Service Discovery in Hybrid Environments

Sabrina Alam Chowdhury

Department of Informatics

Faculty of mathematics and natural sciences

UNIVERSITETET OF OSLO

01/08/2017

Service Discovery in Hybrid Environments

© 2017 Sabrina Alam Chowdhury

Service Discovery in Hybrid Environments

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The thesis topic is based on Service discovery of heterogeneous Web services across hybrid environments. Here it also describes a clear definition of SOA and Web service with different standards to implement those services in different environments. Furthermore an analysis and survey of Web services standards also given in this thesis. An overview also discussed here that how different Web service discovery mechanism solution is currently available to discover services in different environments which include from cloud to non-cloud , non-cloud to cloud and other platforms with some research challenges on service discovery for SOAP and RESTful Web services.

A prototype has been implemented as a proof of concept for enabling common service discovery for hybrid environments for different Web services.

Preface

The thesis represents the final product of my master degree in Informatics at the University of Oslo. The work described herein is conducted under the supervision of Dr. Frank Trethan Johnsen and Cand. Scient. Trude Hafsøe Bloebaum.

The thesis has been a long journey, and I would not have been able to complete it without the precious help and support given by various people. The learning curve of my career becomes so high, and I got to understand lots of interesting things while working on this thesis. I believe the knowledge will help me a lot in my future professional life.

First and foremost I would like to express my gratitude to my supervisors Frank and Trude for their continuous support, feedback, dedication and proofreading.

I would also like to thank my family and friends for their support which gives me the motivation to do this master's program. Specially my mother who always keep her trust on my ability. Finally, special thanks to my husband Nazrul Islam Sujan and my son Izaan for their cooperation during the whole journey and for supporting me to keep a balance between my study and family life.

Sabrina Alam Chowdhury

Kjeller, August 2017

Contents

Service Discovery in Hybrid Environments 3
Abstract 5
Preface 7
Contents 9
1 Introduction 11
1.1 Central Terminology 12
1.2 Scope and Problem Statement 14
1.3 Research Questions 15
1.4 Research Methodology 16
1.5 Outline of this thesis 17
2 Technical Background 18
2.1 Service Oriented Architecture 18
2.2 Web service Definition 20
2.3 Cloud Environment 24
2.4 Service Discovery Definition 29
2.5 Related Work 37
2.6 Requirement Specification 49
3 Design 53
3.1 General Design 54
3.2 Workflow model 57
4 Implementation & Evaluation 62
4.1 Implementation 62
4.2 Evaluation 68
5 Conclusion 77
Bibliography 79
List of Tables 87
List of Figures 88
Glossary 89
Appendix A 91
Appendix B 97
Appendix C 103

1 Introduction

Today, millions of users from all over the world are connected through the Internet. On the World Wide Web, information sharing is easy due to availability. Web services are one of the ways to make information available. A Web service is a framework for a conversation between two computers; these computers are communicating over the network. Service refers to any kind of service, like hardware or software, which can give support e.g. the printing of papers or the booking of air tickets. Web services are bound with the software concept. Web services are popular for some specific features like interoperability, reusability, loose coupling and easy deploy ability and integration, just like web applications. In software engineering, SOA, which means Service Oriented Architecture, is an architectural concept and refers to a combination of services.

Web services are the preferred standard to achieve SOA. The concept of SOA is modeled with a Service Provider, a Service Consumer and a Registry along with some operations like register, find and bind. Service Discovery means finding services when required according to service functionality. Because of increasing use of services over the Web, Service Discovery has recently become a relevant research topic. Service Discovery is the mechanism which enables devices and services to properly discover, configure and communicate with each other over the web. Web services need to be able to be deployed in some environments where they can be functional and discoverable as well. Platforms on which Web services operate can either be servers with local configurations or any cloud environment.

The focus of this thesis is the concept of SOA and Web services, as well as the discoveries of Web services on various platforms.

1.1 Central Terminology

This section defines some central concepts and terms, like “SOA” (Service Oriented Architecture), “Web service,” “Service Discovery,” Hybrid Environments,” which is necessary for understanding the rest of this thesis. Details of the technological concepts will be described in chapter 2.

SOA can define the architecture which uses services. There are many different definitions of SOA, but all agree that SOA is a paradigm for improved flexibility.

According to OASIS [1], *Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.*

Visibility, Interaction, and Affects, three primary key concepts in SOA, are described as able to see the possibilities or capabilities and able to use those capabilities to get the result as real world effect.

According to OASIS [1], *the noun Service is defined in dictionaries as “The performance of work or function by one for another.”*

However, service, as the term is understood, also combines the following related ideas: The capability to work together, the specification of the work offered to another and the offer to perform work for another. In SOA, services are the mechanism by which needs and capabilities are brought together.

Technically, a service is an interface for multiple messages that are exchanged between providers and consumers.

Web services are the standard mechanism to implement an SOA concept. There are two technologies which define a set of rules while designing a Web service

- SOAP (Simple Object Access Protocol) [2]
- REST (Representational State Transfer) [3]

SOAP is XML-based; data is formatted in XML so performance issues could arise if the number of messages is high, but it provides a more secure information exchange due to its signing and encrypting of messages by Web service.

REST is a lightweight alternative that uses HTTP but can handle several data formats like XML and JSON.

SOAP and REST are almost the same, but in the current micro-services era, REST is becoming the more commonly used approach due to its lightweight behavior. The micro-service architecture is an approach to developing an application as a set of small independent services. According to [4], A *Micro-service is a cohesive, independent process interacting via messages. Micro service architecture is a distributed application where all its modules are micro-services.*

According to James Philips [5], Service discovery is a system whose aim it is to find the answer to the question “where is the service located.” chapter 2 has a more detailed definition of Service discovery by SOA.

Service Discovery is an important aspect of the Web service lifecycle. Web services need to be discovered and utilized according to functionality. As the growth of Web services increases rapidly, a problem arises when identifying and selecting appropriate Web services, because of the massive number of Web services available over the Internet. Another obstacle that obstructs finding the proper Web service is a lack of suitable search mechanisms, as most of the search tools are based on syntax rather than semantics. Additionally, existing search tools may fail to involve non-functional parameters such as Quality of Service and Cost of Service.

Service Discovery protocols are designed to help programmers and designers to simplify their design work, as it is not necessary to know all possible interactions between devices and Services at design time. They are also designed to minimize administrative overhead.

Web services need to talk to each other and be invoked from a hosted environment. Services can be located in a Local Area Network (LAN), in a single data center, across servers or as federations of the server in different geographical locations. Sometimes the platform can be a cloud environment with another geographic location.

In distributed computing, the word Cloud is exploited as an allegory for “the Internet,” so that means “distributed computing” signifies “a sort of Internet-based computing.”

This work is motivated both by the need to know the frameworks and tools designed for Web service Discovery and the opportunity to exploit the research done in the past on this vast growing area of Web service Discovery, as Service Discovery is the most important part of a Web service’s lifecycle.

The increasing significance of SOA and the popularity of Web services have attracted the attention of researchers and practitioners. There are some open source solutions currently being developed in the domain of Web service discovery. Some of these solutions are designed for a particular project purpose, while others are drawn up with a general purpose in mind.

This work will focus on analyzing the current state of the topic of SOA and the discovery of Web services in the research. This work also aims to achieve a common platform for multiple open source discovery tools for various Web services. Finally, this thesis discusses contemporary and new concepts that have grabbed the attention of most practitioners, developers and the research community within the field of Service Discovery. It is an opportunity to get acquainted with the past developed and currently available market solutions for Web service Discovery for both local and cloud platforms, in addition to helping to develop a vast knowledge of this field.

1.2 Scope and Problem Statement

During the past decade, researchers, practitioners, and academic communities have been proposing different systems, methods, and approaches for the fast-evolving research area of Web service discovery systems.

This thesis focuses on SOA and the Service Discovery of different Web services across hybrid environments. Both SOAP and REST Web service technologies are considered and analyzed. The analysis and survey of Web service Discovery open source standards will be brought into focus by examining several case studies and innovative solutions. Services are implemented and deployed on different platforms which include both cloud and non-cloud environments.

The aim of the thesis is to enable Service Discovery in hybrid environments platforms, using standards where possible.

1.3 Research Questions

Research questions are a description of methods for developing software, analyzing software, the design, evaluation or implementation of specific systems, sheer feasibility of a task [6].

Research questions make the goal of the thesis clear, precise and structured. The research questions for this study are selected based on the conducted preliminary research. Taking into account the background and the popularity of Web services and their rapid technological growth, SOA and Service Discovery were selected as a focus of this research.

The research questions for this study are as follows:

Question 1: What are the Services and types of Services?

Question 2: How do Service Discovery standards for different kinds of Web services work?

Question 3: How do Service Discovery methods work for different environments?

Question 4: Is it possible to create a common platform to provide support for Service Discovery in hybrid environments?

These research questions can be categorized [6] as Descriptive, Method for Analysis and Feasibility, as illustrated in table 1.

RQ	Type of RQ
Ques 1	Descriptive
Ques 2	Method for Analysis
Ques 3	Method for Analysis
Ques 4	Feasibility

Table 1: Type of Research questions

1.4 Research Methodology

Research Questions guide a study's design and data collection methods. According to Stephen Denning's design approach [7], the process is divided into following steps:

Phase 1: Preparing the requirement analysis

Phase 2: Drive a specification based on requirements

Phase 3: Develop and implement the system

Phase 4: Validate and test the system

Analyzing the needs is the first phase of the process. Focusing on the research questions, 1 to 3, case studies have been chosen to examine the artifacts related to this research.

According to [8]*, case study research is defined as an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not evident.*

The second chapter of this thesis derives all the connected background components and ground technologies. Case study research can be divided into single and multiple case studies [9]. The former involves research that examines a single case, while the later analyzes several cases. Case study research can also be categorized as analysis-holistic (a single unit of analysis) or embedded (multiple units of analysis). According to the definition, the case study is incorporated with Service Discovery where several factors would be analyzed in this thesis for getting better insights into the technology. Also in chapter 2, comparative overviews of several open standards of Web service Discovery are analyzed to get a better overview of the technology.

The second step is covered in the third chapter, which presents the specifications, describes the plan of the design and draws its components detail.

The third phase involves the design and development of the system to answer research question 4, as stated in Section 1.3 above. Chapter 4, the implementation chapter, includes the development of the scheme which has been followed in an agile way. Implementation work has been done in short sprint. The fourth phase is covered by chapter 4 in the evaluation

section. The evaluation was also done between sprints to track progress which helps to meet the time restriction while avoiding being led away from the goal. Also, it describes the testing of the system, including both unit testing and composing testing, which was done to determine whether the system meets the requirements or not.

1.5 Outline of this thesis

The thesis consists of five chapters; each chapter addresses different parts of the process from the very beginning to the finished project. The first chapter provides the background and motivation of the thesis, including the research questions and methodology followed in the development of this project.

The second chapter is the analysis of the technological background. It presents the detailed work summary and requirement specification list for this thesis.

The third chapter represents the overall design of the thesis. Each component and module with work flow will be discussed in the design chapter.

The fourth chapter addresses the implementation and evaluation of the application. A more in-depth design is presented through the introduction of use cases. The chapter also derives the testing of the project with several use cases and presents an evaluation reached by analyzing and comparing the outcome of the testing.

The final chapter is the conclusion of this thesis, which summarizes the overall thesis and implementation and shortcomings and findings of this thesis.

2 Technical Background

This chapter will describe the technical details which are the background for this thesis topic. The research questions provide the outcomes for some technical concepts in this chapter. The main umbrella terms are SOA, Web services and the environments used to deploy the Web services.

2.1 Service Oriented Architecture

SOA has recently become the most popular concept of business integration in the IT industry. SOA is an architecture approach for characterizing, connecting, and coordinating reusable business benefits that have clear limits and which have independent functionalities. Inside this sort of design, one can coordinate the business benefits in business forms. Embracing the idea of administrations (a larger amount reflection that is autonomous of utilization or framework, IT stage, setting or different departments), SOA takes IT to another level, one that is more suited for interoperability and various situations. SOA abstracts services from their realization using the concept of interface, which describes how the interaction between parties will occur.

Web services are a fundamental and common way to implement SOA. The architecture for service-based applications shown in figure 1 has three main parts: a Service Provider, a Service Consumer, and a Service Registry.

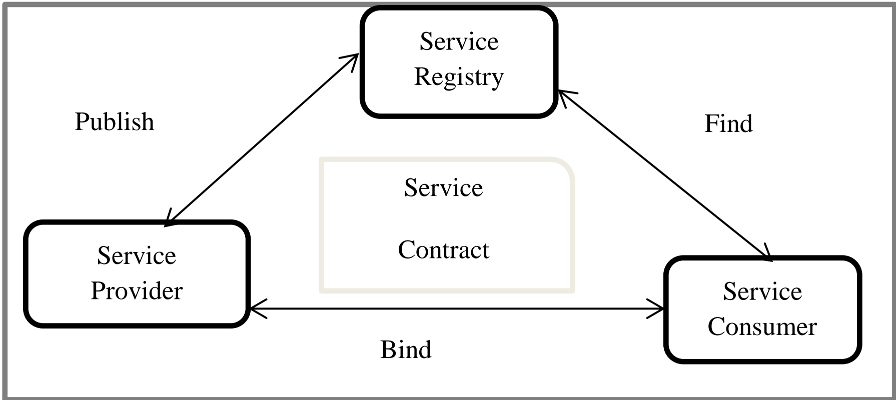


Figure 1: SOA Architecture

Service Provider creates Web service and provides its information to the Service Registry, which is called publishing the service.

Service Registry: responsible for making the information available regarding the Web service to any potential requester.

Service Consumer finds services registered in the registry according to needs and then binds with the Service Provider to invoke one of its Web services.

The Service Contract is the description of the Web service which is the main component of interconnection and binding between the Service Provider and the Service Consumer.

According to [10], SOA implementation usually depends on several facilities including service registries where the services are advertised, service repositories, service definition languages which developers use to define service contracts and service platforms which provide design time and run time support for service creation, deployment, and execution.

High interoperability describes the goal as being able to connect heterogeneous systems easily. Interoperability is not a new concept within enterprise application integration (EAI); EAI had the idea before SOA.

SOA is becoming essential for solving the IT/Business gap. It is an approach that helps systems remains scalable and flexible while growing, which contributes to bridging gaps. SOA acknowledges that the best way to keep up adaptability in massive disseminated frameworks is to help heterogeneity, decentralization, and adaptation to non-critical failure. The key to fulfilling those goals is loose coupling, which means minimizing dependencies. When dependencies reduce, modifications have decreased effects, and the systems will still run when parts of it are broken or down.

According to [11], to establish SOA successfully, it has to introduce concepts appropriately. Key success factors are understanding, governance, management support and homework. Also important are the ingredients of SOA infrastructure, architecture, and processes including meta processes and management.

Infrastructure is the technical piece of SOA which empowers high interoperability in addition to being in charge of keeping up information change, smart directing, managing security and consistent quality, benefits administration, checking and logging. In this thesis,

the infrastructure will also address hybrid environments, which is a Cloud Environment and a Non-Cloud Environment.

Architecture is also necessary to restrict all the possibilities of SOA in such a way that it leads to a working, maintainable system.

Larger systems' complexity is that many different peoples and teams are involved in maintaining these. To control everything, appropriate processes need to be implemented to achieve SOA such as:

Business Process Modelling (BPM) which is the process of breaking processes into smaller units, which are services.

Service Life Cycles involve defining different steps a service takes to become a reality.

Model-driven Software Development (MDSD) is the process of generating code for dealing with services.

Governance is the Meta process of all processes and SOA strategy, whose aim it is to set up the right process to establish SOA in the organization.

2.2 Web service Definition

Web service

Web services are server and client applications that can talk over the World Wide Web [12]. They can be used to implement the SOA principles, and are in widespread use in various sectors. Web services present the answer and realization of the SOA question of seeking the need for interoperability between systems and platforms which helped SOA get up and running quickly. A Web service is a framework for conversation between two computers; these computers are communicating over the internet. Clients send a request to the internet and server receives the request and process it and return the response. When a browser makes a request for a web page, it receives HTML or other related content in the response. But when it just ask for data and use JavaScript or other client side code to process the response then the Web service is used.

It is clear that Web services represent the cornerstone of SOA and its recommended technology for interoperability.

Web services are the foundation of SOA because of them:

- implement gauges and, accordingly, advance similarity and movability
- are cross-stage and cross-dialect supported
- are widely supported, making SOA moderately simple to embrace
- are message-oriented
- give quicker tooling help, which speeds the usage of SOA

Microsoft coined the term Web services in 2000 as "A set of standards which allow the machine to machine communication by a network which supplies a particular set of operations [11]". More precisely, Web services can exchange information via many internet protocols but mostly with *Hypertext Transfer Protocol (HTTP)* – the key communication protocol of the World Wide Web. According to [13], Web services evolved from web applications, which only serve data instead of serving the user-interface along with the data. How to present the information depends on the client application. Key benefits of Web services along with SOA services is that they can be invoked by several consumers, which can also result in the building of more flexible applications.

To exchange information when we design Web services, we need to follow some set of rules. The primary tools for doing that these days are SOAP-based Web services and RESTful Web services [14].

SOAP Version 1.2 is a lightweight version proposed for trading organized data in a decentralized, circulated location. It utilizes XML to characterize an extensible messaging system giving a message format that can be exchanged over a diversity of core protocol. The structure has been intended to be autonomous of a specific programming model and other implementation particular semantics [2].

The protocol specification defines an XML based envelope for exchanging messages, a set of rules specified by the protocol for converting platform specific data types into XML representation.

A SOAP message has three building blocks: an envelope, a header, and a body. The envelope wraps the entire message and contains the header and the body elements. The Header elements include the Security and Routing information, which is optional. The data which are application specific and need to be communicated belong to the body of the SOAP message.

