Architecture for a Fully Decentralized Computing Infrastructure Providing a Platform as a Service Using Commodity Hardware

by

Dany Wilson

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Ottawa

Contents

Chapter 1

Introduction

This thesis presents an architecture for a fully decentralized computing infrastructure built using volunteered resources. This computing infrastructure is composed of a collection of geographically distributed commodity hardware (personal computers) connected via the Internet. The applications deployed using this computing infrastructure, are then hosted by a collection of commodity hardware forming an computing platform akin to a cloud computing infrastructure providing Platform-as-a-Service.

In this chapter we present the motivations that drove this research effort, as well as a definition of the problem we are addressing in this thesis. Then, we outline our contributions and present the different chapters included in this monograph.

1.1 Motivations

In this section we present the motivations that drove our research effort. We were motivated by the current state of Cloud service providers, and the current state of independent web communities.

Cloud Service Providers

Cloud Service Providers (CSP) offer one of the most prominent computing platform online, with which we can create and deploy applications. Although being at the height of its popularity, some security concerns still exist which impedes this paradigm shift. This versatile computing platform, is characterized primarily by its service provisioning model, which offers, virtually, to the consumer an "infinite pool" of resources that can be contracted at any instant to mitigate the fluctuations in the workload. Thus this business model is centered around charging the consumer based on its consumption, this is also known as Utility Computing (UC). A consumer also has the ability to choose from various degrees of responsibilities that he/she wishes to assume. Which can oscillate

from managing a complete Operating Systems (OS), all the way to managing only at the application layer, by choosing the corresponding service model. Amongst all the benefits of using such platforms, some concerns still exists in regards to Cloud Computing. These concerns are geared toward privacy of the data, privacy of any intellectual property and security against possible attackers. Largely because, in order to use these platforms, it is required to move data and applications to their data centers, where resources are provisioned. Another concern, is the potential that a cloud service provider could, either out of strategic positioning or because a tyrannical authority, exercise censorship. This is Orwellian in nature, but not completely farfetched, for example: given two applications that provides similar services, both are hosted on the same cloud service provider, if a competitor can persuade (by any means) the provider to favor its application, the latter could exercise direct or indirect censorship in several ways. One of which could be to prioritize the traffic of one application over the other, or to reduce its priority to access the computing resources (or a specific Application Programmable Interface (API)).

The last concern we present is relative to the benevolence of the individuals employed by this provider and whether they are malicious or negligent in any way. After all they do have direct physical access to the data centers.

Consequently, a trust relationship must be established between the client and the CSP. As of now, the clients of these services do not possess any way to ensure the diligence and the benevolence of these providers other than leaving it up to faith, but so is the case with the majority of the private service providers in many domains. This is due to the private nature of these providers and the fact that they are not legally bound to reveal any of the mechanisms and/or protocols enforcing the security and privacy within their organization.

Web Communities and Communities in General

Local and Web Communities often hosts applications (forum, image board, etc.), which are aimed at, and supported by a small core of dedicated individuals. Normally, these communities need to rely on donations to account for the fees incurred by hosting and serving the community. All of the members of these communities, have a personal computer (to access the application) and have Internet access. With the increase in bandwidth of domestic Internet service providers, and the performance of most household computers augmenting, can their cost of operations be reduced?

The education system can be considered as an interesting example of community where different school boards form micro-organizations. One aspect of these micro-

organizations is their computing resources, primarily composed of personal computers and usually accessed during office (school) hours. In some cases they remain idling through the night, or are simply turned off. Could they monetize their resources during the off-hours, and reduce their cost of operation or at the very least being able to make the resources generate income that could cover their initial purchase cost or extend their usefulness.

Out of these two different perspectives on communities, a common potential desire of recycling their computing resources emerges. Recycling computing resources exhibits many incentives, one of which (but not limited to) is financial restitution of the initial investment to acquire the resources. Prolonging the usefulness of the computing resources, in this era of consumerism, is an important incentive, especially given the eventual scarcity of the resources used to mass produce them. In a similar line of thought, recycling computing resources can help intellectual advancement by making them available to different scientific enterprises to run experiments or computationally intensive tasks. There is benefit to provide means to smaller communities with a technology that enables them to recycle their computing resources and make them profitable (at the very least to recover their initial cost of purchase either in terms of financial restitution or by prolonging their usefulness). We also see potential benefits in solutions to cater all types of requirements, be it in terms of privacy and data ownership or simply to enable a private organization to be part of a fair competitive market and avoid possible censorship from an oligarchy.

1.2 Problem Definition

In this section we present the definition of the problem addressed in this thesis. We define this problem by formulating a series of requirements that a solution should fulfill.

From the motivations presented earlier, we can extract a list of the essential requirements that such a candidate solution should address:

Requirement 1 Given a collection of heterogeneous commodity personal computers, we need to be able to use this system to deploy an application or multiple applications.

Requirement 2 Using this system should not force any third-party to provide services (except domain hosting, everything else should be accomplish by the collection of contributing resources), and thus be self-contained as much as possible.

Requirement 3 The system should be resilient to resources failing, and/or leaving. Thus, it should be fault-tolerant and should not introduce a single point of failure for security purposes, in order to resist DoS type of attacks and possible censorship.

Requirement 4 The system should not require from its user to acquire any special (or dedicated) equipment, but rather recycle the current resources available.

Sub-Requirement 1 The memory footprint should be small, as to be able to construct this system from lower-end devices.

Requirement 5 The system should be able to provide scalability to the application deployed, or the ability to dynamically adjust the amount of resources available and account for the fluctuation in demand. As a consequence, it must provide dynamic membership capabilities to all applications.

1.3 Contributions

Within the context of this thesis we propose a solution for this problem in the form of an architecture and with this proposition we make the following contributions:

Contribution 1 Propose a fully-decentralized collaborative system that provides a web computing platform.

Contribution 2 Propose a candidate API that addresses the minimal requirements of this computing platform, but more generally one that is suitable for (PaaS) Platform-as-a-Service.

Contribution 3 The system we propose make use of light virtualization to abstract the specificities of the contributing resources, and to isolate the host environment from the contributed environment.

Contribution 4 The system that we propose is minimally intrusive and leaves a small memory footprint, thus enabling it use on lower-end computers, even microcomputers.

1.4 Outline

In this section we present the outline of the thesis and the content of each one of the chapters.

Chapter 2 Background
This chapter presents the background material concerning Collaborative Systems in general, the basics of Cloud Computing, the basics of Peer-to-Peer technologies, and the current Virtualization technologies.

Chapter 3 Related Work
This chapter presents two similar systems, namely Cloud@Home and P2P Cloud System and evaluate them with respect to our requirements and the framework proposed by [**?**] for collaborative systems.

Chapter 4 Architecture
This chapter presents the Architecture that we have designed, as well as presenting the design decisions according to three layers of this architecture, namely: Network Layer, Virtual Layer and Application Layer.

Chapter 5 Implementation
This chapter presents the current implementation of our architecture and the technologies used. We then discuss the implementation-choices and the relevant underlying mechanisms that provides the functionalities requested in the requirements. We conclude with the presentation of a proof of concept implementation, a calculator.

Chapter 6 Use Case: Multi-Document Text Summarization using GA
This chapter presents a use-case, namely Multi-Document Text Summarization using Genetic Algorithm, and how to translate it into a web application. We analyze the problem at hand, and walk-through the thought process of re-factoring an existing application into a version of it that is compatible with our architecture.

Chapter 7 Discussion
This chapter will present the main issues encountered throughout this thesis, namely we will focus on:

Chapter 8 Conclusions and Future Work
This final chapter will assess how well did we fulfill the research mandate, in the scope of our requirements and contributions. Future work will also be presented and we will specify some areas on which focus should be lain on from a short-term and from a long-term perspective.

Chapter 2

Background

In this chapter we present an overview of all the material required to understand the context of this thesis. We present first Collaborative Systems, then Cloud Computing, followed by an overview of the current Peer-to-Peer technologies and we finish by glancing at the latest technologies in terms of Virtualization.

2.1 Collaborative Systems

In this section we present what collaborative systems are in a distributed systems context. We outline the difference between the two deployment models of collaborative systems, public and private. We then present the characteristics of public collaborative systems. We conclude by presenting *SETI@Home*, which is an example of collaborative system, and *BOINC*, a middleware to create such systems.

Collaborative Systems can have different meanings in different contexts, and thus we clarify the intended meaning for this thesis. For example, in a Business context, they often refers to Groupware or software which enables work to be accomplished by a group in a cooperative fashion. Where the word *collaborative* is used to describe the interactions between the users of the system [**?**]. Whereas in the context of distributed systems, collaborative systems refers to a system belonging to a sub-class of distributed computing known as *volunteer computing* (or public-resource computing), for which a collection of volunteered resources accomplish a common task [**?**]. Here the word *collaborative* is used to described the compositional structure of the system, composed of disparate components collaborating together to perform a computational task.

In general, *volunteer computing* and *public-resource computing* are interchangeable when referring to this sub-class of distributed computing. We prefer the latter over the former, since it exhibits the public nature of this type of computing explicitly. It is pos-

sible to draw further distinctions for collaborative systems, in the context of distributed systems, such as the deployment model. A collaborative system that is deployed on a grid or a cluster, is deemed to have a *private* deployment model, hence can be referred to as private collaborative systems. Whereas a collaborative system that is deployed using public resources, utilize a *public* deployment model and can be referred to as public collaborative systems. Sharing some similarities in structure with private collaborative systems, their public counterparts have different requirements. Mainly because their infrastructure is composed of publicly volunteered resources. We can formulate the differences as follows:

1. Computers composing the cluster are unreliable: may leave at any moment and computations performed may be incorrect.

2. Quality of the computers are usually on the lower end of the spectrum (commodity hardware).

3. Control over the set of nodes is not centralized, and participation often are incentive-driven.

These differences illustrate challenges that arise from leveraging volunteered resources in a public environment, that are not as present in a private (controlled) environment. This is particularly relevant with respect to the types of problems that are better suited for a public collaborative system platform, as we will illustrate in **??**.

An extensive body of work already exists on the subject, for more information on public collaborative systems in general see [**?**] [**?**], and for more information on private collaborative systems, but more specifically on grid computing see [**?**].

2.1.1 SETI@Home

SETI@Home [**?**] is a notable example of a public collaborative system, built using the BOINC system framework [**?**]. It is used to analyze radio transmissions for extra-terrestrial intelligence across startling different frequency domains. Its architecture consists of a central server which distribute work units to participants in order to perform the computations, then the participants return the result to the server and request another work unit. Due to the unreliability of the publicly volunteered computing resources and the unreliability of the network supporting the communication, it is necessary for the design to factor these in as stated in **??**.

1. Accuracy of the computations is ensured by dispatching the work units to several different participants and by using a consensus mechanism in which a majority of the results returned by the participants must agree.

2. The nature of the problem domain mitigates the unreliability of the network, because it has a high computation-to-data ratio. Work unites are composed of small data-pieces (350kb) providing considerable workload resulting in smaller data-pieces (1kb) for the result.

Consequently, this architecture minimizes the possibility of bottlenecks at the central server while preserving a centralized structure.

This is one of the most successful project of this nature, in terms of aggregate computing power achieving upwards of 660 teraFLOPs [**?**]. The success lies in how well their problem was tailored for distributed computations, or how easily parallelizable was it. Generally speaking, problems for which the dataset can be segregated into independent subsets of the problem are best suited for this paradigm.

2.1.2 BOINC: Berkeley Open Infrastructure for Network Computing

BOINC [**?**] is a middleware system that enables researchers, with limited computer knowledge, to easily create and deploy public-resource computing projects such as SETI@Home. A typical instance of a BOINC project, consists of 4 major components:

- *Master URL* of the project presenting a landing page to register, contribute, and track the progress of the project.

- *Data Server* is responsible for uploading and provisioning the data to and from the participants.

- *Scheduling Server* is responsible for handling the RPCs coming from the participants.

- *Client Application*, enables the participants to connect to the server and request work units, but it also offers a graphical interface to represent progress made.

The ease of use lies in the abstraction of the underlying housekeeping mechanisms required to coordinate the agglomeration of contributing nodes. But also by abstracting the underlying logic required to distribute work units and retrieve the results, and presenting to the researchers a fill-in-the-blanks type of user experience.

The simplicity of setting up and understanding the infrastructure provided by BOINC is the key to its success. Essentially every scientific enterprise resorting to public-resource computing have similar requirements. By creating a middleware that fulfills these requirements out of the box, it enables the researchers to devote more time to significant research problems rather than to computer-related technicalities.

More collaborative projects are created with ease every year using this infrastructure, and this renders the use of public-resource computing more apprehensible from a non-technical standpoint. Ultimately, BOINC removes the technological and technical barriers that possibly impedes scientific data *crunching* when resorting to public computing resources.

2.2 Cloud Computing

In this section, we present the cloud computing paradigm, by identifying the deployment models and the different service models it offers. Then, we present volunteer cloud computing, which is the intersection between public-resource computing and cloud computing.

Cloud computing is the natural evolution of Service Oriented Architecture (SOA), the Web 2.0 and the virtualization technologies. Which results in a paradigm that utilizes the *network as a platform* to provide a variety of services directly to the users. This effectively reduces the gap between content producers and content consumers, by augmenting the availability of the services and reducing the cost of contracting such services. The reduction of cost is impart due to the fact that the services are provided using a consumption-based business model. But also since users can leverage these resources without having to assume the initial cost of purchase or maintenance cost, but simply by paying based on their consumption, which is characterized as utility computing.

A more formal definition of the characteristics of cloud computing is provided by the National Institute of Standards and Technology, U.S.A. (NIST) definition [**?**]:

1. **On-Demand Self-Service :** provides the ability to a user to configure his services automatically through the infrastructure portal by himself.

2. **Broad Network Access :** which enables clients to connect to the Cloud through the Internet.

3. **Resource Pooling :** provides the ability to clients to abstract the underlying specificities of the resources, and simply access a infinite pool of homogeneous resources (virtual or physical).

4. **Rapid Elasticity :** provides the ability to augment the amount of resources to respond to the fluctuations in the workload.

5. **Measured Service :** services can be measured or monitored in a manner x that is transparent to both the provider and the consumer.

2.2.1 Deployment Models

Cloud computing infrastructure can manifest itself as one of 4 different deployment models [**?**].

A deployment model identifies the intended consumer and intended producer of the services that the CSP has to offer.

The most popular deployment model, is a *public* cloud, which is intended to be used by the general public and it is offered by privately owned companies, or CSP (like Google, or Amazon). Another popular deployment model, is a *private* cloud, which is intended to be exclusively used by one entity or organization. They can maintain or not the cloud infrastructure, in which case they delegate to a 3rd-party CSP. The two previous deployment models represent the extrema on the continuum of possible deployment models. In the middle of these two lies a variant which provides exclusive access to organizations sharing common interests, this deployment model refers to *community* clouds. The fourth deployment model is *hybrid* clouds, in which the cloud infrastructure is composed of several sub-clouds, be it private or public. Most common use-case for this deployment model is an organization that requires the security that private clouds offers, but also requires the scalability and availability of public clouds. Thus, it can use their private cloud to store sensitive information and use public clouds to leverage a business intelligence application to manipulate this information, through a Software-as-a-Service (SaaS)

2.2.2 Service Models

An ontological representation of cloud computing has been proposed [**?**], which represents a categorization of the services offered by the CSP.

In Figure **??** several service models are presented for completeness, within the scope of our research only 3 of those service models are relevant, namely: Infrastructure-as-a-Service (IaaS) , Platform-as-a-Service (PaaS) and SaaS. The differences between each service model is better understood when it is illustrated by the separation of responsibilities between the CSP and the consumer.

Figure **??** presents the separation of responsibilities for each of the service models.

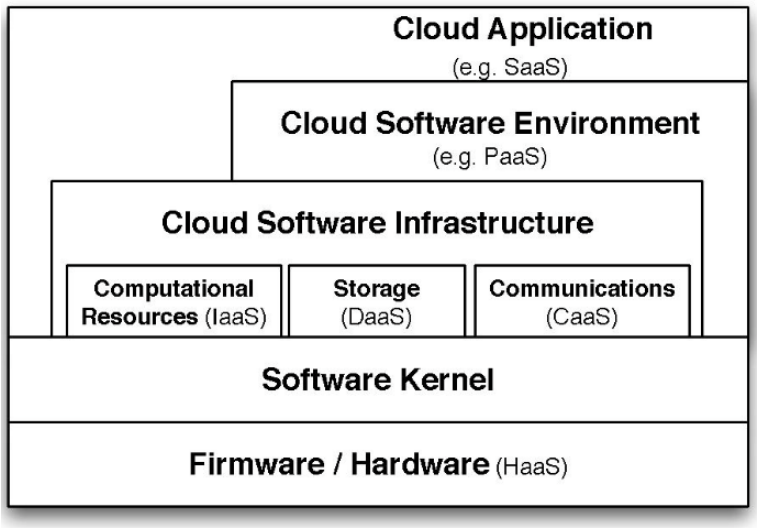


Figure 2.1: Ontological Representation of Cloud[**?**]

IaaS imposes the majority of the responsibilities above virtualization to the consumer, whereas the CSP is only responsible of providing the physical and virtualization layers. It grants the most flexibility to the consumer compared to the other service models, and usually a consumer uses a virtual machine image to encapsulate the *complete* executing environment, from the OS to the applications and anything in between, that he/she wishes to host using the resources of the CSP. An example of such a service model would be *Amazon EC2* [**?**].

Not as flexible, but easier to configure and use, the PaaS provides a set of abstractions (in the form of an API where only the Application layer and the Data layer is presented) for the consumer to use when writing an application. The *Google App Engine* is a popular example of PaaS [**?**].

Finally, SaaS is a service model that consists only of the application, and where everything else is the responsibility of the service provider. Using this model the consumer uses a software through the CSP without owning a copy by leveraging its functionalities as services. An example of SaaS would be *Salesforce*, which offers a multitude of various business applications as SaaS [**?**].

2.2.3 Volunteer Cloud Computing

Volunteer cloud computing is an emerging paradigm. Conceptually, it is the intersection of public collaborative systems and the cloud computing paradigm, it can also be defined as the 5th deployment model of cloud computing.

This new deployment model consist of constructing a cloud computing infrastructure

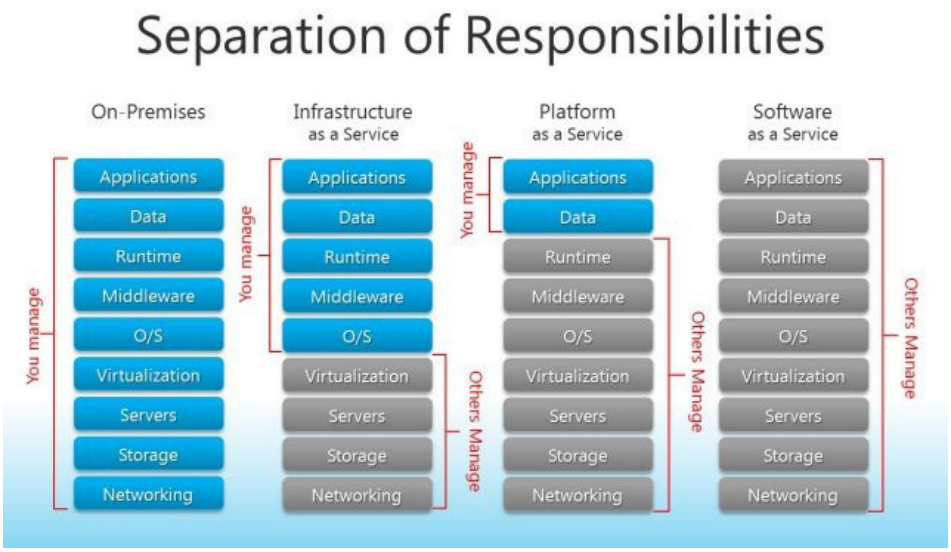


Figure 2.2: Cloud Services w.r.t. Responsibilities

by relying solely on (volunteered) public-resources. It is a fairly recent idea, as it is not even mention in earlier publications such as [?], but it is in later publications such as [?].

This paradigm suffers from different challenges compared to traditional cloud computing [?], which are comparable to the challenges to public collaborative systems. But volunteer cloud computing is concerned with providing a cloud computing infrastructure built using volunteered computing resources, whereas public collaborative systems are concerned in performing a distributed computationally intensive task. The dichotomy of the scopes is what really differentiate volunteer cloud computing from public collaborative systems.

Many different efforts exists in an attempt to provide this type of infrastructure, with varying degrees of success adopting different approaches while addressing the different challenges. Note that some of the efforts, such as [?], targets the full spectrum of service model: IaaS, PaaS, and SaaS. Whereas other focuses on only one IaaS [?] [?]. Finally some offers a completely different approach, such as building a transparent infrastructure using peer-to-peer interception techniques [?].

2.3 Peer-to-Peer Computing

In this section we present an overview of peer-to-peer computing, we introduce the concept of overlay networks and then detail the differences between two types of topologies. Finally we present a comparative analysis of the topologies in the form of a summary table.

2.3.1 Overview

Peer-to-peer computing has many different characteristics that makes it an interesting prospect in a public environment such as the Internet when constructing a distributed system.

