

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 *(degree)*
Graduate Department of *(department)*
University of Toronto

© Copyright *(year of graduation)* by Dany Wilson

Contents

1	Introduction	6
1.1	Motivations	6
1.2	Problem Definition	9
1.3	Contributions	9
1.4	Outline	10
2	Background	12
2.1	Collaborative Systems	12
2.1.1	SETI@Home	13
2.1.2	BOINC: Berkeley Open Infrastructure for Network Computing . .	14
2.2	Cloud Computing	15
2.2.1	Deployment Models	16
2.2.2	Service Models	17
2.2.3	Volunteer Cloud Computing	18
2.3	Peer-to-Peer Computing	19
2.3.1	Overview	19
2.3.2	Overlay Networks	20
2.3.3	Structured	21
2.3.4	Unstructured	22
2.3.5	Slicing	23
2.3.6	Comparison	25
2.4	Virtualization Technologies	26
2.4.1	Overview	26
2.4.2	Full Virtualization vs. Light Virtualization	27
2.4.3	Containers	29
2.4.4	LxC	29
2.4.5	Docker	30

3	Related Work	32
3.1	Evaluation Framework	32
3.2	Cloud@Home	34
3.2.1	Architecture	35
3.2.2	Evaluation	37
3.3	Peer-to-Peer Cloud System	39
3.3.1	Architecture	40
3.3.2	Evaluation	42
3.4	Discussion	45
4	Architecture	47
4.1	Overview	47
4.2	Network Layer	48
4.2.1	The Ring	49
4.2.2	The Fellowships	51
4.3	Virtual Layer	53
4.4	Application Layer	56
4.4.1	Overview	56
4.4.2	Databases and Storages	57
4.4.3	Communication and Networking	59
4.4.4	Load Balancing and Scalability	60
4.4.5	Security	63
4.4.6	Application Deployment and Management	65
4.5	Discussion	67
4.5.1	Collaborative Peer-to-Peer System Framework Implementation . .	67
4.5.2	Research Requirements	69
5	Implementation	71
5.1	Technology Used	71
5.2	Constructs	73
5.2.1	Introduction	73
5.2.2	Task	73
5.2.3	The Ring	74
5.2.4	The Fellowships	75
5.2.5	Conclusions	76
5.3	Workflows and Protocols	76
5.3.1	Initializing a Node	76

5.3.2	Fellowship's Protocol	77
5.3.3	Web Protocol	78
5.4	Proof of Concept: Calculator	79
5.4.1	Overview	79
5.4.2	Component Interaction	79
5.4.3	Application Deploying Node	81
5.4.4	Contributing Node	82
5.4.5	Putting it all together	83
5.5	Conclusion	83
6	Use-Case: Multi-Document Text Summarization using Genetic Algorithms	84
6.1	Introduction	84
6.2	Problem at Hand	85
6.2.1	Automatic Summarization	85
6.2.2	Genetic Algorithms	86
6.2.3	Extensions Proposed for MDTs	86
6.3	Translation of the Problem for this Architecture	88
6.4	Implementation Details	90
6.4.1	Application Deploying Node	90
6.4.2	Data Node	91
6.4.3	Worker Node	93
6.5	Conclusions	93
7	Discussion	94
7.1	Positioning of the System	94
7.2	Problems Encountered	96
7.3	Extensibility	96
7.4	Limitations of the System	96
8	Conclusions and Future Work	97

Chapter 1

Introduction *the*

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 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 presents the different chapters include 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) offers 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 exists 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 the data and application to their data centers, where ~~they will be responsible for the provisioning of the resources.~~

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?

~~Another interesting perspective of communities in this current day and age, is the educational system being divided into different school boards and by regions, of which they form micro-organizations.~~



One ~~common component~~ 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.

Aspect

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.

~~From these incentives, one may be tempted to raise the concern that continuously using these resources will limit their lifespan. This is referred to as *functional obsolescence*, in which a resource no longer function as it was designed to. Thus, one could argue against this by underlining the eminent *technological obsolescence* of these resources, where new technologies supersedes older technologies usually in a matter of months.~~

There is a ~~need~~ 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 ~~need to provide solution~~ 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.

~~There are no silver bullets, but being aware of the reality of the technological landscape is the starting point to formulate a candidate solution. The intersection of all these needs represent the focal point of this thesis.~~

See potential benefits in solutions

benefit

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 accomplished 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 micro-computers.



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 [BJ13] 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 laid on from a short-term and from a long-term perspective.

Chapter 2

Background

of this

In this chapter we present an overview of all the material required to understand the context ~~in which we present our~~ 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 characteristic of public collaborative systems. We conclude by presenting *SETI@Home*, which is an example of collaborative system, and *BOINC*, a middleware to create such system.

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 it 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. [ref]


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


they

former, since it exhibits the public nature of this type of computing explicitly.

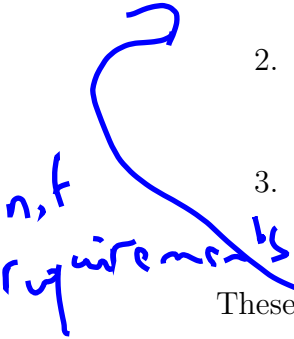


It is possible to draw further distinctions for collaborative systems, in the context of distributed systems, such as the deployment model. 

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, ~~thus 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 ~~requirements present the~~ differences ~~and~~  illustrate the challenges that arise from leveraging volunteered resources in a public environment, that are ~~simply~~  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 2.1.1.

An extensive body of work already exists on the subject, for more information on public collaborative systems in general see [And03] [AF06], and for more information on private collaborative systems, but more specifically on grid computing see [BFH03].

2.1.1 SETI@Home

SETI@Home [ACK⁺02] is a notable example of a public collaborative system, built using the BOINC system framework [And04].

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 2.1.

First, they ensure accuracy of the computations 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.

Second, the nature of the problem domain ease the mitigation of unreliable networking. Because it is possible to divide the work units into smaller sized data-pieces (350kb), and still provide a substantial computational workload for the volunteer. Thus, the downstream of data can be minimal, in relation to computational workload. But it is also the case for the upstream of data, since each work unit generates a small sized result (1kb).

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

This is one of the most successful project of this nature, in terms of aggregate computing power achieving upwards of 660 teraFLOPs [Wik15]. 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 [And04] 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.


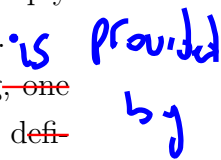
The simplicity of setting up and understanding the infrastructure provided by BOINC is the key to its success. Essentially every scientific enterprise resorting to public-resource computing have similar requirements. By creating a middleware that fulfills these requirements out of the box, it enables 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.  ~~Effectively reducing~~ 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.

~~Learning into~~  a more formal definition of the characteristics of cloud computing, ~~one could consult~~ the National Institute of Standards and Technology (U.S.A.) (NIST) ~~definition~~  [MG11]:

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

as

Cloud computing infrastructure can manifest itself ~~using~~ one of 4 different deployment models, ~~also characterized in~~ [MG11].

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).

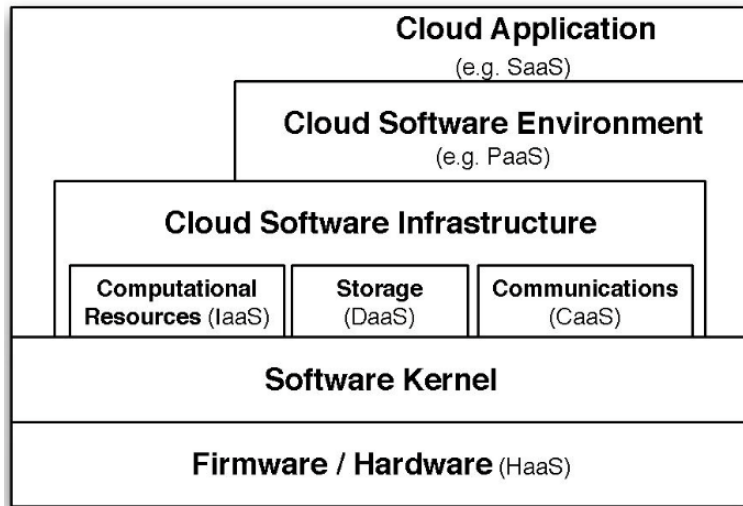
Another popular deployment model, is a *private* cloud, which is intended to be exclusively used by one entity or organization. They can maintain or not the cloud infrastructure, in which case they delegate to a 3rd-party CSP.

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 [YBDS08], which represents a categorization of the services offered by the CSP.



Figure

Figure 2.1: Ontological Representation of Cloud

[YBDS08]

In 2.1 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.

In 2.2, the separation of responsibilities for each of the service models is compared, and we will use it to define them.

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* [Inc15].

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 ~~for this platform~~. A popular platform for this service model, would be undoubtedly *Google App Engine* [Inc14a].

Separation of Responsibilities

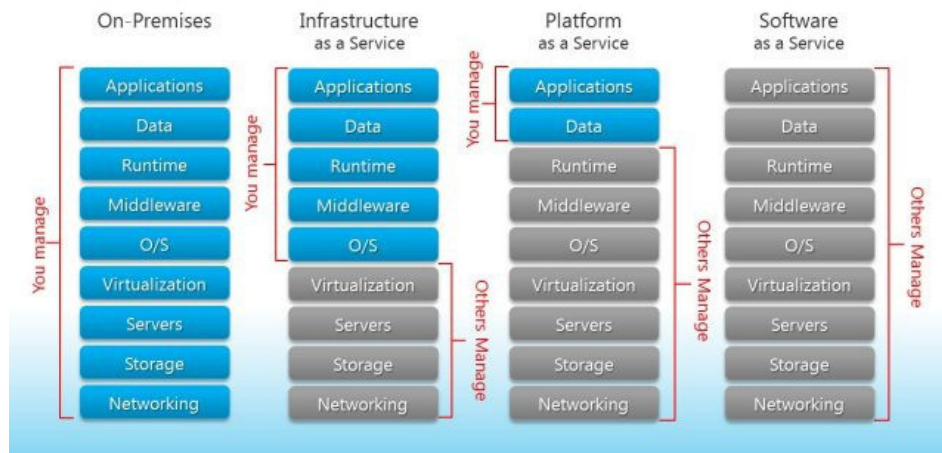


Figure 2.2: Cloud Services w.r.t. Responsibilities

Finally, SaaS is ~~the~~ service model that simply only ~~the~~ application and everything else is the responsibility of the service provider. Using this model the consumer ~~simply~~ uses ~~the~~ software through the CSP without owning a copy, ~~but rather~~ by leveraging its functionalities as services. An example of SaaS would be [sal15], which offers a multitude of various business applications as SaaS.

2.2.3 Volunteer Cloud Computing

Volunteer cloud computing is an emerging paradigm, ~~in the ever growing ontology of cloud computing.~~

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 by relying solely on (volunteered) public-resources. It is a fairly recent idea, as it is not even mention in earlier publications such as [RCL09], but it is in later publications such as [ZCB10].

This paradigm suffers from different challenges compared to traditional cloud computing [ASK15], 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 exist 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 [CDPS10b], targets the full spectrum of service model: IaaS, PaaS, and SaaS. Whereas others focus on only one IaaS [BMT12] [CW09]. Finally some offers a completely different approach, such as building a transparent infrastructure using peer-to-peer interception techniques [MGLPPJ13].

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. *Peer-to-peer*

~~These~~ systems are generally decentralized, ~~which~~ removes any single point of failure. Consequently, they are very resilient to the failure of any of the nodes composing its infrastructure. *This*


~~Also, they are~~ 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). *Peer-to-peer system are*


But out of all the characteristics that are inherent to these systems ⁰ and there are many more than what we have mentioned, is pertaining to the cost of operation. Since every participant in such a system usually assumes the cost of operating their own computers, the cost of operating the system, being relative to the number of participants, is amortized amongst them.

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


move up


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 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 system relies on a bootstrapping server to ease joining new nodes to the system.


~~The joining mechanism used in peer-to-peer system is usually referred to as *bootstrapping*.~~

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 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 needs to 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.  [MKL+02] [Bar01]

~~Extensive literature has been written on this topic, and for brevity purposes we refrain ourselves in the scope of our overview, for a good starting point on peer-to-peer computing see [MKL+02], and for a more complete account any of the books on it would suffice or see [Bar01].~~

2.3.2 Overlay Networks

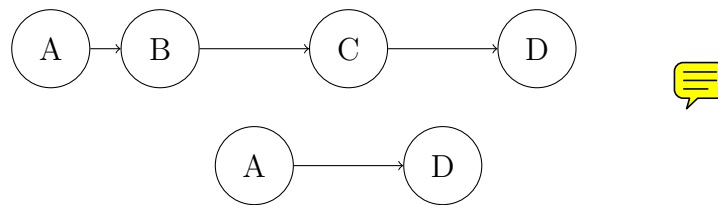
Overlay networks are fundamental to peer-to-peer systems and can be defined as ~~the superposition of two networks.~~  

~~Generally speaking, the network upon which the virtual network is being superposed is the physical network (e.g., Internet). The virtual network represents virtual connections~~

between the different nodes in the system.

