Domain Registry: a Highly Available Infrastructure for Hyperty Discovery

Rui José Gil Garcia Mangas Pereira

Thesis to obtain the Master of Science Degree in Telecommunications and Informatics Engineering

Supervisor: Prof. Ricardo Jorge Feliciano Lopes Pereira

Examination Committee

Chairperson: Prof. Fernando Mira da Silva
Supervisor: Prof. Ricardo Jorge Feliciano Lopes Pereira
Member of the Committee: Prof. Paolo Romano

October 2016
Acknowledgments

Firstly, I would like to thank my advisor, Professor Ricardo Lopes Pereira, for his help, guidance and support over the past year. I also thank him for letting me participate in the reThink project and get to work with great people.

To my family, especially my father and my sister for all the encourage and advice throughout these five years. Special thanks to my childhood friends from Ourém for all the fun and awesome moments. Thanks to all my friends from Instituto Superior Técnico for all the endless discussions about everything related to computers and for all the working hours we have spent together. It was an awesome journey.

Lisbon, October 2016

Rui Mangas Pereira
Abstract

With an increased demand for reliable and performant distributed systems, nowadays infrastructures are built with the common concern of reducing servers downtimes and eliminating single points of failure. High availability represents the quality of a system taking into account the latter considerations. Align with this, we present the Domain Registry, a core component of the European funded research project reThink. The Domain Registry is a highly available distributed system with no single points of failure that exposes a Representational State Transfer (REST) Application Programming Interface (API) that allows reThink enabled applications to register, update and delete information about what applications are running in user’s devices, and thus, allowing the communication between such users. Our approach comprises replication of application servers with traffic being distributed among them using two load balancers in a High Availability (HA) setup. Moreover, we take advantage of floating IP addresses and distributed database systems to achieve our HA infrastructure. Lastly, to ensure a healthy environment we monitor and log all Domain Registry applications and services. It allows to proactively react on possible failures, perform efficient troubleshooting and gather near real-time information about running services.

This document surveys the current state of the art in field of distributed systems, more specifically Peer to Peer (P2P) and client-server architectures. Our proposal, which is comprised by a core application and a failure resistant deployment architecture, is presented in detail, and validated through scalability and performance metrics. We show that the Domain Registry is performant and that it scales horizontally while adding more servers. Therefore, both availability and system capacity increases.

Keywords: Domain Registry, High Availability, Load Balancing, Monitoring, REST, Registry Service, reThink H2020, Logging
Resumo

Com a grande procura que se tem verificado por sistemas distribuídos fiáveis, eficientes e de larga escala, as infraestruturas de hoje em dia são construídas e pensadas de modo a reduzir o tempo em que os servidores de rede se encontram indisponíveis e ao mesmo tempo eliminar únicos pontos de falha das mesmas infraestruturas. Alta disponibilidade refere-se à qualidade de um sistema tendo em conta as considerações acima descritas. É com base nisto que apresentamos o Domain Registry, uma componente essencial de um projecto Europeu chamado reThink; o Domain Registry é uma componente de alta disponibilidade, sem únicos pontos de falha que expõe para o exterior uma Representational State Transfer (REST) Application Programming Interface (API) que faz com que aplicações que usam o reThink possam descobrir, registar e apagar informação acerca das aplicações que estão a ser usadas nos dispositivos de outros utilizadores e, como consequência disso, permitir que os mesmos possam comunicar entre si. A nossa abordagem a este problema consiste em utilizar replicação de servidores, sendo o tráfego distribuído entre eles através de um balanceador de carga. Além disso, usamos IPs flutuantes e bases de dados distribuídas para alcançar um modelo de alta disponibilidade. Finalmente, para assegurar que o sistema se comporta como é esperado, monitorizamos e guardamos registos de todas as aplicações que compõem o Domain Registry.

Este documento começa por abordar as arquitecturas Peer to Peer (P2P) e cliente servidor bem como sistemas de monitorização e gestão centralizada de logs. A nossa proposta de solução, que é composta por uma arquitectura principal e outra que tolera falhas de servidores é apresentada em detalhe e avaliada através de testes de desempenho e escalabilidade. Com este trabalho mostramos que o Domain Registry é eficiente e que escala horizontalmente com o aumento do número de servidores. Em função disso, tanto a disponibilidade como a capacidade do sistema aumentam.

**Palavras-chave:** Domain Registry, Alta disponibilidade, Balanceamento de Carga, Monitorização, REST, Registry Service, reThink H2020, Logging
# Contents

Acknowledgments iii  
Abstract v  
Resumo vii  
List of Figures xiv  
List of Tables xv  
Acronyms xvii  

1 Introduction 1  
  1.1 Motivation ........................................... 2  
  1.2 Problem Statement ................................... 4  
  1.3 Proposed Solution .................................. 4  
  1.4 Thesis structure .................................... 5  

2 Related Work 7  
  2.1 Peer-to-Peer ......................................... 7  
   2.1.1 Peer-to-Peer overlay networks ..................... 8  
   2.1.2 Tapestry ........................................ 10  
   2.1.3 Chord .......................................... 10  
   2.1.4 Content-Addressable Network ................... 13  
   2.1.5 Pastry .......................................... 14  
   2.1.6 Comparison between DHTs ....................... 15  
  2.2 Client-server Web Services ....................... 16  
   2.2.1 Simple Object Access Protocol Web services ..... 17  
   2.2.2 RESTful Web services .......................... 18  
  2.3 Directories services ................................ 20  
   2.3.1 Lightweight Directory Access Protocol .......... 21  
   2.3.2 Domain Name System ............................ 22  
  2.4 Server load balancing techniques ................. 23
# List of Figures

1.1 reThink concept ........................................... 3

2.1 Routing from the perspective of node “4227”. ........................................ 11

2.2 Object publish example. Two copies of object “4378” are published to the root “4377”.

Pointers are stored along the way. ........................................ 11

2.3 Finger tables for nodes 0, 1 and 3, and keys, 1, 2 and 6. .............................. 12

2.4 2-dimensional space with 7 nodes. ........................................ 14

2.5 Pastry routing from 65A1FC to D46A1C. ........................................ 15

2.6 Web services architecture. ........................................ 17

2.7 General structure of a Simple Object Access Protocol (SOAP) message. .......... 17

2.8 Web service usage scenario. ........................................ 18

2.9 REST Web service example scenario. ........................................ 20

2.10 Push vs Pull systems ........................................ 25

2.11 Log management as a service ........................................ 27

3.1 Relation between GUID and UserIDs ........................................ 30

3.2 Domain Registry architecture ........................................ 32

3.3 Domain Registry main architecture ........................................ 34

3.4 Load balancer failover case scenarios. ........................................ 35

3.5 Secure Sockets Layer (SSL)/Transport Layer Security (TLS) passthrough .............................. 38

3.6 SSL/TLS bridging or re-encryption ........................................ 38

3.7 SSL/TLS offloading ........................................ 38

3.8 SSL/TLS encryption ........................................ 38

3.9 Monitoring and centralized logging ........................................ 38

4.1 Interaction between internal code modules ........................................ 42

4.2 Deployment architecture ........................................ 43

4.3 Centralized logging architecture ........................................ 48

5.1 Demanded request rate ........................................ 54

5.2 Average response rate ........................................ 54

5.3 TCP connections time duration ........................................ 55
5.4 Number of errors ......................................................... 55
5.5 Average response rate with only one database node. ......................... 56
5.6 Solicited request rate with only one database node. ............................. 56
5.7 Haproxy software failover .............................................. 57
5.8 Master node failover ..................................................... 57
5.9 Riemann servers deployment ........................................... 58
5.10 Resource levels after deployment ....................................... 58
5.11 Resource levels under load ............................................. 59
5.12 Load balancer statistics ............................................... 59
List of Tables

2.1 REST architectural elements ................................................. 20
3.1 Domain Registry API specification ......................................... 33
5.1 Test scenarios ........................................................................... 50
List of Acronyms

ACID  Atomicity, Consistency, Isolation and Durability
API   Application Programming Interface
AWS   Amazon Web Services
CAN   Content-Addressable Network
CAP   Consistency, Availability, Partition Tolerance
CSP   Communication Service Provider
CSP   Communication Service Provider
CSV   Comma Separated Values
DAP   Directory Access Protocol
DHT   Distributed Hash Table
DNSSEC Domain Name System Security Extensions
DNS   Domain Name System
DSL   Domain-specific Language
HA    High Availability
HTTP  Hypertext Transfer Protocol
IaaS  Infrastructure as a Service
JAR   Java Archive
LAN   Local Area Network
LDAP  Lightweight Directory Access Protocol
MVC   Model–View–Controller
OSI   Open Systems Interconnection
OS    Operating System
OTT   Over The Top
P2P   Peer to Peer
POM   Project Object Model
POP3  Post Office Protocol
REST  Representational State Transfer
RPC   Remote Procedures Call
SMTP  Simple Mail Transfer Protocol
SOAP  Simple Object Access Protocol
SSH   Secure Shell
SSL   Secure Sockets Layer
TCP   Transmission Control Protocol
TLD   Top Level Domain
TLS   Transport Layer Security
UDDI  Universal Description, Discovery, and Integration
URL   Uniform Resource Locator
VM    Virtual Machine
VRRP  Virtual Router Redundancy Protocol
WSDL  Web Services Description Language
XML   Extensible Markup Language
Chapter 1

Introduction

High Availability (HA) clusters, also known as failover clusters, are groups of computer servers that support the development and deployment of server-side applications with minimal down times \cite{1, 2}. They operate by taking advantage of redundant computers that provide continuous operability - by restarting or rerouting work to a capable system - whenever some infrastructure component abruptly fails. This model is often associated with the process of load balancing Internet traffic across a set of servers, which has the goal of optimizing overall infrastructure metrics, such as resource usage, response time and network throughput \cite{3}. With the ever-increasing growth of networked applications, pay-per-use Cloud-based systems have been emerging to function as an almost invisible system that takes care of scaling and maintaining large systems without manual intervention from infrastructure administrators \cite{4}. However, data privacy and shared pieces of hardware between multiple users, compose the most common reasons why some organizations choose to host its servers on its own datacenters \cite{5}.

HA architectures are known to possess several servers to achieve availability and continued service when architecture components fail. As the number of servers grow, we can not, anymore, login into each individual server and look at logs or resource usage properties, such as, CPU or RAM usage. There are too many performance metrics and logs from too many applications to look at, and, on top of that, this information is distributed across several machines. While, in the past, most solutions were based on centralized log gathering, also known as pull based systems, nowadays, due to the very dynamic nature of servers and applications, decentralized log data collectors applications propose a push based model where logs are sent to centralized units for further processing. Fundamentally, it consists of a combination between decentralized log collectors and a centralized server responsible for aggregating, parsing and storing such logs.

This thesis addresses exactly those topics, and its general aim is to develop, test, monitor and deploy a HA cluster infrastructure. The ease of deploy is a critical activity that ensures that everyone can easily run and test the overall system. As a result, and since there are no issues regarding the underlying dependencies needed to run and test the system, both new features and problem solving are performed faster, and hence, time to market is decreased.
1.1 Motivation

The motivation behind this work was the emergence of the European founded research project reThink\textsuperscript{1}. The rapid growth of Over The Top (OTT) services has dramatically changed how people communicate and consume media. No longer limited to consuming content merely through a service provider’s own dedicated services, consumers are now looking for solutions over the Internet that bypasses the traditional operator’s distribution. OTT refers exactly to that: audio, video, and other services delivered through the Internet without any kind of involvement from the user’s Internet service provider. The Internet provider may be aware of the content being transmitted but does not control in any way the distribution of such content. This paradigm has been creating great opportunities for Peer to Peer (P2P) content distribution, however, all of these services and applications work in a closed ecosystem (also known as walled garden), run by corporate giants like Skype or Google’s Hangouts, whose applications restrict the involvement of another services that are not run by them. By doing so, users of said applications can only communicate with users that use the same application; as a result, it is extremely difficult for new communication applications to succeed, on a competitive, already crowded market, dominated by locked-in applications.

Aware of these problems, several European telcos and academic institutions decided to launch the reThink project, aiming to develop architectures and protocols, in order to enable an open, global, identity shared system, in which users with only one verified account may use multiple services and applications from different Communication Service Providers (CSPs). This will create the possibility of communication interoperability over the web, as is found in the telephone network and unlike the walled garden model of today’s Internet. Therefore, it will allow CSPs to provide deperimetrised services, and ultimately compete with large web companies that offer OTT content.

The reThink project describes a communication framework that handles governance, security and identity management for the registered users. The overall goal is for developers to program communication enabled applications using the reThink framework. As a consequence, this will allow users from different reThink enabled applications to communicate with each other without using the same protocols. Communication between reThink applications will be achieved through a service, deployable in a run-time environment, in an end-user device (web browser or native app) or in the network, and instantiated “on the fly”\textsuperscript{2}, called Hyperty. We make a distinction between the Hyperty itself, which is the code to be deployed, and the Hyperty instance which is the running code. This instance is usually associated with a real world entity. This entity can be a human being that makes use of a Hyperty to interact with other users via their Hyperties in a real-time communication session. This service will allow different applications to communicate with each other without having any common protocols and architectures. These Hyperties are maintained by CSPs and are loaded to the user device. In the example depicted in Figure 1.1, each user has loaded one Hyperty from each CSP, and thus, is available for incoming communication calls using the services of any of them. If the user decides to end the services of any CSP, the Hyperty instance is terminated. The Catalogue is analogue to an application store from which

\textsuperscript{1}https://rethink-project.eu/
the user’s runtime may download Hyperties. The Domain Registry is described next.

In order for users to discover one another, the reThink framework also includes the Registry Service. This service will allow a user to discover which Hyperties from which CSP a user currently has registered and instantiated on his devices. After this discovery, the CSP is contacted and, as explained above, the Hyperty is downloaded from the respective CSP catalogue (Figure 1.1). The Registry Service must be a single service available world-wide. Because a single CSP can not individually maintain this service, the Registry Service is split into two components: one that will provide the mapping between a single, global user identifier and a set of domain-dependent identifiers, and secondly, a service that resolves domain-dependent identifiers to the actual information about this user’s Hyperties instances. The objective of this thesis is the development and evaluation of the second component, called Domain Registry. The other service, called Global Registry, was developed and evaluated by other reThink researchers. The Domain Registry is a central repository that contains the information necessary to reach a Hyperty instance. Thus, information about Hyperty instances, are registered, updated and deleted from the Domain Registry. All the information required to start a connection is published here and is removed when the Hyperty instance is terminated. If the details on how to reach a Hyperty change during runtime, that information is updated automatically and seamless. This makes the Domain Registry a live directory of users available to start and receive communications.

