UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Levani Kokhreidze

Service Discovery from Uniform Resource Locators of Monitored Web Applications

Master's Thesis (30 EAP)

Supervisor(s): Marlon Dumas, PhD
Vladimir Šor, PhD

Tartu 2016

Service Discovery from Uniform Resource Locators of Monitored Web Applications

Abstract

This thesis addresses the problem of analyzing Uniform Resource Locators (URLs) of incoming Hypertext Transfer Protocol (HTTP) requests in a Web application server in order to discover the services provided by the applications hosted by the application server, and to group these applications according to the services they provide. The thesis investigates this problem in the context of the Plumbr Java performance monitoring tool. When the hosted applications are implemented using a known web framework (e.g. Spring), the service name and associated data, such as URL parameters, can be extracted directly from the controller. However, this controller-based service discovery approach, which is currently implemented in Plumbr, is not applicable when the hosted applications use unknown framework. This research addresses the problem in this latter more general setting.

The thesis proposes a pure URL-based approach, where the observed URLs are parsed, leading to sequences of tokens, which are then analyzed using natural language processing techniques and graph transformations. The proposed service discovery technique has been implemented in Groovy and Java, integrated into the Plumbr tool and evaluated on data extracted from production server covering over 400K URLs.

Keywords: service discovery, JVM, REST, natural language processing, text mining.

CERCS Classification: P170 - Computer science, numerical analysis, systems, control

Teenuste tuvastamine seiratavates veebirakendustes üldiste ressursilokaatorite abil

Kokkuvõte

Käesolev magistritöö käsitleb veebiserveri sissetulevate HTTP päringute URL-ide analüüsimist veebiteenuste tuvastamise eesmärgiga.

Probleem on aktuaalne veebirakenduste monitooringutööriistade seisukohalt, kuna sissetulevad HTTP päringud on vaja omavahel loogiliselt grupeerida selleks et edasi hinnata ning jälgida teenuse vasteaega.

Uurimistöö keskendub Java veebirakendustele ning analüüsib URLide andmehulka, mis on saadudud monitoorimistarkvarast Plumbr.

Kui monitooritav veebirakendus on realiseeritud mõne Plumbri jaoks tuntud veebiraamistiku abil (näiteks Spring), siis on võimalik seda rakendust instrumenteerida selliselt, et teenuse nimi on üheselt määratletav. Kui aga tegemist on Plumbri jaoks tundmatu veebiraamistikuga, siis ainuke sissetuleva päringu kirjeldus on selle päringu URL.

Kui URLis sisalduvad dünaamilised parameetrid, siis sama teenust kasutavad päringud on erinevad ja neid ei ole võimalik ainult URLi põhiselt grupeerida.

Käesolev magistritöö pakub välja URLi analüüsil baseeruva grupeerimise lahenduse. Lahendus tükeldab URLi, eraldab sealt sõnede ahelad, mida seejärel analüüsib kasutades loomuliku keele töötlemise ning graafitransformeerimise tehnikaid.

Pakutav teenuste tuvastuse lahendus on teostatud kasutades Java ja Groovy programmeerimiskeeli, hinnatud andmehulgal mis koosneb üle 400 000 URList ning on integreeritud Plumbr monitooringutarkvarasse.

Võtmesõnad: teenuste avastamine, JVM, REST, naturaalse keele protsessimine, teksti kaevandamine

CERCS Classification: P170 - Computer science, numerical analysis, systems, control

Acknowledgements

I wish to express my genuine gratitude to my supervisors, Marlon Dumas and Vladimir Šor, for helping me out with this research. I would like to thank whole amazing collective of Plumbr OÜ for proposing this research topic and sharing their expertise and knowledge with me. I would like to thank my family and friends for their support. Lastly, I want to say thank you to the whole academic staff of Software Engineering mater's program for their dedication and professionalism over the past two years; it was an incredible journey.

List of Abbreviations

Table below describes the meaning of the various abbreviations and acronyms used throughout the thesis.

Abbreviation	Meaning
API	Application Programming Interface
DSL	Domain Specific Language
EJB	Enterprise Java Beans
HTTP	HyperText Transfer Protocol
JVM	Java Virtual Machine
MVC	Model–view–controller architecture
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote procedure call
SQL	Structured Query Language
SVG	Scalable Vector Graphics
UML	The Unified Modeling Language
URI	Uniform Resource Identifier
URL	Unified Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Service Definition Language

1. Introduction

Service Discovery is a relatively new area of research and in the context of Java-based web applications. We can think about it as a mechanism or methodology to find application endpoints, which provide direct value to the end user. With increased popularity of REST over the past decade, it is no surprise that Service Discovery is closely related to REST architecture and in the most of the cases, those endpoints are represented as RESTful web services. On the other hand, services can be RMI-related APIs, which is the case for the EJB development architecture. In general, the structure of the service is implementation specific, but in most of the cases, while adopting specific architectural style or framework, certain rules will always be followed with respect to service design.

1.1 Problem statement

As of today, Plumbr supports a limited number of JVM-based frameworks to discover services automatically. In those cases, it instruments framework specific classes to extract class with the corresponding method as the service name. For example:

Given URL: http://www.example.com/invoices/411121/pay

Implemented in: *Spring MVC*

Will yield service name as: *InvoiceController.pay()*

Since there are a vast number of web frameworks in the JVM ecosystem, Plumbr cannot implement support for each of them separately. When Plumbr is used to monitor web application implemented in the unsupported framework, raw URL value is stored as a service name.

At the moment, Plumbr fails to identify parameters or dynamic parts in raw URL. Thus, same service with different values is registered multiple times. This results in poor user experience, increased database size and various other problems related to Plumbr automatic root cause detection.

Existing research in the area of Service Discovery is mainly concentrating on identifying services from a different kind of documentation such as: UML class diagrams, API documentation or WSDL documents. Therefore most algorithms and tools that were developed for automatic service discovery rely on the idea that at least one of the documentation mentioned above would be provided as input. For this day, we have a situation that there is no way to identify meaningful services from raw URL data.

1.2 Contribution

The thesis aims are:

- Study existing methods of discovering services described in the literature.

- Identify applicability of natural language processing techniques on Plumbr dataset, consisting of approximately 400K captured raw URLs from 2 Java web-applications
- Implement Java based application that will analyze URLs and give tree-like graph representation of the URLs as an output.
- Analyze application output and describe the results.

2. Background

The purpose of this chapter is to give a brief overview of the URL structure and to summarize existing knowledge in the context of Service Discovery. Currently, service discovery is mainly used in the situations where legacy style APIs need to be migrated to modern RESTful style web services. Most of the existing research discussed in section 2.3 focuses on analyzing RPC interface documents to identify important services and transform them into REST [11] style web interfaces. Tools and algorithms developed for discovery require different types of application documentation as input. Research showed two most popular documentation choices:

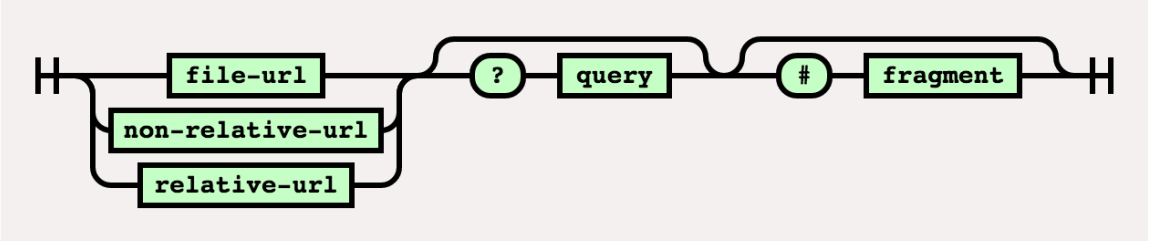
- UML class/interface diagrams
- API documentation (WSDL documents, interface description, etc.)

Research presented in this master's thesis focuses on possibility to provide raw URL data as an input and give normalized service names as an output. Static placeholders will replace dynamic parts in the URL, so that same URL with different parameters or path variables will be registered under the same service name.

In the following chapter, we will discuss in more detail what are current approaches and challenges in this field of research and how can existing techniques and algorithms be used to identify and normalize important services in the URL dataset.

2.1 Introduction to URL

This section introduces the concept of URL, syntax and associated terminology required to proceed with the research presented herein. In general, URLs [9] consist of multiple components, such as: scheme, scheme-data, username, password, host, port, path, query and fragment. Value for each component is either present or not depending on the used URL type.



Railroad diagram that illustrated basic structure of the URL. Taken from the resource [8].

As far as this master's thesis is concerned we won't cover file-url and non-relative-url in details, but rather provide examples to have the basic understanding of their structure.

file-url – consists of file:// prefix and the absolute path of the file in the file system. It can be used to locate files from the local computer or the host filesystem. For example: file:/var/logs/application.log can retrieve log file content associated with the path on the local filesystem, where the file is the schema and /var/logs/application.log is the path. Same result can be achieved by specifying host and retrieving content from remote filesystem:

file://example.com/var/logs/application.log – Where example.com is the host.

non-relative-url – consists of schema prefix and schema-data. It evaluates schema-data for the defined schema. For example, we can understand following non-relative URL: javascript:alert("Hello, world!") as the URL with the schema value is javascript and schema-data – alert("Hello, world!"). Schema-data is specific for non-relative URLs, and its value for the file-url and relative-url is set as NULL.

In this research we will be mainly dealing with the relative-url. Thus, it is important to understand its structure in more details. As we can see in figure 1, there are different ways of constructing relative-url and each way has its own components.

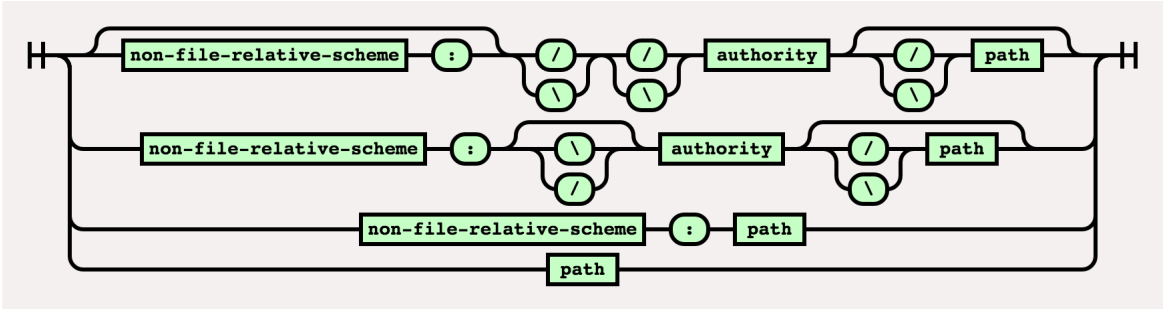


Figure 1: Railroad diagram for the relative URL. Taken from the resource [8].

- **Non-file-relative-schema** – schema with one of the following values: *ftp, gopher, https, http, wss or ws*.
- **Authority** – is constructed using the following components: user, password, host and port. In case if we specify **authority** component – only *host* component is mandatory. Figure 2 gives syntax overview of the **authority** components.
- **Path** – path to the resource.

URL-based services registered by Plumbr are stored with no schema nor authority blocks, thus in this research we will be focusing on the **relative-url** type, where only **path** component is presented.

URL-path consists of different parts separated by the path delimiter. According to the RFC-1738 [9] URL-path delimiter does not have any predefined structure and its design is implementation/schema specific.

Hereafter we will refer to each part of the URL-path tokenized by the path delimiter as **URL part**.

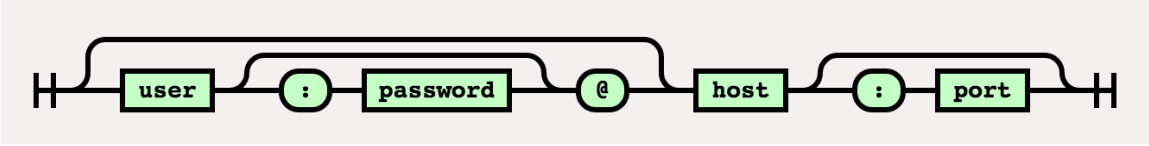


Figure 2: Railroad diagram for the **authority** block presented in the figure 1. Taken from the resource [8].

2.2 Introduction to HTTP and REST

The aim of this section is to give a general overview of the REST architecture and HTTP protocol. Both of them are quite broad topics. Therefore, we will cover only those aspects that are important for this master's thesis.

According to the RFC-2616 [10] documentation, HTTP (The Hypertext Transfer Protocol) is the foundation for the data transmission across the world wide web. It is implemented as a request-response protocol between a client and a server, where in the modern world a client is usually a browser and a server – a machine hosting the corresponding web application. HTTP provides Request Methods (HTTP verbs) to express the intention on the particular resource. Most common HTTP verbs include: GET, POST, PUT and DELETE.

REST is the architectural paradigm proposed by Roy Fielding [11]. In short terms, we can think about the REST as the way of describing resources (services), their current state and available actions using standard HTTP components. RESTful web services consists of 3 main building blocks:

Hypermedia – Links that identify current application state and available actions.

- HTTP – Interaction protocol, with corresponding HTTP verb and HTTP response.
- URI – Resource identifier.