~~In other words~~, 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:



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 structures [LCP⁺05]. In the next sub-sections we present the distinction between a *structured overlay network* and a *unstructured overlay network* respectively.

2.3.3 Structured

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

~~It uses an abstraction to organize the peers in a particular structure, this abstraction is known as the *keyspace*.~~ 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.

The way of partitioning the keyspace corresponds to the *keyspace partitioning scheme*, which dictates the structure of the resulting network. Some keyspace partitioning schemes can produce a ring topology, see [SMK⁺01]; whereas others can produce a tree-based topology, see [JOV05]

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. *→ structured, an*

A class of ~~this type of~~ overlay networks are Distributed Hash Table (DHT) *D* which consist of creating a overlay network by dividing the keys in ~~the~~ 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. *← S*

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* of the portion of the associative array *d* 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, ~~and for more information please see~~ [Sar10] [LCP⁺05] [BJ13].

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. *[Refs]*

2.3.4 Unstructured

Unstructured Overlay Networks differs *0* 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. *[Refs]* *↓*

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 [KC03] [BG09].

And self-stabilization property refers to the ability for a system to start in any arbitrary configuration and eventually converge to a desired configuration [BG09] [Dol00].

Whereas, the self-healing property provides autonomous detection, diagnosis and repair of localized problems [KC03], but also autonomous component-level failure recovery [BG09]. ~~For an extensive survey on all the characteristics inherent to this capability, refer to~~ [GSRU07]. Maintaining the overall health of the system, can be referred to as the *survivability*, which is the prime objective of self-healing components [PD11].

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

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.

~~Also~~ **B** 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 [RV11], 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 [LCP⁺05] and ~~see~~ [BJ13].

2.3.5 Slicing

Slicing is a primitive in distributed systems, which dictates the querying abilities of the underlying network structure. ~~In other words,~~ **T** 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 exist to solve this problem as depicted in [JK06] and [PMRS14]. Techniques vary in terms of the type and number of attributes considered, and consist of ordering the nodes according to these attributes providing different perspectives of the same collection of nodes.

Based on this 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 supports 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 attribute(s).

This problem is central to distinguishing between unstructured and structured overlay networks, since it circumscribes the flexibility of each network. As we will see in the following subsection, by means of comparative summary table.

✓
this refers to what ^!.

2.3.6 Comparison

We can summarize the advantages and disadvantages of the different types of overlay networks according to different characteristics, inspired by [LCP⁺05] 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 $O(\log n)$, 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 contact a live node and a portion of the keypace 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 difference between *full virtualization* and *light virtualization*. Then, we present what is *container-based virtualization*, and take a look at two different instances of it: ~~such as~~ Linux Containers (LxC) and Docker containers.

2.4.1 Overview

~~In computer science, virtualization has the mandate to~~ provide the ability to allocate the physical resources to accomplish a task prescribed by the software, ~~which is generally achieved through the decoupling of software from hardware~~ [Tav12].

The simplest example of virtualization that is used by a majority of OS are processes [Chi08].

~~They~~ 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.

processes

operating systems



The concept can be extended to ~~OSs and they can be virtualized~~. 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) [SPEW11].

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. [REFS]

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 ~~this excellent essay~~ [Tav12], ~~or consult the massive corpus of literature pertaining to the various implementations such as~~ [BDF⁺03] ~~for the Xen technology.~~

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 [BDF⁺03]. , offers



Using VM, rather than physical machines, ~~presents~~ several key features that are desirable in the context of distributed systems. One of these features is the ability to clone ~~them~~, 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* [Tav12].

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, whether or not there is an OS running the hypervisor (Type-2) or it is directly interfacing with the hardware (Type-1) [PG74].

VirtualBox [Wat08], is a prime example of Type-2 hypervisor, 2.3 shows how it interacts with the hardware through the OS.

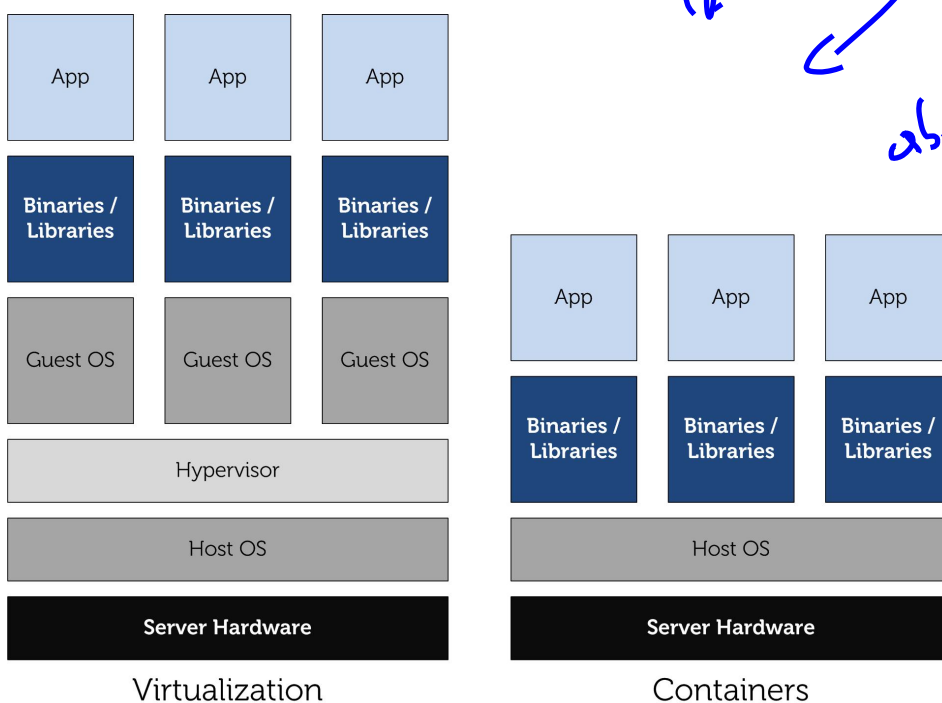


Figure 2.3: VM vs. Containers

~~In contrast~~, 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, they 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.

what do they refer to??

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 [DIRB14].

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.

The central technology behind containers in Linux is *c-groups*, or *control-groups*. Introduced as a kernel patch [Men11], it consists of an aggregation and partitioning mechanism for groups of tasks, that contains (within those aggregations/partitions) all their ~~future~~ children. It 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 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 ~~this 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 [Los15].

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 [EK07]. In other words two tasks belonging to different set of tasks, can have identical ID without incurring any ambiguity.

Resource isolation, provide 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 [DIRB14].

2.4.5 Docker

Docker was released on March 2013 and is ever-growing in popularity. It already has its own convention: DockerCon [Hyk14], which is endorsed by major technological companies such as IBM, Microsoft, Google, and RedHat (to name a few).

It 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 implement a custom version of these primitives, ~~which is~~ largely based on them. As does LxC, it 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. [REFS]

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 advantage 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.

~~This is a very brief overview of the Docker container technology, for more informations and details on the specific components, see the official documentation [Hyk15].~~

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 [BJ13], 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 [BJ13] aims at formalizing the requirements for collaborative systems using a peer-to-peer architecture, or in other words to address the peer-to-peer resource collaboration problem.

Where 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 one or more formal specification (Resource Specification) over a defined set of attributes.
2. **Discover:** Nodes may use 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.

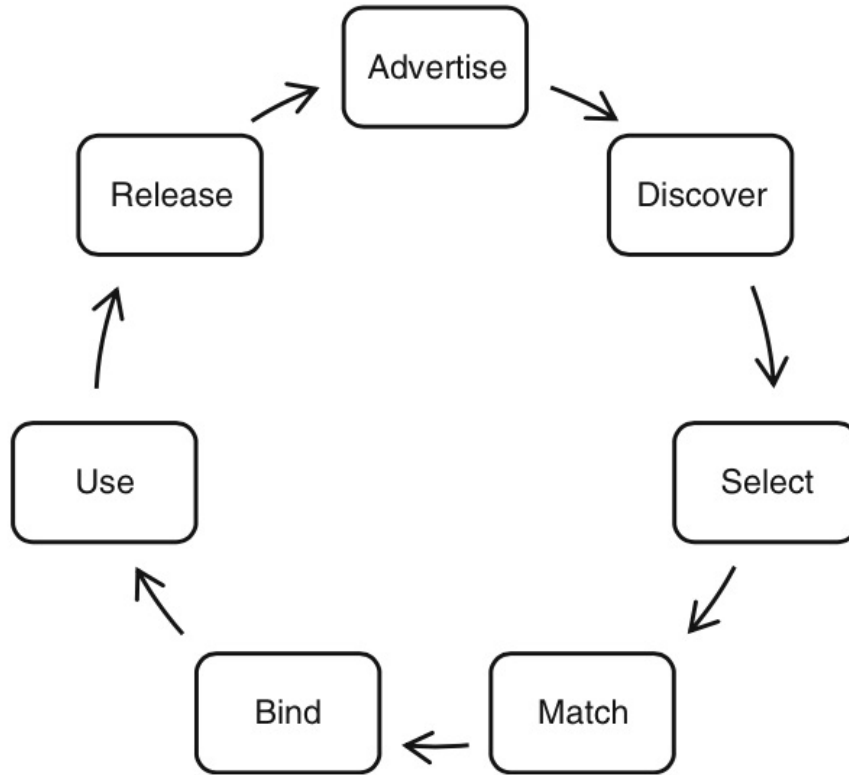


Figure 3.1: Framework for Peer-to-Peer Resource Collaboration Problem

3. **Select:** Provide mechanisms for an user to select a group(s) of resources, that satisfies a formal specification of his requirements, which contains attributes and ranges of acceptable values for these attributes.
4. **Match:** Provide mechanisms to be able to formally specify inter-resource relationship requirements, to ensure that they satisfy the resource and application constraints, in the query.
5. **Bind:** Provide a binding mechanism between the resources and the applications, to prevent that two application select 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 constraints of both application and resources.
7. **Release:** Provide a mechanism that allows to release resources in relation to the

application demand, and/or the contractual binding (if time-sensitive). That mechanism can also provide means to partially release a resource to enable it collaborate with other applications.

This framework covers bootstrapping, resource discovery, resource selection and resource compatibility, and usage. Thus, if one were to implement each one of these functional requirements into a separate module, then the system would be modular and it would ensure proper collaborative behavior. For more details on this framework and the evaluation that they conducted on different collaborative systems, see [BJ13] [BJ12].

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 from the combination of two existing paradigms: the *public-resource computing paradigm* and the *cloud computing paradigm*. This is one of the earliest attempt to create a *volunteer cloud computing infrastructure*, in which they have analyzed extensively the challenges relating to it over several publications, [CDPS09] [CDPS10c] [DFP11] [DP12] to cite a few.

Their vision is to offer a full-fledged cloud computing infrastructure that leverages the resources from its users, rather than incurring the construction of dedicated data center to serve the users. Full-fledged means that they want to provide the following service models: IaaS, PaaS and SaaS.

Their resulting system should offer performances and Quality of Service (QoS) similar to major CSP.

To fulfill some the performance requirements, they focused on *interoperability* between CSP, which provides a user with the ability to contract resources from commercial CSPs, given that the resources currently available are insufficient or inadequate.

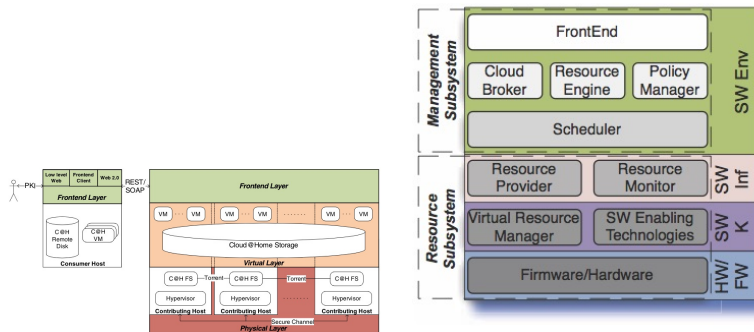
Whereas to fulfill the QoS requirements, they require that both parties negotiate the QoS that are contained in a Service-Level Agreement (SLA) created by the resource consumer. In other words, the consumer provide his/her intended terms of use and the provider decide whether or not he/she accepts them. In the case that the resource provider disagree with the SLA and the QoS it contains, the negotiation process is aborted. For more details pertaining to the QoS and SLA, see this publication centered around this requirement, [DPR⁺11].

Contrary to CSP, having to deal with volunteered resources, they were obligated to ponder on how to entice people to contribute their resources to this infrastructure. They devised a incentive-based business model, that offers financial restitution to the users for their contributions. This proposes to transform current 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 with respect to our requirements and the evaluation framework.

