDSON, 2015)

Essa abordagem resulta em diversos problemas, entre eles, exige que a equipe de desenvolvimento conheça quais os serviços que um determinado *microservice* disponibiliza e torna, em muitos dos casos, extremamente difícil mover um conjunto de serviços ou funcionalidades de um *microservice* para outro.

1.4 Organização do texto

Este trabalho está organizado da seguinte forma:

- **Capítulo 1 - *Introdução***: apresenta a contextualização e descreve o problema abordado pelo trabalho, além de apresentar a motivação e justificativa. O capítulo também descreve os objetivos do trabalho.
- **Capítulo 2 - *Web semântica***: apresenta conceitos da Web semântica, além de seus fundamentos. Descreve o formato de dados RDF, que constitui uma das tecnologias utilizadas por este trabalho.
- **Capítulo 3 - *Microservices***: apresenta os conceitos da arquitetura de *microservices*.
- **Capítulo 4 - *Descoberta de serviço***: apresenta algumas das tecnologias de descoberta de serviços que estão sendo utilizadas no mercado. Descreve o uso do *DNS* e o uso de tecnologia de *Registro dinâmico de serviços* como o *Zookeeper*, *Consul* e *Eureka*.
- **Capítulo 5 - *Hyperbahn***: apresenta a tecnologia de registro dinâmico de serviços que será aprimorada nesse trabalho. Descreve com detalhes como a ferramenta funciona, assim como as tecnologias que ela utiliza, como o *TChannel* e o *Ringpop*. Também apresenta o *Consistent hashing ring*.
- **Capítulo 6 - *Proposta***: apresenta a proposta desse trabalho. Descreve com detalhes o que irá ser feito nesse trabalho. É dividido em três partes, a primeira demonstrando como incorporar semântica, a partir da ferramenta resultante da solução proposta. A segunda descreve o processo de implementação, mostrando o que foi modificado na implementação original e o que há de novo. Por fim, é apresentado um estudo de caso, demonstrando como a solução deve ser utilizada.
- **Capítulo 7 - *Experimentos e resultados***: apresenta os experimentos realizados, comparando a implementação original do Hyperbahn com a implementação proposta. É dividida em duas partes. A primeira descreve em detalhes como foi feita a configuração do ambiente onde os experimentos foram executados, assim como quais tecnologias foram utilizadas no processo. A segunda apresenta os resultados e análises dos experimentos.
- **Capítulo 8 - *Conclusão***: apresenta a conclusão do trabalho com os objetivos alcançados. Também indica possíveis trabalhos futuros.

2 Web semântica

Grande parte do conteúdo disponibilizado na Web é destinado ao consumo humano, impossibilitando o processamento e interpretação automáticos por parte de computadores. Assim, o significado das informações publicadas não é considerado em tarefas comuns executadas na Web, como a busca, por exemplo, resultando em ambiguidade e baixa relevância (FILHO, 2009).

Segundo Filho (2009 apud BERNERS-LEE; HENDLER; LASSILA, 2001, p.29), o propósito da Web Semântica é decorar as informações publicadas na Web através de anotações semânticas formais, o que minimiza as limitações da Web como a conhecemos atualmente. Isso faz com que sintaxe e semântica passem a compartilhar o mesmo grau de importância nos processos de publicação, busca e recuperação de informações.

2.1 Fundamentos

De acordo com Salvadori (2015 apud BERNERS-LEE; HENDLER; LASSILA, 2001, p.46), a Web Semântica é uma extensão da Web atual onde os dados possuem um significado associado, criando a visão de uma rede de informações interligadas, o que possibilita que pessoas e computadores trabalhem de forma cooperativa. Em outras palavras, a Web Semântica retira dos seres humanos a exclusividade de interpretação dos dados (SALVADORI, 2015).

Tradicionalmente, os documentos publicados são formados por um título, alguns parágrafos de texto, frequentemente com imagens ou vídeos e alguns links para direcionar a outros documentos como citado por Salvadori (2015 apud W3C, 2013, p.46). Essas informações só podem ser compreendidas por humanos. Já os computadores estão limitados a interpretar esses dados como textos, links e imagens sem sentido nenhum, como ilustrado na Figura 2.

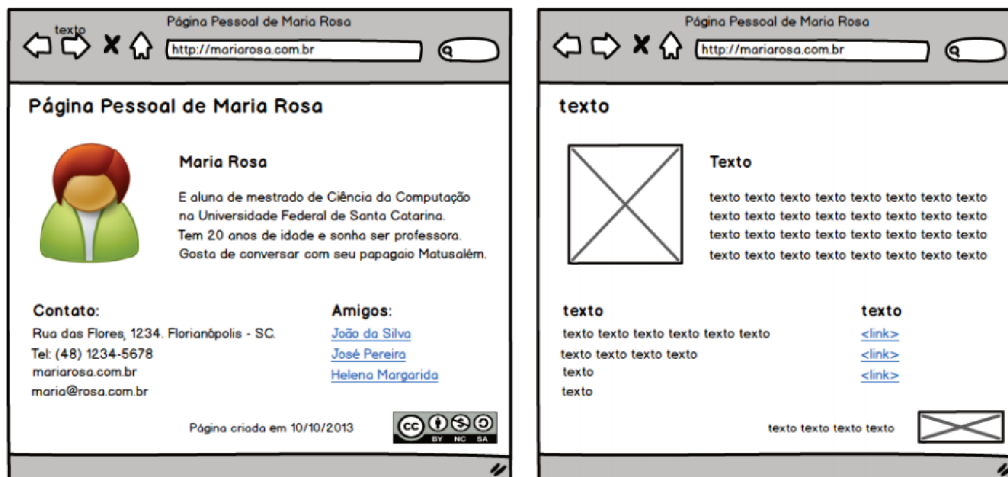


Figura 2 – Interpretações de uma página Web (humano x máquina)

Fonte: (SALVADORI, 2015)

2.1.1 Linked data

Conforme Salvadori (2015 apud BIZER; HEATH; BERNERS-LEE, 2009, p.47), a Web em seu início era denominada de “Web de Hipertextos”, onde as informações eram somente em formato digital e poderiam conter textos, imagens e multimídia (som e vídeo), permitindo ligação a outros textos por hiperlinks. Em seguida, evoluiu para “Web de Dados”, onde as informações publicadas eram melhor estruturadas a fim de que os agentes de software pudessem interpretá-las. Esta estrutura é feita através da descrição de suas prioridades e conceitos, utilizando um meio de tecnologia padronizada, com o objetivo de melhor integrar e reutilizar dados, que é a base do *Linked Data*.

Abaixo seguem as quatro regras criadas por Berners-Lee (2006) para possibilitar o reuso de informações:

- Identificar recursos através de URIs, com um identificador único de recurso;
- Associar um endereço HTTP aos recursos, permitindo que sejam localizados na Web;
- Estruturar os dados com tecnologias padronizadas e recomendadas para Web Semântica, além de estabelecer contextos sobre o relacionamento entre os recursos;
- Interligar diferentes recursos.

Além de definir as regras citadas, Berners-Lee (2006) criou uma classificação por número de estrelas para avaliar a qualidade dos dados publicados na Web, conforme mostrado na Figura 3.

- ★ Independente de formato, licença aberta (dados públicos).
- ★★ Informação em formato legível para máquinas.
- ★★★ Formato não proprietário e legível para máquinas.
- ★★★★ Informação estruturada com tecnologias padronizadas.
- ★★★★★ Relacionar informações e estabelecer contextos.

Figura 3 – Classificação *Linked Data* por Berners-Lee (2006)

Fonte: (SALVADORI, 2015)

2.2 Tecnologias para Web semântica

Como visto anteriormente, a estruturação dos dados e sua padronização é indispensável no sucesso da Web Semântica (SALVADORI, 2015). Conforme recomenda a W3C (*World Wide Web Consortium*), a Figura 4 (A), apresenta a arquitetura de Web Semântica dentro dos padrões requeridos. Disposta em camadas, a arquitetura facilita a aplicação parcial, sem a necessidade da implementação total da Web Semântica (SALVADORI, 2015 apud FILHO, 2009, p.47). Já na Figura 4 (B), as tecnologias JSON e JSON-LD e vocabulários de descrição foram acrescentados.

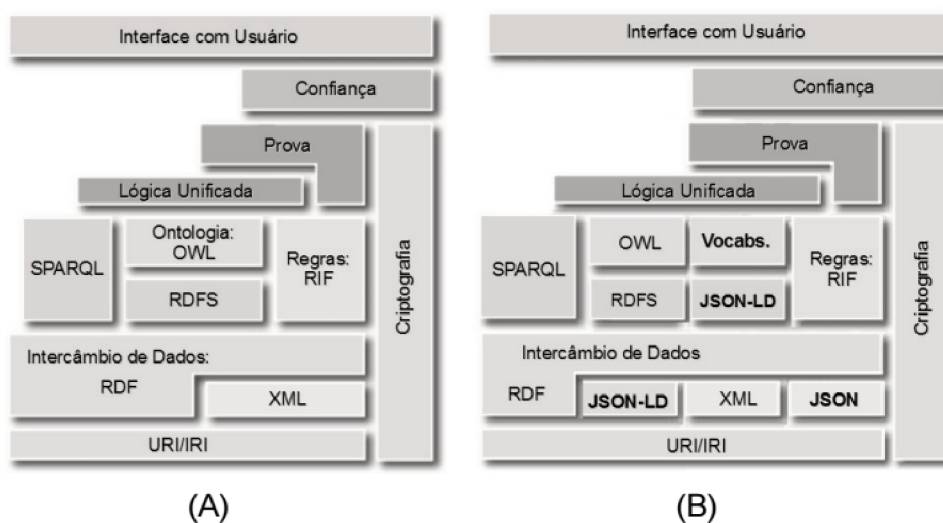


Figura 4 – Arquitetura da Web Semântica

Fonte: (SALVADORI, 2015)

2.2.1 RDF

O RDF (*Resource Description Framework*) é uma linguagem para representar informações na Web. Foi projetado para representar metadados sobre recursos da Web, por exemplo, o título, autor e a data de modificação de uma página na Web. Pode também ser utilizado para representar informações sobre coisas que podem ser identificadas na rede, mesmo quando não podem ser de fato recuperadas da Web, como, por exemplo, informações sobre produtos de uma loja online (preços, especificações, disponibilidade de entrega, etc.) (W3C, 2004).

O RDF é baseado na ideia de identificar recursos utilizando URIs, e descrever esses recursos através de propriedades e valores. A descrição de um recurso através de RDF pode ser vista com um grafo orientado, onde o sujeito e objeto são nodos e o predicado expressa seu relacionamento (SALVADORI, 2015). A Figura 5 exemplifica a descrição RDF de uma pessoa.



Figura 5 – Exemplo de descrição RDF

Fonte: (W3C, 2004)

A descrição RDF é construída no formato XML, e pode estar presente juntamente com o recurso ou em arquivo separado (Listagem 2.1). O Recurso e cada uma de suas propriedades (predicados) devem ser identificado por uma URI, enquanto valores podem ser URIs ou literais.