Such a system built in a peer-to-peer computing fashion, is referred to as a *peer-to-peer system*. Essentially, peer-to-peer systems are defined as a distributed system for which every node in the system is at the same time a consumer of the services offered by the system and a producer, or a server and a client (servent). The cost of operations is then amortized and distributed among the participants. Because every participant in the system usually assumes the cost of operating their own computers, where the aggregate cost of operating the system is the combination of all the costs incurred for operating the participants computers. Peer-to-peer systems are generally decentralized. This removes any single point of failure resulting in systems that are very resilient to the failures of the nodes composing its infrastructure. Peer-to-peer systems are also scalable, inasmuch as more nodes are available to join the system, theoretically *ad infinitum*. This underlines that there are no explicit scalability limitations for this type of system, given that the business logic it implements also does not explicitly prevents it (by design). The equality amongst nodes is usually achieved by creating a decentralized network, where each node assume equal responsibilities in terms of routing and discovery of other nodes within the network. Generally, there are no central servers and all the nodes in the system are equal, but practically some systems relies on a bootstrapping server to ease joining new nodes to the system.

Usually, in order to join a system, a node is required to know at least one other node in the system. Initially if there is no other nodes in the system, the first node to join will become the only node to contact to join this system. Subsequently, any nodes that wishes to join this system contact this node, but only for the second node, whereas the third node has the choice to contact the first or the second node, and so on. Once a node has joined the system, it usually has only a partial view of the entire underlying network and in order to contact any node not contained within this view, it must interact with neighboring nodes for indications on how to proceed. This co-operative location mechanism differs in implementation, but conceptually a node requires the assistance of the other nodes, in some ways, to navigate the underlying network in its entirety. Consequently, peer-to-peer systems usually construct virtual *overlay network* on top of the physical underlying network, to mitigate the complexities inherent to this decentralized architecture [**?**] [**?**].

2.3.2 Overlay Networks

Overlay networks are fundamental to peer-to-peer systems and can be defined as virtual networks superposed on the physical network (the Internet). The virtual network represents virtual connections between the different nodes in peer-to-peer systems. It abstracts the underlying physical connections, and exposes the logical (virtual) connections between the nodes of the system.

For example: If **Node A** is connected to **Node B**, which is connected to **Node C**, finally **Node C** is connected to **Node D** in the underlying physical network topology:

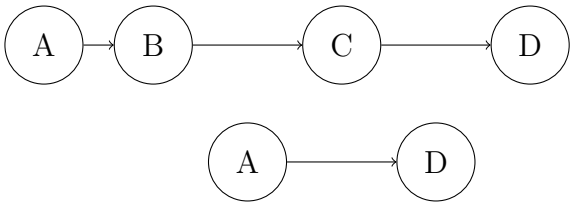


Figure 2.3: Example of Overlay Network Virtual Connections.

It is possible, then, to express the indirect connection between **Node A** and **Node D** as a direct connection in the virtual overlay network. Not only, overlay networks enables us to abstract away the details of the underlying network, but it allows nodes to communicate between nodes using this virtual topology.

Overlay Network topologies are categorized relative to their structure [**?**]. In the next sub-sections we present the distinction between a *structured overlay network* and a *unstructured overlay network* respectively.

2.3.3 Structured Overlay Network

What characterizes structures overlay networks is the fact that they are constructed by organizing the peers into a structure graph.

An abstraction known as *keyspace* is used to organize the participating peers into a structure. Each of the peers is assigned a portion of the keyspace for which they are responsible. Their responsibilities consists of locating keys in this portion of the keyspace, which they are responsible for indexing.

Partitioning of the keyspace is done according to the *keyspace partitioning scheme*, which dictates the structure of the resulting network. Some keyspace partitioning schemes can produce a ring topology, see [**?**]; whereas others can produce a tree-based topology, see [**?**]

As a consequence of its static structure, this type of overlay network is not suited to run complex multi-attribute queries. This is attributed to the organization of the keys according to a single metric, and thus result in supporting single-attribute dominated queries. These queries are deterministic, since the attribute for which the queries are tested against usually corresponds to the key and returns upon exact or partial match.

Ultimately these networks are built around the idea of being able to efficiently locate any key within the network. .

A class of structured overlay networks are Distributed Hash Table (DHT), which consist of creating a overlay network by dividing the keys in an associative array (hash table) among peers, then each peer is responsible for a portion of the associative array. A key in this context represents either a node or an entry in the DHT. To retrieve a specific key, a node query the other nodes to find which one is responsible for this key, or find the node which is responsible for the portion of the associative array closest to the key. If neither has the key, or the node that is responsible for the key, the query returns that there are no value associated with this key, hence the deterministic nature of DHT.

Many implementation of DHT exists [**?**] [**?**] [**?**].

Structured overlay networks are constructed to fulfill a specific functionality, such as storing data in a distributed fashion, and are very good at it. But they are not as versatile as their counterparts in terms of applications, resistance to failures, and querying abilities [**?**] [**?**].

2.3.4 Unstructured

Unstructured Overlay Networks differ with respect to their construction, and results in a unstructured topology, namely a flat or hierarchical random graph. More importantly, there are no relations between the topology and the partitioning of the keyspace, since technically the keyspace is irrelevant in this context.

Rather than utilizing a sophisticated partitioning mechanism for the graph, as with the former type of overlay network, peers connect to a peer in the network, and by performing cyclic exchanges of information among them (generally in pair-wise fashion) they update their view of the network. The peers only have a local view of the overlay network, but by performing cyclic exchanges in order to achieve local convergence between their views of the network it results in global convergence of the network topology. Due to this focus on local convergence, resulting in global convergence (or network-wide convergence), several self-* properties emerges such as: self-configuring, self stabilizing and self-healing properties [**?**] [**?**]. Self-configuration refers to the ability to autonomously

configure the deployment of a system, or to respond to changes in the topology of the composing components [?] [?]. And self-stabilization property refers to the ability for a system to start in any arbitrary configuration and eventually converge to a desired configuration [?] [?]. Whereas, the self-healing property provides autonomous detection, diagnosis and repair of localized problems [?], but also autonomous component-level failure recovery [?]. Maintaining the overall health of the system, can be referred to as the *survivability*, which is the prime objective of self-healing components [?] [?].

Given the emergence of these properties, this type of overlay network is very robust in highly dynamic environments, such as the Internet [?].

Generally, the routing is done by performing random-walks or flooding the network, and consequently querying is not deterministic. Since the structure of the graph, or the lack thereof, is not dependent on any particular arrangement of the peers, the network supports complex queries because of the arbitrary nature of the routing mechanism, which can reinterpret the network accordingly. This is commonly referred to as *slicing* and we present it in the following subsection.

Because of the routing mechanism, location of specific data managed by a peer in the network is rather difficult if it has not been replicated on several nodes. Replication schemes usually target the most popular data in the network, and replicates it across several nodes to ensure availability. Thus, rarely queried data is unlikely to be found by queries if it is managed by a single or very small collection of peers. Because queries are generally have a Time-To-Live (TTL) , which can be represented in terms of hops or Hops-To-Live (HTL) after which the query will simply be drop and no result will be returned.

Examples of unstructured overlay networks are usually based on epidemic protocols, or gossip-based protocols [?], for which a peer joins the network and periodically exchange its local view of the network with another, randomly selected, peer.

For more information on overlay networks, and comparisons on the different types see [?] and [?].

2.3.5 Slicing

Slicing is a primitive in distributed systems, which dictates the querying abilities of the underlying network structure. The ability and degree to which the system can perform slicing, is relative to the complexity of the queries it supports. It can be formulated as the following:

> Given a graph, representing a network of peers, can we partition it according

to a set of node local-attribute(s).

Several techniques exists to solve this problem as depicted in [**?**] and [**?**]. Techniques varies in terms of the type and number of attributes considered, and consists of ordering the nodes according to these attributes providing different perspectives of the same collection of nodes.

Based on this definition of the slicing problem, we can now formulate the differentiation between Single-Attribute Dominated Query (SADQ) and Multi-Attribute Dominated Query (MADQ). The distinction arises from the ability to query resources, according to the specifications of their attributes, whether it support querying 1 attribute per query or multiple. Or we can also distinguish the two, based on the ability to slice the current network into slices (partitions) according to the(se) attribute(s).

This problem helps deciding which topology of overlay network to use, because it illustrate the querying ability provided by the different topologies.

2.3.6 Comparison

We can summarize the advantages and disadvantages of the different types of overlay networks according to different characteristics, inspired by [?] and [?]:

Characteristics	**Structured**	**Unstructured**
Construction	Generate abstract keyspace, which is divided among the peers. Topology results in a structured graph.	Peers query a participating node and retrieve information about the network. Repeats periodically, results in a flat or hierarchical random graph.
Routing	Key-based routing, each node contacts the closest node to the key (according to its local view), and then the closest node to this node, and so on until the node responsible for the key is reached.	Perform a random-walk through the network or using flooding techniques. May not return a result (query could time-out).
Lookup	Deterministic and has a general time complexity of $O(logn)$, where n is the number of keys in the keyspace.	Non-deterministic and is generally a best-effort attempt to locate data, popular data can be easily located due to replication. No defined boundary in terms of time complexity, uses a parameter that represent the TTL, (in seconds or hops).

Continued on next page

Continued from previous page

Characteristics	**Structured**	**Unstructured**
Join/Leave	**Join:** Node contacts a live node and a portion of the keyspace is attributed to it, according to its identifier (or key). **Leave:** Node leaves, eventually the keys will be remapped to the neighboring nodes in the network and all the neighbors table will be updated.	**Join:** Node contact an arbitrary live node and exchange information, and repeats periodically forever achieving a dynamic view of the network. **Leave:** Node leaves, and then since it won't participate in the periodic information exchange, it will be discarded from the dynamic view of the network.
Reliability/Fault-Tolerance	Resist churn and normal level of node failures.	Highly resistant to churn and very high level of node failures.
Slicing	(SADQ) Single-Attribute Dominated Queries	(MADQ) Multi-Attributes Dominated Queries

Concluded

2.4 Virtualization Technologies

In this section we present a general overview of *virtualization*. We differentiate between *full virtualization* and *light virtualization*. Then, we present what are *containers*, and take a look a two different instances of it such as Linux Containers (LxC) and Docker containers.

2.4.1 Overview

Virtualization provides the ability to allocate the physical resources needed to accomplish a task prescribed by the software. It is *generally* achieved through the decoupling of software from hardware [**?**].

The simplest example of virtualization that is used by a majority of OS are processes [**?**]. Processes are isolated into virtual environments that exposes the resources as if they are the sole consumer, by abstracting the other processes away and effectively decoupling the software from the underlying hardware providing the computational resources.

The concept can be extended to the virtualization of whole OSs. The interactions from the OS with the physical hardware are done through a software abstraction layer,

the resulting machine is called a Virtual Machine (VM) [**?**].

Within the context of distributed systems, one type of virtualization pre-dominantly exist, the *full virtualization* or desktop virtualization. Semantically it is different from *para-virtualization* and both types shouldn't be conflated. The latter requires modification of the guest operating system to comply with the interface defined to access the physical hardware. Whereas the former requires no such modifications, and the calls from the operating system to the hardware can be interpreted as-is [**?**].

A newer emergent type of virtualizations, known as *light virtualization* or operating-system level virtualization, is gaining popularity. It is more akin to the type virtualization present with processes.

It is important to draw the differences between these two types of virtualizations, in order to understand which favors the fulfillment of the requirements we have identified best.

We neglect para-virtualization in the context of this thesis since it is not as relevant for our research, as we do not intend to impose any modification on the OS. For more details on the basics of virtualization see [**?**] [**?**].

2.4.2 Full Virtualization vs. Light Virtualization

Several distinctions exists between full virtualization and light virtualization, out of which some are advantageous or disadvantageous in certain contexts.

Full virtualization can be defined as a virtualization technique for which the entire hardware is virtualized, providing an abstract computing base on which it is possible to execute a complete OS without any modifications [**?**].

The use of VM, rather than physical machines, offers several key features that are desirable in the context of distributed systems. One of these features is the ability to clone, which enables the replication of the complete execution environment with ease. Another very desirable feature is the ability to test changes before applying them, and the ability to migrate the VM across different hardware, which can also be done without much interruption of service. The penultimate feature of using VM, as opposed to physical machines, is undoubtedly the ability to execute multiple VMs in parallel on a single physical machine. It grants the ability to have multiple different execution environments without having to dedicate a physical machine for each. When applied to servers, this feature is referred to as *server consolidation* [**?**].

A VM is controlled by an *hypervisor*, which is responsible for orchestrating the VMs access to the underlying physical resources of the host, and it is also known as Virtual

Machine Monitor (VMM). It is possible to distinguish between two types of hypervisors. The distinction depends on whether or not there is an OS running the hypervisor (Type-2) or if it is directly interfacing with the hardware (Type-1) [**?**].

VirtualBox [**?**], is a prime example of Type-2 hypervisor. Figure **??** shows the difference between full virtualization (on the left) and light virtualization (on the right). It illustrates how each type of virtualization interacts with the hardware through the OS.

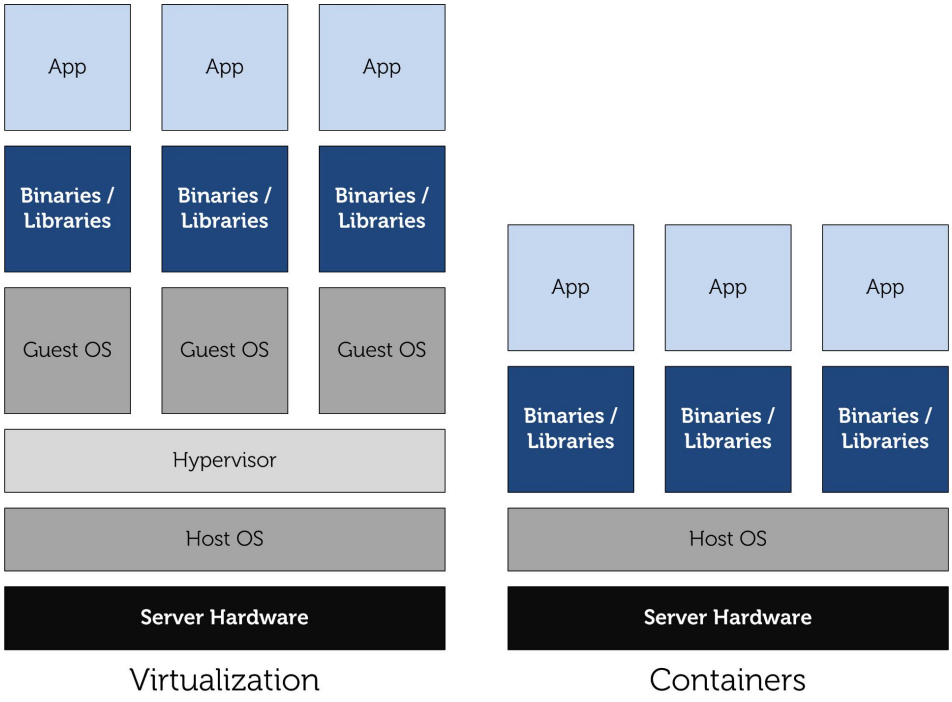


Figure 2.4: VM vs. Containers[**?**]

Light virtualization or operating system-level virtualization, does not attempt to execute a complete OS on top of the current OS, but rather it shares the kernel and the libraries to create fully self-contained environments through isolation mechanisms. As a result, light virtulization technologies are generally referred to as *containers*.

These containers, benefits from similar advantages as their fully virtualized counterparts, such as the ability to be cloned, and the ability to run multiple instances on a single hardware base.

The feature that sets them apart, results from the non-redundant virtualization scheme compared to full virtualization redundant scheme (OS on top of OS), is the ability to host much more instances on a single machine. This scheme grants a full order of magnitude of additional deployment to lightweight virtualization, on identical hardware. From 10-100 VMs to 100-1000 containers [**?**].

Contrary to full-virtualization, containers do not perform any emulation, but rather

through namespace isolation re-use the libraries and kernel of the host operating system. This makes it an interesting candidate when using low-end hardware, to maximize the hosting capability.

2.4.3 Containers

In this subsection we present the characteristics that make containers possible in Linux.

Introduced as a kernel patch [**?**], *c-groups* consists of an aggregation and partition mechanism for groups of tasks, that contains (within those aggregations/partitions) all their children. The central technology behind containers in Linux is *c-groups*, or *control-groups*. A c-group allows different specialized behaviors to be explicitly defined, by generating a hierarchy of groups. The strength of this concept lies in the ability for it to be used with other Linux subsystems and to provide additional properties for groups of processes. One example that illustrates the definition of specialized behaviors, consist of pairing c-groups with the *cpuset* subsystem to restrict a group of processes to a specific Central Processing Unit (CPU)-core.

Through the c-groups technology, and by extending the definition of the behaviors of processes in Linux, emerged the container technology. We present in the following subsections two different implementations of Linux containers, LxC and Docker.

2.4.4 LxC

LxC or Linux Containers, is an open-source implementation of the containerization technology for Linux. It focuses on providing the tools to facilitate the development of system containers in a distribution agnostic fashion [**?**].

LxC leverages c-groups partitioning and aggregating capabilities to provide resource and namespace isolation without any extra virtualization mechanisms, but rather through the Linux kernel native subsystems. Namespace isolation provides the ability to isolate running applications completely from the operating system execution environment, albeit not an exclusive feature of LxC. A general example of namespace isolation is the *PID namespace*, which provides the means to create sets of tasks such that each set is completely independent from one another [**?**]. In other words two tasks belonging to different set of tasks, can have identical ID without incurring any ambiguity. Resource isolation, provides the means (via cgroups) to allocate system resources to different groups of tasks.

Thus, LxC provides capabilities analogous to an operating systems (filesystem, network, processes) and dedicated access to the physical resources of its underlying host, in a isolated environment in the form of a container [**?**].

2.4.5 Docker

Docker was released on March 2013 and is ever-growing in popularity. It already has its own convention: DockerCon [**?**], which is endorsed by major technological companies such as IBM, Microsoft, Google, and RedHat (to name a few).

Docker is open-source and provide means to automate deployment of applications within containers. It provides a high-level interface to containers, by abstracting the intricacies and specificities of the containers into an intuitive and easy to use high-level API, which provide management, configuration and monitoring capabilities.

Docker is not a replacement for LxC, rather it is addition on top of the containerization primitives (LxC) of Linux that provides high-level features to interact with containers. It actually implements a custom version of these primitives, largely based on the LxC primitives. As does LxC, Docker leverages namespace isolation and c-groups to provide isolation to the container, but also uses *union file-systems* to provide a lightweight templating system, known as images. *Docker Images* are used to specify the operating system environment to be instantiated within the container [**?**].

The union file system is a Unix filesystem service, with which it is possible to create virtual filesystems from separate filesystems through branching and layering. One of the advantages of this service is the ability to update a filesystem by simply applying the difference from the previous versions. In Docker's context, this means the ability to update a container by simply uploading the difference (or new layer) and applying it, rather than uploading the entire container again [**?**].

Docker uses a client-server model to communicate between the container (client) and the originator/deployer (server), which can reside on a single host or not. This ease the workload for the client, because it is possible to offload computationally intensive tasks, such as container construction, to more powerful machines while conserving lightweight clients [**?**].

Chapter 3

Related Work

In the previous chapter we have presented the background information required to understand the following chapters and this information is useful to understand the context in which we approach the following related works.

In this chapter we present the work that relates to our research, notably the two most relevant projects: Cloud@Home and Peer-to-Peer Cloud System. But before we introduce a framework designed to evaluate collaborative systems [**?**], which we use to evaluate the two projects. Finally, we discuss the how these solutions fit our requirements and what is the scope of these projects.

3.1 Evaluation Framework

The framework proposed by [**?**] aims at formalizing the requirements for peer-to-peer collaborative systems.

Each phases covers a functional requirement of this problem, and they should be satisfied in a collaborative system to provide a minimally working and scalable. The 7 key phases that mitigates this problem are illustrated in the following figure:

1. **Advertise**: Each node advertises its resources and their capabilities using formal specifications over a set of attributes.

2. **Discover**: A mechanism to discover and keep track of the useful specification advertisements from the other participating nodes. This enables to accelerate the querying mechanisms but also to preserve inter-resource relationship information.