3.2.1 Architecture

They proposed a three-tiered architecture, composed of the *Frontend Layer*, *Virtual Layer* and *Physical Layer*; and a functional representation of the infrastructure, which is divided into the *Management Subsystem* and the *Resource Subsystem*.



Cloud@Home Architecture: Overview C

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 on the cloud.* The relation between the Consumer Host and this layer is based on a client-server relationship, and provides an centralized access point to the infrastructure.

The *Virtual Layer* addresses the problem of providing a homogeneous perspective of a set of heterogeneous resources, by means of virtualization. Through virtualization, it is possible to completely abstract the underlying hardware specificities of the different contributors and provide a mean to exploit them in a platform-agnostic fashion. This layer is further divided into two services: *Execution Service* and *Storage Service*. The former provides means to consumers to spin VMs, according to their needs, and thus they provide a similar service as with the Infrastructure-as-a-Service model. The contributors are simply assigned a arbitrary virtual machine to host (or multiple relative to their contributions). The latter provides distributed storage facility by implementing a

distributed file system such as [GGL03], and consumers can mount locally a remote disk that corresponds to a portion of this file system. As for contributors, they simply host an arbitrary portion of this file system relative to the amount of disk space they wish to contribute, thus this provides a unified view of the complete file system to the consumers.

The *Physical Layer* is responsible for connecting the contributing resources together and providing means for communication. Negotiation between the contributors and the consumers, by requesting resources, is also performed at this layer. Where consumer can specify their contributions, then upon request for resources the specification of the resources are compared to the request and conflict resolution is performed if necessary. They specify several mechanisms to ensure and negotiate the Quality of Service and Service-Level Agreements between consumers and contributors (or commercial providers).

The *Management Subsystem* translates the user's request into multiple sub-requests depending on whether the resource requested can be satisfied by volunteer resources or their commercial counterparts, this is left to the discretion of the user as a quality of service parameter. These request(s) are then passed on to the Resource Subsystem that binds and match the resources requested to the user from which the initial request originated. As stated in [CDPS10b], it is required to make the Management Subsystem a centralized subsystem in order to manage the infrastructure, to manage QoS and SLA, but also to provide dynamic provisioning. They claim that it is the only way to aggregate information from the infrastructure and provide a reliable perspective of it.

Finally, they proposed a middleware that implements the ideas from the architectural overview and the functional representation, which covers the different goals that Cloud@Home addresses. The following figure showcases the deployment of the Cloud@Home infrastructure.

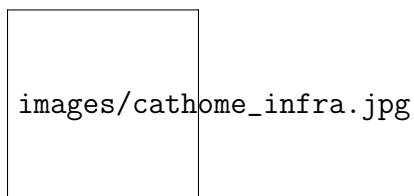


Figure 3.2: Cloud@Home Infrastructure

In this overview of the architecture, we have presented a simplification of the functionalities of the two subsystems for more details consult [CDPS10b]. For more details on the different layers composing its architecture consult [AAC⁺11], and finally, for more information on the middleware consult [CDPS10a].

3.2.2 Evaluation

In this subsection we evaluate the system according to the evaluation framework that we introduce in 3.1. Then we evaluate the system against the functional requirements that we have identified in 1.2.

Evaluation Framework

We use the evaluation framework to identify any functional deficiencies in this system relating to the essential requirements of collaborative systems. These functional requirements are not required to be mutually exclusive, and a subset of them can be combined into a module.

As a matter of fact, it is the case for the first three phases, which are accomplished by the Management Subsystem. The phases being *Advertise*, *Discover*, and *Select*, are all 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.

And it is also the case for the remaining phases like *Match*, *Bind*, *Use*, and *Release*, which are accomplished by the Resource Subsystem. It provides the ability, given a request, to match the resources which satisfies it, then to allocate them to the consumer in order to use them. Upon using the resources, it is possible release or deallocate them when the consumer no longer uses them. As a consequence, it provides dynamic membership of the resources, which permits for resources to be added or removed according to a 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 collaborative system.

Functional Requirements

We now evaluate how this system addresses our functional requirements, presented in ??, to enlighten 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 supporting multiple applications to be deployed simultaneously, it is completely fulfilled.

As for the second requirement, for which no third-party should be introduced for the service provisioning in order to maintain self-containment between the contributors and consumers, we can assume partial fulfillment. Because it is possible, using this system, to resort only to contributed resources and not contract resources from a 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 them (consumer/contributor).

Whereas the third requirement suffers a similar predicament with respect to its fulfillment. This requirement states that the system should be resistant to resources failing and should not introduce a single point of failure, for security purposes. This system focuses on creating a middleware that is lightweight, and is built around the concept of migrations. Upon failure of any resources, the system can swiftly migrate the workload to the any other available resources. Resulting in an infrastructure that is optimized for lightweight services and failure-resistant with respect to the resources. Unfortunately, this system is built around the concept that a central server is required to manage its infrastructure, as stated previously. Although some fault-tolerant mechanisms are put in place to mitigate this single point of failure, including redundant servers, and it seems to be reliable and available, further tests and benchmarks are required to corroborate their claims and to fulfill this requirement.

The following 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 that legacy equipment are able to form the infrastructure of this system. Since this system relies on full virtualization technology, it forces contributing peers to be able to run VMs on their machines in order to contribute a computational resource. Although, they outline that limited resources would be considered as consumers primarily, 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 as geo-location, accelerometers, barometers, and other various sensors that could be used to contribute some stream of information to any applications in the cloud. 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. Mainly because they are not the focus of any of the publications to date but are only mentioned as a (possible) feature, and thus this requirement can only be deemed partially fulfilled, if fulfilled at all. The fifth requirement is satisfied, through the use of thin clients and a simple Internet connection a user can consume or contribute resources to/from the cloud.

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 SLA and guarantees the QoS, and as a consequence it will preempt the resources necessary to ensure that the resource are only used to the extent prescribed in the SLA. Indirectly this system supports dynamic membership, and consequently provide scalability to the applications being deployed. Resulting in the fulfillment of this requirement entirely.

Out of the five requirements: one remains unfulfilled, two are partially fulfilled and two are completely fulfilled. 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 another more adequate solution or create one.

3.3 Peer-to-Peer Cloud System

The next system we present is the Peer-to-Peer Cloud System (P2PCS) , it was developed as part of a doctoral thesis [?], and it propose a slightly different approach than the previous example.

They propose that their infrastructure has no central server to manage the nodes, as is the case with Cloud@Home, rather each nodes interact directly with the other nodes for any operation performed be it joining the pool of available resources or joining an application.

Consequently, the consumer nodes interacts directly with the (potential) contributing nodes in order to select the fittest candidates to be part of the resources for the application. Fitness of a candidate is evaluated through the response of a specially defined query, which reflects the fulfillment of the desired criteria.

The query is defined by the consumer node (e.g., 10 nodes with at least 8 gigabytes of RAM...), and 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 2.3.5.

Then, 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 presents this system as a peer-to-peer cloud computing architecture, for which they focus only on providing a single service model, the IaaS. One of the defining

characteristic of this architecture is the fully decentralized approach adopted to connect the different peers, which is accomplished primarily through the use of epidemic and gossip-based protocols.

In the following subsections we will present the architecture of the system, and proceed to evaluate it 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.

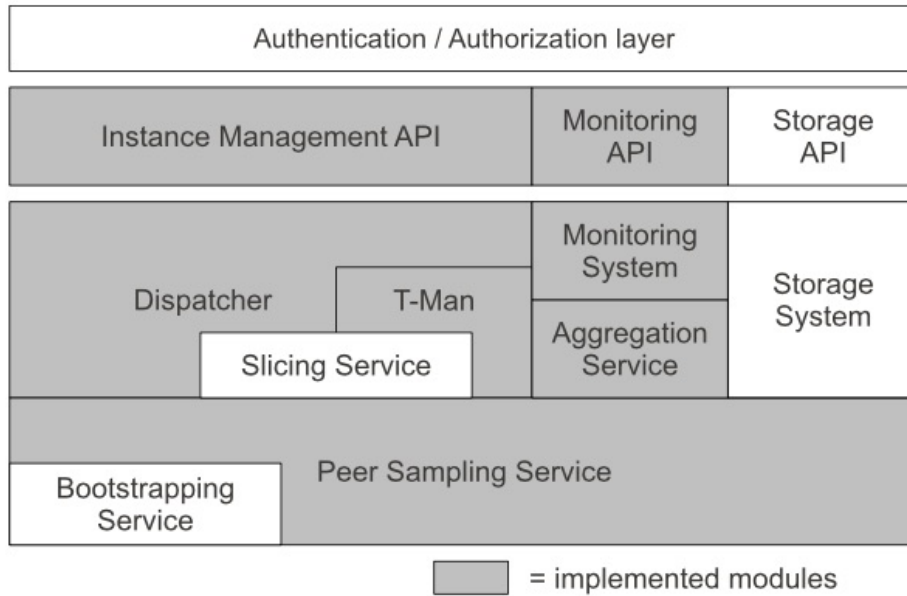


Figure 3.3: Peer-to-Peer Cloud System Architecture

In 3.3, the different components are presented and the authors distinguishes between what has been implemented (in gray) and what has been left as future work.

Adopting a top-down perspective, we analyze the first component. It represents the point of access to the system, or the *Authentication/Authorization layer* where a user (contributor/consumer) is required to identify themselves in order to access the system.

The second component represents the *API layer*, which is the principal access point for the consumers. It is composed of three sub-components, which provides different capabilities through various interfaces. The *Instance Management API* 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* component, provides a visual representation of the slice topology, and the *Storage API* provides the means for configuring (growing or shrinking) the distributed storage system for the current slice.

We conclude this presentation of the architecture with the third component, which represents the *Networking layer*.

This component is composed of three sub-components, which in turns are composed of three or less components.

The first sub-component is composed of two services, which are the *Bootstrapping Service* and the *Peer Sampling Service*. As its 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 from which a peer is able to exchange messages. 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 by means of a simple gossip-based protocol as presented in [JVG⁺07].

The second sub-component is composed of three components, which are the *Dispatcher*, the *T-Man* and the *Slicing Service*.

Starting with the *Dispatcher* component, which is responsible for the translating the commands issued through the various interfaces by the consumer into commands that are compliant with the gossip-based protocol(s) used.

The following component, is *T-Man*, a gossip-based protocol that provides the ability to create and manage structured overlay networks, based on various topologies [JMB09].

Whereas the last component (of this sub-component), the *Slicing Service*, provides the slicing capabilities for this system. This entails, as presented in 2.3.5, to order the nodes composing the system according to node-local attributes, then divide it 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 and consist of the number of nodes a consumer wishes to contract.

The last sub-component of the Networking layer is also composed of three components. These components are the *Monitoring System*, the *Aggregation Service* and the *Storage System*.

The *Monitoring System* provides to the consumer, through the corresponding API, the access to global system information collected and computed by the *Aggregation Service*.

This service provides system-wide parameters, in a decentralized fashion, 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 it resorts to using a *push-pull* gossip-based protocol, as presented in [JMB05].

The last component of this sub-component in the Networking layer is the *Storage System*. It consist of the system providing the distributed storage capabilities of this system, and, as stated previously, it is accessed through its corresponding API.

In 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 thrive 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 [Bir07]. For a more introductory approach to gossip-based and epidemic protocols in the context of distributed systems see [Jel].

3.3.2 Evaluation

In this subsection we will start by evaluating this system against the Evaluation Framework, and then against our requirements.

Evaluation Framework

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

The first two phases, *Advertise* and *Discover*, are coupled together within the Peer Sampling Service. But are not exhaustively implemented, by this we underline the fact that they provide no mechanism to advertise the resources that a node offers to the network explicitly. Either using formal means, as proposed in [BJ13], or any other means, rather this service only makes a node presence known to the other nodes via a simple gossip-protocol. We deem these functional requirements to be partially fulfilled.

The *Select* phase, is accomplished by the Slicing Service. They designed the system to use a gossip-based protocol to automatically partition the network into slices according to a metric as presented in [JK06], but was not implemented yet. Thus, we deem this functional requirement to be only partially fulfilled.

The following phase, referred to as *Match*, is suspected to be realized by the Slicing Service by a further specification of the query, but the limitation of the protocol used are not so clear. Since the slicing mechanism proposed by the authors is based around a utility function that describe the usefulness of a node with respect to the other nodes, based node-local attributes; and to introduce an additional dimension and perform multi-dimensional slicing using a utility function might not be the best approach since it introduces all sorts of statistical distortions as presented in [PMRS14]. Thus, we also deem this functional requirement to be partially fulfilled.