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4     xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#"
5 >
6   <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
7     <contact:fullName>Eric Miller</contact:fullName>
8     <contact:mailbox rdf:resource="mailto:em@w3.org"/>
9     <contact:personalTitle>Dr.</contact:personalTitle>
10  </contact:Person>
11 </rdf:RDF>
```

Listagem 2.1 – Exemplo RDF/XML

2.2.2 Outras tecnologias

Apesar de existirem outras tecnologias para Web semântica, como por exemplo o JSON-LD, estas não serão abordadas por estarem fora do escopo do presente trabalho.

3 Microservices

O termo *microservices* difundiu-se nos últimos anos para representar uma nova forma de projetar aplicações como um conjunto de serviços autônomos.

Sucintamente, a arquitetura de *microservices* é uma abordagem que visa desenvolver uma aplicação como um conjunto de pequenos serviços, cada um rodando em seu próprio processo e se comunicando de forma simples (NEWMAN, 2015). Esses serviços são construídos ao redor das necessidades do negócio e são implantados independentemente. Nesse tipo de arquitetura a heterogeneidade reina, de tal modo que serviços podem ser escritos em diferentes linguagens e podem utilizar diferentes tecnologias de armazenamento (FOWLER; LEWIS, 2014).

3.1 Componentização via serviços

Um dos pontos chave de sistemas distribuídos e de arquiteturas orientadas a serviço é que criamos a possibilidade de reusar uma determinada funcionalidade. Na arquitetura de *microservices* uma funcionalidade pode ser consumida de diferentes formas e para diversas finalidades (NEWMAN, 2015).

Quando falamos em componentes podemos atribuir a definição de que são unidades de software independentes, substituíveis e atualizáveis (FOWLER; LEWIS, 2014). Tradicionalmente temos com o mesmo propósito as bibliotecas de software, que também podem ser definidas como componentes ligados à aplicação e utilizados através de chamadas de função. A arquitetura de *microservices*, por sua vez, tem como principal maneira de componentização, dividir a aplicação em diversos serviços, os quais se comunicam através de mecanismos como uma requisição Web ou chamada de procedimento remoto.

Uma das principais razões de se utilizar serviços como componentes é que serviços são implantáveis independentemente. Tomamos como exemplo uma aplicação que faz o uso de diversas bibliotecas em um único processo. Uma simples mudança pode ter como consequência o reinício total da aplicação. Diferentemente, em uma aplicação dividida em múltiplos serviços, a mudança em um serviço faz com que apenas o serviço alterado seja reiniciado.

Outra razão de se usar serviços como componentes é eliminar o alto acoplamento entre componentes.

A utilização de serviços nesses moldes também têm os seus pontos fracos, principalmente quando falamos em esforço. A comunicação entre componentes é mais custosa e a realocação de responsabilidades entre componentes, ou qualquer outra mudança de

comportamento, se torna muito mais difícil.

De modo geral, um serviço pode ser formado por múltiplos processos que serão desenvolvidos e implantados em conjunto, como por exemplo uma aplicação e seu sistema de armazenamento exclusivo.

3.2 Independência

Microservices são entidades separadas. Podem ser implantadas como serviços completamente isolados ou mesmo podem ser o único processo em execução em um sistema operacional.

Toda comunicação entre serviços é feita através da rede, o que reforça o baixo acoplamento entre serviços. Esses serviços têm a liberdade de serem mudados independentemente e serem implantados sem a necessidade dos serviços consumidores terem que mudar. A independência diminui drasticamente se existe muito acoplamento entre o serviço consumidor e o provedor, além de exigir uma coordenação mais apurada entre os serviços quando mudanças são executadas (NEWMAN, 2015).

3.3 Heterogeneidade tecnológica

Em um sistema composto por múltiplos serviços pode-se decidir fazer uso de diferentes tecnologias para cada um. Isso possibilita a escolha da ferramenta correta para o trabalho certo (Figura 6), ao invés da abordagem tradicional de se usar apenas uma tecnologia, que muitas vezes é a de menor desempenho (NEWMAN, 2015).

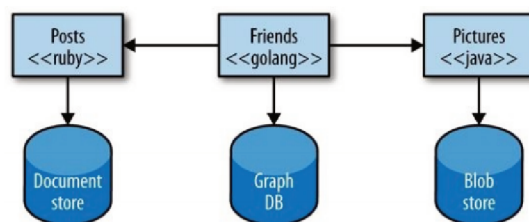


Figura 6 – *Microservices* abrem a possibilidade de usar diferentes tecnologias

Fonte: (FOWLER; LEWIS, 2014)

Se alguma área do sistema necessita de melhoramentos, especificamente de desempenho, podemos optar pelo uso de uma diferente tecnologia para atingir os níveis de desempenho desejados. Podemos também mudar como os dados são armazenados em uma parte diferente do sistema.

Também é possível fazer o uso e experimentar novas tecnologias com menos risco. Em uma aplicação monolítica, a praticidade de se adotar uma nova tecnologia pode ter um grande impacto em outras áreas do sistema. Em *microservices* se tem mais liberdade de escolha. Por exemplo, uma nova linguagem de programação pode ser utilizada em um serviço que tenha o papel menos evidente no sistema final, mitigando possíveis impactos negativos.

3.4 Automação de infraestrutura

As técnicas de automação de infraestrutura evoluíram consideravelmente nos últimos anos - a evolução da computação em nuvem e o AWS (*Amazon Web Services*) em particular reduziram em muito a complexidade operacional de se construir e implantar (Figura 7) *microservices*, principalmente utilizando as práticas de integração contínua (FOWLER; LEWIS, 2014).

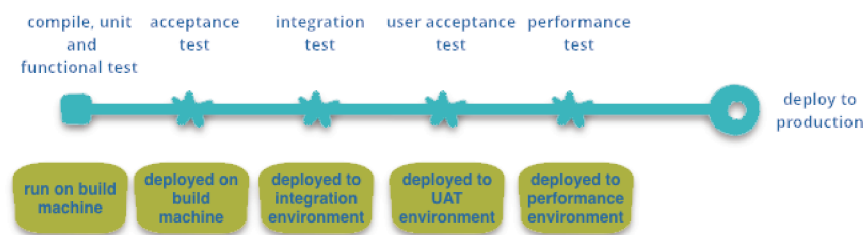


Figura 7 – Fluxo básico de automação de infraestrutura

Fonte: (FOWLER; LEWIS, 2014)

Alguns pontos que se destacam quando se fala em integração contínua são a automatização de testes e a implantação automatizada de serviços.

3.5 Resiliência

Como consequência de se usar serviços como componentes, a aplicação necessita ser tolerante a falhas. Devido ao estilo de comunicação imposto pela arquitetura, a qualquer momento um ou mais serviços podem estar indisponíveis, portanto, as aplicações clientes devem responder a falhas de forma rápida e transparente. Essa é uma das desvantagens comparado a aplicações monolíticas (FOWLER; LEWIS, 2014).

Sabendo que os serviços podem falhar a qualquer momento, é de extrema importância detectar qualquer indisponibilidade rapidamente, e se possível, restaurá-los de forma automática.

Se um componente do sistema falhar, e se essa falha não ocasionar novas falhas em cascata, o problema pode ser isolado e o resto do sistema pode continuar funcionando. No caso de aplicações monolíticas se algo falhar tudo para de funcionar. Diferentemente, na arquitetura de *microservices* é possível criar sistemas capazes de se recuperarem de falha total ou mesmo sacrificar algumas funcionalidades em prol da disponibilidade (NEWMAN, 2015). A rede pode e vai falhar, como máquinas também irão. É vital saber como lidar com cada tipo de falha e qual impacto elas terão no usuário final.

3.6 Escalabilidade

Em uma aplicação composta por vários *microservices*, pode-se escalar apenas os serviços que realmente necessitam ser escalados, permitindo outras partes do sistema rodarem em hardware menos poderoso, como na Figura 8, diferentemente de aplicações monolíticas, onde para se escalar a aplicação é necessário escalá-la como um todo (NEWMAN, 2015).

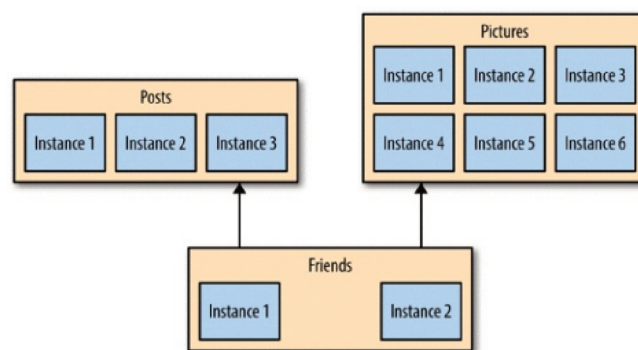


Figura 8 – Pode-se direcionar o escalamento apenas nos serviços necessários

Fonte: (NEWMAN, 2015)

Gilt, uma empresa online de vendas de roupas, adotou a arquitetura de *microservices* exatamente por essa razão. Em 2007 mantinham um aplicação monolítica escrita em Rails e em 2009 o sistema não conseguiu lidar com a carga colocada sobre ele. Extraindo as partes mais importantes do sistema, os picos de tráfego conseguiram ser contidos, e hoje a empresa mantém mais de 450 *microservices* rodando de forma distribuída (NEWMAN, 2015).

3.7 Alinhamento organizacional

Quando falamos de desenvolvimento de software são clássicos os problemas relacionados a grandes times e bases de código extensas. Esses problemas se tornam muito mais sérios quando os times são distribuídos. Também sabemos que times menores, mantendo

bases de código menores, são mais produtivos. Os *microservices* permitem um melhor alinhamento da arquitetura com o organograma da empresa, ajudando a minimizar o número de pessoas trabalhando em um determinado serviço, para manter sempre o ponto certo de produtividade. Também se pode mudar qual time é “dono” de qual serviço, além de manter as pessoas próximas trabalhando sempre na mesma base de código (NEWMAN, 2015).

4 Descoberta de serviço

Inevitavelmente, a partir do momento que exista uma quantidade considerável de *microservices* disponíveis, será essencial saber onde eles estão localizados. Talvez apenas seja necessário saber quais serviços estão localizados em um determinado ambiente para poder monitorá-los ou mesmo tornar o processo de desenvolvimento mais fácil. De modo geral, para todos esses casos que estão no escopo do descobrimento de serviço e na arquitetura de *microservices* já existem algumas soluções disponíveis.

Todas as soluções listadas a seguir oferecem algum mecanismo para o serviço se registrar, assim como para achar um serviço já registrado.

4.1 DNS

O DNS nos deixa associar um nome com o endereço de IP de uma ou mais máquinas. Pode-se decidir, por exemplo, que um determinado serviço sempre será encontrado em “exemplo.ufsc.br”. Teria-se então este ponto de entrada para o endereço de IP do *host* que roda esse serviço, ou mesmo para um balanceador de carga que distribui a carga entre as instâncias presentes no sistema. Com isso, atualizar essas entradas passa a ser parte do processo de implantação dos serviços (NEWMAN, 2015).

O DNS também pode ser utilizado para lidar com instâncias em diferentes ambientes: desenvolvimento e produção. Um método bastante utilizado é usar um modelo pré-definido como “nome do serviço-ambiente.ufsc.br”, gerando entradas como “exemplo-dev.ufsc.br” e “exemplo-prod.ufsc.br”. Múltiplos ambientes também podem ser resolvidos mantendo um servidor DNS próprio. Dependendo de onde é feito o *lookup*, o domínio “exemplo.ufsc.br” pode ser resolvido para um *host* diferente.

Uma série de vantagens podem ser alcançadas utilizando o DNS, a principal delas é que por ser um padrão bem consolidado é suportado por diversas tecnologias. Infelizmente na arquitetura de *microservices* os serviços são altamente voláteis, causando dificuldades quando é necessário atualizar muitas entradas. Além disso, a própria especificação do DNS também pode ser causa de alguns problemas.

As entradas de DNS possuem a propriedade *time to live* (TTL). É dessa forma que clientes definem uma entrada como recente. A mudança do *host* ao qual uma entrada DNS se refere implica na atualização dessa entrada, porém os clientes que já fizeram o *lookup* (consulta) anteriormente a essa atualização ainda continuarão se referindo ao IP antigo até que o valor do TTL expire. Entradas DNS podem estar em *cache* em diversos lugares da rede, e em quanto mais locais elas estiverem em *cache*, mais velhas as entradas

podem ser. Uma forma de se driblar esse problema é ter a entrada DNS mapeada para um balanceador de carga (Figura 9), o qual encaminha as requisições para as instâncias do serviço. Instâncias antigas podem ser removidas do balanceador de carga caso uma nova instância venha a ser implantada.

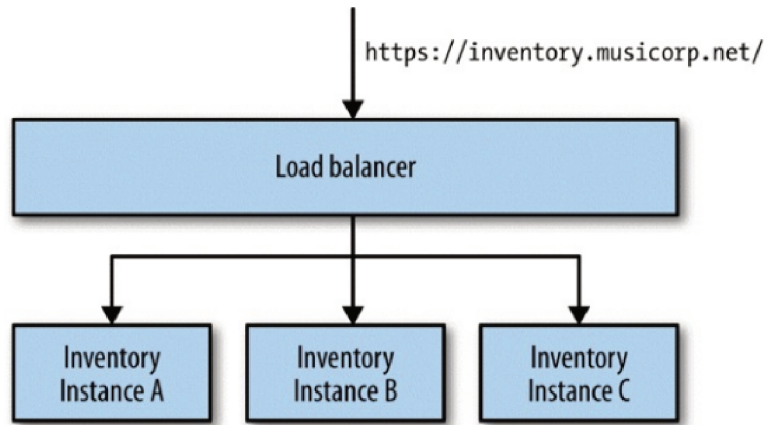


Figura 9 – Entrada DNS mapeada para um balanceador de carga

Fonte: Newman (2015, p. 399)

4.2 Registro dinâmico de serviços

As desvantagens do DNS em um ambiente dinâmico, que a arquitetura de *microservices* proporciona, levaram à criação de diversos sistemas alternativos, que em sua maioria disponibilizam uma forma dos serviços se registrarem em um nó central, o qual em troca oferece um meio para pesquisar os serviços registrados. Alguns desses sistemas alternativos serão apresentados nas subseções seguintes.

4.2.1 Zookeeper

O Zookeeper originalmente foi desenvolvido como parte do projeto Hadoop. É utilizado para diversos casos de uso, incluindo o gerenciamento de configuração, sincronização de dados entre serviços, eleição de líder, filas de mensagem e *naming service*.

Como muitos dos sistemas similares, Zookeeper deve rodar em um número de nodos no *cluster* para proporcionar diversas garantias, que giram em torno da replicação de dados entre os nodos, fazendo com que o sistema permaneça consistente se um dos nodos falhar.

Em seu núcleo, Zookeeper fornece um contexto (*namespace*) hierárquico para armazenar informação. Clientes podem inserir, mudar, observar e consultar nodos nessa hierarquia. Pode-se, por exemplo, armazenar informações sobre onde os serviços estão

localizados nessa estrutura, e como o cliente dessa estrutura pode ser notificado quando essa informação muda. Normalmente, o Zookeeper é utilizado para armazenamento de configurações gerais de um determinado serviço.

4.2.2 Consul

Na mesma linha do Zookeeper, o Consul dá suporte tanto ao gerenciamento de configuração quanto para descoberta de serviço. Ele também expõe uma interface HTTP para descoberta de serviço e provê um servidor DNS pronto para uso.

Se parte do sistema usa o DNS como alternativa de registro e descoberta, não será necessário mudar muita coisa para começar a utilizar o Consul.

A ferramenta também oferece outras competências, como a possibilidade de periodicamente verificar o estado dos nodos registrados, fazendo com que o Consul possa ser utilizado no lugar de outras ferramentas dedicadas ao monitoramento de sistemas.

É utilizada uma interface HTTP RESTful para expor todas as funcionalidades da ferramenta, que incluem registro de serviços, busca no banco de chave-valor ou para monitorar o sistema. Isso torna a integração com diferentes tecnologias fácil e direta.

O Consul é basicamente dividido em duas partes. Uma delas é a sua base, o Serf¹, que controla toda a detecção dos nodos no *cluster* e gerencia falhas e notificações. A segunda parte é responsável pela descoberta de serviço e pelo gerenciamento de configuração.

4.2.3 Eureka

Eureka é um projeto de código aberto desenvolvido e mantido pelo Netflix que, assim como o Zookeeper e Consul, também disponibiliza uma forma para gerenciar configuração.

A ferramenta também fornece um balanceamento de carga na hora de descobrir serviços e também um *endpoint* REST para integração com outras tecnologias. É disponibilizado um cliente escrito em Java que adiciona mais recursos, como verificar a saúde das instâncias do sistema.

Tendo os clientes lidando diretamente com a descoberta de serviço, é evitada a necessidade de ter um processo separado para isso. Porém, é obrigatório que todos os clientes implementem a descoberta de serviço. O Netflix, que tem como padrão a JVM, alcança isso tendo todos os clientes utilizando Eureka. Em um ambiente poliglota, isso é bem mais desafiador.

¹ Serf é uma solução descentralizada para descoberta de serviço e orquestração, que é leve, altamente disponível e tolerante a falhas.

4.2.4 Hyperbahn

Devido à importância dessa ferramenta para o trabalho, ela será abordada no [Capítulo 5](#).

5 Hyperbahn

Hyperbahn é uma rede sobreposta de roteadores, baseada na biblioteca *Ringpop*, projetados para dar suporte ao protocolo RPC *TChannel*, sendo essas tecnologias todas criadas pela *Uber Technologies Inc.*. Seus nodos de roteamento dinamicamente convergem e decidem através de um protocolo epidêmico os serviços conhecidos, armazenando-os em um *Consistent hashing ring*, formando uma rede de serviços prontos para se comunicarem uns com os outros sem intervenção de um desenvolvedor e sem a necessidade de explicitamente especificar portas e endereços de rede.

O Hyperbahn habilita o roteamento e a descoberta de serviço em sistemas de larga escala compostos de diversos *microservices*. Funciona em ambiente distribuído, provê tolerância a falhas e prioriza a disponibilidade dos serviços. Permite que um serviço descubra e se comunique com outros de forma segura e simples, sem a necessidade de saber onde esses outros serviços estão efetivamente rodando.

O Hyperbahn oferece algumas características que facilitam o desenvolvimento de aplicações, dentre elas destacam-se:

- a) **configuração de descoberta** - Sem arquivos de configuração complexos;
- b) **timeouts** - Reforça o acordo de nível de serviço (SLA). Todas as requisições devem especificar um tempo para resposta, o que acarreta na rápida identificação de falhas;
- c) **múltiplas tentativas** - Elimina falhas temporárias, quando dentro da janela especificada para o tempo de resposta.
- d) **balanceamento de carga** - Distribui uniformemente chamadas entre instâncias de um mesmo serviço;
- e) **limitação da taxa de requisições** - Protege serviços contra ataques DoS;
- f) **quebra de circuito** - Previne falhas em cascata e corta comunicação com clientes indisponíveis. Cria um modelo rápido de identificação de falhas em todas as camadas da rede;
