μCloud: A P2P Cloud Platform for Computing Service Provision

Submitted in fulfilment
of the requirements of the degree of

Master of Science

at Rhodes University

Ghislain Fouodji Tasse
Supervisor: Dr. Karen Bradshaw

Grahamstown, South Africa
April 9, 2012

Abstract

The advancements in virtualization technologies have provided a large spectrum of computational approaches. Dedicated computations can be run on private environments (virtual machines), created within the same computer. Through capable APIs, this functionality is leveraged for the service we wish to implement; a computer power service (CPS). We target peer-to-peer systems for this service, to exploit the potential of aggregating computing resources. The concept of a P2P network is mostly known for its expanded usage in distributed networks for sharing resources like content files or real-time data. This study adds computing power to the list of shared resources by describing a suitable service composition. Taking into account the dynamic nature of the platform, this CPS provision is achieved using a self stabilizing clustering algorithm.

So, the resulting system of our research is based around a hierarchical P2P architecture and offers end-to-end consideration of resource provisioning and reliability. We named this system μCloud and characterizes it as a self-provisioning cloud service platform. It is designed, implemented and presented in this dissertation. Eventually, we assessed our work by showing that μCloud succeeds in providing user-centric services using a P2P computing unit. With this, we conclude that our system would be highly beneficial in both small and massively deployed environments.

Acknowledgements

"We cannot solve our problems with the same thinking we used when we created them."
Albert Einstein

This dissertation was made possible through the patience and guidance of my supervisor, Dr. Karen Bradshaw. She has spent countless hours assessing my work, providing invaluable assistance with the research. The years I spent with Karen have been productive both intellectually and socially, and I hope that my dissertation work ends up being a solid foundation for many others.

I would also like to thank all the faculty members who have assisted me in the research, pursuit of funding, and day-to-day administration issues. The IT division of the Computer Science department has been instrumental in the experimental component of my research. Prof. Alfredo Terzoli, Centre of Excellence director, was a key player in my quest for funds, by respecting my work and believing in my skills. I take this opportunity to acknowledge the financial support of Telkom SA, Stortech, Tellabs, Eastel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science.

Last but not least, I thank my family, colleagues and friends who have undoubtedly been the driving force of my success over the years. During my academic career, they have supported me as I pursued challenging and often time consuming courses of study. To them I say: "JE VOUS AIME".

Related Publications

Part of the work that appears in this thesis has been presented in the following conference papers:

1. Ghislain Fouodji Tasse and Karen Bradshaw. Ubiquitous Cloud Computing. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, 2010.

2. Ghislain Fouodji Tasse and Karen Bradshaw. Transparent and Reliable Computing Power Service Provision on P2P systems. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, 2011.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel and distributed computing are two concepts that were introduced in the past few decades to take advantage of the increase in computer hardware capabilities. These concepts have revolutionized the approach to programming and their implementation can be fairly complex. But, with the amount of work done in the area, their usage can be facilitated by abstracting certain complexities. This is why trends such as grid or cluster computing have arisen, which provide in their own way, a certain level of abstraction to one's computation. These trends can also be distinguished by the granularity (fine-grained or coarse-grained) of the computational problems they were natively designed to solve.

Still, with the tremendous development in computing technology, it has now become possible for most users to enjoy the benefits of high performance or multi-purpose computing. In this respect, the "cloud computing" trend has distinguished itself. The term generally refers to the delivery of hosted services (e.g., Database as a Service) over the Internet. Indeed, web technologies nowadays offer greater flexibility of service creation, from mere file transfers to clouds. We are witnessing more and more diversity of web services and how they are increasingly shaping our computing experience. Cloud designs are becoming user-centred, thereby offering a ubiquitous web interaction. Though not entirely novel, clouds are emerging IT delivery models that bring new insight on how to reason about computing services. To cite the definition in Stratus' white paper, "cloud computing is the use of networked infrastructure software and capacity to provide resources to users in an on-demand environment" [12, p. 1]. The term "resource" has also gained a wider meaning; it may denote information, storage or

even pure computing power. In this work, we focus on the latter, as an increase in computing speed is an undeniable advantage for user computations.

Resource-intensive applications are being developed both in research institutions and commercial industry, making computing power (notably processing cycles and memory) an important entity in the success of their operations [66]. Fortunately, the consolidation of virtualization techniques comes with benefits such as resource reservation and/or sharing for local (host) or remote (guest) use. The latter is of importance because it adds another dimension to how a resource can be utilized. Rather than having a resource idle because it is not being used locally, it can be made available to a remote user.