The application specific data is SOAP marked-up as XML and adheres to a particular format, which is defined by the XML schemas, and this formatting enables the recipient to process the data correctly. SOAP messages are received and interpreted by SOAP servers which in turn trigger Web services to perform their tasks [15].

REST is defined by Fielding in [3] as an architectural style that consists of some set of design criteria or set of design principles known as REST constraints that represent the easy way of web standards such as HTTP and URIs. REST was initially identified in the context of the Web; it is becoming a mostly used implementing technology for developing Web services. REST principles include addressability, uniformity, connectivity, and statelessness [10]. The resource is the central artifact in RESTful services. This kind of service implements with web standards and REST principles. RESTful Web services should have appropriate resources naming how servers dispatch requests to resource implementations. A specific URI represents every resource. Resources are by nature self-descriptive messages.

RESTful HTTP uses the four primary HTTP methods: GET POST, PUT and DELETE. These methods are used to read, write or create/perform and delete resources identified by URLs. Because this native usage of the HTTP protocol is straightforward and fast, this can be a good way to provide access to data or resources provided by web servers. RESTful Web services provide scalability which comes from natural support caching and partitioning on URIs. Restful Web services are also accessible as URIs are shared and passed for common purpose application reuse to any dedicated servers. Compared to the ad-hoc partitioning of functionalities behind the SOAP interfaces, URI based partitioning is more generic, flexible and could be easier to realize [16].

Web services are implemented by both SOAP and RESTful Web services. The selection from those two depends on several factors; each has different distinct features and shortcomings. Table 2 presents the comparison between SOAP and REST based Web services according to [17].

Criteria	SOAP based WS	RESTful based WS
Server/Client	Tightly coupled	Loosely coupled
URI	One URI representing the service end points	URI for each Resource
Transport Layer Support	All	Only HTTP
Caching	Not Supported	Supported
Interface	Non uniform Interface	Uniform Interface
Contact aware	Client Context aware of WS behavior	Implicit Web service behavior
Method information	Body Entity of HTTP	HTTP method
Data Information	Body Entity of HTTP	HTTP URI
Describing Web services	WSDL	WADL
Expandability	Not Expandable	Scalable without creating WS (usage xlink)
Standards used	SOAP specific standards (WSDL, UDDI, WS security)	Web Standards (URL,HTTP methods, XML, MIME types)
Security /Confidentiality	WS-Security standard specification	HTTP Security

Table 2: Comparison of SOAP and RESTful Web services

From table 2, there are some issues with the specification provided by SOAP and REST in [17]. In [2], SOAP version 1.2 is a lightweight protocol intended for exchanging information in decentralized, distributed environments. WSDL is currently the only one used for describing SOAP Web services. Also, as WADL is outdated, there are more options for RESTful Web services to describe, like the Swagger framework [18]. In the context of the security of RESTful Web services, there are also OAuth, SAML, and OpenID Connect, which provide better API security.

2.3 Cloud Environment

"Cloud Computing," by definition, refers to the on-demand delivery of IT resources and applications via the Internet with pay-as-you-go pricing. A standard definition according to NIST [19] is:

Cloud Computing is a pay-per-use model for enabling available, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Through cloud computing services, it becomes simple to access to servers, storage, databases and a broad set of applications across the web. Cloud computing providers own the infrastructure and the related environment needed for application services.

The concepts of Grid and Cluster Computing, Virtualization, Web service and SOA are brought about by Cloud Technology. Cloud Technology also makes concepts such Utility Computing meaningful and proposes to establish IT free from the complexity and costs of its common physical infrastructure.

Most well-known cloud provider companies are IBM, HP, Google, Microsoft, Amazon Web services, Salesforce.com, NetSuite, and VMware.

Cloud computing has the following benefits:

- Predictable anytime, anywhere access to IT resources
- Flexible scaling of resources (resource optimization)
- Rapid, request-driven provisioning
- Lower total cost of operations

According to NIST [19], a cloud computing model has five key characteristics, three delivery models, and five deployment models.

The five main features are defined as:

On-demand self-service: a client can provision computing environments as needed without any human interaction with the provider.

Ubiquitous network access: Any device like laptop or mobile can access services with standard mechanisms which are available over the network.

Location independent resource pooling: Geographical areas need not be considered while accessing resources. Examples of resources include a virtual machine, storage, memory, processing.

Rapid elasticity: For quick upscaling and downscaling capabilities, can be rapidly and elastically provisioned and released when needed.

Pay per use: Charging depends on how much service is utilized by the consumer. Advertising-based billing model to promote optimization of resource use.

Cloud Computing Model

As cloud computing arises, there is much discussion about defining a cloud computing model. A better way of defining Cloud computing [20] is to create a stack which represents each component of cloud computing and interaction between them. As seen in figure 2 [20], which illustrates the cloud computing model, the elements of the cloud computing model provide a vast range of services which can be consumed over the Web through a pay-per use model. Most services which were previously accessed through a conventional data center can now be used from the cloud.

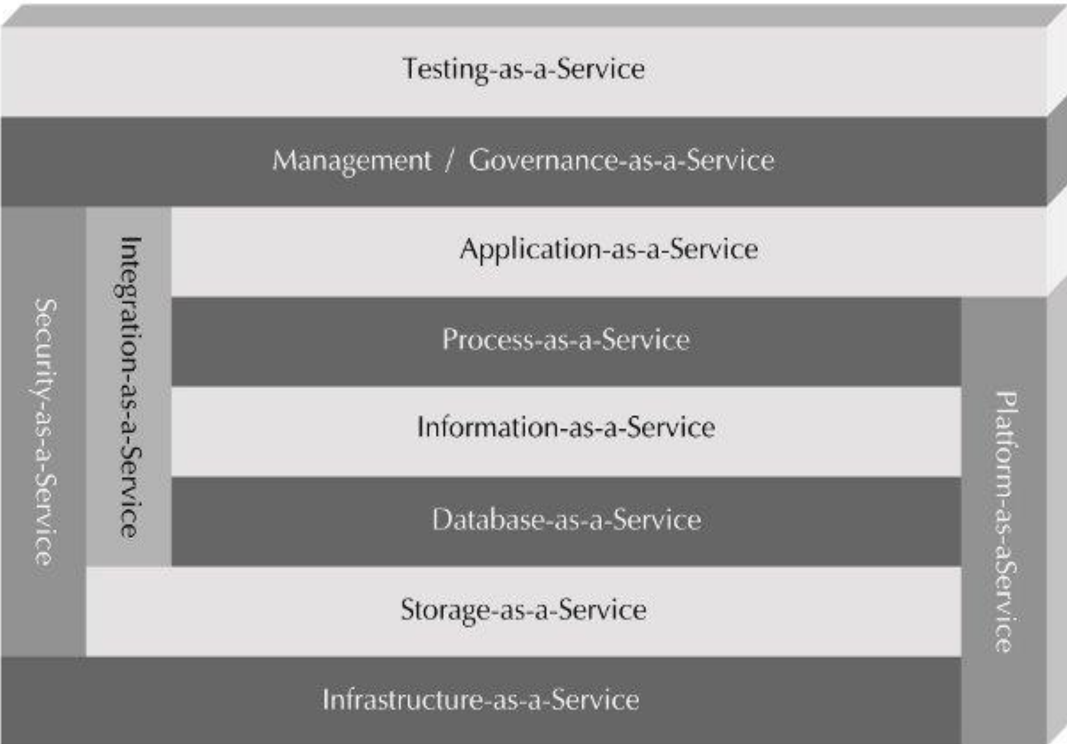


Figure 2: Cloud Computing Components

According to [20], there are eleven categories or patterns of cloud computing technology:

1. Storage as a Service (SAAS): Usually defined as on-demand space. Any application can use a local storage resource which physically exists at the remote site. SAAS is a very core component type of model which can also invoke other cloud computing models.

2. Database as a Service: Delivers the ability to use a remotely hosted database which can be shared by another user, functioning as a locally hosted database. This cloud computing service model provides user access to the database without installing software or hardware set up for performance.

3. Information as a Service: Provides users the ability to consume any remotely hosted information via a well-defined interface such as an application programming interface (API). For example, weather information, stock price information, and phone number validation.

4. A Process as a Service: A cloud computing model which outsources the business process to consumers.

5. Application as a Service: Any application which is offered over the Web to the end user which can be consumed by the browser application. Some examples include Google Docs, Google Calendar, and Gmail.

6. Platform as a Service: Any complete platform remotely hosted which includes application development, interface development, database development, storage, and testing, delivered to subscribers. Modern PAAS providers offer the creation of enterprise class applications for use on demand with a small subscription or free.

7. Integration as a Service: This cloud computing model delivers a complete integration stack including interfacing with applications and semantic mediation, flow control, and integration design. Most of the features and functionality of these types of model are also found in traditional enterprise application integration (technology) but is now provided as a service.

8. Security as a Service: Provides core security services remotely over the Internet, for example, identity management.

9. Management/Governance as a Service: Provides the ability to manage one or more cloud services, including topology, resource utilization, virtualization, and uptime management. Management systems, such as the capacity to enforce defined policies on data and services, are also available.

10. Testing as a Service: Refers to the ability to use testing software and services that are remotely hosted to test local or cloud-delivered systems. In these types of cloud computing, the model provides the service to test enterprise applications, websites, or any other cloud components without knowing anything about the hardware and software within the company.

11. Infrastructure as a Service: In this cloud computing model, the consumer can have access to the entire machine with software on that machine hosted remotely. Provides the

ability to lease a physical server which for all practical purposes acts similar to a local data center or a part of a data center which is the capacity to access computing resources remotely, or Datacenter as a Service (DaaS).

Cloud Computing Deployment Models

Public: A model of cloud computing resources such as storage or applications which are fully offered over the Internet to the general public. In this example, services may be used either for free or through a pay-per-usage model.

Private: A private cloud is a particular model of cloud computing in which a specified client can operate a distinct and secure cloud based environment. Similarly to other cloud patterns, a private cloud also offers computing power as a service within a virtualized platform using an underlying pool of physical computing resources. In this type of model, resource pooling is dedicated to a single organization, providing that organization greater control and privacy.

Community: The infrastructure is a platform which allows several groups have similar needs and concerns to work on the same platform. The community cloud can be either on premises or off premises and can be managed by a third party service provider and governed by dedicated groups or organizations.

Hybrid: A hybrid deployment model refers to interconnected infrastructure and applications that are hosted both outside the cloud and inside the cloud. The most common occurrence of this model is when the organization's cloud services interact with their internal system. This model also uses a mix of public and on premise private clouds with orchestration between platforms.

Table 3 shows the cloud computing deployment model according to [19]

Private	**Community**	**Hybrid**	**Public**
The Cloud Infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise.	The Cloud Infrastructure is shared by several organizations and supports a specific community that has shared concerns. (e.g., mission, security requirements, policy and compliance considerations).	The Cloud Infrastructure is a composition of two or more clouds (private, community or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.	The Cloud Infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

Table 3: According to NIST definition Cloud deployment models

2.4 Service Discovery Definition

The mechanism which drives the finding of Web services is called Service Discovery. In SOA, it is a key component and important aspect which leads Web services to utilize their functions. In SOA implementation, the primary factor is a higher degree of reusing roles in the form of readily implemented services, and the aim is to minimize development time and costs.

Recalling the SOA and Web services definition as shown in figure 1, three primary roles are interacting with each other within SOA architecture. These three main roles are the Service Provider, Service Requester and Service Registry. The roles interact using publish, find and bind operations. The service providers are the business process that provides access to the Web service and publishes the service description for consumption. The service description usually uses to bind with the information. The Service Requester also uses the Meta information in a description to attach and consume a service. According to [5], there are four typical components of Service Discovery: where is the service? What are the IP and port? How do I connect? And a health monitoring piece to detect the functional server. The Service Registry is an optional logical concept where the Service Discovery method is to locate information about the Service Provider and obtain the service details.

Service Discovery provides the functionality to discover capabilities of services automatically. Usually, a Service Discovery system can help services to register their obtainability, locate a single instance of a particular service and also notify when an instance of a service changes.

Components of Web services Discovery

Service Registry

In Service Discovery, the Service Registry is a key component which functions as a kind of storage of information about the network location or service instances. That can work as a database of services and needs to be highly available and up to date. A Service Registry can be within a cluster of servers that use a replication protocol to obtain consistency.

Service Discovery Mechanism

The objective of the proposal is to analyze the Service Discovery mechanisms for different Web service standards by various platforms. There exist numerous different kinds of Web service Discovery protocols; they are responsible for connecting machine to machine to achieve the purpose of a Web service from the service providers to the service requester. With the rising number of Web services and also to fulfill the requirements of scalability, high availability and maintainability of services, Service Discovery techniques and pattern are also changing rapidly. There are several standards which are involved in Service Discovery, namely UDDI, WS-Discovery, and ebXML. Much more are also available for micro-service architectures. The most common protocols are:

UDDI – Universal Description Discovery and Integration – A standard for Web service registry

WS- Discovery –A standard for mainly local Web services discovery

ebXML – Electronic Business Using Extensible Markup Language, which is also registry based.

Universal Description, Discovery, and Integration (UDDI) [21]

UDDI is a protocol which provides the mechanism to register and locate Web services. This protocol is an approved OASIS Standard and a key member of Web Service Protocol Stack [22], which is a platform-independent XML-based registry. With this feature, businesses worldwide can list themselves on the Internet.

The functional purpose of a UDDI's registry is the presentation of data and Meta data about Web services. This registry can be used either on a public network or within an internal organizational infrastructure. UDDI registry offers a standard way to manage, classify and catalog Web services so that Web services can be discoverable and invokable by other applications. UDDI specifies protocols for access, control, and management of the registry for Web services. This standard offers a way to locate a Web service, invoke that service and manage Meta data about that service.

UDDI provides a registry of Web services and programmatic interfaces to publish retrieve and manage data about Web services. In the context of SOA, UDDI itself is a set of Web services. UDDI is mainly building upon several other established industry standards includes HTTP, XML, XML Schema, SOAP, and WSDL.

UDDI uses UBR, which is the UDDI business registry. UBR can be categorized as Yellow Pages, White Pages, and Green Pages. The White Pages contains contact and general information whereas the Yellow Pages group and divide information into categories, and the Green Pages include the technical information like Web service details.

UDDI was proposed in August 2000, but in later years UDDI has lost some of its popularity to other discovery mechanisms. The work was completed and closed late 2007, and there is no longer anyone responsible for maintaining the UDDI registry. However, today the UDDI system is preferred mostly inside companies due to support for design time discovery. One of the drawbacks of UDDI is the centralized repository mechanism, which can affect availability and scalability.

Web services Dynamic Discovery (WS-Discovery)

Web services Dynamic Discovery (WS-Discovery) is an OASIS [23] Service Discovery specification. WS-Discovery characterizes a multicast approach to find Web services over a local network. As a matter of course, probes are sent to a multicast group, and target services that match return a response straightforwardly to the requestor. The protocol characterizes the multicast suppression behavior if a discovery proxy is accessible on the system to scale to a large number of endpoints. To limit the requirement for surveying, target services that desire to be discovered send a declaration when they join and leave the network.

According to OASIS [23], WS-Discovery defines two modes of operation, an ad-hoc mode, and a managed mode. Discovery proxy is an optional feature of the ad-hoc mode but a necessary feature of the managed mode. The reason for implementing a discovery proxy is to increase the scalability of the system and to increase the reach of the services beyond the local or ad-hoc network.

Electronic Business Extensible Markup Language (ebXML) [24]

Electronic Business Extensible Markup Language known as ebXML or e-business XML provides a technical framework through which companies can communicate and exchange data via the internet. A business-to-business XML based framework mostly which has a specific set of specifications for enabling modular frameworks. It is a registry-based solution to store information about available Web services on a network and provide Web service consumers in the network with information about these services. SOAP, WSDL, and UDDI alone were not sufficient to deal with business services interaction. Because WSDL does not address business collaboration and the UDDI repository lacks support for business objects. ebXML can address the needs of business processes and their involved parties and roles. ebXML also supports security, reliability, and quality of service requirements and exchanging XML business collaboration documents.

EbXML defines its registry structure through which service consumers can access XML documents that contain information about service providers. These standards build upon the existing standards such as HTTP, TCP/IP, MIME, SMTP, FTP, UML, and XML.

In SOA, the set of running service instances changes dynamically within the micro services application. In micro service application, if a client wants to make a request to a service, it

must use a Service Discovery mechanism as instances dynamically assign network locations. Depending on the infrastructure and also for micro-service architectures, the mode of operation and purpose Service Discovery can also be categorized in several ways, such as design time and run time, static vs. dynamic, centralized vs. distributed infrastructure, and client side vs. server side discovery patterns.

Static discovery: mostly done in the design time can be done only once by maintaining a fully static configuration which can occasionally be updated.

Dynamic discovery: can be a system or software which can dynamically identify and select services during the operation. To maintain consistency and avoid service interruption, dynamic service registration and discovery become much more important.

According to James Philips [5], static Service Discovery is okay when it is a small application, but also response time will be affected. For the medium to large and large to huge infrastructures, it becomes hard to maintain and respond to time problems using static Service Discovery.

When it comes to run time and design time, Service Discovery approach there is not like that type of thinking, it is most proper phrased as machine oriented or human driven.

Run time discovery usually means that software that is running has some configuration to get the IP address of remote services by sending out probes on the network. Use cases in need of run time discovery such as Operator-driven Integration, Moving Target Defense for a cyber-attack or administrative setup or recovery. Run time discovery can be made in the context of software that has been installed on a machine. The mechanism of Runtime Service Discovery runs the scope from sophisticated and automatic service inquiries across the network to asking some “central repository” by poking the system administrator and asking the information again about the IP address.

Machine Oriented

The mechanism is to get the IP address or any other configuration information and plug that information into some running software.

Design time means where the service is already known and configured during the design time. Usually, the developer, when searching for services or assets, includes them while developing

the application. During application development, design-time Service Discovery is better described as locating and consuming professional profiles and service interface specifications. In this method, developers go through to find technical documentation which allows them to write the software they are working on. Discovering means something searches the internet or intranet, finding a resource to use accordingly.