As we already mentioned, HTTP request methods are used to define the intention on the resource. Regarding RESTful web services, it is utilized for the very same reason. For instance, when someone wants to query the resource, HTTP verb GET is used, for creation – POST, modification – PUT and so on. The server will respond with corresponding HTTP status code: for the successful operations, the status code is from 200 till 300 and for the unsuccessful ones – 400-500.

In the upcoming chapters, we will see how different components of the REST architecture can be used to generate RESTful web services from RPC interface documentation as well as discover important services from the raw URL data.

2.3 State of the Art

Finding relevant literature for selected topic is one of the most important parts of any research. Related reference list was selected using principles of systematic literature review presented by Barbara Kitchenham [1]. Searching for relevant literature was split in two phases. At first we need to find source of scientific papers, secondly we have to find bibliography associated with Service Discovery.

Google Scholar database was selected to find related articles. Considering the fact that Service Discovery is more abstract term and in the context of this research it is strongly related to terms such as: reverse engineering, REST and API primary search was conducted using following phrases:

- "Service Discovery" OR "Reverse Engineering" AND "REST" OR "Software reengineering" and "REST"

Above-mentioned query produced approximately 14,500 results. After initial overview more detailed analysis was performed on result set using following criteria definitions:

- Is the paper about "Service Discovery" or "REST" and "Reverse Engineering" or "REST" and "Software reengineering"?
- Does the paper include real life examples?
- Does the paper have algorithm or pseudocode examples?

In case the paper satisfied above-mentioned criteria, it would be considered as relevant to this research. Based on existing methodology potential answers could be: "Yes" or "No"

Before trying to implement the application for Plumbr case study, it was important to understand what is the most common trend in terms of web services. For this purpose research paper [2] by M. Maleshkova, C. Pedrinaci, J. Domingue was selected as an example. Research presented in this paper was conducted in year of 2010 analyzing approximately 222 Web APIs from the ProgrammableWeb[1] directory. Based on the paper mentioned above we can conclude that RPC and REST style Web APIs were most popular ones. Study also suggested that apart from REST and RPC type web services, there could in fact be combination of these two, called hybrid style services.

Information extracted from above mentioned paper was useful in a sense that during implementation of natural language processing algorithms, it was clear that those 3 types of Web APIs would be most popular ones and first of all algorithm should be able to handle those cases. As research presented herein concentrates on Java technologies, in JVM world analogue of RPC style Web APIs would be services based on Remote Method Invocation (RMI).

After analyzing most popular types of web services logical step forward would be to investigate what is current approach to actually discover services in those types of applications. For this purpose various papers were selected and below are conclusions based on them.

Research papers [3] and [6] aim to accomplish same goal but with different approaches. The idea behind those studies is to generate RESTful style web services from legacy style APIs.

In the research presented in paper [6] authors choose to adapt existing, legacy services as a RESTful interfaces by adding additional application layer, which is implemented in application specific DSL. New application layer is basically a code generator that can map RESTful web services to existing legacy API.

[1] http://www.programmableweb.com/

In paper [3] authors focus on analyzing software documentation to get RESTful web services. In more details purpose of research paper [3] is to design a specific tool that can generate RESTful style web services from UML class/interface diagrams. Paper suggested that, given the structure on how modern frameworks are organized, URLs should be presented in a tree like hierarchy structure as it gives possibility to configure and implement the skeleton of a Web application. As far as this paper is concerned taking into account research [3] final output of the algorithm that we will discuss in following chapters will be acyclic connected graph that will expose hierarchy of URLs giving possibility to its consumer to observe application resource architecture.

Research presented in article [5] also focused their research on two dominant types of web API documentation: REST and RPC-style.

RPC-style Web APIs

Algorithm designed for information extraction from the RPC documentation in paper [5] relies on two main characteristics:

- Operation names follows CamelCase[2] notation, where first part is verb and remaining part is typically a noun.
- Documentation in most of the cases follows some patterns. For example: operation name in most of the cases is in <h3> tag, description in <p> and so on.

Figure 3 shows example of RPC-style web API documentation.

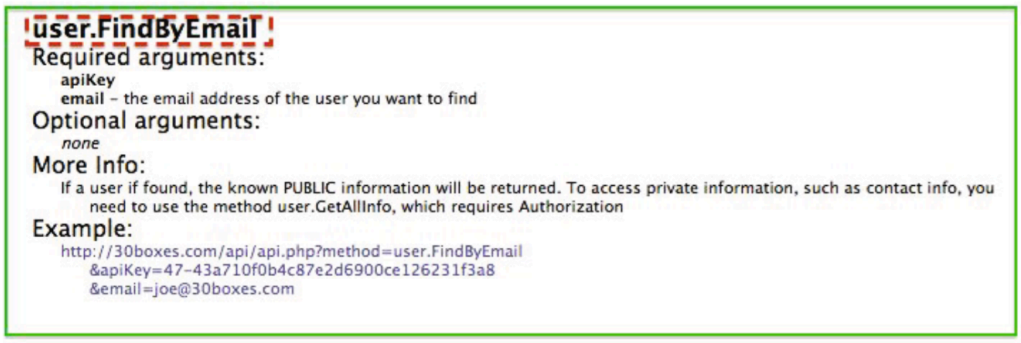


Figure 3: RPC-style web API documentation example. Taken from the paper [5]

Given those characteristics paper presented extraction algorithm to automatically discover technical information (method names, parameters, types, etc.) for RPC-style services.

RESTful Web APIs

Opposed to the previous case, naming conventions here are almost useless because usually RESTful services do not necessarily follow any predefined naming conventions. Instead algorithm presented in the paper uses HTTP methods and URI templates to represent resource model of the application.

[2] https://en.wikipedia.org/wiki/CamelCase

Figure 4 shows example of RESTful style web API documentation

URL	http://del.icio.us/api/[username]/bookmarks/[hash]
Method	GET
Returns	200 OK & XML (delicious/bookmark+xml)
	401 Unauthorized
	404 Not Found

Figure 4: RESTful web API documentation example. Taken from the paper [5]

Although article [4] is mainly focused around RPC style service discovery, main ideas still can be used for URL-based services. The goal of the research is to discover services from WSDL documentation and build hierarchy between them. This is achieved by performing multi-step process, which takes WSDL document as input and produces a potential resource models. Those resource models, combined with HTTP verbs can be interpolated as RESTful service interaction points.

Figure 5 illustrates high-level overview of resource extraction process presented in the paper.

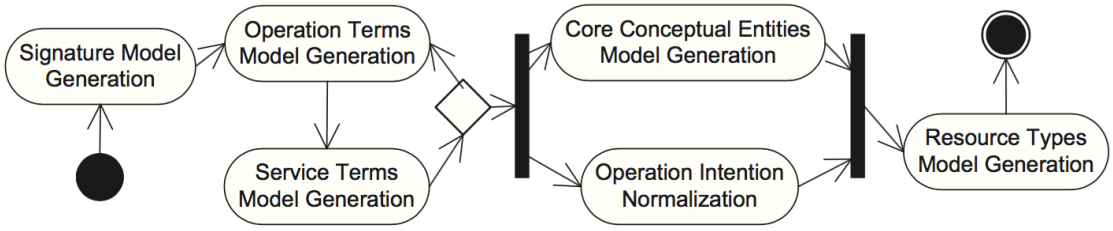


Figure 5: High-level overview of resource extraction process. Taken from the paper [4]

Signature Model Generation

The aim of Signature Models is to present service description in more normalized way. We can think about it as a document structure designed particularly for this paper. This kind of abstraction helps to avoid any specific syntax or structure details presented in original interface description document. Although in the research, authors used WSDL as input, thanks to this abstraction any machine-readable interface description document can be served as an input.

Generated signature model s_i has following properties:

$s_i.name$ – Operation name

$s_i.input$ – A set of input parameters

$s_i.output$ – A set of output parameters

Each parameter has its own characteristics. For example parameter p_i has following attributes:

$p_i.name$ – Parameter name

$p_i.type$ – Parameter type

$p_i.multiplicity$ – Parameter multiplicity

$p_i.class$ – Parameter tag designed to distinguish between *application data* and *metadata*

Let's look at $p_i.class$ tag in more details. First of all we need to understand what's the reason for introducing this parameter tag at all. In the most cases API consumers require to provide some authentication token as parameter. Those kinds of parameters aren't directly used to deliver functionality provided by the operation, as a result they are as *metadata* parameters. On the other hand parameters like *orderId* are tagged as *application data* since they are used directly by the service to extract necessary information. For this purpose TF-IDF categorization algorithm can be used, where parameters play the role of terms and operation name with corresponding signature play role of documents. Categorization score discussed in paper [5] is computed as follows:

$$C_{pf-isf}(p, s_i, S) = pf(p, s_i) * isf(p, s_i, S)$$

S – Number of all operation signatures

s_i – Each operation signature

p – Each parameter given in operation signature s_i

$pf(p, s_i)$ is defined as follows:

2 – if (p $\in s_i.input \cup s_i.output$) $\wedge$ (substring(p.$name$, $s_i.name$))

1 – if (p $\in s_i.input \cup s_i.output$) $\wedge$ (¬substring(p.$name$, $s_i.name$))

0 – otherwise

$isf(p, s_i, S)$ is computed by:

$$log_2 \frac{|S|}{|\{S_i \in S : pf(p, s_i) > 0\}|}$$

Using predefined threshold T parameter can be tagged using following formula:

$if\left(C_{pf-isf}(p, s_i, S) > T\right)$ *then* $\boldsymbol{application\ data}$

$if\left(C_{pf-isf}(p, s_i, S) \leqq T\right)$ *then* $\boldsymbol{metadata}$

In the upcoming chapters we will discuss in details how this formula can be modified to discover parameter parts in the URL dataset.

Operation Terms Model and Service Terms Models

In general service names incorporate valuable information, therefore natural language processing techniques can be used to extract service description directly from service names. Often service name follow CamelCase semantics, thus simple tokenization algorithm can be used to discover words in service name. For example:

tokenize(addOrderItem) -> [add, order, item]

Where *tokenize* function is simple string parsing algorithm, which splits input on capital letters. After extracting each word from the service name operation terms can be generated using natural language processing techniques. Each word is marked with one of the following term: Intent, Concept, Qualifier or Selector.

Intent – In most of the cases *Intent* is the verbal part of the service name. Describes the intention of the action. For example: **get**OrderItem, **pay**Invoice, **add**Purchase

Concept – We can see it *Concept* is the element on which operation is performed. For example: getSubmitted**Order**

Qualifier – In most cases we will see *Qualifier* as adjective, which describe semantic qualities of the *Concept*

Selector – Filtering, selecting, etc. For example: getSubmittedOrders**ByDate**

After successfully generating operation terms for each operation name, dependency graph can be constructed that will basically give us high-level overview of services that modifies the data or queries the data.

Reducing the graph with the least outgoing or incoming paths will lead us to Service Terms Models generation.

Core Conceptual Entities Extraction and Operation Intention Normalization

Core Conceptual Entities Extraction and Operation Intention Normalization are two last steps before getting final output of the process. Input for this step is graph data structure with nodes labeled accordingly (Intent, Concept, Qualifier, Selector). Each node has a property containing all other nodes, which it affects. Core Conceptual Entities Extraction takes nodes, which are labeled as Concept, and gives possible version of the Resource Model. Nodes, which have no incoming or outgoing paths, are omitted and the reason for it is that if *Concept* is not modified nor queried it cannot be used for generating REST services.

Operation intention normalization tries to determine what is the purpose of invoking corresponding operation. Below is the list of possible categories:

- Constructor – corresponds to HTTP verb POST (create, add, etc.)
- Destructor – corresponds to HTTP verb DELETE (delete, remove, etc.)
- Accessor – corresponds to HTTP verb GET (get, fetch, etc.)
- Mutator – corresponds to HTTP verb UPDATE (update, modify, etc.)
- Query – corresponds to HTTP verb GET (query with some additional parameters)
- Investigator – corresponds to HTTP verb GET (find, collect, etc.)
- Process – when service name is not categorized in any above mentioned categories

Categorization is achieved by analyzing graph names and their incoming and outgoing paths. For example when node is categorized as *Constructor* usually node name is verb (add, create, etc.) and its outgoing paths – child nodes, generally are represented as *Intent.*

Research paper [7] describes thirteen different approaches for service discovery. Most of the techniques were not applicable for this research as they were mainly addressing more specific issues like indexing, networks service discovery, etc. There was one exception though, described as *Keyword Clustering*. Using this technique similarity matrix was calculated between different URLs using Pareto principle [12 1 Data structure

]. In the following chapters I will describe how approach similar to this can be used populate reduced number of URL-based services after discovering parameter and dynamic parts in the URL.

3. Contribution

In this chapter, we will discuss how existing knowledge can be modified for URL-based Service Discovery. Also we will review in detail development process of software prototype, which will implement the modified knowledge of existing Service Discovery techniques using Groovy and Java programming languages.