Figure 1.1: reThink concept
1.2 Problem Statement

The Domain Registry is a critical component of reThink. It can be seen as a directory service that facilitates management and lookup of the Hyperties that are being run on user’s devices. If the Domain Registry becomes unavailable, users cannot find another user’s Hyperties, and therefore a communication cannot be established. Our goal and contribution to the reThink project is to develop the Domain Registry as a highly available distributed system that can tolerate network and hardware failures while serving users requests. We need to take into account that if this service ever stops working, the reThink framework becomes unavailable and the users unreachable. This is the service that provides the mapping between the identifier for each Hyperty instance (a Hyperty is used by a user in one or more devices) and the data that characterizes it. Since the Domain Registry implements the mapping between a user’s domain-dependent identifier and a set of Hyperty instances, it is a service that will be deployed and manage by each CSP. Although, the Domain Registry is a critical service in terms of call establishment, it is also a critical service in the sense that it will be used by CSPs that probably may have hundreds, thousands, or even millions of users. As a consequence, and despite this, it should provide low access times and be capable of fast updates (e.g. for when a device changes IP address). Moreover, the Domain Registry should be a distributed system easily scalable as needed, matching the CSP growth and requirements. The identifiers for each Hyperty instance should be a string and there should be some flexibility/transparency about the data that is stored. The stored data should include reachability information and a description of the Hyperty used, namely though a link to the Catalog Service.

As we are addressing a highly available distributed system that encompasses various networked components, the Domain Registry should be monitored, being its behavior registered in order to allow near real-time reaction from developers and maintainers whenever failures happen or when the system is misbehaving.

1.3 Proposed Solution

This thesis presents a solution for a reThink European project’s component, called Domain Registry. We designed a highly available distributed system with no single points of failure that will be run by CSPs and will allow users to lookup information about a user’s Hyperty instances using this user’s domain identifier. The core architecture of this system is comprised by a Representational State Transfer (REST) server that exposes an Application Programming Interface (API) that can be used to register, update and delete Hyperties from the Domain Registry repository. In order to increase availability and reduce single points of failure, the REST server is replicated across several machines and the Internet traffic is distributed among them using two load balancers in a HA/failover setup. The load balancer HA setup is achieved by employing a floating IP that can be instantly moved from one machine to another in the same datacenter. Part of our highly available infrastructure is the capability of being able to immediately point this floating IP address to a redundant server that is configured to be in a always listening, passive
configuration. To persistently store the data that the Domain Registry stores, and having scalability and availability in mind, we opted for deploying a highly available database with no single point of failure that employs a P2P architecture style. It can handle large amounts of data across many servers. With this architectural design, the ones who deploy the Domain Registry can easily scale horizontally our developed architecture by simply adding more machines to the initially deployed cluster. Regarding security and data safety from non unauthorized personnel, we deployed our infrastructure allowing, if needed, Secure Sockets Layer (SSL) connections which allows us to make sure that the components that communicate with the Domain Registry are indeed communicating with it and that all traffic that enters or exists the Domain Registry is encrypted. Since both the Domain Registry and its client will be deployed internally by a CSP the need for secure connections between those two components will be a decision that has to carefully considered by each CSP; for that reason, both HTTP and HTTPS connections are allowed. Of course, in the future, if security is chosen, HTTP connections will be disabled.

Since our infrastructure includes many networked components, i.e. load balancers servers, database servers and application servers, that behave differently and produce different outputs, we added to our main architecture a second one that is responsible for receiving, aggregating and interpreting application logs and monitoring events (e.g. resource usage and total of requests performed), and showing them in near real-time to the developers who maintain the Domain Registry. With this second architecture in play, the ones who are responsible for maintaining the system, receive alerts and notifications about system’s failures and can act proactively to resolve the system’s problems and reduce the affected server’s downtime.

Our proposal uses Docker for deployment, which will allow CSPs to effortlessly deploy and test our architectures. Along with the code and the necessary configuration files, we also include a set of Dockerfiles for each Domain Registry component. From our point of view, Docker represents nowadays a major platform for building, shipping and running applications. It allows application portability across machines running Docker, and more importantly it decreases application maintenance by bundling applications and all its dependencies into a single container that can be run independently of what Operating System (OS) versions the host machine has running.

### 1.4 Thesis structure

This document describes the research and work developed and it is organized as follows:

- **Chapter 1** presents the motivation, background and proposed solution.
- **Chapter 2** describes the previous work in the field.
- **Chapter 3** describes the system requirements and architecture of the Domain Registry.
- **Chapter 4** describes the implementation of the Domain Registry and the technologies chosen.
- **Chapter 5** describes the evaluation tests performed and the corresponding results.
Chapter 6 summarizes the work developed and discusses future work.
Chapter 2

Related Work

This section provides an overview of the state of the art in the fields of P2P networks, client-server architectures, directory systems, distributed systems monitoring tools and load balancing techniques. The first part of this chapter covers the P2P paradigm in detail with a description and comparison of well known Distributed Hash Table (DHT) abstractions. Then it is explained what Web services are, followed by a detailed description of Simple Object Access Protocol (SOAP) protocol and REST architecture. Subsequently, it is explained what a directory system is and its applications, with examples of the two most popular directory systems implementations: Lightweight Directory Access Protocol (LDAP) and Domain Name System (DNS). Afterwards, centralized logging architectures, distributed systems monitoring tools and load balancing techniques are studied.

2.1 Peer-to-Peer

P2P systems can be described as decentralized distributed systems in which all nodes, having the same capabilities and responsibilities, form a topology that enables the sharing of resources (e.g. content, bandwidth, and processing power) without requiring an intermediate central authority [7]. P2P architectures are distinguished by the ability to adapt to failures, and the adaptability to accommodate transient sets of nodes, while maintaining connectivity and performance. Comparing to a client/server model, where the server is the entity in charge of most network resources, and for that reason, becomes at the same time the most important part as well as the bottleneck of the system, in P2P networks, peers are both consumers and suppliers of resources.

One important aspect of P2P networks, is the ability to exchange resources directly between peers, instead of using an intermediate component such as centralized servers. Still, some P2P systems use centralized servers to perform certain functions, such as bootstrapping (providing initial configuration to newly joining nodes) and computation of reputation ratings. Some other systems, for example Napster, use centralized servers to keep information about what users are sharing. Although not fitting the previous P2P definitions, Napster is usually considered a P2P system. The lack of any centralized component, known as decentralized architectures, requires full cooperation among all peers in tasks
that include content location, overlay management, routing, and content replication. Another property of P2P systems is the capacity to detect, deal, and adapt when changes in the underlying network occur. These changes may relate to network instability (connection failures) or with constantly entering and leaving of peers from the network (churn). Therefore, in case of failures, P2P systems must be capable to continue functioning by using other peers to route messages. Comparing with client/server models, where all functionalities of the system would stop if the server crashed, P2P networks are significantly more fault-tolerant.

Over the years, P2P architectures are being used as basis for a wide range of applications. These applications are usually categorized as distributed file systems, database systems or Internet service support systems [8]. More recently, mobile commerce applications (e.g. Tradepal\footnote{https://www.tradepal.com}) and P2P based digital cryptocurrencies (e.g. Bitcoin [9] and Peercoin [10]) were developed.

2.1.1 Peer-to-Peer overlay networks

As mentioned before, a P2P network relies on the successful connection between nodes for its operation. These connections between peers form a network on top of a physical network, typically the Internet, referred to as an overlay network.

A P2P overlay network is defined by the topology structure, degree of centralization, and, the routing and location mechanisms it uses for messages and content. These three properties are directly connected to how well the system perform, as they affect scalability, load balancing, and fault tolerance. Scalability is the ability of a system to keep its performance as it grows in number of participants and objects stored [11]. Due to flooding mechanisms, Gnutella [12], an early P2P system, had several scalability issues with the growth of signalling traffic. Load balancing is an essential technique to provide fair distribution of objects between nodes, and fault tolerance refers to a property that enables a system to continue working in the event of a failure.

P2P networks are usually classified according to the existence, or not, of central authorities. Two classifications have been proposed in [7]: Hybrid and Pure P2P networks.

Hybrid networks are described as P2P systems that have some central server provide part of the offered services. By contrast, systems are characterized as pure P2P networks, if the only entities allowed are servents. In this context, the word servant means a peer that has the capability of being a client and a server.

Alternatively, according to [8], the degree of centralization is divided in purely decentralised, partially centralised and hybrid decentralised architectures. Purely decentralised architectures correspond to the exact definition of P2P, in which all nodes act both as servers and clients, without any centralized authorities coordinating their activities. Therefore, nodes are responsible for initiating connections, forwarding messages on behalf of other nodes, and replying to messages directed towards them.

Partial decentralised architectures refer to the same ideas as purely decentralised, however, some nodes assume more important roles than others, acting as indexers for files shared by peers in the proximity, or belonging to a higher level overlay. These nodes do not form single points of failures...
because they are dynamically assigned (based on some network criteria such as bandwidth), and if they fail, the network automatically replaces them.

 Hybrid decentralised architectures, as described in [7] and explained before, have some kind of central server that facilitates some procedures such as location of nodes that store certain data items. Yet, the file exchange is performed directly between peers. Obviously, these hybrid models have single points of failures, making them not scalable and vulnerable to malicious attacks.

 Based on how nodes connect to each other on the overlay network, and how resources are found, we can classify P2P networks as 

*structured* and *unstructured*. Within unstructured networks, peers are linked to each other randomly (e.g. Gnutella and Kazaa), and there is no relationship between the placement of content and the overlay topology. With this random network structure, a few issues arise. When a peer wants to find some data item, a query must be flooded through the network. Thus, signalling traffic increases, and it is not guaranteed that the data is actually found. A way to reduce the amount of messages exchanged is to define a maximum search radius i.e., to limit how far a query may be flooded and, at the same time, prevent the network from becoming overloaded with queries. However, this results in low retrieve rates, as not all the nodes potentially containing the searched content are reached. As stated before, unstructured P2P networks have poor searching efficiency, specially for rare objects. While popular content (shared by a majority of peers) would be found easily, a search for rare content would be likely to turned out unsuccessful [13].

 On the other hand, in structured P2P networks, peers are organized into a specific topology, and with specific protocols, it is ensured that a peer can efficiently locate the desired resources, even if the file/resource is very rare. These systems essentially provide a mapping between content and nodes. This mapping is achieved through a mechanism known as DHT, in which a service similar to a hash table (lookup a value by key) is provided, even though data is distributed across a network composed of an arbitrary number of connected nodes. Nodes and objects are assigned unique identifiers within an identifier namespace, and usually a node is responsible for the objects whose keys are closer to its own ID. With this functionality, any node can efficiently locate a resource with a given key. To route messages efficiently through the network, peers in structured networks must keep and maintain a list of neighbors.

 As we shall see later in this section, many P2P systems implement a DHT. Nevertheless, regardless of the system architecture, some important features need to be met in order for these systems to be successful. Some of these features are the ones described above: load balancing, scalability, and fault tolerance. As we will see in the DHT abstractions below, load balancing is achieved by choosing random identifiers for the objects and mapping them to a unique identifier, which belongs to a node that controls a portion of the identifier space [14].

 Due to the problems already described with unstructured networks, and since our work must accomplish the metrics presented in section 1.2, the focus of our research will only be related to structured networks and its applications. Therefore, in the next sections several abstractions of DHTs will be described, in a chronological way.
2.1.2 Tapestry

Tapestry [15] is a P2P overlay routing infrastructure that provides scalable, location-independent, and efficient routing of messages using only nearby resources. It is based on Plaxton et al. [16] routing mechanism for efficient propagation of objects in a network.

Tapestry uses prefix-oriented routing, and distance between two nodes is given by the number of prefix digits they have in common; for example, a node with ID “1241” is closer to the node “1246” than the node “1256”. Each node and application-specific endpoints (e.g. objects) are assigned a 160-bit ID, both generated using a hash function such as SHA-1 [17].

To deliver messages, each node \( n \) maintains a routing table, which is comprised by a set of nodes with which it communicates. This set of nodes is referred to as neighbors of \( n \). Tapestry uses local tables at each node, called neighbor maps, in order to route messages to the destination digit by digit (e.g. \( 1*** = \rightarrow 12** = \rightarrow 12A* = \rightarrow 12A4 \)). An example can be seen in the Figure 2.1. Outgoing links point to nodes with a common matching prefix. Higher levels represent more matching digits.

Every object in the network has a root node whose ID is equal or closest to the GUID of the object. This node only stores pointers which indicate where the object is stored. When objects are created at certain nodes, they are published to their root nodes and pointers are cached along the way through the destination. So, if a client wants to locate some object \( O \), it starts by routing a message to \( O \)'s root node. Since pointers are cached along the way, it is very likely that some node in the path has a location mapping for \( O \). If that doesn’t happen, the query is forward to the root, which is guaranteed to have the location. Figure 2.2 describes the publishing of an object to its root node. As we can see, object “4378” is created at two nodes, “4228” and “AA93”. These objects are published to their root node “4377” and pointers to “4228” and “AA93” are stored by nodes who reside in the path from the source through the destination.

The previous examples show a Plaxton network. No fault tolerance or churn rates are considered. When a new node \( n \) is inserted into a Tapestry network, "need-to-know" nodes are informed, \( n \) might become the root of existing objects, a new routing table for \( n \) is defined, and nodes near \( n \) are notified and may choose to use the new node as a routing optimization. When a node leaves the system, two options are considered: voluntary or non voluntary node deletion. In the first case, the leaving node informs all nodes related to it and moves the objects it maintains to a new root. In case of node failure, no warning is given to another nodes. Tapestry solves this problem by periodically sending keep-athives to detect outgoing links and node failures.

In terms of performance, Tapestry routing takes approximately \( O(\log N) \) hops, in a network with size \( N \) and IDs with base \( b \), and requires \( O(\log N) \) information about the others.

2.1.3 Chord

Chord [18] is a distributed lookup protocol that mainly addresses one problem: how to efficiently locate the node that has a certain data item. This protocol only provides one operation, mapping keys to nodes. In order to assign them, Chord uses consistent hashing [19], which is responsible for maintaining load
balancing because each node receives approximately the same number of keys. The adaptability that Chord exhibits when nodes join and leave the network is one of the major advantages of this system. Even if the system is constantly changing, Chord can answer queries. Some Chord features include: load balancing, decentralization (every node is important as any other), scalability and availability. In Chord, the DHT space is a circle, being the IDs calculated with ID \( \mod 2^m \), where \( m \) is the number of bits in the key ID. All IDs are arranged clockwise in a ascending order, and the node responsible for a key \( k \) is the successor(\( k \)), the node with ID higher or equal to \( k \).

Consistent Hashing assigns each node and key a \( m \)-bit identifier using a hash function such as SHA-1. This technique lets nodes enter and leave the system with minimal changes in the network. Only \( O(1/N) \) keys, being \( N \) the number of nodes, need to be moved to a different location. Node ID and Key ID are generated as follows:

- Node ID = hash(IP address);
- Key ID = hash(key).

Chord routing scales very well since only a small amount of routing information is needed to im-
plement consistent hashing. Each node only needs to know its successor on the circle. Queries for a specific node are passed around the circle until the destination node is found. Yet this is a very inefficient scheme considering that a query may be required to traverse all nodes. With the purpose of fixing this problem, Chord accelerates lookups by maintaining additional routing information. Thus, every node in the system manages a routing table called finger table, with at most \( m \) entries. The \( i^{th} \) entry, \( s_i \), is the successor of \( n + 2^{i-1} \), where \( 1 \leq i \leq m \). \( S_i \) is called the \( i^{th} \) finger of node \( n \). An example is provided below in Figure 2.3.