3. **Select**: A mechanism for a user to select a group(s) of resources, that satisfies a formal specification of his requirements defined in a query containing the attributes

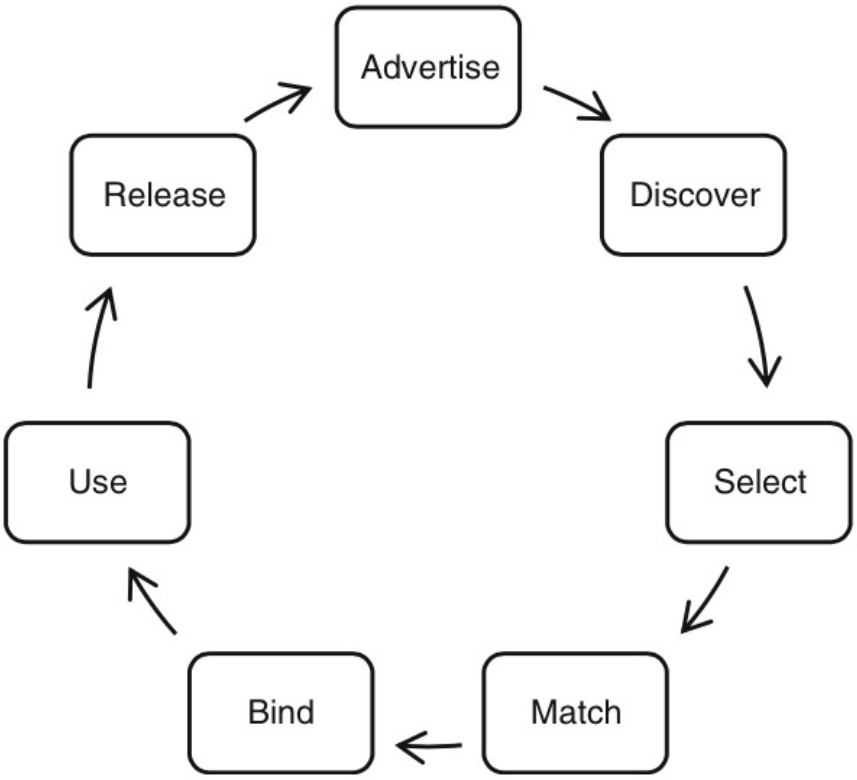


Figure 3.1: Framework for Peer-to-Peer Resource Collaboration Problem

and ranges of acceptable values for these attributes.

4. **Match**: A mechanism to be able to formally specify inter-resource relationship requirements and to ensure that the nodes selected satisfies the resource and application constraints, in the query.

5. **Bind**: Provides a binding mechanism between the resources and the applications, preventing two applications from selecting the same resources. But also to cope with the dynamic nature of p2p, since a node may not be available at the time of use but was at the time of selection.

6. **Use**: Utilize the best subset of available resources, from the resources acquired, to execute the application (and all its tasks) while respecting the utilization agreement between the application and the resources.

7. **Release**: Provides a mechanism allowing to release (some) resources in relation to the application demand, and/or the contractual binding (if time-sensitive). This mechanism can also provide the means to partially release a resource and to enable

that resource to collaborate with other applications simultaneously.

Thus, if one were to implement each one of these functional requirements into a separate module, then the resulting system would be modular and would ensure proper collaborative behavior. For more details on this framework and for an evaluation conducted on different peer-to-peer collaborative systems using this framework, see [**?**] [**?**].

For our purposes this framework acts as a checklist to identify how each of the functional requirements inherent to collaborative systems are implemented in the following two projects. Ultimately, we will use it in further chapters to assess the fulfillment of these functional requirements in *our* own architecture.

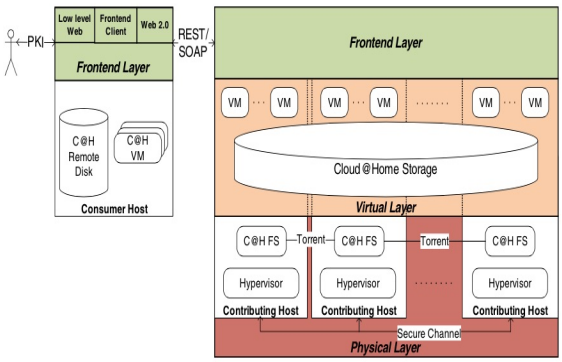
3.2 Cloud@Home

Cloud@Home is a new paradigm which spawned at the intersection of two existing paradigms: the *public-resource computing paradigm* and the *cloud computing paradigm*. It is one of the earliest attempt to create a *volunteer cloud computing infrastructure*. Over several publications, the authors have analyzed extensively the challenges relating to volunteer cloud computing, [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] [**?**] [**?**].

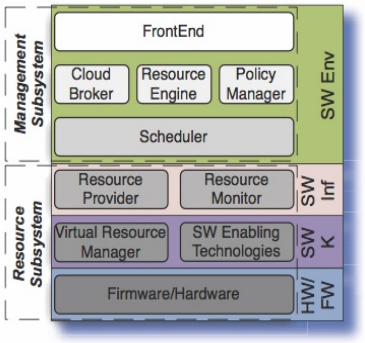
Their vision is to offer a full-fledged cloud computing infrastructure constructed from volunteered. Full-fledged, in this context, means that they want to provide the following service models: IaaS, PaaS and SaaS. The Cloud@Home system aims at offering performances and Quality of Service (QoS) similar to major CSP.

To fulfill some the performance requirements, the authors focused on *interoperability* between CSP, providing a user with the ability to contract resources from commercial CSPs, given that the volunteered resources available are insufficient or inadequate. Whereas to fulfill the QoS requirements, they require that both parties negotiate the expected or desired QoS contained in a Service-Level Agreement (SLA), created by the resource consumer. In other words, the consumers provide their intended terms of use, relative to the resources, and the providers can decide whether or not they accept the terms of use. In case of disagreement with the SLA and the QoS it contains, the negotiation process is aborted. For more details pertaining to the QoS and SLA in the context of this system, see [**?**].

Contrary to the CSP, because Cloud@Home uses volunteered resources, the authors were obligated to ponder on how to entice people to contribute their resources to this infrastructure. They have devised an incentive-based business model, that offers financial restitution to the users for their contributions. This model proposes to recycle current



(a) Overview [**?**]



(b) Functional Representation [**?**]

Figure 3.2: Cloud@Home Architecture

idling resources into a utility that can be sold, with respect to the quality and capabilities of these resources.

In the following subsections we will present and discuss the architecture proposed for this solution, and discuss its conformance to our requirements and the evaluation framework.

3.2.1 Architecture

The Coud@Home system architecture, shown in Figure **??**, is composed of three layers: the *Frontend Layer*, the *Virtual Layer* and the *Physical Layer*. The functional representation of the infrastructure, shown in Figure **??** is divided into the *Management Subsystem* and the *Resource Subsystem*.

Starting by the architectural overview, the first layer or *Frontend Layer*, is responsible for providing a high-level service-oriented point of view of the underlying infrastructure. *Consumer Host, in this context, represents a consumer of the cloud service provider and not a consumer of the application deployed in the cloud.* The relation between

the Consumer Host and this layer is based on a client-server relationship, providing a centralized point of access to the infrastructure.

The *Virtual Layer* provides a homogeneous perspective of a set of heterogeneous resources, by means of virtualization. Because it uses virtualization, it is possible to completely abstract the underlying hardware specificities of the different contributors resources and to exploit them in a platform-agnostic way. This layer is further divided into two services: the *Execution Service* and the *Storage Service*. The former enables the consumers to host and execute VMs, according to their needs, providing a similar service as the Infrastructure-as-a-Service model provides. Contributors are then assigned an arbitrary virtual machine to host, or multiple VMs to host, relative to their intended contributions. Whereas the latter provides a distributed storage facility by implementing a distributed file system such as [**?**]. Consumers can then mount locally a remote disk that corresponds to a portion of the distributed file system. As for contributors, they host an arbitrary portion of this distributed file system, relative to the amount of disk space they wish to contribute, providing a unified view of the entire distributed file system to the consumers.

The *Physical Layer* is responsible for connecting the contributing resources together and providing means for communication. Negotiations for the resources between the contributors and the consumers are performed at this layer. The negotiation process consists of a request for resources emitted by the consumer, followed by an attempt to match the parameters of this request with the parameters of the volunteered resources available. The authors specified several mechanisms to ensure and negotiate the Quality of Service and Service-Level Agreements between consumers and contributors (or CSP) [**?**][**?**].

The *Management Subsystem* translates the user's requests into multiple sub-requests. Whether it is possible for the requests to be satisfied, using volunteer resources or their commercial resources, depends on the QoS specified by the user and if it is compatible with the resources available. These request(s) are passed on to the Resource Subsystem which binds and matches the resources to the user from which the(se) request(s) originated. The Management Subsystem is centralized in order to manage the infrastructure, and to manage the QoS and the SLA of all the resources, but also to provide dynamic provisioning. The authors claim that it is the only way to aggregate the global information of the infrastructure and to provide a reliable perspective of it [**?**].

Finally, the authors proposed a middleware that implements the ideas from the architectural overview and the functional representation, covering the different goals posed by the Cloud@Home paradigm. Figure **??** showcases the deployment of the Cloud@Home

infrastructure.

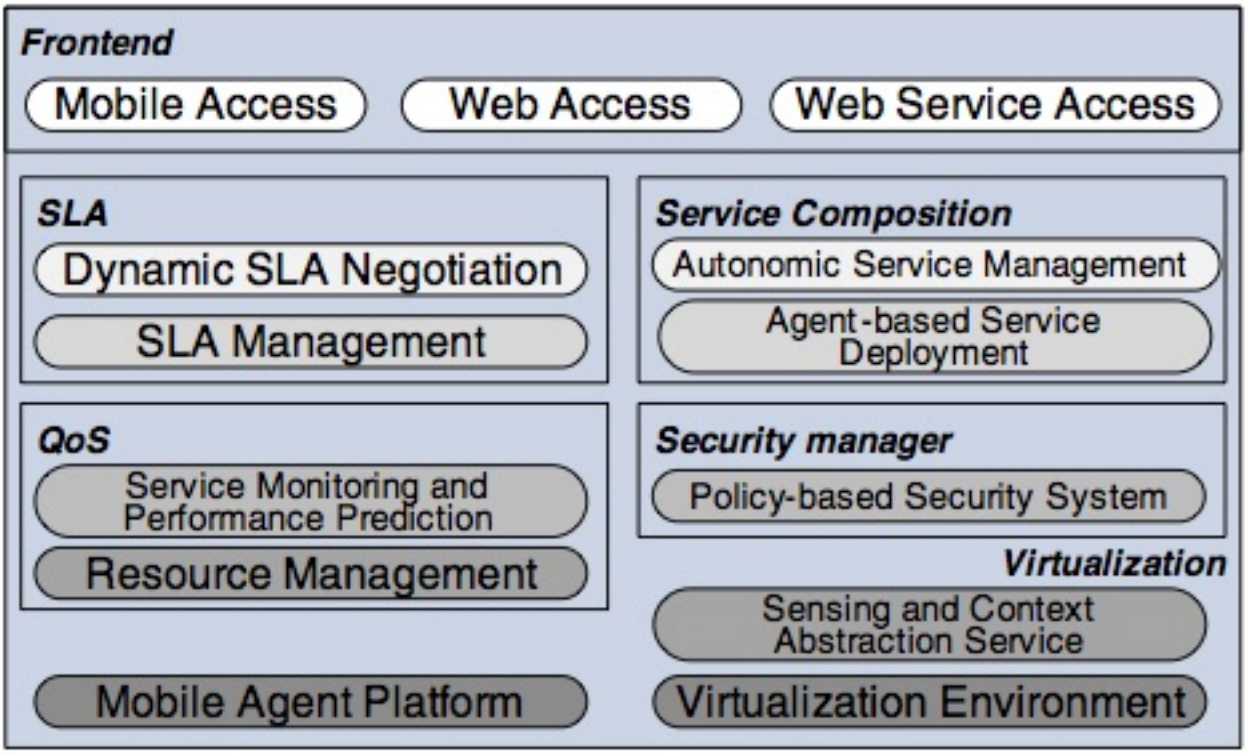


Figure 3.3: Cloud@Home Infrastructure

In this overview, we have presented a simplification of the functionalities of the two subsystems for more details consult [?]. For more details on the different layers composing its architecture consult [?], and finally, for more information on the middleware consult [?].

3.2.2 Evaluation

In this subsection we evaluate the system according to the evaluation framework that we have introduce in Section ??. Then, we evaluate the system against the functional requirements that we have identified in ??.

Evaluation Framework

We use the evaluation framework from Section ?? to identify any functional deficiencies present in this system relating to the essential requirements of peer-to-peer collaborative systems. These functional requirements need not to be implemented as being mutually exclusive, and a subset of these requirements can be combined into a single module.

As a matter of fact, such is the case for the first three phases. **Advertise**, **Discover**, and **Select** are accomplished by the Management Subsystem. They are grouped together to provide the ability to a user to enroll and advertise its resources, or to a consumer to find resources that are available according to their needs.

It is also the case for the remaining phases, **Match**, **Bind**, **Use**, and **Release**, all being accomplished by the Resource Subsystem. It provides the ability, given a request, to match the resources which satisfies the request, then to allocate the resources to the consumer in order to use them. Upon using the resources, it is possible to release or to deallocate the resources when the consumer no longer requires them. As a consequence, it provides *dynamic membership* capabilities to the resources, permitting to add or remove resources according to the SLA or the current workload.

Essentially all the functional requirements present in the evaluation framework are fulfilled in one way or another, directly or indirectly. As a consequence, this project is appropriately devised to address the essential functional requirements of a peer-to-peer collaborative system.

Functional Requirements

We now evaluate how this system addresses our functional requirements, as presented in Section **??**, enlightening us on how it could address the research questions presented in this thesis.

Starting with the first requirement, which entails the use of commodity hardware to form the infrastructure on which it is possible to deploy multiple applications. By leveraging public-resources as part of the resources available to deploy an application, and by supporting the deployment of multiple applications simultaneously, we can assess that this requirement is completely satisfied.

As for the second requirement, which states that no third-party should be introduced to provide any services, in order to maintain self-containment of the system. We can only assess partial satisfaction of this requirement, because it is possible using this system to rely exclusively on contributed resources, and not contract resources from any commercial CSP. But in order to use this infrastructure, both consumers and contributors are required to interface with the Management Subsystem, and to an extent this is a third-party in relation to the consumer and contributor.

The third requirement, states that the system should be resistant to resources failing and should not introduce any single point of failures, for security purposes. The Cloud@Home system focuses on creating a middleware that is lightweight, and this middleware is built around the concept of service migrations. Upon failure of any resources,

the system can swiftly migrate the workload to any other available resources, resulting in an infrastructure optimized for lightweight services and failure-resistant. Unfortunately, this system is built around the concept that a central server is required to manage its infrastructure, as stated previously Section **??**. Although, some fault-tolerant mechanisms are put in place to mitigate this single point of failure, including redundant servers, and the authors claim it is sufficient. Further tests and benchmarks are required to corroborate their claims, in a realistic environment, and to satisfy the this requirement.

The fourth requirement, states that no special equipment should be required to access and participate in this system, but rather encourage resource recycling. Consequently, it implies, as a sub-requirement, that the memory footprint should be small enough to enable legacy equipment are able to participate in this system. Because this system relies on full virtulization technology, it forces the contributing peers to host VMs in order to contribute a computational resource. Limited resources are expected to be consumers rather than contributors, except in very specific cases where the application lends itself to it. Such cases are when the limited resources are used for sensorial inputs, such geo-location, accelerometers, barometers, and other various sensors that could be used as a stream of information available to any applications in the cloud. These stream of informations are analogous to nodes in a sensor network. It is clear to which extent the memory footprint is small in the context of the proposed IaaS model, but not so much in the context of the other service models. Cloud@Home offers PaaS capabilities by using their *SLA Engine*, ensuring the availability and capacity of the resources, and *CHASE (Cloud@Home Autonomic Service Engine*, which determines the optimal configuration parameters for the service application using the current configuration of the cloud. Given that this processing occurs at the centralized server, it shouldn't incur a large memory footprint on the contributing resources. But the contributing resources are still required to host VMs. This requirement is not completely satisfied.

The fifth and final requirement, states that the system should provide dynamic membership capabilities to all applications, which means that for every application it should provide means to scale according to the fluctuations in the workload. The Management Subsystem enforces the resources SLA and guarantees the QoS, and consequently it will preempt the necessary resources to ensure that they are only used to the extent prescribed in the SLA. This system supports dynamic membership, and consequently provides scalability to the applications deployed, resulting in the satisfaction of this requirement.

Out of the five requirements: one remains unsatisfied, three are partially satisfied and one is completely satisfied. We can conclude that this system does not address our research question sufficiently for it to be considered a viable solution, and thus we must

find a more adequate solution or create one.

3.3 Peer-to-Peer Cloud System

The Peer-to-Peer Cloud System (P2PCS), was developed as part of a doctoral thesis [**?**], and proposes a slightly different approach to volunteer cloud computing than the previous system. One of the defining characteristic of this system is the fully decentralized structure connecting the different peers, constructed using epidemic and gossip-based protocols.

The authors propose an infrastructure with no centralized server(s) to manage the resources, conversely to the Cloud@Home system. Rather each node interact directly with the other nodes when performing any system operations. Consequently, the consumer nodes interacts directly with the (potential) contributing nodes in order to select the fittest candidates to host the application. Fitness of a candidate is evaluated using the response of a specially defined query, which contains the desired criteria for candidate resources (e.g., 10 nodes with at least 8 gigabytes of RAM...). The collection of nodes that fulfills these criteria are selected and they form a *Slice*. This slicing mechanism is an *attempt* at solving the slicing problem presented in Section **??**. The consumer would communicate with this slice, or pool of nodes, via an API that is similar to what current IaaS provider (Amazon EC2 or S3) uses. Using this API, the consumer is able to instruct the contributing nodes on when to start or stop VMs according to the application requirements.

The authors present this system as a peer-to-peer cloud computing architecture, focusing to provide only one service model, the IaaS.

In the following subsections we present the architecture of the system, and proceed to evaluate the system against the evaluation framework and our functional requirements, as we did for the previous system.

3.3.1 Architecture

The architecture is divided into 3 functional components, although the authors did not explicitly use nomenclature to distinguish between correlated components, we do.

Figure **??** presents the different components of this system. The authors distinguished between what has been implemented (in gray) and what has been left as future work (in white).

Using a top-down approach, we present the first component, representing the point

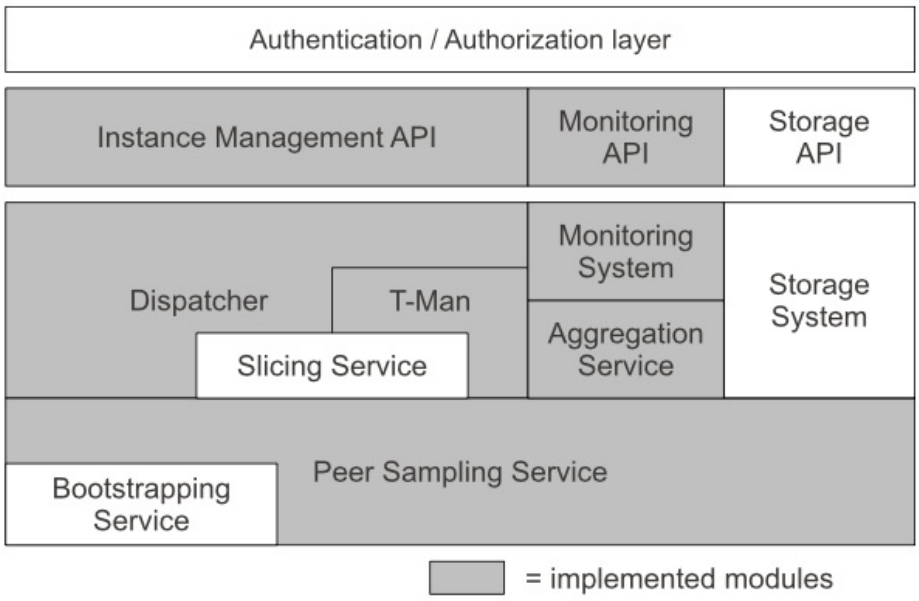


Figure 3.4: Peer-to-Peer Cloud System Architecture

of access to the system. The *Authentication/Authorization layer* is where a user (contributor/consumer) is required to identify themselves in order to access the system.

The second component represents the principal access point to the cloud for the consumers, it is named the *API layer*, It is composed of three sub-components providing different capabilities through various interfaces. The *Instance Management API* sub-component provides an interface that circumscribes the possible interactions between the consumer and the contributing nodes, with respect to the VM instances. Whereas, the *Monitoring API* sub-component provides a visual representation of the slice topology, and the *Storage API* sub-component provides the means to configure the distributed storage system for the current slice.

The third component represents the *Networking layer*. This component is composed of three sub-components, which in turns are composed of three or less components.

The first component of the Networking layer is composed of two services, which are the *Bootstrapping Service* and the *Peer Sampling Service*. As it name implies, the former provides the bootstrapping mechanism for a node to join this system. Whereas the latter, provides a list of peers in the network to exchange messages with. The creation and maintenance of the list is accomplished by creating and maintaining an overlay network over all the peers in the system. This overlay network is created using a simple gossip-based protocol as presented in [**?**].

The second component of the Networking layer is composed of three sub-components,

which are the *Dispatcher*, the *T-Man* and the *Slicing Service*. The *Dispatcher* sub-component is responsible for the translation of the commands issued through the various interfaces by the consumer into commands that are compliant with the gossip-based protocol(s) used in this system. The following sub-component, is *T-Man*, a gossip-based protocol providing the ability to create and manage structured overlay networks, using various topologies [**?**]. Whereas the last sub-component, the *Slicing Service*, provides the slicing capabilities for this system. This entails, as presented in Section **??**, ordering the nodes according to node-local attributes, then dividing the nodes according to various thresholds. Previously, we have presented the slicing capabilities of this system as being an attempt to solve the *slicing problem* because of how it is implemented. The (current) implementation enables the creation of slices using only one metric, which is completely independent of any node-local attribute, consisting of the number of nodes a consumer wishes to contract.