There are also two types of Service Discovery patterns: client-side discovery and server-side discovery [25]. Centralized and distributed infrastructure is the basis for using a service registry which can also be covered by client-side and server-side discovery pattern for Service Discovery mechanisms.

The Client-Side Discovery Pattern

In client side discovery patterns [25], the client is responsible for obtaining the network location of available service instances and manages load balance across them. The client requires a service registry which is a database of available services where queries can make. A load balancing algorithm is used to select one of the available service instances and to make a request then.

This pattern has several benefits and drawbacks. The pattern is direct, and there are no moving parts except for the service registry. Also, the client is aware of the available services which can help to make intelligent, application-specific load balancing decisions such as hashing consistently. The client is directly connected with the service registry which is one of the drawbacks of this service discovery pattern. We must implement client-side Service Discovery logic for each programming language and framework used by service clients. An example of client-side discovery pattern is Netflix OSS, whereas Netflix Eureka is a service registry. It usually provides API for managing registration and querying instances. To load balance requests across the available service instances, Netflix Ribbon, which is an IPC client, works with Netflix Eureka.

The Server-Side Discovery Pattern

In a server-side discovery pattern [25], the client requests a service through the load balancer. The load balancer then queries the Service Registry and routes each request to an available instance. In the client side discovery pattern, registration and deregistration of service instances involved with the service registry are noted. One benefit of this kind of pattern is that the client does not need to know about the details of the discovery. A simple request can be made by the client to the load balancer. This also reduces the overhead of implementation of discovery logic for each programming language and framework used by service clients. The drawback of Server-side discovery patterns involves setting up and managing highly available system components if the load balancer is not provided by the deployment environment.

The Amazon Web services (AWS) Elastic Load Balancer (ELB) is an example of a server-side discovery router. A client can make requests which can be HTTP or TCP via the ELB using a DNS name. An ELB is mostly used to load balance external traffic from the Internet. Also, it can be used to load internal balance traffic to a virtual private cloud (VPC). An ELB can load the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. No separate registry is used. EC2 and ECS can be registered to ELB.

As a server-side discovery load balancer, HTTP servers and load balancers such as NGINX and NGINX Plus can be used. A more recent execution could progressively reconfigure NGINX Plus utilizing HTTP API.

Some deployment environments such as Kubernetes run a proxy on each host in the cluster. A client directs the request through the proxy to make a request to a service. The proxy uses the hosts IP address and port information. The proxy plays the role of server-side discovery load balancer which forwards the requests to an available instance running somewhere in the cluster.

In [26], a survey of Web service Discovery mechanisms draws upon the analysis of several approaches and methods to discover Web services. It is as follows in table 4:

Approach	Proposed by	Advantages	Disadvantages
Context-Aware Web service Discovery	Wenge rong and Kecheng Liu	Optimizes request, result, and personal profile. The method is better than traditional keyword-based methods	It is hard to model context for all the applications
Publisher Subscribe Model	Falak Nawz et al.	Minimum time for Web service Discovery	Adding overhead to develop and maintain new components
Service Request Expansion	A.Paliwal et al	Combinational approach of ontology and Latent Semantic Matching which makes method more accurate	Computation cost of Latent Semantic Index is high
BPEL Process Ranking	D. Grigori et al.	If exact Web service is not found, approximate Web service can be provided to the user	It is purely based on syntactic matching and semantics of user request is not considered
Web service Indexing	B. Zhou et al.	Since index are used, it is fast and easy to retrieve objects	Indexing process is expensive, and it needs additional space
Structural Case-based Reasoning	Georgios Meditskos and Nick Bassiliades	Retrieval of Web services using structural information of OWL ontologies	Semantic Case Based Reasoning (SCBR) measure makes this method computationally expensive
Agent-based Discovery using QoS	T. Rajendran and P. Balasubramanie	Separate agent is used to ranking the Web services which makes method fast	Business specific and performance specific QoS for each Web service needs to be supplied
Collaborative Tagging System	U. Chukmol et al.	Labels associated with each Web service is used which results in efficient Web service Discovery	Porter Stemming algorithm to extract term vector is used which is computationally expensive

Table 4: Provides Comparison on Web service Discovery techniques [26]

2.5 Related Work

To achieve success in the current project, it is crucial to have a thorough knowledge of similar work. Researching and investigating related work can save much time by learning and reusing knowledge and work, rather than reinventing the wheel. This section discusses related work, attempting to draw parallels between this works to the current thesis. In this section, some open source Service Discovery standards will also be discussed to get an overview of the central concepts of this thesis.

There are a case study and report called "Pervasive Web services and Invocation in Military Networks" [27]. This report presents thorough analysis and implementation of different Web service optimization techniques as well as a review of most common Web service Discovery standards and how they perform in military settings. In this report, several challenges related to Service Discovery in dynamic environments, such as military tactical systems, have been addressed. One of the issues is the use of registries in low bandwidth networks and mobile environments. This report also presents a hybrid solution of both using registries and not using registries in a client–service model as a fallback.

In chapter 6, a detailed overview of pervasive Service Discovery is presented, with the answer on how to solve it. This report also proposes suggestions for using different discovery mechanisms per level by considering the differences between the operational levels in the military network.

The three primary approaches to achieving pervasive discovery across multiple domains are Adaptive Service Discovery, Layered Service Discovery and Service Discovery Gateways.

Adaptive Service Discovery is using one Web service Discovery between and in all domains. All applications must be able to interact with the same protocol. The protocol has to be compromised for each particular network, to ensure optimal function. The capacity of data should be minimized while using a small capacity system, and a filter is needed for this task.

Layered Service Discovery is where the network can utilize the protocol that best suits the given network, but all networks will have to be connected using an overlapping protocol that receives and pushes data to all attached to the connected protocols.

Service Discovery Gateways can be viewed as an intermediary of the previous two. Each network will utilize the network protocol that is best suited for the network topology and capacity. A gateway setup is responsible for maintaining data passing through based on a network.

After evaluating all three options, comparing several factors like performance, scalability, reliability, and ease of use and implementation, service gateway is more comprising than the other two. A repository has to be used as storage of information when services are transferred between different domains and discovery mechanisms.

Another work related to the thesis is Federated Service Discovery [28]. In this report, a federation mechanism was introduced that could enable two or more different Web service Discovery mechanisms to communicate over a WAN like the Internet. In that project, a repository was implemented which could store information about Web services. The project aimed to focus on SOAP Web services only. In chapter 5 of the report [28], shortcomings and future work scope of that project discussed. In conclusion, it has been described that the project worked as intended and tested accordingly. However, there were some scopes also addressed which can help to make the project work as full-fledged. The scopes are described as follows:

To achieve full Federation, a feature can be added to enable retrieving data from the connected Web service Discovery mechanism.

In addition to the automation of service registration through Service Discovery mechanism, services should automatically appear in the repository.

More Web service Discovery mechanisms should be addressed, not only SOAP Web services.

It used AMQP as the WAN mechanism, but multi-brokered topology setup can be considered for redundancy.

Through the use of report [28] and future work discussion, it is easier to decide what to focus on and address the main issues accordingly in this thesis. This will save a lot of time and improve the project.

In new micro-services based architectures, applications are not deployed as a set of distributed components. In this kind of architecture, it is important to configure and manage the various applications across multiple instances running in multiple containers.

In SOA, SOAP has the previously mentioned UDDI, ebXML, and WS- Discovery standards for discovering and connecting with services via service description. But In RESTFul-based Web services, no specific industry standard for Service Discovery mechanisms exists. Different solutions are prepared and used based on the context of the application and platforms.

There are several open source solutions available for Web service Discovery across both cloud and non-cloud platforms. In this section, some of them will be openly discussed to understand about how they work. Some features considered are consistency, storage, runtime dependencies, and client integration options.

Some strongly consistent projects used as coordination services but also used for service registries as well include Zookeeper, Doozer, and Etcd. These will be discussed later.

Also available are some interesting solutions specifically designed for service registration and discovery. Airbnb's SmartStack, Netflix's Eureka, Bitly's NSQ, Serf, Spotify and DNS and finally SkyDNS are examples of such kind of projects.

Any service registration and discovery solution also have some development and operational aspects to consider:

Monitoring: The consequences of the failing of the service operation if it is unregistered immediately, after a timeout, or by another process. Services are usually required to implement a heartbeat to ensure availability, and efficient service failure needs to handle by the client.

Load Balancing: Load balancing is necessary to ensure achieving scalability of systems and to serve several thousand requests. If there are several instances and the number of requests is too high, then load balancing is the only smart solution to handle the situation.

Integration Style: Language independence is an important aspect of any Service Discovery mechanism. Integration must be able to support most languages.

Runtime Dependencies: Compatibility check of the environment of the requirement of JVM, Ruby or something that is incompatible.

Availability Concern: Not a single point of failure should be present in a Service Discovery mechanism, which is why nowadays most of the system is aimed towards support in a cluster environment.

Zookeeper

Zookeeper [29] is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [30]. It is written in Java, is strongly consistent (CP) and uses the ZooKeeper Atomic Broadcast (Zab) algorithm [31] protocol to coordinate changes across the ensemble (cluster). The protocol manages small updated to the replicas and is also responsible for selecting the leader in the ensemble. It also synchronizes the replicas and administers the updating of transactions to be broadcast when getting better from a crashed state to a valid state.

Zookeeper typically needs to run with three, five or seven members in the cluster. Specific language bindings need to be accessed which typically reside between services and the client applications.

An ephemeral node under a namespace is used when registering a service. A backend service registered with its location information when a client is connected, and ephemeral nodes only exists that time. Any kind of failure or interruption causes the node to be disconnected from the node hierarchy.

Services appear in the Service Discovery by the listing and namespaces. Clients are able to see all registered and available services. Clients also get notifications of new service integrations and unavailable services. Load balancing and failovers need to be handled by the client application.

The Zookeeper API's hard to use, and language bindings might have subtle differences that could cause problems. For JVM based language, the Curator Service Discovery can be utilized. Zookeeper is a consistent pattern system so when a partition occurs, some of the client applications will not be able to register new services or find existing registered services, even those that functioned properly before partitioning.

Doozer

Doozer [32] is a consistent, distributed data store. This is an open source implementation written in GO. Doozer uses Paxos to maintain consensus. This project around only for few years but staged for a while and now 160 forks only have been made. Unfortunately, this makes it difficult to know whether it is suitable for production use.

Doozer needs to run with three, five or seven nodes in the cluster. The client needs to bind with specific language to access the cluster and integration is embedded into client and services.

Doozer does not have any ephemeral nodes like Zookeeper. So, service registration is not as straightforward. Under a path, a service can register itself, but for unavailable services, it won't help to remove automatically.

This issue can be addressed in several ways. An option might be to add a timestamp and heart beating mechanism to the registration process to handle expired entries during the discovery process, or implementing another cleanup process.

In service Discovery of Doozer, all the entries can list under a path like Zookeeper and changes to the path then monitored. During registration, if a heartbeat mechanism and time stamp use, any expired entries during discovery can ignore or delete.

Doozer also a consistent system. When a partition happens, it has same consequences like Zookeeper.

Etcd

Etcd [33] is a highly available, key-value store for shared configuration and Service Discovery. Etcd was inspired by Zookeeper and Doozer. It is written in Go uses Raft [34] for consensus and has an API based on HTTP and JSON.

Etcd also typically runs with three, five or seven nodes in the cluster. Clients must have a language dependent binding or implementation using an HTTP client.

In order to ensure that the key remains available a key TTL along with heart beating mechanism from the service need to use in service registration of Etcd.

If the update of the key's TTL failed by a service, then Etcd expire it. Clients need to handle the connection failure during service unavailability. Clients also need to look for another instance.

In the service discovery mechanism, a directory has the listing of all services and changes are monitored in that directory. As the API is HTTP based, client application keeps a long-polling connection with the cluster.

Since Etcd uses Raft, it should be a strongly consistent system. Raft protocol utilizes a leader, and all client requests are handled by that leader. However, Etcd also supports reads from non-leaders using an undocumented consistent parameter which will improve availability in the read case. During partition fail writes need to handle by the leader.

Airbnb's SmartStack

Airbnb's SmartStack [35] specifically tailored for service registration and discovery. This is written in Ruby and a combination of two custom tools, Nerve [36] and Synapse [37] that influenced HAProxy [38] and Zookeeper to handle service registration and discovery.

The Nerve runs as a separate process alongside the application and is a sidekick style process. Nerve uses for registering services in Zookeeper. A monitoring endpoint is needed to get registration. An end point /health exposed by the application, for HTTP services that Nerve continuously monitors the status. The sidekick model helps to remove the need for a service to interact with Zookeeper. Zookeeper binding night does not exist whereas that makes way to provide support for different languages

Synapse is in charge of service discovery. This is also a sidekick style process, runs as a separate process. A query has been made to get currently registered services in Zookeeper and reconfiguration made with a locally running HAProxy instance. A local haproxy instance always accessed while any clients need to access another service. The local haproxy instance helps to route the request to the available service.

Synapse design helps to not depend on client side load balancing or failover and also it simplifies the service implementation that contributes to being independent of Zookeeper and its language bindings.

Like Zookeeper some registrations and discoveries in SmartStack may fail during a partition. By keeping the snapshot before partition and use it after partition, the system may be able to continue operation during the partition. The feature can improve reliability and availability of the overall system.

Netflix's Eureka

A middle-tier load balancing and discovery service of Netflix. There are also server components as well as a smart-client that can use within application-services. The server and client implementation are written in Java so the use case would be for the services to be any JVM compatible language.

The server specified as Eureka [39] which is the registry of the services. To form a cluster, one Eureka server is recommended to each availability zone in AWS. An asynchronous model followed while the servers replicate their state to each other. That means each instance may have a different picture of services at any given time.

The client component handles registration of the services. Services must incorporate the client into their application code. During runtime, the client registers the services and sends heartbeats to renew its states.

Service discovery from the smart-client retrieves the current registrations from the server and caches the information about services locally. The client can periodically refresh its states and also handle failovers and load-balancing.

Eureka serves availability over strong consistency. Eureka was designed to support during failures that can operate under some different failure modes. During the partition of the cluster, Eureka turned itself to self-preservation state. This feature helps services to be discovered and registered during partition, and when it heals, the members merge their state again.

Bitly's NSQ lookup

NSQ [40] wrote in GO and provided an HTTP based API. NSQ is distributed messaging platform and also real time. This is not a general purpose service registration and discovery

tool. A novel model of service discovery has been implemented in a nsqlookupd agent to find nsqd instances at runtime for clients.

Nsqlookupd [40] is the service registry. The need instances are essential while in an NSQ deployment. The client can interact directly with nsqd instances, but since run, time changes can affect possible clients can also discover available instances by querying nsqlookupd instances.

Each nsqd instance periodically sends heartbeat of its state to each nsqlookupd for service registration. The state contains the address information and any topics or queries those they have. Moreover, for discovery clients can make a query to each nsqlookupd instance and merge the results.

The fact of this model is that the nsqlookupd instances do not know about each other. Clients have the responsibility to merge the state returned from each stand-alone nsqlookupd instance to determine the overall state. Because of each nsqd instance heart beats its state, each nsqlookupd eventually has the same information can contact all available nsqlookupd instances.

NSQ design is weak in consistent but highly tolerable during partition.

Serf

Serf [41] is written in GO which is a decentralized solution for service discovery and orchestration. Serf is designed by a gossip protocol SWIM [42] which was intended to address the unscalability of old style heart-beating protocols. Serf uses SWIM for failover detection, membership and custom event propagation.

A single binary is installed on all hosts and run as an agent where all the nodes are joins and create a cluster. As a client, it can discover the members in the cluster as well. A serf agent is run to joins an existing cluster for registration purpose. The agent also uses custom tags which can identify the role, IP, port. After joining to the cluster, other members will able to see the host and metadata information about that host.

The Serf is run with the member's command for discovery which returns the current members of the cluster. Depending on the member's output, all the hosts for a service based on the tags can be discoverable.

Serf is a quite new project and is changing quickly. It is the only project discussed in this thesis that does not have a central registry architectural style, which makes it unique. Since it uses an asynchronous, gossip-based protocol, it is inherently weakly-consistent yet more fault tolerant and available.

Spotify and DNS

According to Spotify, instead of using a newer, less mature technology a solution was built on top of DNS. Spotify visions DNS as a “distributed, replicated database tailored for read-heavy loads.”

For service discovery, Spotify uses the relatively public SRV record which is more generalized MX records. To define a service name, port, protocol, priority, weight, TTL and target host can be registered in this mechanism. Based on the information client can load balance and discover services if needed. Service registration is a bit complicated and static as Serf manage all zone files under source control. Some various DNS client libraries and custom tools are used for discovery.

They also run DNS caches on their services to minimize load on the root DNS server.

They mention at the end of their post that this model has worked well for them, but they are starting to outgrow it and are investigating Zookeeper to support both static and dynamic registration.

SkyDNS

SkyDNS [43] is a moderately new project that is written in Go, uses RAFT [34] for consensus and also provides a client API over HTTP and DNS. It has some similarities to Etcd and Spotify’s DNS model and uses the same RAFT implementation as Etcd, go-raft.

SkyDNS servers are clustered together and using the RAFT protocol, elect a leader. The SkyDNS servers expose different endpoints for registration and discovery.

For service registration, services use an HTTP based API to create an entry with a TTL. Services must heartbeat their state periodically. SkyDNS also uses SRV records but modify them also to define service version, environment, and region. As service discovery, the client procedures DNS and retrieves SRV records for the services which need to contact as service

discovery. Load balancing and failover need to implement by the client. Caching and refreshing service location data periodically.

For dynamic service registration, SkyDNS do not depends on another external service but can provide its solution.

Overall, this is an interesting mix of old (DNS) and new (Go, RAFT) technology and expected to see a lot of project evolvement.