3.1 Data structure

As it was already stated, for this master thesis data was obtained from Plumbr's database. For the sake of confidentiality we will skip exploring the internal data structure of Plumbr. Instead, we will describe what kind of structure is required to perform the analysis. Since the URL format and style is application specific, first and foremost, we need some kind of an identifier to perform research separately for each application. For this purpose, we will introduce a separate column for each URL entry, called '*appId*'. This will give us possibility to perform bulk data analysis, for multiple applications or accounts at the same time. Figure 6 illustrates a data structure example, which will be served as an input.

appId	serviceUrl
appId023230123123	/api/rest/invoices
appId023230123123	/api/rest/invoices/2323
appId023230123123	/api/rest/pos
appId023230123123	/api/rest/pos/123232
appId023230123123	/api/rest/invoices/pay
appId46464523461t6	/api/v1/tasks
appId46464523461t6	/api/v1/tasks/23554
appId46464523461t6	/api/v1/tasks/12345/contributors

Figure 6: Example data structure of CSV file that can be served as an input to the application.

3.2 Modifying existing knowledge for URL-based service discovery

In chapter 2 we discussed and summarized existing approaches related to Service Discovery and as we found out the goal of the most research papers was to replace legacy RPC style APIs with modern RESTful architecture. Since majority of designed algorithms and tools require RPC-style interface documents or UML class/interface diagrams as an input, we need to perform some modifications in order to adapt existing knowledge to our needs. To do that, first and foremost we need to understand from existing theories what can be used for URL-based discovery and what kind of adjustments do they need. In the upcoming sections we will try to draw parallels and identify common characteristics between RPC and URL based services.

3.2.1 Initial dataset review

Purpose of this section is to analyze what we need to do on dataset in the first place. As we already mentioned, that data represents raw URLs captured at the JVM boundaries, therefore potentially it contains a lot of noise. At this point size of dataset does not play important role. For now we are mainly interested in the URL structure and potential noise candidates. We will talk about dataset characteristics in more details during chapter 4.

For research purposes following MySql[3] queries were developed, in order to understand how to filter out nonessential URLs.

```
SELECT count(t.*)
FROM `table_*****` t
```

[3] https://en.wikipedia.org/wiki/MySQL

```
WHERE t.rawUrl LIKE '%.html'
      AND t.appId='appId********'

SELECT count(t.*)
FROM `table_*****` t
WHERE t.rawUrl LIKE '%.js'
      AND t.appId='appId********'

SELECT count(t.*)
FROM `table_*****` t
WHERE t.rawUrl LIKE '%.css'
      AND t.appId='appId********'
```

Different Plumbr accounts were selected using above mentioned queries and in all of the cases they returned considerable amount of results, thus URLs containing commonly used static contents such as: '.html', '$', '.php', '.js', '.txt', '.css', '.jpg', '.ico', '.gif' will be excluded in order to reduce noise in the dataset. We can define them as the set of rules for noise reduction.

Based on the initial research results, data parsing algorithm can be developed based on above-mentioned rules. Noise reduction is not application specific, thus it can be done as a very first step, before actual URL-based service discovery.

3.2.2 Data processing

Most of the existing discovery approaches perform their groundwork based on software documentation. On the contrary, URLs do not have any predefined style guideline and fairly in all of the cases they are implementation specific; therefore we cannot have defined or strict understanding about the structure. Furthermore, initial review of the Plumbr dataset showed that it contains not only URLs, but also RMI style services extracted from the frameworks supported by Plumbr. To address all this issues, we need to transform list of raw URL data into data structure eligible for analysis. In order to achieve this, five consecutive steps will be developed. Those steps are:

1. Initialization phase
    a. Constructs required data structures for succeeding steps.
    b. Removed domain from the URL if applicable.
    c. Discovers URL part delimiter per application and saves it in the corresponding data structure.
    d. Performs additional data maintenance based on the delimiter.
2. Grouping phase
    a. Splits URLs using discovered delimiters per application.
    b. Builds application URL groups using application identifier and tokenized URLs.
3. Analysis phase
    a. Performs main analysis in order to identify dynamic and static parts in URLs using TF-IDF and MapReduce algorithm.
4. URL reduction phase
    a. Joins similar URLs as one and assigns new service name.
    b. Alters original URL by replacing identified dynamic parts with predefined

static string.

5. Graph building
    a. Builds graph representation from reduced URLs.
    b. Joins similar nodes, if applicable.

In the upcoming sections we will discuss each step in more details and will try to find a correlation between URL and RPC/UML based service discovery.

Initialization phase

The idea behind initialization step is to build groundwork for forthcoming URL-based service discovery. This includes building initial data structure, containing original URLs with application and URL identifiers. Each of the populated data structures during initialization phase will be used during different steps of the discovery lifecycle. For example during grouping phase application identifier will be used to group tokenized services per application separately. This will allow us to distinguish URL style among different software systems; therefore we can simultaneously analyze multiple Plumbr accounts at once. On the other hand during analysis step we should have a possibility to mutate original services, meaning that if we will successfully identify dynamic parts in the URL we should have a fast mechanism to find one specific URL and change dynamic part by predefined static content. For this case we introduced URL identifier, which will be used to quickly find specific record in the dataset. This will allow us to group similar services as one and generate new service name for group of URLs during reduction phase.

Java code snippet presented below represents basic data structure example constructed during initialization step.

```
class RawUrlData {
    public String appId
    public int urlId
    public String rawUrl
}
```

We already discussed procedure and regulations on how to reduce noise in the dataset by excluding URLs containing commonly used static content. As initial dataset overview showed beside such pollution, approximately 1% of all services take parameters. SQL query presented below extracts such cases:

```
SELECT t.rawUrl
FROM `table_*****` t
WHERE (t.rawUrl LIKE '%&%' OR t. rawUrl LIKE '%?%')
```

Even though it is not significant amount of URLs, without proper string parsing, most likely those kinds of records will introduce problems during service reduction phase. In order to have proper service grouping we need remove parameters from the URL but

keep the actual URL intact. For example suppose we have implemented function with the following signature:

```
String parse(String url)
```

If we will have a following URL:

`https://example.com/service/123/invoke?start=2016-04-17T00:00&end=2016-04-18T23:59`

Then the result should be:

`https://example.com/service/123/invoke`

Although requirement is rather straight forward, before implementing this functionality we need to keep in mind one essential detail, which is that we have no any guarantee that URL delimiter will be same for every software system.

For example, as we found out during literature review, in the research paper [2], '/' is the most commonly used across the RESTful web services, so lets examine query given below:

```
SELECT count(t.*)
FROM `table_*****` t
WHERE SUBSTR(t.rawUrl, 2) NOT LIKE '%/%'
```

This query returns total count of records where 'rawUrl' column does not contain slash. It excludes first character; because in case of Plumbr URL-based service discovery first character is always slash.

Surprisingly we discovered approximately 30% of all URL-based services, which spread across 3 different JVM based applications fall into above-mentioned category.

So before rushing into parameter filtering, above all we need to find out what is the URL delimiter for a given application. To do so we can introduce a delimiter analyzer algorithm that will search for known delimiters in the application URLs. For the sake of brevity of this master's thesis we introduced three known URL delimiters, which are: slash ('/'), dot ('.') and minus ('-'). Figure 7 gives brief overview of the algorithm.

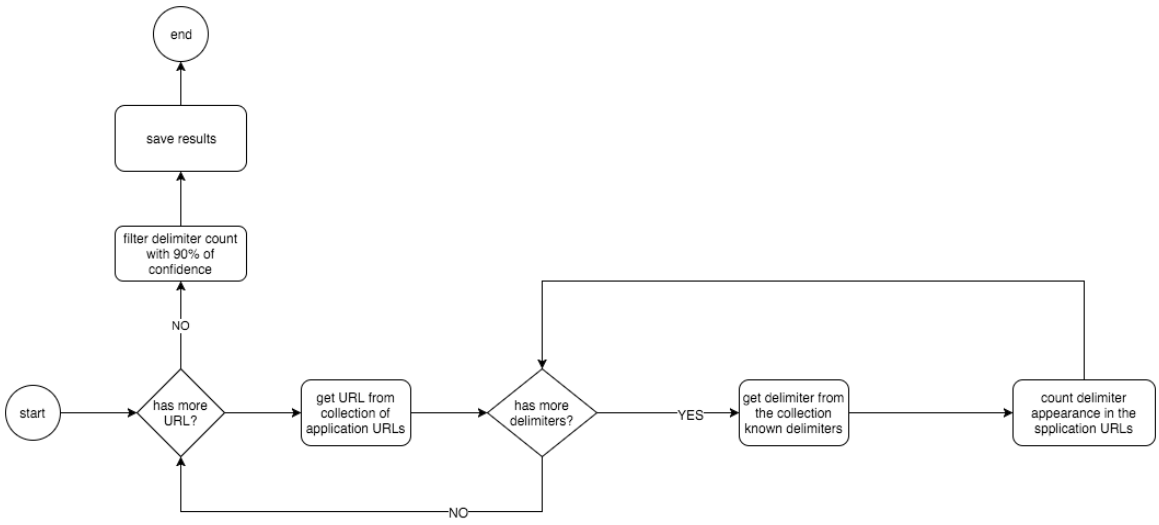


Figure 7: Delimiter analyzer algorithm flowchart diagram

The core concept of delimiter analyzer is to count occurrence of each expected delimiters in the collection of application URLs and select the one with 90% confidence. In the upcoming application development chapter we will discuss in more details how this algorithm can be implemented with functional programming using Groovy-programming language. Since application specific delimiter will be used multiple times from different phases of URL service discovery lifecycle it would make sense to save it into memory with the following map structure:

Key: *appId*

Value: *identified delimiter*

Upon successful identification of the delimiters for individual applications, we can perform parameter trimming in the URLs. One may ask why we need URL delimiter for parameter removal at all. We will see precise answer to this question in the upcoming chapter regarding Tool development and Plumbr case study; there we will be able to see difference between software systems with respect to URL design. As for now we can describe basic rules of parameter trimming.

According to World Wide Web URL documentation [14] – traditionally, URL parameter indicator is question mark symbol (?) and in case of multiple parameters followed by '&' symbol. Based on this statement the most logical way would be to remove everything after '?' symbol, but again this w3 documentation does not guarantee that every application in the world follows same style guideline. For example it is usual use case around the web that some of the web sites include page title in the URLs. This is common case for news, hotels or flights related web sites. So basically there can be the case when URL contains & and it is not actually a parameter indicator. And it is not mandatory to include '?' in case of parameter usage. All things considered, we can say that URL presented below can be considered as valid:

https://www.example.com/sport-&-entertainment/h1di39dg/details&newsId=1234&from=google.com

As we can see URL presented above does not contain '?' but instead parameters start straight with '&' symbol. Considering this fact if we will implement parameter trimming algorithm based on '?' symbol, it would fail since there is no '?' at all and in case of '&' symbol we would get incorrect result since first appearance of '&' does not indicate parameter. All things considered and based on multiple research parameters in the URL should be removed based on this rules given that we have two parameter indicators ('?', '&') :

1. We must have URL delimiter.
2. If URL contains more than 1 part, parameters must be in the last part.
3. Parameter indicator can be identified as last occurrence in the given part.

Based on this rules let's have step by step explanation for the URL presented above:

1. During initialization phase domain name is removed as a result we have relative URL:

   /sport-&-entertainment/h1di39dg/details&newsId=1234&from=google.com

2. Delimiter analyzer identified slash as URL delimiter.
3. URL tokenization is performed giving the following results:
   [*sport-&-entertainment*,
   h1di39dg,
   details&newsId=1234&from=google.com]
4. Based on rules 2 and 3 we will search for parameter indicators ('?' or '&') only in the last part.
5. In the last part of the URL we will remove everything starting from the last index of parameter indicator:

   details&newsId=1234&from=google.com ≫ *details*

   Code snippet given below illustrates how the last step can be achieved

```
String url = 'details&newsId=1234&from=google.com'
int i = url.lastIndexOf('&')
if (i != -1) {
    return url.substring(0, i)
}
```

After all this steps new URL can be constructed with removed parameters using Groovy join function:

```
['sport-&-entertainment', 'h1di39dg', 'details'].join('/')
```

Parameter removal from the original URLs is important step, because it will allow us to avoid pollution related to the similar service grouping.

Grouping phase

The objective of grouping phase is to tokenize URLs using discovered URL delimiter during initialization phase and group them using identified URL style. For instance URLs, which start with '***api***' will be gathered in the same group. This approach was introduced in order to split original dataset into smaller chunks. We will discuss TF-IDF [13] algorithm, which will be used for URL parts classification (dynamic, static) in more details shortly, but at this point we need to mention that algorithm is based on the assumption that it will have multiple documents as an input and actual number of documents is used to calculate the final score. In our case we have only one dataset per application, thus we need to have a mechanism to build different groups out of it and present them as separate documents. Chosen approach has its drawbacks, which we will discuss, in the section 4.3. In the very same section we will also present possible solution for it.

In the research [4] conducted by M. Athanasopoulos and K. Kontogiannis we saw service name tokenization using upper case as delimiter. The article also suggested that for the future work more intelligent ways of tokenization could be achieved, meaning that it shouldn't be dependent on specific delimiter only. In this master's thesis we achieved this goal for URL-based services by introducing delimiter analyzer algorithm. Delimiter analyzer gives us possibility to have more generic way of discovering URL-based services; meaning that its scope isn't restricted by one specific ground rule. Tokenized URLs can be saved in the following map structure:

Key: *appId;URLPart*

Value: Rest of *urlParts as an array of Strings*

Analysis phase

Analysis phase can be described as core step for discovery lifecycle. The main goal of the previous two steps was to prepare and populate data structure for this step to successfully perform analysis. Forthcoming URL reduction and graph building phases are totally dependent on the results returned by this step. URL-based service analysis implements MapReduce and TF-IDF, two widely used information retrieval algorithms.