- g) **rastreamento distribuído** - Todo o fluxo de chamadas pode ser compreendido utilizando Zipkin¹.

¹ Zipkin é um sistema distribuído de rastreamento de requisições, que ajuda a reunir métricas de tempo necessárias para solucionar problemas de latência na arquitetura de *microservices*.

5.1 Tecnologias

O Hyperbahn é a junção de duas tecnologias que serão abordadas na subseções seguintes.

5.1.1 TChannel

O TChannel proporciona um protocolo para clientes e servidores com uma rede de roteamento inteligente (*Hyperbahn*) entre eles (TCHANNEL..., 2015).

Ele foi criado devido a diversos problemas gerados ao se manter um sistema distribuído com uma grande quantidade de *microservices* disponíveis. Dentre esses problemas, os mais comuns são a descoberta de serviço, tolerância a falhas e o rastreamento de requisições. Esses problemas são resolvidos da seguinte maneira:

- a) **descoberta de serviço** - Todos os produtores e consumidores se registram na rede de roteamento. Consumidores só podem acessar produtores através de um nome, sem a necessidade de conhecer portas ou endereços de rede;
- b) **tolerância a falhas** - A rede de roteamento mantém registro de falhas e pode inteligentemente detectar *hosts* defeituosos e os remover do grupo de *hosts* disponíveis;
- c) **rastreamento de requisições** - Inseridas no protocolo como *first class citizen*, ou seja, o objeto de rastreamento é passado adiante no encaminhamento de requisições.

Na concepção do protocolo e da rede de roteamento, os principais objetivos de projeto foram os seguintes:

- a) ser de fácil implementação em múltiplas linguagens de programação;
- b) eficiência no encaminhamento de pacotes, onde processos mediadores podem tomar decisões de encaminhamento rapidamente;
- c) modelo de requisição e resposta totalmente assíncrono para que requisições mais lentas não bloqueiem as requisições mais rápidas subsequentes;
- d) possibilidade de requisições e respostas serem quebradas em fragmentos menores e serem mandados progressivamente;
- e) soma de verificação (*checksum*) opcional;
- f) possibilidade de utilizar diversas formas de serialização, como, por exemplo, JSON e Thrift².

² Thrift é uma linguagem de definição de interface e um protocolo de comunicação binária desenvolvida pelo Facebook, Inc.. É utilizado para definir e criar serviços em diversas linguagens de programação assim como um *framework* para fazer chamada remota de procedimento (RPC).

5.1.2 Ringpop

Ringpop é uma biblioteca que mantém um *consistent hash ring* e pode ser usada para arbitrariamente particionar os dados de uma aplicação de forma escalável e tolerante a falhas (RINGPOP..., 2015).

A biblioteca tem como características fundamentais: um protocolo de *membership* (subseção 5.1.2.1), um *consistent hashing ring* (subseção 5.1.2.2) e o encaminhamento de requisições (Figura 18).

O seu protocolo de *membership* proporciona uma aplicação distribuída, atribuindo a suas instâncias, que anteriormente desconheciam seus pares, a capacidade de se descobrir, se auto-organizar e cooperar. Essas instâncias se comunicam utilizando um protocolo de *Gossip*³ diretamente por TCP e compartilham informações para entrar em acordo sobre os membros que fazem parte da aplicação distribuída.

Com uma lista consistente de membros definidos através do protocolo de *membership*, a biblioteca organiza esses membros em um *consistent hash ring*, que define um comportamento previsível com relação à distribuição dos dados dentro da aplicação.

O Ringpop não é um sistema externo ou mesmo um recurso infraestrutural compartilhado por diversas aplicações, mas sim uma biblioteca de desenvolvimento que funciona como uma camada da aplicação (Figura 10), que permite à aplicação se manter independente. Devido às suas características (subseção 5.1.2), o Ringpop promove escalabilidade e resiliência a falhas, mantendo complexidade e custos operacionais a um mínimo (RINGPOP..., 2015).

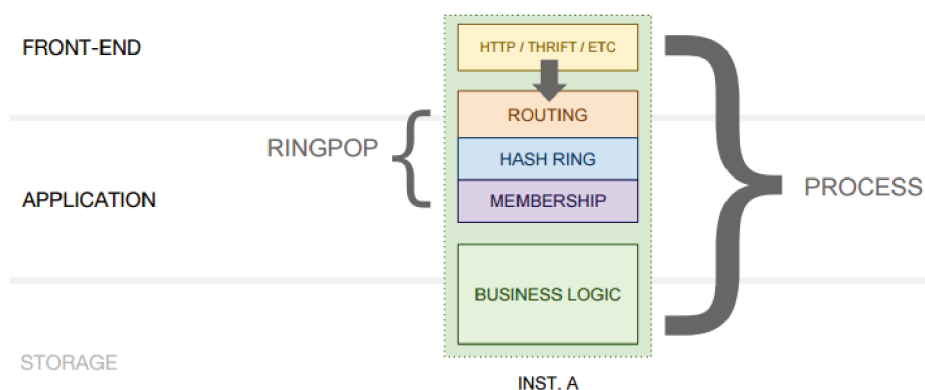


Figura 10 – Ringpop como uma camada da aplicação

Fonte: Wolski (2015, p. 36)

³ É um protocolo de comunicação, que foi inspirado na forma como informações são compartilhadas em redes sociais, comumente utilizado em sistemas distribuídos modernos. Periodicamente nodos pares trocam mensagens entre si, disseminando informação pelo sistema.

O uso do Ringpop pode passar completamente despercebido para usuários da aplicação. Não existe a necessidade dos clientes conhecerem os mecanismos por trás do esquema de particionamento ou mesmo o real destinatário de uma requisição. Balanceadores de carga, *proxies*, redes sobrepostas podem continuar sendo utilizadas sem gerar incompatibilidades.

Ringpop também vem acompanhado de uma API de gerenciamento para inspecionar e controlar a aplicação e todo um ferramental para ajudar a entender o comportamento da biblioteca e da aplicação.

5.1.2.1 Protocolo de Membership

Ringpop implementa um protocolo de *membership* que permite com que nodos se descubram, disseminem informação de forma rápida e preservem uma visão consistente dos membros do *cluster*. É usada uma variação de um protocolo de *Gossip* conhecido como SWIM (*Scalable Weakly-consistent Infection-style Process Group Membership Protocol*) (DAS; GUPTA; MOTIVALA, 2002) para propagar atualizações da lista de membros entre os membros participantes (RINGPOP..., 2015).

A biblioteca utiliza os mecanismos “*ping*” e “*ping-req*” do SWIM. OS *pings* são usados para espalhar informação e detectar falhas. Os membros verificam com *ping* uns aos outros de maneira randômica até chegarem em consenso sobre a lista completa de membros (Figura 11).

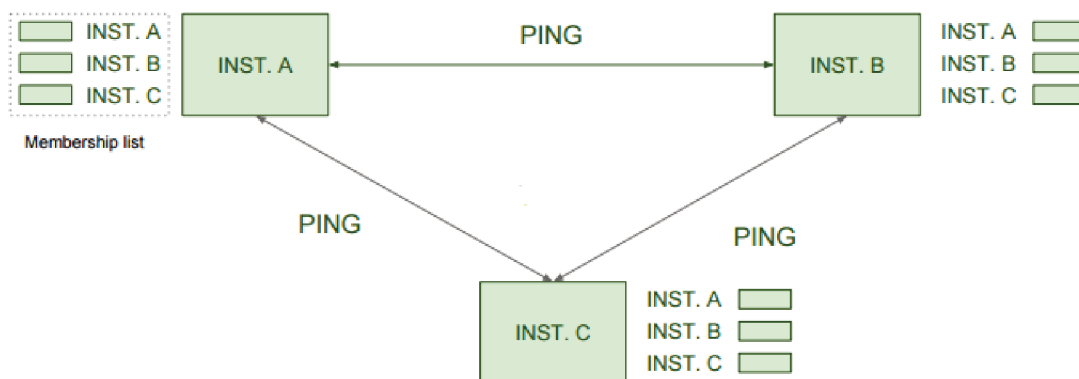


Figura 11 – Mecanismo de *pings* do protocolo de *membership*

Fonte: Wolski (2015, p. 14)

Protocolo *Gossip* SWIM para disseminar informação - Existe um cluster com dois nodos: A e C. A está verificando C utilizando o mecanismo de ping e vice-versa. Um terceiro nodo, B, se junta ao *cluster* após verificar A com um *ping*. Até esse momento apenas A sabe da existência de B. Na próxima vez que A verificar C, o conhecimento

de B será compartilhado fazendo com que o mesmo vire efetivamente parte do *cluster* (Figura 12). Esse é o processo de disseminação de informação do protocolo *Gossip SWIM*.

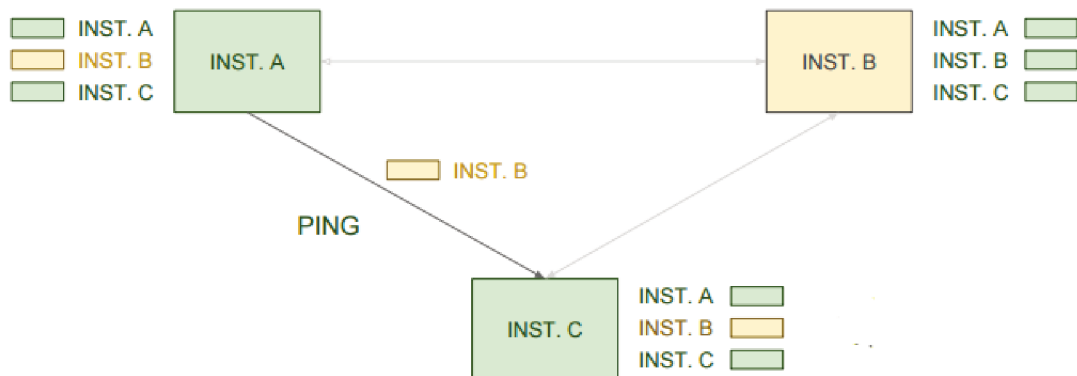


Figura 12 – Processo de disseminação de informação do protocolo *Gossip SWIM*

Fonte: Wolski (2015, p. 19)

Protocolo Gossip SWIM para detectar falhas - O Ringpop utiliza como meio de comunicação para o SWIM o TCP e é dessa forma que é feito o mecanismo de encaminhamento. Nodos do anel estão constantemente compartilhando informações e encaminhando requisições no mesmo canal TCP. Para detecção de falhas, Ringpop computa a listagem de membros e aplica uma soma de verificação (*checksum*) sobre o anel.

A lista de membros contém endereços e estados (*alive, suspect, faulty, etc.*) das instâncias. Também contém metadados adicionais como o relógio lógico. Toda essa informação é combinada e então é feita a soma de verificação com base nesses dados, onde divergências e falhas são detectadas.

5.1.2.2 Consistent hashing ring

Ringpop utiliza o método de *consistent hashing* com a finalidade de minimizar o número de chaves que precisam ser rebalanceadas quando o *cluster* muda de tamanho. *Consistent hashing* aplica uma função de *hash*⁴ não somente aos dados, mas também aos nodos que operam esses dados dentro do *cluster*. É utilizada como implementação da estrutura de dados uma árvore de busca binária balanceada, que proporciona buscas, inserções e remoções na complexidade de $O(\log n)$.

Assim que membros se juntam ou saem do *cluster*, os endereços das instâncias passam pela função de *hash* e então essa informação é adicionada ao *consistent hash ring* (RINGPOP..., 2015).

⁴ Usa o FarmHash, uma biblioteca criada pelo Google, que disponibiliza funções de *hash* para diversos tipos de dados.

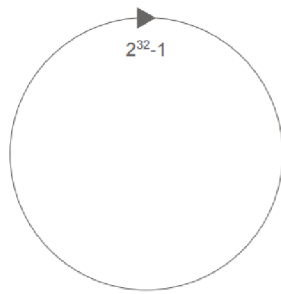


Figura 13 – Intervalo dos números inteiros em um *consistent hashing ring*

Fonte: Wolski (2015, p. 28)

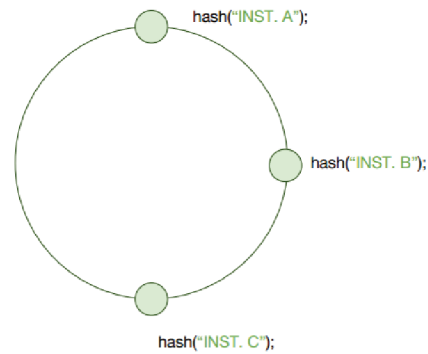


Figura 14 – Mapeamento das instâncias no *consistent hashing ring*

Fonte: Wolski (2015, p. 29)

Uma aplicação inicializa com um *keyspace*, e se torna necessário definir o que esse *keyspace* representa e quais dados serão manipulados pela aplicação. Um *keyspace* nada mais é do que um nome para definir todos os possíveis valores dos identificadores das entidades no sistema. Por exemplo, se a aplicação opera sobre usuários que possuem identificadores, então esses identificadores irão se tornar o *keyspace*.

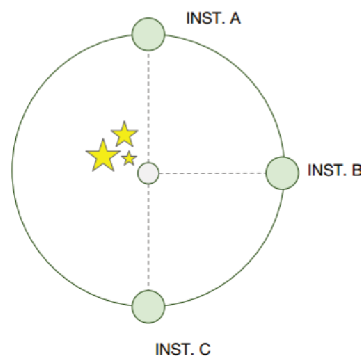


Figura 15 – Particionamento do *keyspace* entre as instâncias no *consistent hashing ring*

Fonte: Wolski (2015, p. 30)

O *keyspace* definido é, então, projetado em um anel, que representa o intervalo dos números inteiros (Figura 13). Em seguida as instâncias são distribuídas no anel utilizando a função de *hash* (Figura 14), e assim particionando o *keyspace* entre as instâncias (Figura 15).

Com uma visão consistente da lista de membros, que foi definida com a ajuda do protocolo de *membership*, dentro do *consistent hashing ring* e com o *keyspace* bem definido e particionado, é possível determinar quais entidades são mantidas por uma determinada instância. Supomos que chegue uma requisição ao sistema direcionada a uma entidade com um identificador. Primeiramente, é necessário descobrir qual nodo do *cluster* opera sobre

aquela informação e para encontrá-lo é preciso aplicar a função de *hash* no identificador da entidade em questão. No caso da Figura 16, a instância “INST. B” detém acesso aos dados referentes à entidade identificada por “USER1”.

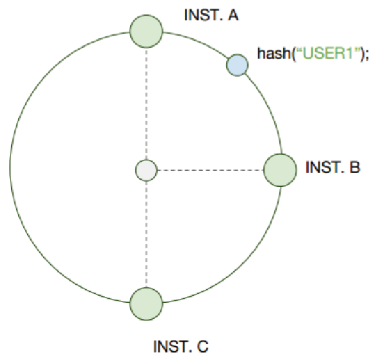


Figura 16 – Mapeamento de uma entidade no *consistent hashing ring*

Fonte: Wolski (2015, p. 32)

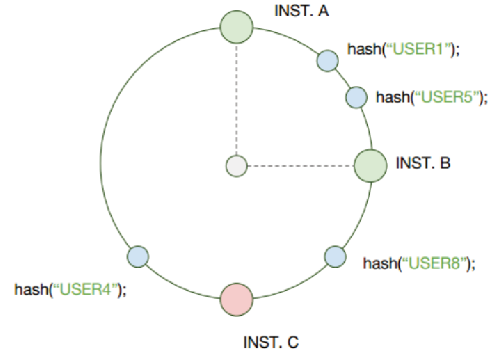


Figura 17 – Rebalanceamento do *keyspace* em caso de falha no *consistent hashing ring*

Fonte: Wolski (2015, p. 34)

Em caso de falha, como, por exemplo, se alguma instância ficar indisponível, o protocolo de *membership* automaticamente detecta essa falha e remove a instância inconsistente da lista de membros do *cluster* e do *consistent hashing ring*, fazendo com que o *keyspace* seja reparticionado automaticamente. Na Figura 17, foi detectado que a instância “INST. C” está indisponível, logo, os dados que anteriormente pertenciam a ela, “USER8”, agora são mantidos pela instância “INST. A”.

5.1.2.3 Forwarding

Ringpop convenientemente encaminha as requisições da aplicação. Geralmente, o tráfego da aplicação é direcionado a alguma entidade no sistema como um identificador de um objeto. Esse identificador pertence a alguma instância em particular dentro do *cluster*, dependendo de como o algoritmo de *hash* foi aplicado. Se a chave mapear para uma instância que não recebeu a requisição, então essa requisição é simplesmente encaminhada de forma subjacente (Figura 18). Todo esse processo funciona em uma camada mediadora (RINGPOP..., 2015).

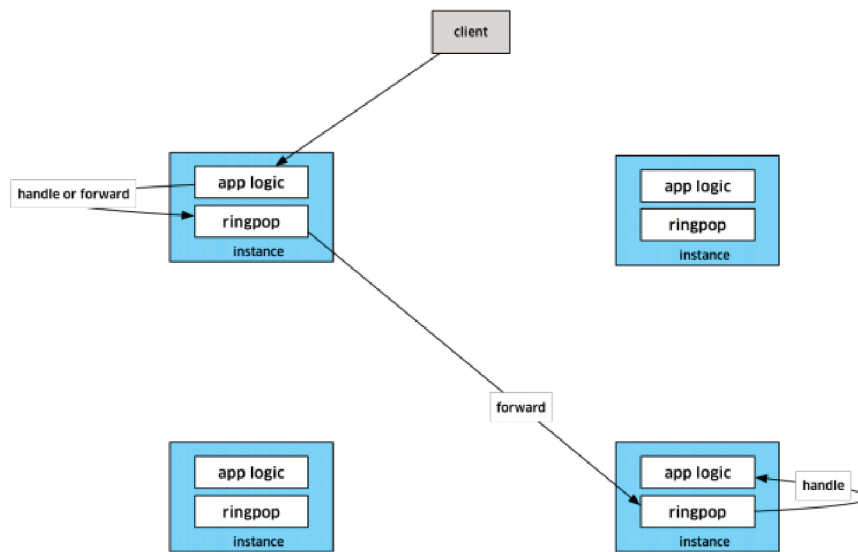


Figura 18 – Exemplo de encaminhamento de requisições no Ringpop

Fonte: Ranney (2015, p. 12)

6 Proposta

Como apresentado no [Capítulo 3](#), a arquitetura de *microservices* é uma abordagem que tem como objetivo geral o desenvolvimento de uma aplicação como um conjunto de pequenos serviços, cada um rodando em seu próprio processo e se comunicando de forma simples. O termo serviço nesse contexto se refere a uma funcionalidade de software, como por exemplo, a busca ou criação de uma determinada informação, que pode ser reusada por diferentes consumidores para diferentes propósitos.

Dado o estilo de descoberta abordado pelas tecnologias de registro dinâmico de serviços apresentadas no [Capítulo 4](#), a proposta visa adaptar uma das tecnologias para descobrir funcionalidades e não somente mapear servidores para um nome específico. Mesmo que simples, essas mudanças darão mais liberdade para realizar a migração de um determinado conjunto de serviços para um novo local ou para realizar integração entre serviços consumidores e provedores. Com esse aprimoramento, passa a ser irrelevante por parte dos desenvolvedores conhecer exatamente os serviços que uma determinada instância de um *microservice* oferece, devido ao novo significado de registro e consulta de serviços.

Diversas tecnologias podem ser adaptadas, já que muitas alcançam os mesmos resultados. Neste trabalho foi escolhido o *Hyperbahn*, pois é uma ferramenta que está bem consolidada e é bastante utilizada na indústria. É importante ressaltar que a descoberta se dá em um ambiente contido, ou seja, somente *microservices* pertencentes a esse ambiente podem utilizar os mecanismos de descoberta e aproveitar as características que o *Hyperbahn* disponibiliza. Para aproveitar toda a estrutura oferecida pela ferramenta, alguns ajustes foram necessários no registro e na consulta de serviços. Múltiplas entradas no registro podem identificar um *microservice* ao invés de somente uma, que é o caso da implementação original. Os *microservices* se registram utilizando URIs da principal representação de dados utilizada na web semântica, o *RDF* (*Resource Description Framework*). Sendo essas URIs representantes de conceitos e relações de uma ontologia. Já na parte da consulta, em um primeiro cenário, todas as entradas de registro (URIs) serão fornecidas ao *microservice* consumidor, o qual ficará com a tarefa de inferir e definir quais conceitos suprem suas necessidades. Se o processo de inferência for um sucesso para uma ou mais entradas de registro, o consumidor poderá utilizar essas entradas para efetivamente descobrir o *host* do *microservice* que lida com as entidades desejadas.

6.1 Incorporação de semântica

Este trabalho propõe modificações no mecanismo de descoberta de serviços oferecidos pelo Hyperbahn. As modificações propostas consideram o uso de uma semântica leve para tratar o problema de registros de serviços.

A descoberta de serviço, como apresentada pela ferramenta original, relaciona um *microservice* a apenas um nome, o que é muito pouco para descrever suas características funcionais. Diferentemente, a solução proposta possibilita que um mesmo *microservice* seja reconhecido por diversos nomes. Porém, isso não é o suficiente para atribuir significado à descoberta, exigindo que alguns conceitos da Web semântica sejam utilizados. Como definido na proposta (Capítulo 6), URIs RDF foram utilizadas como entradas de registro para representar características dos *microservices*, dando um significado para os dados que retornam em uma requisição de descoberta.

Na solução entregue, o Hyperbahn não implementa um motor de inferência, fazendo com que os clientes consumidores tenham a responsabilidade de inferir quais serviços disponíveis na rede se encaixam nas suas necessidades. Para descobrir quais serviços estão disponíveis, um meio de se obter todas as entradas de registro foi disponibilizado na implementação modificada.

De forma geral, o fluxo de uso de uma aplicação cliente pode ser representado da seguinte forma. Após o registro (1), a aplicação cliente precisa descobrir quais serviços estão sendo disponibilizados na rede. Para isso, todas as entradas de registro cadastradas no *Hyperbahn* são obtidas (2). Uma vez com todas as entradas de registro em mãos, a aplicação cliente infere quais entradas suprem as suas necessidades (3). Em seguida, a funcionalidade de descoberta baseada em múltiplos nomes é utilizada (4) tendo como resultado os *hosts* cadastrados para as entradas de registro. Esse fluxo está ilustrado na Figura 19.

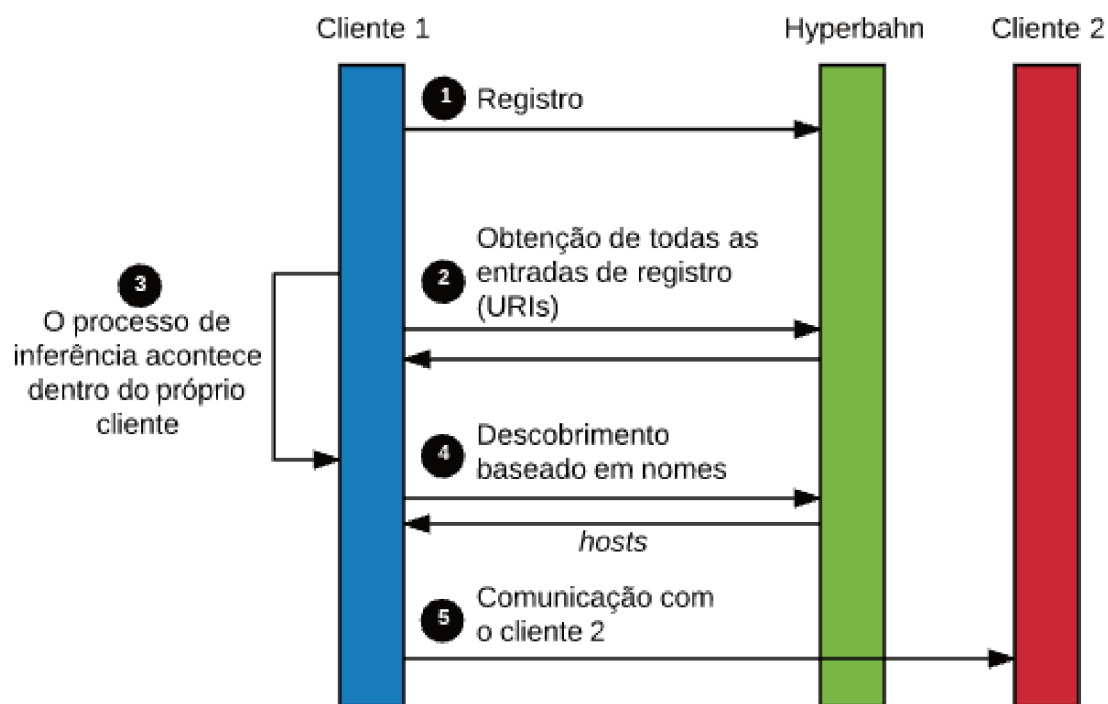


Figura 19 – Fluxo de uso da solução proposta

Fonte: Autor

6.2 Implementação

Pelo fato do trabalho apresentado tratar-se de adicionar novas funcionalidades a uma ferramenta já existente, mantiveram-se os padrões já existentes no código fonte da aplicação original, assim como sua estrutura de entidades. Para não limitar o escopo de uso da ferramenta modificada somente à solução proposta, o termo “*tags*” foi utilizado no código.

Como já citado, foram necessárias mudanças na forma como aplicações de registram e como são consultadas no *Hyperbahn*, assim como uma forma de obter todas as entradas de registro já cadastradas.

6.2.1 Registro de serviço

Para permitir que um *microservice* seja identificado por múltiplos nomes, ao invés de apenas um, foi incorporada a propriedade *serviceTags* (linha 15) na requisição de registro na biblioteca cliente do *Hyperbahn* (Listagem 6.1).

Além disso, para suportar o registro simultâneo de milhares de nomes, foi necessário que esses nomes, que estão representados no código como *tags*, fossem mandadas em lotes de 30 (linha 5), preservando o consumo de I/O e processamento.

