Domain Registry: a Highly Available Infrastructure for Hyperty Discovery

Rui Mangas Pereira

Instituto Superior Técnico, Universidade de Lisboa
ruimangaspereira@tecnico.ulisboa.pt

***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. Align with this, we present the Domain Registry, a core component of the European 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 as a result, allowing the communication between such users. Moreover, 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. Our proposal, which is comprised by a core application and a failure prone 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.**

I. 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 [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 [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 maintain large systems without manual intervention from infrastructure administrators [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 [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 [6]. 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 increased.

The motivation behind this work was the emergence of the European founded research project reThink[1]. Its goal is 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 project addresses these challenges by proposing the use of a service, deployable in a runtime environment, in a end-user device (Web Browser or native app) or in the network, called Hyperty. Hyperties are reusable scripts that implement a service logic (e.g. Javascript file) responsible for interoperate different applications developed by different people using the reThink framework. They have communication capabilities and can be considered as communication endpoints. This also implies that Hyperties are reachable and, for example, in a Human to Human communication (Audio call) Hyperty addresses could be considered the reachable point of communication of the person associated with the Hyperty service used to initiate that call. We can consider that Hyperties are provided by a Service Provider, and authenticated by a Identity Management functionality. Our work focus on the development and evaluation of a Hyperty directory called Domain Registry. It can be seen as a directory service that facilitates management and lookup of the Hyperties that are

[1] https://rethink-project.eu/

being run on user's devices. If the Domain Registry becomes unavailable, users can not find another user's Hyperties, and therefore a communication can not be established. As a result, users from that specific service provider become 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 identifiers, 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. 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.

II. Background and related work

The Domain Registry is a highly available distributed system responsible for providing an important service within the reThink framework. Peer-to-Peer (P2P) systems, almost often associated with scalability and availability, 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, bandwith, 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.

Throughout the first half of the last decade, systems such as Chord [8], Pastry [9], Tapestry [10], CAN [11] and later Kademlia [12], have proposed efficient mechanisms to locate the node that stores a certain data item, and moreover, techniques to deal with the constantly entering and leaving of peers (knows as churn rate), while successfully serving requests. However, and although they have been successful, these kind of systems suffer from a big problem: the loss of control over where the data is stored. Furthermore, since there is not a centralized unit, useful global values calculation, such as, total number of users, introduce additional communication cost and delay because all nodes have to join the process and agree on a final value.

On the other hand, in client-server environments the server assumes the central location where users share and access network resources. This results in centralization of services, not robust architectures and possible network congestions. In order to resolve these questions, server load balancing techniques are employed. 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.

Web services are a common way to develop client-server systems, and thus, enable the distribution of content between two heterogeneous systems. They often are defined as an architecture style for client-server application-to-application communication using existing Web protocols, such as Hypertext Transfer Protocol (HTTP) [13]. Many well known platforms used my million 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) [14], Universal Description, Discovery, and Integration (UDDI)) [15], and Simple Object Access Protocol (SOAP) [16]. At the same time, REST architectures have been gain popularity by there 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.

Highly available architectures are known to possess several servers and applications to reduce the likelihood of system failures. Server monitoring and centralized log management are two techniques available to aid the maintainers and developers to prevent and act proactively on possible failures. Both of these techniques are defined by using pull of push based systems in order to collect or receive resource usage metrics and logs from applications and servers. While, in the past, most solutions were based on centralized gathering, also known as pull based systems, nowadays, due to the very dynamic nature of servers and applications, decentralized data collectors applications propose a push based model where data is sent to centralized units for further processing. Fundamentally, it consists of a combination between decentralized collectors and a centralized server responsible for aggregating, parsing and storing such data.

III. Architecture

Our main design goal is to provide reThink with a highly available architecture for one of its most important and critical components, the Domain Registry. 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. This service stores, for each Hyperty instance, the data that enables other applications to contact it. It 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 service 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 systems 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, will 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 will 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.

A. Core architecture

In order to comply with the functional requirements, 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, 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, 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. Thus, as can be seen from Figure 1, 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.

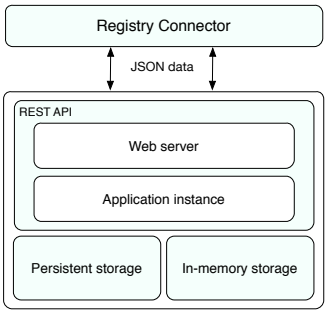

Fig. 1. Domain Registry architecture

Therefore, the 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), and a description of the state transition of the data elements. The API endpoints that were defined are presented in the Table I. 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

#	Endpoint	GET	PUT	DELETE
1	/hyperty/user/:user/:hyperty	Returns a Hyperty that belongs to a user	Creates or updates a Hyperty and associates it to a user	Deletes a Hyperty
2	/hyperty/user/:user	Returns all the Hyperties that belong to a user	N/A	N/A
3	/hyperty/user/:user/hy?resources=com&dataSchemes=video	Returns the Hyperties that contain the specified resources and dataSchemes	N/A	N/A
4	/hyperty/user/:user/hy?resources=com	Returns the Hyperties that contain the specified resources	N/A	N/A
5	/hyperty/user/:user/hy?dataSchemes=video	Returns the Hyperties that contain the specified dataSchemes	N/A	N/A

TABLE I
DOMAIN REGISTRY API SPECIFICATION

the endpoints are utilized to perform advanced searches based on Hyperties characteristics, i.e. DataSchemes and Resources.