Bind, being the fifth phase, is performed by the T-Man gossip-based protocol. It is accomplished by creating a ring overlay of all the peers contained in a specific slice, 1 ring per slice, which results in the slices being mutually exclusive. There are no indications on how to cope with concurrent attempt at binding overlapping set of resources, given that these resources satisfies multiple selection queries. This is partly due to the fact of implementing the slicing mechanism using a single metric, namely the number of requested nodes, there is no eminent concurrent binding problem since it would only occur in a very specific case. It would manifest in the form of two or more concurrent requests for which the total requested number of nodes combined, would exceed the total number of available nodes, but the number of nodes per request would be inferior to the total available nodes. The problem then becomes, which application is entitled to have their request fulfilled? And since no a-priori ranking mechanism (or reputation mechanism) to decide which consumer should be favored, 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.

The sixth phase, referred to as the *Use* phase, is satisfied by the Dispatcher, which translates the higher-lever API requests into the appropriate low-level gossip protocol commands which are sent to the other nodes, and it results in the utilization of the resources. This functional requirement is completely fulfilled.

The last phase, *Release*, can be accomplished by the *Instance Management API*, which grants the ability to the user to control the instances it was assigned. The automation of this phase would be the result of the co-operation of the *Monitoring System* and the *Aggregation Service*, by collection global information about the system, such as the current load in a slice, and finally acting on these results by preempting the necessary amount of resources. This functional requirement is also completely fulfilled.

According to the evaluation framework, this architecture exhibits some of the characteristics that are essential in a collaborative system. Due to the fact that only two out of the seven functional requirements are completely fulfilled, whereas five are only partially

fulfilled either conceptually or because of the implementation. Nonetheless, this system presents a solid foundation that could be extended to meet all the requirements, since there are no apparent design decision that would prevent or hinder it.

Functional Requirements

We now evaluate this architecture against our functional requirements, defined earlier in 1.2.

The first requirement, states that it should be possible for the infrastructure to be constructed using only commodity hardware and that it should support 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 (no 3rd-parties), we can conclude that it is fulfilled. We arrive at this conclusion by observing that the proposed system relies solely on contributed resources and thus it does not introduce any 3rd-parties.

Whereas the next requirement states that system should not introduce any single point of failure, and that it should be (somewhat) resistant to ubiquitous failures. As a consequence of being fully decentralized this architecture does not introduce any single point of failure. And by using gossip-based protocols to create and maintain the overlays it is able to cope with a very dynamic networking environment. Failures or arrivals, 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 to enable a wider range of (weaker or lower-end) to participate. Nothing inherently forces a participant to acquire any special or dedicated equipment, thus the first portion of the requirement is satisfied. But by adopting the IaaS model, the memory footprint could become substantially large, as stated for the previous system, and it is not clear as how it would fare on lower-end resources. Thus we cannot deem this functional requirement to completely fulfilled, primarily because it uses full-virtualization technologies.

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 and thus this functional requirement is deemed to be fulfilled.

Ultimately, this system fulfills four out of the five functional requirement explicitly, which makes it closer to being a viable solution, but not quite. We put the emphasis on the fact that the system adopt enticing approach to solving our problem, but the limitations that are 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 configuring (properly) the various VMs required for any applications. Thus, we think that this system does not, and could not without substantial modifications, provide a PaaS computing platform.

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 very decent solution to the volunteer cloud computing paradigm but also with respect to the functional requirements of collaborative systems. This solution trades off full decentralization and the capacity to leverage very low-end resources for augmented performance and quality of service. It introduces possible third-parties in order to provide these kind of guarantees of service and thus it breaks our perspective of being fully self-contained with respect to the contributing peers. Finally, 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, abiding to the IaaS model. It provides the means to leverage these resources through full-virtualization and does not provide a business model to justify the viability of the infrastructure, or any incentive to entice possible contributors. Rather, like many open-source and community-driven project it anticipate a self-policing behavior from the community. The API proposed to interface with the different resources is analogous to traditional IaaS APIs, and 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 complexity and difficulties of operating such a system over a unreliable communication medium, such as the Internet.

Both of these projects offers interesting architectures supporting a multitude of desirable features for volunteer cloud computing and/or public-resource computing, but

from our perspective some key features are still missing. Notably the ability to leverage a collection of commodity hardware (by limiting its memory footprint) to provide a multi-application computing platform akin to a PaaS model. Concerns change when focusing on this service model, rather than the IaaS model, such as security and isolation which is accomplished traditionally by using VMs, to isolate the contributing environment from the hosting environment. It is not clear how this could be accomplished in either projects without resorting to Virtual Machines, and thus how to provide a PaaS using the contributed resources, rather than leveraging a commercial service provider, remains unresolved.

A complimentary discussion on the positioning of this research effort is provided in ???. Ultimately, due to the deficiencies, which are out of the scope of these projects and thus weren't expected to be addressed, 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 our architecture and illustrate the different decisions that influenced our design. We based this architecture on the observations made on the related work, and our design take into account our requirements as well as the essential functional requirements for collaborative systems [BJ13].

The first section provides a high-level perspective of the architecture, where as the subsequent sections delves into each layer, presenting and expressing the design rational behind them.

We then conclude with an assessment of the requirements that were addressed, with respect to the evaluation framework we have used previously to evaluate the other systems, but also according to our list.

4.1 Overview

We used a similar multi-tiered approach for the architecture to Cloud@Home and P2PCS, because it provides high modularity and loose coupling of the concerns of each tier.

In this architecture we differentiate between 3 types of nodes, namely a *worker node*, a *data node* and an *application deployer node*. A worker node consist of a computational resource that can execute tasks encompassing *all* the computational functionalities.

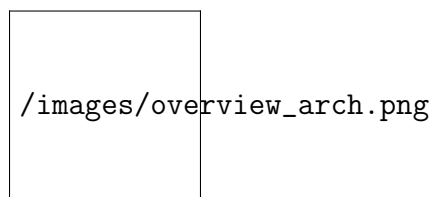
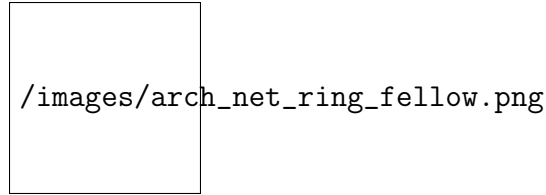


Figure 4.1: Overview of the Proposed Architecture.



Whereas a data node consist of storage resource that can performs all the database and storage related tasks. The application deployer is the node that contracts contributing nodes to host an application using this system.

Starting at the lowest-level, we have the **Network Layer**, which is responsible for handling all the network responsibilities, such as creation, maintenance, communication, and providing the high-level layer with an abstraction of the connections between the components.

On top of the Network Layer, we built the **Virtual Layer**, which is responsible for abstracting the physical characteristics of each node and provide a homogeneous interface to a collection of heterogeneous devices via virtualization mechanisms.

Finally, the top-most layer is the **Application Layer** exposing an Application Programmable Interface to the consuming node, presenting a minimal specification of the services required to transform the resources into a complete computing platform, similar to PaaS platforms.

4.2 Network Layer

In this section we present the *Network Layer* and the rational from which it originates, and as stated in the previous section 4.1, it is responsible for providing all the networking capabilities for this system. It accomplishes this using two abstractions, *the Ring* and *the Fellowship*, which we created to divide the networking concerns into two distinct characterization. We present the two abstractions and explain their relevance in this architecture.

When devising a distributed system, one is confronted with fundamental problems such as: *How can we connect a collection of computers, via the Internet, and maintain connectivity among them?*

From what we have presented in 2.3.2, we can safely assume that we can address this problem by creating an overlay network to connect this collection of computers. But then the question becomes, which topology is best suited for our requirements, or is it not relevant in this context?

A case-study presented in [BJ13], outlines the difference between the various topolo-

gies a overlay network can have in the context of a collaborative peer-to-peer system. They conclude that both, structured and unstructured, class of topologies of overlay networks exhibits different characteristics which can be desirable for collaborative systems, but none provides a comprehensive solution with respect to the functional requirements necessary for collaborative system presented in this same publication. And thus, we know that a overlay network will not cater all of the requirements, but 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, where each nodes contact this entity to be part of the network, and it also maintains connectivity in the network. The latter focuses heavily on epidemic and gossip-based protocols to generate the overlay networks and to maintain them. It adopts a slightly different strategy where each application generates and maintain their own overlay (slices) and thus management is not centralized, but rather it becomes federated where each slice is responsible for its nodes.

In order to respect **Requirement 3**, in our list of requirements 1.2, we must not opt for a centralized solution because it will introduce a single point of failure in our architecture. We must then opt for a suitable overlay network topology, and our topology of choice is structured. We chose a structured topology for our first abstraction, the *Ring*, since it provides the characteristics and features that are desirable for its intended purpose. We open the possibility for using any topology for our second abstraction, the *Fellowships*.

In the following subsections we will present these abstractions, and we present the design decisions that were made (including the choice of topology).

4.2.1 The Ring

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

The necessity of devising an abstraction to encapsulate the *public* environment in which our architecture will operate, is the result of trying to provide a construct that exhibits some of the desirable characteristics in a collaborative system. We have identified four distinct characteristics and/or functionalities that we deemed desirable for the resulting construct, which are:

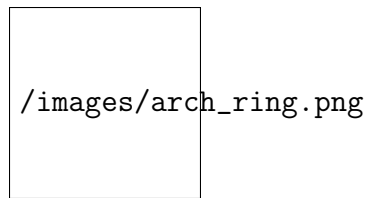


Figure 4.2: Abstract Representation The Ring.

1. Provide means to connect any participants together in a public environment, such as the Internet, resulting in a *public meeting point*.
2. None of the management responsibility should 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 or data that might be required by a(ny) participant to join an application should be contained in this construct, and its consistency should be ensured to prevent malicious participants from corrupting this information and paralyzing the system.

In order to provide the first requirement, it should be clear that an overlay network would suffice, and both topologies would provide this connectivity requirement. We can use the same rational for the following characteristic, since none of the topologies imposes a centralized management entity.

But in order to fulfill the third requirement, it is necessary to opt for a *structured* topology, rather than its counter-part. Because, as we have presented in the section on overlay networks 2.3.2, unstructured overlay networks do not provide a deterministic querying mechanism. Furthermore, the last requirement requires this construct to be fully self-contained and thus must possess some storage capabilities.

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 means to locate any node, deterministically, in a time complexity of $O(\log n)$ [MM02].

Using this DHT for storage of information is intuitive, but our application of this technology into the context of this architecture imposes further restrictions. Namely with respect to data and/or information consistency in a distributed context. We need to

ensure that malicious nodes cannot compromise the system by polluting or corrupting the information stored in the DHT, which corresponds to a *storage attack*, see [UPS11] for an extensive survey of the security techniques applicable to DHTs. Essentially, this problem is inherent to all the DHT, because by design it assumes a cooperative environment, where each node are benevolent, and most implementations do not address this directly, rather they leave it to the application developer.

Thus we propose the following change in the implementation, which will ensure consistency and correctness of the information stored in the DHT to an acceptable degree. By restraining the writing access of the participants to only those that are *application deployers*¹, we mitigate most of the possible storage attacks.

Then, the problem becomes: how can we ensure the benevolence of this type of participants? It is not possible to absolutely ensure their benevolence, but it is possible to limit the potential consequences of a storage attack by some malicious writer. By restraining their ability to write information to the DHT, we limit the application deployers capabilities to write values for only two keys.

One of those keys is named, *template key*, which corresponds to a public repository for the attributes that a node can use advertise its resources, be it dynamic attributes such as CPU usage, or static attributes such as total amount of memory.

Whereas the other key refers to the application that is deployed by a participant, and is only writable by this participant but readable by all. It contains the list of nodes composing the *Fellowship*, providing means of reconnecting 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 layer².

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 accentuate the modularity of this architecture, but primarily to have a clear separation between the nodes that are contributing and the

¹Further information on the types of participants and the core constructs of this architecture is provided in 5

²Again more information about the interfaces to the different layers in the 5

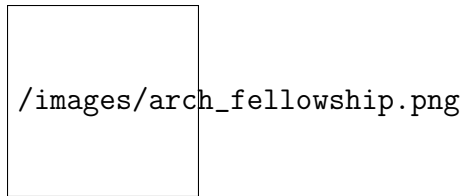


Figure 4.3: Fellowship Abstraction.

nodes that wishes to contribute by segregating them into different networks. This confers the private characteristics to this abstraction. For our architecture, we need to ensure that essential characteristics or capabilities were accounted for by this abstraction, such as:

1. The ability to discriminate between contracted nodes and malicious nodes (pretending to be a contracted contributing node).
2. To allow for nodes to join and leave in a graceful fashion, and consequently remove any single point of failure.
3. Providing means for secure communication among the nodes contributing to the same application.

The requirements of the *Fellowships* are more relaxed compared to those of the *Ring*, because they consists of private collection of nodes interacting together and are not required to interface with the public space directly (as would the *Ring* when new nodes are arriving).