Usage of TF-IDF was discussed in the research paper [4] as the technique to identify relevant parameters for specific service. In the context of our research we will use TF-IDF as procedure to identify dynamic parts in the URL. For instance suppose we have list of following relative URLs:

```
/api/v2/invoice/inv123
/api/v2/invoice/inv424
/api/v2/invoice/inv948
/api/v2/invoice/inv000
/api/v2/invoice/inv782
/api/v2/invoice/inv545
```

Given this data, developer can easily spot that `api/v2/invoice` is the static part in the URL followed by invoice identifier as a result this 6 URLs can be combined as:

`/api/v2/invoice/PARAM`

TF-IDF gives us opportunity to accomplish this goal by computing relevance score, which can tell us how important specific part is for the given URL in the context of a whole application. In the upcoming section we will explore TF-IDF in more details.

TF-IDF

TF-IDF [13] is combination of two statistical techniques, TF – Term Frequency and IDF – Inverse Document Frequency. Main benefit of TF-IDF score is that its value increases with the corresponding number of times a word appears in the document, but is offset by the occurrence of the word in the collection of documents, which helps to confirm the fact that some words in our case URL parts appear more frequently. There are numerous variations for TF and IDF, with a different score calculation techniques. Figure 8 and 9 illustrates several adaptations for TF and IDF score calculation, respectively.

Name	Value
Binary	(0, 1)
Raw Frequency	$f_{t,d}$
Log Normalization	$1 + \log(f_{t,d})$
Double Normalization by 0.5	$0.5 + 0.5 \times \frac{f_{t,d}}{\max\{t' \in d\} f_{t',d}}$
Double Normalization by K	$K + (1 - K)\frac{f_{t,d}}{\max\{t' \in d\} f_{t',d}}$

Figure 8: Different versions of Term Frequency calculation

Name	Value
Unari	1
Inverse Document Frequency	$\log\frac{\mathrm{N}}{n_t}$
Inverse Document Frequency Smooth	$\log(1 + \frac{\mathrm{N}}{n_t})$

Inverse Document Frequency Max	$\log(1 + \frac{\max\{t' \in d\} n_t}{n_t})$
Probabilistic Inverse Document Frequency	$\log \frac{\mathrm{N} - n_t}{n_t}$

Figure 9: Different versions of Term Frequency calculation

Exploring each of the variations of TF-IDF is outside the scope of this master's thesis, but interestingly enough during the literature review in the research [4] we have already seen combination of Binary version of the Term Frequency and Inverse Document Frequency. Authors were interested to categorize between application data and metadata thus their choice of Binary version of the Term Frequency seems obvious. Up to this point we mentioned several times that in terms of URL-based services we want to categorize URL parts as static or dynamic, therefore one may ask why we cannot use Binary form of the Term Frequency? Reason is that for the research [4] authors had clear structure of services, meaning that they could easily distinguish between service name and parameter, part of their research question was that how relevant was given parameter for whole dataset. During the literature review in chapter 2, we have already seen how they assigned 0 or 1 depending on simple string parsing algorithm. As far as this master's thesis is concerned we should treat URLs as text document, without any predefined structure. For better illustration let's discuss text-mining case for books. Suppose we want to search find list of most important words across 10 books, each of them with different sizes. For this case we want to exclude commonly used words such as: "and", "or", "with", etc. We can accomplish this task by computing Term Frequency using Raw Frequency formula along with Inverse Document Frequency. The only thing that we need is delimiter for words and as far as we already know in the books it is usually space. If computed score is close to 0, this means that given word is rarely seen among the books, meaning that it can be marked as specific for certain book. On the other hand high score value illustrates that selected word is observed across the most of the books. Usually words like: "and", "or", "with" have high TF-IDF score.

If we will try to draw parallels between URLs and books we can think about collection of URLs as separate book and given the fact that we already know delimiter from initialization phase we can compute TF-IDF score for each part of the URL in a given collection, this will give us score value and based on this value we can categorize URL part as dynamic or static. Usually threshold for classifying word as relevant or non-relevant is predefined. Through the years multiple studies we conducted to dynamically calculate threshold based on dataset. As the matter of fact research paper [4] used threshold 0.2 for classification and according to their studies it has shown really good results. For our research we will also use predefined threshold, with an exception that tool that we develop will have a possibility to change its value during runtime based on user's needs. We will discuss this in more details in the tool development section.

TF-IDF Using MapReduce

Through the years MapReduce programming model justified its efficiency for text mining, information retrieval and various other large data processing tasks. The model was influenced by map and fold functions commonly used almost in any functional programming language. MapReduce programming model is based on key value pairs, where map generates values for a specific key and values for the same keys goes to the same reducer. But before going into MapReduce implementation details, we need to take a look at how map and fold functions look like in functional programming.

Map applies given function to each element of the list and gives modified list as an output.

```
map f lst: (’a -> ’b) -> (’a list) -> (’b list)
```

Fold applies given function to one element of the list and the accumulator (initial value of accumulator must be set in advance). After that result is stored in the accumulator and this is repeated for every element in the list.

```
fold f acc lst: (acc -> ’a -> acc) -> acc -> (’a list) -> acc
```

Lets take a look simple sum of squres example using map and fold functions. As we already described Map applies given function to every element of the list and gives us modified list. For the task stated above we have list of integers and we want to have each number in power of two. So our map function would look like this:

```
(map (lambda (x) (* x x))
       '(1 2 3 4 5))
→ '(1 4 9 16 25)
```

As for the Fold, since we want to have sum of elements we need to set initial value of accumulator as 0 and we can pass + as a function.

```
(fold (+) 0 '(1 2 3 4 5)) → 15
```

Combination of map and fold for sum of squares would look like this:

```
(define (sum-of-squares v)
  (fold (+) 0 (map (lambda (x) (* x x)) v)))

(sum-of-squares '(1 2 3 4 5)) → 55
```

If we think about TF-IDF calculation for URLs, we need to apply given function for each URL and fold calculated results by predefined formulas, thus MapReduce programming model can be very handy for us. In order to calculate Term Frequencies and Inverse Document Frequencies we need to generate data for several intermediate steps such as word count in each URL, total number of words for the collection of URLs and etc. Therefore calculating everything in a single MapReduce task can be overwhelming with respect to algorithm design and implementation. To address this issue we can split the task into four interdependent MapReduce iterations, this will give us possibility to perform step-by-step analysis on the dataset and during final iteration we will have a possibility to calculate score for each URL part in the dataset.

First MapReduce Iteration

During first MapReduce iteration we need to split each URL into parts and output each URL part separately, where URL identifier and a part itself will be defined as keys. This will give possibility to the reducer to calculate total occurrence of the specific URL part in the dataset.

The implementation would look like this:

Map:

Input: (raw URL)

Function: Split the URL into parts and output each pert.

Output: (urlPart;urlCollectionId, 1)

Reduce:

Input: (urlPart;urlCollectionId, [counts])

Function: Sum all the counts as n

Output: (urlPart;urlCollectionId, n)

Second MapReduce Iteration

For the second iteration input for the map function is the output of the first iteration. The purpose of the map function is to modify key values of the first iteration so that we can collect number of times each URL part appeared in a given collection. This will give possibility to the reducer to calculate number of total terms in each collection.

The implementation should look something like this:

Map:

Input - (urlPart;urlCollectionId, n)

Function – We need to change the key to be only collection identifier and move the url part name into the value field.

Output - (urlCollectionId, urlPart;n)

Reduce:

Input - (urlCollectionId, [urlPart;n])

Function – We need to sum all the n's in the collection of URLs as N and output every url part again.

Output - (word;filename, n;N)

Third MapReduce Iteration

The goal of the third iteration is to calculate URL part frequency in the collections. To do so we need to send to the reducer calculated data for the concrete URL part, therefore

map function should move *urlCollectionId* to the value field and the key should only contain *urlPart* value. After such modification reducer can calculate how many times urlPart appeared across different collections.

The implementation should look something like this:

Map:

Input - (urlPart;urlCollectionId, n;N)

Function - Move *urlCollectionId* to value field

Output - (urlPart, urlCollectionId;n;N;1)

Reduce:

Input - (urlPart, [urlCollectionId;n;N;1])

Function - Calculate total number of *urlParts* in a collection as ***m***. Move *urlCollectionId* back to the key field

Output - (urlPart;urlCollectionId, n;N;m)

Fourth MapReduce iteration

At this point we will have all required data to calculate Term Frequency and Invert Document Frequency for each URL part, such as:

- n – How many times given URL part appeared in the collection
- N – Number of occurrences of the given URL part across different URL collections.
- m – Total number of URL parts in a collection.

For the TF we will use following formula:

$$f_{t,d}$$

Where f represents number of times given word appeared in a document.

We can calculate Inverse Document Frequency using following formula:

$$\log \frac{\mathrm{N}}{n_t}$$

Where N is total number of documents and n_t - number of terms in a document. We do not need reduce function for this final stage, since TF-IDF calculation done for each URL part in a given collection.

The Implementation should look something like this:

Map:

Input - (urlPart;urlCollectionId, n;N;m)

Function - calculate TF-IDF based on n;N;m and D. Where D is the total number of url collections.

$$TFIDF = \frac{n}{N} \times \log(\frac{D}{m})$$

Output - (urlPart;urlCollectionId, score)

As we already described we need to label each URL part as *static* or *dynamic* based on the score. As soon as calculation is done in the map function, we can check the score value against predefined threshold. If score is lower than the threshold we will assign mark URL part as *dynamic*, else – *static*. In addition the parameter identification we would also like to exclude some random URLs, which can be marked as "not important" for the application. TF-IDF is essentially the importance score of the given URL part. If the sum of the scores for all the parts in URL is really low for example 1.7647059e-10 this means that URL appeared 1 or 2 times in the dataset, thus we can say that in the application with more than 200 000 services, URL is not important because it wasn't used intensively enough to be considered as a service, and we defined service as the endpoint which provide a set of operations that are of direct value to the end-user. In this research we will use threshold of 0.005 to filter out not important services.

To summarize upon successful completion of the MapReduce Iterations analysis phase will be finished. At this point we will have each URL part across the dataset marked as *dynamic* or *static*, thus forthcoming URL Reduction Phase will have all the necessary information to group URLs and generate appropriate service names for them.

URL Reduction Phase

URL Reduction Phase take as an input list of following class, which is generated during Analysis phase:

```
class AnalyzedUrlData {
  public String appId
  public int urlId
  public List<String> dynamicParts = []
  public List<String> staticParts = []
  public double score
  public String originalUrl
}
```

For the service name generation we are mainly intersted in three fields of this class, which are:

- dynamicParts – holds list of dynamic parts in the current URL.
- staticParts – holds list of static parts in the current URL.
- originalUrl – Value of the the original URL.

For the service name generation we can iterate over the list of dynamic parts and replace it in the `originalUrl` field with predefined static string. Below is the small example using Groovy programming language:

```
dynamicParts.each { part ->
  originalUrl.replace(part, "dynamic")
```

```
}
```

Figure 10: Code snippet for replacing dynamic parts in the URL with static value

Upon replacing all the dynamic parts with a static string, we will have new URL-based service names. Let's look at the following example:

Suppose we have list of following URLs

`/api/v2/invoice/inv123`

`/api/v2/invoice/inv424`

`/api/v2/invoice/inv948`

`/api/v2/invoice/inv000`

`/api/v2/invoice/inv782`

`/api/v2/invoice/inv545`

During analysis phase we have successfully marked `api, v2, invoice` as static parts and invoice identifiers as dynamic parts. Therefore we have following list of dynamic pats for the URL Reduction Phase:

`[inv123, inv424, inv948, inv000, inv782, inv545]`

As soon as code snippet presented in the figure 10 finishes its execution we will have the following result:

`/api/v2/invoice/dynamic`

`/api/v2/invoice/dynamic`

`/api/v2/invoice/dynamic`

`/api/v2/invoice/dynamic`

`/api/v2/invoice/dynamic`

`/api/v2/invoice/dynamic`

Now we have list of identical URLs and we can use Groovy's built-in function to remove duplicates.

```
['/api/v2/invoice/dynamic'
 '/api/v2/invoice/dynamic',
 '/api/v2/invoice/dynamic',
 '/api/v2/invoice/dynamic',
 '/api/v2/invoice/dynamic',
 '/api/v2/invoice/dynamic'].unique() -> ['/api/v2/invoice/dynamic']
```

URL Reduction Phase gives us possibility to significantly reduce amount of data and generate proper service name for URL-based services. After service name generation Graph Building Phase can build graph representation of the URLs, with generated statistics per node.