```

1 HyperbahnClient.prototype.sendBatchRequests =
2 function sendBatchRequests(opts, endpoint, cb) {
3     var self = this;
4     var finished = false;
5     var chunk = DEFAULT_REQ_DATA_CHUNK_SIZE;
6     var counter = Math.ceil(self.serviceTags.length / chunk);
7
8     var i, j;
9     for (i = 0, j = self.serviceTags.length; i < j; i += chunk) {
10        var req = self.newRequest();
11        self.tchannelJSON.send(req, endpoint, null, {
12            services: [{
13                cost: 0,
14                serviceName: self.serviceName,
15                serviceTags: self.serviceTags.slice(i, i + chunk),
16            }]
17        }, sendRequestInternalCb);
18    }
19
20    // Código para enviar a resposta ao consumidor
21    ...
22 };

```

Listagem 6.1 – Código executados por clientes para se registrar no mecanismo de

descoberta de serviço

No servidor foram feitas modificações ([Listagem 6.2](#)) para dar suporte à nova propriedade adicionada (linhas 14 a 17). Além de ser feito o registro distribuído do nome (*serviceName*) fornecido, as *serviceTags* também foram incluídas e registradas (linhas 19 a 40).

```
1 HyperbahnHandler.prototype.sendRelays =
2 function sendRelays(req, arg2, arg3, endpoint, cb) {
3     var self = this;
4     var services = arg3.services;
5     var finished = false;
6
7     var servicesByExitNode = {};
8
9     for (var i = 0; i < services.length; i++) {
10        var service = services[i];
11        service.hostPort = req.connection.remoteName;
12
13        var serviceNames = [service.serviceName];
14        if (service.serviceTags && service.serviceTags.length) {
15            serviceNames = serviceNames.concat(service.serviceTags);
16        }
17
18        for (var j = 0; j < serviceNames.length; j++) {
19            var serviceName = serviceNames[j];
20            if (serviceName.length === 0) {
21                continue;
22            }
23
24            // Obtenção dos nodos do Hyperbahn que mapeiam para um
25                determinado nome
26            var exitNodes = self.egressNodes.exitsFor(serviceName);
27            var exitHosts = Object.keys(exitNodes);
28
29            var amendedService = deepExtend({}, service, {serviceName:
30                serviceName});
31            for (var k = 0; k < exitHosts.length; k++) {
32                var exitNode = exitHosts[k];
33
34                var relayReq = servicesByExitNode[exitNode];
35                if (!relayReq) {
36                    relayReq = servicesByExitNode[exitNode] = [];
37                }
38                relayReq.push(amendedService);
39            }
40        }
41    }
42}
```

```

39     }
40 }
41
42 // Código para enviar nome dos serviços para os outros nodos
43 // presentes no cluster do Hyperbahn
44 ...
45 // Código para lidar com as respostas das requisições enviadas aos
46 // outros nodos do Hyperbahn no cluster
47 ...
48 // Código para enviar a resposta ao consumidor
49 ...
50 }
51 };

```

Listagem 6.2 – Código que lida com requisições de registro

6.2.2 Consulta de serviço baseada em múltiplos nomes

Para habilitar a consulta baseada em múltiplos nomes, foi criado um novo serviço (*TagDiscovery*).

Como a comunicação entre cliente e *Hyperbahn* se dá através de chamadas de procedimento remota descritas através do *Thrift*, foram necessária algumas mudanças nessa descrição.

Foram criadas novas *structs* que definem os parâmetros da requisição (Listagem 6.3) e o formato da resposta (Listagem 6.4) e, por fim, a definição do serviço (Listagem 6.5).

```

1 struct TagDiscoveryQuery {
2     1: required list<string> serviceTags
3 }

```

Listagem 6.3 – *TagDiscovery*: parâmetros - *Thrift*

```

1 struct TagDiscoveryResult {
2     1: required map<string, list<ServicePeer>> peers
3 }

```

Listagem 6.4 – *TagDiscovery*: resposta - *Thrift*

```

1 TagDiscoveryResult tagDiscover(
2     1: required TagDiscoveryQuery query
3     ) throws (
4     1: NoPeersAvailable noPeersAvailable
5     )

```

Listagem 6.5 – *TagDiscovery*: serviço - *Thrift*

No servidor foi criado um método para lidar com as requisições do serviço de *TagDiscovery* recém criado.

Para cada nome a ser consultada, é utilizado o serviço *discover* já implementado na aplicação original (linhas 12 a 29 na [Listagem 6.6](#)). Aqui é utilizado um dos conceitos já explicados do *Ringpop*, o *Forwarding*. Caso um nome não mapear para o nodo que está processando a requisição em questão, a mesma é encaminhada para o nodo que o consiga. Uma vez que o processo de descoberta termina, uma resposta é encaminhada ao serviço consumidor ([Listagem 6.6](#)).

```
1 HyperbahnHandler.prototype.tagDiscover =
2 function tagDiscover(handler, req, head, body, cb) {
3     var serviceTags = body.query.serviceTags;
4     var responses = {};
5     var finished = false;
6
7     //Codigo validando parametros da requisicao
8     ...
9
10    var counter = 1 + serviceTags.length;
11    for (var i = 0; i < serviceTags.length; i++) {
12        var serviceTag = serviceTags[i];
13
14        var topChannel = handler.channel.topChannel;
15        var svcchan = topChannel.handler.getOrCreateServiceChannel(
16            serviceTag);
17
18        if (svcchan.serviceProxyMode === 'forward' &&
19            req.headers.cn !== 'hyperbahn'
20        ) {
21            handler._forwardToRemoteDiscover(
22                req,
23                svcchan,
24                {query: {serviceName: serviceTag}},
25                onDiscoverSent.bind(null, serviceTag)
26            );
27            continue;
28        }
29        handler._getDiscoverHosts(serviceTag, onDiscoverSent.bind(null,
30            serviceTag));
31    }
32    //Codigo para lidar com as respostas da chamada de "discover"
33    ...
34
```

```

35     // Código para enviar a resposta ao consumidor
36     ...
37 };

```

Listagem 6.6 – Processo de descoberta baseada em *tags*

6.2.3 Obtenção de todas as entradas de registro

Para obter todas as entradas de registros cadastradas no *Hyperbahn*, foi necessário criar mais um serviço primário e um outro auxiliar.

Inicialmente foram obtidos os endereços de todos os nodos “vivos” (linha 8) do *Hyperbahn* inseridos na listagem de membros (linha 4) presente no Ringpop ([Listagem 6.7](#)).

```

1 EgressNodes.prototype.nodes = function nodes() {
2     var self = this;
3
4     var members = self.ringpop.membership.members;
5     var hosts = [];
6     for (var i = 0; i < members.length; i++) {
7         var member = members[i];
8         if (member.status === MemberStatus.alive) {
9             hosts.push(member.address);
10        }
11    }
12    return hosts;
13 };

```

Listagem 6.7 – Obtendo membros “vivos” do Ringpop

Em seguida se deu a criação do serviço primário *serviceNames*, que tem como objetivo obter as entradas de registro armazenadas nos nodos listados pelo método citado anteriormente.

Para cada membro do *cluster* é feita uma requisição ao serviço auxiliar *relay-serviceNames* (linhas 7 a 14 na [Listagem 6.8](#)). Lembrando que essas requisições são feitas e processadas paralelamente.

```

1 HyperbahnHandler.prototype.handleServiceNames =
2 function handleServiceNames(handler, req, head, body, cb) {
3     var nodes = handler.egressNodes.nodes();
4     var responses = [];
5     var finished = false;
6
7     var counter = 1 + nodes.length;
8     for (var i = 0; i < nodes.length; i++) {
9         handler.sendRelay({

```

```
10         hostPort: nodes[i],
11         inreq: req,
12         endpoint: 'relay-serviceNames'
13     }, onRelaySent);
14 }
15
16 // Código para lidar com as respostas das requisições enviadas aos
17 // nodos "vivos" do cluster
18 ...
19 // Código para enviar a resposta ao consumidor
20 ...
21 };
```

Listagem 6.8 – Obtendo entradas de registro cadastradas em todos os nodos pertencentes ao *cluster* do Hyperbahn

Finalmente, o serviço auxiliar, que ao ser executado pelos nodos pares, retorna todas as entradas de registro cadastradas em cada instância, como pode ser visualizado na [Listagem 6.9](#).