The last component of the Networking layer is also composed of three sub-components, the *Monitoring System*, the *Aggregation Service* and the *Storage System*. Through the corresponding API, the *Monitoring System* provides to the consumer access to global system information collected and computed by the *Aggregation Service*. The *Aggregation Service* provides system-wide parameters to the peers of the system. These parameters are computed by aggregating the various local parameters from each peer, through local message exchange among peers. To accomplish this in a decentralized and dynamic environment, this service uses a *push-pull* gossip-based protocol, as presented in [**?**]. The last sub-component is the *Storage System*. It provides the distributed storage capabilities of this system and it is accessed through its *Storage API*.

From this overview of the architecture of this system, we can observe that gossip-based protocols are prevalent, which is *the* defining characteristic of this system. These protocols are praised for their ability to strive in highly dynamic and volatile environments, and a considerable amount of literature has been published on the subject. For a discussion on the limitations inherent to these protocols, and its strengths see [**?**]. For a more introductory approach to gossip-based and epidemic protocols in the context of distributed systems see [**?**].

3.3.2 Evaluation

In this subsection we evaluate this system against the evaluation framework, as presented in Section **??**, then against our requirements, as presented in Section **??**.

Evaluation Framework

Again, we rely on the evaluation framework to identify any functional deficiencies present in this system, with respect to the essential requirements of peer-to-peer collaborative systems. As stated previously, the requirements of this framework can be implemented alongside other functional requirements as part of a single module.

The first two phases, **Advertise** and **Discover**, are coupled together within the *Peer Sampling Service*. But they are not exhaustively implemented. We underline the fact that the system provides no mechanism to advertise the resources, a node has to offer, to the network explicitly. Either by using formal specifications, as proposed in [**?**], or using any other strategies. Rather, this service presents a node to the other nodes using a gossip-based protocol. We deem these functional requirements to be partially fulfilled, because it is desirable to be able to advertise and discover resources based on the capabilities.

The **Select** phase, is accomplished by the *Slicing Service*. The authors designed this system to use a gossip-based protocol to automatically partition the network into slices according to a metric [**?**], but was not implemented and it create slices based on the number of nodes desired. Thus, we deem this functional requirement to be partially fulfilled, because it is not possible to select the resources according to their capabilities.

The **Match** phase is suspected to be realized by the *Slicing Service*, by furthering the specification of the select query. But the limitations of the gossip-based protocol used are not clear. Because the slicing mechanism proposed by the authors is based on a utility function that describes the usefulness of a node with respect to the other nodes, using node-local attributes; and to introduce additional dimensions to perform multi-dimensional slicing using a utility function is not be the best approach since it introduces all sorts of statistical distortions [**?**]. Thus, we also deem this functional requirement to be partially fulfilled, because support for multi-dimensional slicing is not explicitly provided.

Bind, the fifth phase, is achieved using the *T-Man* gossip-based protocol. This protocol is used to create a ring overlay of all the peers contained in a specific slice, 1 ring per slice, resulting in mutually exclusive slices. There are no indications on how to cope with concurrent attempts to bind overlapping sets of resources, given that these resources satisfies multiple selection queries. This is due to the fact that the slicing mechanism uses a single metric, namely the number of requested nodes. Then instances of the concurrent binding problem would only occur in very a specific case. Manifesting itself in the form of two or more concurrent requests, for which the total requested number of nodes combined exceeds the total number of available nodes, but the number

of nodes per request is inferior to the total available nodes. The problem then becomes, which application is entitled to have their request fulfilled? Because no a-priori ranking mechanism (or reputation mechanism) to decide which consumer should be favored is provided, the behavior of the system in response to this characterization of the problem is undefined. Again, we deem this functional requirement to be only partially fulfilled, because of the lack of concurrent binding requests support.

The sixth phase, referred to as the **Use** phase, is satisfied by the *Dispatcher*. It translates the higher-lever API requests into the appropriate low-level gossip protocol commands sent to the other nodes, resulting in the utilization of the resources. This functional requirement is completely fulfilled, because this system provide the ability to use the selected resources.

The last phase, **Release**, can be accomplished by the *Instance Management API*, granting the ability to the consumer to control the VMs instances it was assigned. The automation of this phase would be the result of the co-operation between the *Monitoring System* and the *Aggregation Service*. collecting global information about the system, such as the current load in a slice, and acting on these results by preempting the necessary amount of resources. This functional requirement is also completely fulfilled, because using the *Instance Management API* in concert with the *Monitoring System* and *Aggregation Service* provides the ability to release resources when they are no longer required.

According to the evaluation framework, this architecture exhibits *some* of the characteristics that are essential in a peer-to-peer collaborative system. Only *two* out of the *seven* functional requirements are completely fulfilled, and *five* are only partially fulfilled either conceptually or because of the lack implementation. Nonetheless, this system presents a solid foundation that could be extended to meet all the requirements, because there are no apparent design decision that would explicitly prevent it.

Functional Requirements

We evaluate this system against our functional requirements, defined in Section **??**.

The first requirement, states that it should be possible for the infrastructure to be constructed using only commodity hardware and that it should host multiple applications at once. By relying solely on user contributed resources and by providing the ability, to the consumers, to create mutually exclusive slices for their applications, this requirement is deemed to be completely fulfilled.

As for the subsequent requirement, which states that every services should be provided by the contributing resources, not 3rd-parties, we can conclude that it is fulfilled. We arrive at this conclusion by observing that the proposed system relies solely on con-

tributed resources and thus it does not introduce any 3rd-parties.

The following requirement states that system should not introduce any single point of failure, and the system should be (somewhat) resistant to ubiquitous failures. As a consequence of being fully decentralized this architecture does not introduce any single point of failure. Using gossip-based protocols to create and maintain the overlays it is able to cope with a very dynamic networking environment. Failures or node leaving/joining the network, are well supported by these protocols, thus we deem this functional requirement to be fulfilled [**?**].

The following requirement states that no special or dedicated hardware should be necessary to participate in this system, and consequently that memory footprint should be as low as possible as to enable a wider range of (weaker or lower-end) resources to participate. Nothing requires from a participant to acquire any special or dedicated equipment to participate in the system. For the IaaS model it is required to use full virtualization technologies, increasing the memory footprint, and it is not clear as how it would fare on lower-end resources. Thus we cannot deem this functional requirement to completely fulfilled, because it uses full-virtualization technologies and the authors have not shown the memory footprint to be small.

The last requirement states that the system should be able to provide scalability to the applications deployed, and consequently provide dynamic membership capabilities. As stated previously, the use of gossip-based protocols facilitate the support for dynamic membership capabilities because of their resistance to failures and arrival/departure of nodes. Consequently, this functional requirement is deemed to be fulfilled.

Ultimately, this system fulfills *four* out of the *five* functional requirements explicitly, making it almost a viable solution, but not quite. We put the emphasis on the fact that the system adopts sensible approach to solve our problem, but the limitations present dissipates any hopes of redeeming it. To back these claims, we want to outline the fact that a lot of the problems of providing a distributed computing platform are simply relayed to the end-user when adopting a IaaS model. Because it reverts the responsibilities to mitigate the problems, back to the user, in the form of (proper) configuration of the VM instances required for any applications. Thus, we think that this system does not, and could not without substantial modifications, provide a PaaS computing platform or a system capable of addressing our research requirements.

3.4 Discussion

In this section, we discuss the various details and implications of both projects, and provide a reflection on what they accomplish from a high-level overview.

The Cloud@Home system offers a decent solution to the volunteer cloud computing paradigm but also with respect to the functional requirements of peer-to-peer collaborative systems. This solution trades off full decentralization and the capacity to leverage lower-end resources for augmentation of performance and QoS. It introduces possible third-parties, in order to provide these performance and QoS guarantees, and thus it breaks our perspective of being fully self-contained with respect to the consumers and contributors. It aims at providing a business model to transform current computing resources into a utility that can be monetized.

In contrast, P2PCS provides an architecture that can transform a group of computing resources into a application deployment platform, corresponding to the IaaS model. This system provides the ability to leverage resources using full-virtualization, but it does not provide a business model to justify the viability of the infrastructure, or to provide any incentive to entice possible contributors. Like many open-source and community-driven project it anticipates a self-policing behavior from the community. The API proposed to interface with the resources is analogous to traditional IaaS APIs. The strength of this system lies in the adoption of gossip-based protocols to accomplish the underlying network creation and maintenance. Consequently, by using these protocols it mitigates a lot of the complexities and difficulties of operating a distributed system over unreliable network infrastructure, such as the Internet.

Both of these projects offer interesting architectures supporting a multitude of desirable features for volunteer cloud computing and public-resource computing, but from our perspective some key features are still missing. Notably the ability to leverage a collection of commodity devices to provide a multi-application computing platform akin to a PaaS model, by limiting the memory footprint. The PaaSe model has different concerns compared to the IaaS model, with respect to security and isolation. The latter uses VMs, to isolate the contributing environment from the hosting environment. And it is not clear how this could be accomplished in either projects without resorting to VMs. Consequently, providing a PaaS model, using the contributed resources rather than leveraging a commercial service provider, remains unresolved for P2PCS and problematic for Cloud@Home when using lower-end resources.

Ultimately, due to the deficiencies outlined, we express the need of creating an architecture based on the observations we made while evaluating these projects.

Chapter 4

Architecture

In this chapter we present the architecture we have proposed, and illustrate the different decisions that influenced our design. We designed this architecture based on the observations made in the related works, and our design takes into account our requirements, as presented in Section **??**, and the essential functional requirements for peer-to-peer collaborative systems, as presented in Section **??**.

The first section provides a high-level perspective of the architecture, whereas the subsequent sections present and express the design rational behind each layer. We then conclude with a discussion of the requirements that were addressed by designing this architecture.

4.1 Overview

We used a multi-tiered approach for the architecture, similar to Cloud@Home and P2PCS, because it provides high modularity and loose coupling of the concerns addressed by each tier. In this architecture we differentiate between 3 types of nodes, namely a *Worker node*,

Figure 4.1: Overview of the Proposed Architecture.

a *Data node* and an *Application Deployer node*. A *Worker node* consists of a computational resource, executing tasks encompassing *all* the functionalities of the applications.

Whereas a *Data node* consists of a storage resource, performing all the database and storage related tasks. The *Application Deployer node*, is the node that contracts contributing nodes to host an application in this system.

Starting at the lowest-level, we have the **Network Layer**. It is responsible for all the network responsibilities including the creation and maintenance of the overlays, the communication amongst the participants, and to provide the high-level layers with an abstraction of the underlying physical network structure.

On top of the Network Layer, we built the **Virtual Layer**. It is responsible for abstracting the physical characteristics of each node and providing a homogeneous interface to a collection of heterogeneous resources using virtualization technologies.

Finally, the top-most layer is the **Application Layer**. It exposes an API to the consuming node used to build applications. This API is a minimal specification of the services required to transform raw resources into a complete computing platform, similar to PaaS computing platforms.

4.2 Network Layer

In this section we present the *Network Layer* and the rational from which it originates, as stated in Section **??**, it is responsible for providing all the networking capabilities for this system. It accomplishes this using two abstractions, *the Ring* and *the Fellowship*. We present the two abstractions and explain their relevance for this architecture.

When devising a distributed system, one is confronted with fundamental problems such as: *How can we connect a collection of computers, using the Internet, and maintain connectivity among them?* From what we have presented in Section **??**, we can address this problem by creating an overlay network to connect a collection of computers and maintain connectivity among them. But then the question becomes, which topology of overlay network is best suited for our requirements, or is it not relevant in this context?

A case-study presented in [**?**], outlines the difference between the various topologies an overlay network can have in the context of a peer-to-peer collaborative system. The authors conclude that *both* topologies of overlay networks, structured and unstructured, exhibits different characteristics which can be desirable for peer-to-peer collaborative

systems, but none of the topologies provides a comprehensive solution addressing the functional requirements identified by the authors. Consequently, we know that a single overlay network will not address all of the functional requirements of peer-to-peer collaborative systems. Before proceeding further lets recall quickly what the two projects have done to mitigate these requirements. If we look at the networking layer for both projects, Cloud@Home and P2PCS respectively, we notice that the former favors a centralized management entity. This entity is used as a central connection end-point, each node contacts this entity to be part of the network, and this entity maintains the connectivity in the network. The latter focuses primarily on epidemic and gossip-based protocols to generate the overlay networks and to maintain them. Adopting a slightly different strategy, each application generate and maintain their own overlay network (slices) resulting in a federated management structure, where each slice is responsible for its nodes.

In order to respect **Requirement 3**, as presented in Section **??**, we must not opt for a centralized solution, because it introduces a single point of failure in the system. We chose a structured overlay network topology for our first abstraction, the *Ring*, providing the characteristics and features that are desirable for its intended purpose. For our second abstraction, the *Fellowships*, we open the possibility to use any overlay network topology.

In the following subsections we present these abstractions, and we present the different design decisions made and the rational behind them.

4.2.1 The Ring

We have created the *Ring* abstraction to separate the concerns of the publicly available portion of the networking infrastructure from the privately available portion or the application environment. In this subsection we present the design rational, followed by the conceptual manifestation of this abstraction and its implications for the architecture.

Figure 4.2: Abstract Representation The Ring.

The necessity of devising an abstraction to encapsulate the *public* environment in which our architecture operates, is an attempt to provide a construct that exhibits some of the desirable characteristics of a peer-to-peer collaborative system. We have identified four distinct features that we deemed desirable for this construct:

1. Connecting the participants together in a public environment, such as the Internet, resulting in a *public meeting point*.

2. The management responsibility should not be centralized, as to prevent a global system failure occurring as the result of a(ny) subset of participants failing or leaving.

3. Provide querying mechanism to *publicly* locate applications that are deployed using this architecture, and ensure its reliability.

4. Any information that might be required by a(ny) participant to join an application should be contained in this construct, and the consistency of the information should be ensured to prevent malicious participants from corrupting the information and paralyzing the system.

In order to provide the first feature, it is clear that an overlay network would suffice, independent of its topology. We can use the same rational for the second feature, because none of the topologies imposes the centralization of the management responsibilities and they can operate in fully decentralized environments. In order to provide the third feature, it is necessary to opt for a *structured* topology, rather than an unstructured topology. Because, as we have presented in the Section **??**, unstructured overlay networks do not provide a deterministic querying mechanism. Furthermore, the last feature entails that this construct is fully self-contained and thus, it must possess some storage capabilities to store the information.

Because of its storage capabilities, its deterministic querying mechanism and its fully decentralized architecture we opted for a structured overlay network. As a matter of fact, we are using *Kademlia* DHT, which provides the ability to locate any node, deterministically, in a time complexity of $O(logn)$ [**?**].

Using this DHT to store information is intuitive, but using this technology into the context of this architecture imposes further constraints. We need to ensure that malicious nodes cannot compromise the system by polluting or corrupting the information stored in the DHT. This corresponds to a *storage attack*. Essentially, all DHT are inherently vulnerable to this type of attacks, because the design assumes a cooperative environment, where each node is benevolent and most implementations do not address this directly, rather they leave it to the application developer. For an extensive survey of the security techniques applicable to DHTs, see [**?**]. Thus we require the following changes to the DHT, to ensure consistency and correctness of the information stored in the DHT to an

Figure 4.3: Fellowship Abstraction.

acceptable degree. Restraining the writing access of the participants, to only those that are *application deployers*[1], reduces the threat model posed by possible storage attacks.

Then, the problem becomes: how can we ensure the benevolence of the *application deployers*? It is not possible to ensure absolute benevolence, but it is possible to limit the potential consequences of a storage attack by some malicious writer. We limit the application deployers ability to write values, to only two keys. One of those keys is named, the *template key*, and it corresponds to a public repository containing the attributes a node can use to advertise its resources, be it dynamic attributes such as CPU usage, or static attributes such as total amount of physical memory. Whereas the other key, refers to the application deployed by the participant, and can only be written to by this participant, but it can be read by all. It contains the list of nodes composing the *Fellowship*, providing the ability to a node to reconnect to a previous application following a failure by contacting any nodes in the list.

Ultimately, the implementation of the DHT remains unchanged, and we enforce these restrictions in the interface defined for this layer[2].

4.2.2 The Fellowships

This abstraction is responsible for the *private* portion of the networking infrastructure. We use the word *private* to distinguish between publicly available networks, such as the *Ring*, and privately available networks, such as the network of contributing nodes for a single application. In this subsection we present the design rational for this abstraction and also the conceptual manifestation, and its implications for this architecture.

We design this abstraction to have a clear separation between the nodes that *are* contributing and the nodes that *wishes* to contribute, by segregating them into different networks. This confers the private characteristics to this abstraction, and accentuates the modularity of this architecture. Consequently, this abstraction needs to account for the following essential features:

[1] Further information on the types of participants is provided in Section **??**

[2] More information about the interfaces to the different layers in Chapter **??**

1. The ability to discriminate between contracted nodes and malicious nodes (pretending to be a contracted contributing node).

2. Allowing any nodes to join and leave in a graceful fashion, and consequently remove any single point of failure.

3. Providing means for secure communications among the nodes contributing to the same application.

The *Fellowship* consist of a private collection of nodes interacting together exclusively. They are not required to interface with any public environment directly as required for the *Ring* when new nodes are arriving.

Our primary focus, with respect to this abstraction, is the isolation of these private application networks, by controlling which participant can join and which can't, as expressed in first feature. We can ensure this using a *whitelisting mechanism*. The application deployer will create a list, by selecting candidate contributors, and will **only** contract these contributors to host the application. This is especially useful in *semi-private* or *semi-trusted* environments. A *semi-trusted* environment, is an environment in which the different parties (i.e., contributors and consumers) are known to each other a-priori, but are connected together through public networks, such as the Internet. If it is impossible to establish a semi-trusted environment then other means of authentications are required. In a fully untrusted environment, such as the open Internet, it is very difficult to ensure the identity of a specific participants without any 3rd-party acting as an authority. Although, we could mitigate this by implementing a reputation system that would gradually increase the trust component for each participants as they contribute [**?**], but we leave it as future work.

For the second feature, we are inclined to implement a similar strategy as Peer-to-Peer Cloud System did with the slices. Similarly, we suggest to create one overlay network per application, and we do not impose any topology, but rather leave it as an application dependent design decision. Since we are not restraining the topology of the overlay network, we extend greatly the extensibility and flexibility of this architecture, because depending on the application's networking requirements a topology might present more advantages over another topology. For example, if your application interacts with a plethora of sensing devices distributed geographically, an *unstructured overlay network* would provide the ability to aggregate the information while flooding or performing random walks through the network, thereby providing the ability to perform complex queries. Whereas, a *structured overlay network* would require multiple simple queries and

post-processing of the results in order to aggregate them, achieving the similar results less efficiently.

The last feature, can be provided by implementing a Public-Key Infrastructure (PKI), where the application deployer is the certificate authority providing public keys to every node, given that their identity has been verified by the registration authority, which can (and should) also be the application deployer. Then, by using this technology we can ensure that the communications are encrypted, certified, and secured.

Finally, this abstraction provides the ability to control which participant is allowed to participate in the collaborative application, but also the ability to secure the communication channels using public-key cryptography. Depending on the networking requirements of each application, it is possible to tailor the *Fellowships* structure to fulfill these requirements.

4.3 Virtual Layer

In this section we present the *Virtual Layer*, defining its purpose and the implications for this architecture. In other words, we present what this layer should provide for our architecture and how should it provide it in order to respect and fulfill our requirements. We then showcase how our approach differs from the previous approaches, used by the two projects evaluated in the Chapter **??**.

There are three essential features that this layer should provide. The first and foremost feature this layer should provide, a security mechanism to isolate a dedicated execution environment in contributing host system. It should *sandbox* the contributed resources from the contributors operating system, in such a way that no matter what gets executed within this *sandbox*, it can never access the OS hosting it or better yet, it cannot know whether it is a virtualized resource or a physically dedicated resource.

Another important feature this layer should provide, is the ability to abstract the resources from their underlying physical architectures. Using virtualization, it is possible to abstract away the specificities of the underlying physical resource, and to present all the resources in an agnostic fashion to possible consumers. In other words, it is possible to present two physically different resources, such as a computer using a *Intel* CPU and a computer using a *AMD* CPU or even a *ARM* CPU, as identical computational resources, distinguishing them solely based on their capacities rather then proprietary physical architectures.

The last feature this layer should provide, is the ability to control and restrict the amount of resources contributed to the system. That is, a contributing node should

be able to restrict its contribution to desired threshold. Consequently, when allowing only a small percentage of the available resources to be contributed, the virtualization technology should aim at minimizing its overhead memory footprint as to maximize the usage of the contributed resources.

Then, the design question becomes, which virtualization technology offers these features without compromising our initial research requirements, presented in Section **??**. *Light virtualization* technologies, are a potential solution to this design question, but let's examine to which extent they are different from *full virtualization* technologies.