Argo WS

The Argo Runtime Service Discovery is a straightforward and robust protocol without the use of the central or federated registry. Argo mainly for the discovery and location of services on a wide-area network without the utilization of a central or federated registry [44]. The primary use case is to communicate configuration information – such as IP Address and Port – to service consumers. The technology and architecture behind Argo are not novel and have been in common use for decades (such as Bonjour [45] and WS-Discovery).

Argo WS is similar to Bonjour (mDNS), WS-Discovery and Simple Service Discovery protocol (SSDP), but the primary use case is the same. The main alternative use case for the Argo protocol is Network-Based Moving Target Defense against cyber-attacks.

Argo, like the rest of the protocols, is based on IP Multicast. Specifically, in the multicast to get the matching services (i.e. client configuration information) out of network a query has been sent. For like other existing SLP's, the similarity of Argo ends there.

Argo's architecture and implementation are geared toward the following things (and by implication, other SLPs do not do these things):

Open Source: It is open source and easily adaptable

Long-range (wide-area network routable): Avoid protocol-specific gateways if at all possible.

Staged adoption: Flexible topological location of the Responder for service advertisement.

Network efficient: Argo is unreliable and slow, which is a good thing, especially for a long-range protocol.

Service Query (payloads as simple as possible): Provide the ability to ask for any application protocols

Service Description (payloads as simple as possible): Provide broad adoption potential for the universe of application protocols

Expandable – Argo is scalable in the context of scalability.

Argo is a dedicated type service discovery tool and has a particular purpose to operate. Argo aimed to work for military work and based on AP patterns. Argo is not in use with cloud platforms.

Consul by Hashicorp

Consul [46] is a tool for discovering and configuring services with infrastructure and has multiple components. Some key features of Consul followed as:

Service Discovery: A service from the clients of Consul can provide an API which can use by other clients of Consul to discover providers of a given service. Using either HTTP or DNS, applications can easily find the services they depend upon.

Health Checking: Consul provides health checking service which can be utilized by clients to get status either about the application or local node. Like if the local nose is out of memory or high usage of memory. To monitor cluster health, that information can be used by the operator.Also for service discovery components as to know the route is not from hosts that are unhealthy.

Key Value Store: Applications can take advantage for any number of purposes of Consul's hierarchical key/value store. Purposes can be dynamic configuration, coordination, leader election and more. An HTTP API makes it easy to use.

Support for Multi data center: Consul provides the support for multiple data centers which means not to worry about any additional layer of abstraction for growing to different regions.

Consul is designed to work for distributed application and also worked across the Cloud environments. Consul makes handy for DevOps and developers as well.

Table 5 presents an overall overview of the discussed open source tools, grouping them based on their Type, Availability, Consistency, Language, Dependency, and Integration.

Name	Type	AP or CP[1]	Language	Dependencies	Integration
Java WS Discovery	General	CP	Java	JVM	Client Binding
Zookeeper	General	CP	Java	JVM	Client Binding
Doozer	General	CP	Go		Client Binding
Etcd	General	Mixed	Go		Client Binding/HTTP
SmartStack	Dedicated	AP	Ruby	Haproxy/Zookeeper	Sidekick(nerve/synapse)
Eureka	Dedicated	AP	Java	JVM	Java Client
NSQ (lookupd)	Dedicated	AP	Go		Client Binding
Serf	General	AP	Go		Local CLI
Spotify/DNS	Dedicated	AP	N/A	Bind	DNS Library
SkyDNS	Dedicated	Mixed	Go		HTTP/DNS Library
Argo-WS	Dedicated	AP	JAVA	JVM	Client Binding
Consul	General	CP	Go		Client Binding

Table 5: Overview of Open source Service Discovery tools

[1] In the table 5 AP means Availablity pattern and CP means Consistency Pattern

2.6 Requirement Specification

Based on the previous sections and discussions about currently available and relevant technologies, a list of system specifications will be presented in this chapter. This thesis aims to establish a setup which can contain several platforms hosting different Web services. One single Service Discovery mechanism is not enough to achieve the goal of this thesis because there are several factors like availability, consistency, security, automation, and interoperability which have to be considered when choosing the Service Discovery mechanism which best serves the thesis purpose. The focus of the thesis to build a proof of concept which can answer the following research question stated in section 1.3 in chapter 1.

Is it possible to create a common platform to provide support for Service Discovery in hybrid environments?

The absolute requirements which should be fulfilled to support this research question form the basis of this thesis. Table 6 represents the primary needs of this thesis:

Requirement	**Description**
Req 1	A SOAP Web service which can provide and hosted on a cloud platform can register on the service discovery mechanism and discoverable from the cloud platform.
Req 2	A SOAP Web service can register to the service discovery mechanism, and discoverable from a Non-Cloud platform also be discoverable.
Req 3	A RESTful Web service can register and discoverable from the cloud platform.
Req 4	A RESTful Web service should register and discoverable from Non-cloud environments.

Table 6: List of primary requirements

According to requirements from table 6 stated above of thesis SOAP and REST Web services need to able to discover which organized in a hybrid environment both local and Cloud platform. By discussion on section 2.5 and mapping the requirements listed from table 6 an overview of supported features represented following in table 7.

Name	SOAP	REST	Native(Non-Cloud Environment)	Cloud Environment
Java-WS discovery	Yes	No	Yes	No
Zookeeper	Yes (SOAP over HTTP)	Yes	Yes[2]	No
Doozer	Yes (SOAP over HTTP)	Yes	Yes[1]	No
Etcd	Yes (SOAP over HTTP)	Yes	Yes[1]	No
SmartStack	No	Yes	Yes	No
Eureka	No	Yes	No	Yes
NSQ (lookupd)	Yes	No	Yes	No
Serf	Yes	No	Yes	No
Spotify/DNS	Yes	Yes	Yes	No
SkyDNS	Yes	Yes	Yes	No
Argo-WS	Yes	No	Yes	No
Consul	Yes	Yes	No[3]	Yes

Table 7: Overview of Open source discovery tools by context of thesis

To fulfill the goal of this thesis, it can be visible from table 7 that one single service discovery mechanism is not enough for serving the requirements listed in table 6 without tailoring. In that case, pervasive service discovery mechanism can use to achieve the goal of this thesis. For fulfillment of the goal of the thesis, a prototype needs to design and implement for proof of concept of the pervasive service discovery mechanism.

[2] Zookeeper, Doozer and etcd needs 3 to 5 nodes set up in the environment.
[3] Consul can be used in local environment but it needed minimum 3-5 instances for fully operational.

The specification in table 8 forms an essential part of the task and the requirements to the finished projects. The specification table can be seen below contain three rows, specification name, a short description and the importance for the given specification.

Name	Description	Importance	No
User Interaction Interface	Implementation of a UI that can be easily implemented by client without having prior knowledge of application	Medium	1
Expandability	The addition of new features and new frameworks should be easy to integrate.	High	2
Scalability	It should be operable for multiple instances.	High	3
Automatic	Servers and services integrations must be easy to handle. After integration of new server or service should be available to the client automatically.	High	4
Security	Proper authentication must be maintained.	High	5
Easy integration	Integration must be simpler to client and infrastructures.	High	6
Ability to meta information	Provides functions to add extra information about services to the system	Medium	7
Admin Settings	An admin panel to handle the manual settings	High	8
Documentation	Proper documentation necessary according to standards which can be understandable to other developers	High	9
Information sharing	The user should have right to which services to share and which not to, based on service type.	High	10
Discoverability	Web services should be discoverable from a local network or any cloud network.	High	11
Modularity	The system should consist of several modules so that they can be easily changed or added if needed.	High	12
Maintain SOA principles	Strive to follow the SOA principles in the project, making it easier to maintain, adapt and develop for future use. Following SOA principles make the system robust and increases the performance of the system.	High	13
Heterogeneity	Web services mixed SOAP and REST in this project	High	14
Hybrid platforms	The environment set up of the project should be mixed with a local noncloud network and a remote cloud network	High	15
HTTP and XML supported	Both XML and HTTP message transportation should be supported.	High	16
Language independent	Although the project is aimed to be made in java, the services made of node.js, java, c++, .net should still be supported	High	17
Notification Engine	A notification engine should be there to notify if any service goes down or is not available.	High	18
Monitoring	Continuous monitoring should be there to health check the connection between service and servers	High	19
Load balancing	Load balancing feature must be there as it is intended to have scalable features.	High	20
Low response time	Response time should be less than 10 seconds	High	21
Zero Down time	System should have some fallback plan if it goes down	High	22

Table 8: List of System requirement specification

From the overview of the open source discovery tools given in table 7, it is clear that none of the discussed discovery mechanisms can support both REST and SOAP service discovery across both cloud and non-cloud environments. It is also clear that Consul is the best

candidate, as it supports everything except non-cloud deployments. That leaves two options for supporting service discovery in hybrid networks:

1. Combine multiple mechanisms and bridge between these
2. Modify an already existing mechanism to ensure that it supports all service types and environments

Due to the fact that Consul already supports the majority of the requirements from table 8, in this thesis, we opt to investigate how we can support service discovery using method number 2.

3 Design

This chapter derives the specification, represents planning for the system which meets the specification. It serves as a blueprint for the execution of the experiment and interpretation of its results. The design is a process where we create and shape artifacts that solve problems. In software, for example, design means crafting software that does jobs users want to be done. Software designers intentionally support practices, worlds, and identities of the software users. According to Peter J. Denning [47], the latest trend for finding better solutions to problems is the Design thinking. Computational thinking combination with design thinking deals some real possibilities for improving software design. Design principles in computing guide us to ways of building machines whose behaviors are useful and meaningful in their user communities [47]. The design based on the research goal and hypotheses that support and a matching research design is then selected. Following that, the details of the experimental design are discussed, including its parameters, variables, planning, objects and procedures for data collection and analysis. Finally, an evaluation is made on the validity of the experimental design. Case studies are well suited to capture and describe how software processes occur in real-world settings, what kinds of problems emerge, how they addressed, and how software engineering tools, techniques, or concepts are employed.

Based on the requirements specification derived in chapter 2 a project plan has been defined, and small goals have been set in each sprint, developed and tested functional part. This process maintains the project goals at the end.

In chapter 2, technologies related to this thesis have been discussed which are the primary artifacts of this research. In chapter 2, an overview of available open source standards is provided. From Section 1.3 of chapter 1 the research question was stated as follows:

Ques 4: Is it possible to create a common platform to provide support for service discovery in hybrid environments?

To achieve an answer to the above question a goal in this phase is to design an environment which contains different types of Web services deployed in several different environments and try to discover those Web services in the same manner.

3.1 General Design

The thesis divides into two parts in the design phase: First is the environment set up, followed by the Web Service Discovery Mechanism application design.

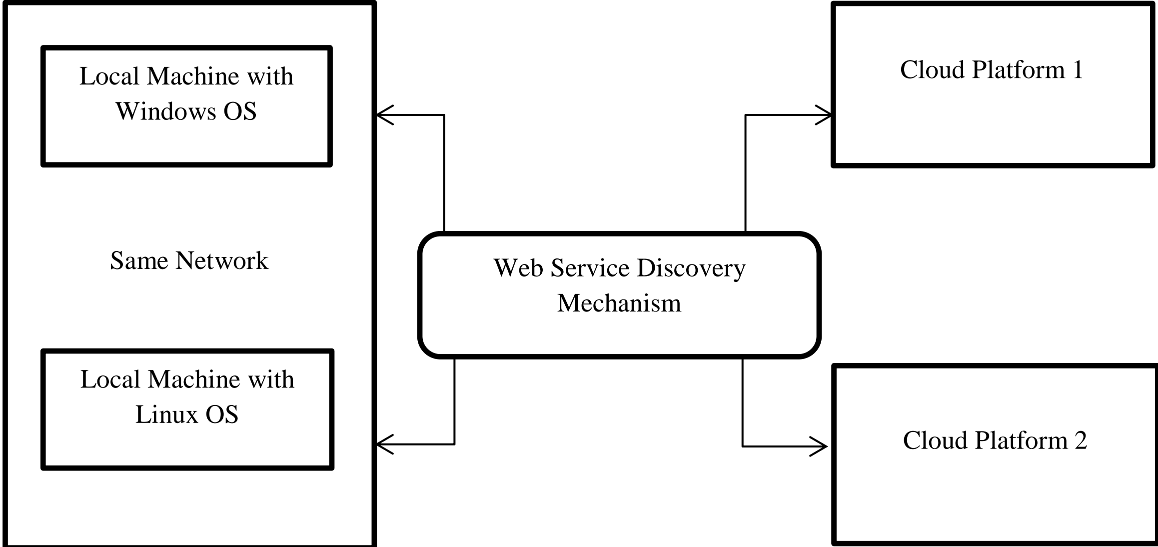


Figure 3: An overall view of the project design

A preliminary overall view with building blocks is presented in figure 3. The focus is the Web Service Discovery Mechanism and also the setup of environments. The Web Service Discovery mechanism can have a composition of open standards which are going to discover Web services from any environments. To meet the requirements number 14 and 15 from the specification list, two different cloud platforms from different vendors. There is also a setup of two single local machines with two different Operating Systems in the same LAN. The goal is to discover different Web services which hosted on various platforms in the same manner. This is done by the Web Service Discovery mechanism, which is represented by the middle box in figure 3.

Figure 4 presents a graphical overview of the central components of the design, which is the "Web Service Discovery Mechanism" in this thesis.

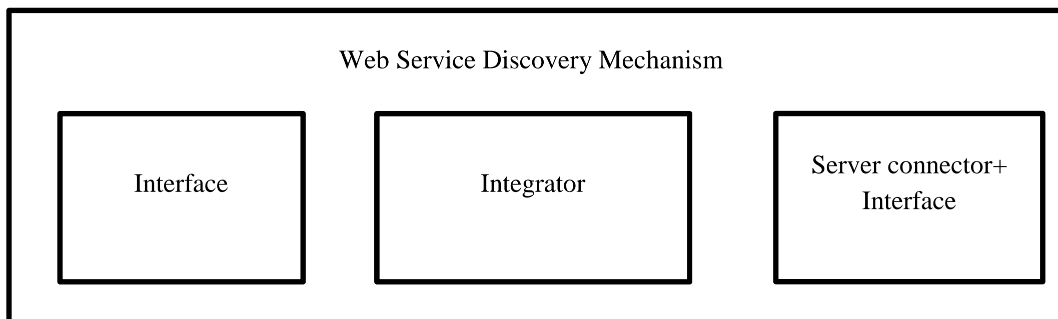


Figure 4: Overview of the Web Service Discovery mechanism

In the box, three separate blocks constitute the essential parts of this service discovery mechanism: An interface, an integrator, and a server connector. The Interface is a simple interface which has some basic functionality. The Integrator captures the client request and validates according to request type. A Server Connector can route the request based on the output of the integrator. All of the modules related to the design are discussed further in the following sections.

Service discovery tools manage how services and processes in a cluster can find and talk to each other. They involve creating a directory of services, registering services in that directory, and then being able to look up and connect to services in that directory. From the background analysis in chapter 2, we found that a single open source service discovery mechanism is not able to meet all the requirements on the setup of figure 3. The pros and cons of different service discovery mechanisms can be found in table 4 and 6. An adaptive pervasive service discovery mechanism can be introduced by adding some features and customizing the solution according to need. According to [5], there are four typical components of service discovery: Where is the thing? What are the IP and port? How do services connect? Moreover, a health monitoring piece to detect the functional server.

Principal facts of Web Service Discovery projects are:

- Services need to send a notification to each other about the status and can supply connection information.
- A periodic update is necessary to the records to strip out old information.
- An easy integration with application is mandatory which mostly use standard protocol like HTTP or DNS
- Notification mechanism on services which are starting and stopping must be integrated.

For supporting these features as well as our requirements stated in chapter 2, Consul is chosen as the backend, with moderation and customization with our set up as described below.

Why Consul?

Consul [46] is a newer protocol by HashiCorp [48]. A short description of Consul was given in chapter 2. It is a general purpose distributed Web service discovery tool with a key value store. Consul has the features of service discovery, monitoring, load balancing, and health check and multi data center supported. This meets requirements 2, 13,18,19,20 specified in chapter 2, section 2.6. Consul follows SOA principles which are also requirements specification of this thesis. The key value store mechanism of Consul helps to integrate quickly with the other application.

According to figure 3, this thesis aims to have a platform set up with a combination of cloud and the non-cloud environment. Consul is also suitable for cloud environment like Amazon Cloud, and non-REST endpoints are supportable via DNS. In the Consul a service can be registered by service description or a direct call to an HTTP API.

Consul has several benefits and fits for this thesis as because of integrated DNS, ease of use with Docker and clear documentation.