```
1 HyperbahnHandler.prototype.handleRelayServiceNames =
2 function handleRelayServiceNames(handler, req, head, body, cb) {
3     var keys = Object.keys(handler.channel.topChannel.subChannels);
4     cb(null, {
5         ok: true,
6         body: {
7             serviceNames: pull(keys, 'autobahn', 'hyperbahn', 'ringpop')
8         }
9     });
10 };
```

Listagem 6.9 – Obtendo entradas de registro cadastradas em um nodo

6.3 Estudo de Caso

Este capítulo apresenta um estudo de caso com o objetivo de exemplificar a aplicação da solução proposta. O estudo de caso considera os conceitos presentes na ontologia ilustrada na Figura 20 e por *microservices* que manipulam as entidades associadas a esses conceitos.

A ontologia utilizada no estudo de caso é constituída por seis classes semânticas: Obra, Filme, Livro, Pintura, Gênero e Terror.

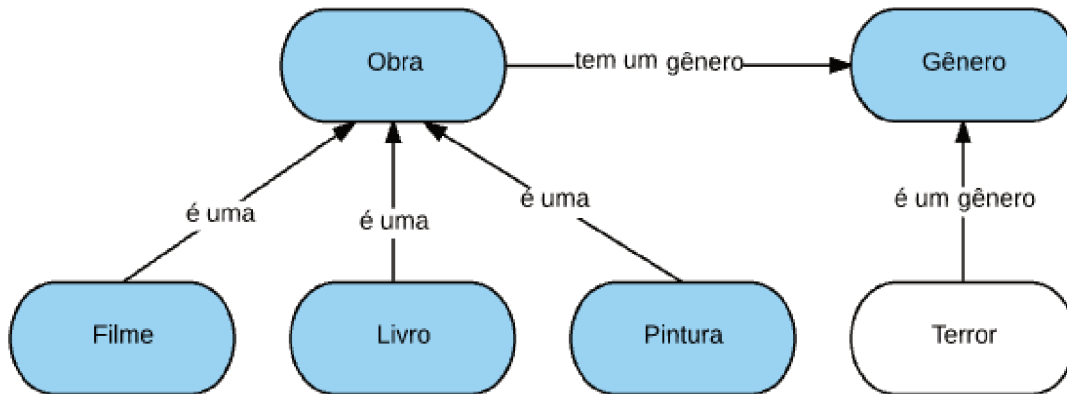


Figura 20 – Ontologia de exemplo para o estudo de caso

Fonte: Autor

A estrutura de *microservices* presente no servidor é representada pela Figura 21. Fazem parte dessa estrutura os *microservices* que manipulam as entidades representantes dos conceitos da ontologia exemplo, um *API Gateway* responsável por agregar informações dos demais *microservices* e entregá-las ao usuário e pelo Hyperbahn.

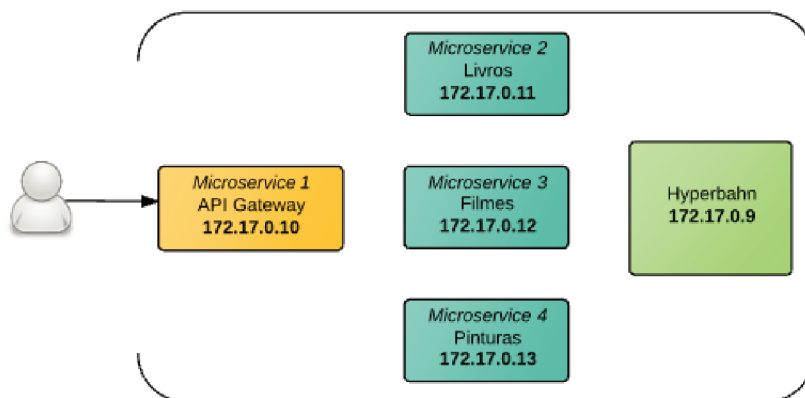


Figura 21 – Estrutura de *microservices* presente no estudo de caso

Fonte: Autor

Os *microservices*, que foram apresentados para o estudo de caso, após serem implantados serão registrados no Hyperbahn resultando na configuração de registro mostrada

na [Tabela 1](#) e, seguindo a solução proposta, as entradas de registro devem representar conceitos ou propriedades de classes semânticas da ontologia.

Como dito anteriormente, cada *microservice* opera sobre entidades de um ou mais conceitos da ontologia.

- O *API Gateway* manipula entidades presentes em todas as hierarquias de Obra.
- *Microservice 2* manipula livros de qualquer gênero.
- *Microservice 3* manipula apenas filmes de terror.
- *Microservice 4* manipula pinturas de qualquer gênero.

ID	Entradas de registro	Endereço de IP
1	“<Thing> a <Obra>”	172.17.0.10
2	“<Obra> a <Livro>”	172.17.0.11
2	“<Livro> <temUmGenero> <Terror>”	172.17.0.11
3	“<Filme> <temUmGenero> <Terror>”	172.17.0.12
4	“<Obra> a <Pintura>”	172.17.0.13

Tabela 1 – Estudo de caso - Entradas de registro no Hyperbahn

O *API Gateway* é o único *microservice* que recebe estímulos externos provenientes do usuário. Nele são disponibilizados serviços que agregam informações armazenadas nos demais *microservices* registrados no Hyperbahn.

A partir de uma requisição de usuário, o processo de descoberta começa primeiramente no *API Gateway*, onde todas as entradas de registro são requisitadas ao Hyperbahn. Em seguida é executado um processo de inferência sobre as entradas de registro obtidas, que pode ser executado facilmente por ferramentas como o Apache Jena. A inferência tem como objetivo principal explorar os relacionamentos hierárquicos definidos na ontologia de acordo com as entidades gerenciadas pelos *microservices*. As entradas de registro resultantes do processo de inferência são utilizadas para descobrir o *host* dos *microservices* responsáveis por manipular tais conceitos. Finalmente, as informações armazenadas nos *microservices* descobertos serão agregadas pelo *API Gateway* e devolvidas ao usuário.

Para fins de exemplificação, suponha que existe um usuário que está buscando informações sobre obras de terror e ele fez uma requisição ao serviço no *API Gateway* capaz de devolver esses dados. Nesse serviço a relação “<Obra> <temUmGenero> <Terror>” é utilizada como a base do processo de inferência hierárquica sobre todas as entradas de registro obtidas do Hyperbahn. Esse processo resulta em apenas duas entradas de

registros, que são “<Livro> <temUmGenero> <Terror>” e “<Filme> <temUmGenero> <Terror>”, que suprem as necessidades do usuário. Em seguida as entradas inferidas serão utilizadas para descobrir os *microservices* (“172.17.0.11” e “172.17.0.12”) associados a elas no Hyperbahn. Por fim, as informações serão obtidas do *microservices* descobertos.

7 Experimentos e resultados

Este capítulo apresenta as análises e validações para a proposta do trabalho. A validação da proposta é realizada através da comparação de desempenho entre a implementação original e a modificada, levando em consideração a incorporação de conceitos semânticos à consulta e ao registro de serviços.

7.1 Experimentos

Para realizar a validação da implementação proposta, é necessário que os serviços, tanto consumidores quanto produtores, sejam implantados em um ambiente computacional, de modo a possibilitar a comunicação com as aplicações clientes. O servidor possui o sistema operacional CoreOS 1122.2.0 (*stable*), instalado em um *cloud server* na Digital Ocean. A instância escolhida possui dois processadores Intel Xeon operando a 2,4 GHz e 4GB de memória principal.

O ambiente de execução já possui como estratégia de implantação Linux *containers*. O sistema operacional conta com o Docker versão 1.10.3, que será utilizado para auxiliar na implantação das aplicações. O uso de Linux *containers* é de grande importância pois traz uma série de vantagens na execução dos experimentos, como a rápida mudança de parâmetros para a realização de testes e comparações, assim como facilita a troca entre a solução proposta e a implementação original.

Para os testes de desempenho um conjunto de triplas RDF foi obtido no website DBpedia¹ para apoiar a execução do estudo. Cerca de 136,0 KB de dados, que contabilizam 1000 triplas RDF, foram armazenados em um arquivo, que foi carregado pelas aplicações cliente no momento adequado.

O próximo passo foi adaptar o *Hyperbahn* para ser implantado em um *container* e torná-lo disponível para as aplicações consumidoras. Foram criadas duas imagens Docker², uma contendo a implementação original e outra a solução proposta, o que facilitou a troca de uma para outra quando necessário. Lembrando que não foi utilizado nenhum serviço externo para o armazenamento de imagens Docker. Da mesma forma, as aplicações cliente também foram implantadas em contêineres.

Foi criado um *script* para automatizar a execução dos cenários. Para cada teste o ambiente foi reiniciado e todos os seus dados apagados. Cada cenário foi executado

¹ <http://dbpedia.org>

² Uma imagem Docker é uma especificação do que o *container* vai possuir quando for executado. Ela não pode ser executada, pois apenas armazena as instruções e dados que depois serão usadas para criar o container.

10 vezes, de modo a possibilitar o cálculo de uma média de tempo de resposta após a execução das requisições.

Os fatores variáveis escolhidos para a realização dos experimentos foram a quantidades de nodos do *cluster* e o número de réplicas de uma determinada entrada (k-value).

7.2 Resultados

A [Tabela 2](#) apresenta os dados obtidos na execução da requisição de registro. Para obter essa média na implementação original foi necessário executar a requisição diversas vezes para simular a funcionalidade de registro por múltiplos nomes implementada na solução proposta. Mesmo com as requisições de registro para a implementação original sendo executadas paralelamente, não foi possível obter bons resultados para o tempo de resposta em comparação com implementação proposta.

Número de nodos no <i>cluster</i>	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (<i>original</i>)	Tempo de resposta (ms) (<i>proposta</i>)
2	1	2185,8	355,6
2	5	3737,9	527,5
2	10	3553,5	636,0
3	1	2508,3	387,5
3	5	3778,7	1039,5
3	10	3979,1	1072,3
4	1	2678,0	420,3
4	5	4980,6	1137,8
4	10	4840,5	1841,4
5	1	2847,2	447,8
5	5	4434,3	1854,0
5	10	5985,0,6	2970,1

Tabela 2 – Média do tempo(ms) das requisições de registro

Os dados obtidos das execuções da requisição de descobrimento baseado em múltiplos nomes podem ser encontrado na [Tabela 3](#). Para obter os tempos relacionados à implementação original foram feitos ajustes similares aos realizados para obter os dados da [Tabela 2](#). O tempo de resposta está intrinsecamente ligado com a distribuição dos dados e como estão organizados no *cluster*. Uma taxa de distribuição alta implica em um tempo de resposta maior. Em contrapartida, a carga de processamento não fica direcionada a apenas uma instância, e sim distribuída entre todas as instâncias do *cluster*.

Número de nodos no <i>cluster</i>	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (<i>original</i>)	Tempo de resposta (ms) (<i>proposta</i>)
2	1	130,8	94,5
2	5	119,5	61,9
2	10	115,1	26,0
3	1	151,0	130,0
3	5	105,7	70,6
3	10	116,0	44,2
4	1	137,7	123,3
4	5	139,1	112,5
4	10	102,8	53,6
5	1	134,4	149,0
5	5	157,3	92,8
5	10	185,2	160,6

Tabela 3 – Média do tempo(ms) das requisições de 100 nomes

A Tabela 4 corresponde aos dados obtidos nas execuções sucessivas da requisição para se obter todas as entradas de registro cadastradas no Hyperbahn. O tempo de resposta pode sofrer variação dependendo de qual instância for responsável por agregar todos os registros. Isso se dá devido às chaves registradas não estarem distribuídas igualmente no *cluster*.

Número de nodos no <i>cluster</i>	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (<i>proposta</i>)
2	1	25,3
2	5	41,7
2	10	58,9
3	1	30,4
3	5	49,3
3	10	38,1
4	1	28,9
4	5	60,9
4	10	56,4
5	1	35,8
5	5	70,5
5	10	74,9

Tabela 4 – Média do tempo(ms) de resposta para obter todas as entradas de registro cadastradas

8 Conclusão

Com o intuito de aprimorar os mecanismos de descoberta de serviço disponibilizadas pelas ferramentas estudadas, esse trabalho utilizou a web semântica como meio para atingir seus resultados. A ferramenta Hyperbahn foi a escolhida para ser modificada. Originalmente uma solução que possibilitava apenas a atribuição de um nome para o endereço de uma máquina na rede, foi modificada ao ponto de também proporcionar o mapeamento de múltiplos nomes a um mesmo *host*.

Em meio a um ambiente altamente heterogêneo, volátil e muitas vezes caótico, que é a realidade no mundo de desenvolvimento com *microservices*, se dá pouca importância a uma descoberta de serviço que descreve semanticamente as funcionalidades dos *microservices*, o que resulta, na maioria dos casos, em acoplamento entre consumidores e provedores de serviços. Com relação a esse problema, o presente trabalho procurou um meio de atribuir significado ao método de descoberta de serviço nas ferramentas estudadas, estabelecendo com que os nomes utilizados no registro fossem substituídos por nomes com significado. Foram escolhidas URIs RDF para isso, que podem representar conceitos ou relações entre conceitos de uma ontologia, com o fim de atribuir sentido aos dados retornados do processo de descoberta. Tais mudanças, proporcionam um ambiente que reforça ainda mais as características boas resultantes do uso da arquitetura de *microservices*, como a independência, resiliência e escalabilidade.

Como podem ser visualizadas no [Capítulo 7](#), as vantagens oferecidas pela ferramenta de descoberta escolhida, o *Hyperbahn*, tentaram ser aproveitadas ao máximo, como a clusterização e a fragmentação do dados entre as instâncias pertencentes ao *cluster*. Também foi feito proveito da estrutura provida pelo ambiente utilizado para execução dos experimentos, a implantação dos *microservices* utilizando Linux *containers*, que vêm sendo extensamente utilizados em ambientes de produção na indústria, facilitou em muito a troca de ambiente entre cenários de teste.

8.1 Trabalhos futuros

Já que a ferramenta escolhida para ser modificada não disponibiliza um motor de inferência, trabalhos futuros podem fazer um estudo sobre como transferir o motor de inferência dos clientes para o mecanismo de descoberta de serviços ou mesmo criar um serviço auxiliar que desempenhe esse papel.

Outra possibilidade seria fazer com que as aplicações clientes fornecessem uma ontologia que seria registrada junto à ferramenta de descoberta. Essa ontologia seria adicionada a outras já registradas e a inferência, no momento da descoberta, seria feita com base nessa ontologia completa.

Referências

- BERNERS-LEE, T. *Linked Data: Personal view only. imperfect but published*. 2006. Disponível em: <<http://www.w3.org/DesignIssues/LinkedData.htm>>. Acesso em: 23 outubro 2016. Citado 3 vezes nas páginas 13, 30 e 31.
- BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The semantic web. *Scientific American*, v. 284, n. 5, p. 28–37, 2001. Citado na página 29.
- BIZER, C.; HEATH, T.; BERNERS-LEE, T. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, v. 5, n. 3, p. 1–22, 2009. Citado na página 30.
- DAS, A.; GUPTA, I.; MOTIVALA, A. Swim: Scalable weakly-consistent infection-style process group membership. *Dependable Systems and Networks, 2002*, p. 303–312, 2002. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1028914>>. Acesso em: 7 julho 2016. Citado na página 48.
- ELGAZZAR, K.; HASSAN, A. E.; MARTIN, P. Clustering wsdl documents to bootstrap the discovery of web services. *Proceedings of the International Conference on Web Services (ICWS)*, p. 147–154, 2010. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5552792>>. Acesso em: 20 julho 2016. Citado na página 26.
- FILHO, O. F. F. *Serviços Semânticos: Uma Abordagem RESTful*. Dissertação (Mestrado) — Universidade de São Paulo, 2009. Citado 2 vezes nas páginas 29 e 31.
- FOWLER, M.; LEWIS, J. *Microservices*. 2014. Disponível em: <<http://martinfowler.com/articles/microservices.html>>. Acesso em: 6 junho 2016. Citado 3 vezes nas páginas 35, 36 e 37.
- MCILRAITH, S. A.; SON, T. C.; ZENG, H. Semantic web services. *IEEE Intelligent Systems*, v. 16, n. 2, p. 46–53, 2001. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=920599>>. Acesso em: 24 maio 2016. Citado na página 23.
- NEWMAN, S. *Building Microservices*. Eds. Gravenstein Highway North, Sebastopol, CA: USA: O’Reilly Media, 2015. Citado 7 vezes nas páginas 23, 35, 36, 38, 39, 41 e 42.
- RANNEY, M. *Explorations in Cooperative, Distributed Systems with Uber’s Ringpop*. 2015. Disponível em: <<http://www.slideshare.net/Codemotion/jeff-wolki-explorations-in-cooperative-distributed-systems-with-ubers-ringpop>>. Acesso em: 7 julho 2016. Citado na página 52.
- RICHARDSON, C. *Service Discovery in a Microservices Architecture*. 2015. Disponível em: <<https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>>. Acesso em: 7 julho 2016. Citado na página 26.
- RINGPOP Docs. 2015. Disponível em: <<https://ringpop.readthedocs.io/>>. Acesso em: 7 julho 2016. Citado 4 vezes nas páginas 47, 48, 49 e 52.

SALVADORI, I. L. *Desenvolvimento de Web APIs RESTful Semânticas Baseadas em JSON*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2015. Citado 4 vezes nas páginas 29, 30, 31 e 32.

STUBBS, J.; MOREIRA, W.; DOOLEY, R. Distributed systems of microservices using docker and serfnode. *7th International Workshop on Science Gateways. IEEE, jun 2015*, p. 34–39, 2015. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7217926>>. Acesso em: 24 maio 2016. Citado na página 23.

TCHANNEL Docs. 2015. Disponível em: <<https://tchannel.readthedocs.io/>>. Acesso em: 7 julho 2016. Citado na página 46.

W3C. *RDF Primer*. 2004. Disponível em: <<http://www.w3.org/TR/rdf-primer>>. Acesso em: 23 outubro 2016. Citado na página 32.

W3C. *RDFa 1.1 Primer - Second Edition: Rich structured data markup for web documents*. 2013. Disponível em: <<http://www.w3.org/TR/rdfa-primer>>. Acesso em: 23 outubro 2016. Citado na página 29.

WOLSKI, J. *Explorations in Cooperative, Distributed Systems with Uber's Ringpop*. 2015. Disponível em: <<http://www.slideshare.net/Codemotion/jeff-wolki-explorations-in-cooperative-distributed-systems-with-ubers-ringpop>>. Acesso em: 7 julho 2016. Citado 5 vezes nas páginas 47, 48, 49, 50 e 51.

9 Apêndices

9.1 APÊNDICE A - Códigos fonte

```
1 // Copyright (c) 2015 Uber Technologies, Inc.
2 //
3 // Permission is hereby granted, free of charge, to any person obtaining
4 //   a copy
5 // of this software and associated documentation files (the "Software"),
6 //   to deal
7 // in the Software without restriction, including without limitation the
8 //   rights
9 // to use, copy, modify, merge, publish, distribute, sublicense, and/or
10 //   sell
11 // copies of the Software, and to permit persons to whom the Software is
12 //   furnished to do so, subject to the following conditions:
13 //
14 // The above copyright notice and this permission notice shall be
15 //   included in
16 // all copies or substantial portions of the Software.
17
18 'use strict';
19
20 /* jshint maxparams:5 */
21
22 var assert = require('assert');
23 var util = require('util');
24 var deepExtend = require('deep-extend');
25 var fs = require('fs');
26 var path = require('path');
27 var setTimeout = require('timers').setTimeout;
28 var clearTimeout = require('timers').clearTimeout;
29 var Errors = require('tchannel/errors.js');
30 var TChannelJSON = require('tchannel/as/json');
31 var TChannelThrift = require('tchannel/as/thrift');
32 var TChannelEndpointHandler = require('tchannel/endpoint-handler');
33 var pull = require('lodash/pull');
34 var union = require('lodash/union');
35
36 HyperbahnHandler.MAX_RELAY_AD_ATTEMPTS = 2;
37 HyperbahnHandler.RELAY_AD_RETRY_TIME = 1 * 1000;
38 HyperbahnHandler.RELAY_AD_TIMEOUT = 500;
39 HyperbahnHandler.RELAY_FANOUT_TIMEOUT = 0;
40 HyperbahnHandler.RELAY_TIMEOUT = 500;
```

```
36
37 var thriftSource = fs.readFileSync(
38     path.join(__dirname, 'hyperbahn.thrift'), 'utf8'
39 );
40
41 module.exports = HyperbahnHandler;
42
43 // TODO: should be part of Hyperbahn error file
44 var TypedError = require('error/typed');
45 var NoPeersAvailable = TypedError({
46     type: 'hyperbahn.no-peers-available',
47     nameAsThrift: 'noPeersAvailable',
48     message: 'no peer available for {serviceName}',
49     serviceName: null
50 });
51
52 var InvalidServiceName = TypedError({
53     type: 'hyperbahn.invalid-service-name',
54     nameAsThrift: 'invalidServiceName',
55     message: 'invalid service name: {serviceName}',
56     serviceName: null
57 });
58
59 function HyperbahnHandler(options) {
60     if (!(this instanceof HyperbahnHandler)) {
61         return new HyperbahnHandler(options);
62     }
63
64     var self = this;
65     TChannelEndpointHandler.call(self, 'hyperbahn');
66
67     assert(options && options.channel, 'channel required');
68     assert(options && options.egressNodes, 'egressNodes required');
69     assert(options && options.callerName, 'callerName required');
70
71     // TODO support blacklist
72
73     self.channel = options.channel;
74     self.egressNodes = options.egressNodes;
75     self.callerName = options.callerName;
76
77     self.tchannelJSON = TChannelJSON({
78         logger: self.channel.logger
79     });
80
81     self.tchannelThrift = TChannelThrift({
82         channel: self.channel,
```

```
83     logger: self.channel.logger,
84     source: thriftSource
85   });
86
87   // TODO replace JSON with real bufrw handlers for this
88   self.tchannelJSON.register(self, 'ad', self,
89     self.handleAdvertise);
90   self.tchannelJSON.register(self, 'relay-ad', self,
91     self.handleRelayAdvertise);
92   self.tchannelJSON.register(self, 'unad', self,
93     self.handleUnadvertise);
94   self.tchannelJSON.register(self, 'relay-unad', self,
95     self.handleRelayUnadvertise);
96   self.tchannelJSON.register(self, 'serviceNames', self,
97     self.handleServiceNames);
98   self.tchannelJSON.register(self, 'relay-serviceNames', self,
99     self.handleRelayServiceNames);
100
101   self.tchannelThrift.register(self, 'Hyperbahn::discover', self,
102     self.discover);
103   self.tchannelThrift.register(self, 'Hyperbahn::discoverAffine', self
104     ,
105     self.discoverAffine);
106   self.tchannelThrift.register(self, 'Hyperbahn::tagDiscover', self,
107     self.tagDiscover);
108
109   self.relayAdTimeout = options.relayAdTimeout ||
110     HyperbahnHandler.RELAY_AD_TIMEOUT;
111   self.relayAdRetryTime = options.relayAdRetryTime ||
112     HyperbahnHandler.RELAY_AD_RETRY_TIME;
113   self.maxRelayAdAttempts = options.maxRelayAdAttempts ||
114     HyperbahnHandler.MAX_RELAY_AD_ATTEMPTS;
115
116   self.relayTimeout = options.relayTimeout || HyperbahnHandler.
117     RELAY_TIMEOUT;
118
119   self.relayFanoutTimeout = HyperbahnHandler.RELAY_FANOUT_TIMEOUT;
120   if (options.relayFanoutTimeout !== undefined) {
121     self.relayFanoutTimeout = options.relayFanoutTimeout;
122   }
123 }
124 util.inherits(HyperbahnHandler, TChannelEndpointHandler);
125
126 HyperbahnHandler.prototype.type = 'hyperbahn.advertisement-handler';
127
128 /* req: {
129     services: Array<{
```

```
128         serviceName: String,
129         cost: Number
130     }>
131 }
132
133 res: {
134     connectionCount: Number
135 }
136 */
137 HyperbahnHandler.prototype.handleAdvertise =
138 function handleAdvertise(handler, req, arg2, arg3, cb) {
139     handler.sendRelays(req, arg2, arg3, 'relay-ad', cb);
140 };
141
142 HyperbahnHandler.prototype.handleUnadvertise =
143 function handleUnadvertise(handler, req, arg2, arg3, cb) {
144     handler.sendRelays(req, arg2, arg3, 'relay-unad', cb);
145 };
146
147 HyperbahnHandler.prototype.sendRelays =
148 function sendRelays(req, arg2, arg3, endpoint, cb) {
149     /*eslint max-statements: [2, 30], max-params: [2, 6]*/
150     var self = this;
151     var services = arg3.services;
152     var finished = false;
153
154     var servicesByExitNode = {};
155
156     for (var i = 0; i < services.length; i++) {
157         var service = services[i];
158         service.hostPort = req.connection.remoteName;
159
160         var serviceNames = [service.serviceName];
161         if (service.serviceTags && service.serviceTags.length) {
162             serviceNames = serviceNames.concat(service.serviceTags);
163         }
164
165         for (var j = 0; j < serviceNames.length; j++) {
166             var serviceName = serviceNames[j];
167             if (serviceName.length === 0) {
168                 continue;
169             }
170
171             // TODO: cache / use sub-channel peers
172             var exitNodes = self.egressNodes.exitsFor(serviceName);
173             var exitHosts = Object.keys(exitNodes);
174
```

```
175         var amendedService = deepExtend({}, service, {serviceName:
176             serviceName});
177         for (var k = 0; k < exitHosts.length; k++) {
178             var exitNode = exitHosts[k];
179
180             var relayReq = servicesByExitNode[exitNode];
181             if (!relayReq) {
182                 relayReq = servicesByExitNode[exitNode] = [];
183             }
184
185             relayReq.push(amendedService);
186         }
187     }
188
189     var exitNodeKeys = Object.keys(servicesByExitNode);
190     var counter = 1 + exitNodeKeys.length;
191     for (var k = 0; k < exitNodeKeys.length; k++) {
192         var hostPort = exitNodeKeys[k];
193         var exitNodeServices = servicesByExitNode[hostPort];
194
195         self.sendRelay({
196             hostPort: hostPort,
197             services: exitNodeServices,
198             inreq: req,
199             endpoint: endpoint
200         }, onRelaySent);
201     }
202
203     var fanoutTimeout = null;
204     if (self.relayFanoutTimeout > 0) {
205         fanoutTimeout = setTimeout(finish, self.relayFanoutTimeout);
206     }
207
208     onRelaySent();
209
210     // TODO remove blocking on fanout finish. Requires fixing
211     // hyperbahn tests upstream
212     function onRelaySent() {
213         if (--counter <= 0) {
214             finish();
215         }
216     }
217
218     function finish() {
219         if (finished) {
220             return;
```