Our primary focus, with respect to this abstraction, is the isolation of these networks by controlling which participant can join and which can't, as expressed in requirement one. We can ensure this using a *whitelisting mechanism*, where the application deployer will create a list of candidate contributors and only contract them for 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) knows 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, which can act as an authority. Although, we could implement a reputation system that would gradually increase the trust component for each participants, we leave this as future work and we discuss this in more details in the ??.

For the second requirement, we are inclined to implement a similar strategy as Peer-to-Peer Cloud System did with the slices. 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 an applications networking requirements a topology might present more advantages over another.

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 a random walk through the network, thereby providing the ability to perform complex queries. Whereas a structured network would require multiple simple queries, and then post-processing the results in order to aggregate these simple queries, from which we could achieve the same result less efficiently.

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

Finally, this abstraction provides the ability to control which participants are allowed to participate in this collaborative application, but also the ability to secure the communication channels by means of public-key cryptography. Depending on the networking requirements of each application, it is possible to tailor the networking topology to fulfill these requirements.

4.3 Virtual Layer

In this section we present the *Virtual Layer*, by defining its purpose and the implications on our 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, concerning the virtualization technologies used by the two projects evaluated in the previous chapter.

There are three essential features that this layer should provide. We now present them and discuss how we intend on providing them.

The first and foremost responsibility of this layer is to isolate a dedicated execution environment in contributing host system, as a security mechanism. 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 characteristic of this layer is its abstractive qualities that it bestow upon resources. By means of virtualization it is possible to abstract away the specificities of the underlying physical resource, and present all the resource in 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 where we distinguish them solely based on capacity-based metrics rather than proprietary capabilities.

The last required feature in our architecture from this layer, is the ability to control 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 1.2.

Light virtualization technologies, are a potential candidate solution to this design question, let's examine to which extent do they differentiate themselves from full virtualization technologies.

As prescribed in the first required characteristic, it provides the isolation needed to securely execute applications without interfering with the host executing environment. Due to their sandboxing properties, the executions done within a container are opaque to the host operating system, as would be the case with VMs or traditional full virtualization technologies. Thus both technologies, light and full virtualization, fulfills this requirement completely.

Same goes for the second characteristic, which is also provided by both types of virtualization technologies. Since, both provides homogeneous abstractions for the physically heterogeneous computing resources which result in that desired agnostic way of perceiving the actual resources. Still both technologies are equally desirable in the context of the first two requirements.

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 2.4.5, offers this functionality through their

image system. Thus, users hosting containers are capable using the c-group technology, as presented in 2.4.3, to restrict the processes spawned within a container to a specific subset of the available resources. As for full virtualization technologies it is also possible to restrict the amount of resources available to the VM, through various configuration parameters. But a difference ultimately persist between the two types of virtualization technologies.

As we've presented in 2.4, by using lightweight virtualization we can leverage the libraries and the kernel of the host operating system, which reduce the possible redundancy of the libraries and binaries to a minimum. Whereas, in a traditional VM, or 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. Since by design the overhead incurred by full virtualization, with the notion of hypervisor residing on top of the OS and 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 concept of hypervisor, and uses isolation (of namespace and resources) to access the resources directly.

Thus 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.

This decision is justified only because of our specialized requirements, and we can take an interlude to examine how the previous projects (Cloud@Home and P2PCS) addressed their virtualization requirements to fully understand its dependency to the requirements.

If we recall what both projects proposed, we can observe that both advocated the use of full virtualization technologies. Which does provide the isolation properties, as well as possessing the abstractive qualities necessary for their architecture. And in the context of IaaS, it is mandatory to resort to VMs in order to provide the ability to encapsulate an entire OS, 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 ???. This conflation of multiple concerns ultimately hinders productivity and usability of the computing platform.

We believe that when constructing an application using a computing platform, such as a PaaS, this platform is meant to abstract away the characteristics of the underlying physical resources, and enable to write an application using the defined interfaces independently of the underlying implementation. Whereas full virtualization provides exactly the opposite experience, by exposing all the underlying resources, and forcing the user to decide which implementation to use and how 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 an 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 any distributed computing platform, and also for PaaS.

Initially we present the reason why we need such a layer, and then proceed to present each component of composing this API, which are *Databases and Storages*, *Communication and Networking*, *Load Balancing and Scalability*, *Security* and *Application Deployment and Management*.

4.4.1 Overview

The purpose of this layer is to provide to the application developer the necessary building blocks to construct a distributed web application that is scalable. We propose a minimal API specification that provides all the building blocks necessary to developing scalable distributed web applications. It is minimal in the sense that it is sufficient to construct any applications, but more features could be added to ease the construction 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. Adopting an API to construct applications provides greater modularity with respect to the underlying system, since these applications will always be compatible with this system even if several updates occur, as long as it respects the interface it defined initially.

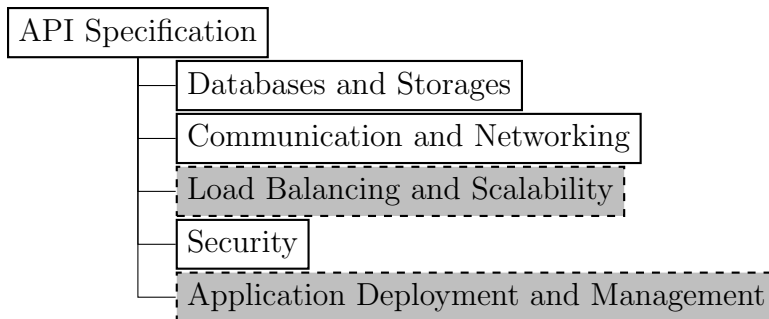


Figure 4.4: API Specification Overview

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*, then finding the overlapping components, and eliminating the superfluous or redundant components. Here is a pictorial representation of the resulting proposed API:

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. This occurs because the division of concerns was not absolute, and we chose to provide access to these components to the application developer through configuration parameters rather than building blocks.

In the following subsections we discuss in further details each of the five components identified, and provide an overview of what functionality they contribute to this architecture and why we included them.

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 [Inc14a] [Inc15] [Inc14b].

4.4.2 Databases and Storages

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

Out of our investigation, we could circumscribe the following taxonomy for the data persistence features and services, which resulted in the *Databases and Storage* component:

- **Relational Databases:** 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, which are stored 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. Thus they are a multiple solutions available, one of which is the distributed file system from Google, the *Google File System* [GGL03] As a matter of fact, this is exactly what Cloud@Home proposed to implement to offer distributed file system capabilities within their infrastructure.

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 and 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 that provides a plethora of different features crossing the boundaries between relational database, non-relational databases and even storage type solutions. We explore these a bit further when discussing the actual implementation in 5. Distributed databases also provides the ability to adjust the availability of the data in response to fluctuations in

request by adding more instances to the database cluster. They are devised around two important concepts: *replication* and *fragmentation*. The former represent the ability to replicate the data across several instances, to ensure availability and consistency of the data. Whereas the latter represents the ability to decomposed the relations into several sub-relations and then distributing them across several nodes. After which it is possible to reconstruct the original relations from these sub-relations (fragments), and this is useful to balance the workload across the nodes. 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 [Lin76] or [DP80], or even [ÖV11].

But rather than committing our architecture to a single distributed database implementation, or even to an implementation for each of these three services (RDBMS, No-SQL, and storage solutions) we opted for an extensible design. We provide an extensible abstraction of the interface to these database and storage systems, and thus if the solution provided doesn't suit their need it is possible to simply extend the interface to account for any other functionalities. We discuss this interface in greater details in 5.

The last service comprised in this component, is *caching*, and it is deemed useful in many distributed web application. 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.

An example of such a distributed caching mechanism is *Memcached*, which allows to logically combine unused memory to form a bigger unified cache [Fit11]. It is open-source, and we could provide it as a service without much problems.

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

4.4.3 Communication and Networking

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

But before going any further 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 problems 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 [AB14] 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. Then we can provide a service to programmatically present the information of the web application, by supporting a web framework.

The last responsibility attributed to this component is to provide a communication interface between the application and the end-user. We provide such interface 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 favors communication over HTTP as the underlying protocol. For a more detailed portrayal of the guidelines and design patterns present in REST, see [RR08].

Ultimately, we provide the services for the end-user to communicate with the web application using HTTP requests, and to access the underlying RESTful APIs to interact with the web application. Receiving these requests is a web server, which is accessible using the current Internet infrastructure, that provides 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 the 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 attempt at 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 fluctuation in the workload. The workload could diminish to a point that most resources are idling, then by enforcing

scalability it would diminish some of the resources in order to maximize the utilization of the remaining resources, and vice-versa. Which can be achieved using various domain specific algorithms to preempt the required resources.

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

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 5. Because it digresses too much from the service offered in the context of the application layer.

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 typical workflow commonly used with task queues is:

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

As stated above, these task queues are central to this architecture and this is why we do not provide them as a service in the application layer since it would be redundant. When explicitly necessary one could extend this architecture to incorporate a 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 architecture, especially in the context of a minimal specification for an API.

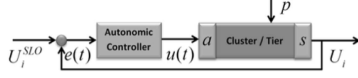


Figure 4.5: Proportional-Integral-Derivative (PID) Controller [GSL14]

The other aspect of this component is scalability, we can rationalize it as being the ability to respond to the fluctuation in demand, either manually or autonomously.

The PaaS providers that we have investigated, offers the ability to the user to specify static or dynamic policies. These policies usually are expressed using the SLA , and ensure that the different Service-Level Objective (SLO) are respected by requesting or returning resources accordingly. SLA are simply contracts that specifies the terms of a service between the consumer and the producer, whereas the SLO are metrics to measure the service provisioning performance of a provider and thus prevents any misunderstanding between both parties. There are many characteristics to defining SLOs and forming proper SLA, we focus on a more pragmatic approach to provide scalability and thus we refer the reader to this publication [KL03] and this one on service level management [SMJ00].

In our case we provide scalability by implementing a decentralized autonomic controller for each type of node³, based on [GSL14]. 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.

The controller used is Proportional-Integral-Derivative (PID) , and 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, and adjusts 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

³see ??

the observed performance, and attempts 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 specify the desired utilization level for each type of nodes, and this autonomic controller will adjust the resources to respond to the fluctuations in the workload automatically.

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. We deem this component to be application-independent and thus do not provide it as a collection of services but rather as intrinsic features of our architecture.

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. Other major concerns are access control, which includes authentication of the end-user and providing multi-tenancy. We present each of these concerns, and then explain how they are accounted for in this API.

The first concern is to provide means for secure communication between the nodes, and can be ensured using standardized technologies such as communicating over TCP/IP using Secure Sockets Layer (SSL) .

Since SSL certificates are usually negotiated by a web server and a client, then in the context of this architecture the web server is (usually) hosted on the application deployer and the client is a candidate node. Thus, a candidate node would negotiate its certificate with the application deployer, to establish a SSL communication channel between them. Upon successful negotiations the node and the application deployer are able to communicate in secure manner using the SSL protocol.

Similarly concerning the communications between the end-user and the front-end node can also be secured by using standardized protocols, by communicating using HyperText Transfer Protocol Secure (HTTPS) , which consists of using SSL in concert with HTTP.

Whereas in this case, the negotiation for SSL certificates would be done between the web hosting node and the end-user.

We use the same technologies (SSL, HTTPS, etc.) that are well established, as any other web application would when required to communicate over unsecured networks,

such as the Internet.

Access control is another very important security concern, it controls access to resources based on various schemes, and encompasses authorization regulation mechanisms and authentication mechanisms notably.

We can identify two basic types of accesses in our architecture: *access from node to node* and accesses from a end-user to the web application. Notice that the security concerns occurs at the same end-points in architecture.

The first type of access can best be illustrated by this example:

If we introduce a Byzantine node (that is a node for which the requests are incorrect or responds to requests incorrectly, either by error in computation 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 very simplistic strategy to access control, namely only the application deployer is allowed to perform administrative tasks. This is also known as Mandatory Access Control (MAC) , and the verification is done using a simple challenge-response between the nodes to assess the identity.

The second access type, refers to authentication of the end-users of the application. Authentication, as a service for applications, *can* be provided through Single-Sign-On (SSO) technologies, more precisely by incorporating hooks to the Google SSO API. By providing these hooks, the application deployer will be able to offload the authentication responsibilities to a trusted third-party, upon authentication the users will interact with a *OAuth* token, also known as a *bearer token*. By presenting such token when interacting with the application, the user will be able to maintain a session. For more information on Google's SSO API see the documentation, [Dev15].

Authentication is usually application dependent, because various applications handles authentication and authorization in various ways. 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, which is multi-tenancy. It consists of providing parallel multi-user support to an application without providing

each user with a dedicated instance of the application. Commonly provided in web applications by maintaining a *session* per active user, we propose to do exactly this. By doing this, seasoned web developers will find using our architecture very intuitive, rather than forcing sophisticated design patterns on them which would result in a steeper learning curve and consequently diminish the ease of use of this architecture.

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 provide 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 through a SSO service from a trusted third-party or maintained by the web server.