In dynamic networks, participants may join or leave at any time. However, keys need to be reached. To achieve this, each node’s successor needs to be correctly maintained. It is also desirable that finger tables remain correct. Thus, when a new node \( n \) joins the system, the predecessor and fingers of \( n \) need to be initialized. Additionally, it is also necessary to update the finger tables, successors and predecessors of existing nodes in order to reflect the new arrival to the network. Finally, values associated with keys that the new node is now responsible for need to be transferred to it. When a node fails or leave the system, some other node can lose its successor. To ease this problem, nodes maintain a list of successors. If one of the nodes from this list leaves or fails, the next one on the list is used.

Lastly, in a \( N \)-node network, each node only needs to know \( O(\log N) \) other nodes and just \( O(\log N) \) messages are needed to resolve a lookup. Updates when nodes join or leave the network, require only \( O(\log^2 N) \) messages.

![Figure 2.3: Finger tables for nodes 0, 1 and 3, and keys, 1, 2 and 6.](image)
2.1.4 Content-Addressable Network

Content-Addressable Network (CAN) [20] is a scalable, fault-tolerant and self-organizing Internet scale hash table. Each node is assigned a d-dimensional cartesian coordinate on a d-torus, and the distance between two nodes is given by the Euclidean distance in the d-dimensional hyper-cube. Over time, this dimensional space is dynamically divided among all the nodes; with each individual node maintaining its own distinct zone in the space.

Figure 2.4 illustrates a CAN network with 2 dimensions and 7 nodes. Each dimension covers [0, 1) and each node handles a zone in the grid. For example, node 1 maintains the zone (0.5-0.75,0.5-0.75) and node 6 the zone (0.25-0.5,0.75-1). Each node in the system maintains information about $2d$ neighbors, which are responsible for the neighboring zones. Here, the notion of neighbors means two zones that are adjacent along d-1 dimensions. It is important to state that $d$ is a parameter independent of the number of nodes, which means that the number of neighbors is constant, no matter how many nodes the system has.

Routing in CAN is very intuitive, and works by following the path from one node to another. Many routes between two nodes exist. Even if one or more nodes crash, a node would automatically find another path. Each node maintains a routing table containing the IP addresses and coordinates for each of its neighbors. Thus, using this coordinate system, a node sends messages to the neighbor that is closer to the destination. For instance, Figure 2.4 shows a routing path from node 1 to point (x,y). The dashed line illustrates the path taken from source (node 1) to the destination’s point (x,y).

In order to store data in CAN, \{key, value\} pairs are mapped onto a point $P$ using a hash function on the key. Then, the \{key, value\} pair is stored at the node that owns that specific zone. As an example, in Figure 2.4 if a pair was mapped to a point $P$ with coordinates (0.45,0.20), the node responsible for storing that pair would be node 7. Similarly, to retrieve a value $v$ for a key $k$, a node should first obtain the point $P$ by mapping the key $k$, and then retrieve the value $v$ from the node that maintains $P$.

Since we’re dealing with dynamic networks, entries and departures of nodes need to be considered. CAN addresses these problems in a more complex way than Chord, because the d-dimensional CAN’s structure is more complex than Chord’s one dimensional network. When a new node enters the network, a portion of space must be assigned to it. This is done by splitting an existing zone in two; half is handed to the new node, and the other half remains with the existing node. Then, the neighbors of both nodes are notified so that new routing paths may include the new node. When a node leaves, its zone must be taken over by a remaining node.

As a means to detect node failures, CAN uses a mechanism of periodical keep-alives. When a node has not received any messages from some other node in a long time, it assumes that the node has failed and starts a takeover procedure. This procedure ensures that the zone occupied by the failed node is merged with one of its neighbors.

Some design improvements, such as multi-dimension and multi-coordinate spaces were also implemented. The first to reduce path length and the second to give each node a zone in multiple, independent coordinate spaces.
2.1.5 Pastry

Pastry [21] is a self-organizing distributed object location system for wide-area P2P applications. Every node in a Pastry network is assigned a 128-bit identifier (nodeID). This nodeID, randomly assigned when nodes join the system, is used to give a peer’s position in a circular nodeID space that ranges from 0 to $2^{128} - 1$. It is considered that these identifiers are uniformly distributed across the 128-bit nodeID space. Node and key IDs are represented as a sequence of digits with base $2^b$, with $b$ being a 128 divisor system parameter.

Every node in the network has a routing table, a neighborhood set and a leaf set. The routing table contains $\log_2 N$ rows with $2^b - 1$ entries each, and a leaf set $L$ is the group of nodes with $|L/2|$ numerically closest larger and smaller IDs, comparatively to the present node’s nodeID, where $N$ is the number of Pastry nodes in the network and $L$ a configuration parameter. The neighborhood set, not used for routing messages, comprises the IDs and IP addresses of the nodes closest to the local node. In Pastry, the node responsible for a key $k$, is the node whose nodeID is closest to $k$’s ID. As an example, in Figure 2.5 the node responsible for key D46A1C is D467C4.

The routing operation is executed every time a node receives a message and the idea is to get closer and closer to the destination. The first thing nodes do, is to verify if the leaf set contains the node closest to the key. If so, the message is passed directly to the destination node. When the key is not present in the leaf set, the routing table is used to discover for whom to forward the message. The selected node is the one whose ID shares a prefix with the key that is longer by at least one more digit. The diagram presented in Figure 2.5, illustrates the routing of a message from node 65A1FC to D46A1C. Blue dots exemplify alive nodes.

One of Pastry main features is self-organization. Thus, the network must be able to detect and adapt
when nodes join and leave the system. When a node $n$ joins, it is assumed that it already knows an active node $n'$ present in the network. The existing node $n'$ routes a special join message on behalf of node $n$ to a node $z$. All nodes between $n'$ and $z$ that receive the message, send to node $n$ their tables. With this information, node $n$ builds its own tables and informs specific nodes that have to be aware of its arrival. In case of failure or departure, keys that those nodes controlled, are reassigned to a new node whose ID is now the closest to the keys IDs. Pastry routes messages to any node in $O(\log N)$ and keeps routing tables with $O(\log N)$ entries.

![Figure 2.5: Pastry routing from 65A1FC to D46A1C.](image)

### 2.1.6 Comparison between DHTs

Chord represent the simplest network off all DHTs variations analyzed in this document. Its architecture consist of an uni-directional ring topology with nodes having links to its successors and predecessors. Also, each node maintains a *finger table* containing up to $m$ entries. CAN topology is a $d$-dimensional cartesian coordinate space on a $d$-torus. Each node handles a specific zone on the space, and it is responsible for the keys that lay in that area. Pastry and Tapestry form a one-directional topology, which can be seen as a tree structure. They assign fix-length identifiers as the nodes and data objects IDs (Tapestry use 160-bit and Pastry 128-bit). Furthermore, Pastry (like Chord) uses a ring structure when the tree is insufficient to find the target node.

Chord nodes store the following information: a predecessor node, a list of successors, and a routing table with $m$ entries (being $m$ the number of bits in the key/node IDs). Every node maintains information about $O(\log N)$ other nodes. In CAN $d$-dimensional network, each node has $2^d$ neighbors. This value is independent of the number of nodes in the system. In Pastry, each node has a routing table, a leaf set, and a neighborhood table. The routing table has approximately $\log_2 N \times (2^b - 1)$ entries. Tapestry’s nodes have a neighborhood map with multiple levels corresponding to matching prefixes. Also, Tapestry
nodes maintain links to nodes that store certain objects. These pointers exist to help reduce lookup time significantly.

Regarding churn, when nodes join or leave a Chord network, only a small number of nodes need to update their routing tables. With the stabilization procedure, Chord nodes update periodically its successors list and their finger tables. If a node can’t reach some other node, it simply selects the next node on its successors list. In CAN, zones handled by nodes are split or rebuild as nodes join and leave. Nodes only need to update their neighbors in order to maintain good routing routes. In Pastry and Tapestry, a new arriving node must learn the existence of already in the system peers and inform them of its presence. In Tapestry, nodes use keep-alives messages to verify if a node is running or not.

\section{2.2 Client-server Web Services}

The term Web services describes an architecture style for client-server application-to-application communication using existing Web protocols, such as Hypertext Transfer Protocol (HTTP) \cite{22}. Say, for example, that an individual wants to book a hotel reservation using a hotel aggregator website. To locate the best deals, this website needs to pull information from multiple sources, each of which using different and incompatible architectures and applications. Web services purpose is to simplify this process by defining a standardization mechanism to smoothly interoperate the communication of different applications that may be running on heterogeneous platforms (Figure 2.6) \cite{23, Chapter 9}. As a result, a client application in one organization can interact with another application in another organization without human supervision or administration. For this to be possible, Web services generally provide a description of the offered service, which includes among other information, the destination server’s Uniform Resource Locator (URL), the format for requests and an example response that will be generated by the service. This information is then used as basis of communication between the client and the server. In essence, a Web service makes available a collection of operations that can be used by any client application over the Internet.

Many well known platforms used by millions of people everyday, such as Twitter, Ebay or Facebook offer Web services interfaces that can be used by developers for building another applications which use the abovementioned platform’s services. A common example of this, are applications that interact with Ebay services to place bids during the last seconds of a closing auction. Although users can perform the same actions using directly a Web browser, they will never be as fast.

A variety of standards were developed to support the deployment of web services, including, the Web Services Description Language (WSDL) \cite{24}, Universal Description, Discovery, and Integration (UDDI)) \cite{25}, and SOAP \cite{26}. At the same time, REST architectures have been gained popularity by their lightweight modus operandi on how to work with Web Services. However, although being different (SOAP is a protocol and REST an architectural style) both answers to the exact same issue: how to access Web Services.

In the following topics, are described both SOAP (and its underlying standards WSDL and UDDI) and REST.
2.2.1 Simple Object Access Protocol Web services

SOAP is a communication Extensible Markup Language (XML)-based protocol for exchanging structured information (Web services messages) over the Internet. It is, in other words, a way to structure information before transmitting it over the network. SOAP works with already existing network protocols (e.g. HTTP and Simple Mail Transfer Protocol (SMTP)) and has a very simple XML structure comprising one XML element with two child elements: a body and a header. The header is an optional attribute that contains information about authentication, data encoding - that is, how to process the given message at an intermediary point or at the ultimate endpoint. The body dictates how the message should be interpreted. It is a mandatory field. Figure 2.7 describes a general SOAP message.

```
<?xml version="1.0"?>
      SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   ...
   ...
   </SOAP-ENV:Header>
   ...
   ...
   <SOAP-ENV:Body>
      ...
      ...
      <SOAP-ENV>
         ...
         ...
      </SOAP-ENV>
      ...
      ...
   </SOAP-ENV:Body>
</SOAP_ENV:Envelope>
```

Figure 2.7: General structure of a SOAP message.

SOAP is analogous to Remote Procedures Call (RPC) [27] protocol, used by programs to request services from another computer programs located in another networks without needing to comprehend its underlying details. Particularly, with RPC, procedures and functions methods can be called as if they were in the local machine's address space. SOAP supports RPC, and early on, most SOAP-based...
services used RPC. However, later on the industry switch to document-based SOAP, where documents describing the services are passed on over the network. These two styles - document and RPC based SOAP - are not related with a programming model. It merely indicates how to translate information about Web services into SOAP messages. Document-based SOAP messages indicate that the body of such messages include a XML document that can be validated against predefined schemas. On the other hand, with RPC-based SOAP messages, the message body contains a XML representation of what methods return and what arguments they expect.

SOAP offers basic communication, but does not describes how the Web service actually works or what messages should be exchanged to use it. WSDL fills this gap by providing a XML-based document that defines the functionality of the Web service. This schema describes how the remote methods work by specifying what parameters they expect and what data they return. However, something is missing. How do we find these services? Using the, platform-independent, UDDI, it is possible to easily and dynamically, describe, publish and find Web services all over the Internet.

The use of these three standards (WSDL, SOAP, and UDDI) is described next (Figure 2.8).

- **Phase 1**: An organization that offers a Web service uses WSDL to describe it and UDDI to publish it to a service repository;
- **Phase 2**: A service requester uses UDDI to find the Web service by its name or by its characteristics. As a result, the repository returns information about what Web services were found;
- **Phase 3**: Finally, the client calls the service using SOAP messages. The service is then accessed by the client with XML data being transferred over the network.

![Figure 2.8: Web service usage scenario.](image)

### 2.2.2 RESTful Web services

In 2000, Roy Thomas Fielding wrote a doctoral dissertation [28] where he describes an architecture style (opposed to a set of standards) named representational state transfer. It relies on a stateless, client-
server communication, using in all cases, the HTTP protocol, which is used for machine-to-machine communication, making REST fair more simpler than mechanisms such as SOAP or RPC. Fielding’s idea was to minimize network communications – and, as a result, latency while at the same time boosting scalability and independence of networked application’s components.

REST requires the communication between the client and the server to be stateless, such that every request from a client to a server carries all required information for the server to understand the request. This restriction leads REST to increase its scalability since the servers do not store or manage state between requests. Although, network performance may be affected as a series of related requests can contain repetitive information that can not be stored on the server. In order to solve this problem, REST introduces cache, given the client the possibility to reuse response data for later, similar requests.

REST architectural elements are divided into three classes: data elements, connectors and components. Data elements are summarized in Table 2.1. Connectors (i.e., client connectors, server connectors and cache connectors) present an interface for component communication with the following objectives: separation of concerns, increasing simplicity and hiding communication mechanisms. These properties combined, form a major advantage of this model: sustainability. Since all clients access an abstract communication endpoint, implementation changes may occur in server side without affecting the client application’s behaviour. In other words, connectors manage the network communication of components. These components are comprised by origin servers, gateways, proxies and user agents. Each component implement one or more connector type. The user agent, e.g., a Web browser, uses a client connector to make requests and becomes the receiver of requests. Origin servers implement a server connector to receive and process requests. Intermediary components (proxies and gateways used to improve performance) act as both client and server in order to forward requests and responses. Finally, REST components perform actions on resources by transferring resource’s representations (e.g. JSON or XML documents) between components.

REST Web-service example

To better understand REST architectural style a small example was developed. This service provides the functionality to create or read information about an individual movie or a set of them. In this example, resources will be movies, and the representation of these will be JSON. Regarding addressing, we will have: http://some.domain.com/movies/:movie_id and http://some.domain.com/movies. Note that only resources are addressable, not representations. HTTP [22] defines a set of methods to indicate desired actions to be performed on the server. We will use PUT (to create and update), GET and DELETE. PUT method is used to create (if the resource does not exists yet) or update a movie with a given identifier and is carried out on http://some.domain.com/movies/:movie_id. It returns 201 Created code response if the request has succeed or 400 Bad Request for an unsuccessful PUT. DELETE is used to delete a movie from the list and is applied to http://some.domain.com/movies/:movie_id. If the server successfully deletes the resource it returns 202 Accepted. Otherwise, as in PUT, 400 Bad Request is returned. Finally, the GET method can be used in either of the previously defined URLs. If it uses http://some.domain.com/movies/:movie_id, the movie identified by that identifier is returned.
### Table 2.1: REST architectural elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>intended target, e.g. A movie title.</td>
<td>A resource is anything that can be named, e.g., a document image or non-virtual object</td>
</tr>
<tr>
<td>Resource identifier</td>
<td>URL or URN, e.g. <a href="http://some.domain.com/movies/1">http://some.domain.com/movies/1</a></td>
<td>Identifier to identity resources involved in a interaction between two components.</td>
</tr>
<tr>
<td>Representation</td>
<td>JPEG image, HTML document</td>
<td>Sequence of bytes. Something that is sent back and forth between servers and clients. It is the current or the intended state of the resource.</td>
</tr>
<tr>
<td>Representation metadata</td>
<td>last-modified type</td>
<td>Describes the representation. Usually used to verify message integrity.</td>
</tr>
<tr>
<td>Resource metadata</td>
<td>source link</td>
<td>Describes the resource.</td>
</tr>
<tr>
<td>Control data</td>
<td>cache control</td>
<td>This defines the goal of a message between components, such as the action being seek or the meaning of the response.</td>
</tr>
</tbody>
</table>

Otherwise, requesting the other URL, and since no identifier is passed on, all movies are returned. In both cases, **200 OK** is returned if the resource was found or **404 NOT FOUND** in case of error. In all the previous cases, movies’s representations - JSON elements containing the intended state of a specific movie resource - are returned.