```
221     }
222     finished = true;
223     clearTimeout(fanoutTimeout);
224     cb(null, {
225         ok: true,
226         head: null,
227         body: {
228             connectionCount: exitNodeKeys.length
229         }
230     });
231 }
232 };
233
234 HyperbahnHandler.prototype.sendAdvertise =
235 function sendAdvertise(services, options, callback) {
236     var self = this;
237
238     if (typeof options === 'function') {
239         callback = options;
240         options = {};
241     }
242
243     options.serviceName = 'hyperbahn';
244     options.trace = false;
245     options.hasNoParent = true;
246     options.headers = options.headers || {};
247     options.headers.cn = self.callerName;
248
249     var req = self.channel.request(options);
250     self.tchannelJSON.send(req, 'ad', null, {
251         services: services
252     }, callback);
253 };
254
255 /* req: {
256     services: Array<{
257         serviceName: String,
258         hostPort: String,
259         cost: Number
260     }>
261 }
262
263 res: {}
264 */
265 HyperbahnHandler.prototype.handleRelayAdvertise =
266 function handleRelayAdvertise(handler, req, arg2, arg3, cb) {
267     handler.handleRelay('ad', req, arg2, arg3, cb);
```



```
268 };
269
270 HyperbahnHandler.prototype.handleRelayUnadvertise =
271 function handleRelayUnadvertise(handler, req, arg2, arg3, cb) {
272     handler.handleRelay('unad', req, arg2, arg3, cb);
273 };
274
275 HyperbahnHandler.prototype.handleRelay =
276 function handleRelay(endpoint, req, arg2, arg3, cb) {
277     var self = this;
278     var services = arg3.services;
279     var logger = self.channel.logger;
280
281     for (var i = 0; i < services.length; i++) {
282         var service = services[i];
283         if (endpoint === 'ad') {
284             self.advertise(service);
285         } else if (endpoint === 'unad') {
286             self.unadvertise(service);
287         } else {
288             logger.error('Unexpected endpoint for relay', {
289                 endpoint: endpoint,
290                 service: service
291             });
292         }
293     }
294
295     cb(null, {
296         ok: true,
297         head: null,
298         body: {}
299     });
300 };
301
302 HyperbahnHandler.prototype.sendRelay =
303 function sendRelay(opts, callback) {
304     var self = this;
305
306     var attempts = 0;
307
308     tryRequest();
309
310     // TODO: move functions out to methods
311     function tryRequest() {
312         attempts++;
313
314         self.channel.waitForIdentified({
```

```
315         host: opts.hostPort
316     }, onIdentified);
317
318     function onIdentified(err) {
319         if (err) {
320             return onResponse(err);
321         }
322
323         self.tchannelJSON.send(self.channel.request({
324             host: opts.hostPort,
325             serviceName: 'hyperbahn',
326             trace: false,
327             timeout: self.relayAdTimeout,
328             headers: {
329                 cn: opts.inreq.callerName || self.callerName
330             },
331             parent: opts.inreq,
332             retryFlags: {
333                 never: true
334             }
335         }), opts.endpoint, null, {
336             services: opts.services
337         }, onResponse);
338     }
339
340     function onResponse(err, response) {
341         if (response && response.ok) {
342             return callback(null, response);
343         }
344
345         var codeName = Errors.classify(err);
346         if (attempts <= self.maxRelayAdAttempts && err &&
347             (
348                 codeName === 'NetworkError' ||
349                 codeName === 'Timeout'
350             )
351         ) {
352             setTimeout(tryRequest, self.relayAdRetryTime);
353         } else {
354             self.logError(err, opts, response);
355
356             callback(err, null);
357         }
358     }
359 }
360 };
361
```

```
362 HyperbahnHandler.prototype.logError =
363 function logError(err, opts, response) {
364     var self = this;
365
366     var logger = self.channel.logger;
367
368     var logOptions = {
369         exitNode: opts.hostPort,
370         services: opts.services,
371         responseBody: response && response.body,
372         error: err
373     };
374
375     if (Errors.isFatal(err)) {
376         logger.error('Relay advertise failed with unexpected err',
377             logOptions);
378     } else {
379         logger.warn('Relay advertise failed with expected err',
380             logOptions);
381     }
382 };
383
384 HyperbahnHandler.prototype.advertise =
385 function advertise(service) {
386     var self = this;
387     self.channel.topChannel.handler.refreshServicePeer(service.
388         serviceName, service.hostPort);
389 };
390
391 HyperbahnHandler.prototype.unadvertise =
392 function unadvertise(service) {
393     var self = this;
394     self.channel.topChannel.handler.removeServicePeer(service.
395         serviceName, service.hostPort);
396 };
397
398 function convertHosts(hosts) {
399     var res = [];
400     for (var i = 0; i < hosts.length; i++) {
401         var strs = hosts[i].split(':');
402         var obj = {
403             port: parseInt(strs[1], 10)
404         };
405         strs = strs[0].split('.');
406         obj.ip = {
407             ipv4: parseInt(strs[3], 10) + (parseInt(strs[2], 10) << 8) +
408                 (parseInt(strs[1], 10) << 16) + (parseInt(strs[0], 10)
```

```

                << 24)
405         };
406
407         res.push(obj);
408     }
409
410     return res;
411 }
412
413 HyperbahnHandler.prototype.discover =
414 function discover(handler, req, head, body, cb) {
415     var serviceName = body.query.serviceName;
416     if (serviceName.length === 0) {
417         cb(null, {
418             ok: false,
419             body: InvalidServiceName({
420                 serviceName: serviceName
421             }),
422             typeName: 'invalidServiceName'
423         });
424         return;
425     }
426
427     var topChannel = handler.channel.topChannel;
428     var svcchan = topChannel.handler.getOrCreateServiceChannel(
429         serviceName);
430
431     if (svcchan.serviceProxyMode === 'forward' &&
432         req.headers.cn !== 'hyperbahn')
433     {
434         handler._forwardToRemoteDiscover(req, svcchan, body, cb);
435         return;
436     }
437     handler._getDiscoverHosts(serviceName, cb);
438 };
439
440 HyperbahnHandler.prototype.discoverAffine =
441 function discoverAffine(handler, req, head, body, cb) {
442     var serviceName = body.query.serviceName;
443     if (serviceName.length === 0) {
444         cb(null, {
445             ok: false,
446             body: InvalidServiceName({
447                 serviceName: serviceName
448             }),
449             typeName: 'invalidServiceName'

```

```
450     });
451     return;
452 }
453
454     handler._getDiscoverHosts(serviceName, cb);
455 };
456
457 HyperbahnHandler.prototype.tagDiscover =
458 function tagDiscover(handler, req, head, body, cb) {
459     var serviceTags = body.query.serviceTags;
460     var responses = {};
461     var finished = false;
462
463     if (!Array.isArray(serviceTags) || serviceTags.length === 0) {
464         cb(null, {
465             ok: false,
466             body: InvalidServiceName({
467                 serviceName: ''
468             }),
469             typeName: 'invalidServiceName'
470         });
471         return;
472     }
473
474     var counter = 1 + serviceTags.length;
475     for (var i = 0; i < serviceTags.length; i++) {
476         var serviceTag = serviceTags[i];
477
478         var topChannel = handler.channel.topChannel;
479         var svcchan = topChannel.handler.getOrCreateServiceChannel(
480             serviceTag);
481
482         if (svcchan.serviceProxyMode === 'forward' &&
483             req.headers.cn !== 'hyperbahn')
484         {
485             handler._forwardToRemoteDiscover(
486                 req,
487                 svcchan,
488                 {query: {serviceName: serviceTag}},
489                 onDiscoverSent.bind(null, serviceTag)
490             );
491             continue;
492         }
493
494         handler._getDiscoverHosts(serviceTag, onDiscoverSent.bind(null,
495             serviceTag));
496     }
497 }
```

```
495
496     onDiscoverSent();
497
498     function onDiscoverSent(serviceName, err, resp) {
499         if (!err && resp && resp.ok === true) {
500             responses[serviceName] = resp.body.peers;
501         }
502         if (--counter <= 0) {
503             finish();
504         }
505     }
506
507     function finish() {
508         if (finished) {
509             return;
510         }
511         finished = true;
512         cb(null, {
513             ok: true,
514             head: null,
515             body: {
516                 peers: responses
517             }
518         });
519     }
520 };
521
522 HyperbahnHandler.prototype._forwardToRemoteDiscover =
523 function _forwardToRemoteDiscover(parent, svcchan, body, cb) {
524     var self = this;
525     var serviceName = svcchan.serviceName;
526
527     var endpoint = 'Hyperbahn::discoverAffine';
528     // Since Hyperbahn is fully connected to service hosts,
529     // any exit node suffices.
530     self.tchannelThrift.send(svcchan.request({
531         serviceName: 'hyperbahn',
532         headers: {
533             cn: 'hyperbahn'
534         },
535         parent: parent,
536         retryFlags: {
537             never: true
538         },
539         timeout: 1000,
540         trace: false
541     }), endpoint, null, body, handleForward);
```

```
542
543     function handleForward(err, resp) {
544         if (err) {
545             self.channel.logger.error(
546                 'Failed to call discover API on exit node',
547                 parent.extendLogInfo({
548                     error: err,
549                     serviceName: serviceName
550                 })
551             );
552             return cb(err, null);
553         }
554
555         cb(null, resp);
556     }
557 };
558
559 HyperbahnHandler.prototype._getDiscoverHosts =
560 function _getDiscoverHosts(serviceName, cb) {
561     var self = this;
562
563     var hosts = [];
564     var svcchan = self.channel.topChannel.subChannels[serviceName];
565     if (svcchan) {
566         hosts = convertHosts(svcchan.peers.keys());
567     }
568
569     if (hosts.length === 0) {
570         cb(null, {
571             ok: false,
572             body: NoPeersAvailable({
573                 serviceName: serviceName
574             }),
575             typeName: 'noPeersAvailable'
576         });
577         return;
578     }
579
580     cb(null, {
581         ok: true,
582         body: {
583             peers: hosts
584         }
585     });
586 };
587
588 HyperbahnHandler.prototype.handleServiceNames =
```

```
589 function handleServiceNames(handler, req, head, body, cb) {
590     var nodes = handler.egressNodes.nodes();
591     var responses = [];
592     var finished = false;
593
594     var counter = 1 + nodes.length;
595     for (var i = 0; i < nodes.length; i++) {
596         handler.sendRelay({
597             hostPort: nodes[i],
598             inreq: req,
599             endpoint: 'relay-serviceNames'
600         }, onRelaySent);
601     }
602
603     onRelaySent();
604
605     function onRelaySent(err, resp) {
606         if (!err && resp && resp.ok === true) {
607             responses = union(responses, resp.body.serviceNames);
608         }
609         if (--counter <= 0) {
610             finish();
611         }
612     }
613
614     function finish() {
615         if (finished) {
616             return;
617         }
618         finished = true;
619         cb(null, {
620             ok: true,
621             body: {
622                 serviceNames: responses
623             }
624         });
625     }
626 };
627
628 HyperbahnHandler.prototype.handleRelayServiceNames =
629 function handleRelayServiceNames(handler, req, head, body, cb) {
630     var keys = Object.keys(handler.channel.topChannel.subChannels);
631     cb(null, {
632         ok: true,
633         body: {
634             serviceNames: pull(keys, 'autobahn', 'hyperbahn', 'ringpop')
635         }
636     });
637 }
```



```
636     });
637 };

1  // Copyright (c) 2015 Uber Technologies, Inc.
2  //
3  // Permission is hereby granted, free of charge, to any person obtaining
4  // a copy
5  // of this software and associated documentation files (the "Software"),
6  // to deal
7  // in the Software without restriction, including without limitation the
8  // rights
9  // to use, copy, modify, merge, publish, distribute, sublicense, and/or
10 // sell
11 // copies of the Software, and to permit persons to whom the Software is
12 // furnished to do so, subject to the following conditions:
13 //
14 // The above copyright notice and this permission notice shall be
15 // included in
16 // all copies or substantial portions of the Software.
17
18 'use strict';
19
20 var assert = require('assert');
21 var inherits = require('util').inherits;
22 var EventEmitter = require('tchannel/lib/event_emitter');
23 var MemberStatus = require('ringpop/lib/membership/member').Status;
24
25 module.exports = EgressNodes;
26
27 function EgressNodes(options) {
28     if (!(this instanceof EgressNodes)) {
29         return new EgressNodes(options);
30     }
31     var self = this;
32
33     assert(options && options.defaultKValue, 'defaultKValue required');
34
35     EventEmitter.call(self);
36
37     self.ringpop = null;
38     self.defaultKValue = options.defaultKValue;
39
40     self.kValueForServiceName = {};
41
42     // Surface the membership changed event (for use in particular by
43     // service
44     // proxies).
45     self.changedEvent = self.defineEvent('changed');
```

```
40 }
41
42 inherits(EgressNodes, EventEmitter);
43
44 EgressNodes.prototype.setRingpop = function setRingpop(ringpop) {
45     var self = this;
46
47     assert(self.ringpop === null, 'EgressNodes#setRingpop called twice')
48         ;
49
50     self.ringpop = ringpop;
51
52     self.ringpop.on('ringChanged', onRingChanged);
53     function onRingChanged() {
54         self.changedEvent.emit(self);
55     }
56 };
57
58 EgressNodes.prototype.kValueFor = function kValueFor(serviceName) {
59     var self = this;
60     return self.kValueForServiceName[serviceName] ||
61         self.defaultKValue;
62 };
63
64 EgressNodes.prototype.setDefaultKValue = function setDefaultKValue(
65     kValue) {
66     var self = this;
67     assert(typeof kValue === 'number' && kValue > 0, 'kValue must be
68         positive: ' + kValue);
69     self.defaultKValue = kValue;
70 };
71
72 EgressNodes.prototype.setKValueFor = function setKValueFor(serviceName,
73     k) {
74     var self = this;
75     self.kValueForServiceName[serviceName] = k;
76 };
77
78 EgressNodes.prototype.exitsFor = function exitsFor(serviceName) {
79     var self = this;
80
81     assert(
82         self.ringpop !== null,
83         'EgressNodes#exitsFor cannot be called before EgressNodes has '
84         +
85         ' ringpop set'
86     );
87 }
```

```
82
83     var k = self.kValueFor(serviceName);
84     // Object<hostPort: String, Array<lookupKey: String>>
85     var exitNodes = Object.create(null);
86     for (var i = 0; i < k; i++) {
87         var shardKey = serviceName + '~' + i;
88
89         // TODO ringpop will return itself if it cannot find
90         // it which is probably the wrong semantics.
91         var node = self.ringpop.lookup(shardKey);
92
93         // TODO ringpop can return duplicates. do we want
94         // <= k exitNodes or k exitNodes ?
95         // TODO consider walking the ring instead.
96
97         exitNodes[node] = exitNodes[node] || [];
98         exitNodes[node].push(shardKey);
99     }
100     return exitNodes;
101 };
102
103 EgressNodes.prototype.isExitFor = function isExitFor(serviceName) {
104     var self = this;
105
106     assert(
107         self.ringpop !== null,
108         'EgressNodes#isExitFor cannot be called before EgressNodes has '
109         +
110         ' ringpop set'
111     );
112
113     var k = self.kValueFor(serviceName);
114     var me = self.ringpop.whoami();
115     for (var i = 0; i < k; i++) {
116         var shardKey = serviceName + '~' + i;
117         var node = self.ringpop.lookup(shardKey);
118         if (me === node) {
119             return true;
120         }
121     }
122     return false;
123 };
124 EgressNodes.prototype.nodes = function nodes() {
125     var self = this;
126
127     assert(
```