Therefore, the core of our intended service is to be able to serve, conveniently, the computer resource of a node to a different (remote) node. Because computing power is a low level entity, transparency is required in the service provision. In addition, because the operating environment consists of distinct nodes, reliability needs to be consider in the service provision. By doing this, we comply with Verizon's definition that a computing service is an on-demand computing platform suitably set up to satisfy user needs and to exhibit ease of use [3]. The backbone of the platform, μCloud in our case, is built using a P2P network, where nodes may have resources to share with its peers. This sharing of computer resources by leveraging processing cycles, cache storage, and memory is referred to as P2P computing [66]. We assume that peer nodes are not permanently busy, or that nodes, fully dedicated to sharing, exist. Therefore, we stipulate that such nodes should *work* during their "idle" times. Effectively, one can maximize resource utilization within a P2P system by minimizing starvation.

With this brief background, we motivate our research in the next section. Thereafter, we raise some research questions, followed by a description of the research objectives and the structure of the thesis content in subsequent sections.

1.1 Motivation

Although high performance computing is far from reaching regular computing users, the ease of a cloud interface allows us to make this possible. Moreover, the extensible nature of a constructed P2P infrastructure allows individuals or groups to donate their local infrastructure

either on a permanent or on-demand basis. We are motivated by the ability of harnessing a distributed set of computer resources and building an intelligent cluster with the following characteristics:

- Self adaptive, scalable, self optimizing.
- Transparency (case of use): A user can easily run a computation on a remote machine while dynamically visualizing the results on a local machine.
- Safety or reliability of the infrastructure (fault tolerance, multiple points of failure).

1.2 Research Questions

How can we increase computing performance without investing in a new infrastructure?
In other words, we need to figure out ways of using available computer resources more efficiently. In fact, hardware is currently under-utilized and it is believed that adequate software frameworks can be created to provide a set of new services to users. As defined previously, the use of cloud computing in a way, is an answer to this question.

Can a reliable computing service be provided on a non-deterministic heterogeneous network?
Though we know our answer to the previous question is related to the cloud computing trend, we still need to cogitate on the design and the desired structure of the underlying computing infrastructure. Because our research implementation attempts to generate a cloud on top of a set of disparate computer resources, via clustering technologies, reliability is an expected concern, which we will have to resolve. The ownership and the spatial distribution of the computers are most likely to be different, but a safe collaboration needs to be established between them.

Is this research a step towards "personal computing"?
The term "personal computing" refers natively to the usage of personal computers (PCs). However, in this new age of technology, we believe it is no longer limited to PCs, and has invaded the area of clouds. A generic term that names this type of computing is introduced

in Chapter 2, namely the term ubiquitous computing. Effectively, this research question is related to how supercomputing can be made accessible to the average PC user. A model, which is compliant with the initial interface of the user computing environment, is obviously required.

1.3 Aim

With the information given so far, our aim is quite clear: to design a robust cloud environment with the underlying hardware being a set of peer nodes with different abilities. It is a ubiquitous shared computing environment. The ubiquity occurs at two levels: presentation level and computing level. The former simply refers to the ease of use and deployment of the system. The second level refers to the fact that peer nodes operate independently, but also cooperate appropriately to exhibit a global behaviour; that of a sustainable cloud. For this, we identify the main goals of this research to be the following:

- create an algorithm to cluster nodes in P2P networks such that the cost of resource discovery is minimized,
- compose a service that can respond to client computing needs, and
- build a general purpose platform for hosting such services. This platform is named μCloud where μ represents its small footprint.

Note that the last goal inherits solutions obtained in the first two. With respect to our aim, we intend to implement μCloud in four components, to handle the following roles: administration (i.e., coordinator), resource management (i.e., scheduler), service creation, service migration and storage management (see Fig. 1.1).

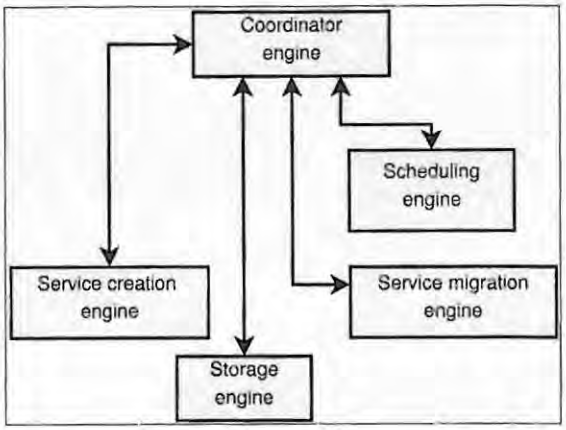


Figure 1.1: μCloud main components.