Graph Building Phase

Graph building is the final stage of the service discovery lifecycle and the main goal of it is to create tree like JSON structure from the generated service names. In Groovy programming language we can define tree as the following:

```
def tree = { ->
  return [:].withDefault {
    tree()
  }
}

def node = tree()
```

Figure 11: Code snippet for generating tree data structure implemented in Groovy

Where `node` is the implementation of the Java based Map interface. Groovy is dynamic language therefore we can populate the map with the following code:

```
node.root.child1
node.root.child2
node.root.child3
node.root.child1.child1_1
node.root.child2.child2_1
```

Figure 12: Code snippet for populating tree data structure

Figure 13 shows us graph diagram of the code snippet presented above.

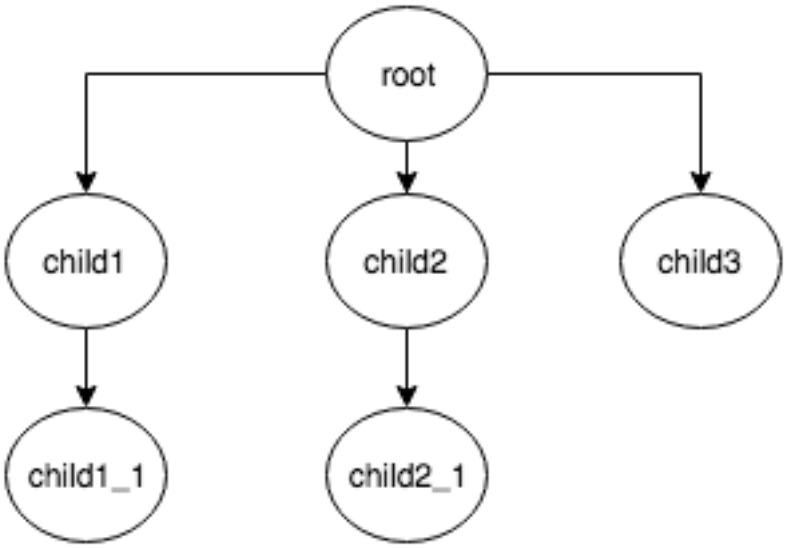


Figure 13: Graph representation of the code snippet presented in the figure 12

Now as we have an idea how tree data structure can be built with Groovy we can start thinking on how to generate JSON tree representation for generated services names. We know that Graph Building Phase receives generated service names in the following format:

```
'/api/v2/invoice/dynamic'
'/api/v2/invoice/dynamic/add'
'/api/v2/invoice/generate'
'/api/v2/invoice/dynamic/update'
'/api/v2/invoice/dynamic/generate
```

Figure 14: Example input for graph building phase

In order to build a tree from the service names, we need to split each service name with application specific delimiter.

```
'/api/v2/invoice/dynamic' -> ['api', 'v2', 'invoice', 'dynamic']
'/api/v2/invoice/dynamic/add' -> ['api', 'v2', 'invoice', 'dynamic', 'add']
'/api/v2/invoice/generate' -> ['api', 'v2', 'invoice', 'generate']
'/api/v2/invoice/dynamic/update' -> ['api', 'v2', 'invoice', 'dynamic', 'update']
'/api/v2/invoice/dynamic/generate -> ['api', 'v2', 'invoice', 'dynamic', 'generate']
```

Figure 15: Tokenization example of the input data for the graph-building phase

Using `node` variable from the figure 11 we can build a tree with a code snippet presented in the figure 15.

```
listOfTokenizedServices.each { tokenizedService ->
  tokenizedService.collect { service ->
    node."$service"
  }
}
```

Figure 16 illustrates final output of the service discovery tool for the example presented in the figure 14.

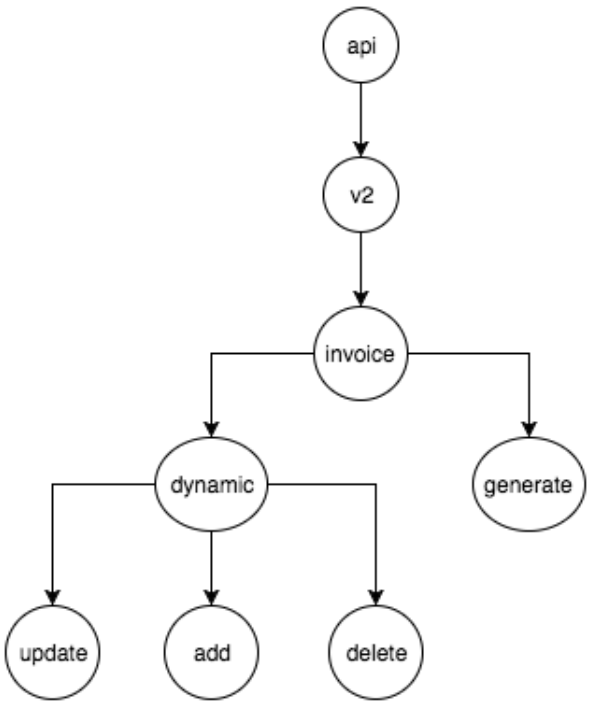


Figure 16: final output of the Graph Building Phase

During tool implementation section we will see that graph building in real-life scenario can become rather complex operation, especially considering the fact that we want graph in very specific format (Which we will discuss during tool development section). It requires deep recursion, which means we need to explore every level of the list.

Graph Building step wraps up our URL-based service discovery. Let's briefly take a look what we achieved.

- We were able to successfully tokenize and clean large dataset of URLs without any prior information.
- We managed to perform initial analysis to identify application specific delimiter, which gave us possibility to support different types URL design.
- We performed URL part classification as static or dynamic depending on calculated TF-IDF score.
- We generated new service names and significantly reduced number of URLs.
- We built URL tree structure to illustrate application API design.

To summarize we were able to adapt different forms of service discovery techniques to our needs, giving us possibility to discover important services in the URL dataset. As we now have the idea about the algorithms we need to implement we can start thinking about the actual tool development, which will be discussed in the upcoming section 3.3.

3.3 Tool Development

In this chapter we will try implement all the designed algorithms and ideas discussed during the previous chapter. Tool that we will implement can be used as standalone web application, which runs inside the Jetty container, or as Java library that can be added as a dependency in existing project. For the development, combination of Groovy and Java programming languages were used. Groovy is dynamic language for the Java platform, with the features similar to functional programming languages therefore implementing algorithms in it are much faster than in traditional Java language. In addition to Groovy and Java following tools, frameworks and libraries will be used:

- Gradle – Build automation system with Groovy based domain specific language.
- Angular – Front-end development.
- Treant-js – SVG based JavaScript library for drawing tree diagrams.
- Spring framework – Back-end development.
- Jetty – Application server.

For the sake of brevity of this research, I won't describe each library and framework in details; instead I will provide the concrete examples during this chapter how and why they are used in terms of development process.

Before going into implementation details I would like to mention that during the review of the development process I will not cover whole codebase, but rather describe the most interesting cases and how they were solved. Also I will try to give an overall architectural picture and describe public API and how it can be used in different projects.

Source code is available on a Github - https://github.com/lkokhreidze/service-discovery. Please also note this this tool will be used at Plumbr for research purposes. Core algorithms will remain intact, but from time to time some code modifications may take place to improve overall performance or to add new features.

3.3.1 Overall Architecture

As it was stated in during section 3.2, before starting data processing we need to clean the dataset, since it can contain large amount of irrelevant URLs, which may corrupt final result. During research we found out that URLs captured by Plumbr at the JVM boundaries may contain different types of static content such as '.html', '.js', '.text', etc. Based on the various query results, URLs ending with the following extensions are eliminated from the input dataset:

'.html', '$', '.php', '.js', '.txt', '.css', '.jtp', '.ico', '.gif', '.text'

This was achieved by implementing following Groovy method

```
public static List<String> parse(String id, List<String> records) {
  CollectionUtils.init()
  records.collect {
    "$id;$it".toString()
  }.filter '.html', '$', '.php', '.js', '.txt', '.css', '.jtp', '.ico', '.gif'
}
```

Where `filter` is dynamically added in the `List` interface implementation with the following code:

```
List.metaClass.filter { String[] patterns ->
    delegate.removeAll { entry -> patterns.any { entry.endsWith(it) } }
    delegate
}
```

With the help of this method we exclude any URL which can potentially corrupt final output of the Service Discovery tool. After performing initial data maintenance we can start actual discovery process.

During the research in chapter 3.2 we divided URL-based service discovery into five consecutive steps. Tool implements those steps with the help of a `DiscoveryProcessor` interface. Each class that implements this interface should provide solution for the following 5 methods:

```
public void init(List<String> services, Configuration configuration)

public DiscoveryProcessor group()

public DiscoveryProcessor analyze()

public ResultSetWithStats reduce()

public TreeResultSetWithStats toGraph()
```

This interface is implemented in two discovery provider classes: `DiscoverUrlServices` and `DiscoverRmiServices`. Research presented herein is mainly concentrated on Plumbr use case and dataset provided by Plumbr does not contain only URL-based services. RMI based services extracted from the Plumbr supported frameworks are also presented in the dataset. As the main focus of this master's thesis is service discovery for URLs, we need to distinguish between RMI and URL based service names. To do so during invocation of the `init` method we can apply String parsing rule, which will populate records for corresponding discovery provider. Below is the service name parsing rules to populate records for the corresponding classes. For RPC-based service discovery:

```
this.services = services.findAll { serviceName ->
  !serviceName.startsWith("/") && serviceName.contains(".")
}
```

For URL-based service discovery:

```
this.services = services.findAll { serviceName ->
  serviceName.startsWith("/")
}
```

Those service name-parsing rules are based on the careful dataset review and research through the research paper [5]. As we discovered during the literature review, usually RPC based services start with CamelCase class name followed by method name, thus

service names matching this specification will be sent to RMI discovery provider class (`DiscoverRmiServices`). On the other hand URL-based services always start with "/", therefore service names matching this rule will be sent to URL discovery provider class (`DiscoverUrlServices`). For this master's thesis RPC based services do not qualify for the analysis, therefore such services are ignored. Although during chapter 5 we will discuss possibility of implementing RPC based service analysis for the future work.

3.3.2 Discovery Process

After performing initial data maintenance and service name classification we can start actual data process. During this we will sequentially implement remaining `group`, `analyze`, `reduce` and `toGraph` methods.

Group()

As we stated during previous chapter purpose of this method is to perform URL based tokenization and group tokenized URLs based on the identified URL style. If we will take a look at the method signature it takes no arguments and returns implementation of `DiscoveryProcessor` interface. The reason to do so is that by itself, this method do not provide any relevant information to the end user, but rather represents intermediate step before actual output, thus it is logical to return intermediate state of the discovery provider class with populated grouped dataset. Below is the initial implementation of this method:

```
@Override
DiscoveryProcessor group() {
  this.grouped = this
    .initialGroups
    .collect { k, v ->
      def d = delimiterAnalyzer.getDelimiter(k)
      def group = v.collect { split(it, d) }.groupBy{ it[0] }
      [(k): group]
    }
  this
}
```

Where `this.initialGroups` is the dataset populated during initialization phase by `init` method. `delimiterAnalyzer.getDelimiter(k)` is an entry point for the application specific delimiters, where *k* variable represents application identifier. The `group` variable will hold the list tokenized URL parts grouped by the first part of the URL. Final output will be saved in the `this.grouped` map representation with application identifier as key. This intermediate map dataset can be used by forthcoming method implementation in the service discovery lifecycle.

This was the initial approach for building groups from the original dataset in order to get different number of documents for TF-IDF calculation. During the research in the

previous section we mentioned that this approach has its drawbacks, which we will discuss, with possible solutions. Up-to-date tool version is implemented using new, more advanced approach, which more or less isn't dependent on the data size or structure. Technique presented above was demonstrated in order to show progress of the research and its bottlenecks as we move along with the service discovery process.

Analyze()

During the chapter 3 we described Analysis phase as the backbone of the URL-based service discovery, thus it is no surprise that actual implementation is the largest compared to any other. This method uses following classes for calculating TF-IDF score:

- `uni.tartu.algorithm.TfIdf`
- `uni.tartu.algorithm.MiniMapReduce`

`MiniMapReduce` class holds two static classes `Mapper` and `Reducer` with *map* and *reduce* methods respectively. Both this methods accept closure[4] as parameter, so that Mapper and Reducer will have a clear understanding on how to perform its functionality.

During the research in chapter 3 we mentioned that due to large amount of intermediate processes and data preparation before actual TF-IDF calculation, implementing everything in single MapReduce iteration could be too overwhelming, therefore we decided to split MapReduce process into four steps, where first three iterations are for the data preparation and final process is the TF-IDF score calculation. Considering this fact, `TfIdf` holds 3 inner static classes, which should be passed into `TfIdf` constructor. Each of the inner class has an object of `Mapper` and `Reducer` classes. With this approach we ensure 3 things:

1. Instance of the `TfIdf` class cannot be created without providing three MapReduce iteration steps.
2. While initializing iterations closure specifications for `Mapper` and `Reducer` should be passed as parameters.
3. `TfIdf.calculate()` is the final method for score calculation and it cannot be performed without satisfying two upper conditions first.

We now have a brief understanding how analysis work and what requirements need to be satisfied before calculation TF-IDF. Now let's take a look at each iteration separately.