3.2 Workflow model

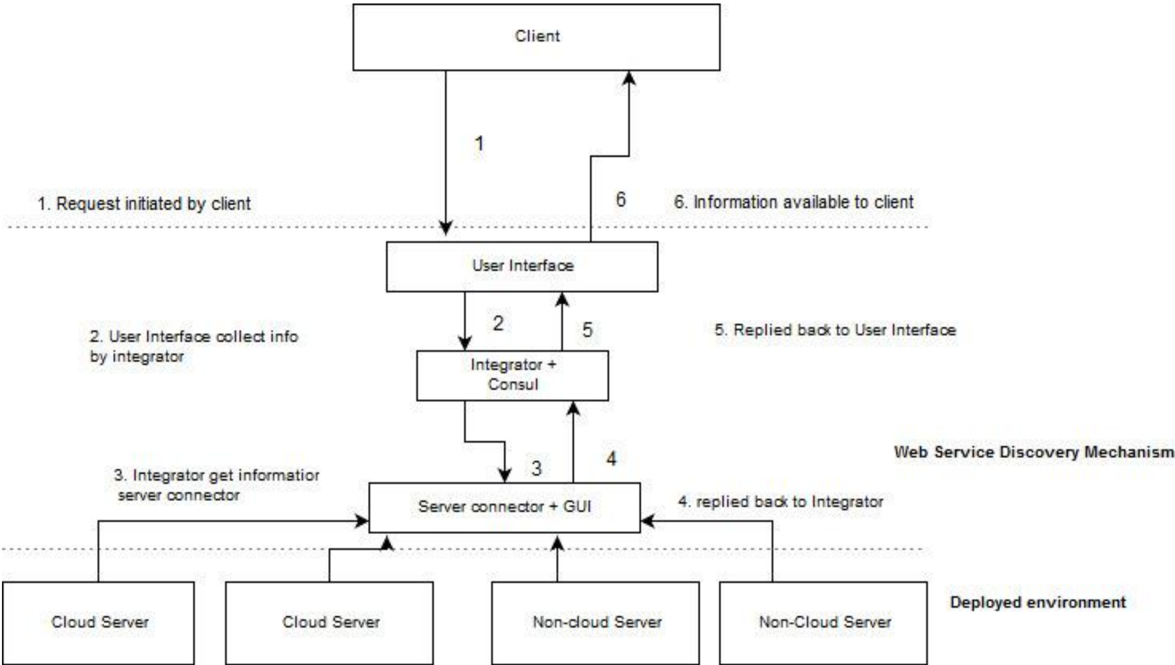


Figure 5: Basic flow model of Web service Discovery mechanism

For requirements number 6, 12, 13, 14, 15 specified in chapter 2, the application is designed based on modular form. There are three modules that either work together or independently to maintain the simplicity of integration with any other project. Modular based design is preferable because components can be modified or changed without hampering the other modules so that changes have only minimal impact. The application design works in two parts: The first part considered as the Frontend part and the second part is the Backend.

According to figure 5, the basic workflow can describe as follows:

1. A client can make a request, and it comes to the application into the User Interface.

2. The request flows down to the Integrator module based on service requests. There can be three types of requests available to the Integrator module:

(a) The request can have authentication to access the Interface of Web service discovery mechanism.

(b) The request may be to show the available services registered in the Service discovery mechanism.

(c) The request can be to connect with the Web service discovery mechanism for notification to get new services.

3. Based on the type of request, the Integrator fetches the information from the Server Connector module.

4. Reply from the Server Connector module response is sent back to the Integrator.

5. Integrator module processes the response and sent a reply to the client accordingly.

The workflow model shows the basic flow of request in the Application. To meet the requirements 18, rather than that there also some background requests served through the User Interface to the client. Figure 6 presents the workflow between the client request to Integrator module. Background application can add the new service Information, update information as needed and delete the offline Service information.

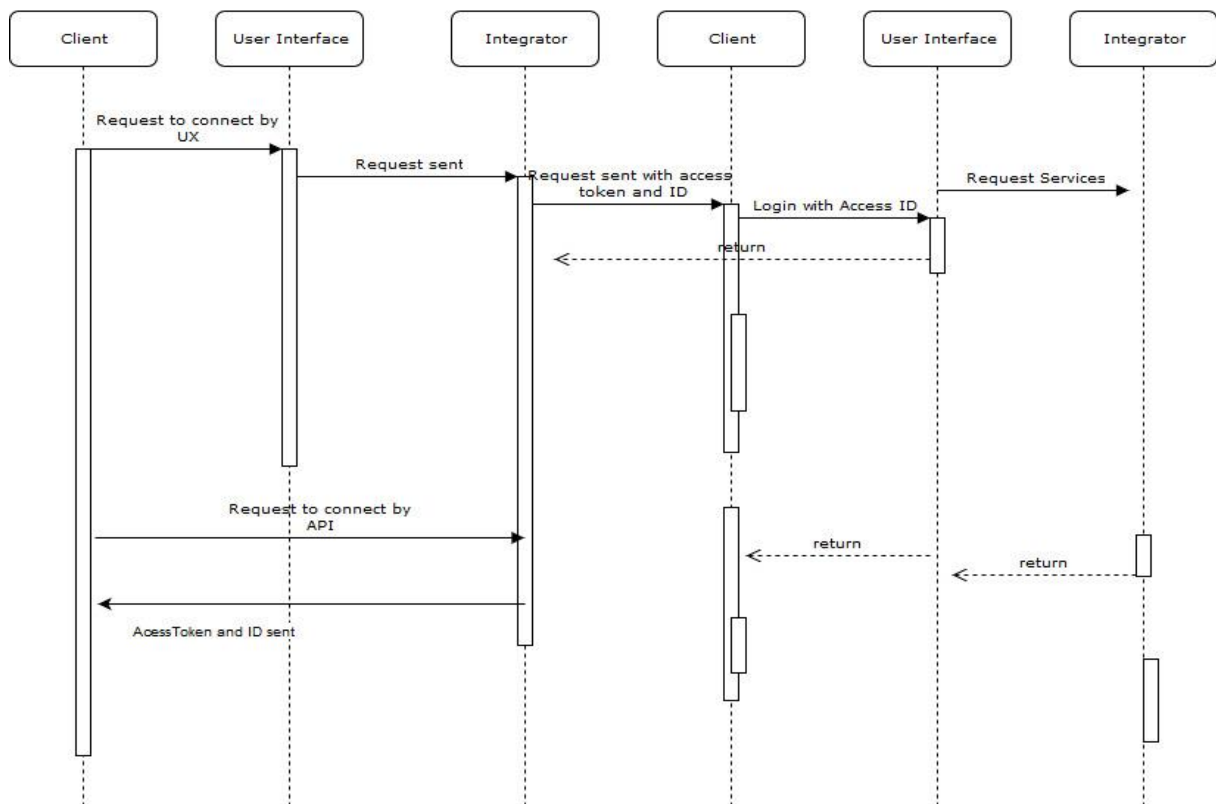


Figure 6: Client- Application request workflow

The Integrator part of the application is an essential feature which implements Consul API for service discovery. To comply with requirements 1, 3, 11,14,16,20 from chapter 2, integrator has a utility class like adapter which supports both SOAP and REST request.

The integrator can persist all Web service information with type and category. The integrator also holds a Server Configuration file. The integrator of the application is responsible for:

- Connecting and Updating Server information in the Server Configuration file
- Maintaining the connection with Server connector
- Implementing Consul services within the application
- Notification processing and sending to the User Interface module
- Generation of Access Token and ID to the client
- Authentication and validation maintenance of Client and Server

Integrator module has a ping mechanism to check the server status, so a notification engine has been set up for better monitoring.

Server Connector is the final module of this application. This module also represents a simple GUI interface for management manually to comply with requirement no 4,5,7,8.

Server Connector UI interfaces are responsible for manual interaction with following jobs:

Add New Server Information

According to figure 5, the application has a setup of working with heterogeneous servers so Servers information can be added to the admin panel through this interface. For dynamically maintaining the application and increasing the usability. This also sends a request to update the Server Configuration into Integrator module.

Delete unusable Server Information

Unused server information can be deleted by this function. In this application, platforms are also considered so server which is not actively deleted from the configuration and update server configuration information accordingly.

Update Server Information

Also, can update existing server information if needed. For example, if any server needs to reconnect with the application. As all applications considered for this thesis is distributed manner any kind of change of server information like Port, IP and other metadata information can update through the application.

Push notification

This is the mechanism where information about servers' status is sent to the Integrator. And also new service information which added to the server is also sent to the Integrator module.

Servers are integrated with the application by connecting accordingly. The Load balancer feature of has been used while capturing requests and deliver the request. The Integrator module gets connected with a connector to monitor the server status and health check. The Server Connector also has a notification engine for notifying about server status to the Integrator module and updates the server configuration file. The Integrator module can query service information from the servers and displayed them through the User Interface. Figure 7 represents the server connector interface:

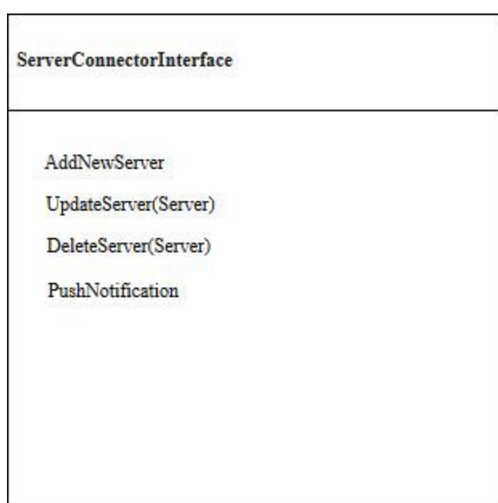


Figure 7: ServerConnector is the interface between Servers and Integrator

Workflow of Server Connector with Integrator module as follows in figure 8:

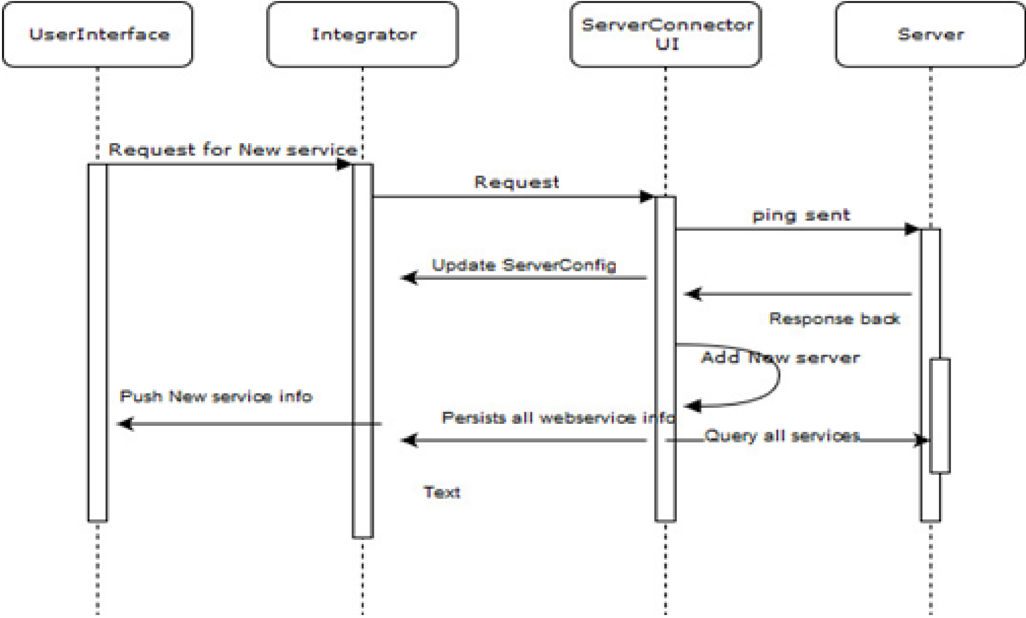


Figure 8: Server Connector workflow diagram

The next chapter will discuss more in detail in Implementation of the design. Chapter 4 also analyzes the evaluation and testing of this application.

4 Implementation & Evaluation

This chapter presents the Implementation and Evaluation of the design. The first section of this chapter addresses the details of this project with application development. The second part of this chapter represents the testing and evaluation of the design and implementation.

4.1 Implementation

The implementation matches the design described in chapter 3, although some minor changes have been made during development. In this section, the detailed implementation is discussed to get an overview of the application development.

According to figure 5 in chapter 2, the application has been set with three parts: Interface, Integrator, and Server Connector part. All the implementation has been done in Java language.

Consul is written in the GO language, but it has a Java library which can be used as an integrated library with the application; this feature is very much applicable for the thesis. By modification and by using it as a library of this thesis project it can be used to work with small scale. In this project, Consul Interfaces and methods are implemented and use Consul-HTTP client in development.

As Consul is recommended to use with at least 3-5 servers [46], so Docker [49] has been used to containerized this modular application services and fit for this project and run from a single server.

The Model view controller pattern follows to structure the code of the application. In the view package, two User Interfaces have been created; one for the Application client interaction and another one for Server Connection management. In the model, package service models have been introduced, for example, the Web service class, Server class, and some helper entities.

In the controller, the package includes the Integrator part of the Consul Package. The Consul Discovery package has been added as a library and included with the application for better management. An authentication module also has been included in the integration module to validate the client's access id and token. The User interaction interface has a dashboard for viewing the status of the services. New service integration in the server has sent an auto

notification to the Client from the system. Client interaction user interface can be used to update client information and the subscription also.

In the front end part of this application, a simple Interface was implemented with some basic functionality for the Client. The client can login with a token which has been created earlier. If the client wants to connect, authentication gets done by access token and ID. In the user interface, a client can get notifications of new service integration, edit profile and can see a dashboard of all available services. The client can choose information what to share and what not. When a client requests to connect with the integrator backend part, the integrator API sends an access token and ID to the client. This can maintain the secure interaction between the Client Request and Application. The client can access the UI by that token and ID to navigate the services of the application. A simple OAuth [50] protocol and login system has been used in that case. OAuth2.0 has several open source implementation like OpenID connect [51] . OpenID Connect has the feature of interoperability, Security, Ease of deployment, and flexibility. OpenID Connect is designed to support native apps, mobile application, and Web-based applications. OpenID connect uses a simple JSON/REST based protocol and has a system level API to interact with the machine to machine implementations. OpenID connect defines a discovery [52]and a client registration [53] protocol. A library has been added to support OpenID connect in this application for authentication. During implementation, a simple user interface has been made to work with.

```
curl 'http://webservicediscovery.com:8080/mythesisdemo/WebServiceDiscoveryDemo?accessID=accessid&accessToken=accessToken'
```

Client Interface mock has been shown in figure 9 below.

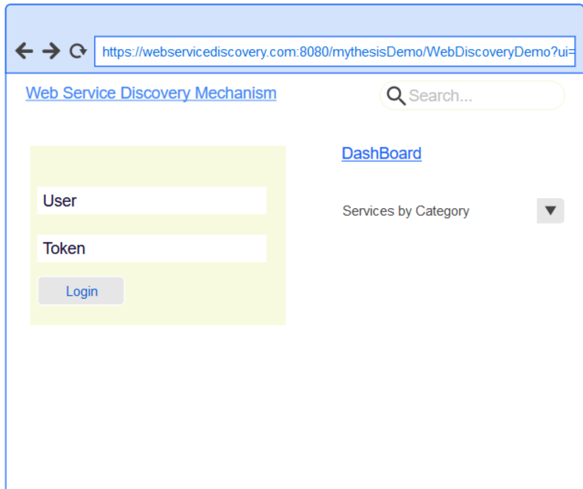


Figure 9: User interface mock

The Server Connector Interface can add new server information, update existing server information and delete the server. The Server Connector adds the server with Server name, URL, authentication ID and token. The Delete server operation can then be done by presenting the serverID. This interface can also generate a notification and send a ping at regular intervals to find out about the server status and push notifications on the basis of that. This maintains some core functions defined in the server connector backend.

Web Service Class

```
public class WebService {
    private String serviceName;
    private String hostName;
    private String port;
    private String type;
    private String key;
    private boolean isUp;
```

Figure 10: WebService class

In the Web Service class in figure 10 there is a service name variable, which represents Name of the Service Name, Host Name is host URL, port details of Web service, type represents the Web Service type, key is the string value of storing key which is the indicator of types of Web service, and there is also a Boolean variable which represents the status of the Web service.

The server configuration file, which is maintaining information about how to connect with different servers, can be a JSON or XML file. It helps the Server Connector module to connect to the servers and to fetch information from the server. When we invoke GET API to display all the servers connected it returns a JSON string.

This is a simple application that is easy to integrate. The model of the application mainly contains the Server Configuration and Web services information. An Object Mapper is used to map the JSON values to the model. A Utility class also has been defined for handling some validation and setting.

The Integrator implements the Consul Service discovery mechanism and also adds some external features. Appendix A describes Consul and its working principles.

The Integrator module uses Consul which deployed using Docker shown in figure 11. In appendix B, Docker is described. As Consul needs 3 to 5 instances to work consistently,

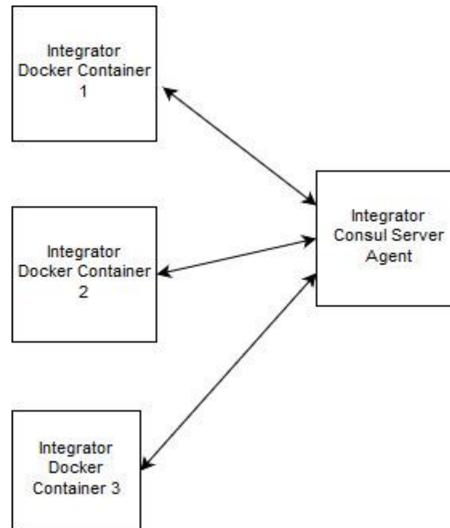


Figure 11: Integrator Consul with Docker

Docker images can be created with different service IDs. One Integrator with the Consul Server agent has been set up, which will elect the other three Docker images as client instances and coordinate accordingly. This makes the application consistent and available .When a client subscribes to the Integrator API, it responded with an access token and ID which is used in future accesses to the User Interface of the application.

The Integrator module makes the new service information available via the user interface. This happens automatically when a new Service is added to the server. The Integrator module implements the Consul Discovery interface. The Integrator module has utilized the agent, catalog, health check, load balance, session and status of Consul.

SOAP and REST services have some differences as discussed in chapter 2. SOAP has WSDL description while REST has only resources to work with. In the Integrator module, Web Service class has a parameter key which can track about the service type.

The server config JSON file, which is shown in figure 12, is maintained in Integrator module. This JSON file is updated through the Server Connector interface with the Server information. In the Utility function, Java socket connections are used to connect with the application to register the services through the Integrator module and Consul to the application. With the help of Consul Health check system service status can be monitored.

```
{
  "server": [
    {...16 lines },
    {
      "id": "X_124",
      "meta": {
        "name": "Servername B",
        "description": "local"
      },
      "host": "http://192.168.0.20:8080",
      "accessinfo": {
        "id": "sdHFGsd23",
        "token": "lsdfklsdf"
      },
      "type": "local",
      "created_time": "2010-08-02T21:27:44+0000",
      "updated_time": "2010-08-02T21:27:44+0000",
      "status": "connected"
    },
    {
      "id": "X_125",
      "meta": {
        "name": "Servername C",
        "description": "cloud"
      },
      "host": "http://demo.com:7979",
      "accessinfo": {
        "id": "sSDLFKL3",
        "token": "ksdlfhlk"
      },
      "type": "cloud",
      "created_time": "2010-08-02T21:27:44+0000",
      "updated_time": "2010-08-02T21:27:44+0000",
      "status": "connected"
    }
  ]
}
```

Figure 12: ServerConfig file.