![Figure 2.9: REST Web service example scenario.](image)

#### 2.3 Directories services

A directory is a service somewhat similar to a database, although it is accessed (read or queried) much more often than it is written. Inside these directories resides information that describes network resources (e.g. printers), with the purpose of being queried by users to find certain objects within a network. For example, a directory can be searched to find a person’s email address or the location of a printer. The term *yellow pages* can be seen as an analogy used to describe how directories services work. If a person's name is known, its characteristics (e.g. city, phone number) can be retrieved. If the name was not found, the directory can be queried for a list of objects that meet a specific requirement.

There are three independent concepts that characterize a directory: scope of information (local or...
global, location of clients, and distribution of servers. The clients that access the directory can be local or remote: local clients reside within the organization or on the same Local Area Network (LAN); remote clients might be distributed across the globe. Regarding the server distributing, a directory may be centralized or distributed. If a directory is distributed, the data stored is shared across many machines that coordinate among themselves to provide the directory itself. The information stored can be local or global, depending on what it describes. If we were thinking about a company, local information may describe objects inside a department or workgroup, and global information would probably describe objects belonging to the entire company [29]. An example of a directory service is DNS, wherein each DNS server store mappings of domain names to their respective IP addresses [30]. In the next two sections, will be described two directory services implementations: LDAP and DNS.

2.3.1 Lightweight Directory Access Protocol

LDAP is an asynchronous, client-server, message based protocol. It is a standard that defines methods and protocols for read and update information contained in a directory. It defines the communication protocol, by specifying the format of the messages exchanged in each interaction with a directory service [31]. A client may issue multiple requests, and responses to those requests may be received in a different order.

The LDAP protocol was developed in 1993. Its main goal was to supersede Directory Access Protocol (DAP), which was used to access X.500 directories [32]. A X.500 directory organizes data in a hierarchical namespace capable of dealing with large amounts of data. The problem with DAP was that it required the client and the server to communicate using the Open Systems Interconnection (Open Systems Interconnection (OSI)) protocol stack. Alternatively, LDAP was intended to be a lightweight alternative to access X.500 directories using the simpler Transmission Control Protocol (TCP)/IP protocol stack.

As stated before, LDAP specifies how data is accessed, and not how data is stored. To make it clear, a database provides LDAP access to itself, not the other way around. The client should never see how the backend mechanism is implemented. Thus, LDAP specifies operations such as:

- Searching for entries;
- Adding an entry;
- Deleting an entry;
- Updating an entry.

The communication between an LDAP client and an LDAP server is performed in four steps [29]:

1. A connection between the client and the server is established. This process is usually known as a binding to the server.
2. Client authenticates itself or uses default access rights. A session with stronger security measures (data encryption) can also be established.

21
3. Client then executes actions on the directory data. LDAP offers the operations described above.

4. The session is closed (unbinding).

The use of P2P technology for building a directory system has been proposed, in order to overcome the centralized properties of LDAP [33]. It would consist of all entities working together to provide information about their set of objects to each other. As a result, clients would not access the server for information. Instead, all that information would be replicated across cooperative clients. This service would offer better scalability, self-management, and lower administrative overhead.

### 2.3.2 Domain Name System

Internet hosts can be identified in many ways. Hostnames such as, www.ist.utl.pt or www.google.com, are easily remembered by users and therefore appreciated by them. However, hostnames provide very little information about the location of the hosts. Usually, and depending on the host name, only the country that hosts the domain is known. Hence, for systems to locate each other in a distributed environment, a uniquely identifier is needed to identify a particular host on the Internet. Therefore, besides names host are also identified by IP addresses.

Since routers only deal with IP addresses, a directory service that maps hostnames onto IP addresses is needed. This is the main task of DNS. DNS is a distributed database implemented in a hierarchy of DNS servers, and a protocol that allows queries to be made to this database.

We now present an overview of how DNS works. Suppose that some application (e.g. email reader) needs to translate a hostname to an IP address. A local DNS resolver creates a query and sends it to the name server(s) listed in the local computer's TCP/IP settings. After a short period of time (milliseconds to seconds), DNS in user's local machine receives the reply message that provides the mapping. The map is then sent to the invoking application, which treats DNS as a black box. However, the DNS architecture is complex, consisting of DNS servers distributed across the globe. The DNS database is distributed, meaning that no single DNS server maintains all the mappings for all hosts in the Internet. The mappings are distributed across DNS servers. There are three types of DNS servers: root servers, Top Level Domain (TLD) servers, and authoritative servers. Let's suppose that a DNS client wants to determine the IP address for the hostname www.youtube.com. The next events will take place. Root servers are the ones contacted first and return IP addresses for TLD servers for the top level domain com. The client then contacts one of these TLD servers, which returns an IP address of an authoritative server for youtube.com. Finally, the authoritative server is contacted for youtube.com, which returns the IP address for www.youtube.com. Nevertheless, if the domain has several subdomains (e.g. www.tagus.ist.utl.pt), a few more steps are required in order to resolve each subdomain.

In order to enhance performance, DNS provides caching, aiming to reduce the number of DNS messages passed around the Internet. The idea is simple: every time a DNS server receives a DNS reply, it caches the mapping in its local memory. When a query for a cached hostname is received, the DNS server can provide the mapping, even if it is not authoritative for that specific hostname.
Regarding security, the Domain Name System Security Extensions (DNSSEC) [34] is a set of security extensions to DNS that provide the means for protecting and authenticating DNS records - in other words, allow applications to validate the data received. DNSSEC does not prevent DNS record manipulation from occurring, but it is intended to allow a client to detect if such alteration has occurred. DNS cache poisoning, also known as DNS spoofing, is a type of attack that diverts Internet traffic away from legitimate servers towards fake ones [35]. The reason this is so dangerous is that it can spread from DNS server to DNS server. As a result of this attack, e-mails can be redirected and copied before they are delivered to their final destination, voice calls can be tapped by third parties, etc. DNSSEC was designed to deal with this and a set of other DNS vulnerabilities, such as man in the middle attacks.

After reviewing these two systems, and cross-check them with the reThink Registry Service requirements, we realized that neither of these systems would be a feasible solution for the implementation of this service. By having a centralized architecture, LDAP would compromise the scalability of the system; moreover, LDAP is not an optimal solution for store very dynamic objects, such as Hyperty instances information. Regarding DNS, its use would mean the impediment of achieving a major reThink requirement: seamless migration between different service providers. For example, vodafone.pt/ruimangas would be controlled by vodafone.pt. When the user decides to move from vodafone.pt to orange.fr, his ID would still be managed by vodafone.pt.

2.4 Server load balancing techniques

Load balancing is defined as a process to distribute traffic across a set of servers. This process, which goes completely unnoticed to the end user, aims to optimize resource usage, maximize throughput and minimize response time [3]. Moreover, load balancers offer content-aware distribution, redundancy and health checking to ensure that the servers are indeed running and accepting requests. If a server is found to be down, the load balancer removes it from rotation and stops sending it requests.

This process of load balancing Internet traffic is entirely related to scalability. As servers become overloaded, system administrators are generally faced with two possibilities: vertically or horizontal scalability. The first is performed by adding more resources to a single server, typically by adding more RAM or CPUs. However, a single server could only scale so far. At some point, it is impossible to add more resources since the hardware platform has its limits. Also, the server needs to be taken down in order for this upgrade to be concluded. On the other hand, horizontal scalability is the ability to add more nodes to the system. This usually requires one of several load balancing techniques, topic that will be explored further on - but first, DNS-based load balancing will be summarized since it is also a process to distribute traffic across multiple servers.

DNS-based load balancing, also known as DNS round robin, is a function of DNS that allows one hostname to be associated with one or more IP addresses. Although very easy to deploy, round robin DNS has a few drawbacks, such as if a server corresponding to one of the IP addresses is down, DNS will continue to deliver that IP address and clients will attempt to connect to a service that has failed.

Load balancing can be accomplished at various layers of OSI. Here we make an overview of the two
most used load balancing options: layer 4 and layer 7 load balancing.

- **Layer 4 load balancing** operates at the transport layer, which redirects requests no matter the type of the request or its contents. It is simplest method of balancing traffic across servers. This simplicity means fastness balancing with minimal hardware. However, limitations are present. Since the load balancer can not see the contents of the request, it can not make routing decisions based on that. That is, it can not decide what is the best server to deal with a specific request.

- **Layer 7 load balancing** operates at a high level application layer, which deals, and can make decisions based on the actual content of each message. This kind of load balancers differ form layer 4 load balancers because the servers do not need to serve the exact same content. Instead, each of the servers can specifically and efficiently serve specific content such as, video or images. So now a request for an image or video can be routed to specific servers that store and are optimized to serve multimedia content.

Since we are talking about scalability it is also important that the load balancer itself does not become a single point of failure. In order to work around that problem, load balancers are usually deployed in HA pairs in which one of the load balancers is in passive configuration constantly monitoring the other one to replace it in case of failure. This technique is usually associated with a floating IP address that points to one of the load balancers, and can be remapped to the other one if the first one fails.

Comparing load balancing options, Nginx and HAproxy are both extremely performant reverse proxies/load balancers, and they both work on layer 4 (TCP) and layer 7 (HTTP). However, while HAproxy is really just a load balancer, Nginx is a webserver that can also work as a load balancer.

### 2.5 Distributed systems monitoring architectures

As distributed systems with a lot of moving parts continue to grow on complexity and size, it is crucial to verify that they maintain their correctness properties at all times. To achieve this level of trustworthiness, this kind of systems must be design to be highly fault-tolerant. However, and because failures happen, system administrators need to have ways to predict and see in real-time how the system is doing at carrying out its job. Therefore, monitoring is used to obtain information about an observed system with the goal of collecting and displaying that information in real-time dashboards. For example, server’s processing times, error counts, servers lifetimes, query counts and resource usage are common metrics that are obtained and displayed in human-readable format to system administrators.

Over the recent years, and to fulfill the necessity for monitor large systems, several tools were developed. Tools such as Prometheus\(^2\) (developed by SoundCloud), Nagios\(^3\) or Riemann\(^4\), consume or receive, and aggregate data from multiple hosts feeding it into event processing systems to be manipulated and then shown in real-time dashboards. A description of these tools is presented below.

\(^2\)https://prometheus.io/
\(^3\)https://www.nagios.com
\(^4\)http://riemann.io/
- **Nagios** is a free and open source application that monitor infrastructures, networks, servers and switches. It is a pull-based system that queries the components being monitored. Services that can be monitored by Nagios include SMTP, Post Office Protocol (POP3) and HTTP protocols. It also alerts users when something is wrong and alerts them a second time when the issues are solved.

- **Prometheus** is a monitoring system and a time series database. As Nagios, Prometheus scrapes metrics from the monitored services, either directly or through an intermediary gateway. It also employs a multi-dimensional data model and a highly flexible query language to leverage it.

- **Riemann** is a fast and highly configurable network monitoring system that aggregates events with a powerful stream processing language. It also provides alerts, notifications, and the ability to send and receive events from other services, that is, integrations. Although not providing scalability out of the box, its stateless principles makes it easy to scale by distributing the load across several Riemann servers.

Even though all of these tools have the same purpose (monitor and display system metrics) their architectures differ in many ways. The biggest difference between Riemann and the other two applications is the Riemann’s event-driven push model rather than the usual pull/pulling models. In event-driven models, the application does not do any active monitoring. The monitored services generates events. Those events are then sent to a Riemann server. On the other hand, in pull-based monitoring (Nagios and Prometheus), the application actively pull the monitored services. If any of those services fails to respond an event is generated and an alert is sent. This active pulling monitoring generally results in a not ideal, centralized, vertically scaled and monolithic architecture. Figure 2.10 describes these two models.

![Figure 2.10: Push vs Pull systems](image)

In the next section will be described centralized logging architectures and the importance of having such systems in a complex distributed, networked system.
2.6 Centralized logging architectures

A log is a record of events that occurred in an organization's networks or systems. They record what happened and when, and are aggregated in Logfiles comprised by log entries, each of which containing a description of an event that was created within communications between systems or with user interaction with such systems. Logs may serve multiple functions within large architectures, such as troubleshooting problems, recording the actions of users and providing data for exploring possible malicious activities [36].

Over the years most organizations have faced the same challenges regarding an approach to dealing with large, ever-increasing, amounts of computer-generated log messages. Since in a typical organization's architecture every computer and application generate logs, in [36] are identified the following complications:

• Many log sources: As said before every piece of networked equipment may generates logs. Thus, log management is a necessary technique to be performed throughout an organization. Furthermore, since many applications may be running on the same host, numerous logfiles can be generated from a single computer.

• Inconsistent log messages: Since every log recorder application generates logs in different formats, it complicates the process of linking logs generated from different sources. As an example, one application may use Secure Shell (SSH) in its logs and another one may only use SSH’s port number (22). However, both ways are useful to identify a user login in a remote machine.

• Inconsistent timestamps: When applications are producing logs, they generally rely on the host's internal clock to generate timestamps. Therefore, this can difficult the process of analysing logs from different hosts. Moreover, each application may generates dates in various formats, such as one being MMDDYYYY and another MM-DD-YYYY.

Regarding the architecture design process of a log management infrastructure, it usually comes down to three decisions: how to generate logs, how to analyze, normalize and store them, and how to displaying it to the end user. Concerning how to generate logs, it generally occurs by letting other applications connect to the server and get a copy of the logfiles (pull-based system - as described above), or having some logging client services that ship those logs through networks to logs analysis tools (second tier of the three decisions). The second decision to be made is where to store all the logs received and how to analyzed them. It can vary greatly in structure and complexity. Log data may be stored on the servers that perform the analysis or be sent to another database servers. The second alternative is more useful if we have scalability in mind because database servers and log analysis servers can be scaled-out as needed. After all logs be processed, a tool must be chosen to visualize and understand the large amount of data generated by the whole architecture.