Below is the code for the first iteration with closure specification:

```
/**
 * first MapReduce job closure specification
 *
 * Mapper
 **/
FirstIteration.build({ k, v ->
```

[4] http://groovy-lang.org/closures.html – A closure in Groovy is an open, anonymous, block of code that can take arguments, return a value and be assigned to a variable.

```
  v.collect { i ->
    i.collect { j ->
      def keys = getKey(k as String)
      j.equals(keys[0]) ?: "${keys[0]};${j}__${keys[1]}".toString()
    }
  }.flatten().each {
    def keys = getKey(it as String)
    if (keys) {
        def urlPart = keys[0],
            urlId = keys[1] as int,
            parts = split(urlPart, ";")
       populate(parts, urlPart, urlId, originalServices.get(urlId).rawUrl)
     }
  }
},
/**
 * Reducer
 **/
{ map, k, v ->
 map << [(k): v.sum(0)]
})
```

Figure 17: Code snippet for first MapReduce iteration

As we already know first step in TF-IDF calculation is to count how many times given URL part appears in the application URLs. Analysis is performed on the data generated during the grouping phase. Let's look at mapper closure and analyze it line by line. Entry point for closure is ***k*** and ***v*** parameters. ***k*** parameter is the application identifier and ***v*** - values. As we know for each application identifier (***k***) we have list of tokenized URLs, therefore in order to calculate count for each of the URL part we need nested loop. In Groovy we can achieve this by declaring to nested `collect` closures, where first collect gets the array of tokenized URLs and second one – each element in the array. Using `flatten()` method we can `populate` intermediate data structures, that are required during different parts of the service discovery process. `populate` method puts URL part value and 1 in the corresponding data structure. Below is the method implementation:

```
def name = parts.length < 2 ? 'null' : parts[1]
putUrlIdHolder(name, new UrlInfoData(urlPart: urlPart,
                                     urlId: urlId,
                                     originalUrl: originalUrl))
put((urlPart), 1)
```

`putUrlIdHolder` populates data structure that during reduction phase will be used to retrieve original URL value and modify static and dynamic part. At this point we are interested in `put((urlPart), 1)`, which inserts into the dataset URL part value as the key and 1 as count. This will be key-value pair that is sent to the reducer.

As we can see from the figure 17, reducer part is fairly simple after constructing proper mapper; it will sum all the 1's for a specific key. As a result, input for the second iteration will be count of occurrences of a certain URL part in a given URL collection. With the

first iteration we answered first TF-IDF question regarding how many times given URL part appeared in the collection, defined during the previous chapter.

```
/**
 * second MapReduce job closure specification
 *
 * Mapper
 **/
 SecondIteration.build({ k, v ->
   def arr = (k as String).split(";")
   arr.length < 2 ?: put(arr[0], "${arr[1]};${v}".toString())
 },

 /**
  * Reducer
  **/
 { map, k, v ->
   int N = v.sum {
     (it as String).split(";")[1] as int
   }
   v.flatten().each {
     def parts = (it as String).split(";"),
         key = "${k};${parts[0]}",
         val = "${parts[1]};$N"
     map << [(key): (val)]
   }
   map
 })
```

Figure 18: Code snippet for the second MapReduce iteration

During the second iteration we should find out number of occurrences of each URL part in a given URL collection. In order to do so we need to calculate how many URL parts does a URL collection have? For that mapper closure should move URL name to the value field, so that key for the reducer will be only collection identifier. During first iteration we joined key value for collection identifier and URL name with ';' symbol, therefore in the mapper we can split key part with the very same identifier and move URL name to the value fields. Mapper in the figure 18 illustrates how we can do this.

As soon as mapper is done, Reducer will receive URL collection identifier as the key and URL part with count as the value - (urlPart;n). To calculate total number of URL parts in a collection we can sum all the ***n*** as ***N*** and output URL part with collection identifier as the key and n, N as the value. Reducer in the figure 18 demonstrates how we can do this using built in Groovy and Java string functions.

```
/**
 * third MapReduce job closure specification
 *
 * Mapper
 **/
 ThirdIteration.build({ k, v ->
```

```
    def parts = (k as String).split(";"),
        id = parts[1],
        urlPart = parts[0] ?: 'null'
    put((urlPart), ("$id;${v};1"))
  },
/**
 * Reducer
 **/
 { map, k, v ->
    def m = v.sum {
      (it as String).split(";")[3] as int
    }
    v.flatten().each {
      def parts = (it as String).split(";"),
          key = "${k};${parts[0]}" as String,
          val = "${parts[1]};${parts[2]};$m" as String
      map << [(key): (val)]
    }
    map
 })
```

Figure 19: Code snippet for the third MapReduce iteration

Third iteration is the final one in terms of data preparation for the TF-IDF calculation. At this point for any given URL part we have ***n*** – occurrence of an URL part in a collection and ***N*** – total number of URL parts in a collection. Only that is remaining is the ***m,*** which is will tell us in how many collections given URL part appeared. As figure 19 shows to do this we need to tell the mapper closure to set the key value to as ***urlPart*** with the value (***collectionId;n;N;1***) where **1** is occurrence value for the URL part. Reducer parts in the figure 19 sums all the **1**'s for the given URL part as ***m*** and returns URL part and collection identifier as the key with ***n***, ***N*** and calculated ***m*** as the value.

As we now have all required data we can calculate ***TF-IDF***. In terms of implementation we can define function in `uni.tartu.algorithm.TfIdf` class that will take analyzed data as the argument and calculate TF-IDF score for each URL part. In order to calculate TF-IDF we need to get one additional parameter ***D*** – which is defined as total number of documents in a corpus. In our case it will be total number of URL collections. Below is method signature.

```
Map<String, AnalyzedUrlData> calculateTfIdf(Map data, long D, Configuration conf)
```

Figure 20: Method signature for TF-IDF calculation

As we can see method returns Map where original URL value will be defined as key. The reason for this is that during URL reduction phase it will be easier and faster to get analyzed data for any given URL and generate new service name based on that. For further proceeding it is important to understand structure of `AnalyzedUrlData` class, so let's take a brief look at it.

```
class AnalyzedUrlData {
  public String appId
  public int urlId
  public List<String> dynamicParts = []
  public List<String> staticParts = []

  @Override
  public String toString() {
    return "$accountId;$originalUrl - ($urlId) params: ${urlPart} static parts:
            ${staticParts}".toString();
  }
}
```

Figure 21: Class structure for holding results for TF-IDF analysis.

As we can see most of the attributes are standard and we used them during different phases of tool implementation. What are interesting in this class are `dynamicParts` and `staticParts` defined as list of strings. During TF-IDF calculation we will fill this class based on calculated score. For static parts score will be close to 0.0 meaning that they appeared multiple times across different URL collections. And for dynamic parts score will be much greater than 0. During initial research tool showed the best results with the threshold of 0.003. Let's take a look how data will be populated for the following example:

`/api/v2/invoice/inv123`

Suppose we calculated TF-IDF score for each URL part. Given threshold 0.003, we will populate data for `Map<String, AnalyzedUrlData>` as following:

```
(/api/v2/invoice/inv123, new AnalyzedUrlData(accountId: 'sampleAppId',
                                             urlId: 1,
                                             staticParts: ['api', 'v2', 'invoice'],
                                             urlPart: ['inv424']))
```

As we can see for any original URL we have its TF-IDF analysis results. This will be useful during URL reduction phase, when we need to quickly look up for the results for a specific URL.

This concludes our analysis phase. At this point we saw how TF-IDF score can be calculated in Groovy using MapReduce programming model. Now we have everything at place for reducing number of URLs and for proper service name generation.

Reduce()

After constructing proper `AnalyzedUrlData`, reduce() method functionality becomes rather simple. When *reduce* method starts new service name generation, it will have all required data at place. Code snippet presented in the figure 22 shows how *reduce* method is implemented in the URL discovery provider class.

```
@Override
DiscoveryProcessor reduce() {
  log.info("started reduction phase for URL discovery")
  def originalSize = this.originalServices.size()
  def urlReducer = new UrlReducer(scores)
  this.reducedUrls = urlReducer.reduce()
  this
}
```

Figure 22: Reduce method implementation.

As we can see `uni.tartu.algorithm.UrlReducer` is the class which takes analyzed URL data in the constructor and performs service name generation with `reduce()` method. Logic how it is done is quite simple. Basically we have original URL data as the key and analyzed data as the value. As we already reviewed during *analyze* method discussion, each analyzed URL data holds static and dynamic parts in separate class fields (see figure 21). Therefore all we need is to iterate over map, get its value, find dynamic part in the URL key and change it with some predefined static content. Figure 23 shows implementation of reduce method in ***UrlReducer*** class

```
public List<String> reduce() {
  List<String> reducedUrls = []
  for (AnalyzedUrlData it in this.analyzedUrls) {
    def delimiter = delimiterAnalyzer.getDelimiter(it.accountId)
    def regexToInject = 'PARAMETER_PART'
    def currentStr = it.originalUrl
    it.urlPart.each { dynamic ->
      def inj = delimiter + regexToInject + delimiter
      currentStr = replace(dynamic, inj, currentStr, delimiter)
    }
    reducedUrls.add(currentStr)
  }
  reducedUrls.collect().unique()
}
```

Figure 23: *reduce* method implementation for uni.tartu.algorithm.UrlReducer class.

As soon as we will replace or dynamic parts with static content we can use Groovy's built in function `unique()` to output unique newly generated service names. URL reduction phase gives us possibility to create more meaningful service names from raw URL data and remove duplicate records. During the next section concerning result evaluation and case study at Plumbr we will see how raw URLs can be dramatically reduced in real-life scenarios.

ToGraph()

After proper service name generation we can start with the final step for the URL-based service discovery process. As we already mentioned during the initial research graph-building phase is rather complex step, with deep level recursion. Before exploring how it can be developed, let's take a look how we want to display our graph. First and foremost tree representation of the graph always have root. So before building any other node we should define what the root is. Each node of the tree should have following attributes:

- Name – node name, in our case it will be URL part name
- Children – List of children of a current node
- HTMLclass – If node has children than value should be 'the-parent', else nothing
- Collapsed – Boolean field, which indicates if the children of a current node are collapsed.

All of the above mentioned attributes are required in order to draw proper tree representation in JavaScript. We will see actual examples in the upcoming section.

Actual implementation of the tree builder is in the `uni.tartu.algorithm.tree.TreeBuilder` class. Due to the complexity of the class I won't include whole source code of the algorithm in this research, instead in the 24 I will provide two main recursive methods, which play major part in tree construction.

```
private def collectNodes(e) {
  e.with {
    if (!(it instanceof Map)) {
      return []
    }
    it.collect { k, v ->
      if ('children'.equals(k)) {
        return [:]
      }
      def children = []
      children.addAll(collectNodes(v))
      constructNode(k as String, children)
    }
  }
}

private def constructNode(String k, List children, boolean collapsed = true) {
  def node = tree()
  node.text.name = k
  if (children) {
    node.HTMLclass = 'the-parent'
    node.children = children
    node.collapsed = collapsed
  }
  node
}
```

Figure 24: Partial implementation of the *TreeBuilder* class.

In the `collectNodes` method we will terminate the recursion if the ***e*** parameter isn't instance of the **java.util.Map**, which means that parameter ***e*** has no inner attributes or children and we reached current max level of the recursion. In any other case we will try to collect all the children of the current node and write them into `children` field. `constructNode` build the required node structure, where ***k*** parameter is the URL part name and ***children*** is the list of sub-nodes of the current node. During the initial research we mentioned that we can recursively define the tree where each Map attribute will inner field of a current node (see figure 11). This `def node = tree()` piece of code gives us possibility to do so. Each new attribute of the `node` variable will be defined in a existing node structure. This enables us to create complex tree structures in more easy way compared to classic Java approach.

After ***toGraph()*** method finishes operation, we will be done with URL based service discovery. We defined ways how we can perform initial data filtering, complex service grouping, TF-IDF score calculation using our own implementation of MapReduce programming model, large set of string mutation and dynamic graph building with complex structure. All of this was implemented using Groovy-programming language with some standard Java features. In the upcoming section we will see how well tool is performing in real life scenarios, using data provided by Plumbr.

4. Evaluation

In this chapter we will discuss experiments and their technical setup. Also we will see the final output of the developed tool for Plumbr case to understand how well developed algorithms and application scale in real life scenario. In order to understand this before-after analysis will be performed to see percentage of data reduction and improved scalability of Plumbr. Last but not the least we will discuss current limitations of the tool and threats to validity.

4.1 Experimental setup

In this section we will try analyze data acquired from the Plumbr database using developed service discovery tool. For the research purposes we would like to select accounts with large enough data and with completely different URL-based service designs. For this objective following query was designed:

```
select count(*) as count, t.id
from table**** as t
group by t.id
order by count desc;
```

Based on query results two accounts were selected with completely different URL design. For the sake of confidentiality URL values will be modified as we will move along with the research, but actual URL style will remain the same. In this paper we will

refer to the above-mentioned accounts as "**account-A**" and "**account-B**". Before proceeding with the research let's summarize what we know about the dataset.

Account-A

- 103,193 detected services in total.
- Contains controller based services.
- Approximately 0.43% is noise.

Account-B

- 279,521 detected services in total.
- Does not contain controller based services.
- Approximately 0.2% is noise.