The integrator also works as a backend service of the server connector module. A notification engine is implemented to monitor server status, and pings are sent to the servers as a health check.

The Integrator implements Consul Load balancing features with this application to support requests in large scale without service interruption. Consul is intended to work with multiple instances, so the Integrator set up has been done to operate the application in a single instance.

A Resource folder has also been added to the project. The resource folder has some configuration file: logger.properties, config.properties, services. properties and server. properties.

Slf4j [54] is a well-known logging framework enabling multiple logging levels and customizable outputs. The use of a logging framework in the application simplifies the development process and is a great advantage for other developers who may want to continue to work on the application, as it makes it easier to debug and expand. Config.properties files are used for application specific settings; this includes many options for the module based settings.

The application is based on several external services so the service. Properties files contain service information which needs to start with an application running.

Server.properties files contain the information by which servers can connect with the application.

There is also some internal validation, functions and classes have been introduced which are not in the design chapter. Those have been used to make easy with the development process and increase reusability of code in the application. Some necessary libraries added for several purposes. The PropertyReader class was implemented to read values from the properties file. The XML parser class was introduced to parse any XML. JSON reader and JSONConverter are used to read parameters. In the Utility class, some functions have been added to have re-using facility and better code management.

The implementation is done according to the design of the application program. The entire application can be found on GitHub and technical guide to the application can be found in appendix C.

4.2 Evaluation

This section represents the evaluation of this project. According to Peter J. Denning design [47], testing has a great impact on the research method of this thesis. Testing is part of the Software Development Lifecycle (SDLC) [55]. Testing helps to improve reliability, performance and also to check that the application work as intended. In this thesis evaluation defined to meet the requirements specified, design and implementation of the design work expected. There are several different approaches to test an application. These approaches can be broken into smaller pieces, which will help to understand and differentiate between them.

Software testing is mostly based on two types: Manual Testing and automated testing. Manual testing, as the name implies, is when a tester or developer tests code component by component. These tests are very flexible and almost every piece of code can be tested, but are not feasible when the time is short and resources limit has been set.

An automated testing tool can be applied to test the system. Automated testing is applicable when a large amount of code needs to be tested.

There are also two types of testing depending on the domain knowledge about the inner working principle of the software: White Box Testing and Black Box Testing. White box testing refers to a system where everyone can see the functionality of the system work as intended. A Developer or Tester with some good understanding and knowledge of the system can do white box testing with some test cases. White box testing is not possible for anyone who does not have the prior knowledge and setup of the system.

Black Box Testing is where the user can input data and gets expected output without having any knowledge of system internal working principles. It is good for the organization as the testing can be done by anyone without any involvement of the Developer.

For this thesis, manual testing is used with the white box as a testing method. Some practical use cases have been set to test the system.

The application has been tested through some technical and feasibility tests. Use cases for this application in table 9 are:

Use Case 1	**The system should connect with different platforms.**
Use Case 2	A SOAP Web service can register to the system.
Use Case 3	A REST Web service can log into the system.
Use Case 4	The client can reach different platforms through the system.

Table 9: Primary Use cases of application

Use cases have been tested and are described through the rest of the chapter.

As stated in chapter 3, the thesis has a two part setup. One is the environmental configuration, and the other one is the application.

The Application followed Test Driven Development (TDD) [56]. Unit testing has been conducted during the implementation phase. A unit test is the basic form of testing. In the low-level unit test usually, the small unit of code focuses on testing like a class or a method. In most of the cases, unit testing is conducted as a manual test. The benefit of the Unit test is to discover issues during development so that they can be resolved as early as possible. It saves time and effort while the system goes into production.

In this thesis, the application is mainly driven by the Integrator, which implements the Consul HTTP API. The integrator has been tested to see that the functions are working properly. See figure 13 for sample test code.

```
71
72          assertTrue(found);
73      }
74
75      @Test
76      public void getAllservices() {
77          ConsulClient client = ConsulClient.consul();
78          String id = UUID.randomUUID().toString();
79          client.agent().registerService("192.168.0.52", 8081, 20L, UUID.randomUUID().toString(), id);
80
81          boolean found = false;
82
83          for (Map.Entry<String, Service> service : client.agent().getServices().entrySet()) {
84              if (service.getValue().getId().equals(id)) {
85                  found = true;
86              }
87          }
88
89          assertTrue(found);
90      }
91
```

Figure 13: Sample Test code

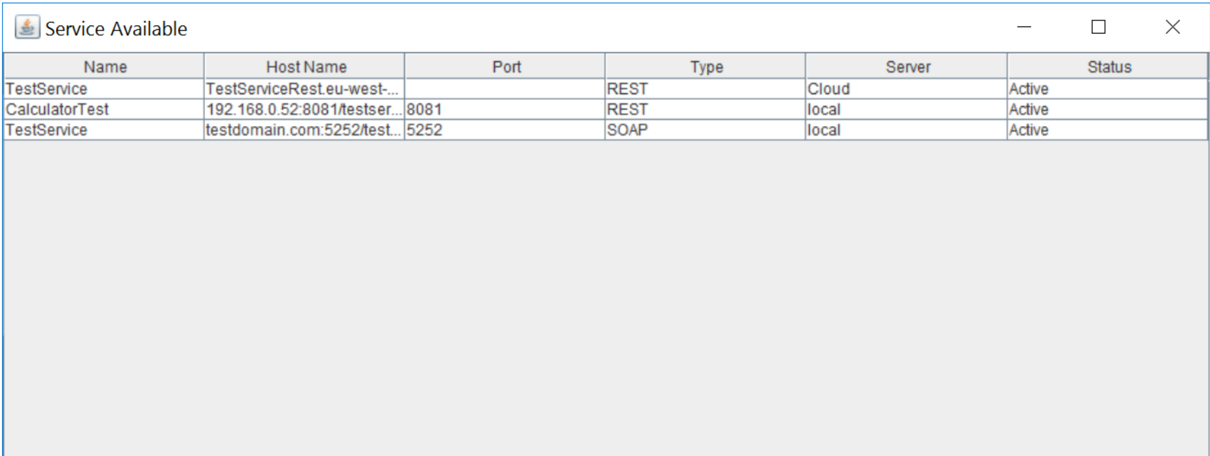
Functionality Test

A functional test is intended to ensure the application is functioning as desired. The design of the application is modularized, and three main components have been introduced in figure 4. So, test cases were designed according to module and function. A series of different kinds of functional tests for this application have been conducted. Recall from chapter 3 that the application has been set up with three components. The User Interface part has been tested with respect to functionality. Through the User Interface, a Client can log in with proper information with mock data. (S = Success; P = Partial; F = Fail)

Test Nr	Test Case	Description	Result
T1	Log in	Client Can invoke with proper UserID and Token	S
T2	Update Profile	User can update information	S
T3	Validation	UserID need to be validated	S
T4	Available Service visibility	Client Can see the available service information	S
T5	Notification	A new service Integration can be notified to the client connected to the application	S

Table 10: Test case for User Interface

In table 10 few use cases have been identified and test accordingly. In the T1 client should able to login with proper Username and Token. UserID and token are persisted so the client can login and update information T2, which is so simple client can only change the name or token. In the T4 client can see the available service information those are registered into the application. A mock interface is shown in figure 14. In this test, three mock Web services are registered with the application: one SOAP Web service and two REST services deployed on three different platforms.


Service Available

Name	Host Name	Port	Type	Server	Status
TestService	TestServiceRest.eu-west-...		REST	Cloud	Active
CalculatorTest	192.168.0.52:8081/testser...	8081	REST	local	Active
TestService	testdomain.com:5252/test...	5252	SOAP	local	Active

Figure 14: Service Information page

There is another interface which is responsible for servers. According to table 11, some test code was written to add a server, delete a server and update server information to the configuration. This function was handled by the Integrator in the backend, so it was tested as part of the unit tests.

Test Nr	Test Case	Description	Result
T6	Add server	A server add through that interface	S
T7	Update Server	Server Information Can Update through the interface ex: URL, port	S
T8	Delete Server	Server can delete from the Application	S
T9	Update Server Config	A JSON file need to update after adding new service	S

Table 11: Test case for Server Connector Interface

Some test code was developed to check that the Integrator can create a Consul agent. This test was done manually from command line. Figure 15 shows the Consul agent started

```
jan@node-1:~$ consul agent -dev -bind=127.0.0.1
· Starting Consul agent...
· Starting Consul agent RPC...
· Consul agent running!
       Node name: 'node-1'
      Datacenter: 'dc1'
          Server: true (bootstrap: false)
     Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
    Cluster Addr: 127.0.0.1 (LAN: 8301, WAN: 8302)
  Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
           Atlas: <disabled>

· Log data will now stream in as it occurs:

  2017/07/30 01:30:09 [INFO] serf: EventMemberJoin: node-1 127.0.0.1
  2017/07/30 01:30:09 [INFO] serf: EventMemberJoin: node-1.dc1 127.0.0.1
  2017/07/30 01:30:09 [INFO] raft: Node at 127.0.0.1:8300 [Follower] entering Follower state
  2017/07/30 01:30:09 [INFO] consul: adding LAN server node-1 (Addr: 127.0.0.1:8300) (DC: dc1)
  2017/07/30 01:30:09 [INFO] consul: adding WAN server node-1.dc1 (Addr: 127.0.0.1:8300) (DC: dc1)
  2017/07/30 01:30:09 [ERR] agent: failed to sync remote state: No cluster leader
  2017/07/30 01:30:10 [WARN] raft: Heartbeat timeout reached, starting election
  2017/07/30 01:30:10 [INFO] raft: Node at 127.0.0.1:8300 [Candidate] entering Candidate state
  2017/07/30 01:30:10 [DEBUG] raft: Votes needed: 1
  2017/07/30 01:30:10 [DEBUG] raft: Vote granted from 127.0.0.1:8300. Tally: 1
  2017/07/30 01:30:10 [INFO] raft: Election won. Tally: 1
  2017/07/30 01:30:10 [INFO] raft: Node at 127.0.0.1:8300 [Leader] entering Leader state
  2017/07/30 01:30:10 [INFO] raft: Disabling EnableSingleNode (bootstrap)
  2017/07/30 01:30:10 [INFO] consul: cluster leadership acquired
  2017/07/30 01:30:10 [INFO] consul: New leader elected: node-1
  2017/07/30 01:30:10 [DEBUG] raft: Node 127.0.0.1:8300 updated peer set (2): [127.0.0.1:8300]
  2017/07/30 01:30:10 [DEBUG] consul: reset tombstone GC to index 2
  2017/07/30 01:30:10 [INFO] consul: member 'node-1' joined, marking health alive
  2017/07/30 01:30:13 [INFO] agent: Synced service 'consul'
```

Figure 15: Consul Started

Test Environment Set Up

The test of the whole application was integrated with the environment set up. According to figure 3 in chapter 3, both local and cloud servers have been set up to make a hybrid environment. As a cloud platform, Amazon Web Services (AWS) was used. Two simple Web services were deployed in Amazon Beanstalk. A script can also use to create the Amazon BeanStalk environment and the application was deployed into that environment, as shown in figures 16 and 17.

```
# http://docs.aws.amazon.com/elasticbeanstalk/latest/APIReference/API_CheckDNSAvailability.html
    ["ebn", "check-dns-vailability", CheckDNSAvailability, [
     ["", CNAMEPrefix],
    ]],
    # http://docs.aws.amazon.com/elasticbeanstalk/latest/APIReference/API_CreateApplication.html
    ["ebn", "create-application", CreateApplication, [
     ["", ApplicationName],
     ["description", ApplicationDescription]
    ]],
    # http://docs.aws.amazon.com/elasticbeanstalk/latest/APIReference/API_CreateApplicationVersion.html
    ["ebn", "create-application-version", CreateApplicationVersion, [
     ["", ApplicationName],
     ["autocreate", AutoCreateApplication],
     ["description", Description],
     ["sourcebucket", 'SourceBundle.S3Bucket'],
     ["sourcekey", 'SourceBundle.S3Key'],
     ["versionlabel", VersionLabel]
    ]],
    # http://docs.aws.amazon.com/elasticbeanstalk/latest/APIReference/API_CreateConfigurationTemplate.html
    ["ebn", "create-configuration-template", CreateConfigurationTemplate, [
     ["", ApplicationName],
     ["description", Description],
     ["environmentid", EnvironmentId],
     ["solutionstackname", SolutionStackName],
     ["sourceconfiguration", SourceConfiguration],
     ["templatename", TemplateName]
    ]],
    # http://docs.aws.amazon.com/elasticbeanstalk/latest/APIReference/API_CreateEnvironment.html
    ["ebn", "create-environment", CreateEnvironment, [
     ["", ApplicationName],
     ["cnameprefix", CNAMEPrefix],
     ["description", Description],
     ["environmentname", EnvironmentName],
     ["solutionstackname", SolutionStackName],
     ["versionlabel", VersionLabel],
     ["templatename", TemplateName]
    ]]
```

Figure 16: Amazon Beanstalk set up

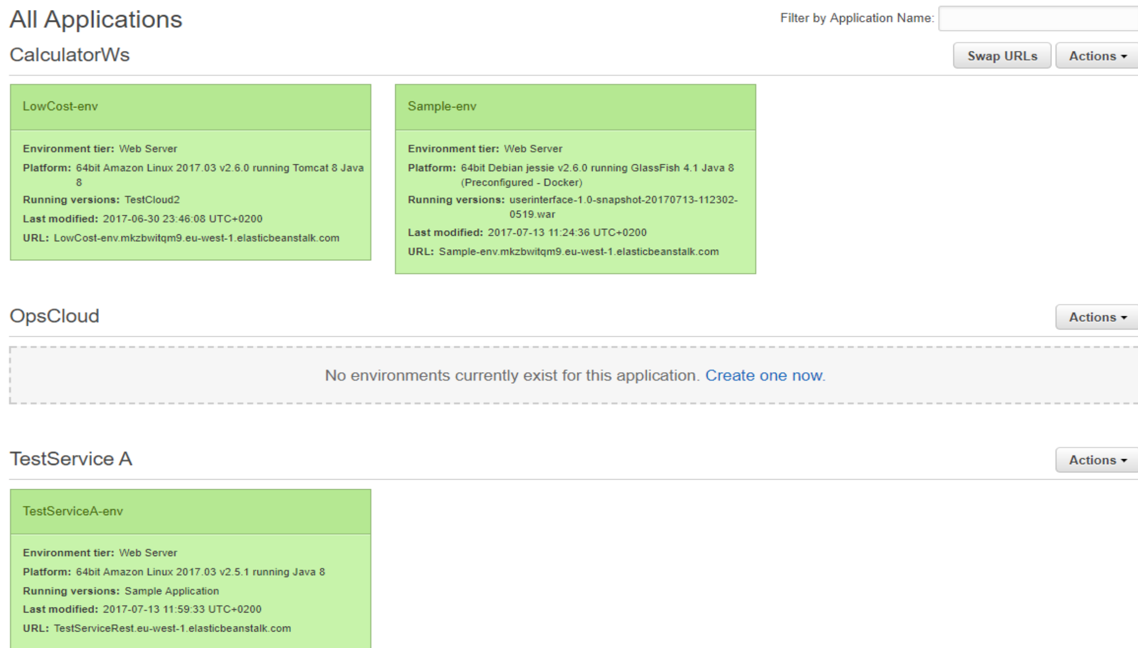


Figure 17: AWS BeanStalk console

Test SOAP Web service

A simple calculator as a SOAP Web service. The Web service can take to integer values and returns integer sum.

Name: CalculatorWs

Type: SOAP

Input Parameters: int value1, int value 2

Methods: int add (int value1, int value 2)

Test RESTful Web service

A simple RESTful Web service for getting user information.

Name :TestServiceA

Type: REST

Class: User. Java and UserService.java

A test environment was also setup with Oracle Cloud server. A test SOAP service was deployed on that server by command line. A Windows machine in a LAN was set up with a Glassfish Server 4.1where two different Web services have been deployed. Windows machine configuration is shown in figure 18.

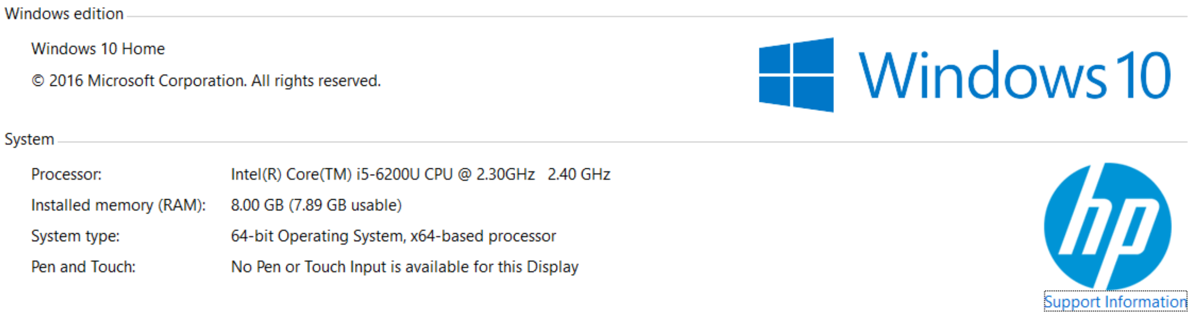


Figure 18: Windows machine configuration information

A Linux machine which configuration is shown in figure 19, also set up with a Glassfish Server 4.1 and deployed a RESTful Web service API into that.

Figure 19: Linux Machine configuration

Through deployment in these environments, all aspects of the application were tested and evaluated against the primary requirement specified in table 6 in chapter 2. As currently

SOAP is not supported by AWS, only a REST Web service was deployed on the AWS platform. The implementation of the work is kept simple so that the main goals have been accomplished.