B. 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 III) 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 following topics will introduce an architecture that was designed to meet such requirements.

Figure 2 depicts the overall deployment architecture of the Domain Registry. It comprises two load balancers in failover mode, and at least, three application servers and four 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 database node. These two alternatives allow developers to rapidly test the API with the purpose of getting to know, and experiment, the available endpoints.

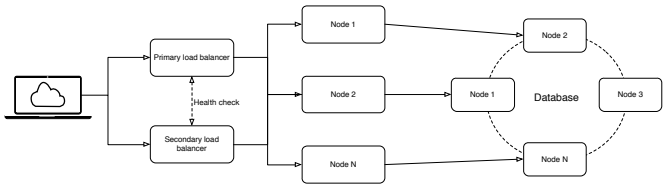

Fig. 2. Domain Registry main architecture

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 Secure Sockets Layer (SSL) connections.

C. Load balancing

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 a operational server.

To achieve this goal, we used the Virtual Router Redundancy Protocol (VRRP) [17], 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 can not 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.

D. Database

Based on our requirements presented in Section III, 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). 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. As studied in Section II, 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.

E. 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. However, 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 various alternatives on how to implement Transport Layer Security (TLS)/SSL security between the client, the load balancer and the REST application servers. The first scenario we studied 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 off 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. Secondly, scenario know as SSL/TLS offloading works by having 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 need to be properly scaled to meet the overhead introduced by the SSL handshakes [18]. We end up choosing the last scenario because 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.

F. Network Management Architecture

Last but not least, we present an architecture aimed at resolving the maintainability non-functional requirement present in Section III. It is system directly connected to the deployment architecture which aims at providing network management tools, i.e. monitoring and centralized logging.

In Figure 3 is represented the overall monitoring and centralized logging architecture of our infrastructure. It incorporates fives 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.

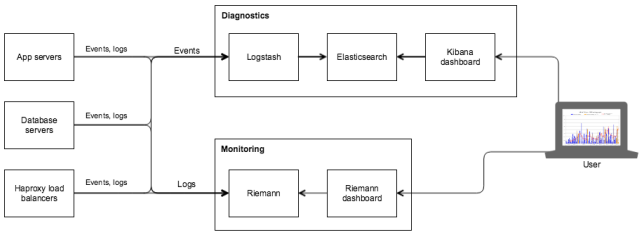

Fig. 3. Monitoring and centralized logging.

G. 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. Therefore, a push based system was design that collect the following (most important) metrics: resource level events, requests per seconds, response codes, writes and reads and average response times.

H. Centralized Log Management

Gathering and parsing logs from multiple sources have several problems since most of the sources generate logs in different formats. 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.

IV. Implementation

This section 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 III images, were important to perform some internal actions. 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[2]. Not to be confused with Apache Spark, Spark Framework, inspired by Ruby's Sinatra[3], 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. Regarding the executing 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.

[2]http://sparkjava.com/

[3]www.sinatrarb.com

A. Deployment architecture

For the Deployment Architecture 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 [19] 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 Operative System (OS).

1) Load balancing: 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 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.

On the backend of the load balancer with 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.

In order for overcome a possible load balancer failure, floating IP addresses were used. To achieve this goal, we used a tool called *keepliaved* [20] 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 *vrrp_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.

2) Database: We chose a NoSQL database to persistently store the data about each Hyperty instance. As the Consistency, Availability and 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 [21]. 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. We end up using Cassandra database 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. 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.

3) Monitoring: 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.

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. Starting by the Haproxy load balancers, we develop a program that first, scrapped its statistics web page to a Comma Separated Values (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[4], wrapped it into a script and again sent its observed values to the Riemann server.

4) Centralized logging: 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) [22]. Consequently, whenever a search occurs, only the index is queried, rather than the original documents. For that purpose we used Elasticsearch (along with Logstash), which is a full text highly available search engine based on Apache Lucene [23]. Elasticsearch 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. As a means of shipping logs to Logstash, we installed in each of our servers another ELK stack underlying product called Beats[5]. Beats are lightweight processes written in Golang that capture and 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.