As we can see there are significant difference between accounts among all the criteria. But at this point the most interesting thing for us is URL delimiter. During this master's thesis we mentioned Delimiter Analyzer algorithm several times. It is the very first algorithm that will be evaluated against the input data after the initial noise filtering. Its main objective is to find account specific delimiter and save it into the memory. As we saw during Tool Development section discovered delimiter would be used almost in all of the phases of service discovery. Considering this facts it is important to do some initial research on the data to have some understanding on what kind of results should we expect from the algorithm. In order to do so query presented in the figure 25 was implemented, that randomly fetches 1000 services, given account identifier as the parameter.

```
SELECT t.service
FROM table***** AS t
JOIN (SELECT CEIL(RAND() * (SELECT MAX(id)
                            FROM   table*****)) AS id) AS rand
WHERE t.id >= rand.id
AND t.identifier = :inputId
ORDER BY t.id ASC
LIMIT 1000;
```

Figure 25: SQL query for fetching random records.

Table presented in the figure 26 illustrates sample records extracted from the query results.

Account-A	Account-B
/sample/123d21asd1242123	/dynamic-news-title-hello-world.n1399j3.news-info
/operation/643/jkqflac_/go	/san-francisco-weather.w1319q3.news-weather

/operation/724/xa127beg/go	/new-york-daily-weather.n3419v2.news-weather
/operation/724/bl1d343/go	/bmw-cars-2010-title.c929ch3.car-rental
/operation/902/xa555beg/go	/dynamic-news-title-hello-california.n0000j3.news-info
/operation/981/sdcceer/go	/cars.n1399j3.car-guide
/operation/323/gurg12/go	/cars.n9238j3.car-guide
/feed/724	/world-news-2016.n1399j3.world-news
/crm/si2hs213f023	/cars.n8fhryr3.car-guide
/crm/heet1334023	/cars.njekwo10.car-guide
/crm/okrug8134fgs	/cars.njehey102.car-guide

Figure 26: Sample results from the query described in the figure 20.

Figure 26 clearly shows fundamental difference between two accounts in regards to URL structure and design. Most likely URL delimiter for the **account-A** is "/" and for **account-B** – ".", thus we know what to expect from the Delimiter Analyzer algorithm. Presented query results can also help us to build up initial assumptions about the final output of the tool. With this we sum up our introductory research, for now we have basic idea about the content of the dataset, with the minimal notion of the final results. In the upcoming sections we will take a look how each phase of the Service Discovery tool performed for the given data.

4.2 Summary of Results

Account-a

Figure 27 illustrates minimalistic version of the user interface for the service discovery process. On this screen all major statistics and results for the input dataset will be displayed. In the input field we specify the ID of the account (or application) we want to analyze. After this UI will inform us that service discovery process has started. Screen will dynamically inform user about the current progress.

Figure 27: User interface for starting service discovery process.

As we already know very first thing that service discovery process will do is to reduce the noise in the input dataset. We already covered in details how and why it is done, thus in this section we will only discuss the actual results. According to the logs during filtering process, tool discovered 1439 polluted URLs, below is the log from the tool:

```
2016-05-03 19:52:44.359  INFO 24154 --- [tp1091781053-19] uni.tartu.parser.Parser
: Found 1439 polluted URLs. Current filters: [.html, $, .php, .js, .txt, .css, .jtp,
                                              .ico, .gif, .text, .pdf]
```

Please also note URLs which are markes as polluted due to the applied filter will be part of the final statistics, which means that we will see what kind of URLs were ignored during the process.

After initial filtering, next steps will be actual service discovery phases that are defined in `uni.tartu.discovery.DiscoveryProcessor` interface and implemented in `DiscoverRmiServices` and `DiscoverUrlServices` which are located in `uni.tartu.discovery.providers` package.

```
2016-05-03 19:52:44.360  INFO 24154 --- [tp1091781053-19]
uni.tartu.discovery.DiscoveryRunner      : started service discovery process.
```

We are now at the point when we need to start initialization phase and discover account specific delimiter in order to process further.

```
2016-05-03 19:52:44.814  INFO 24154 --- [tp1091781053-19]
u.t.d.providers.DiscoverUrlServices      : started initialisation phase for URL
                                           discovery
2016-05-03 19:52:45.553  INFO 24154 --- [tp1091781053-19]
uni.tartu.algorithm.DelimiterAnalyzer    : started to analyze delimiter per account
```

As we can see tool successfully started analyzing delimiter for account and we received following results:

```
2016-05-03 19:52:45.743  INFO 24154 --- [tp1091781053-19]
uni.tartu.algorithm.DelimiterAnalyzer    : for account: account-A
got delimiter: {dot=1.6789262750, slash=99.9025139582}
```

In the beginning of this section we talked a bit about the assumptions and we said that it is very likely that for the **account-A** we will get the "/" as delimiter. During the research we agreed upon 90% threshold for selecting the account specific delimiter. As we can see algorithm discovered that in 99.9% cases slash is indeed used as URL delimiter and in 1.6% - dot. One may ask how come that sum of this two results is more than 100%?! Actually using one delimiter in the URL does not exclude possibility to use another one as well. This means that we have really interesting cases where URL delimiter is not one symbol but combination of several ones. Algorithm will select "/" as account delimiter given that it is used in almost every URL presented in the dataset.

As soon as we discovered account specific delimiter we will move further with the process. Logs for the remaining phases is pretty straight forward, therefore I will just include logs, that describes intermediate statistics before the final output.

```
2016-05-03 20:45:38.017  INFO 24416 --- [tp1280128554-19]
u.t.d.providers.DiscoverUrlServices          : started grouping phase for URL
                                               discovery
2016-05-03 20:45:38.627  INFO 24416 --- [tp1280128554-19]
u.t.d.providers.DiscoverUrlServices          : created intermediate URL groups with
                                               size: 101
2016-05-03 20:45:38.628  INFO 24416 --- [tp1280128554-19]
u.t.d.providers.DiscoverUrlServices          : started analyzing phase for URL
                                               discovery
2016-05-03 20:45:42.969  INFO 24416 --- [tp1280128554-19]
uni.tartu.algorithm.TfIdf                 : started  calculating TF-IDF score
2016-05-03 20:46:11.197  INFO 24416 --- [tp1280128554-19]
uni.tartu.algorithm.TfIdf                 : got TF-IDF scores with size: 14276
2016-05-03 20:46:11.197  INFO 24416 --- [tp1280128554-19]
u.t.d.providers.DiscoverUrlServices          : started reduction phase for URL
                                               discovery
2016-05-03 20:46:11.736  INFO 24416 --- [tp1280128554-19]
u.t.d.providers.DiscoverUrlServices          : reduced and generated URLs with size:
                                               308
2016-05-03 20:46:11.752  INFO 24416 --- [tp1280128554-19]
uni.tartu.storage.ResultSetWithStats     : started building graph from reduced URLs
2016-05-03 20:46:11.737  INFO 24416 --- [tp1280128554-19]
uni.tartu.discovery.Discovery             : got intermediate results, building result
                                            set. Discovery process is done!
```

URL-based service discovery process is now finished. We were able to generate all required data in order to display statistics and built tree. Below are screenshots from the Service Discovery tool UI that illustrates what final output looks like for **account-A**.

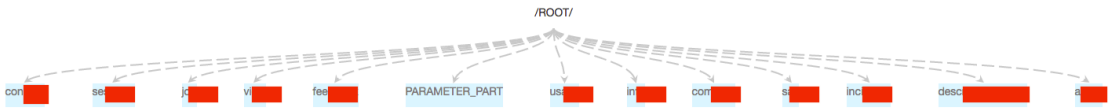


Figure 28: Discovered first level URL services for **account-A**.

Figure 28 shows us what we call First Level service names. First Level service names represent entry point for generated service names. For instance if we have following raw URL data: "/api/rest/invoices/123jfew", first level service would be ***api***. In the figure 29 we can low level service names, in other words "children" of the upper level services.

Displaying generated services like this gives us opportunity to build dependency between URL parts, therefore end user has possibility to observe resource structure of the URLs. Any application tends to grow and as it expends and advances, it gets even harder for architects or developers to maintain their software. With the help of Service Discovery tool they can actually see high level, structured overview of their application URL-based services captured during JVM monitoring.

Figure 30 concludes our service discovery results with interesting statistics calculated during the process.

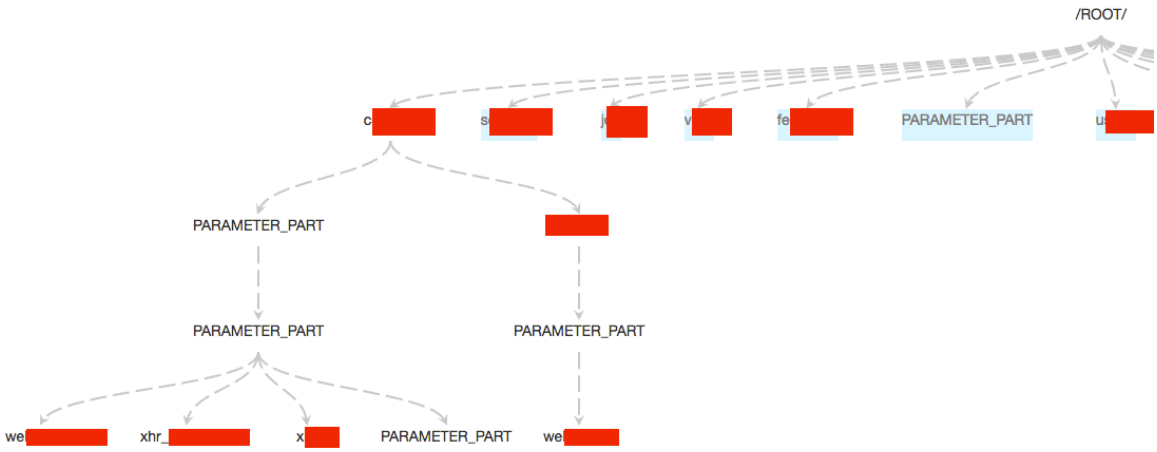


Figure 29: Example of generated low-level URL services for **account-A**.

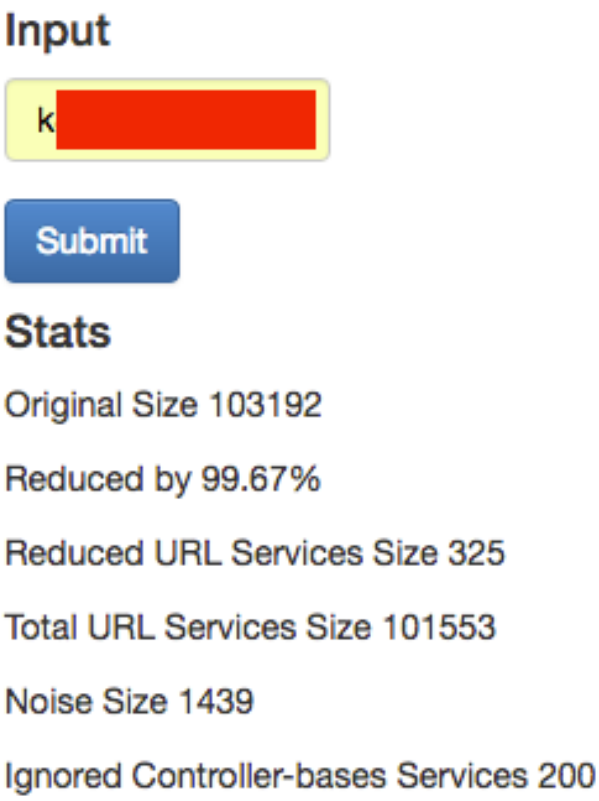


Figure 30: Calculated statistics for **account-A**.

Overall tool performed really well for the **account-A**. As we can see URLs were reduced by **99.67%**, **which** is 101228 services. From 101553 raw URL data we were able to discover 325 services and generate meaningful name for each of them.

Account-b

Discovery process for **account-b** will be the same, thus here I will only describe final output of the tool. It seems that **Account-b** has different URL style, presumably with dot as delimiter. Also data size is more than twice as large as for **account-a**, thus it will be interesting how well our solution performs in terms of larger dataset with completely different structure.

As we expected Delimiter Analyzer detected dot (.) as the URL parts delimiter for the **account-b.**

```
2016-05-04 00:24:53.929  INFO 25383 --- [tp1280128554-20]
uni.tartu.algorithm.DelimiterAnalyzer    : for account: 1d***************
                                           got delimiter: {dot=96.5803332315,
                                           slash=3.5785538401}
```

Figures 31 and 32 show us structural overview of discovered services, while figure 33 sums up statistical data for the **account-b.**

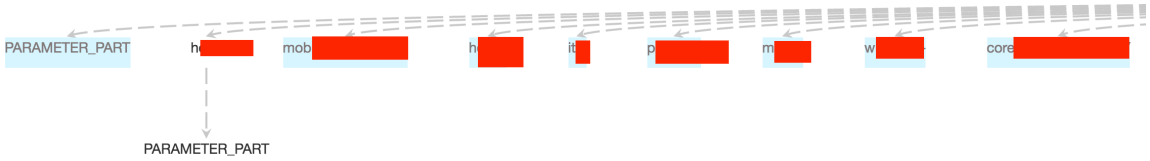


Figure 31: Discovered first level URL services for **account-B**.