An Ubuntu machine in figure 20 also has been used to set application and used as server application with Docker installed.

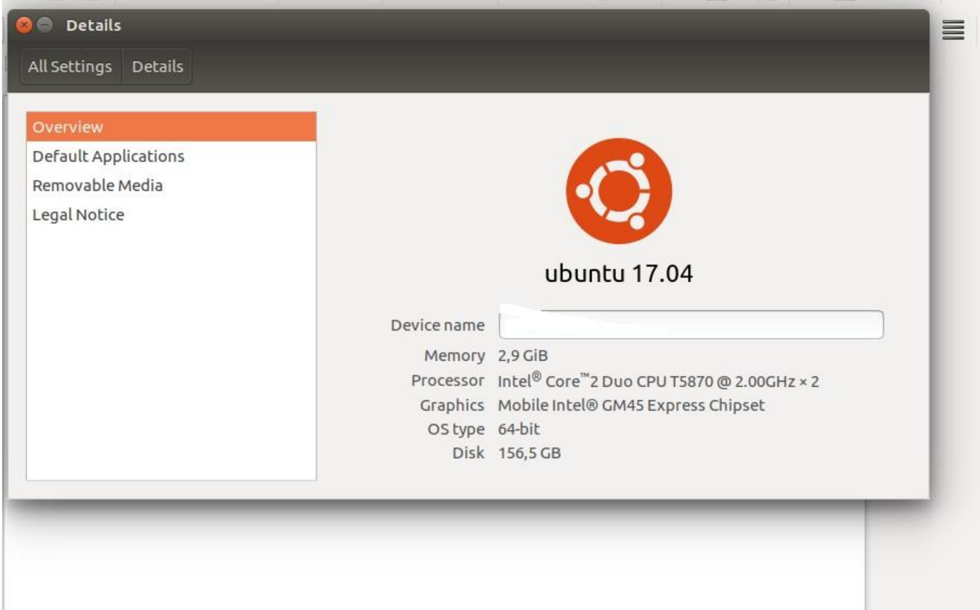


Figure 20: Ubuntu configuration for Application setup

Evaluation & Findings

The design and implementation of the application evaluate to achieve the answer to research question 4 stated in sections 1.3 in chapter 1. A prototype is presented by the application and tested accordingly. The application fulfills the requirements stated in table 8, functional and works as intended.

The application is tested with simple Web services where endpoints are single, and SOAP test Web service also has small XML payload. Consul set up is a little bit complex as also the java client does not have any user interface to interact.

The most important findings of the evaluation by testing the application are that the common platform is possible when the Service end points are not nested. If the Web services are too big and Service has many endpoints, then customization is necessary to handle this kind of scenario. Also for a very big XML of SOAP Web services, it is difficult to achieve the JSON converter to fetch the information.

5 Conclusion

This chapter presents a summary of this master thesis. Findings and observations are given in this chapter. The objective of the thesis was to analyze and create a way of Service discovery of different Web services hosted in hybrid environments. A prototype has been implemented and tested according to the requirements stated in chapter 2.

The research questions stated in chapter 1 were as follows:

Ques 1: What are the services and types of the services?

Ques 2: How do service discovery standards for different kinds of Web services work?

Ques 3: How do service discovery methods work for different environments?

Ques 4: Is it possible to create a common platform to provide support for service discovery in hybrid environments?

The first part of this thesis focused on answering research questions 1 through 3. In chapter 2, the technological background was discussed. SOAP and REST Web services were discussed in detail. Service Discovery mechanisms for these two kinds of Web services which are currently used were also discussed in chapter 2.

Another important focus of this thesis was the Hosted environment. A Web services hosted environment can be local or on a Cloud platform. As the growth curve of Web Services technology is increasing rapidly, Cloud environments become increasingly popular for hosting Web services. Cloud environments are much cheaper and easier to get and start up with compared to traditional hosting. Cloud services can be private, public and hybrid in manner.

Based on research question 4 a depth analysis of currently available open standard Service Discovery Mechanisms has been done in this thesis. The use case of this thesis was defined in table 9 in chapter 4.

Zookeeper, Doozer, Etcd, AirBnB SmartStack, Netflix Eureka, Serf, Spotify, SkyDNS, and Consul have been analyzed with examples and observation found that the above four scenarios are not possible by one single discovery mechanism without tailoring. During that

research one more finding is that most of the Web service discovery mechanisms are not in general purpose use. So in this thesis, a prototype has been implemented with Consul Service discovery mechanism and Docker, thus tailoring Consul to support service discovery in hybrid environments.

Findings & Limitations

The application works as intended. The implementation of the thesis focused only on SOAP and REST Web services. So, the most important finding of the thesis is that the common platform is possible when the Service end points are not nested. If the Web services are too big and Service has many endpoints, then customization is necessary to handle this kind of scenario.

The Consul Service discovery mechanism is suitable for hybrid environments because it has some key feature such as a key-value store, load balancing, multi data center support, and health check. It can be used as a common platform by modifying it with some adapters and interfaces for many endpoints.

Contribution

The contribution of this thesis is twofold. Firstly, it presents a description of enabling service discovery approaches. Secondly, it presents a proof-of-concept implementation of Service discovery in hybrid environments. In Conclusion, the application fulfilled the requirement stated, and the thesis goal was reached.

Future work

The thesis focused on a small scale proof-of-concept. The area of research is so vast, so it is not possible to understand and discover the area during this thesis due to time shortage. If the implementation can be done in big scale and testing can be done with more resources, it will be more fruitful. One of the limitations of the implementation is that it is not fully automated for maintaining platforms. The Client interface is also simple, providing only basic functionality. More functionality can be provided with the interface.

Bibliography

[1] OASIS, "Reference Model for Service Oriented Architecture," 18 February 2006. [Online]. Available: https://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf. [Accessed 8 November 2016].

[2] W3C Recommendation, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," W3C Recommendation, 27 April 2007. [Online]. Available: https://www.w3.org/TR/soap12/. [Accessed 14 Feb 2017].

[3] Roy Thomas Fielding, "Architectural styles and the design of network-based software architectures," UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[4] D. Nicolini, "Choosing the methodology and Methods. Making decisions about your study," 2017.

[5] C. Anderson, Interviewee, *James Philips on Service Discovery.* [Interview]. 2016.

[6] M. Shaw, "What Makes Good Research in Software Engineering?," *International Journal of Software Tools for Technology Transfer,* vol. 4, no. 1, pp. 1-7, 2002.

[7] S. Denning, The Spring Board, Boston, London: Butterwhort Heinmenn, 2000.

[8] R. K. Yin, "Case Study Research: Design and Methods. Applied Social Research Methods," SAGE Publications, Inc., 2009.

[9] W. J, "Essential of Business Research, A Guide to Doing Your Research Project," England, London, Sage Publications, 2010.

[10] S. Vinoski, "REST Eye for the SOA Guy," *IEEE Internet Computing,* vol. 11, no. 11, pp. 82-84, January 2007.

[11] Nicolai M. Josuttis, SOA in Practice, Sebastopol: O'Reilly Media Inc, 2007.

[12] Oracle, "The Java EE 6 Tutorial," Oracle, 2013. [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/gijvh.html. [Accessed 15 June 2017].

[13] N. Serrano, J. Hernates and G. Gallardo, "Service-Oriented Architecture and Legacy Systems," *IEEE Software,* vol. 31, no. 5, pp. 15-19, September/October 2014.

[14] S. Mumbaikar and P. Padiya, "Web Services Based on SOAP and REST Principles," *International Journal of Scientific and Research Publications,* vol. 3, no. 5, pp. 1-4, 2013.

[15] Dr. Hervery M. Deitel et al., WEB SERVICES -A Technical Introduction, New Jersey: DEITEL Developer SERIES, 2003.

[16] H. Zhao and P. Doshi, "Towards Automated Restful Web Service Composition," in *IEEE International Conference 2009*, Los Angeles, CA, USA, 2009.

[17] A. Feda and K. Moessner, "Providing SOAP Web Services and RESTful Web Services from Mobile Hosts," in *2010 fifth International Conference on the Internet and Web Applications and Services*, Barcelona, 2010.

[18] Smartbear, "SWAGGER - The Worlds Most Popular Framework for API's," SMARTBEAR, 2017. [Online]. Available: https://swagger.io/. [Accessed 22 July 2017].

[19] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," vol. 800, no. 145, September 2011.

[20] D. S. Linthicum, Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide, Addison-Wesley Professional ©2009, 2009.

[21] OASIS, "UDDI XML ORG," OASIS, 04 January 2007. [Online]. Available: http://uddi.xml.org/. [Accessed 12 December 2016].

[22] CodeGear, "Web Services Protocol Stack," CodeGear, 2008. [Online]. Available: http://docs.embarcadero.com/products/rad_studio/radstudio2007/RS2007_helpupdates/HUpdate3/EN/html/devnet/webservicesprotocol_xml.html#4C6179657273206F662074686520576562205365727669636573205072F746F636F6C20537461636B. [Accessed 22 July 2017].

[23] OASIS, "OASIS Web Services Discovery and Web Services Devices Profile (WS-DD) TC," OASIS, 7 May 2009. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-dd. [Accessed 24 March 2017].

[24] OASIS, "ebXML," OASIS, 2006. [Online]. Available: http://www.ebxml.org/specs/index.htm. [Accessed 12 December 2016].

[25] C. Richardson, "Service Discovery in a Microservices Architecture," NGINX Software Inc, 12 October 2012. [Online]. Available: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. [Accessed 24 February 2017].

[26] M. Suchithra and M. Ramakrishnan, "A Survey on Different Web Service Discovery Techniques," in *Indian Journal of Science and Technology*, 2015.

[27] F. T. Johnsen, "Pervasive Web Services Discovery and Invocation in Military Networks," FFI Report Number 2011/00257, Kjeller, 2011.

[28] A. Thuen, "Federated Service Discovery -Interconnecting different Web Service Discovery Mechanisms," University of Oslo, Oslo, 2015.

[29] Apache ZooKeeper, "Apache ZooKeeper - Home," Apache, 20 July 2016. [Online]. Available: http://zookeeper.apache.org/. [Accessed 2 April 2017].

[30] Apache Curator, "Apache Zookeeper," Apache, 10 August 2014. [Online]. Available: http://zookeeper.apache.org/. [Accessed 8 March 2017].

[31] B. Reed, "Apache’s Wiki page of a Zab documentation," Apache, January 2012. [Online]. Available: https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab1.0. [Accessed 2 April 2017].

[32] M. B and K. Rarick, "Doozer," Doozer, 2010. [Online]. Available: https://github.com/ha/doozerd. [Accessed 13 February 2017].

[33] etcd, "etcd," Coreos, December 2013. [Online]. Available: https://coreos.com/etcd/. [Accessed 13 February 2017].

[34] D. Ongero and J. Ousterhout, "The Raft Consensus Algorithm," March 2015. [Online]. Available: https://raft.github.io. [Accessed 10 April 2017].

[35] S. Serebryany and I. Rhoads, "SmartStack: Service discovery in the cloud," Airbnb, 23 October 2013. [Online]. Available: https://medium.com/airbnb-engineering/smartstack-service-discovery-in-the-cloud-4b8a080de619. [Accessed 13 February 2017].

[36] Airbnb, "Nerve," Airbnb, 2013. [Online]. Available: https://github.com/airbnb/synapse. [Accessed 13 February 2017].

[37] Airbnb, "Synapse," Airbnb, 2013. [Online]. Available: https://github.com/airbnb/synapse. [Accessed 2017 February 13].

[38] HAProxy, "HAProxy," HAProxy, 2002. [Online]. Available: www.haproxy.org. [Accessed February 2017].

[39] C. Quinn, "Eureka," Netflix, 23 May 2013. [Online]. Available: https://github.com/Netflix/eureka. [Accessed 26 January 2017].

[40] NSQ, "NSQ v1.0.0-compat," Bitly, 08 November 2012. [Online]. Available: nsq.io/overview/design.html. [Accessed 26 January 2017].

[41] Hashicorp, "About Serf," Hashicorp, 2013. [Online]. Available: https://www.serf.io/intro/index.html. [Accessed 12 May 2017].

[42] A. Das, A. Gupta and A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol," 2005.

[43] B. Ketelsen, "skynetservices/skydns," SkyNet, March 2014. [Online]. Available: https://github.com/skynetservices/skydns1. [Accessed 18 February 2017].

[44] J. Simpson, "DI2E Runtime Service Discovery," 31 March 2015. [Online]. Available: www.argo.ws. [Accessed 8 March 2017].

[45] Apple, "About Bonjour," Apple, 23 April 2013. [Online]. Available: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/NetServices/Introduction.html. [Accessed 12 March 2017].

[46] HashiCorp, "Introduction to Consul," Hashicorp, May 2017. [Online]. Available: https://www.consul.io/intro/index.html. [Accessed 20 April 2017].

[47] P. J. Denning, "The Profession of IT : Design Thinking," *Communications of the ACM,* vol. 56, no. 12, pp. 29-31, 2013.

[48] HashiCorp, "Hashicorp," 2017. [Online]. Available: https://www.hashicorp.com/. [Accessed 12 April 2017].

[49] Docker, "Build , Ship and Run Any App in Anywhere," Docker, 2017. [Online]. Available: https://www.docker.com/. [Accessed 14 March 2017].

[50] A. Parecki, "OAUTH," AUTH, 7 September 2007. [Online]. Available: https://oauth.net/. [Accessed 25 June 2017].

[51] OpenID connect, "OpenID Connect," ÒpenID, 2017. [Online]. Available: http://openid.net/connect/. [Accessed 24 June 2017].

[52] N. Sakimura, J. Bradley, M. B. Jones and E. Jay, "OpenID Connect discovery 1.0," OpenID Foundation, 8 November 2014. [Online]. Available: http://openid.net/specs/openid-connect-discovery-1_0.html. [Accessed 14 June 2017].

[53] N. Sakimura, J. Bradley, M. B. Jones and E. Jay, "OpenID Connect Dynamic Client Registration 1,0," OpenID connect Foundation, 8 November 2014. [Online]. Available: http://openid.net/specs/openid-connect-registration-1_0.html. [Accessed 14 June 2017].

[54] QOS.ch, "SLF4J user manual," The Apache Software Foundation, 2004. [Online]. Available: https://www.slf4j.org/manual.html. [Accessed 18 April 2017].

[55] T. R. Devi, "Importance of Testing in Software Development," *International Journal of Scientific & Engineering Research,* vol. 3, no. 5, pp. 1-5, 2002.

[56] S. W. Ambler, "Introduction to Test Driven Development," Ambysoft Inc. , 2012. [Online]. Available: http://agiledata.org/essays/tdd.html. [Accessed 15 June 2017].

[57] Docker, "Docker Overview," Docker, 2015. [Online]. Available: https://www.docker.com/what-docker. [Accessed 12 June 2017].

[58] C. Bettstetter and C. Renner, "A Comparison Of Service Discovery Protocols And Implementation Of The Service Location Protocol," in *In Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications.*, Munich, Germany, 2000.

[59] S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamura, P. Frementle, D. Konig and C. Zentner, Building Web Services with Java, Sams Publishing, 2005.

[60] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner and P. R.young, "Computing As A Discipline," *Communication of the ACM (Jan 1989),* vol. 32, no. 1, pp. 9-23, 1989.

[61] F. Chen, X. Bai and B. Liu, "Efficient Service Discovery for Cloud Computing Environments," in *Advanced Research on Computer Science and Information Engineering*, Springer, Berlin, Heidelberg, 2011.

[62] N. Levina, "Grounded Theory : Philosophy, Myths and Beauty of it!!," 2017.

[63] "InfoQ," [Online]. Available: http://www.innoq.com/soa/ws-standards/poster/innoQ%20WS-Standards%20Poster%202007-02.pdf.

[64] T. Dybå, R. Prikladnicki, J. Sillito and K. Rönkkö, "Qualitative Research in Software Engineering," vol. 16, no. 4, pp. 425-429, August 2011.

[65] P. Wiliams, "Restful Service Discovery and Description," 22 January 2008. [Online]. Available: http://barelyenough.org/blog/2008/01/restful-service-discovery-and-description/. [Accessed 22 January 2017].

[66] L. Richardson and S. Ruby, Restful Web Services, O'Reilly Media, 2008, p. 448.

[67] C. Pautasso, O. Zimmermann and F. Leymann, "RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision," in *WWW 2008/ Refereed Track : Web Engineering- Web Service Deployment*, Beijing, 2008.

[68] M. R. Brenner and M. R. Unmehopa, "Service-Oriented Architecture and Web Services Penetration in Next-Generation Networks," *Bell Labs Technical Journal,* vol. 12, no. 2, pp. 147-160, Summer 2007.

[69] J. P. Lawler and H. Howell-Barber, Service-Oriented Architecture SOA Strategy, Methodology and Technology, New York: Auerbach Publications, 2008.

[70] W.-T. Tsai, X. Sun and J. Balasooriya, "Service-Oriented Cloud Computing Architecture," in *2010 Seventh International Conference on Information Technology*, Las Vegas, 2010.

[71] P. Runeson and M. Höst, "Springerlink.com," *Empirical Software Engineering,* 19 December 2008.

[72] F. T. Johnsen, J. Flathagen, T. Gagnes, R. Haakseth and T. Hafsøe, "Web Service and Service Discovery," FFI-2008/01064, Lillestrøm, 2008.

[73] S. Pakari, E. Kheirkhah and M. Jalali, "Web Service Discovery Methods And Techniques: A Review," *International Journal of Computer Science, Engineering and Information Technology (IJCSEIT),* vol. 4, no. 2, 2014.

[74] S. H. Vossen and C. L. Gottfried, "Web Service Discovery -Reality Check 2.0," in *Third International Conference on Next Generation Web Services Practices*, 2007.