As future work, we could investigate more elaborate access control scheme for our architecture to support enterprise software development, but this would be a feature and not essential to the vision and mandate of this architecture. Similarly with multi-tenancy support.

4.4.6 Application Deployment and Management

This component encompasses all that pertain 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 services 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 *would* entail that a contributing node trusts the application deploying node. Multiple sophisticated scheme were presented to reduce the trust-level assumed between parties, with regard to source-code distribution, in a distributed context.

One example, [DKMyC04], consists of creating *bundles*, or atomic units, of code and data, which can then be distributed and executed. This deployment framework would provide authentication and authorization mechanisms, as well as certifying capabilities would be conferred to a specific server; to provide a trust environment where source code can be distributed and executed securely. An undesirable consequence of this is that it forces a completed infrastructure into the architecture to provide these guarantees, which leaks into the development process. Applications are required to be designed respecting a set of guidelines, which impedes the flexibility of the architecture.

Instead we propose a novel pragmatic way of providing similar guarantees, that ultimately relies on the users judgment. Since we are providing an open-platform, we apply the same principles of open-source code 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, by presenting the source-code in a text editor or browser, of its content.
- 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.

As far as *application configuration* is concerned, it is done through configuration files, and optionally using a Graphical User Interface (GUI) . The configurable parameters includes: minimum number of nodes, performance policies, type of environment deployed, communication primitives and any security primitives. Then, as a participant becomes a node in the network, before the creation of the node instance, the configuration file is read and interpreted. This configuration mechanism is easily extensible and could compensate for any other requirements.

Whereas *monitoring*, an important feedback mechanism for the application deployer, is primarily used to anticipate possible bottlenecks or upcoming failures. 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 in order to be presented as system-wide monitoring information to the application deployer. With respect to the load-balancing and scalability mechanism presented earlier, this mechanism provides a sufficient perspective of the application state.

Finally, we provide a flexible workflow for code distribution adopting a white-box approach. We also provide means to statically configure an application through various parameters. We provide monitoring capabilities via a heartbeat mechanism that generates system-wide information using local information about the system.

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, by means of 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, and of this thesis as presented in 1.2.

4.5.1 Collaborative Peer-to-Peer System Framework Implementation

In 3.1, we presented a framework that illustrates the essential functionalities required for a collaborative peer-to-peer system to scale and mitigate the complexities inherent to these type of systems. This subsection aims at presenting the different mechanisms implemented in this architecture to provide such functionalities.

If we recall, the framework was composed of 7 key phases of 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, creates a *resource specification* using a common template which can be represented as follows:

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

Note that the set of *Static Attributes* or *Dynamic Attributes* are extensible to accommodate any desirable attributes, and starts empty. Each application deployer then 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*).

Then, upon creation of their resource specification, the nodes will gather the list of application deployers and periodically send candidacy messages consisting of their resource specification. We say best-effort, since they simply repeat sending until an application deployer contracts them for contribution.

Other solutions, as outlined in [?], proposes to publish resource specifications to a repository where it is possible for the consumer to query the repository for the most relevant resources. Those solutions, are adequate if the repository is protected against possible storage attacks, and as the authors of [UPS11] have shown this is not the case for DHTs. Thus, in our context, since we are using a DHT to connect the nodes together and that it would violate **Requirement 3** to introduce a centralized managed repository, which states that this system should not introduce a single point of failure.

The following phase in the framework, is the **Select** phase and it relates to the selection mechanism offered to possible consumers, with which they are able to query the resource specifications. In order for an application deployer to *select* any node, it simply opens up a TCP/IP server connection on a specific port, and evaluates the upcoming candidates individually by examining their resource specifications that they publish. 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 the resource specifications, which represents a *topic*. The nodes uses the concept of topics to publish the various attributes that compose the resource specification. More specifically, application deployers do a tentative selection, by notifying the resources that it is considering them as a candidate resource. Only after all the required resources are *tentatively selected*, are they actually selected. This is considered 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 next phase is **Match**, and it encompasses the ability to formally specify the inter-resource relationship requirements and enforce them on the selection of resources. As a consequence of adopting this best-effort mechanism, it is possible to define these relationship as part of the selection policy. For example, once a group of *tentatively selected* resources is formed it would *then* be possible to enforce these relationship requirements, and repeat the process until the *tentative selection* fulfill all the requirements.

We provide a best-effort binding mechanism to account for the **Bind** phase. Best-effort, in the sense that the priority for an application to contract a specific node is determine with respect to the response to an emitted the advertisement. In other words

applications which response are received first are given priority over applications that received later advertisements.

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

Release phase, is concern with providing the capability to release a resource after contracting it for contribution, either because it 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.

Further work can be done to provide more extensive functionalities, but will be left as future work, since the mechanisms described above are sufficient to operate the system efficiently. Such as enabling to progressively release a node, where as it is no longer completely committed to a single application, but could contribute its free resources (using another container to prevent possible security concerns) to another application.

4.5.2 Research Requirements

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

Concerning the *Network layer*, we can make the following observations. By 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 fully isolated manner.

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 best. As an example, if operating in a highly dynamic environment (Internet), using a unstructured overlay network might be better to cope with the higher churn rate, but if operating in a moderately dynamic environment (shared-cluster) using the current DHT implementation would be more than sufficient.

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 require-

ment of being able to host a container, which inherently all Linux OS are capable of. We satisfy partially **Requirement 3**, because of *Ring* abstraction which provides means of creating and maintaining a decentralized structured overlay network to connect the nodes.

As for the *Virtualization layer*, through the use of lightweight virtualization we are able to abstract a resource from its underlying physical specificities, but also to provide similar isolation guarantees to full virtualization technology. We are also able to define the environment of an application and all its dependencies using a single configuration file, which ease the portability and creates fully self-contained construct that can be deployed anywhere.

Consequently, we satisfy **Sub-Requirement 4.1**, to the extent that it reduces the overhead induced by the virtualization technology to practically none. Similarly for **Requirement 2**, which we satisfy with respect to self-containment, through the use of light virtualization.

Finally, when it comes to the *Application layer*, we can assess that we fulfill **Requirement 2** completely, since 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 4.4.3, it remains an open-problem because of the underlying infrastructure of the Internet, which is beyond the scope of this thesis.

The *security* component of the API specification proposed, satisfies the subsequent part of **Requirement 3** left unfulfilled by the *Ring*, by providing secure communication channels and enforcing MAC to regulate the administrative tasks. In no means is this component providing *absolute* security to this architecture, and more attack vectors should be analyze to ensure that most of them are mitigated.

The remaining requirement, which states that no special or dedicated equipment should be necessary to participate into this system, is also fulfilled. We deem it, **Requirement 4**, satisfied because there are no explicit or implicit restrictions on the intended hardware to be used, and consequently by recycling currently available hardware is the most cost-efficient and eco-efficient, *ceteris paribus*, choice.

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 means of developing application that can be deployed using this architecture. The code was written using Python and the Twisted Event-Driven Networking Framework [Lef14]. We adopted a event-driven programming model, since 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 fundamental to the framework, and is offered as an extra feature. 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 success of an event, and a callback to handle the failure of an event; both are encompassed into the *Deferred* abstraction. The way Twisted orchestrates the callbacks contained in different deferreds is by registering them with the event-loop, called the *Reactor*. This event-loop controls the thread of execution, by enabling operations to

from twisted book!.png from twisted book!.pdf from twisted book!.jpg from twisted book!.mps from twisted book!.jpeg from twisted book!.jbig2 from twisted book!.jb2 from twisted book!.PNG from twisted book!.PDF from twisted book!.JPG from twisted book!.JPEG from twisted book!.JBIG2 from twisted book!.JB2 from twisted book!.eps

Figure 5.1: Image From TWISTED BOOK

block, and keeping track of which callbacks are associated with which operations. 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 and their interactions among one another, especially concerning the *Task* construct. Compared to multi-threaded programming, which consists of maintaining a collection of different threads to accomplish different tasks and incurs complex concurrency and synchronization mechanisms, it reduces the complexity inherent to the development of distributed applications by focusing on the possible events and their appropriate response. The following figure illustrates perfectly the distinction between these two programming models:

For the overlay-network which corresponds to *The Ring*, we chose a Python implementation of the Kademlia DHT using Twisted [Mul14]. Our decision was based on the fact that it was implemented using Twisted, and could then be easily integrated to our architecture with minimal disruptions. Also we benefited from having a 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 minimalist Python Object-Oriented Web Framework [Tea14], because of its simplicity and illustrative capabilities. It enables us to use have a web server, without any bloating features and focuses on strictly what is necessary, which ease the development of prototypes and proof of concept. We could substitute it with a more complete web framework, incurring minimal changes since of our clearly defined interfaces.

To provide virtualization capabilities to the nodes we used Docker Containers [Hyk15]. It also provides self-containing characteristics to each node, by specifying the dependencies inherent to this architecture, but also the dependencies of the applications developed via the use of *DockerFiles*. These DockerFiles act as configuration files and prescribes the requirements to execute the containers, as well as the quantification of the resources available.

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

5.2 Constructs

5.2.1 Introduction

In this section we present the fundamental constructs of this architecture, for which we can distinguish between 4 elementary constructs, namely the **Task**, the **ApplicationNode**, the **Ring** and the **Fellowships**. We present each one of the constructs, and we discuss their practical implementations.

5.2.2 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 means to return the response to the corresponding requester, without ambiguity in the presence of large amount of requests.

Tasks were heavily inspired by Twisted's concept of *Deferreds*, since they make similar promises. As we have shown earlier, *Deferreds* offers a promise of eventually returning from a blocking operation with a result, upon which the processing logic to be applied can be specified. Similarly, a task embodies this promise of eventual completion and provides the ability to specify any processing logic to be applied to the result. As a matter of fact, *Deferreds* are used to chain various processing steps to any given task. Upon creation of a task, a *Deferred* is created and it consists of the processing logic that must be followed when this task is dispatched to a node. The principal function required to process this unit of work, is the first to be chained into the callback chain of the *Deferred*. Then, any subsequent post-processing will be further chained to that same callback chain, which represents the complete sequence of operation

A task corresponds to an atomic unit of work. Analogous to database atomic transaction properties, 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 states, and by receiving the task it is sufficient to execute and carry out the corresponding sequence operations. We distinguish between two types of tasks, *Worker Task* and *Data Task*. The former consists of **ANY**

type of computational task required, whereas the latter consists of task that pertains to persistent data (either storing or retrieving).

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: *task_object* and

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

Application Deploying Nodes are the nodes that wishes to contract contributing nodes in order to deploy an application across a collection of nodes. Deploying, in this context, refers to providing the nodes with the appropriate processing logic (according to their role) and then dispatching the incoming workload to the nodes accordingly. It is responsible for translating incoming requests into tasks and providing tasks to available contributors. Then, once a response is formulated it returns it to the originator of the request.

On the other hand, **Contributing Nodes** processes the tasks that they are assigned. 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).

It is necessary to make this distinction explicitly in the context of this architecture, due to centralized characteristics of web hosting. It is not trivial to provide decentralized web hosting, as shown in [AB14], using the current Internet infrastructure and remains an open-problem.

5.2.3 The Ring

This abstraction was presented in the previous chapter, and provides the means to connect all the participants in a public environment through a well-defined interface. It also enables application deployers to find potential contributing nodes, and provides a public repository for the common resource specification template.

By 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 defines the *Resource Specification (RS)* and include any desirable attributes, simply by appending the attributes to a public repository. This public repository corresponds to the public RS template, which is 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* [LCP⁺05]. 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 means for connecting 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 necessary between the nodes.

Currently we use a whitelisting mechanism to provide a private environment, which means that only the nodes contained in the list are allowed to participate in this environment. The list is the result of selecting the resources, and publishing a 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 provisional 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 [LLC11]

Due to time constraints, we have not implemented this construct as a stand-alone overlay network as prescribed in the previous chapter, and simply uses the whitelisting approach to security. As future work we would implement Fellowships using a protocol similar to T-Man [JMB09], 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 the 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 implementation of the platform effortlessly. Finally, we are also providing a private networking environment to contain deployed application execution, using a whitelisting like mechanism.