Figure 31 shows us discovered first level URLs that will be used as major containers for the rest of URL parts. Figure 32 shows us low-level URLs for the selected first level part.

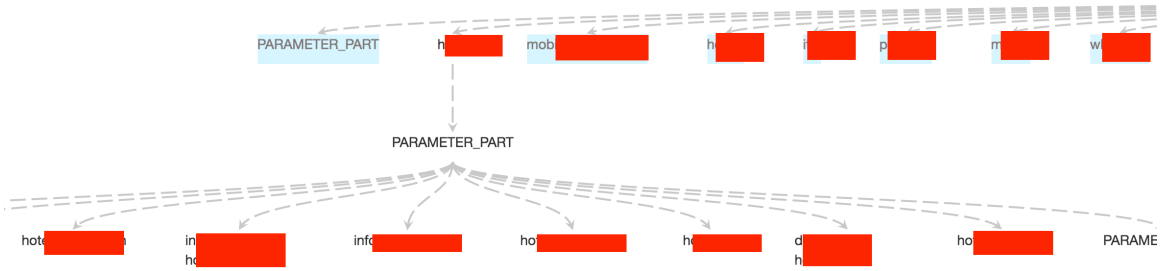


Figure 32: Example of generated low-level URL services for **account-B**.

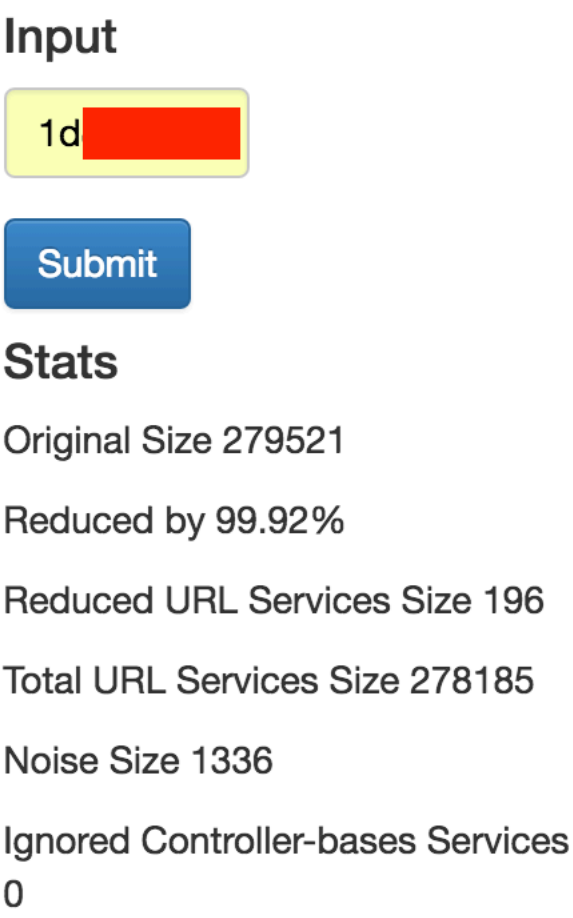


Figure 33: Calculated statistics for the **account-b**.

Based on the statistics presented in the figure 33 we can say that service discovery tool performed well for the **account-b**. In the Plumbr database this account has the largest amount of services, since it isn't implemented in one of Plumbr supported frameworks. We were able to successfully discover **196** services out of **278185**, which gave us reduction score of **99.92%**, which is really impressive.

Current implementation isn't the best one with respect to performance. For instance, calculation for **account-b** dataset takes approximately 2 minutes and 18 seconds to complete and to display results on the UI. For the **account-a** – 35 seconds. Time isn't dependent only on input data size. URL structure for the **account-b** is much more complicated, thus parsing, tokenizing and analyzing complicated URL structure takes quite amount of resources. Performance can be improved in the context of future research.

To summarize tool, which was implemented and designed during this master's thesis, was able to identify meaningful services from raw URL data, without having any information or description about the input data. This gives possibility to significantly reduce URL data among per account or application and group services with different parameters or path variables under the same generated service name.

4.3 Discussion

In this section we will examine discovered limitations for service discovery tool. Even though tool performed really well in for the large datasets with variety of registered URLs, but when it comes to small amount of data, tool fails to generate meaningful services. At this point it is hard to define what is "small" amount of data, since quality of discovery process also depends on the URL difference within the dataset itself. For example if we have input data with 5000 records and all of the URLs represent same service, with different path variables like in the following examples:

- /api/rest/invoices/14sfrfra/generate
- /api/rest/invoices/yhr1234/generate

Tool won't identify any service, because it won't have any comparable URL styles. During the discussion of the ***Group()*** method we identified some limitation and problems with the approach of having same style of URLs into the same group. Now we can discuss this issue, because this problem mainly occurs in case of small datasets. As we defined in the section 3.3.2, ***Group()*** method will try to save smaller chunks of the URLs in a same group, depending on their style. For example URLs starting with '**api**' would go to the same group, with 'rest' – to second one and so on. If we consider example of having same style of URLs as an input, ***Group()*** method will generate only one chunk for all of the URLs. As we said size of the groups in terms of TF-IDF algorithm, is the same as number of different documents. As we know number of documents in the algorithm is defined as ***D*** (see sections 3.2.2) and it is used to calculate Inverse Document Frequency score. In the case presented above we would have following situation, value of the ***D*** would be 1 because there is only 1 generated group, logically ***m*** which represents occurrence of a given URL part in different documents would also be 1, thus we would have following formula for IDF score calculation: $\log(\frac{1}{1})$ which is 0. It does not matter what the value of **n** or **N** is, final score of TF-IDF will always be 0, thus we won't be able to identify any parameter in the input data.

To address this issue we introduced `uni.tartu.algorithm.ServiceGrouping`, with `getMaxRange` method. This method takes the size of the input data as parameter and generates range of the grouping dataset using following formula:

$$\frac{dataSize}{1000}$$

For example for the data with 5000 services it will generate range of 1 to 5 groups. Services are randomly assigned to particular group. This gives us possibility to support low volume datasets with more or less same URL structure. But we should also note that there would be same issue if the data size were less than 1000. At this point only solution that comes to mind is to dynamically set group range depending on the data size. To do that we introduced `discovery.properties` file were user can specify the grouping ratio depending on the data which he/she wants to analyze. Suppose we have data with 500 URLs, than grouping ration of 100 would be sufficient enough to generate 5 groups before TF-IDF calculation.

It is worth mentioning that this approach does not eliminate the problem, but rather provides enough flexibility to its consumer to get meaningful results even on the small amount of data. At this point following properties can be set in the `discovery.properties`, which should be included in the class path of the tool:

```
discovery.importanceThreshold=0.005
discovery.parameterThreshold=0.003
discovery.filters=.html,$,.php,.js,.txt,.css,.jtp,.ico,.gif,.text,.pdf
discovery.groupingRatio=1000
```

We already discussed importance and parameter thresholds during the chapter 3. As for filters property it enables users specify filtering URL filtering rules of their choice, as we already know URL cleaning will be done at a very first stage of service discovery process. In this section we covered last dynamic property for the service discovery tool, which gives possibility to improve service discovery quality for small amount of data.

TF-IDF and MapReduce algorithms are still considered to be applicable for large amount of data, thus using this approach for low amount of URLs is still not the best option, even though we minimized impact as much as we could.

5. Conclusion & Future Work

Based on the conducted experiments and the case study, we can say that chosen approaches showed really good results on a reasonably large dataset with different URL designs. We managed to successfully perform noise reduction before actual analysis in order to avoid final output pollution. In addition, we have implemented “smart” tokenization technique suggested in the paper [4]. This gave us the opportunity to analyze URLs with completely different characteristics. As we observed in this research URLs tend to be rather different depending on the implementation and the framework, therefore analysis should flexible enough to handle various types of datasets. In order to achieve this we introduced discovery.properties file, which gives possibility to configure analysis for users needs. Visualization is the important part for any data analysis, in terms of URLs we have chosen to represent URLs in a hierarchy model, which gives high-level overview of the application service structure.

Although we answered our research questions and managed to identify services in the raw URL data for Plumbr use case, there are still things that can be improved in terms of future research. It includes, but isn’t limited to discovering important services through the RPC interface documentation. In this research we didn’t concentrate on this aspect of the service discovery, since it wasn’t relevant for the Plumbr use case. But as for the future research it can be implemented in the existing tool. As a result we can get a universal set of techniques that can generate RESTful style services through raw URL data as well as over RPC interface documentation. Research presented in the paper [4] can be good starting point for that.

In the previous chapter we spoke about current limitations for small datasets. Although we managed to introduce a way to minimize impact of false service discovery, more broaden research can be conducted in this respect. Maybe using different approach, like building similarity matrix for the URLs will give better results for small amount of data? That's the question that can be answered in the future studies.

In the research presented herein we introduced "minified" version of MapReduce for TF-IDF calculation. MapReduce is widely used programming model, which can handle large amount of data running in multi cluster environment. Most popular implementation of MapReduce in that regard is via hadoop ecosystem. In that case millions of raw URL data can be analyzed multiple applications in parallel. Although modifying current service discovery tool does not make much sense, since hadoop MapReduce is completely different framework, therefore new tool should developed for this use case. Also it is worth to mention to experiment with different variations of TF and IDF score calculations. Although TF-IDF version that we used in this research is considered to be the most appropriate for text mining, there was no proper research for using TF-IDF for URL data classification. Maybe different version can give better results?

Last but not the least we can implement Resource Model generation over the identified URL services. It will make perfect sense if RPC style service discovery is in place. Therefore we would have a scenario were completely unstructured URL data can be transformed into Resource Models along with the generated RESTful style services from RPC documentation. Combination of this two (URL data and RPC documentation) would give excellent REST Resource Model generator. Probably in that case we may need additional data for URLs, such as HTTP verbs to generate proper Resource Models. For RPC documentation we may infer HTTP verbs using more advanced natural language processing techniques. For example method presented in the RPC documents which involves word ***get*** (getInvoices(), getPurchaseOrders(), etc.) can be mapped to HTTP verb GET, for word create (createPO(), createInvoice()) – POST and so on.

6. Bibliography

[1] B. Kitchenham - "Procedures for performing systematic reviews," *Keele, UK, Keele Univ.*, vol. 33, no. TR/SE-0401, p. 28, 2004.

[2] M. Maleshkova, C. Pedrinaci, J. Domingue - "Investigating Web APIs on the World Wide Web" *In Proceedings of the 8th IEEE European Conference on Web Services (ECOWS 2010)*, *1-3 December 2010, Ayia Napa, Cyprus. IEEE Computer Society*, 2010, pp. 107-114.

[3] M. Laitkorpi, J. Koskinen, T. Systä - "A UML-based Approach for Abstracting Application Interfaces to REST-like Services" *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy. IEEE Computer Society,* 2006, pp. 134-146.

[4] M. Athanasopoulos, K. Kontogiannis - "Extracting REST resource models from procedure oriented service interfaces", *Journal of Systems and Software 100*, pp. 149-166, 2015.

[5] P.A. Ly, C. Pedrinaci, J. Domingue - "Automated information extraction from web APIs documentation" *In Proceedings of the 13th International Conference on Web Information Systems Engineering (WISE 2012), Paphos, Cyprus, November 28-30, Springer,* 2012, pp. 497-511.

[6] M. Gulden, S. Kugele - "Concept for generating simplified RESTful interfaces", *WWW (Companion Volume) 2013:* pp. 1391-1398.

[7] D. Mukhopadhyay, A. Chougule - "A Survey on Web Service Discovery Approaches", *In Proceedings of the Second International Conference on Computer Science, Engineering and Applications (ICCSEA 2012), May 25-27, 2012, New Delhi, India, Springer,* 2012, pp. 1001-1012.

[8] Web Platform Specs, World Wide Web Consortium, accessed May 11, 2016. <https://specs.webplatform.org/url/webspecs/develop>

[9] Uniform Resource Locators, Internet Engineering Task Force (IETF), accessed May 16, 2016. <https://www.ietf.org/rfc/rfc1738.txt>

[10] Hypertext Transfer Protocol – HTTP/1.1, World Wide Web Consortium (W3C), accessed May 16, 2016. <https://www.w3.org/Protocols/rfc2616/rfc2616.html>

[11] Roy Thomas Fielding - "Architectural Styles and the Design of Network-based Software Architectures", *University Of California*, *Irvine, USA,* 2000.

[12] Tianlei Zhang, Hui Meng, Liping Xiao, Guisheng Chen, Deyi Li - "Web service discovery based on keyword clustering and ontology", *PLA Communication Command Academy, WuHan, 430010, China,* 2008.

[13] G. Salton, J. Michael - "Introduction to modern information" *McGraw Hill, New York,* 1983.

[14] URL - W3C Working Draft, accessed May 18, 2016 <https://www.w3.org/TR/2012/WD-url-20120524>

Non-exclusive license to reproduce thesis

I, Levani Kokhreidze (date of birth: 10th of July 1992),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

    Service Discovery supervised by Marlon Dumas and Vladimir Šor.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 19.05.2016