[75] Y. Lafon, "Web Services Activity," W3C, 18 May 2011. [Online]. Available: https://www.w3.org/2002/ws/. [Accessed 15 January 2017].

[76] F. T. Johnsen, T. Hafsøe, A. Eggen, C. Griwodz and P. Halvorsen, "Web Services Discovery across Heterogeneous Military Networks," *IEEE Communications Magazine,* vol. 48, no. 10, pp. 84-90, October 2010.

[77] "Wkipedia," [Online]. Available: www.wikipedia.com.

[78] Margunn Aanestad, "Qualitative Research Methods Applied to Information Infrastructures," University of Oslo, Oslo, 2017.

[79] B. Bygstad, "Competing Paradigms in Qualitative Research in Information Systems," University of Oslo, Oslo, 2017.

[80] D. Nicolini, "What is Qualitative Research," University of Oslo, Oslo, 2017.

[81] Amazon Web Services, "AWS Deploy," AWS, March 2015. [Online]. Available: https://d0.awsstatic.com/whitepapers/overview-of-deployment-options-on-aws.pdf. [Accessed 8 February 2017].

[82] D. F. Birks, W. Fernandez, N. Levina and S. Nasirin, "Grounded theory method in Information systems research: Its nature, diversity and opportunities," 2013.

List of Tables

Table 1: Type of Research questions 15
Table 2: Comparison of SOAP and RESTful Web services 23
Table 3: According to NIST definition Cloud deployment models 29
Table 4: Provides Comparison on Web service Discovery techniques 36
Table 5: Overview of Open source Service Discovery tools 48
Table 6: List of primary requirements 49
Table 7: Overview of Open source discovery tools by context of thesis 50
Table 8: List of System requirement specification 51
Table 9: Primary Use cases of application 69
Table 10: Test case for User Interface 70
Table 11: Test case for Server Connector Interface 71

List of Figures

Figure 1: SOA Architecture 18
Figure 2: Cloud Computing Components 26
Figure 3: An overall view of the project design 54
Figure 4: Overview of the Web Service Discovery mechanism 55
Figure 5: Basic flow model of Web service Discovery mechanism 57
Figure 6: Client- Application request workflow 58
Figure 7: ServerConnector is the interface between Servers and Integrator 60
Figure 8: Server Connector workflow diagram 61
Figure 9: User interface mock 63
Figure 10: WebService class 64
Figure 11: Integrator Consul with Docker 65
Figure 12: ServerConfig file. 66
Figure 13: Sample Test code 69
Figure 14: Service Information page 70
Figure 15: Consul Started 71
Figure 16: Amazon Beanstalk set up 72
Figure 17: AWS BeanStalk console 73
Figure 18: Windows machine configuration information 74
Figure 19: Linux Machine configuration 74
Figure 20: Ubuntu configuration for Application setup 75

Glossary

AMQP Advanced Message Queue Protocol

AP Available Pattern

API – Application Program Interface

B2B Business to Business

B2C – Business to Client

CP Consistent Pattern

DNS Domain Name Server

EbXML Electronic Business using Extensible Markup Language

EC Elastic Computing

ECS Elastic Computing Service

ELB Elastic Load Balancer

FTP File Transfer Protocol

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

JVM Java Virtual Machine

JSON JavaScript Object Notation

LAN Local Area Network

MIME Multi-Purpose Internet Mail Extension

MVC Model View Controller

OASIS Advancing Open Standard for Information Security

QOS Quality of Service

REST Representational State Transfer

RMI Remote Method Invocation

RPC Remote Procedure Call

SMTP Simple Mail Transfer Protocol

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

TCP/IP Transport Control Protocol/ Internet Protocol

TDD Test Driven Development

UDDI Universal Data Description Integration

UML Unified Modeling Language

URL Uniform Resource Locator

WAN Wide Area Network

W3C World Wide Web Consortium

WS-D Web Service Discovery

WSDL Web Service Definition Language

XML Extensible Markup Language

Appendix A

The most texts and images used in this appendix have mostly written from [46] [41] [34].

Consul

Consul [46] is a distributed service discovery key value store application designed by HashiCorp.

Consul is distributed which means it runs as a cluster of systems so that there is no single point of failure. Consul uses Serf [41] protocol to manage cluster membership, failure detection and general orchestration. Managing cluster state via Serf is only part of the picture, though: the cluster must also manage consistency via a consensus protocol which called Raft [34].

Consul is also a service discovery tool where applications can register with Consul to provide a service, such as MySQL or HTTP. Other systems can then use Consul, wither via DNS or via HTTP, to discover the providers of a specified service. Besides that service discovery functionality Consul also has health checking functionality, to ensure that the provider of a service is actually working as expected.

As a key/value store, Consul has a comparison between Consul and etcd, the distributed key/value store bundled with CoreOS Linux. Both etcd and Consul provide HTTP APIs to both store and retrieve key/value data in distributed key/value store.

Members and Agents

Consul members define as the list different agents and server modes using which a Consul cluster is deployed. Consul provides with a command line feature using which can easily list all the agents associated with Consul.

The agent is the core process of Consul which maintains information about membership, registers services, runs health checks, responds to queries etc. Any agent can be run in one of two modes: Servers or Client. These two modes can be used according to their role as decided while using Consul.

The Consul agent helps by providing us information, which is listed below:

- Node name – which is the hostname of the machine.
- Datacenter – the data center in which the agent is configured to run. Each node must be configured to report to its data center.
- Server – It indicates whether the agent is running in server or client mode. Server nodes participate in the consensus quorum, storing cluster state and handling queries.
- Client Address – It is the address used for client interfaces by the agent. It includes the ports for the HTTP, DNS, and RPC interfaces.
- Cluster Address – It is the address and the set of ports used for communication between Consul Agents in a cluster. This address must be reachable by all other nodes.

How Consul works

The architecture diagram for Consul working in one data center can be best described as shown in figure A1 below –

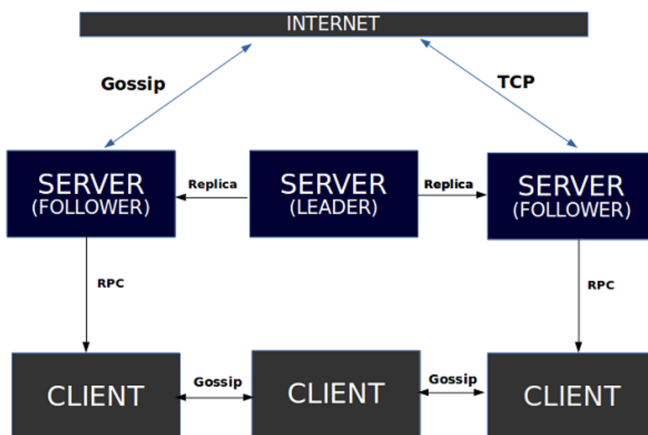


Figure A 1: Consul Architecture

As from figure A 1, there are three different servers, which are managed by Consul. The working architecture works by the using raft algorithm, which helps us in electing a leader out of the three different servers. These servers are then labeled according to the tags such as Leader and Follower. As the name suggests, the follower is responsible for following the decisions of the leader. All these three servers are further connected with each other for any communication.

Each server interacts with its own client using the concept of RPC. The Communication between the Clients is possible due to Gossip Protocol as mentioned below.

The Communication with the internet facility can be made available using TCP or gossip method of communication. This communication is in direct contact with any of the three servers.

Raft Algorithm

The Raft [34] is a consensus algorithm for managing a replicated log. It relies on the principle of CAP Theorem, which states that in the presence of a network partition, one has to choose between consistency and availability. Not all the three fundamentals of the CAP Theorem can be achieved at any given point of time. One has to tradeoff for any two of them at best.

A Raft Cluster contains several servers, usually in the odd number count. For example, if we have five servers, it will allow the system to tolerate two failures. At any given time, each server is in one of the three states: Leader, Follower, or Candidate. In a normal operation, there is exactly one leader and all of the other servers are followers. These followers are in a passive state, i.e. they issue no requests on their own, but simply respond to requests from leaders and the candidate. The following figure A 2 describes the workflow model using which the Raft algorithm works −

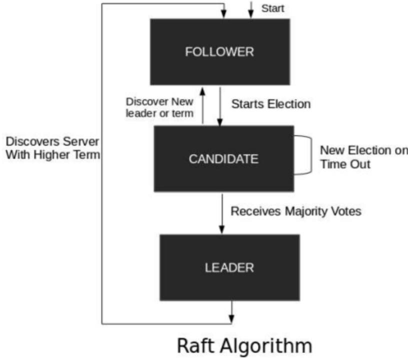


Figure A 2: Raft Algorithm

Key Value Data

Since the Consul's version 0.7.1, there has been an introduction of separate key value data. The KV command is used to interact with the Consul's key-value store via the command line. It exposes top-level commands for Inserting, Updating, Reading and Deleting from the store. To get the Key/Value object store, we call the KV method available for the consul client –

```
kv := consul.KV()
```

The KVPair Structure is used to represent a single key/value entry. We can view the structure of Consul KV Pair in the following program.

```
type KVPair struct {
  Key string
  CreateIndex uint64
  ModifyIndex uint64
  LockIndex uint64
  Flags uint64
  Value []byte
  Session string
}
```

Here, the various structures mentioned in the above code can be defined as follows –

- Key – It is a slash URL name. For example – sites/1/domain.
- CreateIndex – Index number assigned when the key was first created.
- ModifyIndex – Index number assigned when the key was last updated.
- LockIndex – Index number created when a new lock acquired on the key/value entry
- Flags – It can be used by the app to set the custom value.
- Value – It is a byte array of maximum 512kb.
- Session – It can be set after creating a session object.

Types of Protocol

There are two types of protocol in Consul, which are called as −

Consensus Protocol and

Gossip Protocol

Consensus Protocol

Consensus protocol is used by Consul to provide Consistency as described by the CAP Theorem. This protocol is based on the Raft Algorithm. When implementing Consensus protocol, the Raft Algorithm is used where raft nodes are always in any one of the three states: Follower, Candidate or Leader.

Gossip Protocol

The gossip protocol can be used to manage membership, send and receive messages across the cluster. In Consul, the usage of gossip protocol occurs in two ways, WAN (Wireless Area Network) and LAN (Local Area Network). There are three known libraries, which can implement a Gossip Algorithm to discover nodes in a peer-to-peer network −

Teknik-gossip − It works with UDP and is written in Java.

Gossip-python − It utilizes the TCP stack and it is possible to share data via the constructed network as well.

Smudge − It is written in Go and uses UDP to exchange status information.

Gossip protocols have also been used for achieving and maintaining a distributed database consistency or with other types of data in consistent states, counting the number of nodes in a network of unknown size, spreading news robustly, organizing nodes, etc.

Remote Procedure Calls

The RPC can be denoted as the short form for Remote Procedure Calls. It is a protocol that one program uses to request a service from another program. This protocol can be located on another computer on a network without having to acknowledge the networking details.

Appendix B

Docker

Docker [49] is an open platform for developing, shipping, and running applications. Docker enables to separate applications from infrastructure so it can deliver software quickly. Docker can manage the infrastructure in the same ways that manage applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, the delay between writing code and running it in production can significantly reduce. Texts and images used in this appendix have been taken from [57] [49].

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow running many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. This means that can run more containers on a given hardware combination than if there were using virtual machines. Docker containers can even run within host machines that are actually virtual machines.

Docker provides tooling and a platform to manage the lifecycle of containers:

Develop the application and its supporting components using containers.

The container becomes the unit for distributing and testing of the application.

After being ready, the application can deploy into a production environment, as a container or an orchestrated service. This works the same whether if the production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker Components:

Docker is composed of following four components

- Docker Client and Daemon.
- Images
- Docker registries
- Containers

How Does Docker Work?

Docker has a client-server architecture. Docker Daemon or server is responsible for all the actions that are related to containers. The daemon receives the commands from the Docker client through CLI or REST API's. Docker client can be on the same host as a daemon or it can be present on any other host.

Images are the basic building blocks of Docker. Containers are built from images. Images can be configured with applications and used as a template for creating containers. Images are organized in a layered manner. Every change in an image is added as a layer on top of it.

Docker registry is a repository for Docker images. Using Docker registry, you can build and share images with your team. A registry can be public or private. Docker Inc provides a hosted registry service called Docker Hub. It allows you to upload and download images from a central location. If your repository is public, all your images can be accessed by other Docker hub users. You can also create a private registry in Docker Hub. Docker hub acts like git, where you can build your images locally on your laptop, commit it and then can be pushed to the Docker hub.

The container is the execution environment for Docker. Containers are created from images. It is a writable layer of the image. You can package your applications in a container, commit it and make it a golden image to build more containers from it. Two or more containers can be linked together to form tiered application architecture. Containers can be started, stopped,

committed and terminated. If you terminate a container without committing it, all the changes made to the container will be lost.

How to Install Docker

At first, Docker was only available on Ubuntu. Nowadays, it is possible to deploy Docker on RHEL based systems (e.g. CentOS) and others as well.

Installation Instructions for Ubuntu

The simplest way to get Docker, other than using the pre-built application image, is to go with a 64-bit Ubuntu 14.04 VPS

Update droplet:

sudo apt-get update

sudo apt-get -y upgrade

Make sure aufs support is available:

sudo apt-get install Linux-image-extra-`uname -r`

Add docker repository key to apt-key for package verification:

sudo apt-key adv --keyserver hkp://pgp.mit.edu:80 --recv-keys 58118E89F3A912897C070ADBF76221572C52609D

Add the docker repository to Apt sources:

echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo tee /etc/apt/sources.list.d/docker.list

Update the repository with the new addition:

sudo apt-get update

Finally, download and install docker:

sudo apt-get install docker-engine

Ubuntu's default firewall (UFW: Uncomplicated Firewall) denies all forwarding traffic by default, which is needed by Docker.

Enable forwarding with UFW:

Edit UFW configuration using the nano text editor.

sudo nano /etc/default/ufw

Replace:

DEFAULT_FORWARD_POLICY="DROP"

With:

DEFAULT_FORWARD_POLICY="ACCEPT"

Press CTRL+X and approve with Y to save and close.

Finally, reload the UFW:

sudo ufw reload

Working with Images

There are many freely available images shared across docker image index and the CLI allows simple access to query the image repository and to download new ones.

Searching for a docker image:*

Usage: sudo docker search [image name]

sudo docker search ubuntu

Container:

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment.

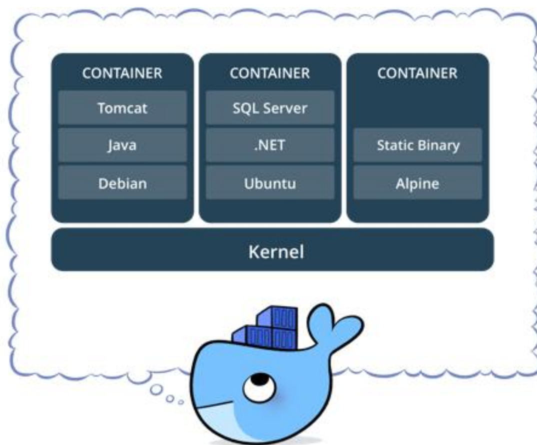


Figure B 1: Docker Container overview

Containers isolate software from its surroundings, for example differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure.

Docker for Developers

Docker automates the repetitive tasks of setting up and configuring development environments so that developers can focus on what matters: building great software. Developers using Docker don't have to install and configure complex databases nor worry about switching between incompatible language toolchain versions. When an app is dockerized, that complexity is pushed into containers that are easily built, shared and run. Onboarding a coworker to new code bases no longer means hours spent installing software and explaining setup procedures. Code that ships with Docker files is simpler to work on: Dependencies are pulled as neatly packaged Docker images and anyone with Docker and an editor installed can build and debug the app in minutes.

Docker for Ops

Docker streamlines software delivery. Develop and deploy bug fixes and new features without roadblocks. Scale applications in real time. Docker is the secret weapon for developers and IT ops teams everywhere, allowing them to build, ship, test, and deploy apps automatically, securely, and portable with no surprises. No more wikis, READMEs, long runbook documents and post-it notes with stale information. Teams using Docker know that their images work the same in development, staging, and production. New features and fixes get to customers quickly without hassle, surprises, or downtime.

For the Enterprise

Docker is at the heart of the modern app platform, bridging developer and IT, Linux and Windows. Docker works in the cloud just as well as on-premise; and supports both traditional and micro services architectures. Use Docker to build, network, secure and schedule containers and manage them from development to production. Docker sets enterprises on the path to digital transformation by enabling all apps to be agile, cloud-ready and secure at optimal costs.

Comparing Containers and Virtual Machines

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware, containers are more portable and efficient. Figure B2 shows the comparison between Docker and virtual machine set up.

CONTAINER		
App A	App B	App C
Bins/Libs	Bins/Libs	Bins/Libs
Docker		
Host OS		
Infrastructure		

VM		
App A	App B	App C
Bins/Libs	Bins/Libs	Bins/Libs
Guest OS	Guest OS	Guest OS
Hypervisor		
Infrastructure		

Figure B 2: Docker Vs Virtual Machine

Appendix C

Technical guide

This section presents the overview of the application set up and how to install and use it. There are some prerequisites for that application.

Prerequisites

The Web Discovery Mechanism implemented in JAVA programming language built in Gradle and GitHub is used for version control.

- IntelliJ IDE has been used as a framework for application development.
- Understanding of Gradle build is important.
- Docker is needed to install in the system to make functional the application
- Basic understanding of GitHub is important to work with version control.

Getting Started

The steps follow as:

1. Download the code from the GitHub repository
2. Import the project into IntelliJ IDE or Eclipse
3. Configure the project properties
4. Change the code if needed
5. Push the code to the repository

Download code

```
# Clone the repository in the workspace
 git clone https://github.com/sabrina05/INF5950.git
```

Importing the project into IntelliJ or eclipse

1. Start IntelliJ IDE or Eclipse studio
2. Click on Open projects on file system
3. Locate the application
4. Import Gradle properties