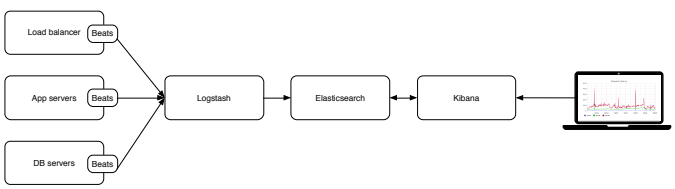

Fig. 4. Centralized logging architecture.

[4]https://github.com/nethacker/usagewatch
[5]https://www.elastic.co/products/beats

V. 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*). 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, 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.

The second part of our evaluation aimed at testing the failover processes of the Haproxy load balancers. 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.

The succeeding line graphs depict the first three scenarios that were evaluated. 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 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 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 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 6 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. Finally, from Figure 7 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 7 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.

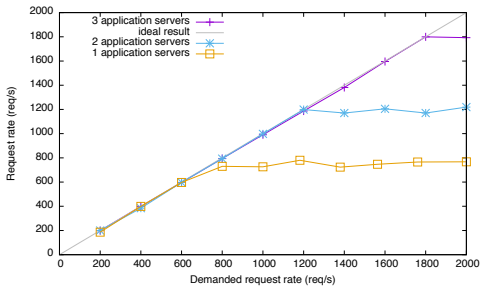

Fig. 5. Demanded request rate.

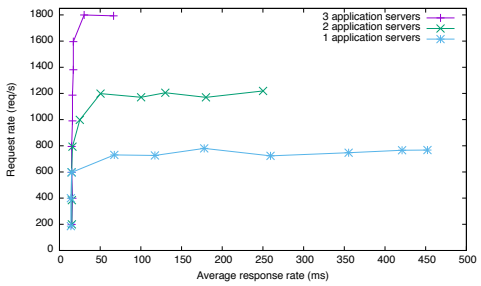

Fig. 6. Average response rate.

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

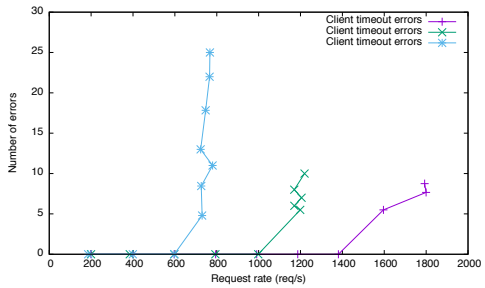

Fig. 7. number of errors.

presented a simple deployment with only one database node and a more complex one with four nodes. From both Figures 8 and 9 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 6.

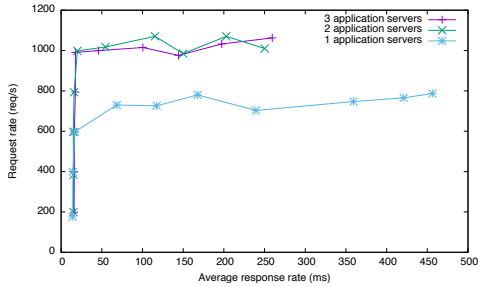

Fig. 8. Average response rate with only one database node.

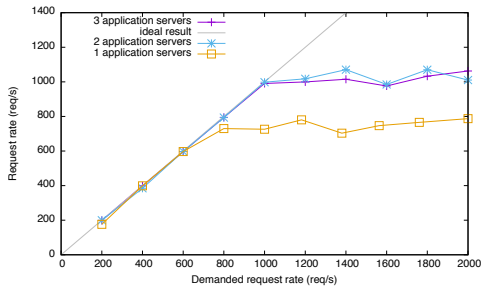

Fig. 9. Solicited request rate with only one database node.

Testing the failover mechanism of Haproxy was done using the *curl* script mentioned above. We run the script during 60s and at $\approx$ 20 and 40 seconds we stopped first the Haproxy (Figure 10) and then the keepalived process (Figure 11) 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 10. 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 10 and 11 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 11). 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 11 and 10).

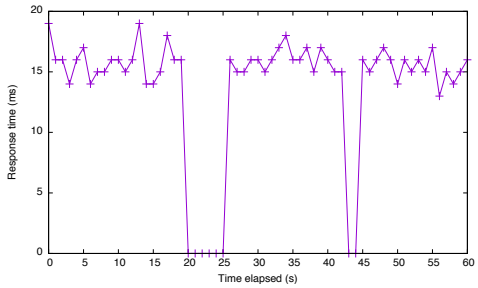

Fig. 10. Haproxy software failover.

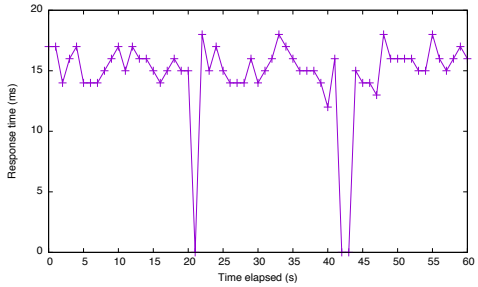

Fig. 11. Master node failover

5) Riemann monitoring testing: 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 12 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 13 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.