As prescribed in the first feature, light virtualization provides the isolation required to securely execute applications without interfering or compromising the host execution environment. Due to their sandboxing properties, executions done inside a container are opaque to the host OS, as would be the case with full virtualization technologies. Thus both technologies, light and full virtualization, fulfills this requirement completely. Same goes for the second feature, which is also provided by both types of virtualization technologies. Because, both types provides homogeneous abstractions for the physically heterogeneous computing resources, and both types presents the physical resources in an agnostic way. Still, both technologies are equally desirable in the context of the first two features.

Now we need to look at whether or not it provides the ability to restrict the virtualization to a subset of the available computational resources. Docker, one of the containerization technology presented in Section **??**, offers this functionality using their image system. Using the c-group technology, as presented in Section **??**, users hosting containers are able to restrict the processes spawned within a container to only a specific subset of the available resources. As for the full virtualization technologies, it is also possible to restrict the amount of resources available to a VM, through various configuration parameters. But a *difference* ultimately persist between the two types of virtualization technologies. By using lightweight virtualization technologies, we can leverage the libraries and the kernel of the host OS, reducing the possible redundancy of the libraries and binaries to a minimum, as presented in Section **??**. Whereas, using full virtualization technologies, it is likely that some of the libraries and binaries are installed twice, once in the VM itself and another copy could be installed on the host OS. This outlines the major difference between full and light virtualization technologies. Because, by design the overhead incurred by full virtualization, with the notion of a hypervisor residing on top of the OS having to translate the requests and commands from the VMs into intelligible commands for the OS, is far greater than by light virtualization. Because it removes the intermediate hypervisor, and instead uses namespace isolation and resource isolation to

access the resources directly.

We conclude that light virtualization technologies are more apt for limited-resources, because they provide isolation as well as minimize their memory footprints, thereby fulfilling the last requirement.

Using light virtualization technologies is justified due to our specialized requirements, and in this interlude we examine how the previous projects (Cloud@Home and P2PCS) addressed their virtualization requirements. If we recall what both projects proposed, we can observe that **both** advocated the use of full virtualization technologies. It provides the desired isolation properties, as well as possessing the abstractive capabilities necessary for *their* architecture. In the context of IaaS, it is mandatory to resort to VMs in order to provide the ability to host an entire OS on the resources, as advertised by this service model.

We beg to differ with respect to full virtualization as being adequate for *every* service model. For a service model akin to PaaS, VMs provide too much flexibility and extensibility of configuration which complicates the application creation and deployment process. It reduces usability by forcing the person writing the application to consider details about the configuration of the execution environment, across multiple layers rather than only across the application and data layers as presented in **??**. Conflating multiple concerns ultimately hinders the productivity and the usability of the computing platform. We believe that constructing an application using a computing platform, such as a PaaS, that this platform is meant to abstract away the characteristics of the underlying physical resources, enabling the developer to write an application using the interfaces independently of the underlying implementations. Whereas full virtualization provides exactly the opposite experience, by exposing all the underlying resources, it forces the user to decide which implementation to use and how the resources should interact with each other on a lower-level.

Consequently, we choose to use operating system-level virtualization or lightweight virtualization, because it satisfies our virtualization requirements, but also vastly improves the usability of the system. To the best of our knowledge and in all humility we believe to be the first proposing this type of virtualization as a integral part of a computing platform.

4.4 Application Layer

In this section we present the *Application Layer*, and we present how we came about this minimal API specification that reflects the essential features of distributed computing platforms, such as PaaS.

Initially we present the reasons for devising this layer, and then proceed to present each of the components of composing this API: the *Databases and Storages* component, the *Communication and Networking* component, the *Load Balancing and Scalability* component, the *Security* component and the *Application Deployment and Management* component.

4.4.1 Overview

The purpose of this layer is to provide to the application developer the necessary building blocks to develop a scalable distributed web application. We propose a minimal API specification that provides these building blocks. It is minimal in the sense that it is sufficient to develop most applications, but more features can be added to ease the development process of more complex applications.

By definition APIs are extensible, since they provide interfaces to the functionalities contained, while abstracting away the details of the implementations. Using an API to develop applications provides greater modularity with respect to the underlying system, because these applications will always be compatible with this system even if several update occurs, as long as it respects the interface defined initially.

We have defined the essential components of this API by investigating the major CSP that provides a PaaS service model, such as *Google, Amazon, and Microsoft*. For more details on the different services offered by the major service providers, which served as a basis for this proposed specification, and reproducibility's sake please refer to [**?**] [**?**] [**?**]. Then, we have identified the overlapping components, and eliminated the *redundant* components. Figure **??** is a pictorial representation of the resulting API specification:

The components identified by a *solid box* are those exposed to the application developer, and those identified by a *grayed dashed box* are integral parts of the distributed computing platform. The division of concerns is not absolute, and we chose to provide access to the components represented by a *grey dashed box* to the application developer through configuration parameters rather than building blocks.

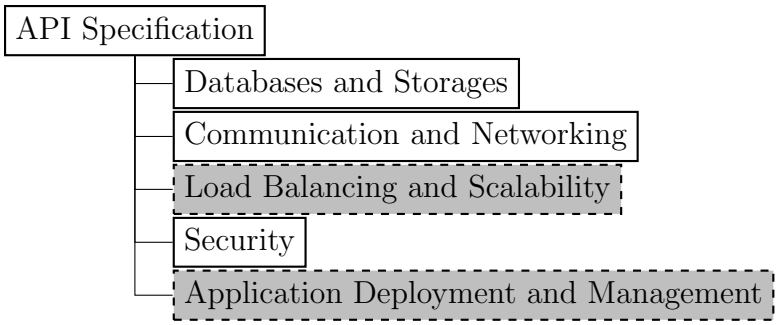


Figure 4.4: API Specification Overview

In the following subsections we discuss in further details each of the five components identified, providing an overview of the functionalities they contribute to this architecture and the reason of their inclusion.

4.4.2 Databases and Storages

The first component of this API, is centered around the necessity to *persist* data or information in the context of a distributed application. We have defined a taxonomy of the primary storage services offered by the major CSPs, and then we discuss the concerns relative to the provision of these services and its consequences on this architecture.

- **Relational Databases:** provides traditional Relational Data Base Management System (RDBMS) facilities.
- **Non-Relational Databases:** commonly referred to as *No-SQL*, this type of databases offers schemaless database facilities, such as key-value stores.
- **Storage:** provides all storage needs with larger space requirements per entry (up to 1TB) and for heterogeneous objects that usually are represented as a binary string (for which the format and content are not relevant and all objects are represented as the same type of object). *A common way of using such storage components is to pair it with a RDBMS, in which we store the meta-data for all the objects stored in the data store. Then this meta-data is indexed and associated with a key that represent the location of the actual data in the data store.*
- **Caching:** provides caching capabilities for fast access to small chunks of data, storing them in memory for future access.

Relational databases, non-relational databases and storage systems in general shares similar concerns about *availability*, *reliability*, and *consistency* in a distributed environment, and thus we can analyze them in parallel.

In all cases, due to the distributed nature of the underlying compositional resources, there is a need to properly *replicate* the persistent data of the application, in order to ensure *consistency* and *availability* of the data.

Were we offering a IaaS computing platform, we would extend concern about the distributed nature of the system not only towards databases, but also towards file systems, because each component have their own file system. Thus, multiple solutions exists, one of which is the distributed file system from Google, the *Google File System*[3] [**?**]. But we are focusing on a computing platform that shares more in common with PaaS than any other service models, and consequently within this context a file system is irrelevant because it is too low-level. Our concerns with respect to the distributed nature of the system persists, and reliability, availability and consistency must be ensured nonetheless.

We then resort to *distributed databases*, and there exists different implementations[4] that provides a plethora of different features crossing the boundaries between relational database, non-relational databases and even storage type solutions. Distributed databases provide the ability to adjust the availability of the data in response to the fluctuations of the number of incoming requests, by adding more instances to the database cluster. Distributed databases are devised around two important concepts: *replication* and *fragementation*. The former represent the ability to replicate the data across several instances, to ensure the availability and the consistency of the data. Whereas the latter represents the ability to decompose the relations between schema entities into several sub-relations and to distribute them across several instances. After which, it is possible to reconstruct the original relations using these sub-relations (or fragments), thus balancing the workload across the instances. Much more information can be found on distributed databases, and to provide an extensive overview of all the characteristics and features of the many variations is out of the scope of this thesis, instead refer to [**?**] [**?**] [**?**].

Rather than committing this architecture to a single distributed database implementation, or even to an implementation for each of these three services *(RDBMS, No-SQL, and storage solutions)*, instead we opted for an extensible design. We provide an universal interface[5] to these database and storage systems, and thus if the solution provided doesn't

[3] As a matter of fact, this is exactly what Cloud@Home proposed to implement to offer distributed file system capabilities within their infrastructure.

[4] We explore these a bit further when discussing the actual implementation in Chapter **??**.

[5] We discuss this interface in greater details in Chapter **??**.

suit their need it is possible to extend the interface to account for other functionalities.

The last service comprised in this component, is *caching* and it is useful in many distributed web applications. It consists of a service that enables the application developer to store data in a cache for *faster* future access. It is primarily used in database-driven web applications, in order to reduce latency by caching popular data in-memory, and thus reducing the number calls to external or physical data sources. *Memcached* is an example of a distributed caching mechanism, allowing to logically combine any unused memory in servers to form a bigger unified cache [**?**]. It is open-source, and we can provide it as a service without much problems.

Finally, we provide database and storage capabilities through the implementation of an interface, enabling the application developer to quickly integrate any open-source and readily available database or storage system implementation. We agree that caching is important for data-intensive applications, but we leave the implementation of this feature as future work due to the scope of this thesis.

4.4.3 Communication and Networking

This component is responsible for everything that is related to online accessibility, presentation of the information using markup languages, and the communication interface between the user and the application.

But, first, we need to underline that the problem of collaborative web hosting remains an open problem in the context of the current Internet's infrastructure. The problem revolves around the ability to name the resources in a dynamic distributed environment, but also how to provide searching and indexing capabilities in that same environment, as well as ensuring content availability. An extensive reflection already exists about this specific problem, and we diligently refer the reader to [**?**] for more information.

Therefore, we must assume that a domain is already hosted in order to access the application using a Uniform Resource Locator (URL), or it is accessed using the IP address directly. By supporting a web framework, we can provide a service to programmatically present the information of the web application.

We provide a communication interface between the application and the end-user using REpresentational State Transfer (REST) based APIs, based on the unanimity amongst the CSP investigated. REST, is a collection of design patterns and guidelines to create scalable web services, and it uses HTTP as the underlying communication protocol. For a complete account of the guidelines and design patterns present in REST, see [**?**].

Ultimately, we provide the services for the end-user to communicate with the web

application using HTTP requests, and provide access to the underlying RESTful APIs to the end-user to interact with the web application. A web server accessible using the current Internet infrastructure receives these requests, providing means to programmatically present the information of the application to the user.

4.4.4 Load Balancing and Scalability

It is common for people to conflate *load balancing* and *scalability*, because their of semantic similarities relating to their purpose. In other words both are concerned with optimizing the performance of the system, but operates on different levels. Load balancing consist of distributing the workload as evenly as possible across the different resources to optimize the resource consumption. It can be achieved by using *task queues* as a service, as shown by the CSP investigated. Whereas scalability differs in how it attempts to optimize the performance of the system. Load balancing optimized the system performance by devising the optimal scheduling plan for the current workload. Conversely, scalability optimizes the system performance by preempting resources to respond to the fluctuations in the workload. The workload could diminish to a point that most of the resources are idling, and enforcing scalability would preempt some of resources in order to maximize the utilization of the remaining resources, and vice-versa.

In this subsection we present *task queues*[6], as a service or feature of a distributed computing platform. Then we present one, out of many, applicable autonomous techniques to achieve scalability in a distributed application.

Task queues are used to manage the work that happens outside of the normal request-response cycle of web applications. Tasks that are queued are handled *asynchronously*, in order to prevent interruptions or delays to the request-response cycle. Tasks can also originate from other sources, such as time-consuming maintenance operations and long-standing background processes A workflow commonly used alongside task queues is:

1. Define the maximum request-response interval your application will tolerate.
2. Evaluate if a job will surpass that interval.
3. Given that it surpass this interval, schedule a task in the task queue.
4. Upon completion of the task, store the result in a cache mechanism.

[6] *caveat lector*: Since queues are a fundamental component of this architecture we will only address *task queues* as a service or feature to build applications, and then we will discuss the details of this architecture in further details in Chapter **??**. Because it digresses too much from the service offered in the context of the application layer.

5. Respond, when possible, by reading the value from the cache.

Combining such a workflow with the concept of independent computational entities, such as *worker nodes* in a distributed system, monitoring the queues for any new tasks to perform, is generally sufficient to distribute the workload across the different nodes and to achieve *decent* load balancing.

These task queues are central to this architecture and this is why we do not provide them as a service in the application layer, because it would be redundant. When explicitly necessary one could extend this architecture to incorporate any task queue implementation of their choice. Although, we acknowledge that we can get into more details with respect to load balancing since sophisticated algorithms have been proposed to handle different specific cases, we deem it to be *application-dependent* and thus not as relevant for this minimal specification for an API.

The other aspect of this component is scalability, we rationalized scalability as being the ability to respond to the fluctuations in the workload, either manually or autonomously. The PaaS providers we have investigated, offers the ability to the user to specify *static* or *dynamic* policies. These policies usually are expressed in terms of SLA, and ensure that the different Service-Level Objective (SLO) are respected by requesting or returning resources accordingly. SLA are contracts that specifies the terms of a service between the consumer and the producer, whereas the SLO are metrics used to measure the service provisioning performance of a provider, preventing any misunderstanding between both parties. There are many characteristics important when defining SLOs and forming proper SLA, we focus on a slightly different approach to provide scalability and thus we refer the reader to [**?**] [**?**], for more information on service level management.

We provide scalability by implementing a decentralized autonomic controller for each type of node[7], based on [**?**]. In order to achieve scalability, the application developer will specify it's policies with respect to the utilization of the two types of nodes, in the form of a percentage. Using this policy we will attach an autonomic controller for each type of node, and it will be used to monitor and make the appropriate adjustments in response to the changes in the performance.

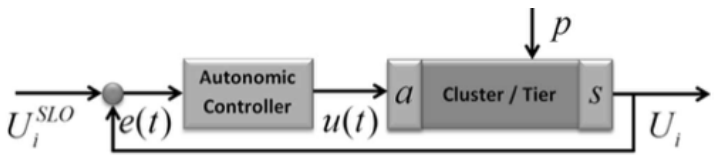


Figure 4.5: Proportional-Integral-Derivative (PID) Controller [**?**]

[7]see Section **??**

The controller used is Proportional-Integral-Derivative (PID), shown in Figure **??**, and it operates as closed-loop control system. It computes the difference between the desired performance U_i^{SLO}, described in the SLO for that type of node, and the performance observed U_i, which is represented by $e(t)$ in this figure. It will interpret the result, $e(t)$, according to this function and return the changes required to remain within the U_i^{SLO}, denoted by $u(t)$:

$$u(t) = K_P e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

The first component of this controller, is known as the *proportional component*, and it adjust the result in *direct proportion* of the delta between the desired performance level and the observed performance level. The second component computes the integral of the delta between the desired SLO performance and the observed performance, adjusting the result with respect to the *historical observations* relative to the delta. The last component computes the derivative of the delta between the desired SLO performance and the observed performance, attempting to *anticipate* the upcoming performance level. Three coefficients are specified to adjust the weight of each of those components, and regulate the behavior of the controller. It is then possible for the application deployer to only specified the desired utilization level for each types of nodes, and this autonomic controller will adjust the resources to respond to the fluctuations in the workload autonomously.

Ultimately, using task queues as an integral part of this architecture and leveraging decentralized autonomic controllers, we are able to provide autonomic scalability and efficient load balancing. This component is *application-dependent*, thus we provide some functionalities as intrinsic features of our architecture, but we leave the more sophisticated functionalities to be provided by the application developer.

4.4.5 Security

The Security component is concerned with providing means to establish/entertain *secure communication channels* between the different nodes, but also between the end-user of the application and the node responsible of handling the requests. Another major concern is access control, including *authentication* of the end-user and providing *multi-tenancy*. We present each of these concerns, and explain how they are accounted for in this API.

The first concern is to provide means for secure communication between the nodes. It can be ensured using standardized technologies such as communicating over TCP/IP using Secure Sockets Layer (SSL). SSL certificates are usually negotiated between a web

server and a client. In the context of this architecture the web server is (*usually*) hosted on the application deployer node and the client is a candidate node. Consequently, a candidate node would negotiate its certificate with the application deployer to establish a secure communication channel between them. Upon successful negotiations the node and the application deployer are able to communicate securely using the SSL protocol. The communications between the end-user and the web server can also be secured using standardized protocols, by communicating using HyperText Transfer Protocol Secure (HTTPS), which consists of using SSL on top of HTTP. The negotiation for SSL certificates would be done between the web server and the end-user. We use well established standard technologies (SSL, HTTPS, etc.), as would any other web application when required to communicate over unsecured networks, such as the Internet.

Access control is another important security concern, controlling access to resources based on various schemes, using *authorization* mechanisms and *authentication* mechanisms. We can identify two basic types of accesses in our architecture: *access from node to node* and *access from end-user to the web application*.

The first type of access is illustrated using this example. If we introduce a *Byzantine* node, that is a node for which the requests or responses are incorrect, either because of an error in the computations or by malicious intent, and it requests to drop all the tables in the database. How do we differentiate between this request being erroneous or malicious in intent, and a legitimate request to drop all the tables? There exists multiple schemes and strategies to enforce various level of access control, and depending on the context in which the application is deployed it can vary substantially. Thus, we implement a pragmatic strategy for access control, where only the application deployer is allowed to perform administrative tasks. This is known as Mandatory Access Control (MAC). The identity verification is done using a simple challenge-response between the nodes to assess the identity.

The second access type, refers to the authentication of the end-users interfacing with the application deployed. Authentication, as a service for applications, *can* be provided using Single-Sign-On (SSO) technologies. For example, incorporating hooks to the Google SSO API provides the ability to the application deployer offload the authentication responsibilities, using a *delegation protocol* such as OAuth 2.0 [**?**], to a trusted third-party. Upon authentication, the user will receive a *OAuth* token, also known as a *bearer token*. When interacting with the application, the user presents this token to demonstrate its identity as certified by the trusted third-party. These tokens have are valid for a certain period of time, after which the user is require the obtain a

new token. The user will also be able to maintain a session using this token. For more information on Google's SSO API, see [**?**].

Authentication can be application dependent. Some prefer resorting to a 3rd-party to provide authentication, whereas others may prefer holding the credential in a database and enforce authentication themselves. It really depends on the security requirements of the application, and thus we leave the responsibility to the application developer to provide control over this type of access.

This leads to the final portion of the security component, *multi-tenancy*, consisting of the ability to provide parallel multi-user support for an application without providing each user with a dedicated instance of the application. Consequently, seasoned web developers will find using this architecture very intuitive, because instead of forcing sophisticated design patterns, resulting in a steeper learning curve, we favor the use of *sessions* to achieve multi-tenancy. By using one session per user, it is possible to encompass all the information specific to this user in a self-contained web primitive.

Finally, we provide secure inter-node communication via SSL, whereas secure communication between the end-user and the web application via HTTPS. The access control scheme provided is minimal and only the application deploying node has the ability to perform administrative tasks. We provide multi-tenancy by maintaining parallel user-sessions, which are provided using a SSO service from a trusted third-party or maintained by the web server.

4.4.6 Application Deployment and Management

This component encompasses all that pertains to the administrative tools to manage an application. Due to the distributed nature of our architecture, we are faced with a challenge foreign to current PaaS providers and it is related to *source-code distribution*. Every other service in this component, can be commonly found in most of the PaaS providers, notably *application configuration* and *various monitoring capabilities*.

The distribution of the source-code of the application is a security concern. Because of the distributed nature of the infrastructure it entails that a contributing node trusts the application deploying node and the source-code distributed. Multiple sophisticated scheme exists to *reduce* the trust-level assumed between parties, with regard to source-code distribution, in a distributed context.

One example consists of creating *bundles*, or atomic units, of code and data, which can then be distributed and executed [**?**]. This deployment framework provides authentication and authorization mechanisms, and certifying capabilities, creating a

trust environment where source code can be distributed and executed securely. An undesirable consequence, is that it forces a complete deployment infrastructure into the architecture. Applications are required to be designed respecting a set of guidelines, leaking into the development process and impeding the flexibility of the architecture. Instead, we propose a way of providing similar guarantees, that ultimately relies on the users judgment and effectively removes any trust factor. Because we provide an open-platform, we apply the same open-source principles[8] to the application it hosts by adopting a *white-box* approach. The workflow of distributing code can be summarized as follows:

- Application deployer stores the source-code into a public or private repository supporting any version control (git, mercurial, svn).
- Upon contracting a node for contribution, the application deployer provides the location of the application repository to this node, and any credentials if necessary.
- The contributing node will then be notified of its content, by presenting the source-code in a text editor or browser.
- The contributing node is then asked whether or not it accepts to execute this piece of code (in a container).
- Finally, depending on the response, the code will or will not be downloaded into the container (and all of its dependencies).