1.4 Thesis Organization

The remainder of this thesis is structured as follows.

- Chapter 2 presents a background study on the computing trends introduced here. This study is completed with the description of previous work done in the research area.
- Chapter 3 proposes a simple P2P framework for resource clustering and discovery.
- Chapter 4 discusses two methods for composing computing services.
- Chapter 5 presents the μCloud platform. The rationale and concepts behind its design and implementation are discussed.
- Chapter 6 analyses the results and observations obtained from network simulations designed to evaluate the P2P methods presented in Chapter 3. For this evaluation, a lightweight simulator is implemented.
- Chapter 7 analyses the results and observations obtained from real test experiments of the μCloud system. Key aspects of this system are assessed.
- Chapter 8 concludes the thesis and suggests ways to further improve a P2P cloud implementation and deployment as future work.

Chapter 2

Background and Related Work

Building a system for dynamic service provisioning requires knowledge of P2P platforms and cloud services. This chapter presents an essential background study on these major topics. It starts with an analysis of P2P techniques and systems in Sections 2.1 and 2.2. This is followed by a discussion on the cloud computing paradigm, which exposes the origin and nature of cloud services. Finally, we look at how these topics relate to each other and how they will help us achieve our goal.

2.1 P2P Frameworks

A P2P network is any distributed network architecture composed of participants that make a portion of their resources directly available to other network participants (peers). This concept is mostly known for its expanded usage in systems for sharing resources like content files or realtime data [65]. A comprehensive survey on current P2P systems is given in [45]. A P2P networked system has two distinctive properties: churn rate and potential for scalability. It is essential that a P2P system benefits from this potential; that is, it should be scalable. By handling the churn property adequately, a P2P framework will allow the system to scale. We study how peers affect their network when they die or become "live". Desirably, a network should become more powerful when a node joins, and should not break when a node leaves. This dynamic effect (due to adhoc connections) in the network is the core issue

a P2P framework addresses. Clustering algorithms are used to handle this issue efficiently. Since our working environment is distributed by nature, it follows logically that a distributed clustering algorithm is needed. The next two sections present candidate algorithms.

2.1.1 Dominating sets

Overview

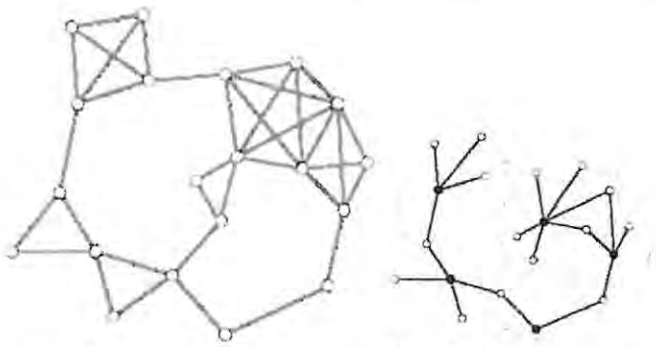
This is a technique used by many applications to defer a rigid structure from an adhoc network topology. This structure is necessary for routing or further clustering. To quote Rajaraman in [56, p. 68], "the most basic clustering that has been studied in the context of adhoc networks is based on dominating sets". Elements of the set (*dominators*) constitute the backbone of the network, by ensuring its stability. A *dominator* node is a supervisor responsible for nodes within its vicinity. It may represent those nodes in routing negotiations for instance [39]. Further details of this concept are provided later in this section. The dominating set (DS) solution is similar in concept to the so-called *super-node* solution, where nodes in the P2P system are divided into two classes: *leaf nodes* and *super nodes*. Examples of such systems are Gnutella 0.6 or Kazaa [45].
Now, an issue with building DS is the complexity of the process. Effectively, an algorithm may need to run several times (fully or partially), depending on the topological changes in the network, to reach completeness. In fact, most classical algorithms stabilize in an exponential number of moves. This is why some of them optimize either by minimizing the size of the set, or by weakening the connectivity between the *dominators*, or even by settling for a non optimal solution [68].

Connected dominating sets

A connected dominating set (CDS) is a dominating set where each member has at least one connection with a neighbour node. This number of connections is defined by the term K-consistency, denoted as k. A set is fully connected if for each node, there is at least one route from it to any other node in the network (K-consistency ≥ 1) [48].
The CDS concept was first proposed by Chen and Listman [34], using a distributed clustering

algorithm based on a centralized approximation algorithm for finding a small CDS. The algorithm partitions the network into graph regions. Then, in each region, it chooses one node and moves outwards using a Minimum Spanning Tree algorithm in order to build a CDS for that region. Negotiations are done at the borders of regions to generate a CDS of the whole network graph. But, this procedure is approximative and has a high message complexity [15].