```
128     self.ringpop != null ,
129     'EgressNodes#isExitFor cannot be called before EgressNodes has '
130         +
131         ' ringpop set'
132 );
133 var members = self.ringpop.membership.members;
134 var hosts = [];
135 for (var i = 0; i < members.length; i++) {
136     var member = members[i];
137     if (member.status === MemberStatus.alive) {
138         hosts.push(member.address);
139     }
140 }
141 return hosts;
142 };
```

9.2 APÊNDICE B – Artigo Descoberta Semântica de Microservices em Contêineres

Descoberta Semântica de Microservices em Contêineres

Lucas M. Barrozo¹

¹Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

barrozo.lucas@graf.ufsc.br

Abstract. *The Web has been increasing as well as the quantity of services available in it. What makes essential to know where these services are located. Several technologies have appeared in the market to solve this problem, especially those that focus on dynamic service registration. This work proposes the adaptation of one of these discovery mechanisms, Hyperbahn, changing the way services are registered and queried. Instead of just one name, services can be identified by multiple registry entries. Such modifications will allow the modified tool to have its use adapted to give more meaning to the discovery. But to really give meaning to services, some semantic Web concepts must be used. It will be established that the registry entries of a microservice will represent concepts or properties of an ontology. Finally, for the purpose of comparison and validation, experiments will be performed on the delivered solution and on the original implementation. With the use of the proposed solution it is expected to provide an environment that enhances the characteristics employed by the use of microservices.*

Resumo. *A Web vem aumentando cada vez mais suas proporções. De natureza igual é o crescimento da quantidade de serviços disponíveis nela. Isso torna indispensável saber onde esses serviços estão localizados. Diversas tecnologias surgiram no mercado para solucionar esse problema, principalmente as que focam no registro de serviço dinâmicos. Este trabalho propõe a adaptação de um desses mecanismos de descoberta, o Hyperbahn, alterando a forma como os serviços são registrados e consultados. Ao invés de apenas um nome, serviços poderão ser identificados por várias entradas de registro. Tais modificações darão liberdade para a ferramenta modificada ter seu uso adaptado para dar mais significado à descoberta. Significado aos serviços será alcançado utilizando alguns conceitos da Web semântica. Será estabelecido que as entradas de registro de um microservice representem conceitos ou propriedades de uma ontologia. Por fim, com o propósito de comparação e validação, experimentos serão realizados na solução entregue e na implementação original. Com a utilização da solução proposta espera-se proporcionar um ambiente que reforce ainda mais as características empregadas pelo uso de microservices.*

1. Introdução

A última década trouxe grandes mudanças para a tecnologia da informação em diversas frentes. Dispositivos móveis, proliferação de dados, virtualização e *cloud computing* são apenas alguns dos grandes avanços. Como resultado temos a complexidade dos sistemas computacionais crescendo de forma rápida e quase que totalmente sem freios

[Stubbs et al. 2015]. O típico sistema Web agora mantém múltiplos componentes: um *web front-end*, um gerenciador de acesso, *workers* assíncronos, plataforma de *analytics*, sistema de logs, etc. A arquitetura de micro serviços substitui a monolítica por um sistema distribuído de serviços leves, independentes e de propósito único, o que torna os sistemas cada vez mais granulares, impactando diretamente na quantidade de serviços disponíveis.

Apesar da forma simples e objetiva com que conseguimos descrever a arquitetura de micro serviços, não conseguimos ter a mesma simplicidade na prática, que acaba se mostrando desafiadora. A simples tarefa de implantação, que pode ser realizada trivialmente na arquitetura monolítica, se torna extremamente complexa nesses novos moldes. Enquanto a implantação e distribuição de micro serviços pode ser simplificada utilizando *linux containers*, pouco eles fazem para solucionar o problema de comunicação entre serviços, o que se torna um problema bem evidente quando a aplicação começa a tomar grandes proporções. Em um ambiente regido pela arquitetura de micro serviços sabemos que uma instância redundante pode ser criada ou destruída sob demanda e que serviços podem não estar disponíveis em um determinado momento, portanto serviços consumidores precisam de um mecanismo para localizar serviços produtores em tempo real.

Diversas tecnologias surgiram no mercado para solucionar esse problema, principalmente as que focam no registro de serviço dinâmicos. Muitas dessas tecnologias apenas mapeiam um endereço de rede a um nome identificador, o que agrega muito pouco na hora de descrever suas características funcionais dos serviços disponibilizados.

Este artigo propõe adaptações em uma dessas tecnologias de registro de serviço dinâmicos para descobrir funcionalidades e não somente mapear servidores para um nome específico. Mesmo que simples, essas mudanças darão mais liberdade para realizar a migração de um determinado conjunto de serviços para um novo local ou para realizar integração entre serviços consumidores e provedores. Com esse aprimoramento, passa a ser irrelevante por parte dos desenvolvedores conhecer exatamente os serviços que uma determinada instância de um *microservice* oferece, devido ao novo significado de registro e consulta de serviços.

Diversas tecnologias podem ser adaptadas, já que muitas alcançam os mesmos resultados. Neste trabalho foi escolhido o Hyperbahn, pois é uma ferramenta que está bem consolidada e é bastante utilizada na indústria. É importante ressaltar que a descoberta se dá em um ambiente contido, ou seja, somente *microservices* pertencentes a esse ambiente podem utilizar os mecanismos de descoberta e aproveitar as características que o Hyperbahn disponibiliza.

O resto deste trabalho está organizado da seguinte forma. A Seção 2 apresenta alguns fundamentos da Web semântica. A Seção 3 apresenta a arquitetura de *microservices*. O Hyperbahn, que foi a ferramenta de descoberta de serviço escolhida, é apresentado na Seção 4. A solução proposta é apresentada na Seção 5. A Seção 6 apresenta experimentos comparativos entre a solução proposta e a implementação original. A Seção 7 conclui esse artigo.

2. Web Semântica

Grande parte do conteúdo disponibilizado na Web é destinado ao consumo humano, impossibilitando o processamento e interpretação automáticos por parte de computadores.

Assim, o significado das informações publicadas não é considerado em tarefas comuns executadas na Web, como a busca, por exemplo, resultando em ambiguidade e baixa relevância [Filho 2009].

O propósito da Web Semântica é decorar as informações publicadas na Web através de anotações semânticas formais, o que minimiza as limitações da Web como a conhecemos atualmente. Isso faz com que sintaxe e semântica passem a compartilhar o mesmo grau de importância nos processos de publicação, busca e recuperação de informações [Berners-Lee et al. 2001].

2.1. Linked data

Conforme [Bizer et al. 2009], a Web em seu início era denominada de “Web de Hiper-textos”, onde as informações eram somente em formato digital e poderiam conter textos, imagens e multimídia (som e vídeo), permitindo ligação a outros textos por hiperlinks. Em seguida, evoluiu para “Web de Dados”, onde as informações publicadas eram melhor estruturadas a fim de que os agentes de software pudessem interpretá-las. Esta estrutura é feita através da descrição de suas prioridades e conceitos, utilizando um meio de tecnologia padronizada, com o objetivo de melhor integrar e reutilizar dados, que é a base do *Linked Data*.

Abaixo seguem as quatro regras criadas por Berners-Lee (2006) para possibilitar o reuso de informações:

- Identificar recursos através de URIs, com um identificador único de recurso;
- Associar um endereço HTTP aos recursos, permitindo que sejam localizados na Web;
- Estruturar os dados com tecnologias padronizadas e recomendadas para Web Semântica, além de estabelecer contextos sobre o relacionamento entre os recursos;
- Interligar diferentes recursos.

A Web Semântica é uma extensão da Web atual onde os dados possuem um significado associado, criando a visão de uma rede de informações interligadas, o que possibilita que pessoas e computadores trabalhem de forma cooperativa [Berners-Lee et al. 2001]. Em outras palavras, a Web Semântica retira dos seres humanos a exclusividade de interpretação dos dados [Salvadori 2015].

Tradicionalmente, os documentos publicados são formados por um título, alguns parágrafos de texto, frequentemente com imagens ou vídeos e alguns links para direcionar a outros documentos [W3C 2013]. Essas informações só podem ser compreendidas por humanos. Já os computadores estão limitados a interpretar esses dados como textos, links e imagens sem sentido nenhum.

3. Microservices

O termo *microservices* difundiu-se nos últimos anos para representar uma nova forma de projetar aplicações como um conjunto de serviços autônomos.

Sucintamente, a arquitetura de *microservices* é uma abordagem que visa desenvolver uma aplicação como um conjunto de pequenos serviços, cada um rodando em seu próprio processo e se comunicando de forma simples [Newman 2015]. Esses serviços são construídos ao redor das necessidades do negócio e são implantados independentemente.

Nesse tipo de arquitetura a heterogeneidade reina, de tal modo que serviços podem ser escritos em diferentes linguagens e podem utilizar diferentes tecnologias de armazenamento [Fowler and Lewis 2014].

Abaixo seguem algumas características da arquitetura:

- Componentização via serviços
- Independência
- Heterogeneidade tecnológica
- Resiliência
- Escalabilidade

4. Hyperbahn

Hyperbahn é uma rede sobreposta de roteadores, baseada na biblioteca Ringpop (4.2), projetados para dar suporte ao protocolo RPC TChannel (4.1), sendo essas tecnologias todas criadas pela *Uber Technologies Inc.*. Seus nodos de roteamento dinamicamente convergem e decidem através de um protocolo epidêmico os serviços conhecidos, armazenando-os em um *consistent hashing ring*, formando uma rede de serviços prontos para se comunicarem uns com os outros sem intervenção de um desenvolvedor e sem a necessidade de explicitamente especificar portas e endereços de rede.

O Hyperbahn habilita o roteamento e a descoberta de serviço em sistemas de larga escala compostos de diversos *microservices*. Funciona em ambiente distribuído, provê tolerância a falhas e prioriza a disponibilidade dos serviços. Permite que um serviço descubra e se comunique com outros de forma segura e simples, sem a necessidade de saber onde esses outros serviços estão efetivamente rodando.

4.1. TChannel

O TChannel proporciona um protocolo para clientes e servidores com uma rede de roteamento inteligente (Hyperbahn) entre eles [Technologies 2015b].

Ele foi criado devido a diversos problemas gerados ao se manter um sistema distribuído com uma grande quantidade de *microservices* disponíveis. Dentre esses problemas, os mais comuns são a descoberta de serviço, tolerância a falhas e o rastreamento de requisições. Esses problemas são resolvidos da seguinte maneira:

- **descoberta de serviço** - Todos os produtores e consumidores se registram na rede de roteamento. Consumidores só podem acessar produtores através de um nome, sem a necessidade de conhecer portas ou endereços de rede;
- **tolerância a falhas** - A rede de roteamento mantém registro de falhas e pode inteligentemente detectar *hosts* defeituosos e os remover do grupo de *hosts* disponíveis;
- **rastreamento de requisições** - Inseridas no protocolo como *first class citizen*, ou seja, o objeto de rastreamento é passado adiante no encaminhamento de requisições.

4.2. Ringpop

Ringpop é uma biblioteca que mantém um *consistent hash ring* e pode ser usada para arbitrariamente particionar os dados de uma aplicação de forma escalável e tolerante a falhas [Technologies 2015a].

A biblioteca tem como características fundamentais: um protocolo de *membership*, um *consistent hashing ring* e o encaminhamento de requisições.

O seu protocolo de *membership* proporciona uma aplicação distribuída, atribuindo a suas instâncias, que anteriormente desconheciam seus pares, a capacidade de se descobrir, se auto-organizar e cooperar. Essas instâncias se comunicam utilizando um protocolo de *Gossip* diretamente por TCP e compartilham informações para entrar em acordo sobre os membros que fazem parte da aplicação distribuída.

Com uma lista consistente de membros definidos através do protocolo de *membership*, a biblioteca organiza esses membros em um *consistent hash ring*, que define um comportamento previsível com relação à distribuição dos dados dentro da aplicação.

O Ringpop não é um sistema externo ou mesmo um recurso infraestrutural compartilhado por diversas aplicações, mas sim uma biblioteca de desenvolvimento que funciona como uma camada da aplicação, que permite à aplicação se manter independente. Devido às suas características, o Ringpop promove escalabilidade e resiliência a falhas, mantendo complexidade e custos operacionais a um mínimo [Technologies 2015a].

O uso do Ringpop pode passar completamente despercebido para usuários da aplicação. Não existe a necessidade dos clientes conhecerem os mecanismos por trás do esquema de particionamento ou mesmo o real destinatário de uma requisição. Balanceadores de carga, *proxies*, redes sobrepostas podem continuar sendo utilizadas sem gerar incompatibilidades.

Ringpop também vem acompanhado de uma API de gerenciamento para inspecionar e controlar a aplicação e todo um ferramental para ajudar a entender o comportamento da biblioteca e da aplicação.

5. Proposta

Este trabalho propõe modificações no mecanismo de descoberta de serviços oferecidos pelo Hyperbahn. As modificações propostas consideram o uso de uma semântica leve para tratar o problema de registros de serviços.

A descoberta de serviço, como apresentada pela ferramenta original, relaciona um *microservice* a apenas um nome, o que é muito pouco para descrever suas características funcionais. Diferentemente, a solução proposta possibilita que um mesmo *microservice* seja reconhecido por diversos nomes. Porém, isso não é o suficiente para atribuir significado à descoberta, exigindo que alguns conceitos da Web semântica sejam utilizados. Como definido na proposta, URIs RDF foram utilizadas como entradas de registro para representar características dos *microservices*, dando um significado para os dados que retornam em uma requisição de descoberta.

Na solução entregue, o Hyperbahn não implementa um motor de inferência, fazendo com que os clientes consumidores tenham a responsabilidade de inferir quais serviços disponíveis na rede se encaixam nas suas necessidades. Para descobrir quais serviços estão disponíveis, um meio de se obter todas as entradas de registro foi disponibilizado na implementação modificada.

De forma geral, o fluxo de uso de uma aplicação cliente pode ser representado da seguinte forma. Após o registro (1), a aplicação cliente precisa descobrir quais serviços

estão sendo disponibilizados na rede. Para isso, todas as entradas de registro cadastradas no Hyperbahn são obtidas (2). Uma vez com todas as entradas de registro em mãos, a aplicação cliente infere quais entradas suprem as suas necessidades (3). Em seguida, a funcionalidade de descoberta baseada em múltiplos nomes é utilizada (4) tendo como resultado os *hosts* cadastrados para as entradas de registro. Esse fluxo está ilustrado na figura 1.

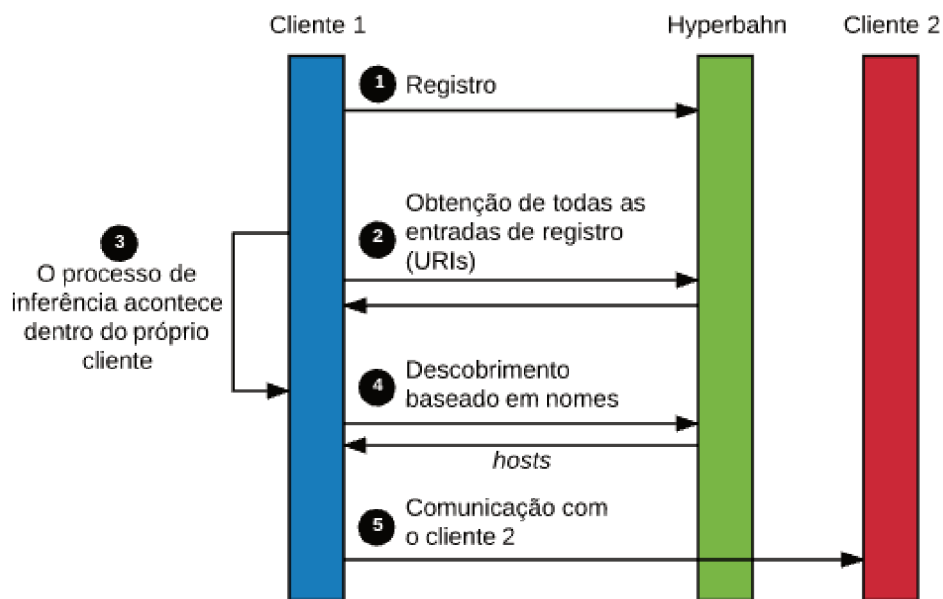


Figura 1. Fluxo de uso da solução proposta

5.1. Implementação

Pelo fato do trabalho apresentado tratar-se de adicionar novas funcionalidades a uma ferramenta já existente, mantiveram-se os padrões já existentes no código fonte da aplicação original, assim como sua estrutura de entidades. Para não limitar o escopo de uso da ferramenta modificada somente à solução proposta, o termo “*tags*” foi utilizado no código.

Como já citado, foram necessárias mudanças na forma como aplicações de registram e como são consultadas no *Hyperbahn*, assim como uma forma de obter todas as entradas de registro já cadastradas.

5.1.1. Registro de serviço

Para permitir que um *microservice* seja identificado por múltiplos nomes, ao invés de apenas um, foi incorporada a propriedade *serviceTags* (linha 15) na requisição de registro na biblioteca cliente do *Hyperbahn* (1).

Além disso, para suportar o registro simultâneo de milhares de nomes, foi necessário que esses nomes, que estão representados no código como *tags*, fossem mandadas em lotes de 30 (linha 5), preservando o consumo de I/O e processamento.

```

1 HyperbahnClient.prototype.sendBatchRequests =
2 function sendBatchRequests(opts, endpoint, cb) {
3     var self = this;
4     var finished = false;
5     var chunk = DEFAULT_REQ_DATA_CHUNK_SIZE;
6     var counter = Math.ceil(self.serviceTags.length / chunk);
7
8     var i, j;
9     for (i = 0, j = self.serviceTags.length; i < j; i += chunk) {
10        var req = self.newRequest();
11        self.tchannelJSON.send(req, endpoint, null, {
12            services: [{
13                cost: 0,
14                serviceName: self.serviceName,
15                serviceTags: self.serviceTags.slice(i, i + chunk),
16            }]
17        }, sendRequestInternalCb);
18    }
19
20    // Código para enviar a resposta ao consumidor
21    ...
22 };

```

Listagem 1. Código executados por clientes para se registrar no mecanismo de descoberta de serviço

No servidor foram feitas modificações (2) para dar suporte à nova propriedade adicionada (linhas 14 a 17). Além de ser feito o registro distribuído do nome (*serviceName*) fornecido, as *serviceTags* também foram incluídas e registradas (linhas 19 a 40).

```

1 HyperbahnHandler.prototype.sendRelays =
2 function sendRelays(req, arg2, arg3, endpoint, cb) {
3     var self = this;
4     var services = arg3.services;
5     var finished = false;
6
7     var servicesByExitNode = {};
8
9     for (var i = 0; i < services.length; i++) {
10        var service = services[i];
11        service.hostPort = req.connection.remoteName;
12
13        var serviceNames = [service.serviceName];
14        if (service.serviceTags && service.serviceTags.length) {
15            serviceNames = serviceNames.concat(service.serviceTags);
16        }
17
18        for (var j = 0; j < serviceNames.length; j++) {
19            var serviceName = serviceNames[j];
20            if (serviceName.length === 0) {
21                continue;
22            }
23
24            // Obtenção dos nodos do Hyperbahn que mapeiam para um
                determinado nome

```

```

25     var exitNodes = self.egressNodes.exitsFor(serviceName);
26     var exitHosts = Object.keys(exitNodes);
27
28     var amendedService = deepExtend({}, service, {serviceName:
29     serviceName});
30     for (var k = 0; k < exitHosts.length; k++) {
31         var exitNode = exitHosts[k];
32
33         var relayReq = servicesByExitNode[exitNode];
34         if (!relayReq) {
35             relayReq = servicesByExitNode[exitNode] = [];
36         }
37
38         relayReq.push(amendedService);
39     }
40 }
41
42 // Código para enviar nome dos serviços para os outros nodos
43 // presentes no cluster do Hyperbahn
44 ...
45
46 // Código para lidar com as respostas das requisições enviadas aos
47 // outros nodos do Hyperbahn no cluster
48 ...
49
50 // Código para enviar a resposta ao consumidor
51 ...
52 }
53 };

```

Listagem 2. Código que lida com requisições de registro

5.1.2. Consulta de serviço baseada em múltiplos nomes

Para habilitar a consulta baseada em múltiplos nomes, foi criado um novo serviço (*TagDiscovery*).

Como a comunicação entre cliente e *Hyperbahn* se dá através de chamadas de procedimento remota descritas através do *Thrift*, foram necessária algumas mudanças nessa descrição.

Foram criadas novas *structs* que definem os parâmetros da requisição (Listagem 3) e o formato da resposta (Listagem 4) e, por fim, a definição do serviço (Listagem 5).