This workflow is designed to work in a fully untrusted environment. In a trusted or semi-trusted environment (cluster or private network), we can simply turn this option off and upon contracting a node for contribution the source-code is directly downloaded into the container, implicitly assuming that the user accepts to execute its content. The CSP investigate offers application configuration services using a web portal that lets the user adjust the different configurable parameters of their application. The configurable parameters varies depending on the CSP and are largely dependent on the underlying proprietary technologies used. We offer *application configuration* using configuration files. The configurable parameters includes: minimum number of nodes, performance policies, type of environment deployed, any communication primitives and any security primitives. Before the creation of a node instance, its configuration file is

[8]Adopting a similar perspective to the Free Software Foundation, who promotes free software and open source development, ensuring maximal transparency [**?**].

read and interpreted. This configuration mechanism is easily extensible and could compensate for any other requirements.

Monitoring is an important feedback mechanism for the application deployer, it is primarily used to anticipate possible bottlenecks or the track the resource consumption of the application. We provide monitoring capabilities by implementing a heartbeat mechanism, in which every nodes periodically sends information for a collection of (user-defined) dynamic attributes. Then the information is collected and aggregated, presenting the result as system-wide monitoring information to the application deployer. This mechanism provides a sufficient perspective of the application state, with respect to the load-balancing and scalability mechanism presented earlier.

Finally, we provide a flexible workflow for code distribution adopting a *white-box* approach. We provide the ability to statically configure an application through various parameters. We provide monitoring capabilities using a heartbeat mechanism that generates system-wide information, using locally available information from the participants.

As future work, we intend to provide dynamic configuration capabilities, in which the nodes will be able to modify the configuration parameters dynamically, when applicable. Providing a visual representation of the data monitored, using graphs to represent the topology and various metrics, would also be desirable from a user experience point of view.

4.5 Discussion

In this section we present a discussion of all the layers, and how they contribute in fulfilling the requirements of both the evaluation framework, as presented in Section **??** and the requirements of this thesis, as presented in Section **??**.

4.5.1 Collaborative Peer-to-Peer System Framework Implementation

Section **??**, presents a framework that illustrates the essential functionalities required for a peer-to-peer collaborative system to scale, by mitigating the complexities inherent to these type of systems. This subsection aims at presenting the different mechanisms implemented in our architecture to provide such functionalities.

If we recall, the framework was composed of 7 key phases relating to the life-cycle of a participant in a peer-to-peer collaborative systems, which were: *Advertise*, *Discover*,

Select, *Match*, *Bind*, *Use*, and *Release*.

The first two phases, **Advertise** and **Discover**, which relates to the advertisement and discovery of resources via formal specification, are implemented as a best-effort mechanism. By this we mean that the newly arrived nodes create a *Resource Specification (RS)* using a common template, and it can be represented as follows:

$$RS(node_i) = [IPaddress, Port, SA, DA] \text{ where}$$
$$SA = \{StaticAttribute_1, ..., StaticAttribute_n\} \text{ and}$$
$$DA = \{DynamicAttribute_1, ..., DynamicAttribute_n\}$$

Note that the set of *Static Attributes* and *Dynamic Attributes* are extensible to accommodate any desirable attributes, and are initially empty. Each application deployer publish any desired attributes to a common repository (or in the context of a DHT, append the values to a specific key used by all nodes to construct their resource specification simply known as the *template*).

Upon creating their resource specification, the nodes gather the list of application deployers using the *Ring*, and periodically send candidacy messages, containing their resource specification. We say best-effort, because the nodes send repeatedly their candidacy messages, until an application deployer contracts them for contribution. Other solutions proposes to publish resource specifications to a repository where it is possible for the consumer to query the repository for the most relevant resources [**?**]. These solutions are adequate if the repository is protected against possible storage attacks, this is not the case for DHTs [**?**]. Our mechanism enables us to mitigate the possible storage attacks, without having to resort to centralized management of the repository, because we do not persist the information.

The **Select** phase relates to the selection mechanism offered to possible consumers to query the resource specifications. In this architecture, in order to select a node, an application deployers must open a TCP/IP server connection on a specific port, and evaluate the upcoming candidacy messages individually by examining its content. As a matter of fact, this selection mechanism resort to a *publish-subscribe* messaging pattern, where the nodes are the publishers and the application deployers are the subscribers. The application deployers are then allowed to subscribe only to a *meaningful* subset of the attributes published as part of a resource specification, representing a *topic*. The nodes uses the concept of topics to publish the various attributes that compose the resource specification. Then, application deployers will do a tentative selection, notifying the resources that it considers them as candidate resources. Only after all the required resources, to host the application, are *tentatively selected* resources actually

selected. This is a blocking operation and can be deferred to a background thread, and automated by defining a selection policy. This policy contains the desired values for the static attributes, and the intervals of desired values for the dynamic attributes.

The following phase is **Match**, it encompasses the ability to formally specify the inter-resource relationship requirements and to enforce them on the selection of resources. As a consequence of using a best-effort mechanism, it is possible to define these relationships as an integral part of the selection policy. For example, once a group of resources is *tentitavely selected*, it is*then* be possible to enforce these inter-resource relationship requirements, and repeat the process until the *tentative selection* fulfills all the requirements in the selection policy.

The **Bind** phase, is also best-effort, because of the mechanism in the *Select* phase. The priority for an application to contract a specific node is determined with respect to the time of the response to a candidacy message. In other words applications which responses were received first are given priority, by the contributing node, over applications which responses were received later.

The **Use** phase states that it should be possible to use the resources contracted in the previous phase to execute the tasks pertaining to the application. It is accomplished by sending tasks to be executer after that the enrollment process (selection, binding, handshaking, initialization of the node) has completed.

The **Release** phase is concerned with providing the capability to release a resource after contributing, either because its SLA prescribes it or because the workload has diminished to the point that this resource is no longer needed. Depending on the SLA between the application deployer and the contributing node (encompassed in a policy), release will happen if possible. Releasing a contributing node consists of a node leaving its current fellowship and returning to the ring, to advertise its resources again.

Future work includes providing more extensive functionalities for each of the phases. We do not include extend these functionalities, because the mechanisms described above are sufficient to operate the system efficiently.

4.5.2 Research Requirements

We now present how the research requirements were fulfilled by this architecture through the different layers.

We can make the following observations concerning the *Network layer*. Using the *Ring* and the *Fellowship* abstractions, we are able to separate the concerns of public-resource pooling and resource provisioning. Furthermore, we are able to provide

multi-application support in a full isolation using the *Fellowships*. Subsequently, the *Network layer* can be easily adapted to different environments (cluster, grid, or open Internet) without incurring any major changes, but by simply selecting which underlying networking primitive is a better fit. As an example, when operating in a highly dynamic environment (Internet), using a unstructured overlay network might be better to cope with the higher churn rate, but when operating in a moderately dynamic environment (shared-cluster) using the current DHT implementation would be more than sufficient to cope with the moderate churn rate. Consequently, by designing this layer accordingly we satisfy **Requirement 1**, because we provide multi-application support using the *Ring* and we do not impose any restrictions on possible contributors, as long as they meet the light virtualization requirement of being able to host a container, which inherently all Linux OS are capable of. We satisfy partially **Requirement 3**, because the *Ring* abstraction provides the ability to create and maintain a decentralized structured overlay network to connect the nodes.

The *Virtualization layer* uses lightweight virtualization technologies to abstract the resources from their underlying physical specificities, providing isolation guarantees similar to full virtualization technology. We can define the environment of an application and all its dependencies using a single configuration file, easing the portability and creating a fully self-contained deployable construct. Consequently, we satisfy **Sub-Requirement 4.1**, to the extent that it reduces the overhead induced by the virtualization technology to *practically* none. Through the use of light virtualization, we satisfy **Requirement 2**, with respect to the desired self-containment properties.

Finally, with the *Application layer* we satisfy **Requirement 2** completely, because we do not introduce any third-parties to provide any of the services described in the API. Although, we do admit that web hosting has to be outsourced, but as we stated earlier in Section **??**, it remains an open-problem because of the underlying infrastructure of the Internet. The *security* component of the API specification, satisfies the part of **Requirement 3** unfulfilled by the *Ring*, by providing secure communication channels and enforcing MAC to regulate and control the administrative tasks. In no way is this component providing *absolute* security to this architecture, and more attack vectors need to be analyzed to account for a realistic threat model.

The following requirement, which states that no special or dedicated equipment should be necessary to participate into this system, is also fulfilled. We deem **Requirement 4** to be satisfied, because there are no explicit or implicit restrictions on the intended hardware to be used. Consequently, recycling the currently available hardware is the

most cost-efficient and eco-efficient, *ceteris paribus*, alternative to participate in this system.

Requirement 5 states that this system should provide *dynamic membership* capabilities as well as scalability to the applications that are deployed. This requirement is satisfied using task queues as a foundational construct of this system, but also by providing dynamic membership capabilities through the use of autonomic controllers to regulate the load on the resources.

Chapter 5

Implementation

In this chapter we present the implementation of the architecture proposed in the previous chapter. In order to provide context, we present the technologies used to implement the architecture and how they influenced the design, when applicable. We then present the central constructs of this architecture, followed by the algorithms and workflows used to provide the underlying logic composing this architecture. Finally, we present a very simple proof of concept that illustrates the orchestration of these constructs and algorithms.

5.1 Technology Used

We have used different technologies in concert to provide the ability to develop applications that can be deployed using this architecture.

The code was written using Python and the Twisted Event-Driven Networking Framework [**?**]. We adopted an *event-driven programming model*, because it provides the ability to reason about problems from a non-sequential perspective, which is useful when implementing distributed application due to their complexity. Event-driven programming is conceptually single-threaded, although multi-threading is supported by Twisted it is not mandatory, and it is offered as an extra feature. Multi-threaded programming, consists of using a collection of threads to accomplish different tasks and requires complex concurrency and synchronization mechanisms. Whereas event-driven programming reduces the complexity inherent to the development of distributed applications, focusing on the possible events and their appropriate responses. Figure **??** illustrates perfectly the distinction between these two programming models and the single-threaded programming model.

Twisted leverages the event-driven programming model by inter-leaving multiple

blocking operations and responding to consequential events, such as completion or failure, using callbacks. A *callback* is a function to call upon the occurrence of an event to handle the results. Twisted allows the developer to specify a callback to handle the successful completion of an event, and a callback to handle the failure of an event; both are encompassed into the *Deferred* abstraction. Twisted orchestrates the callbacks, contained in different deferreds, by registering them with an event-loop, called the *Reactor*. This event-loop controls the thread of execution, enabling operations to block, and keeping track of which callback is associated with which operation. Upon completion (or unblocking) of an operation the event-loop is notified and it passes the result to the corresponding callback, to be processed. By chaining callbacks, it is possible to achieve very complex asynchronous behavior in a cooperative fashion which is at the heart of the philosophy of the Twisted framework. The concept of deferreds influenced the design of the constructs composing this system and their interactions among each other.

The overlay-network used for to *The Ring*, is a Python implementation of the Kademlia DHT using *Twisted* [**?**]. Our decision was based on the fact that it was implemented using *Twisted*, and it could be easily integrated to our architecture incurring minimal disruptions. Another benefi, is to have uniformity in the programming-model used throughout the architecture, by focusing on the event-driven programming model.

For the web server, we used CherryPy, a minimal Python Object-Oriented Web Framework [**?**], because of its simplicity and illustrative capabilities. It enables us to use a web server, without any bloating features and focuses strictly on what is necessary, easing the development of prototypes and proof of concepts. We can substitute this web framework with a more complete web framework, incurring only minimal changes by using our clearly defined interfaces.

To provide virtualization capabilities to the nodes we used Docker Containers [**?**]. Docker provides self-containing capabilities to each node, by specifying the dependencies inherent to this architecture and the dependencies of the applications developed, using *DockerFiles*. They act as configuration files, prescribing the requirements of the environment, all the dependencies, and the quantification of the resources available to execute the containers.

Thus, using these technologies we were able to quickly and efficiently create prototypes, while constructing a solid foundation for this architecture. This foundation is the result of creating well-defined interfaces between the various components, accentuating the modularity and maximizing the extensibility of this architecture.

5.2 Constructs

In this section we present the fundamental constructs of this architecture, namely the **Task**, the **ApplicationNode**, the **Ring** and the **Fellowships**. We present each one of the constructs and discuss their practical implementations.

5.2.1 Task

There is a need to represent a series of consequential actions resulting from an incoming request, into an easily distributable and self-contained entity. This entity *must* provide the ability to return a response to a requester, without any ambiguity regarding the originator of the request in the presence of large amount of requests.
Tasks are heavily inspired by Twisted's concept of *Deferreds*, because they make similar promises. As we have shown earlier, a *Deferred* promises to eventually return from a blocking operation with a result, and to apply the processing logic, in the form of callbacks, to this result. Similarly, a task embodies this promise of eventual completion and provides the ability to specify the result processing logic. As a matter of fact, *Deferreds* are used to chain the various processing steps for any given task. Upon the creation of a task, a *Deferred* is attached to it, consisting of the processing logic that must be applied when this task is dispatched to a node. The principal function required to process this unit of work, is the first callback to be chained in the callback chain of the *Deferred*. Then, any subsequent post-processing function will be chained to that same callback chain, representing the complete sequence of operations. A task corresponds to an atomic unit of work in this system. Analogous to the database atomic transaction property, a task can be in one of two states, completed or not processed at all. Task(s) are fully self-contained and stateless in the sense that any task can be dispatched to any nodes, without having to synchronize the states of the nodes, receiving the task is sufficient to execute it, and carry out the corresponding sequence operations. We distinguish between two types of tasks, a *Worker Task* and a *Data Task*. The former consists of **ANY** type of computational task, whereas the latter consists of task that pertains to persistent data (either storing or retrieving data). Task objects are simple constructs containing the parameters, the module and the function names related to a unit of work. Once created they are dynamically linked to the module provided and the corresponding function.
A **task function** takes exactly 2 parameters: a *task object* and a *list of arguments*. Where the former is the instance of the task itself, and the latter is any parameters that are necessary to execute this function. Thus, a normal function can be refactored into a

task function simply by modifying the signature of the function to take only the *task object* and a list representation of all the (current) parameters; and include logic to extract the parameters from the list.

Developing applications by defining the business logic into self-contained units of work, helps mitigate a large portion of the complexity inherent to distributed systems, but imposes a fairly strict programming model on the developer. Such a programming model may be difficult to abide to in some cases, notably where the services provided by an application cannot be easily parallelized or exhibit inherent serial properties.

5.2.2 ApplicationNode

This construct represents any participating node in the network, and distinguishes between application deployers, and contributing nodes.

Application Deploying Nodes are the nodes contracting the contributing nodes in order to host an application using this system. Hosting, in this context refers to the ability to provide to a nodes the appropriate processing logic (according to their role) and to dispatch the incoming workload to the nodes accordinqgly. The *Application Deployer* is responsible for translating the incoming requests into tasks and for providing tasks to the qcontributors. Then, once a response is formulated as the result of executing a series of tasks, the *Application Deployer* returns the response to the originator of the request.

On the other hand, **Contributing Nodes** process the tasks assigned by the *Application Deployer*. Similarly to the types of tasks, such a node can adopt one of two roles: *Data Node* or a *Worker Node*. The former is responsible for processing Data Task(s) and the latter is responsible for processing Worker Task(s).

5.2.3 The Ring

This abstraction was presented in Section **??**, and provides the ability to connect all the participants in a public environment through a well-defined interface. It also enables the *application deployers* to find potential *contributing nodes*, and it provides a public repository for the common resource specification template. Using a DHT, we are able to connect all the nodes together in a public environment, the Internet. This construct serves as a public meeting space for contributing nodes and application deploying nodes. In order for nodes to interact with(in) the Ring, we have defined a network interface that declares the following functions:

- **bootstrap():** defines the bootstrapping mechanism for the Ring allowing to configure any newly arriving node.
- **connect():** defines the connection procedure once a node has been configured, in order to join the Ring.
- **set():** defines how to store a value in the Ring.
- **get():** defines how to retrieve a value in the Ring.

Application Deployers collectively define the Resource Specification (RS) template and include any desirable attributes by appending them to the template stored in a public repository. This public repository corresponds to the *template key* stored in the DHT using *set()*. Contributing Nodes are then capable of advertising their resources according to the template, by retrieving it from the DHT using *get()*. Upon contracting all the necessary resources, the collection of nodes (including the application deployer) will form a *Fellowship*.

We have decided to use a DHT, because of its storage capabilities. Although, as we have presented in the previous chapter, we could supplant the DHT with any overlay network of our choice to satisfy any (other) networking requirements, such as better response to very dynamic networking environment by using a Unstructured Overlay Network [**?**]. This could be done utilizing the network interface we have defined, and by redefining the mandatory functions.

5.2.4 The Fellowships

This construct provides the ability to connect the selected nodes with the application deployer in a private networking environment. It ensures privacy for the data transmitted, but also enforces any security policing between the nodes.

Currently we use a *whitelisting* mechanism to provide a private environment, meaning that only the nodes contained in the list defined by the application deployer are allowed to participate in this environment. This list is the result of selecting the resources, and publishing the list of these resources under the application deployer's corresponding key in the DHT. Participating nodes are then able to retrieve this list, and use as a provisionary measure of validating incoming communications. More elaborate security schemes can be devised to satisfy a variety of security requirements, such as implementing a public-key infrastructure using a DHT [**?**]

Due to time constraints, we have not implemented this construct as a stand-alone overlay network as prescribed in Chapter **??**, and instead uses the whitelisting approach

to security. As future work we would implement Fellowships using a protocol similar to T-Man [**?**], to provide an overlay network; and we would explore more complete security schemes.

5.2.5 Conclusions

By using these constructs we enforce the adoption of a task oriented programming model from the application developer's perspective, which is fully compatible with the event-driven programming model used to construct this architecture. We are required to have centralized public point of access, due to the underlying Internet infrastructure, which force us to distinguish between two types of nodes, Application Deploying nodes and Contributing nodes. We are providing a flexible public networking platform using a well defined interface, that provides the ability to change the peer-to-peer networking primitive of the platform effortlessly. Finally, we are also providing a private networking environment to contain deployed application execution, using whitelisting like mechanism.

5.3 Workflows and Protocols

In this section we present the procedure to initialize a node, and the protocol necessary to interact and be part of this architecture. We present the initialization workflow of a node, and illustrate the differences between the workflows of an Application Developer's node and a Contributing node. Then we present the protocol used by the nodes to communicate within a Fellowship. We conclude with the presentation of the web protocol interfacing with the web server.

5.3.1 Initializing a Node

A node follows a specific initialization procedure, which varies slightly depending on the type of this node. We distinguish between the Contributing node's workflow and the Application Deployer's node workflow.
The contributing node iterate through this progression of steps, forming a sequential workflow:

1. Node creation.

2. Connect to the Ring.

3. Retrieve Resource Specification Template.
4. Initiate Advertisement mechanism, and wait until selected.
5. Start communication with the Fellowship.

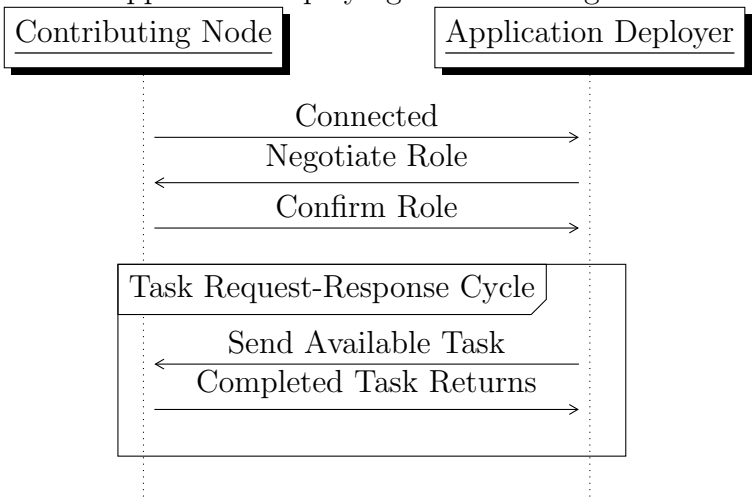
Whereas the Application Deployer's node workflow contains additional steps:

1. Node creation.
2. Connect to the Ring.
3. Read configuration file and update Resource Specification Template.
4. Initiate resource Seeking mechanism, and select adequate nodes.
5. Publish the list of nodes selected to the Ring.
6. Start the web server, and start receiving requests.
7. Start communication with the Fellowship.

Using these two workflows we are able to create a node and successfully join the system. We are also able to deploy applications, by selecting the nodes that satisfies our requirements and joining them together to form a Fellowship.

5.3.2 Fellowship's Protocol

The communications inside a *Fellowship*, are conducted according to the **Fellowship's Protocol**. This protocol defines 6 different actions, that can be taken and by whom. Similar to client-server protocols, here we depict the contributing node as being a client and the application deploying node as being a server[1].



[1]Again, the application deploying node is depicted as a server since it hosts the web server.