The mathematical definition of CDS given in [68] is as follows: "Let G = (V , E) be a connected undirected graph. A dominating set S of G is a subset of V such that each v $\in V \setminus S$ has at least one neighbour in S". The notion of *neighbour* is dependent on the algorithm used. That is, some algorithms define a *neighbour* as a node one or two hop(s) distant from its *dominator*. In the case that it is *one hop*, the algorithm interprets the network as a unit graph, and deduces the CDS. This is depicted in Fig. 2.1.



(a) A sample unit disk graph. (b) Weakly connected dominating set.

Figure 2.1: Connected dominating set example [15].

Now, going back to the complexity of CDS algorithms, it has been proved in [39] that there is no deterministic distributed algorithm that in O(n) time is able to remove at least one edge and preserve connectivity. To reduce their complexity or improve their effectiveness, some algorithms add a few tweaks to the CDS concept.

- **Weakly Connected Dominating Sets.** A CDS can be rather large in size. By relaxing the connectivity property of the set, its size can be reduced considerably. It should be noted that the set should still maintain a good "low stretch", i.e., interconnection should be fairly acceptable for the targeted usage of the network [39].

- **Weakly Connected Minimal Dominating Sets.** As defined in [68], "S is a minimal dominating set if for any node v $\in$ S the set S\{v} is not dominating". Adding the "minimality" property definitely increases the complexity of the algorithm, as it requires more iterations to achieve completeness. However, this property increases the speed performance of the routing in the network, as the number of *dominators* is reduced.

Basically, most CDS algorithms assume a graph G representing the network at hand, then build a CDS from it using all the vertices available in G. By requiring some knowledge of G (the size of G is assumed to be known), these algorithms simply react to changes in the network. To optimize this, Akbari and Reza [15] proposed a distributed learning automata approach for taking proactive action.

2.1.2 Hierarchical clustering

Setting up a hierarchy is a way to abstract large scale networks. Effectively, this means creating a virtual architecture [34, 56]. Without the latter, there is no obvious heuristic that can be used for routing a message through the network. In this case, the prevalent method used is "flooding", which requires multicasting or broadcasting capabilities in the network. This is a trivial solution to the clustering problem, as it offers simplicity and is theoretically scalable [65]. However, *N-casting* (where N > 1 is the number of target nodes) reduces network bandwidth and increases time complexity as N increases.

In a hierarchical clustering, nodes are organized in order of relevance. This relevance is dependent on any combination of distinct factors, examples of which are geographical location and computing power. This reduces or eliminates the use of flooding by providing a heuristic routing in the network graph. The methodology offers a robust structure, by decentralizing administrative power in the cluster down the hierarchy. A trusted authority node is chosen to be responsible for maintaining or updating information about nodes it supervises (beneath it in the tree), thereby facilitating routing (see Fig. 2.2) [20]. Because this technique requires strong self-organization capabilities, the signalling traffic is quite high. Though it is often criticized for this maintenance cost, the algorithm offers greater scalability. The next section presents an example of a virtual network architecture implementation.

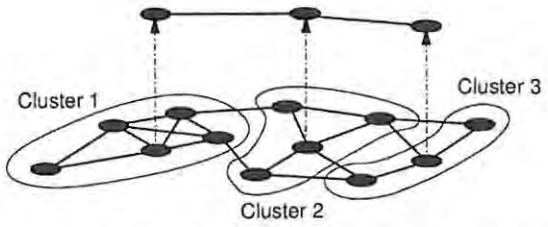


Figure 2.2: Virtual hierarchical topology [27].

2.1.3 IPOP (IP Over P2P)

IPOP is a network virtualization implementation that creates virtual IP networks on top of a P2P overlay [42]. It establishes a decentralized, private and self-configured network system. This private environment eases task scheduling and execution. Effectively, with IPOP, peers spanning multiple administrative domains can be aggregated in a virtual IP network providing bidirectional connectivity or seamless peer interactions. System state can easily be transmitted in the entire network [52]. IPOP is based on the Brunet P2P library to create a structured routing within the network. The Brunet framework establishes a structured overlay network where each node keeps k $\leq$ log(N) shortcut connections and routes to any other node with a latency of O($\frac{1}{k}log^2(N)$) where N is the total number of peer nodes [28].
Fig. 2.3 depicts the structure of an IPOP packet by showing the position of the Brunet header. It is argued that this header causes overhead by creating latency during routing and consequently reducing network performance.