5.3 Workflows and Protocols

In this section we will present the procedure for the initialization of 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. We then present the protocol that nodes uses in order to communicate within a Fellowship and conclude with the presentation of the web protocol which interfaces 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 will distinguish between the Contributing node's workflow, which consist of a subset of the Application Deployer's node workflow, and that 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.*

By using these two workflows we are able to create a node and successfully join this architecture. But 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 that occurs inside a Fellowship, are conducted according to a well-defined protocol, called **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. Again, the application deploying node is depicted as a server since it hosts the web server.*

The protocol can be depicted as follows:

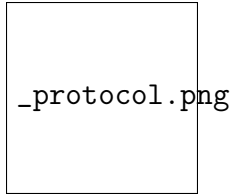
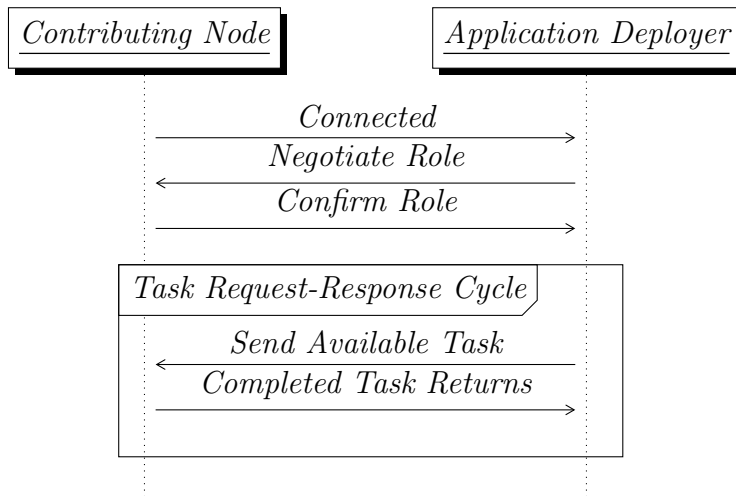


Figure 5.2: Web Request Translation into Task(s).



5.3.3 Web Protocol

This 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 consists of receiving a web request, and then the web server must generate 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 completion of a task, the result is retrieved and a *HTTP POST* request is formulated by the web protocol and sent to the web server. The web server then is responsible to present the information back to the originator of the initial request.

The rationale 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 hosting 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 this 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 without major changes required.

5.4 Proof of Concept: Calculator

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

5.4.1 Overview

The application itself 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 were considered: the addition (+), the subtraction (-), the multiplication () and the division (/).*

We implemented this application, making 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, the more concurrent requests can be satisfied.*

5.4.2 Component Interaction

In this subsection we present a style of reasoning that is central to this architecture, and represents the main interactions between the different components of this application.

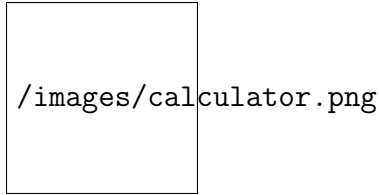


Figure 5.3: Application Overview.

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 such a reasoning could be:

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 the a POST request containing the two operands and the operator 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, and is dispatched to any available nodes.
6. **(Task Completion)** Upon completing the task, the node will send the result to the dispatcher.
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 [Mic06], 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 be sent 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 (the derived form of events).
- **Downstream Event-Driven Activity** is the downstream activity initiated by event(s), and 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 from 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 of the event channel to dispatch the events and will use it also to handle any downstream event-driven activity. 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 concerns, and avoid possible design mistakes to provide scalability.

5.4.3 Application Deploying Node

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

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 it is queued. Then, using the Fellowship protocol it will dispatch the task(s) to available nodes and collect the result(s) of completed task(s). The results are extracted from the

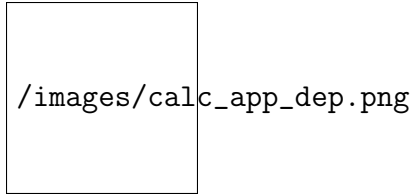


Figure 5.4: Application Deployer Node as part of the Calculator Application.

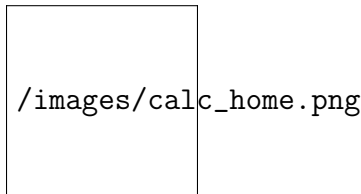


Figure 5.5: Calculator's Web Interface

completed task(s), and will be presented to the end-user, as a downstream event-driven activity emerging from the processing of that event.

The application deploying node is responsible for the task queues in this specific application, since 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:

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 the Contributing Node, and then the logic to process tasks.

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

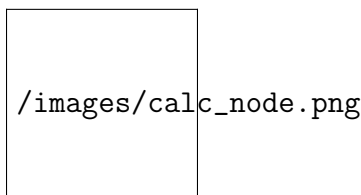


Figure 5.6: Contributing Node as part of the Calculator Application.

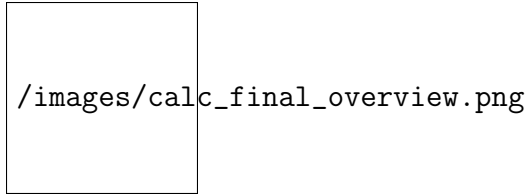


Figure 5.7: Calculator Application Architecture Overview.

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

In the context of this application, we defined the logic for processing task into a module called `worker_process`. This is able to encapsulate all the logic necessary to perform any operation on the two dependent code and the architecture related code, and augments portability of the source code.

In order to execute a task it will call the appropriate function inside the module and pass it any parameters (via a list of parameters) necessary. It will then append the results to the task object, passed in parameters, and reply to the application deployer with the completed task.

5.4.5 Putting it all together

We present the final representation of our Calculator application, illustrating the different components and their interactions.

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. Or we can adopt a Event-Driven Architecture, if we have more an Enterprise type of application. Nonetheless, this architecture offers the ability to construct simplistic web applications, as well as more computationally intensive web applications (as we will see in the next chapter).

Chapter 6

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

6.1 Introduction

In the previous chapter, we have introduced the technologies used as well as the 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 of a system for *Multi-Document Text Summarization using Genetic Algorithms* that was implemented based on [QHH08], in the context of a term project. By no means, is this an efficient multi-document text summarization system, rather it embodies a computationally intensive highly parallelizable problem which is appealing to showcase this architecture.

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 extended the existing work on genetic algorithms and text summarization to account for multiple documents. We then use a similar analysis-based approach to be able translate this problem into an application that can be deployed on our architecture. This analysis will serve as a foundation to devise an algorithm that is compatible with an event driven architecture. Finally, we will present the characteristics and roles of each type of nodes; then we will present a discussion on this implementation.

6.2 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 [MM99] as a starting point for Automatic Summarization, and see [ES08] as an introduction to Evolutionary Algorithms.

6.2.1 Automatic Summarization

To understand how to automate summarization of documents, we must understand what constitutes a summary. Which 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 [J⁺99].

To generate a summary, one of the technique that can be used, relies on extraction of linguistic units, from a document to represent the information conveyed in it, 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 condense form, which is known as *abstraction-based summarization*.

We are interested, for this use-case, in extraction-based summarization, more specifically we are interested in a technique named *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, a summary is 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 so will provide means 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)*, see [GMCK00] as a starting point for MDTS using sentence extraction.

6.2.2 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 generating a summary of a given length containing the most important sentences. Thus, one can interpret this as an optimization problem and apply an evolutionary algorithm to search the solution space for near-optimal solutions, such as the authors of [QHH08] did 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, can be presented as follows: There are several considerations to take into account while using GAs, especially on how

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.
```

to represent an individual in the population, which type of crossovers should be applied, enforcing elitism in the selection process, what 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 [ES08].

6.2.3 Extensions Proposed for MDTs

We have proposed extensions to the work of [QHH08], in order to provide multi-document summarization capabilities. But first, we present a high-level overview of their procedure, then present the extensions we proposed.

This algorithm is conceptually simple and intuitive. Some difficulties can be encountered with respect to tuning the parameters of the genetic algorithm, but it is a problem inherent to genetic algorithms in general and not to this specific problem of automatic summarization.

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 indices.

```

We propose to utilize this methodology integrally, and execute in parallel on multiple documents that shares a common topic to generate a collection of summaries. This collection of summaries is then used a corpus of salient sentences, but instead of using the GA to find the final summary, we use a redundancy reduction function to remove excessively similar sentence and achieve the desired summary length. Thus, a new and updated algorithm that is able to summarize multiple document on a single topic looks like this:

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 indices.

```

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 decided to simply 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 well suited for this architecture.

6.3 Translation of the Problem for this Architecture

In this section we analyze the MDTs system, and translate it into an application for our architecture. 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, present the resulting workflow of the application.

To start our analysis, we need to make the following assumptions:

- (some) data need to be persisted, if we want to preserve the statelessness properties of the tasks.
- Computations will be performed.
- Provide support for multi-tenancy, no ambiguities between task(s) corresponding to a user's request and another user's request.

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. We start by identifying the logical event flow layers, then we present the different possible event flows. 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.



Figure 6.1: Event Flow: Incoming Request.

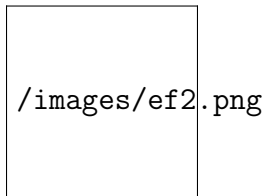


Figure 6.2: Event Flow: Processing a Request.

- **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 completed 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*.

The Incoming Request Event Flow 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 corresponds to the processing task for this data, which implies retrieving the data (documents) from the database, process it using the genetic algorithm, and persisting the results back into the database.

The Returning a Response Event Flow consists of monitoring the database for all the

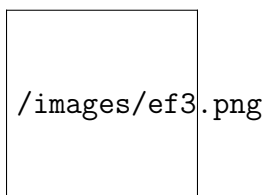


Figure 6.3: Event Flow: Returning a Response.

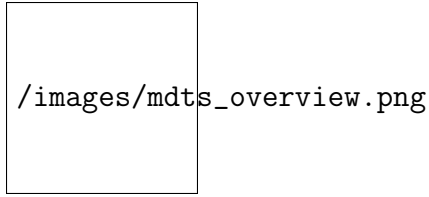


Figure 6.4: MDTs: General Workflow.

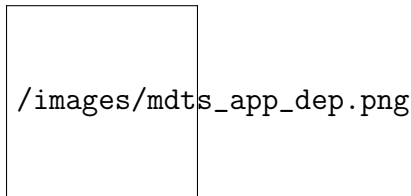


Figure 6.5: Application Deployer Node as part of the MDTs Application.

results to be persisted, and then creating and dispatching a task to process it by applying the redundancy reduction algorithm. Upon completion, the results are sent back to the end-user.

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 by using these session primitives to organize the data in the database.

6.4 Implementation Details

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

6.4.1 Application Deploying Node

The interactions between the end-users and the contributing nodes happens through this node, which acts as an interface to both.

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, since 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. Also, since we are using data

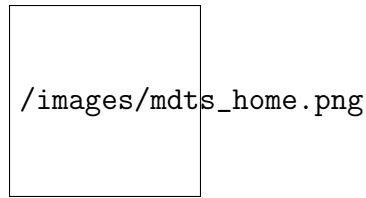


Figure 6.6: MDTs Application's Web Interface

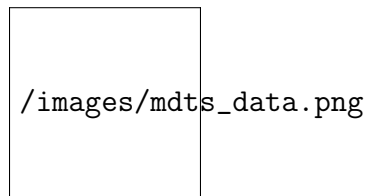


Figure 6.7: Data Node as part of the MDTs Application.

node(s) we have an additional task queue, thus one task queue for the worker task and one task queue for the data task.

Another characteristic of this implementation of the application deploying node, is the task creation logic as a result of a completed data task. As part of the core logic of this architecture, or the application independent components, 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 consisting of the application logic.

We have implemented the web server, as mentioned previously, using CherryPy. We present a minimalistic web interface that let the users upload their files, as well as specify the different parameters of the genetic algorithm and the summarization parameters:

And the results is 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 5.4.3, 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.

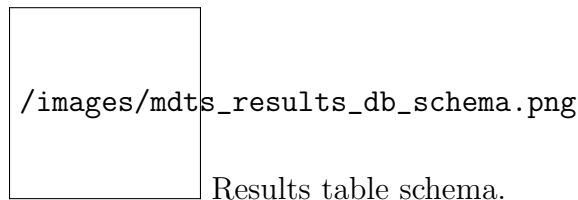
6.4.2 Data Node

The persistence of the data is the responsibility of this type of node, which is achieved by using a relational database, such as *RethinkDB* [Ope15].

We have group all of the database functionalities into a single module named *dataprocess*, and thus if *It defines a DataProcess instance, which corresponds to the process created for the*



Figure 6.8: Files table schema.



Results table schema.

database server, since we are required to have a long standing process 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 also it provides the facilities to start and stop a database instance gracefully, through eponymous functions.

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

The task functionalities are implemented in the format defined in 5.2.2, using a list of parameters and passing it alongside the task object. We defined two different functions that corresponds to actions for each of the database schemas.

The first function is `save_file`, it consists of extracting a file's content and name from a task object and storing it in the database.

The second function is `results_lookup` and consists of creating a trigger that monitors the changes for the results table.

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. The functionalities provided 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 spinning database instances with the proper arguments.

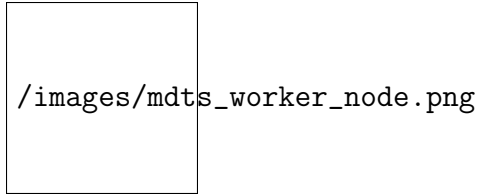


Figure 6.9: Worker Node as part of the MDTs Application.

6.4.3 Worker Node

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

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 following functions:

The first function, `retrieve_data`, as its name implies is responsible of retrieving the data from the database.

The second function, `retrieve_data_post_processing`, retrieves the results from the database, applies the necessary transformations (e.g., filtering, aggregation, etc.) and returns the results to the user).

6.5 Conclusions

In this section we present the conclusions drawn from this use-case, and the transformation of a monolithic application into a distributed web application that can be hosted on our computing platform. Among these conclusions, we discuss the ...

By using the EDA reasoning approach we are able to circumscribe any stateful dependencies in the event flows and mitigate them, to enforce the task properties of being stateless, by persisting data to the database. This is the central point of this methodology for translating application to this architecture. It also provides the tools necessary to divide the application logic into meaningful atomic units of work. Although, this application is highly parallelizable, the same principles applies to more serial problems.

Chapter 7

Discussion

In this chapter we present various musings about our architecture that outlines its purpose, advantages, and limitations. We initiate the discussion by presenting the positioning of this project amongst the wide landscape of distributed computing platforms. After which we present various open problems that were encountered along this research path. We conclude this chapter by discussing the extensibility of this system and its limitations.

7.1 Positioning of the System

In this section we present the positioning of this system amidst the sea of different other distributed computing platform or systems. It is necessary to do, because the purpose of many of these systems gets conflated together, and it provokes a severe misunderstanding of what is provided, this is especially true concerning 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 researcher try to devise a proper ontology for this *new* paradigm, see [YBDS08]. What we realize out of this attempt, is that the definition of cloud computing can be rationalized, and with the help of [MG11] it is possible to clearly differentiate, based on those characteristics, what constitute cloud computing. But if we attempt to complete this already expansive definition, by defining what the end-product is that cloud computing offers, this term *cloud computing* becomes an *umbrella* term.

Thus, we present this system as being part of the 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** aiming 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.

This 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, then could use a system such as this one to provide these 200-300 computers to the academics to a local university wishing to run experiments. Which 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 via untrusted networks, such as the Internet.

Another example, concerning local communities could be with respect to 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 statistics about the players, recorded games, event announcements, and any other desirable features to chess players and the like. Then a subset of the community, at the very least 10 members, could contribute their household contributing resources to the hosting requirements of their web applications, and to serve their community without incurring extra cost ¹. Two or three nodes could host the database, thus providing 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 failure, exposing only 3 IP addresses. They are not required to host their 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.

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 the

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

re-factor these resources in order to provide them 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 projects, using event-driven programming model and the Python programming language.

7.2 Problems Encountered

We have encountered many

7.3 Extensibility

7.4 Limitations of the System

Chapter 8

Conclusions and Future Work

Bibliography

- [AAC⁺11] Rocco Aversa, Marco Avvenuti, Antonio Cuomo, Beniamino Di Martino, Giuseppe Di Modica, Salvatore Distefano, Antonio Puliafito, Massimiliano Rak, Orazio Tomarchio, Alessio Vecchio, et al. The cloud@ home project: towards a new enhanced computing paradigm. In *Euro-Par 2010 Parallel Processing Workshops*, pages 555–562. Springer, 2011.
- [AB14] Reaz Ahmed and Raouf Boutaba. *Collaborative Web Hosting: Challenges and Research Directions*. Springer, 2014.
- [ACK⁺02] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.
- [AF06] David P Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80. IEEE, 2006.
- [And03] David P Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, pages 17–19, 2003.
- [And04] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.
- [ASK15] Mohammadfazel Anjomshoa, Mazleena Salleh, and Maryam Pouryazdanpanah Kermani. A taxonomy and survey of distributed computing systems. *Journal of Applied Sciences*, 15(1), 2015.
- [Bar01] David Barkai. *Peer-to-Peer Computing: technologies for sharing and collaborating on the net*. Intel Press, 2001.

- [BDF⁺03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.
- [BFH03] Fran Berman, Geoffrey Fox, and Anthony JG Hey. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley and sons, 2003.
- [BG09] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*, pages 10–19. IEEE, 2009.
- [Bir07] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007.
- [BJ12] HMN Dilum Bandara and Anura P Jayasumana. Evaluation of p2p resource discovery architectures using real-life multi-attribute resource and query characteristics. In *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*, pages 634–639. IEEE, 2012.
- [BJ13] HMN Dilum Bandara and Anura P Jayasumana. Collaborative applications over peer-to-peer systems—challenges and solutions. *Peer-to-Peer Networking and Applications*, 6(3):257–276, 2013.
- [BMT12] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a p2p cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 412–417. ACM, 2012.
- [CDPS09] Vincenzo D Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Cloud@ home: Bridging the gap between volunteer and cloud computing. In *Emerging Intelligent Computing Technology and Applications*, pages 423–432. Springer, 2009.
- [CDPS10a] Vincenzo D Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Applying software engineering principles for designing cloud@ home. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 618–624. IEEE, 2010.

- [CDPS10b] Vincenzo D Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Open and interoperable clouds: The cloud@ home way. In *Cloud Computing*, pages 93–111. Springer, 2010.
- [CDPS10c] Vincenzo D Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Open and interoperable clouds: The cloud@ home way. In *Cloud Computing*, pages 93–111. Springer, 2010.
- [Chi08] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.
- [CW09] Abhishek Chandra and Jon Weissman. Nebulas: Using distributed voluntary resources to build clouds. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 2–2. USENIX Association, 2009.
- [Dev15] Google Developers. Google identity platform. <https://developers.google.com/identity/>, 2015. Accessed: 12-05-2015.
- [DFP11] Salvatore Distefano, Maria Fazio, and Antonio Puliafito. The cloud@ home resource management system. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 122–129. IEEE, 2011.
- [DKMyC04] Alan Dearle, Graham Kirby, Andrew McCarthy, and Juan Carlos Diaz y Carballo. A flexible and secure deployment framework for distributed applications. In *Component Deployment*, pages 219–233. Springer, 2004.
- [DIRB14] Jose De la Rosa and Kent Baxley. Lxc containers in ubuntu server 14.04 lts. <http://en.community.dell.com/techcenter/os-applications/w/wiki/6950.lxc-containers-in-ubuntu-server-14-04-lts>, June 2014. Accessed: 30-12-2014.
- [Dol00] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.
- [DP80] IW Draffan and F Poole. *Distributed data bases*. CUP Archive, 1980.
- [DP12] Salvatore Distefano and Antonio Puliafito. Cloud@ home: Toward a volunteer cloud. *IT Professional*, 14(1):27–31, 2012.
- [DPR⁺11] Salvatore Distefano, Antonio Puliafito, Massimiliano Rak, Salvatore Venticinque, Umberto Villano, Antonio Cuomo, Giuseppe Di Modica, and

- Orazio Tomarchio. Qos management in cloud@ home infrastructures. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 190–197. IEEE, 2011.
- [EK07] Pavel Emelyanov and Kir Kolyshkin. Pid namespaces in the 2.6. 24 kernel. *LWN. net*, November, 2007.
- [ES08] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer Science & Business Media, 2008.
- [Fit11] Brad Fitzpatrick. Memcached: a distributed memory object caching system. *Memcached-a Distributed Memory Object Caching System*, 2011.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.
- [GMCK00] Jade Goldstein, Vibhu Mittal, Jaime Carbonell, and Mark Kantrowitz. Multi-document summarization by sentence extraction. In *Proceedings of the 2000 NAACL-ANLP Workshop on Automatic summarization-Volume 4*, pages 40–48. Association for Computational Linguistics, 2000.
- [GSL14] Ian Gergin, Bradley Simmons, and Marin Litoiu. A decentralized autonomic architecture for performance control in the cloud. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 574–579. IEEE, 2014.
- [GSRU07] Debanjan Ghosh, Raj Sharman, H Raghav Rao, and Shambhu Upadhyaya. Self-healing systemssurvey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.
- [Hyk14] Solomon Hykes. Dockercon. <http://www.dockercon.com/>, 2014. Accessed: 30-12-2014.
- [Hyk15] Solomon Hykes. Docker. <http://docs.docker.com/>, 2015. Accessed: 16-02-2015.
- [Inc14a] Google Inc. Google app engine documentation. <https://cloud.google.com/appengine/docs>, 2014. Accessed: 08-01-2015.
- [Inc14b] Microsoft Inc. What is microsoft azure? <https://azure.microsoft.com/en-us/overview/what-is-azure/>, 2014. Accessed: 30-12-2014.

- [Inc15] Amazon Web Services Inc. Amazon web services. <http://aws.amazon.com/>, 2015. Accessed: 08-01-2015.
- [J⁺99] K Sparck Jones et al. Automatic summarizing: factors and directions. *Advances in automatic text summarization*, pages 1–12, 1999.
- [Jel] Márk Jelasity. Gossip-based protocols for large-scale distributed systems.
- [JK06] Mark Jelasity and A-M Kermarrec. Ordered slicing of very large-scale overlay networks. In *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on*, pages 117–124. IEEE, 2006.
- [JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.
- [JMB09] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.
- [JOV05] Hosagrahar V Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.
- [JVG⁺07] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.
- [KC03] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [KL03] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81, 2003.
- [LCP⁺05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.
- [Lef14] Original Author: Glyph Lefkowitz. Twisted matrix labs. <https://www.twistedmatrix.com/trac>, 2014. Accessed: 30-12-2014.

- [Lin76] James G Linders. Distributed data bases. *Computers & Geosciences*, 2(3):293–297, 1976.
- [LLC11] Zhen Luo, Zhishu Li, and Biao Cai. A self-organized public-key certificate system in p2p network. *Journal of Networks*, 6(10):1437–1443, 2011.
- [Los15] Canonical Ltd. and open source. Linux containers. <https://linuxcontainers.org/>, 2015. Accessed: 12-05-2015.
- [Men11] Paul Menage. Linux kernel documentation/cgroups/cgroups.txt, 2011.
- [MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. x, 2011.
- [MGLPPJ13] Rubén Mondéjar, Pedro García-López, Carles Pairot, and Lluís Pamies-Juarez. Cloudsnap: A transparent infrastructure for decentralized web deployment using distributed interception. *Future Generation Computer Systems*, 29(1):370–380, 2013.
- [Mic06] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.
- [MKL⁺02] Dejan S Milojevic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing, 2002.
- [MM99] Inderjeet Mani and Mark T Maybury. *Advances in automatic text summarization*, volume 293. MIT Press, 1999.
- [MM02] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- [Mul14] Brian Muller. Kademlia: A dht in python. <http://findingscience.com/python/kademlia/dht/2014/02/14/kademlia:-a-dht-in-python.html>, 2014. Accessed: 30-12-2014.
- [Ope15] Opensource. Rethinkdb: The open-source database for the realtime web. <http://rethinkdb.com/>, 2015. Accessed: 12-05-2015.
- [ÖV11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer, 2011.

- [PD11] Harald Psailer and Schahram Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011.
- [PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- [PMRS14] Mathieu Pasquet, Francisco Maia, Etienne Rivière, and Valerio Schiavoni. Autonomous multi-dimensional slicing for large-scale distributed systems. In *Distributed Applications and Interoperable Systems*, pages 141–155. Springer, 2014.
- [QHH08] Vahed Qazvinian, Leila Sharif Hassanabadi, and Ramin Halavati. Summarising text with a genetic algorithm-based sentence extraction. *International Journal of Knowledge Management Studies*, 2(4):426–444, 2008.
- [RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM’09. Fifth International Joint Conference on*, pages 44–51. Ieee, 2009.
- [RR08] Leonard Richardson and Sam Ruby. *RESTful web services*. ” O’Reilly Media, Inc.”, 2008.
- [RV11] Etienne Riviere and Spyros Voulgaris. Gossip-based networking for internet-scale distributed systems. In *E-Technologies: Transformation in a Connected World*, pages 253–284. Springer, 2011.
- [sal15] inc. salesforce.com. Crm and cloud computing to grow your business: Salesforce.com canada. <http://www.salesforce.com/ca/>, 2015. Accessed: 16-02-2015.
- [Sar10] Siamak Sarmady. A survey on peer-to-peer and dht. *arXiv preprint arXiv:1006.4708*, 2010.
- [SMJ00] Rick Sturm, Wayne Morris, and Mary Jander. *Foundations of service level management*, volume 13. Sams Indianapolis, IN, 2000.
- [SMK⁺01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

- [SPEW11] Amir Ali Semnanian, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 719–724. IEEE, 2011.
- [Tav12] Djamshid Tavangarian. Virtual computing: The emperors new clothes? In *Software Service and Application Engineering*, pages 53–70. Springer, 2012.
- [Tea14] CherryPy Team. Cherrypy: A minimalist python web framework. <http://www.cherrypy.org>, 2014. Accessed: 30-12-2014.
- [UPS11] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):8, 2011.
- [Wat08] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.
- [Wik15] Wikipedia. Seti@home wikipedia article. <http://en.wikipedia.org/wiki/SETI@home>, 2015. Accessed: 16-02-2015.
- [YBDS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.
- [ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.