Once again, as in monitoring tools, several tools and frameworks have been developed to solve the problems above mentioned. In this document we will be start by looking at the Elasticsearch, Logtash
and Kibana (ELK stack, now the elastic stack)\(^5\), and then to another ELK stack alternatives. The ELK stack provides an end-to-end log management that delivers results in near real-time for almost all data formats. Logtash is responsible for collect the logs data, make transformations like parsing (e.g. using regular expressions), adding fields and store them for future use. Finally, if we decide to store the logs somewhere else, Logtash can send them to various destinations (e.g. databases such as Amazon S3\(^6\) or Elasticsearch). Elasticsearch is a RESTful data indexer, providing a cluster solution to perform searches and analysis on a set of data. In the ELK stack, Elasticsearch serves as a backend datastore for Kibana. Kibana queries it and provides visualization capabilities of the content indexed by Elasticsearch. Users can create among other things, bar, line and scatter plots, or pie charts.

FluentD is open source data collector that can be used to substitute Logtash in the ELK stack. Both of these applications have a rich plugin ecosystem covering a lot of input systems (e.g. file and TCP), filters and output destinations. However, Logtash lacks a persistent internal queue, relying on external queues like Redis for persistence across restarts. FluentD, on the other hand, can store data in-memory or on-disk. Moreover, it can work seamlessly with various data visualize tools, such as Kibana or Graphite\(^7\).

![Figure 2.11: Log management as a service](image)

In essence, choosing a log management architecture can vary a lot depending on the core architecture. A lot of tools, such the ones presented above, can be combined to achieve desired results based on what the outcome should be. However, all those tools aim at resolve the issues present in the bullet points above.

### 2.7 Chapter summary

In the previous subsections was described several architectures and protocols to develop large scale distributed systems. We outline the advantages and drawbacks of each system being the P2P paradigm knows for its huge scalability and availability. However, the uncertainty of where the data is stored com-

\(^5\)https://www.elastic.co/products  
\(^6\)https://aws.amazon.com/pt/s3/  
\(^7\)https://graphiteapp.org/
prises its most unpleasant disadvantage. On the other hand, every evaluated client-server architecture is
easily manageable and accessible. Yet, these architectures are not robust as P2P infrastructures. In the
end of the Chapter are introduced load balancing techniques, and network management architectures,
i.e., centralized log management and monitoring.
Chapter 3

Architecture

This chapter describes the overall system architecture of the Domain Registry and outlines its central architectural components. The main design goal is to provide reThink with a highly available architecture for one of its most important and critical components, the Registry Service. We identify two actors: the CSP, which provides and deploys the system, and the Registry Connector, a microservice also deployed by a CSP (and part of reThink), which interacts with the Domain Registry.

We introduce the system requirements in 3.2 and its architecture through sections 3.3, 3.4 and 3.5. We follow a bottom-up approach starting with the purpose of solving the functional requirements (Section 3.3) and then we progress upwards until both the functional and non-functional requirements are solved and unified in a single architecture (sections 3.4 and 3.5).

3.1 Design decisions

As discussed previously in Section 1.1, in order for users to discover one another, the reThink framework introduces the Registry Service, a single service that will be used very often for Hyperty related information discovery. It should provide a world-wide service. As the idea is for multi CSP participation, and because this is a single service, the responsibility for maintaining it should not lie with a single CSP. Therefore, the Registry Service design was split into two components: one based on a client-server model that will provide CSP-dependent information and another, based on DHTs, that will map reThink unique identifiers to CSP-dependent identifiers. The latter will be used to access the first, client-server based, service. The first service is called Global Registry and the second Domain Registry. The Domain Registry will be deployed by each CSP while the Global Registry will be a DHT in which each node belongs to a distinct CSP. Figure 3.1 depicts the relation between the Global Registry and the Domain Registry. The Global Registry stores reThink unique identifiers (also known as global unique identifiers) that resolve to CSP dependent identifiers that are used to access the Domain Registry, and therefore, discover what Hyperties are running in other user’s devices. The example from Figure 3.1 shows the steps performed by Bob’s runtime to reach Alice’s Hyperties. It contacts the Global Registry with Alice’s reThink identifier, and then, it uses Alice’s identifier from CSP A to reach Alice’s Hyperties. Please note
that this is a simplified version that hides other reThink components. Its solely purpose is to demonstrate the relation between these two reThink modules. Moreover, ideally, Alice's GUID would resolve to more than one CSP.

By employing this design, we are able to achieve the following:

1. Each CSP deploys its own Domain Registry on its own servers. Thus, CSPs do not lose control over their data.

2. The Global Registry will act as an address list, where users can discover in which CSPs other users have registered services.

3. The Global Registry becomes a decentralized service, and thus, no single CSP has total control over it.

The Global Registry was developed and evaluated by other reThink researchers. Thus, there are no references to its development and architecture in this document. This thesis focuses on the Domain Registry. Over the next sections are presented its requirements, architecture and design decisions.

### 3.2 Requirements

This thesis addresses the problem of providing and developing a highly available service for reThink’s Registry Service. It is a critical part of reThink since it is a service that stands in the critical path for establishing a call or any sort of communication between two users. Our overall goal is to create a service, called Domain Registry, that stores, for each Hyperty instance, the data that enables other applications to contact it. This is the service that provides the mapping between the identifier for each
Hyperty instance (a Hyperty is used by a user in one or more devices) and the data that characterizes it. Therefore, the Domain Registry should provide the following functional requisites:

- Map identities to the Hyperty instances they are using;
- Provide information about a given Hyperty instance;
- Provide an interface for the other reThink services to harvest data.

Moreover, our system must fulfill the following non-functional requirements:

- **Fast query response time:** Since users connect with each other through the framework reThink will provide, our service must provide low latency and a consistent performance. Otherwise, it could have an influence on the performance of the reThink platform;
- **Scalability:** This system must provide a service for a large number of service providers. It should easily scale as needed;
- **High availability:** Without this service, there is no way to establish a call or communication. Thus, our system needs to be continuously operational.
- **No single points of failure:** A certain amount of resilience must be provided, so that the failure of one node does not bring the others down. It means that at any time, any given node can be shut down or disconnected from the network while the system continues operational.
- **Security:** Since we do not know the environment in which CSPs will deploy both the Domain Registry and the Registry Connector, we have to ensure that the communication between these two systems can be configured in a secure manner.
- **Developers usability:** Ensuring that every developer’s computer is configured properly will delay the development process and introduce complications with software versions incompatibilities. Thus, from the standpoint of reThink’s deployment team, the Domain Registry needs to be easily deployable with all its dependencies.

From the point of view of the CSP that deploys the service, our design must also include a second architecture (directly linked to the first one), that enables system administration mechanisms to constantly monitor the behaviour of the deployment system, including all the interactions of its internal components. For that reason, the upcoming, maintainability also non-functional requirements, shall be present in our global architecture.

- **Support for component monitoring:** Monitoring is an important part of cluster management and should be provided. As a result, we should be able to detect, before they lead to service outage, network component problems, as well as analyzing long-term trends (e.g. database or user base growth);
- **Support for centralized log management:** All logs must be searchable in a single place. That way, we can correlate logs from different applications, which can be useful to identify user actions and applications problems.
### 3.3 Core architecture

In order to comply both with the functional requirements of Section 3.2 and the reThink design decisions presented in the beginning of the chapter, we introduce a client-server REST API that exposes, and allow other systems to harvest, the services offered by the Domain Registry. The API will run on application servers that will reside in the middle tier of our deployment architecture (see Figure 3.3), and will return, in all cases, JSON documents containing the responses. This REST service will allow the Registry Connector to register, delete and perform different types of searches on Hyperties. Thus, as can be seen from Figure 3.2, the Registry Connector will issue HTTP requests to the Domain Registry, which, in turn, will deal with the requests and save (or retrieve) them from a persistent or in-memory database. Despite the knowledge that P2P systems have an ideal system design when considering high availability and failure resilience, given the reThink project constraints for the Domain Registry, we introduce it as a client-server system, with high availability being achieved using server replication and load balancing techniques.

Considering why the choice for a RESTful architecture instead of the traditional SOAP and all its underlying conventions (i.e. WSDL and UDDI), it was a matter of developing a future-proof, easy to develop, and maintain system. This ease of use will make it easy for other developers to understand what was done and write or modify services against it. Besides, for third-party application integration, which is what the Registry Connector does, it is more straightforward to issue HTTP requests and parse the JSON output (easier to parse than XML, used in SOAP), than dealing with all the interactions required by SOAP to request a specific resource from a Web service.

![Figure 3.2: Domain Registry architecture](image)

Therefore, our HTTP-based RESTful API is defined with the following aspects:

- Base URL, such as `http://api.domain.registry.com/hyperties`;
- Standard HTTP methods (e.g. GET, PUT, and DELETE);
- A description of the state transition of the data elements.
Table 3.1: Domain Registry API specification

The API endpoints that were defined are presented in the Table 3.1. The first endpoint is the most important one since it let us create, return and delete an individual Hyperty for a specific user. The second is used to return all the Hyperties associated to a user, and the rest of the endpoints are utilized to perform advanced searches based on Hyperties characteristics, i.e. DataSchemes and Resources.

As can be seen in the Table 3.1, HTTP PUT was chosen to create and update Hyperties instead of using PUT to update and POST to create. That decision was made based on some considerations. First, since the identifiers of the Hyperties are chosen a priori, it does not make sense to use POST because then the server would decide what identifiers to use. Secondly, PUT is idempotent [22], that is, a client can PUT a object twice and the result will be same. This is a nice property, which does not happen with POST. If two POST requests came at the same time making changes to an URL, they may generate different objects. Lastly, the same URL is used by PUT to create or update a Hyperty. It is simpler and reduces the number of API endpoints - that is, complexity.

3.4 Deployment architecture

In the previous section we established the core architecture of the Domain Registry. It will be a REST API running on application servers that will allow reThink components to manage Hyperties. However, no non-functional requirements were addressed. These requirements (introduced in Section 3.2) are hugely important because if the Domain Registry is not reliable (for instance, while under load, or when failures happen), then it is not going to serve the client’s needs. For that reason, the next section will introduce an architecture that was designed to meet such requirements.

In Section 3.4.1 we present an overview of the global architecture and, in sections 3.4.2, 3.4.3 and 3.4.4 we will address the decisions that were made to achieve the non-functional requirements introduced in Section 3.2.
3.4.1 Infrastructure overview

Figure 3.3 depicts the overall deployment architecture of the Domain Registry. It comprises two Haproxy load balancers in failover mode, and at least, three application servers and four Cassandra database nodes. All database nodes work in a P2P model and thus any application server can query any database server, and get the expected results. All application servers will run the REST API discussed in the previous section. Moreover, besides this production ready architecture, and for the purposing of testing, are also available two other deployment alternatives: the first with requests being saved on memory and the second with requests being saved in a single node Cassandra database. These two alternatives allow developers to rapidly test the API with the purpose of getting to know, and experiment, the available endpoints.

![Diagram of Domain Registry main architecture]

Over the next sections, will be provided, individually, an explanation of each component that comprises our deployment architecture design. First, it is described the load balancers and floating IP mechanisms, then the database design and lastly, the security concerns that will allow a CSP to deploy, if needed, the Domain Registry using SSL connections.

3.4.2 Load balancing

As already explained in 2.4, load balancers are added to a client server environment to improve performance and reliability by distributing client workload across multiple server machines. Between Layer 7 and layer 4 load balancers, we end up configuring a layer 7 load balancer because, although currently all the application servers serve the same content, as the system will grow, it may be useful to reassess the load balancing technique, and maybe employ a request awareness traffic distribution and choose different servers to deal with different requests. Moreover, in terms of traffic encryption, layer 4 load balancers treat connections as just a stream of information, rather than using its functions to evaluate and interpret the HTTP requests. This would mean that we would be forced to configure traffic encryption on the application servers.

Nevertheless, an architecture with a single load balancer can easily become unavailable if that load balancer fails. Since we needed to take into account high availability and scalability we decided to use a HA pair of load balancers with a failover mechanism in an active/passive configuration. This configuration is achieved by having a floating (or virtual) IP address which can be instantly moved from one server
to another in the same datacenter. Our infrastructure must be capable of immediately assigning this floating IP to an operational server.

To achieve this goal, we used the Virtual Router Redundancy Protocol (VRRP) [37], which is responsible for providing automatic assignment of an available floating IP address to participating hosts while, at the same time, ensuring that one of them is the active node (master node).

While using VRRP, failover should occur when either of the following conditions occur:

- *When the load balancer health check on the primary server indicates that the load balancer is no longer running*: In this model, the master node constantly monitors the load balancer process, and when this process goes down, it sends a message to the slave node, which takes over almost seamlessly and instantly, allowing the service to resume.

- *When the secondary server loses its VRRP connection to the primary server*: If the secondary server cannot reach the primary server for any reason, it will change its state to ‘master’ and will attempt to claim the shared IP address.

In the case where there are more than one backup load balancers with the same priority values, the one with the highest IP address wins and becomes the master. If the primary server later recovers, it will change back to being the master node and will reclaim the shared IP address, because it will have the higher priority number in its configuration.

In figure 3.4 is described the two case scenarios that can occur in a active/passive load balancer configuration. On the left is the normal scenario and on the right is the expected outcome of VRRP, when the primary fails, the secondary load balancer takes over and assumes the shared IP.

![Figure 3.4: Load balancer failover case scenarios.](image)

### 3.4.3 Database servers

Database systems are a ubiquitous and critical component of many modern computing based applications. As a consequence, it is a component that must be selected taking into account several factors, such as replication, failure resilience and scalability. The first big decision while selecting such system is whether use a relational (SQL) or NoSQL database. NoSQL databases are known and designed to
handle extremely large data sets with hundred or thousands of entries. Moreover, most of these systems claim to scale horizontally near linearly, i.e. duplicate the number of rows means duplicate the number of nodes.

Based on our requirements presented in Section 3.2, our infrastructure must provide high availability with no single point of failure, and every component should be easily scaled. Thus, our main concern while choosing a database system, is to preserve availability during network partitions and failures nodes. Easily scaled architectures are almost often analogous with horizontal scalability, which is the process of adding, incrementally, hardware as needed. Also, a database that follows this design must allow a seamless addition of new nodes with no downtimes. This level of scalability flexibility easily grants a very efficient deployment on either hardware components or in cloud based Infrastructure as a Service (IaaS). Our goal here, is the ability for the CSP to scale our already developed and configured cluster as needed, and even do it on the fly (if IaaS is used).

Regarding storage, the Domain Registry stores on every request JSON documents, which are then associated with a specific user and its Hyperties. This data will be updated often, e.g. due to Hyperties being started, stopped or IP address changing; hence, write operations will be frequent while reads will occur less frequently. Since this a service which will be deployed by a CSP with probably hundreds of thousands or even millions of clients (with each client having dozens of applications running on several devices), the Domain Registry storage is expected to be extremely large.

Therefore, and by taking into account the above requirements, we chose to use a NoSQL database cluster with a P2P architecture, comprised of four nodes and a replication factor of three, allowing us to survive the loss of two nodes. With this configuration, there will be three copies of each document stored across three different nodes. Thus, every node will hold three-quarters of the data. As studied in Section 2.1.1, the decentralization nature of P2P architectures grants us the robustness needed because it removes the single point of failure from the database design. Moreover, with this database architecture we achieve horizontal scalability by adding more nodes as system’s capacity increases. The overall Domain Registry capacity also increases, while the likelihood of a system failure decreases.