5.3.3 Web Protocol

The **Web Protocol** is used to translate incoming web requests into tasks, by implementing a RESTful API between the application and the web server, which enables complete separation from the web hosting and processing facilities. Translation of web requests to task, consists of receiving a web request and then the web server generates the appropriate RESTful request to be sent to the application. For all intent and purposes, any web request is translated into a POST request containing the information about the task, such as which module and function to call, and the arguments to call it with. Upon receiving the request from the web server, the application create a task using the information contained in the request. When a task is completed, the result is retrieved by the application deployer node, who formulates a HTTP POST request using the web protocol and sends it to the web server. The web server then is responsible to present the information back to the originator of the initial request.

The rational behind this, is to completely decouple the presentation logic from the application logic, thus maximizing extensibility and modularity. Any web framework can be used with this architecture, and it can be hosted on any web service provider, as long as it is able to emit/receive HTTP requests it is compatible. It is possible to have more elaborate web hosting scenarios, where multiple web server are spun and they all emit HTTP requests to a subset of nodes of an application for translation. More meticulous synchronization is required to ensure the responses are sent back to the corresponding requesters in that case, but by using sessions primitives it is trivial to locate the originating web server, and consequently the originator of the request. This component is crucial to the claims of extensibility that this architecture makes. It is also important in order to achieve scalability for the application deployed, where web hosting can be an important bottleneck. Lastly, it provides the foundation to implement robust web application, and to provide fault-tolerant mechanisms, such as redundant web servers, without incurring any major changes.

5.4 Proof of Concept: Calculator

In this section we present our very first proof of concept application using this architecture. We demonstrate how to create a sensible solution, by reasoning about the problem at hand in terms of an Event-Driven Architecture (EDA) and illustrate the interaction between the different components. We present the role of each type of nodes

and their responsibilities and present some conclusions relating to this proof of concept.

5.4.1 Overview

The application is a simple web-based calculator. It takes two operands, and using a button signifying the operator to be applied, computes the result using the resources available to the application (nodes). Four different operators are implemented: the addition (+), the subtraction (-), the multiplication (*) and the division (/).

Implementing this application, we make the following assumptions:

- Web Hosting is the responsibility of the application deployer, thus it will host the web server.
- No data is persistent, thus we will not require any Data nodes (or database).
- No fixed number of Worker nodes. The more nodes available, the more concurrent requests can be satisfied.

5.4.2 Component Interaction

In this subsection we present a way to reason about developing applications that is central to this system. It represents the main interactions between the different components of the application.

When creating an application using this architecture, it is fundamental to reason about the problem at hand in terms of the request-response cycle. This enables the developer to understand how to formulate the problem into a compatible solution, using the different constructs available with this architecture. In this particular case we reason about the application as follows:

1. **(Incoming Web Request)** User inputs two operands and click the operator of his choice.
2. **(Generate HTTP Request for App.)** Web server receives an incoming request, then formulate a POST request containing the two operands and indicates operator to apply, and sends it.
3. **(Incoming RESTful API Request)** A node receives a request from the web server, using its web protocol.

4. **(Create Task)** It extracts the information from the request, creates a task and queue it up.

5. **(Task Available)** Upon queuing the task, it becomes available for offloading. It is then dispatched to any available nodes.

6. **(Task Completion)** Upon completing the task, the node will send the result to the dispatcher of the task.

7. **(Task Returns)** Upon receiving the results, using its web protocol it will formulate a POST request containing the result, and it will send it back to the web server.

8. **(Result Returns)** Finally, the web server receives the result and formulate the appropriate response to present the information back to the user.

By writing applications using an event-driven programming model, it results in application built around the notion of an Event-Driven Architecture (EDA). We can identify the four logical event flow layers of EDA, as presented in [**?**], in this application:

- **Event Generator** is the source of (all) the events, which is the web server in this case.

- **Event Channel** is the medium used to transport events from generator(s) to event processing engine(s). In this case the event channel corresponds to the web protocol, where the incoming events from the generator are used to derive the task(s) to send to the event processing engine(s). At a higher-level, we say that the application deployer itself is the event channel, but more precisely the web protocol does the translation from raw event into events that can be processed.

- **Event Processing** is done by *event processing engines* which takes appropriate actions in response to events. In this case, the contributing nodes are the *engines*, which process task (a derived form of event).

- **Downstream Event-Driven Activity** is the downstream activity initiated by an event, and it occurs upon processing the event. In this case, it consists of returning the result of the completed task to the node that dispatched it.

They also distinguish between different types of event flow processing, either **simple**, **stream** or **complex**. Since our problem imposes only one *event generator*, we can deduce that it corresponds to a *simple event flow processing*.

By keeping this kind of reasoning, it is possible to effectively separate presentation logic from the business logic. We now have a clear understanding of the role of both types of nodes: the application deployer will contain the *event generator* and will be responsible to dispatch the events using the *event channel*. It will also use the event channel to handle any *downstream event-driven activities*. Whereas the contributing nodes will only serve as *event processing engines*. This rationalization of the problem at hand is crucial to ensure: proper understanding of the problem domain, how to appropriately separate the presentation concerns from the application logice, and to avoid possible design mistakes impeding scalability.

5.4.3 Application Deploying Node

In this subsection we present the implementation of the Application Deploying Node for the Calculator application.

As presented in the previous subsection, this node encapsulate the sole event generator and event channel of this application. When an event is generated (from the web server), it is passed to the event channel (web protocol) where it is translated into a task and then queued. Using the Fellowship protocol, this node will dispatch the task(s) to the available nodes and collect the result(s) of the completed task(s). The results are extracted from the completed task(s), and are presented to the end-user as a downstream event-driven activity emerging from the processing of the event.

The application deploying node is responsible for the task queues in this specific application, because we host the web server on that same node and there are little to no value in distributing task queues among several nodes in such a simplistic example.

We have implemented the web server, as mentioned previously, using CherryPy. We present a minimalistic web interface that contains 2 fields for each possible operators, 8 fields in total and 4 buttons, as illustrated in Figure **??**xs

And the results is presented using a simple string, consisting of: *"The result is :"* followed by the result of the computation.

5.4.4 Contributing Node

In this subsection we present the implementation of a *Contributing Node*, and then the logic necessary to process the tasks.

A contributing node receives a task from the application deployer node, executes it, and returns the completed task to the application deployer node. All the communication

between the application deployer node and the contributing node is performed using the Fellowship protocol.

In the context of this application, we have defined the logic to process the various tasks into a module called *worker process*. It encapsulates all the logic necessary to perform any operation on the two operands, and facilitates the addition of new operation, by simply to adding a new corresponding function. It makes for a clear separation between the application-dependent code and the architecture related code, and it augments portability of the source code.

In order to execute a task, the *contributing node* will call the appropriate function inside the module and pass it any parameters necessary using a list of parameters. The *contributing node* will append the results to the task object, passed in parameters, and reply to the application deployer with the completed task.

5.5 Conclusion

We can observe the rudimentary workflow of this architecture: receive a request, translate it to task, dispatch the task, execute the task, return the results, and respond back to the originator of request. It exemplify the request-response cycle perfectly, corresponding to an Event-Driven Architecture.

Resorting to this way of reasoning is intuitive when developing web-applications. It is a central characteristic of this system. This way of reasoning facilitates the development of new applications, but also considerably reduces the effort required to translate an existing monolithic application into a distributed application, as we will see in the following chapter.

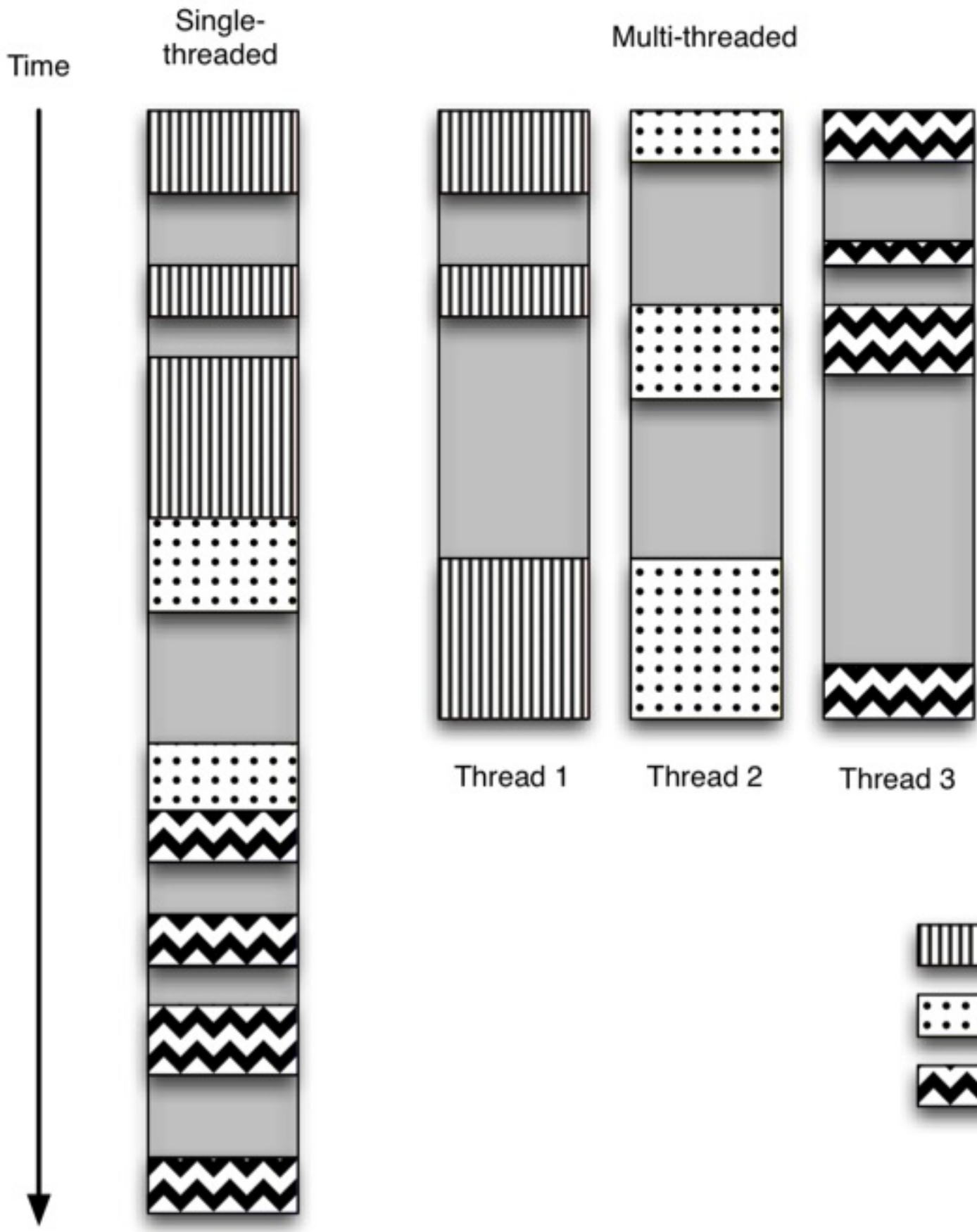


Figure 5.1: Comparison between Programming Models. [?]

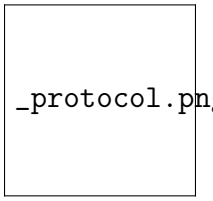


Figure 5.2: Web Request Translation into Task(s).

Figure 5.3: Application Overview.

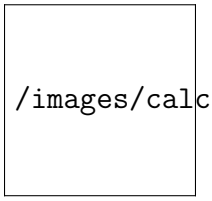


Figure 5.4: Application Deployer Node as part of the Calculator Application.

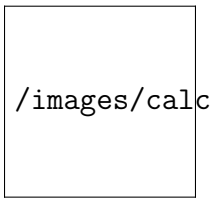


Figure 5.5: Calculator's Web Interface

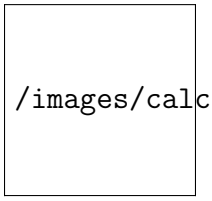


Figure 5.6: Contributing Node as part of the Calculator Application.

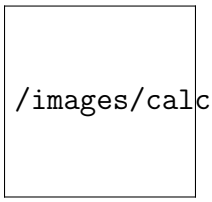


Figure 5.7: Calculator Application Architecture Overview.

Chapter 6

Use-Case: Multi-Document Text Summarization using Genetic Algorithms

In the previous chapter, we have introduced the technologies used and the fundamental constructs of this architecture. Then, we have presented a proof of concept that took us through the mental exercise of thinking about a trivial application, a calculator, in terms of our architecture and proceeded to implement it. In this chapter, we are presenting an example use-case for a more computationally intensive application. Namely we present an adaptation[1] of a system for *Multi-Document Text Summarization using Genetic Algorithms* that was implemented based on [**?**], in the context of a term project.

In the following sections, we present the problem at hand and illustrate how a genetic algorithm is capable of addressing it. We present how we extend the existing work on Genetic Algorithm (GA) and text summarization to account for multiple documents. We then use a similar analysis-based approach to translate this problem into a distributed application. This analysis will serve as the foundation to devise an algorithm that is compatible with an event driven architecture. Finally, we present the characteristics and roles of each type of nodes; then we will present a discussion of this implementation.

[1]By no means, is this an efficient multi-document text summarization system. But it embodies a computationally intensive highly parallelizable problem which is appealing to showcase this architecture.

6.1 Problem at Hand

In order to provide some context, we present what *Automatic Summarization* is in general. Then we present the characteristics of *Multiple-Document Text Summarization*, followed a high-level explanation on how to use *Genetic Algorithms* to solve this problem.

Caveat lector: extensive literature has been written on both, Automatic Summarization and Evolutionary Algorithms, and should be consulted for more information: see [**?**] as a starting point for Automatic Summarization, and see [**?**] as an introduction to Evolutionary Algorithms.

6.1.1 Automatic Summarization

To understand how to automate the summarization of documents, we must understand what constitutes a summary. A summary can be defined as: the product of a reductive process to include salient statements of a source text, by selecting or abstracting the original statements, forming a condensed representation of the argumentations and conclusions drawn; based on [**?**].

To generate a summary, one of the techniques that can be used relies on the *extraction* of linguistic units from a document to represent the information conveyed in it. This technique is known as *extraction-based summarization*. Another popular technique that can be used to generate a summary, consists of *abstracting* the sections from a source text and generate statements, using natural language generation techniques, that convey the same information in a condensed form. This technique is known as *abstraction-based summarization*.

For this use-case, we are interested in extraction-based summarization, more specifically we are interested in a technique called *Sentence Extraction*. Generally, this technique consists of scoring sentences in a text for a set of given metrics, and apply statistical heuristics to filter superfluous or not-so relevant sentences.

Up until now, we have defined a summary has being derived from a single source text, but what if we are presented with a set of documents on a single topic, can we summarize the aggregation of all these texts into a single summary? Doing this will provide the ability to aggregate all the different perspectives of each text, but also to reveal the overlapping perspectives concerning a singular topic. This variation of automatic summarization corresponds to *Multi-Document Text Summarization (MDTS)* [**?**].

6.1.2 Genetic Algorithms

The problem of creating a summary, using sentence extraction, can be defined as the problem of extracting the most salient sentences of a source text, and use them to generate a summary of a given length. We can interpret this problem as an optimization problem and apply an evolutionary algorithm to search the solution space for near-optimal solutions, as proposed by the authors of [**?**] for single document text summarization.

A genetic algorithm, is defined as an adaptative heuristic search algorithm inspired by biological evolution concepts, such as natural selection, mutations, cross-overs, etc. [**?**]. A very high-level overview of the outline of a simple GA is presented in Algorithm **??**. There are several considerations to take into account while using GAs, especially on

Algorithm 1 Genetic Algorithm: An Overview

```
1: procedure GA
2: Initialize population:
3:     population = randomPopulation()
4: Evaluate Population:
5:     fitness = fitnessFunction(population)
6:     while fitness <= desired fitness do
7:         Evaluate Population.
8:         Select parents for next generation.
9:         Perform Crossovers on the parents selected.
10:        Perform Mutations on the resulting population.
```

how to represent an individual in the population, which type of crossovers should be applied, enforcing elitism or not in the selection process, the type of mutations to apply and how to formulate a meaningful fitness function. But this digression is not relevant in the context of this thesis and thus for more information see [**?**].

6.1.3 Extensions Proposed for MDTS

We have proposed extensions to the work of [**?**], in order to provide multi-document summarization capabilities. But first, we present a high-level overview of their procedure, and then we present the extensions have we proposed.

Algorithm **??** is conceptually intuitive. The document is represented using a directed acyclic graph to preserve the order of the sentences in the text. In this graph, the sentences are the vertices and the similarity between two sentence is represented by a directed edge from one sentence to another sentence occurring after this one, ensuring a

Algorithm 2 Automatic Text Summarization using GA: An Overview

```
1: procedure ATS-GA
2:     Represent document as a Directed Acyclic Graph.
3:     Compute similarity metrics, and weight of the sentences.
4: Initialize population:
5:     population = randomPopulation()
6: Evaluate Population:
7:     fitness = fitnessFunction(population)
8:     while fitness <= desired fitness do
9:         Evaluate Population.
10:        Select parents for next generation.
11:        Perform Crossovers on the parents selected.
12:        Perform Mutations on the resulting population.
13:    Extract the summary from the graph.
```

continuous progression in the traversal of the graph. The problem then becomes finding a path that traverses the graph, containing a number of vertices that corresponds exactly to the desired summary length, that maximize the fitness function.Some difficulties can be encountered when tuning the parameters of the genetic algorithm, but it is a problem inherent to GAs in general and not to this specific problem of automatic summarization.

We propose to use this methodology integrally, and execute it in parallel on multiple documents sharing a common topic in order to generate a collection of summaries. This collection of summaries is used as a corpus of salient sentences. Instead of using the GA to find the final summary, we use a redundancy reduction function to remove excessively similar sentence and to achieve the desired summary length. Thus, our new and updated algorithm that is able to summarize multiple document on a single topic is presented in Algorithm **??**.

The rational for this extension is the following, given that the genetic algorithms finds a near-optimal summary for each document independently then by collecting all of these summaries together we have the most salient sentences (in relation to the topic) across all documents. Then, we remove the number of less salient sentences for which the redundancy score was the highest. The score was computed with respect to the number of overlapping words between two sentences, and the closeness to the topic. Although the results were mediocre, we think that with more time (since this was completed in a 2 months period for a term project) we could improve on them by optimizing our redundancy reduction algorithm. Nonetheless, for illustrative purposes this system is adequate.

Algorithm 3 Multi-Document Text Summarization using GA: An Overview

```
1: procedure MDTS-GA
2:     for each document in collection of documents do
3:         Represent document as a Directed Acyclic Graph.
4:         Compute similarity metrics, and weight of the sentences.
5: Initialize population:
6:         population = randomPopulation()
7: Evaluate Population:
8:         fitness = fitnessFunction(population)
9:         while fitness <= desired fitness do
10:            Evaluate Population.
11:            Select parents for next generation.
12:            Perform Crossovers on the parents selected.
13:            Perform Mutations on the resulting population.
14:    while len(summary) > desired summary length do
15:        Apply redundancy reduction algorithm on collection of summaries.
16:    Extract the summary from the graph.
```

6.2 Translation of the Problem for this Architecture

In this section, we analyze the MDTS system, and translate it into a distributed application for our system. First we state the assumptions, then the node requirements. Then we identify the logical event flow layers, and present the different possible event flows. Finally, we present the resulting workflow of the application.

Starting our analysis, we need to make the following assumptions:

- We need to persist (some) data. At least one *Data Node* is required.
- We require an arbitrary number of computational resources. At least one *Worker Node* is required.
- Provide support for multi-tenancy.

Based on the assumptions above we can draft out the node requirements for a minimal implementation of this use-case: (1) Application Deploying Node, (1) Data Node, and (1) Worker Node.

Using these assumptions, we are now capable of analyzing the hypothetical event-driven architecture of our application. First we need to identify the logical event flow layers:

- **Event Generator(s):** in this case the initial events are generated by the web server (incoming request(s) from the user(s)).

- **Event Channel(s):** the web protocol is the event channel.
- **Event Processing Engine(s):** events communicated through the web protocol (event channel) are processed by a data node.
- **Downstream Event-Driven Activities:** once a data node completes the persistence of the data relative to the user's request, the application logic is applied as downstream event-driven activities. This triggers the creation of a series of tasks that will carry the execution of the application, involving data and worker nodes, and finally returning the result to the web server.

From these logical event flow layers we devise three event flows: *Incoming Request Event Flow*, *Processing a Request Event Flow* and *Returning a Response Event Flow*.

Figure 6.1: Event Flow: Incoming Request.

The **Incoming Request Event Flow**, as shown in Figure **??**, embodies the logic of receiving a request from a user and persisting the data (documents) of this request in the database, and queuing up a processing task for this data.

The **Processing a Request Event Flow**, as shown in Figure **??**, corresponds to processing a task related to a document, which implies retrieving the data (documents) from the database, processing it using the genetic algorithm, and persisting the results back into the database.

The **Returning a Response Event Flow**, as shown in Figure **??**, consists of monitoring the database for all the results to be persisted, and then creating and dispatching a task to process the results by applying the redundancy reduction algorithm. Upon completion of the task, the results are sent back to the web server and it presents the results to the end-user.

Figure 6.2: Event Flow: Processing a Request.

Figure 6.3: Event Flow: Returning a Response.

Figure 6.4: MDTS: General Workflow.