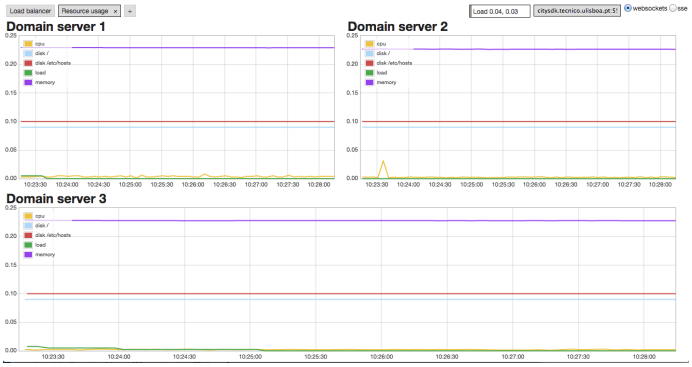

Fig. 12. Resource levels stable

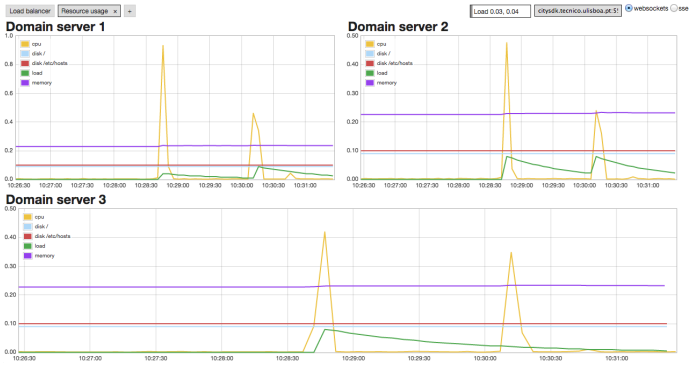

Fig. 13. Resource levels under load

Conclusion

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 Distributed Hash Table (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 [24] and Eclipse [25], 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 high 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.

VI. 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 affecting the other reThink's 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 propose an evaluation of the utilization of another formats, such as, Google's Protocol Buffers. They provide a very dense output, and thus, a very fast processing. However, data is internally ambiguous, and thus, requires a knowing schema to perform data decoding.

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 environment.

REFERENCES

[1] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier, "Cluster-based scalable network services," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 78–91, 1997.

[2] R. Rabbat, T. McNeal, and T. Burke, "A high-availability clustering architecture with data integrity guarantees." in *Proceedings of the 3rd IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2001, pp. 178–182.

[3] A. Jindal, S. B. Lim, S. Radia, and W.-L. Chang, "Load balancing in a network environment," Dec. 4 2001, uS Patent 6,327,622.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] R. L. Grossman, "The case for cloud computing," *IEEE IT professional*, vol. 11, no. 2, pp. 23–27, 2009.

[6] K. Kent and M. P. Souppaya, "Sp 800-92. guide to computer security log management," Gaithersburg, MD, United States, Tech. Rep., 2006.

[7] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *Proceedings of the First International Conference on Peer-to-Peer Computing*, 2001, pp. 101–102.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2001, pp. 149–160.

[9] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, 2001, pp. 329–350.

[10] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on selected areas in communications*, vol. 22, no. 1, pp. 41–53, 2004.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2001, pp. 161–172.

[12] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," pp. 53–65, 2002.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," Tech. Rep., 1999.

[14] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web services description language (wsdl) 1.1," 2001.

[15] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: An introduction to soap, wsdl, and uddi," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/4236.991449

[16] G. Mein, S. Pal, G. Dhondu, T. K. Anand, A. Stojanovic, M. Al-Ghosein, and P. M. Oeuvray, "Simple object access protocol," Sep. 24 2002, uS Patent 6,457,066.

[17] S. Nadas, "Virtual router redundancy protocol (vrrp) version 3 for ipv4 and ipv6," Internet Requests for Comments, RFC Editor, RFC 5798, March 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5798.txt

[18] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008.

[19] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2014.

[20] P. Hollenback, "Improving network reliability with keepalived," 2008, http://www.keepalived.org/pdf/UserGuide.pdf.

[21] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[22] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, vol. 24, no. 5, pp. 513–523, 1988.

[23] A. Jakarta, "Apache lucene-a high-performance, full-featured text search engine library," 2004, https://lucene.apache.org/core/4_0_0/.

[24] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 251–260.

[25] A. Singh *et al.*, "Eclipse attacks on overlay networks: Threats and defenses," in *In IEEE INFOCOM*. Citeseer, 2006.