### 3.4.4 Security concerns

Network security consists of the practices used by an organization to prevent unauthorized access or modification of networked resources. In our infrastructure, even though all components are to be run inside the same organization, we decided to implement a secure connection with HTTPS between the Registry Connector and the Domain Registry. Despite the fact that the Domain Registry interface is not available from the outside, if the CSP decides that the connection between those two components should be secure, HTTPS can be enabled and HTTP disabled. This way, we give the possibility for the CSPs to choose what is the best mode to deploy the communication between such components given their infrastructure, requirements and objectives. Moreover, making this connection secure introduces a significantly level of trust since, by the usage of encrypted traffic between those two components, malicious employees can not see or modify what they were not authorized to.
In order to achieve this requirement, we were faced with four alternatives (depicted in 3.5, 3.6, 3.7 and 3.8) on how to implement Transport Layer Security (TLS)/SSL security between the client, the load balancer and the REST application servers. In the first scenario (Figure 3.5), the load balancer does not decipher any traffic. It just opens a TCP tunnel between the client and the server, and let them together deal with the SSL traffic. With this model, the CPU load is distributed across the backend servers; however, we lose the possibility to add or edit HTTP headers, as the connection is simply routed through the load balancer. The second scenario (Figure 3.6), works by having the load balancer decipher the traffic on the client side and cypher it on the server side. It can access the content of the request and make decisions based on that. Here, we have the concern of having both the load balancer and the application servers dealing with high CPU loads. It would probably be necessary to vertically scale these two components in order to achieve good performance levels. Next, in Figure 3.7, it is represented the SSL/TLS offloading scenario. In this case, the load balancer deciphers the traffic on the client side and sends it in clear to the backend servers. The application servers do not handle encrypted SSL traffic. However, as in the first two scenarios, the load balancer needs to be properly scaled to meet the overhead introduced by the SSL handshakes [38]. Lastly, in Figure 3.8, the load balancer receives clear traffic from the clients and uses SSL connections with the application servers.

As our architecture will grow in complexity and number of backend servers, we expect the load balancing to be a highly effective process. For that reason, the load balancer needs to capable of making decisions based on what the clients will request. Thus, it is necessary to be the load balancer itself to decipher the client requests. As a result, the scenario represented in Figure 3.5 will not be considered, and the scenario from Figure 3.8 will also be excluded since it does not meet the most basic security requirement we are trying to achieve, that is, a secure connection between the clients and the load balancer. These considerations also assume the usage of a layer 7 (HTTP) load balancer. Concerning scenarios four and five (from 3.6 and 3.7 respectively), it is a decision that depends primarily on how secure will be the deployment of both the Domain Registry and the Registry Connector inner architectures. Assuming that both architectures will be deployed in a secure manner, we ended up choosing a SSL/TLS offloading architecture. Moreover, from a performance point of view, it is way more feasible to scale-up only one component, which in this case will be the load balancer, than to scale-up multiple backend servers. Also, by offloading an heavy task from the application servers, we let the servers to focus on the application itself, while at the same time we save hardware resources that can be used by the load balancers. Although we are focusing on application servers performance, we also know that the load balancer can itself become saturated, while dealing with SSL connections under heavy loads of traffic. It is a trade-off that has to be carefully re-evaluated as the system will grew.

3.5 Network management architecture

Last but not least, we present an architecture aimed at resolving the maintainability non-functional requirement present in Section 3.2. It is system directly connected to the deployment architecture which aims at providing network management tools, i.e. monitoring and centralized logging. We will start by
presenting an overview of the system and, in the following sections the choices we have done to design such system.

### 3.5.1 Architecture overview

In Figure 3.9 is represented the overall monitoring and centralized logging architecture of our infrastructure. It incorporates five servers being three of them responsible for dealing with application logs and two of them with monitoring events. As depicted, all three components from the deployment architecture (database servers, load balancers and application servers) generate logs and events that are then sent to other servers responsible for interpreting, parsing and displaying the results to the administrators.

![Monitoring and centralized logging](image)

In the next two sections, the choices behind the architecture above represented are explained. We start by explaining how the monitoring is performed and end up with a description on how logs are routed through the components and how they are parsed and searchable.

### 3.5.2 Servers monitoring

Monitoring is the process of collecting, processing, aggregating and displaying quantitative real-time events to the users. As we are dealing with a lot of servers, each of which with different exposed metrics and resource usage, monitoring is a crucial component of our infrastructure. It will help us to tell us
when something is broken, or perhaps what is about to break. For that reason, we implemented a model where all of the servers generate and send monitoring events to another server responsible for parsing and saving them. We opted for a push model in which the servers responsible for dealing with the monitoring events do not do active monitoring. They just wait for the events to reach them, and then when they do, the servers start to perform the tasks they were assign to do. It is a data driven model. Once the deployment architecture servers realize that they have some content to be published, they will sent it without any request from the receiving end. This model has a big advantage over the pull based systems: the monitored nodes do not need to be constantly interrupted with demands for data that they probably do not have yet. Moreover, a pull based system, would mean that, as our deployment architecture grew, it would also grow the number of servers that the system would need to query. We then needed to scale our pull based system vertically, which would bring several problems, which were already discussed in the Chapter 2. Therefore, a push based system was design that collect the following (most important) metrics:

- **Resource level events**: Were collected, from all the deployment architecture servers, events that corresponded to values from RAM, CPU usage, CPU load and disk usage.

- **Number of active servers**: From the event’s origin was estimated how many applications and database servers were up and running. That is, this number is equal to the number of unique servers that were sending events. If this number ever decreased (meaning that a server was down), an alert would be sent to the dashboard.

- **Request per second**: From the load balancer status page it was harvested how many requests per second the load balancer was receiving at the given time. If this number was ever higher than 1000 req/s, an alert would be sent to the same dashboard as before.

- **Response codes**: Once again, from the load balancer status page, it was gathered individually statistics from the number of 2xx, 4xx and 5xx response HTTP codes, that the load balancer was receiving from the application servers. These response codes are an important metric that we needed to keep up with. For example, existence of 5xx response codes indicate that some of the servers may not be operating correctly.

- **Writes and reads**: From the events received from the application servers, it was aggregated how many writes and reads were issued to the Domain Registry at a given time. This values may be interesting to perform some long-term trends analysis, such as how big is the database and how fast it is growing.

- **Average response time**: Yet again, from the load balancer, it was gathered the average response time of the last 1024 requests. If this value ever exceeded one second, once again, an alert would be created notifying the operations team about a problem.

After all these events were processed and calculations made, another server running a dashboard connects to and query the first server, and display, in near real-time, with counters and graphs the aggregated results.
3.5.3 Centralized log management

Centralized log management is a very useful component in any networked infrastructure since it help us search for all application and server logs in a single place. In our deployment architecture every server and application generate logs. The load balancers generate all kinds of logs which go from logs about servers health to logs about user requests. These are very complex and verbose logs, but in the end, help us identify and keep track of everything related to both the front and the backend of the load balancer. The application servers also generate logs about user actions against the REST API and its respective results. Finally, the database servers produce logs which can be very useful to troubleshoot problems with the database, as well as, for example, to obtain information about the most issued queries and its results. The latter are exceptionally helpful for debugging purposes. It is important to note that all this logs are created along with timestamps from each logged event.

Gathering and parsing logs from multiple sources have the problems already identified in 2.6. Therefore, we needed a central component, which would be responsible for parsing and storing logs for future use (e.g. dashboards). Bearing this in mind, we deployed a system where all those logs are first received by a server responsible for normalize varying schemas and data formats. This normalization aims at defining a common logging format before inserting it into an analytics datastore. Storing is the second stage of this system. For displaying near real-time data to the developers, fast searches and powerful analytics capabilities were needed. Consequently, all of our logs are sent from the parsing server to a second one that does exactly that. It is vital that the chosen tool to carry out this task can be able to scale horizontally as fast as our dataset grows. In Chapter 4 we will evaluate some tools and explain why the choice for Elasticsearch. Hence, all logs end up in an Elasticsearch cluster and are then queried by another server (the third in this model) that will then present that data in several dashboards with bar charts, line and scatter plots, histograms and pie charts.

3.6 Chapter summary

The Domain Registry is a REST server deployed in a HA infrastructure with no single points of failure which works by using floating IPs and a database with a P2P architecture. Furthermore, the Domain Registry leverages a monitoring and a centralized logging architecture which performs a highly necessary role in nowadays big web servers infrastructures: near real-time information gathering to prevent and act on possible failures across a server cluster.
Chapter 4

Implementation

This chapter addresses the main decisions adopted regarding the implementation and configuration of the Domain Registry’s internal components. Thus, the following sections cover the technologies that were used in the development process of those components, as well as, other modules that, although not represented in Chapter 3 images, were important to perform some internal actions.

4.1 Core Architecture

The Domain Registry’s core architecture, that is, the REST application servers, was developed with a micro framework for creating Web applications with Java called Spark. Not to be confused with Apache Spark, Spark Framework, inspired by Ruby’s Sinatra, is a lightweight Web framework built around Java version 8 lambda functions, which makes Spark a lot less verbose than the typically Java Web frameworks. This possibility started with the choice of Java as the primary programming language to develop the Domain Registry, since it was a programming language that was already being used in many reThink services. For code maintainability reasons it was the best choice which will allow, if needed, other developers to maintain and enhance the Domain Registry features in short periods of time. Inside Java’s ecosystem, there were other Web application frameworks that were considered. For instance, the Play Framework, known as a web framework for both Java and Scala, was a great candidate. However, Play, for being a fullstack web development framework that even includes its own build tool, would be an overkill tool for building our RESTful microservice. Consequently, for those reasons, Play has a steep learning curve which would be a difficulty for other reThink developers or open source contributors to came up with new features.

Regarding the execution of the Domain Registry it has two storage models: in-memory database and a persistent database. The persistent database is a production ready model, while the in-memory database is used for its deployment simplicity when running tests and integrations with the others components. The storage type is chosen a priori with a configuration parameter.

1 http://sparkjava.com/
2 www.sinatrarb.com
The architecture behind our code followed a Model–View–Controller (MVC) inspired approach without the views. This approach was used to make a clear division between modules (separation of concerns), and therefore easily test them individually. The code organization within this MVC based structure provides a clean and organized codebase, making it easier to scale in terms of adding new features. From a development standpoint it also provides easy integration with other frameworks and backend services (e.g. databases).

The controller is the link between the user and the system. It interprets the Registry Connector HTTP requests and passes them to the model, which captures the behavior of the application by interacting with the database and returning the results back to the controller. After that, the controller wraps the results within a JSON document and sends it to the user along with a HTTP response code. These interactions can be seen in Figure 4.1.

![Figure 4.1: Interaction between internal code modules](image)

Apache Maven was used as a build automation and management tool. It provides the concept of a Project Object Model (POM) file to manage the project's build, dependencies and documentation. A major advantage of using this tool is its ability to download all project dependencies automatically from central repositories. It is probably the biggest advantage that Maven has over Apache Ant, since with Ant we need to download the Java Archive (JAR) manually and add them to the classpath. The next section will address how our REST server is deployed within the application servers.

### 4.2 Deployment Architecture

For the Deployment Architecture (refer to Section 3.4) we used several tools that will be explained throughout the next sections. However, and since it served as the basis of our deployment, we will introduce Docker [39] here. Docker, sometimes described as lightweight Virtual Machines, is a new container technology, that eases the process of packaging and shipping distributed applications, whether on personal computers, VMs, or the cloud. It allows applications to be isolated within containers with instructions for what they will need to be ported from machine to machine. VMs allow exactly the same thing and with configuration management tools, such as Puppet, Chef or event Vagrant, the process of configuring portable and reproducible applications becomes less complicated. However, where Docker stands out is on resource efficiency. If we have fifteen Docker containers we can run all fifteen with a single command on a single VM. By contrast, if we have fifteen VMs, we need to boot fifteen operative
systems instances with a minimum of resources from the base OS. Besides the clearly outstanding performance of Docker, what really made us use it was its painless way of deploying applications. Docker containers are created using images. These images can be very basic (containing nothing but the OS fundamentals), or it can comprise sophisticated pre-built applications ready to use. Applications are run through Dockerfiles that contain various instructions to automatically perform actions on a base image. After writing the necessary Dockerfiles, with only two commands, i.e., `docker build` and `docker run`, the application is launched and is ready to be used.

Referring back to our requisites presented in Section 3.2, with pre-built containers and some Dockerfiles of our own, everyone who wishes to use, deploy, or test the Domain Registry, can do it in an effortless manner by simply installing Docker and running those two commands. From now on, please assume that every software module was deployed using Docker. Figure 4.2 shows our deployment flow. Dockerfiles are downloaded from Github and pushed to Docker Hub, a central repository of Docker images. Developers may run the project and make some experiments with it. When the components are ready to be deployed in production, Docker images are downloaded from Docker Hub and the code is executed in production machines.

![Deployment architecture](image)

**Figure 4.2: Deployment architecture**

### 4.2.1 Load Balancers

The load balancer mechanisms implementation was split in two phases: first, its foremost role, that is, the distribution of traffic across a set of servers, and then, the failover strategy using the VRRP protocol. Accordingly, the procedures introduced in this section will follow the same order.

The choice of what load balancer software to use was narrowed down to open source software, form which Haproxy [40] and Nginx [41] stands out. Nginx software claims to be the world’s number one web server, and besides that, it is also a high performant reverse proxy. On the other hand, Haproxy, released in 2002, is just TCP/HTTP load balancer with a lot of advanced routing and load balancing techniques. They both support SSL offloading and layer 4/7 load balancing. By having these characteristics, both of them are suitable for what we are trying to achieve. However, for the purpose of monitoring, Haproxy
provides a live statistics web page in which we can, over HTTP, extract its representation in a Comma Separated Values (CSV) file. Although, Nginx provides some of these services, these are paid features. As a consequence, we ended up using Haproxy as a load balancer to distribute requests over the Domain Registry application servers.

The most important Haproxy configuration sections are the *frontend* and the *backend* of the load balancer. The *frontend* defines how requests should be forwarded to the backend servers, while in the *backend* it is specified what load balance algorithm to use and which servers are available to receive requests. On the *frontend* we listen for incoming connections on the load balancer public IP address, add the HTTP header X-Forwarded-Proto to the end of the HTTP request, and redirect incoming traffic to the backend section. The X-Forwarded-Proto header defines the originating protocol of a HTTP request. That is, it helps us identify if a client used HTTP or HTTPs to connect to our server. Moreover, we add the header X-Forwarded-For to identify the IP address of the clients that connected to the load balancer. This way we can identify and send alerts if there are any clients trying to connect to the load balancer other than the Registry Connector.

On the backend of the load balancer we decided to use the *roundrobin* algorithm to serve requests to the Domain Servers. With *roundrobin*, each server is used in turn, or if some servers are more hardware powerful than the others we can assign weights to each one. In our setup, and since our servers are equal hardware wise, we assigned the same weight to all servers. Another common algorithm is *leastconn*, which works by selecting the server with the least number of active connections. This is a very useful algorithm whenever we are load balancing something that might have long lived connections. Since it is not the case, *leastconn* was not considered.