Based on this analysis we devise a general workflow that encompasses all the interactions between the nodes and the end-user concerning a single request. Multitenancy is provided through the use of sessions, and we use these session primitives to organize the data in the database for each user.

6.3 Implementation Details

In this section we present the implementation details for each type of nodes. And we outline the specificities of this particular application in the context of our architecture.

6.3.1 Application Deploying Node

The application deploying node is responsible for receiving the requests from the users and translating them into initial data tasks. It is also responsible of dispatching any tasks in its queues, to any available nodes. The application deploying node is also responsible for the task queues, because we host the web server on that same node and there are little to no value in distributing task queues among several nodes in such a

Figure 6.5: Application Deployer Node as part of the MDTS Application.

simplistic example. Because we are using data node(s) we need an additional task queue, thus we have one task queue for the worker tasks and one task queue for the data tasks. Another characteristic of this implementation of the application deploying node, is the task creation logic resulting from a completed data task. As part of the core logic of this architecture, when a data task completes the data node can specify a processing task to be scheduled as the result of this data task. We leverage this capability to generate the downstream event-driven chain of activities, and it represents the application logic. We have implemented the web server, as mentioned previously, using CherryPy. In Figure **??**, we present a minimalistic web interface, where the users can upload their files, specify the different parameters of the genetic algorithm and the summarization parameters.

Figure 6.6: MDTS Application's Web Interface

The results are presented using a simple string, consisting of: *"The result is :"* followed by the topic of the documents and the resulting summary. There is nothing inherently different between this version of the application deploying node and the version presented in the previous chapter, see Section **??**, aside from the addition of a data task queue. It still hosts the web server and it is still responsible for the interactions between the contributing nodes and the end-users.

6.3.2 Data Node

Data nodes are responsible for persisting the data of the application, which is achieved by using a relational database, such as *RethinkDB* [**?**].
We grouped all of the database functionalities into a single module named *data process*,

Figure 6.7: Data Node as part of the MDTS Application.

Figure 6.8: Files table schema.

Figure 6.9: Results table schema.

and thus if we would want to replace the implementation of the database, only this module needs to be rewritten.

It defines a *DataProcess* instance, corresponding to the process created for the database server instance. We created a class to represent a database instance, because a long standing process is required to provide database capabilities throughout the execution of the application. This instance contains all the information necessary to access the database, such as the client driver port and the authentication key. But it also provides the facilities to start and stop the database server instance gracefully, through eponymous functions.

We created a simple database schema, to represent the data that needs to be persisted, and we make a distinction between two types of data: *files*, shown in Figure **??**, and *results*, shown in Figure **??**.

The task functionalities are implemented in the format defined in Section **??**, using a list of parameters and passing it alongside the task object. We defined two different functions that corresponds to the actions for each of the database schema.

The first function is *saveFile*, it consists of extracting a file's content and name from a task object and storing it in the database. Each user has its own database, allowing to store very large sets of relating documents. If it is the first file to be stored, the database and the table corresponding to the *files* table schema will be created. Upon completion of the task containing the *saveFile* function, a task is created to retrieve the file from the database and process it (generate a summary) and to write the results back to the database.

The second function is *resultsLookup* and consist of creating a trigger that monitors the

changes for the *results* table. It can be done using the primitives offered by *RethinkDB*, such as the *changes()* function, which returns a stream of all the changes made to a table or database [**?**]. When all the results have been written to the database, the data node will create a task indicating that the results are ready to be processed.
It is possible to distribute the database across a cluster of data nodes, and we can leverage the clustering capabilities of the DBMS to do it. Sharding and Replication is automatically taken care of, and it is done transparently [**?**]. We only need to specify the argument *–bind all* when starting the instance of the first data node, then any other data node can join the cluster by specifying that argument, but also by specifying the *–join* argument with the IP address of the first data node. Thus to augment the number of data nodes of **ANY** application simply specify the corresponding arguments. Data nodes are simply database servers, augmented to function within this architecture. Their functionalities oscillates around the Search-Create-Replace-Update-Delete (SCRUD) operations. Augmenting availability of the data can be done effortlessly, on-the-fly by recruiting new nodes and creating additional database server instances using the proper arguments.

6.3.3 Worker Node

Worker nodes are the central component to the business logic of any application hosted with this system, as they form the computational resources of the system. The re-factoring effort required to transform any function into a task function, is minimal.

/images/mdts_worker_node.png

Figure 6.10: Worker Node as part of the MDTS Application.

Similarly to the Calculator example, we have encapsulated all the application logic into one single module, named $worker_process. This module exposes two task functionalities, again using the format defined in Section$ **??**
The first function, *retrieveData*, as it name implies is responsible of retrieving the data from the database, to create a summary, and to write the results back into the database. Our original summarization code incurred only one modification, and it was to tailor the code to process only one document (instead of a directory). Ultimately,

this function contains the logic to retrieve a document from the database, and calls each of the summarization functions in the corresponding order.
The second function, *retrieveDataPostProcessing*, retrieves the results from the database, and applies the redundancy reduction algorithm, then sends the resulting summary back to application deploying node (who then will present it to the end-user).

6.4 Conclusions

By reasoning using the event-driven architecture approach, we were able to circumscribe any stateful dependencies in the event flows and mitigate them, enforcing the task properties of being stateless and persisting data to the database. This is a central aspect of this methodology used to translate monolithic applications into distributed applications for this system. It also provides the tools necessary to divide the application logic into meaningful atomic units of work.
The amount of modifications and coding required to translate this application was very small, only 99 additional lines of code to transform the summarization logic into worker task functionalities and 164 lines of code for the DataProcess class and its (data) task functionalities. The *Web Protocol* was written in 113 lines of code, and the web server was written in 145 lines of code.
The *Fellowship Protocol* is application independent, and is written in 514 lines of code. Similarly for the core components of this system totalizing 556 lines of code, including all the logic pertaining to the nodes and the network interface to the *Ring*.
These numbers include the comments, and it showcases how minimal is the translation from a monolithic application to a distributed application, incurring a total of 521 lines of code for the application-dependent components.

Chapter 7

Discussion

In this chapter we present various musings about our architecture and the problems we have encountered. We initiate the discussion by presenting the positioning of this project amongst the wide landscape of distributed computing platforms. Then conclude by presenting various open problems that were encountered along this research path, including peer-to-peer networking primitives for collaborative systems, the state of the current Internet infrastructure, and ethical problems inherent to public collaborative systems.

7.1 Positioning of the System

In this section we present the positioning of this system amidst the sea of different distributed computing platform and systems. It is necessary, because the purpose of many of these systems gets conflated together, and it provokes a severe misunderstanding of what they provide. This is especially true with cloud computing. Cloud computing has become a *buzzword* nowadays, which implies that it becomes the go-to solution for all problems. Even today, a majority of professionals are still kept in the dark amidst all these promises and features, that very few are capable of providing a concise definition of what cloud computing entails. We don't blame them. We agree that the information circulating around about this technology is, confusing, at best. On the other hand, many researchers have tried to devise a proper ontology for this *new* paradigm, see [**?**]. This attempt made us realizes that the definition of cloud computing can be rationalized, and with the help of documents like [**?**] it is possible to clearly define cloud computing. But if we attempt to complete this already expansive definition, by defining what the end-product of cloud computing is, this term *cloud computing* becomes an *umbrella* term and can be used to define any distributed system.

Thus, we present this system as being part of a sub-class of cloud computing, referred as *volunteer cloud computing*, offering a PaaS-*like* computing platform. Notice how we *italicized* the preposition like, we put emphasis on the fact that this system is **not** aimed at providing a volunteer cloud computing PaaS service model, but rather a computing platform akin to it. What sets them apart, is the environment in which they are intended to operate.

Our system is meant to operate in a semi-trusted environment, in which the contributors (usually) knows (in)directly the application deployers, and thus there exist some sort of trust relationship between the two. For example, if we refer to a medium to large size secondary school, such an establishment could be the host of upwards to 200-300 computers. These computers are used during the day for pedagogical purposes, but remains idling or off at night. The school IT administrator, could then use this system to provide these 200-300 computers to academics from a local university wishing to run scientific experiments. This example prescribes a semi-trusted environment, because the academics are in a trust relationship with the IT administrator. We qualify this environment as *semi*-trusted, because both consumers and producers trust each others, but are required to communicate using untrusted networks, such as the Internet. Another example, in the context of local communities, could be a chess club, amongst any other clubs. Given that the club contains a small community of members, in the range of 100-150 members, this system could be used to host their web application. Such a web application could contain the statistics about the players, recorded games, event announcements, and any other desirable features to chess players. Then a subset of the community, at the very least 10 members, could contribute their household computing resources to host their web application, and to serve their community without incurring any extra cost [1]. Two or three nodes could host the database, to provide fault-tolerance in terms of reliability, availability and consistency. Whereas one node could host the web server, and delegate two other nodes to take over in case of failures, exposing only 3 IP addresses. These nodes are not required to host a domain, and could simply connect directly using the IP addresses, but for convenience sake's lets assume that they register a domain name to one of these IP addresses. The remaining 3-4 nodes can be used as worker nodes, to respond to incoming requests from the web server. Again, this is a *semi*-trusted environment because the producers and consumers have a pre-existing trust relationship, but operates in a public environment. The only cost incurred by this chess club would be the domain registration, and it is optional.

[1] Given that their computers are usually turned on, if not then one should factor in the cost of electricity.

Consequently, we emphasizes that this computing platform caters to a specific niche of potential users, but it empowers them to recycle currently available resources and to re-factor these resources with a new purpose. All of this, while maintaining a focus on providing an intuitive and rational way to reason about how to port such applicationss, using event-driven programming model and the Python programming language.

7.2 Problems Encountered

We have encountered many problems while conducting this research effort, and we present the problems that remains open and require further research.

Peer-to-Peer Networking Primitives and Collaborative Systems

The first problem we have encountered was related to the peer-to-peer networking primitives and the collaborative systems, more precisely how well suited are the current networking primitives to be used in very-large scale collaborative systems.

One publication debated this question and the authors of this publication concluded that there are no peer-to-peer networking primitive, out of the 12 investigated, that caters to all the requirements of a collaborative system [**?**]. We have presented these requirements in this thesis in Section **??**. These requirements encompasses the lifecyle of a contributing resource in a collaborative system, and present an adequate starting point for the development of such systems. Out of the 12 peer-to-peer networking primitives investigated, there were 4 unstructured overlay network-based solutions and 8 structured overlay network-based solutions. All of these solutions were mutually exclusive with respect to their structure and functionalities. The authors are currently working on a solution that would account for all their requirements, which has yet to be published.

The major difficulty outlined, was the mutually exclusive characteristics of some features. Such as the ability to perform deterministic queries and to performMADQ, seems to be completely incompatible because they occur in different networking primitives exclusively. Similarly to support mutable object and provide the deterministic querying capabilities of structured primitives. Face with this problem, we've had to analyze our intentions and the position of our system. We decided to create an architecture focused on extensibility, to be able to supplant the underlying networking primitive at any desirable point in time, whether because a new and more efficient networking primitive has been devised or simply to port this system to a different environment.

Collaborative systems imposes several constraints as a consequence of the environment

in which these systems operate, but also because of the very collaborative nature of these system requiring trust between the participants. We think that this is why this problem is difficult, but also why it is interesting. This operating environment is paradoxical, because it requires the participants to trust other participants to just the right extent, for the it to still be beneficial to offload the workload to the participants. Consequently, novel peer-to-peer networking primitives are required for this specific class of distributing systems, if not to fulfill all the authors requirements but to cater to this semi-trusted computing environment, emerging out of peer-to-peer collaborative systems.

Collaborative web hosting

As we already mentioned in Section **??**, the current Internet infrastructure is not adequate for collaborative web hosting. Furthermore, the authors of this publication, [**?**], present an account of these deficiencies and propose a solution to mitigate them. They outline the fact that these deficiencies arise from two characteristics inherent to peer-to-peer systems. First, is the volatility of the resources and the dynamic nature of the networking infrastructure in peer-to-peer system, whereas the current Internet infrastructure is stable and *somewhat* static. The second characteristic emerges for the constant migration of the data among peers to ensure availability and consistency, meaning that one peer may have some data for some time, and then it goes offline, it no longer hosts data and it delegates (indirectly) to another peer.

The *naming* scheme employed by the Internet infrastructure is the first deficiency presented, it refers to the Domain Naming System (DNS). By definition DNS, operates by resolving an IP address from a URL, using a map of URLs and IP addresses. This is not adequate in the dynamic environment of peer-to-peer systems, because there are no guarantee that the IP address associated with a specific URL corresponds to a live node, possibly attempting to access this inaccessible resource.

The second deficiency presented relates to *searching and indexing* of resources. It is traditionally done using a centralized search engine that provides the service to locate any resource on the web. Whereas, in peer-to-peer systems indexing is (usually) distributed and done on a voluntary basis.

The last deficiency presented relates to the *content availability*. The current web infrastructure relies on web servers to be online continuously. Whereas in peer-to-peer systems, the nodes join and leave the system constantly. It is not easy to provide similar availability guarantees using peer-to-peer systems and it is even worst in fully decentralized systems. These guarantees are essential for the usability of a collaborative web hosting infrastructure.

This provides a glimpse into the limitations of the current web infrastructure with regards to collaborative web hosting, and again for a full and comprehensive account followed by a candidate solution please refer to [**?**].

Ultimately, if an application needs to be accessible from the web, it must commit to at least one static IP address for theDNS to include this application as a contactable resource on the web. This effectively induce a single point of failure into this web-based resource, namely its IP address, exposing the resource to potential Denial of Service types of attacks.

This showcases the immaturity of the Internet as a platform, and how the initial design decisions still transpires to this day by exposing the limitation of the design. This is not a critic of the Internet current infrastructure, just a mere comment on the fact that it initial purpose might have been outlived and a new infrastructure should be considered, at the very least from an academic perspective.

Ethical use of Computational Resources

The last problem we have encountered as a form of criticism of providing aggregates of computational resources openly and freely.

One of the underlying factor that drove this research effort, was to explore the possibility for the consumer to emancipate itself from current cloud service providers and rather, leverage resources from the Internet as a community. We were naively inspired by this utopic vision of a computing platform that promotes: ecological awareness by recycling rather than consuming new resources, censorship-resistance by resorting to decentralized topologies and community by sharing *private* resources.

Upon further investigation, and helpful comments from other researchers we were faced with the reality that this vision was in fact utopic, because of which we reposition ourselves to preserve our vision but in a more fitting context. We position our system to operate in semi-trusted or semi-private environment, which is more realistic and provides the liberty to enact our vision.

The question raised with this anecdote gravitate around the capacity for ethical use of freely available computational resources in an environment such as the Internet. This relates to the a field known as *cyberethics*, being highly debated in the early days of the Internet, some guidelines were devised such as [**?**] or [**?**]. The guidelines states that wasting computational resources, disrupting the functioning of the Internet, gaining unauthorized access to resources, or violating privacy of the users are all deemed unethical.

Now, a quarter-century of continuously presenting similar ethical guidelines to the users, it seems that the state of the *open* Internet is more unethical than ever. What gives?

This question is more apt as a philosophy of technology (or information) question, and its root cause as well as its potential solutions digresses too much for the scope of this thesis. Nonetheless, the implications and consequences of this question are a real impeding factor when researching *and* developing Internet related technologies. For more information on cyberethics, see [**?**] or [**?**].

Chapter 8

Conclusion

This thesis presented a distributed computing platform similar to the PaaS computing platform offered by cloud service providers, but composed of volunteered resources rather than dedicated resources. We have illustrated how to provide such a platform without introducing any additional resources, but by simply recycling resources commonly available. As a consequence the computing platform we have proposed is well suited for a variety of devices including lower-end computational resources.

8.0.1 Requirements Fulfilled

In the context of this research effort, we focus our attention to respond to two sets of requirements simultaneously, the *evaluation requirements* and the *functional requirements*. The former set of requirements originates from [**?**], and contain the essential phases for a contributing resource in a collaborative system. Where as the latter set of requirements, are functional requirements that we devised for our intended application of this system.

We consider our five *functional* requirements, enumerated in **??**, to be fulfilled as follows:

Requirement 1 Was completely fulfilled by the devising of the *Ring* abstraction by providing means to connect participants and manage the application deployment, effectively support multiple application being deployed simultaneously.

Requirement 2 Was respected since the API proposed for the *Application layer* does not introduce and/or force any third-party to provide any of the services it encompasses. As well as opting for light virtualization enforced the self-containment portion of this requirement, not only to the system-level but also to the application-level.

Requirement 3 The *security* component of the propose API covers very basic threat models common to all web-based application, including access control, authentication and secure communication channels. Also by choosing a DHT as the networking primitive for the *Ring* abstraction, we adopted a decentralized networking topology.

Requirement 4 By design no extra hardware is required, but rather encourage the recycling of the resources by permitting lower-end and legacy resources to participate. More specifically we reduced the overhead incurred by traditional (full) virtualization technologies by using light virtualization technologies, such as containers.

Requirement 5 Dynamic membership is designed to be supported by this application, using the autonomous PID controllers, but still requires implementation. Using these same controllers and the based on the fact that this system is built around asynchronous task queues, we then provide scalability to applications deployed.

We also consider having address the *evaluation requirements*, if not exhaustively, at least sufficiently for the proper functioning of this system as a *collaborative system*. We demonstrate this claim as follow:

- Providing a way to represent resources based on a set of desirable attributes, and means to **Advertise** and **Discover** these resources using the *Ring* abstraction.
- Using a two-step best-effort mechanism to **Select** the most relevant resources based on a consumer-defined query using a pub/sub messaging pattern, and then in the second step providing means to evaluate the inter-resource relationships, effectively addressing the concerns expressed in the **Match** phase.
- Using acknowledgments in the *Select* mechanisms, provides this system with the ability to **Bind** the resources to a tentative consumer, thereby mitigating any potential concurrent binding of the same resource to two different applications.
- Resources are **Use**d to execute tasks immediately after the initialization process is completed.
- **Release** of the resources, is performed autonomously by a Proportional-Integral-Derivative controller that continuously monitors the workload and performance of a type of node in an application, and release any superfluous resources.

8.0.2 Contributions

In this thesis we introduced four contributions, see **??**, which were novel to the best of our knowledge.

Fully-Decentralized Collaborative Web Computing Platform

This system presents a fully-decentralized collaborative Web platform to the extent that is possible within the current infrastructure of the Web. Because of the Web's infrastructure, it is not possible to associate a URL to non-static IP addresses or simply impractical, every Web-based application built for this system will have at most one single point of failure, the Web server.

The computing platform is flexible enough to cater the a similar variety of applications, akin to what current cloud service providers offer in terms of *paas*. We have demonstrated this by implementing two proofs of concept, and showing how intuitive it is to refactor applications for compliance with this architecture, using the event-driven programming model.

Candidate Minimal API Specification for Computing Platforms and/or PaaS

The API we have devised from an investigation of the three major cloud service providers: *Google*, *Amazon* and *Microsoft*, circumscribes the essential services and features provided in a computing platform. It is minimal, since it provides the basic building blocks for any web application. Given that a *specific* feature is missing, it is easily extensible to incorporate it into the API as long as it is included as a application-specific dependency in the container configuration file.

Using Light Virtualization to Abstract and Isolate Contributors

To the best of our knowledge, we are the first to propose the use of *light virtualization*, or containers, as a atomic unit composing the computing platform and also in the context of PaaS. Since the start of this thesis, 2013, the technological landscape shifted and now containers are gaining in popularity, especially with the support of major partners like Microsoft.

Minimally Intrusive System Inducing Small Memory Footprint

Our system is minimally intrusive, as shown in the use-case of *Multi-Document Text Summarization using a Genetic Algorithm* and consequently showing how intuitive and little effort is required to port an existing application to this computing platform. By resorting to *light virtualization* rather than full virtualzation, we have reduced the overhead induced by virtualization to a minimum, while maintaining the desirable isolation and abstractive properties of virtualization.

8.0.3 Future Work

Several *short-term* research issues were identified along the way, amidst some *long-term* research issues. We have identified five short-term research issues that pertain to the implementation of this system, and one long-term research issue pertaining to the environment of the system. We have presented some open problems that we have encountered while writing this thesis, see **??**, but are not specific to our system.

Short-term Research Issues

1. Implement *T-Man* overlay network gossip-based protocol to construct the overlay network topologies in the *Fellowship*, rather than connecting directly to each node.

2. Provide dynamic configuration support for nodes, enabling them to modify their configuration at run-time rather than through the modification of configuration files.

3. Implement PID controllers to provide dynamic membership capabilities, and test how well it operates under various workloads.

4. Investigate more elaborate access control schemes to evaluate how much change is required to adapt this system to an *enterprise* environment.

Long-term Research Issues

The primary long-term research issue that rises out of this academic inquiry, is how can we adapt (if possible) the current system to operate in a fully untrusted environment? Researching this issue will enlighten us on how to provide a truly public volunteer cloud computing infrastructure using a *decentralized* networking primitive. Multiple concerns arises in a fully untrusted environment, security being the most obvious one but more precisely how to prevent this system in such an environment to be used for malicious intents, such as raising a bot-net army? Thus security concerns are supplemented with ethical concerns in an untrusted environment, and is it possible to mitigate them using this design?