```

1 struct TagDiscoveryQuery {
2     1: required list<string> serviceTags
3 }

```

Listagem 3. *TagDiscovery*: parâmetros - Thrift

```

1 struct TagDiscoveryResult {
2     1: required map<string, list<ServicePeer>> peers
3 }

```

Listagem 4. *TagDiscovery*: resposta - Thrift

```
1 TagDiscoveryResult tagDiscover(  
2     1: required TagDiscoveryQuery query  
3     ) throws (  
4         1: NoPeersAvailable noPeersAvailable  
5     )
```

Listagem 5. *TagDiscovery*: serviço - Thrift

No servidor foi criado um método para lidar com as requisições do serviço de *TagDiscovery* recém criado.

Para cada nome a ser consultada, é utilizado o serviço *discover* já implementado na aplicação original (linhas 12 a 29 na Listagem 6). Aqui é utilizado um dos conceitos já explicados do Ringpop, o Forwarding. Caso um nome não mapear para o nodo que está processando a requisição em questão, a mesma é encaminhada para o nodo que o consiga. Uma vez que o processo de descoberta termina, uma resposta é encaminhada ao serviço consumidor (Listagem 6).

```
1 HyperbahnHandler.prototype.tagDiscover =  
2 function tagDiscover(handler, req, head, body, cb) {  
3     var serviceTags = body.query.serviceTags;  
4     var responses = {};  
5     var finished = false;  
6  
7     // Codigo validando parametros da requisicao  
8     ...  
9  
10    var counter = 1 + serviceTags.length;  
11    for (var i = 0; i < serviceTags.length; i++) {  
12        var serviceTag = serviceTags[i];  
13  
14        var topChannel = handler.channel.topChannel;  
15        var svcchan = topChannel.handler.getOrCreateServiceChannel(  
16            serviceTag);  
17  
18        if (svcchan.serviceProxyMode === 'forward' &&  
19            req.headers.cn !== 'hyperbahn'  
20        ) {  
21            handler._forwardToRemoteDiscover(  
22                req,  
23                svcchan,  
24                {query: {serviceName: serviceTag}},  
25                onDiscoverSent.bind(null, serviceTag)  
26            );  
27            continue;  
28        }  
29        handler._getDiscoverHosts(serviceTag, onDiscoverSent.bind(null,  
30            serviceTag));  
31    }
```

```

32 // Codigo para lidar com as respostas da chamada de "discover"
33 ...
34
35 // Codigo para enviar a resposta ao consumidor
36 ...
37 };

```

Listagem 6. Processo de descoberta baseada em tags

5.1.3. Obtenção de todas as entradas de registro

Para obter todas as entradas de registros cadastradas no *Hyperbahn*, foi necessário criar mais um serviço primário e um outro auxiliar.

Inicialmente foram obtidos os endereços de todos os nodos “vivos” (linha 8) do *Hyperbahn* inseridos na listagem de membros (linha 4) presente no Ringpop (Listagem 7).

```

1 EgressNodes.prototype.nodes = function nodes() {
2   var self = this;
3
4   var members = self.ringpop.membership.members;
5   var hosts = [];
6   for (var i = 0; i < members.length; i++) {
7     var member = members[i];
8     if (member.status === MemberStatus.alive) {
9       hosts.push(member.address);
10    }
11  }
12  return hosts;
13 };

```

Listagem 7. Obtendo membros “vivos” do Ringpop

Em seguida se deu a criação do serviço primário *serviceNames*, que tem como objetivo obter as entradas de registro armazenadas nos nodos listados pelo método citado anteriormente.

Para cada membro do *cluster* é feita uma requisição ao serviço auxiliar *relay-serviceNames* (linhas 7 a 14 na Listagem 8). Lembrando que essas requisições são feitas e processadas paralelamente.

```

1 HyperbahnHandler.prototype.handleServiceNames =
2 function handleServiceNames(handler, req, head, body, cb) {
3   var nodes = handler.egressNodes.nodes();
4   var responses = [];
5   var finished = false;
6
7   var counter = 1 + nodes.length;
8   for (var i = 0; i < nodes.length; i++) {
9     handler.sendRelay({
10      hostPort: nodes[i],
11      inreq: req,

```

```

12         endpoint: 'relay-serviceNames'
13     }, onRelaySent);
14 }
15
16 // Código para lidar com as respostas das requisições enviadas aos
17 // nodos "vivos" do cluster
18 ...
19 // Código para enviar a resposta ao consumidor
20 ...
21 };

```

Listagem 8. Obtendo entradas de registro cadastradas em todos os nodos pertencentes ao *cluster* do Hyperbahn

Finalmente, o serviço auxiliar, que ao ser executado pelos nodos pares, retorna todas as entradas de registro cadastradas em cada instância, como pode ser visualizado na Listagem 9.

```

1 HyperbahnHandler.prototype.handleRelayServiceNames =
2 function handleRelayServiceNames(handler, req, head, body, cb) {
3     var keys = Object.keys(handler.channel.topChannel.subChannels);
4     cb(null, {
5         ok: true,
6         body: {
7             serviceNames: pull(keys, 'autobahn', 'hyperbahn', 'ringpop'
8                 )
9         }
10    });
11 };

```

Listagem 9. Obtendo entradas de registro cadastradas em um nodo

5.2. Estudo de caso

Este capítulo apresenta um estudo de caso com o objetivo de exemplificar a aplicação da solução proposta. O estudo de caso considera os conceitos presentes na ontologia ilustrada na figura 2 e por *microservices* que manipulam as entidades associadas a esses conceitos.

A ontologia utilizada no estudo de caso é constituída por seis classes semânticas: Obra, Filme, Livro, Pintura, Gênero e Terror.

A estrutura de *microservices* presente no servidor é representada pela figura 3. Fazem parte dessa estrutura os *microservices* que manipulam as entidades representantes dos conceitos da ontologia exemplo, um *API Gateway* responsável por agregar informações dos demais *microservices* e entregá-las ao usuário e pelo Hyperbahn.

Os *microservices*, que foram apresentados para o estudo de caso, após serem implantados serão registrados no Hyperbahn resultando na configuração de registro mostrada na tabela 1 e, seguindo a solução proposta, as entradas de registro devem representar conceitos ou propriedades de classes semânticas da ontologia.

Como dito anteriormente, cada *microservice* opera sobre entidades de um ou mais conceitos da ontologia.

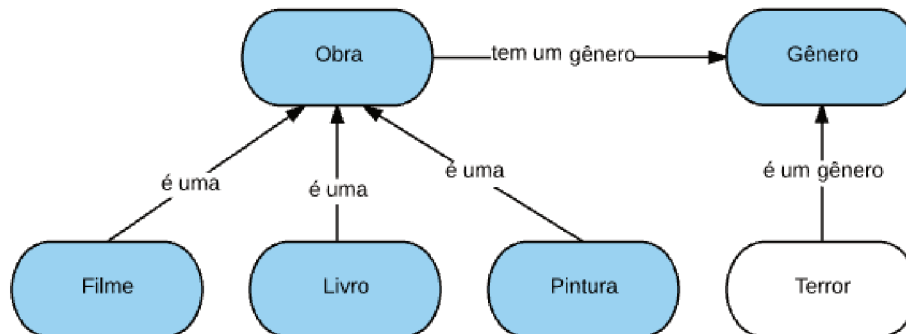


Figura 2. Ontologia de exemplo para o estudo de caso

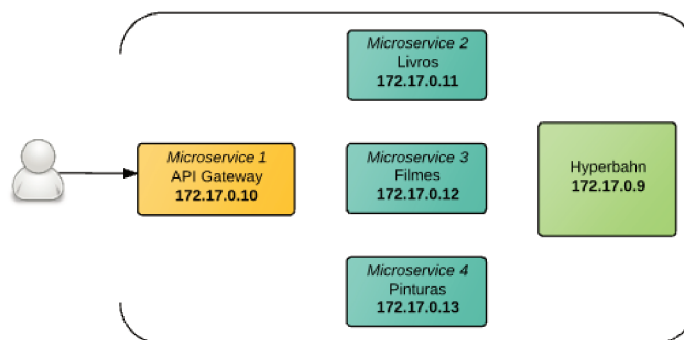


Figura 3. Estrutura de *microservices* presente no estudo de caso

- O *API Gateway* manipula entidades presentes em todas as hierarquias de Obra.
- *Microservice 2* manipula livros de qualquer gênero.
- *Microservice 3* manipula apenas filmes de terror.
- *Microservice 4* manipula pinturas de qualquer gênero.

O *API Gateway* é o único *microservice* que recebe estímulos externos provenientes do usuário. Nele são disponibilizados serviços que agregam informações armazenadas nos demais *microservices* registrados no Hyperbahn.

A partir de uma requisição de usuário, o processo de descoberta começa primeiramente no *API Gateway*, onde todas as entradas de registro são requisitadas ao Hyperbahn. Em seguida é executado um processo de inferência sobre as entradas de registro obtidas, que pode ser executado facilmente por ferramentas como o Apache Jena. A inferência tem como objetivo principal explorar os relacionamentos hierárquicos definidos na ontologia de acordo com as entidades gerenciadas pelos *microservices*. As entradas de registro resultantes do processo de inferência são utilizadas para descobrir o *host* dos *microservices* responsáveis por manipular tais conceitos. Finalmente, as informações armazenadas nos *microservices* descobertos serão agregadas pelo *API Gateway* e devolvidas ao usuário.

Para fins de exemplificação, suponha que existe um usuário que está buscando informações sobre obras de terror e ele fez uma requisição ao serviço no *API Gateway* capaz

ID	Entradas de registro	Endereço de IP
1	“<Thing> a <Obra>”	172.17.0.10
2	“<Obra> a <Livro>”	172.17.0.11
2	“<Livro> <temUmGenero> <Terror>”	172.17.0.11
3	“<Filme> <temUmGenero> <Terror>”	172.17.0.12
4	“<Obra> a <Pintura>”	172.17.0.13

Tabela 1. Estudo de caso - Entradas de registro no Hyperbahn

de devolver esses dados. Nesse serviço a relação “<Obra> <temUmGenero> <Terror>” é utilizada como a base do processo de inferência hierárquica sobre todas as entradas de registro obtidas do Hyperbahn. Esse processo resulta em apenas duas entradas de registros, que são “<Livro> <temUmGenero> <Terror>” e “<Filme> <temUmGenero> <Terror>”, que suprem as necessidades do usuário. Em seguida as entradas inferidas serão utilizadas para descobrir os *microservices* (“172.17.0.11” e “172.17.0.12”) associados a elas no Hyperbahn. Por fim, as informações serão obtidas do *microservices* descobertos.

6. Experimentos e Resultados

Este capítulo apresenta as análises e validações para a proposta do trabalho. A validação da proposta é realizada através da comparação de desempenho entre a implementação original e a modificada, levando em consideração a incorporação de conceitos semânticos à consulta e ao registro de serviços.

6.1. Experimentos

Para realizar a validação da implementação proposta, é necessário que os serviços, tanto consumidores quanto produtores, sejam implantados em um ambiente computacional, de modo a possibilitar a comunicação com as aplicações clientes. O servidor possui o sistema operacional CoreOS 1122.2.0 (*stable*), instalado em um *cloud server* na Digital Ocean. A instância escolhida possui dois processadores Intel Xeon operando a 2,4 GHz e 4GB de memória principal.

O ambiente de execução já possui como estratégia de implantação Linux *containers*. O sistema operacional conta com o Docker versão 1.10.3, que será utilizado para auxiliar na implantação das aplicações. O uso de Linux *containers* é de grande importância pois traz uma série de vantagens na execução dos experimentos, como a rápida mudança de parâmetros para a realização de testes e comparações, assim como facilita a troca entre a solução proposta e a implementação original.

Para os testes de desempenho um conjunto de triplas RDF foi obtido no website DBpedia¹ para apoiar a execução do estudo. Cerca de 136,0 KB de dados, que contabilizam 1000 triplas RDF, foram armazenados em um arquivo, que foi carregado pelas aplicações cliente no momento adequado.

O próximo passo foi adaptar o *Hyperbahn* para ser implantado em um *container* e torná-lo disponível para as aplicações consumidoras. Foram criadas duas imagens Doc-

¹<http://dbpedia.org>

ker², uma contendo a implementação original e outra a solução proposta, o que facilitou a troca de uma para outra quando necessário. Lembrando que não foi utilizado nenhum serviço externo para o armazenamento de imagens Docker. Da mesma forma, as aplicações cliente também foram implantadas em contêineres.

Foi criado um *script* para automatizar a execução dos cenários. Para cada teste o ambiente foi reiniciado e todos os seus dados apagados. Cada cenário foi executado 10 vezes, de modo a possibilitar o cálculo de uma média de tempo de resposta após a execução das requisições.

Os fatores variáveis escolhidos para a realização dos experimentos foram a quantidades de nodos do *cluster* e o número de réplicas de uma determinada entrada (k-value).

6.2. Resultados

A tabela 2 apresenta os dados obtidos na execução da requisição de registro. Para obter essa média na implementação original foi necessário executar a requisição diversas vezes para simular a funcionalidade de registro por múltiplos nomes implementada na solução proposta. Mesmo com as requisições de registro para a implementação original sendo executadas paralelamente, não foi possível obter bons resultados para o tempo de resposta em comparação com implementação proposta.

Número de nodos no <i>cluster</i>	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (<i>original</i>)	Tempo de resposta (ms) (<i>proposta</i>)
2	1	2185,8	355,6
2	5	3737,9	527,5
2	10	3553,5	636,0
3	1	2508,3	387,5
3	5	3778,7	1039,5
3	10	3979,1	1072,3
4	1	2678,0	420,3
4	5	4980,6	1137,8
4	10	4840,5	1841,4
5	1	2847,2	447,8
5	5	4434,3	1854,0
5	10	5985,0,6	2970,1

Tabela 2. Média do tempo(ms) das requisições de registro

Os dados obtidos das execuções da requisição de descobrimento baseado em múltiplos nomes podem ser encontrado na tabela 3. Para obter os tempos relacionados à

²Uma imagem Docker é uma especificação do que o *container* vai possuir quando for executado. Ela não pode ser executada, pois apenas armazena as instruções e dados que depois serão usadas para criar o container.

implementação original foram feitos ajustes similares aos realizados para obter os dados da tabela 2. O tempo de resposta está intrinsecamente ligado com a distribuição dos dados e como estão organizados no *cluster*. Uma taxa de distribuição alta implica em um tempo de resposta maior. Em contrapartida, a carga de processamento não fica direcionada a apenas uma instância, e sim distribuída entre todas as instâncias do *cluster*.

Número de nodos no <i>cluster</i>	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (<i>original</i>)	Tempo de resposta (ms) (<i>proposta</i>)
2	1	130,8	94,5
2	5	119,5	61,9
2	10	115,1	26,0
3	1	151,0	130,0
3	5	105,7	70,6
3	10	116,0	44,2
4	1	137,7	123,3
4	5	139,1	112,5
4	10	102,8	53,6
5	1	134,4	149,0
5	5	157,3	92,8
5	10	185,2	160,6

Tabela 3. Média do tempo(ms) das requisições de 100 nomes

A tabela 4 corresponde aos dados obtidos nas execuções sucessivas da requisição para se obter todas as entradas de registro cadastradas no Hyperbahn. O tempo de resposta pode sofrer variação dependendo de qual instância for responsável por agregar todos os registros. Isso se dá devido às chaves registradas não estarem distribuídas igualmente no *cluster*.

Número de nodos no cluster	Número de réplicas de uma entrada (k-value)	Tempo de resposta (ms) (proposta)
2	1	25,3
2	5	41,7
2	10	58,9
3	1	30,4
3	5	49,3
3	10	38,1
4	1	28,9
4	5	60,9
4	10	56,4
5	1	35,8
5	5	70,5
5	10	74,9

Tabela 4. Média do tempo(ms) de resposta para obter todas as entradas de registro cadastradas

7. Conclusão

Com o intuito de aprimorar os mecanismos de descoberta de serviço disponibilizadas pelas ferramentas estudadas, esse trabalho utilizou a web semântica como meio para atingir seus resultados. A ferramenta Hyperbahn foi a escolhida para ser modificada. Originalmente uma solução que possibilitava apenas a atribuição de um nome para o endereço de uma máquina na rede, foi modificada ao ponto de também proporcionar o mapeamento de múltiplos nomes a um mesmo *host*.

Em meio a um ambiente altamente heterogêneo, volátil e muitas vezes caótico, que é a realidade no mundo de desenvolvimento com *microservices*, se dá pouca importância a uma descoberta de serviço que descreve semanticamente as funcionalidades dos *microservices*, o que resulta, na maioria dos casos, em acoplamento entre consumidores e provedores de serviços. Com relação a esse problema, o presente trabalho procurou um meio de atribuir significado ao método de descoberta de serviço nas ferramentas estudadas, estabelecendo com que os nomes utilizados no registro fossem substituídos por nomes com significado. Foram escolhidas URIs RDF para isso, que podem representar conceitos ou relações entre conceitos de uma ontologia, com o fim de atribuir sentido aos dados retornados do processo de descoberta. Tais mudanças, proporcionam um ambiente que reforça ainda mais as características boas resultantes do uso da arquitetura de *microservices*, como a independência, resiliência e escalabilidade.

Como podem ser visualizadas no experimentos realizados, as vantagens oferecidas pela ferramenta de descoberta escolhida, o *Hyperbahn*, tentaram ser aproveitadas ao máximo, como a clusterização e a fragmentação do dados entre as instâncias pertencentes ao *cluster*. Também foi feito proveito da estrutura provida pelo ambiente utilizado para

execução dos experimentos, a implantação dos *microservices* utilizando *Linux containers*, que vêm sendo extensamente utilizados em ambientes de produção na indústria, facilitou em muito a troca de ambiente entre cenários de teste.

Referências

- Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):28–37.
- Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22.
- Filho, O. F. F. (2009). Serviços semânticos: Uma abordagem restful. Master's thesis, Universidade de São Paulo.
- Fowler, M. and Lewis, J. (2014). *Microservices*.
- Newman, S. (2015). *Building Microservices*. USA: O'Reilly Media, Eds. Gravenstein Highway North, Sebastopol, CA.
- Salvadori, I. L. (2015). Desenvolvimento de web apis restful semânticas baseadas em json. Master's thesis, Universidade Federal de Santa Catarina.
- Stubbs, J., Moreira, W., and Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. *7th International Workshop on Science Gateways. IEEE, jun 2015*, page 34–39.
- Technologies, U. (2015a). Ringpop docs.
- Technologies, U. (2015b). Tchannel docs.
- W3C (2013). Rdfa 1.1 primer - second edition: Rich structured data markup for web documents.