To allow Haproxy to detect and act on failed backend nodes, some additional configurations were included. The first parameter was *inter*, which sets the interval for server health checks. We kept the default value, which was 2000 milliseconds. Besides that, *fall* and *rise* were used. *Fall* sets the numbers of checks that are done to declare a server as dead and *rise* does exactly the same but to declare a server as operational. Both were configured to perform two checks to confirm that those nodes are indeed not running or operational.

In order for overcome a possible load balancer failure, floating IP addresses were used. To achieve this goal, we used a tool called *keepliaved* [42] that implements the VRRP protocol, allowing us to setup Haproxy nodes in a master/slave configuration. If the master goes down (hardware or software failure), the slave will be elected as master and will start accepting requests. We started its configuration by opening a *vrp_script* on both load balancers. This will allow *keepliaved* to monitor the Haproxy process and start recover measures when its process stops claiming a pid. Besides Haproxy monitoring failover, if the backup load balancer ever stops receiving VRRP advertisements from the master, it assumes the master role and assigns the floating IP to itself. The only differences between the master and the slave configurations is the priority setting. The master server must have a high priority value than the slave. Otherwise, when the master node comes back up, it can not assume its role because it would have a lower priority value. Thus, in our configuration, the master and the slave have priority values of 101 and 100 respectively.
In order to use Haproxy for SSL termination, and since we are securing the communication between two internal applications, we generated a self signed certificate with the `openssl` tool. `Openssl` is a cryptography toolkit that implements the SSL protocol. It provides several for using various cryptography techniques such as, certificates, cryptography keys and message digests.

### 4.2.2 Database

As was explained in Chapter 3, we chose a NoSQL database to persistently store the data about each Hyperty instance. Unlike relational databases, NoSQL databases do not guarantee Atomicity, Consistency, Isolation and Durability (ACID) properties. One of the key features that differentiates them from relation databases, is their approach to preserve consistency or availability during network partitions. As the Consistency, Availability, Partition Tolerance (CAP) theorem states, it is impossible for any networked shared-data system have more than two of the three desirable properties: consistency, availability and network partition tolerance [44]. Taking this into account, and since we were trying to achieve high availability with no single failures, we started the process of choosing the ideal NoSQL database. The ideal system would be one that was designed to be AP (from CAP theorem), while at the same time, could provide some sort of configuration flexibility around consistency. Amazon’s DynamoDB [45] is an extremely flexible database that allows developers to configure stronger consistency models while trading off some performance when accessing the database. Like any other product from Amazon Web Services (AWS), Dynamo was designed for dealing with faults by having built-in resilience and self-recovery models. It has a P2P based architecture that uses consistent hashing to engage replication and data partitioning. However, using Dynamo implied that we needed to depend on AWS services. Since we are dealing with CSPs, that could not happen because they might not be willing to lose control over where their data was stored.

Over the years some Dynamo derivatives have emerged in the open source world. Cassandra [46], LinkedIn’s project Voldemort [47] and Riak [48] are three Dynamo-inspired databases that offer high availability and fault tolerance. We end up using Cassandra for two reasons: it supports a multi-datacenter aware topology that can be very useful as reThink grows and second, because Cassandra’s design focused on handling large write volumes. Moreover, the lack of documentation and use cases of both Riak and Voldemort discourages their usage. Another appealing feature of Cassandra is its design. Although, sometimes being referred to as a key/value store, Cassandra architecture diverges from Dynamo by being based on Google’s Big Table [49]. Cassandra is essentially a key/key/value store (map of maps) in which each row is mapped to inner columns that are sorted by a key. By breaking rows into columns, Cassandra design allow those columns to be updated independently. This way Cassandra can resolve changes in different columns automatically. For example, Riak, by being a pure key/value store uses vector clocks [50] to resolve merge conflicts by keeping both versions of an object, and when the client reads the object it will decide what version it wants to keep or what merges to perform. However, the issue with vector clocks is that they keep one entry per node, which means that they can become very large as the number of nodes in the system grows. Besides that, updating a single field in a
pure key/value store requires serialization and deserialization processes. Updating a field in Cassandra requires only the row key, the column key and the field itself.

Regarding consistency, whenever the Registry Connector makes a read operation, it should read the last updated value. However, for providing strong consistency, we need to give up on availability during a network partition. This happens because we can not prevent disparity between two replicas that can not communicate with each other while accepting write requests on both sides of the partition. Consequently, we might get old data from some nodes and new data from others until it has been replicated across all devices (eventual consistency). However, for what we are trying to accomplish with the Domain Registry, it is preferable to have weak consistency than not having availability, since in the latter scenario communication between two reThink users will not be possible. Moreover, lack of availability will affect, by far, many more users than eventual consistency will. In essence, we designed and configured the Domain Registry Cassandra cluster to be an AP system.

4.3 Monitoring

In Section 3.5.2 we provided an overview of push and pull based event processing systems. The main reasons why we chose a push-based model are related to scalability as the number of machines that generate events grows. Riemann was designed as a distributed system monitoring tool. It aggregates events from network hosts and feeds them into a stream processing language so they can be manipulated and aggregated. We used Riemann to monitor the Domain Registry architecture because, besides it featuring a push-based model, it benefits from a stateless architecture that makes it easy to partition and distribute the load across multiple Riemann servers. Once again, as we are expecting the Domain Registry architecture to grow in number of servers, we are ensuring that our actual Riemann architecture can be scaled with small effort.

Looking back at Section 3.5.2 many quantitative data about our architecture was monitored. However, the only data that is processed using code written by us before being sent directly to the dashboard are event aggregator sums that represent two things: the total number of HTTP requests made to our API and the number of servers (i.e. application and database servers) that were working at a given time. These metric aggregations were manually programmed by us using the Closure programming language (Riemann is written in it) based on the overall events received by Riemann. These events originated from our servers and were harvested and sent to Riemann using Ruby scripts. Starting by the Haproxy load balancers, we develop a program that first, scrapped its statistics web page to a CSV document and second, that sent the values parsed from the CSV to our Riemann server. To monitor the Docker containers state, we run the `docker inspect` command periodically, extract its result and sent it to Riemann for further processing. The API related metrics were sent to Riemann directly from the Domain Registry core architecture. Finally, to detect the resource level state of each machine (e.g. CPU and RAM) we used a Ruby gem called usagewatch\(^3\), wrapped it into a script and again sent its observed values to the Riemann server. Each time a event is sent to Riemann, it comprises the following fields:

\(^3\)https://github.com/nethacker/usagewatch
- **Hostame**: A string containing the name of the machine from which the event originated;

- **Service**: A string containing the unit of the metric of the monitored service (e.g. requests/seconds, average response time from last the 1000 requests and CPU usage percentage);

- **Metric**: The most important field. It contains the observed value that will be sent to Riemann and shown on the dashboard;

- **Tags**: An array containing a list of tags. This is very useful if we wish to aggregate events by tag.

In order for developers to visualize in real-time what was monitored, we used another server that runs the Riemann dashboard. The dashboard connects to the Riemann server using websockets and allow us to compose graphs based on queries issued to Riemann’s index.

The above mentioned Ruby scripts are open-source and available at Github @rujose user account with clear instructions on how to run them with or without Docker (Dockerfiles also provided).

### 4.4 Centralized Logging

As we stated earlier in Section 3.5.3, the current state of the art in central logging techniques is very vast and many tools can be combined to achieve similar results. Even messaging broker applications, such as Apache Kafka [51] and Redis [52], have been commonly used as alternatives to collect and store logs.

In order to achieve near real-time log analysis we needed text to be indexed on some sort of database. Text indexing refers to the technique of scanning full text documents and building a list of search terms (usually called index) [53]. Consequently, whenever a search occurs, only the index is queried, rather than the original documents. For that purpose we used Elasticsearch, which is a full text highly available search engine based on Apache Lucene [54]. Elasticsearch [55] fulfills our needs by letting us perform fast searches over logs, and also by allowing horizontally scalability which is achieved by partitioning the data into smaller chunks that can be stored in several Elasticsearch cluster nodes. Nevertheless, prior to storage, logs need to be collected in a central unit to be processed, normalized, and then sent to Elasticsearch. Both Logstash⁴ and Fluentd⁵ address the problem of transporting and collecting log documents. Logstash provides a large variety of inputs, codecs, outputs and filters. Inputs are sources of data and codecs are responsible for converting incoming data formats to a unified format as well as converting it back to a desired output format. Filters are processing actions on events and finally, outputs are destinations onto which events can be routed. Fluentd also has inputs, outputs and mechanisms to route logs, and although they are both performant, we ended up using Logstash by its seamless integration with Elasticsearch and Kibana⁶ (ELK stack).

As a means of shipping logs to Logstash, we installed in each of our servers another ELK stack underlying product called Beats⁷. Beats are lightweight processes written in Golang that capture and

---

⁴https://www.elastic.co/products/logstash  
⁵http://www.fluentd.org/  
⁶https://www.elastic.co/products/kibana  
⁷https://www.elastic.co/products/beats
send all sort of logs, directly or through Logstash, to Elasticsearch. Basically, what we did was configure each of our applications (i.e. Load balancer, REST server and database servers) to produce its logs to a predefined file which was then read by Beats and sent back to Logstash for further processing. Lastly, we configured Kibana. Kibana reads from Elasticsearch and displays its results in dashboards that can be consulted by developers. The overall idea of our centralized logging implementation is depicted in Figure 4.3.

![Centralized logging architecture](image)

**Figure 4.3: Centralized logging architecture.**

### 4.5 Chapter summary

In this Chapter we presented the main implementation details of the Domain Registry prototype and all its components. The main development challenges were to implement, configure and choose the appropriate tools to achieve a highly available and fault tolerant distributed system. In terms of scalability, the challenge was to implement the Domain Registry with easy scalability properties in order to allow other maintainers to enhance it if needed. To let anyone deploy the Domain Registry with ease, we found Docker to be an proper solution, allowing any developer to run it in a matter of seconds without any sort of configuration, except of course having Docker engine installed.
Chapter 5

Evaluation

In order to evaluate the developed solution, we performed several tests to measure the performance and scalability of the Domain Registry. Due to public cloud IaaS costs, we did the evaluation on IST’s network infrastructure using several Virtual Machines provided by DSI (Direção de Serviços de Informática).

The following sections detail the steps and decisions made throughout our evaluation procedures, starting by a description of the overall objectives and the evaluated scenarios. We then present our evaluation methodology and the challenges we faced to conclude the tests. Lastly, we show and discuss the results obtained from the Domain Registry’s evaluation.

5.1 Tests Objectives and scenarios

Our evaluation intended to demonstrate that the Domain Registry is performant and scales horizontally while adding more nodes. Furthermore, we aim to show the responsiveness of the failover processes that were configured on the load balancers.

For the first part of our tests, and given the Domain Registry requirements presented in 3.2, the following metrics were chosen to determine the suitability of the implementation:

• **Response time for read**: As the Domain Registry is a critical component in the call establishment process, the time it takes to perform a read should be small, in the order of the tens of ms. We will test the evolution of this metric as the load on the server increases.

• **Number of concurrent requests**: A large Service Provider is expected to have a large number of users, which will result in a high number of requests to the Domain Registry. Thus the Domain Registry should be able to scale to accommodate a large number of requests/s while providing a reasonable response time.

• **Error rate**: Measured in number of the requests that fail to be successfully replied to within the timeout period (defined as 5s). This value should be zero.

With these metrics in mind, two types of tests were performed: performance and scalability tests. The performance and scalability tests were conducted using 1, 2 and 3 application servers. The number
of database servers was always 4 in order to maintain data availability and force the server to became
the bottleneck of the system. For each number of servers, 10 tests were conducted varying the rate from
200 requests/s up to 2.000 request/s with a step of 200 requests/s. Each HTTP connection was used to
issue 10 requests and 1.000 connections were used, totalling 10.000 HTTP requests per test point. The
option to have each HTTP connection issue 10 requests was due to the fact that the Domain Registry’s
client, the Connector running on the Message Node, uses connection pooling and reuse with HTTP
persistent connections. Every test was repeated 50 times. Each data point is the average of all these
runs. The tests are interleaved and were performed over the length of a few days to prevent eventual
effects due to time of day network and VMs traffic. The same exact tests were performed with only
1 database node for the purpose of seeing how the Domain Registry response times and concurrent
requests were affected by a smaller database cluster.

A summarization of the tests scenarios is presented in Table 5.1. As can be seen, for each test
scenario 10 tests were performed combining into a total of 40 tests.

<table>
<thead>
<tr>
<th>Test #</th>
<th># Load balancers</th>
<th># Application servers</th>
<th># Database servers</th>
<th># Requests/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>[200, 2000] step: 200</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>[200, 2000] step: 200</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>[200, 2000] step: 200</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>[200, 2000] step: 200</td>
</tr>
</tbody>
</table>

Table 5.1: Test scenarios

The second part of our evaluation aimed at testing the failover processes of the Haproxy load bal-
ancers. For that reason, we tested the two following scenarios:

- **Haproxy process fails**: In this scenario we purposely stop the Haproxy process to see that in fact
  the backup load balancer assumed the role of master load balancer;

- **Primary load balancer fails**: Here, again on purpose, we suddenly stopped keepalived’s process
to make sure that the backup load balancer claimed the shared IP address.

## 5.2 Tests methodology

Throughout this section we report our methodology for evaluating the Domain Registry prototype imple-
mentation. First, we explain how the Domain Registry was deployed, followed by a summary description
of several load testing tools and some of the challenges we faced during the evaluation, and finally, we
end up with a short description of rubyPerf\(^1\), a tool that we developed to aid the process of HTTP load
testing web servers.

\(^1\)https://github.com/ruijose/ruby-perf

50
5.2.1 Domain Registry deployment

The Domain Registry was deployed on DSI’s Tagus Park datacenter, using 9 VM with 1vCPU and 2GB RAM each. The VMs were assigned the roles described in Figure 3.3: 4 Cassandra DB nodes, 3 application servers and a two load balancers in active/passive configuration. All requests are sent to the load balancer, that distributes them in round-robin through the 3 application servers.

The Operating System used was Ubuntu 14.04 64bit and all software was deployed using Docker 1.6.2. The load balancer uses haproxy 1.5. The Cassandra DB was deployed using version 3.5 with a replication factor of 3. The application server was deployed using the Spark Java framework 2.2. The Domain Registry version used was R 0.2.0.

The load testing tools were run on a server with 2 Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz CPUs (total of 32 cores), 128GB of RAM running Debian 8.2.

5.2.2 Testing tool choosing

After reviewing and testing several open source load testing applications, we ended up choosing httperf\(^2\) along with Autobench\(^3\). Autobench is a wrapper around httperf that executes it several times with different parameters and parses the results at the end of each test. Autobench’s goal is to load test web servers by increasing the load on each test to understand at which point the server becomes saturated.

In order to support the choice for httperf, we present a succinct description of the other tools that we run and analyzed while testing the Domain Registry.

- *ApacheBench (ab)* is a very basic tool that can be highly useful to evaluate an API endpoint after an optimization was performed. However, ApacheBench does not allows us to define an intended request rate;

- *Apache Jmeter* is a powerful Java application to simulate heavy loads and measure performance of several networked applications, such as databases and web servers. It is extremely flexible when used with plugins. However, it seems that it was designed mainly to simulate users interaction with websites.

- *Tsung*, as Jmeter, does a great job simulating users using a web site. It provides an extensively configuration file that can be used to realistic reproduce a user using the web page. For instance, it allows defining the probability of a specific user hitting a certain menu.

- *Httperf* attempts to send a continuously stream of requests at a given speed regardless if they are answered or not. This allow us not only to determine at which requests/s the server is saturated but more importantly to see the behaviour of such server under load.

Since Jmeter and Tsung are better suited for testing users using a website, and the Domain Registry does not interact with users, we chose httperf as the load testing tool to evaluate our prototype.

\(^2\)http://www.labs.hpe.com/research/linux/httperf/
\(^3\)http://www.xenoclast.org/autobench/
Additionally, in order to evaluate the load balancer failover, and since we only wanted a continuously stream of requests during a specific time, we programmed the following `curl` script:

```bash
#!/bin/bash
counter=0
while sleep 1
  do
curl -sL -w "$counter %{time_total}\n" http://server/hyperty/user/rui
  -o /dev/null | tee -a curl_times.out & counter=$((counter+1))
done
```

This script allows us to verify the transitions between the master and the slave load balancer because it sends a continuously stream of requests and register the time it took to fulfill such requests. As a consequence, by analyzing in how much time responses with a value of zero arrived, we know that it is equal to time between master and slave transitions.

### 5.2.3 Evaluation challenges

`Httperf` presents some performance limitation that must be taken into account in order to understand the results obtained. In particular, `httperf` limits the number of concurrent connections (due to file descriptor limits). If the server is unable to keep up with the request rate, `httperf` will eventually run out of TCP connections and will be unable to sustain the request rate. This happens because, by default `httperf` on Linux compiles with a maximum number of open file descriptors (sockets) equal to 1024. Consequently, this forms an enormous limitation since the bottleneck will become the client and not the server. Thus, it is not possible to issue high bandwidth traffic with many concurrent connections to web servers. In order to solve this problem we changed `/usr/include/bits/typesizes.h` file by amending the following line `#define FD_SETSIZE 1024` to `#define FD_SETSIZE 65535`. We then applied the changes by recompiling `httperf` and the problem was solved.

Another problem appeared while using `Autobench`. As stated previously, `Autobench` executes `httperf` several times, parse the results and generates CSVs documents. However, it seems that `Autobench` developers thought that `httperf` output was a little verbose and summarized the error output in one single parameter called ‘errors’. As a consequence, while analyzing `Autobench` CSV output, we could not understand what kind of errors had occurred. To counteract against it, we develop our own wrapper around `httperf` with some additional functionalities that will be presented in the next section.

### 5.2.4 Development of `rubyPerf`

As a means to solve the above mentioned problems with `Autobench`, we decided to develop our own tool that uses `httperf` to evaluate the performance of HTTP web servers. It is a Ruby command line application named `rubyPerf` that is available as open source at Github @rujisse account. `Rubyperf` acts exactly like `Autobench` but does a better job parsing the `httperf` results, and moreover, a couple additional features were programmed. It basically differs from `Autobench` in the following topics:
• It parses every single value from httperf output. While Autobench combine many metrics from it, rubyPerf extracts all of them to a CSV file.

• It allows us to define the number of times a test should be repeated and the time interval between each test (default is 3600s);

• It generates the average of all the metrics in the CSV documents generated from each repetition test;

• Finally, rubyPerf also generates Gnuplot graphs for the average metrics.

5.3 Domain Registry evaluation

In the next sections we show and examine our Domain Registry evaluation results. The first section focuses on load and scalability tests and the second on the load balancers active/passive configuration failover processes.

5.3.1 Load and scalability tests

The succeeding line graphs depict the first three scenarios from Table 5.1. Each point on the graphs represents an individual test type which is the average off such test type repetitions. For instance, in graph from Figure 5.1 the point (200,200) illustrates the first test's result in which was issued 200 requests/second and the server indeed sustained the 200 requests/second.

The graph from Figure 5.1 represents the relation between the solicited request rate and the effective request rate, with the Domain Registry infrastructure varying from one to three application servers. We can see that with three application servers (purple line), the Domain Registry becomes saturated at around 1750 request/second. After that it stabilizes on that value. With two application servers deployed (blue line), our prototype becomes saturated at 1200 request/seconds, and with one application server (yellow line) it saturates at around 700 requests/seconds. We can see that, in fact, the Domain Registry scales horizontally whenever more nodes are added. From Figure 5.1 we observe an increase in capacity of approximately 600 requests/second when a new server is added. The line $x=y$ represents the ideal scenario where the system respond successfully to all requests.

In the following graphs we will use the effective request rate instead of the solicited request rate. Figure 5.2 presents the average response rate for an increasing request rate. Considering that the client and server are in the same network, a value of $\approx 15$ ms is considered acceptable since it will not delay the reThink framework. In earlier tests, not represented here, with the client separated from the server, we got values below 50 ms, which is also acceptable since the two were separated by the Internet. As excepted, when the request rate increases past the server capacity, the server becomes saturated and the average response time increases. Again, each point represents the average of a single load test type. As an example, when we tried to perform 2000 requests/second with only one application server
(blue line's last point), and as expected from the last graph, it saturated at \( \approx 700 \) requests/s with an average response response delay of \( \approx 450 \) ms.

![Figure 5.1: Demanded request rate.](image1)

![Figure 5.2: Average response rate.](image2)

Figure 5.3 represent the duration of each TCP connection. Since we perform 10 request per TCP connection, this graph is basically the same as in Figure 5.2 but with average times multiplied by ten. Finally, from Figure 5.4 we conclude that, although we should have no errors, when the web servers become saturated some requests are not fulfilled in less than 5 seconds. This value (5 seconds) was defined by us as the time we think anyone is willing to wait for a response. The errors we see in Figure 5.4 were not server or client errors. Those requests would probably be successfully if we did not set a timeout value. However, we can see that, until the servers become saturated there were no errors.

The next step was to evaluate how the Domain Registry would perform with only one database node. This significant drop of the cluster’s size was tested because, first we get to know how the database cluster scaled, and secondly because in our deployment proposal for the reThink project partners, we presented a simple deployment with only one database node and a more complex one with four nodes.
From both Figures 5.5 and 5.6 we can see that with only one database node, the database is obviously the bottleneck of our infrastructure. In spite of that, the Domain Registry was able to sustain up to 1000 requests/second with average response times similar to the ones presented in Figure 5.2.

5.3.2 Load balancers failover tests

Testing the failover mechanism of Haproxy was done using the curl script mentioned above. We run the script during 60s and after ≈ 20 seconds we stopped first the Haproxy (Figure 5.7) and then the keepalived process (Figure 5.8) on master node. Regarding the load balancer fail, we set keepalived to monitor Haproxy every 5 seconds. That is why there is a 5 second gap in the first graph in Figure 5.7. However, this value was used just for testing to actually see the transition. In production this value will be decreased to 2 seconds. That was the only value that was manually set by us. The other three transitions that we see on both Figure 5.7 and 5.8 are related to VRRP advertisements. When the backup node stops receiving this advertisements it claims the shared IP address and becomes
the master node (Figure 5.8). While assuming the master node role, if the backup node ever starts receiving VRRP advertisements again, it elects the first node as master (because the master was set up with a higher priority level) and transits back to being the backup node, in a always listening, passive configuration (second transition of both Figure 5.8 and 5.7).

5.4 Monitoring with Riemann

Throughout the next two sections is described our evaluation of the monitoring system that was deployed by us in order to perform monitoring over the main Domain Registry architecture. Our objective was to confirm that the Domain Registry servers and the load balancer were indeed sending events to the Riemann server and that we can see them (and their changes) on the Riemann dashboard.
5.4.1 Riemann server and dashboard deployment

As we already stated before on Chapter 4, the servers from the first architecture periodically send events to a Riemann server that works as a database that the Riemann dashboard queries. As a result of the limited number of Virtual Machines (VMs) available, we deployed, both the Riemann server and the Riemann dashboard on the same server using Docker containers. This server was the same as described above to run the load testing tool. In an ideal production scenario each of the Riemann servers (the main and the dashboard) would be deployed in separated machines. This scenario is depicted in Figure 5.9.

5.4.2 Riemann evaluation

In order to perform the Riemann monitoring system evaluation, we used the testing tool above mentioned called Httperf solely for the purpose of visualizing changes on the dashboard and confirming that the
Riemann server was receiving events. Figure 5.10 shows the Riemann dashboard right after being deployed. It comprises three dashboards splits, each of which comprising the resource level state of each Domain Registry server. At that moment it has not received any load yet. It shows the levels of CPU utilization, RAM and disk usage, and CPU load average for each of the servers. After a while we issued two load tests separated by a couple minutes. The first test was issued with 1000 requests/seconds and the second with 500 requests/second. Figure 5.11 shows the same dashboard page while the three Domain Servers were under load. We can see in each of the three splits that each Domain Registry server is receiving requests by analyzing the CPU usage line on the three graphs. Moreover, when both of the tests end, the CPU usage lines decrease to the normal state while not serving requests. The other lines present in the pictures did not change because those resource properties were not affected by the load tests.

Figure 5.10: Resource levels after deployment

Figure 5.12 represents a live statistics page from the Haproxy load balancer. It is comprised of four splits: 4xx, 2xx and 5xx response codes and the number of requests/second at the moment of the test. The Figure was taken during the second load test. As excepted, since we were retrieving existing
resources, it does not show 404 response codes. HTTP status codes 5xx were also inexistent which means that the server was not aware of any errors and that was being capable of serving the requests. Lastly, predictably, the requests/second split shows that we were indeed issuing 500 requests/second to the load balancer. These splits show a grey color because everything was working as expected. As we explained on Chapter 4 these splits would turn red if, for instance, the requests/second exceed 1000 requests/second. It represents a great feature since it let us know that we are serving a great number of requests per second and allow us to perform safety measures to ensure that the system will continue to work properly as the load continues to increase.
5.5 Chapter summary

The main conclusions of the Domain Registry evaluation presented in this chapter are:

1. The response time of our REST API is for each request $\approx 15$ ms. We shown that before the servers become overloaded the response time in on average $\approx 15$ ms. Considering that both the client that issues the requests and the infrastructure are deployed in the same network, these values are acceptable and will not delay the reThink framework.

2. When our infrastructure is deployed with only one database server, the bottleneck of our system becomes the database as excepted. Even so, we are able to sustain $\approx 1000$ requests/second with response times similar to the ones achieved using a database cluster.

3. The Domain Registry infrastructure scales horizontally as more machines are added.

4. The failover recovery process of the load balancers HA setup is fast and works as excepted.
Chapter 6

Conclusions

This document describes the Domain Registry, a reThink’s project internal component. We aimed at defining an easily scalable and fault tolerant architecture that allow CSPs to run it with minimal downtime. The Domain Registry is available as open source to any CSP that wishes to try out and evaluate the reThink framework and its setup is fairly simple using Docker. The success of the Domain Registry will be measured mostly by the performance of the core REST architecture and the distributed database. This chapter reflects on our contributions to reThink and discusses future work.

6.1 Summary

Our approach to develop a highly available and scalable distributed system began with an evaluation of P2P systems and architectures. The idea behind a P2P Domain Registry was each CSP contribute to a DHT by providing one or more nodes. Although being an ideal design by their scalability and fault tolerance properties, we soon understood that the major disadvantage of these kind of systems - the loss of control over where data is stored - would not work in reThink because CSPs want to control where their data is stored. Moreover, the existence of and the lack of full proof solutions to some security attacks, such as the Sybil [56] and Eclipse [57], also discouraged the use of a P2P Domain Registry. We then proceed to evaluate client-server systems and decided to implement the Domain Registry core architecture as a REST API server that would allow the creation, change and deletion of user’s Hyperties. In order to achieve the performance requirements, we allow the Domain Registry REST server to be replicated across several machines that will serve content in a round robin fashion, mechanism that will be performed by two load balancers in a failover state. Furthermore, load balancers are responsible for actively monitoring the state of each of the servers and stop sending requests to the failed ones. We decided to implement layer 7 load balancers which will allow us to interpret the requests in the load balancer. Although we are not currently using all the advantages of a layer 7 load balancer, we leave the architecture prepared for future layer 7 capabilities improvements. Regarding persistent data store we discussed and analysed several scalable database proposals and end up using a Cassandra database cluster that can be scaled to several nodes. Since we have chosen a distributed database,
we matched the Domain Registry requirements with the CAP theorem and conclude that the Domain Registry would be an AP system, that is, a highly available and network partition tolerant distributed system.

In order to support monitoring and centralized log management, we configured, programmed and deployed a second architecture that will interact with the first one and generate graphs and near real time information about the first architecture behaviour. We began to study pushing and pulling architectures, and for scalability reasons, we end up using for both logs and monitoring push-based systems in which the monitored components periodically sends events and logs for the analysis systems.

We performed our evaluation on DSI's virtual machines and conclude that the Domain Registry scales horizontally when more nodes are added and that it favours response times of 15ms while serving user requests. In worst case scenario, that is, when a load balancer fails, we shown that the recovery process is quick, preventing the clients form using the service only a couple of seconds.

Thus, we achieved the main goal that we set out at the begging of this dissertation: develop a highly available and scalable service for Hyperty reachability information with fast response times.

6.2 Future work

While we have achieved our set of goals, this work may still be improved. As the Domain Registry and its client, the Registry connector are two architectures deployed internally within a CSP, the data generated by the Domain Registry could be serialized in another format than JSON, without effecting other reThink components. JSON data favours a human readable/editable format that can be parsed without knowing any schema in advance. However, since the Domain Registry is not intended to be used by the rethink’s end users, we would like to evaluate the use of another data serialization format, such as, Google’s Protocol Buffers. Protocol Buffers, known as protobuf, provide a very dense binary output, and thus, a very fast processing without losing information. However, data is internally ambiguous, and thus, it requires a knowing schema to perform data decoding. As a consequence of the low overhead introduced by the Protocol Buffers, we think that the Domain Registry may leverage them as the system scales and the number of message between the Registry Connector and the Domain Registry increases.

Currently the Domain Registry is deployed within DSI's virtual machines. However, we would like to deploy the whole architecture in a IaaS environment, such as Amazon’s AWS or Google’s Computer Engine, and perform a comparison analysis of the performance of both deployments. Moreover, related to that deployment we would like to perform a Domain Registry deployment cost analysis in such IaaS environments. Although the monthly cost of deploying the Domain Registry in a public cloud service can be expensive, we can leverage the near unlimited scalability and an ‘on demand’ service that enables the provisioning of resources whenever they are required.
Bibliography


sourceforge.net/src/rfc-06-draft.html.


global-scale overlay for service deployment. IEEE Journal on selected areas in communications
22(1) (2004) 41–53

[16] Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distri-
and Architectures, ACM (1997) 311–320


lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications,

random trees: Distributed caching protocols for relieving hot spots on the world wide web. STOC
’97, ACM (1997) 654–663


[21] Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for
large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM International Conference on


[23] Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed systems: Concepts and design. 5th
edn. Addison-Wesley Publishing Company, USA (2011)

language (wsdl) 1.1 (2001)

services web: An introduction to soap, wsdl, and uddi. IEEE Internet Computing 6(2) (March 2002)
86–93