With a static (virtual) structure, it is easy to schedule a computation using a deterministic algorithm [60]. In fact, clustering techniques are ultimately developed to facilitate and/or speed up resource discovery (peer selection) in a system.

2.1.4 Resource discovery techniques

A peer resource has to be detected and examined, to determine in which way it could be useful to the overall system. With respect to this, some P2P frameworks provide resource registration facilities. Depending on the P2P approach (unstructured or structured) chosen for the cluster (grid) design, either an information policy or a resource indexing mechanism

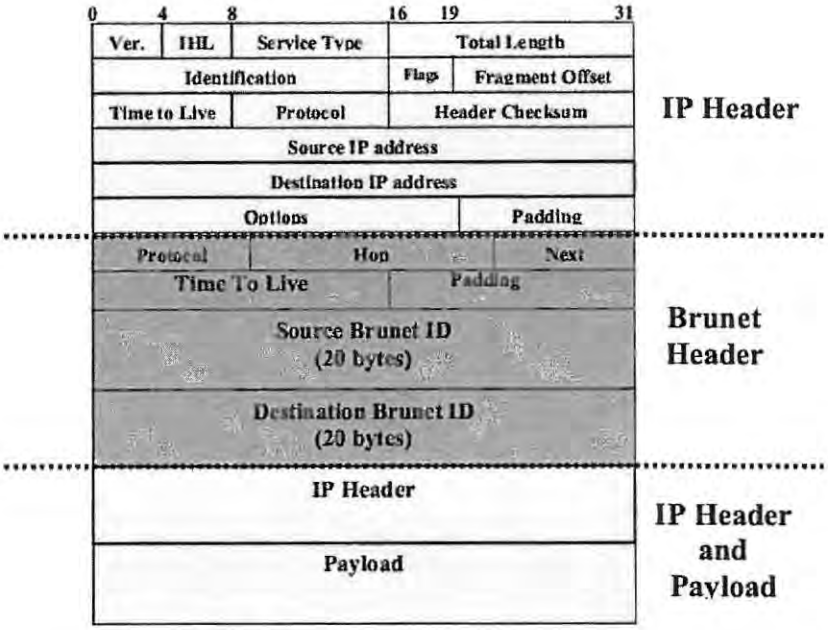


Figure 2.3: Structure of an IPOP encapsulated IP packet. The IP and Brunet headers are used for routing in the IP and IPOP networks (overlay network), respectively [42].

can be used to resolve this issue. Most unstructured P2P frameworks develop a resource query protocol based on flooding. On the other hand, structured P2P systems use a popular indexing technique known as a Distributed Hash Table (DHT). In a DHT, a node keeps a hash table revealing the resources it supervises [45, 65]. This is mainly used in file sharing, where resources (files) can only be efficiently identified via their hashes.
Trunfio et al. provide an explanatory comparative study of resource discovery methodologies in [65]. Appendices A.1 and A.2 give a summary of their study.

2.2 P2P Systems

2.2.1 Adhoc systems

Adhoc is basically a communication mode or setting that allows devices to directly communicate with each other without a router or a preset configuration. It also characterizes the

freedom of its participating agents to join or leave the communication. Extensively called an emergent network, an adhoc network allows a clustering environment to be constructed spontaneously. Its inherent independence of a fixed infrastructure or centralized control constitutes an undeniable benefit for distributed systems [44]. Although adhoc technology was first introduced to cope with the behaviour of wireless technologies (e.g., MANET), these systems have common properties with P2P systems. We claim from this that solutions that work in adhoc systems may also work in P2P systems. With the success of CDSs in adhoc systems for self management, it follows that the DS concept can be helpful in a P2P system [44].

2.2.2 Grid systems

The term Grid was introduced in the mid 1990's, to name the idea of making effective use of a large set of disparate computer resources [67]. Since then, the concept has matured in research centres and has gained a place within the industry. Grids can be viewed as distributed systems on an evolutionary scale. If the resources involved offer mainly storage or computing power, the grid is classified as a data grid or computational grid, respectively. Because of the movement in the area, we focus on computational grids, which are those where multiple computers cooperate via networking technologies to solve a single computing problem in a simultaneous and coordinated way. This cooperation yields a better overall performance than any of the participating entities can offer by itself [38].
Structured P2P algorithms are heavily used in this type of system. It should be noted that some grids are built on top of P2P, termed P2P grids [20].

2.2.3 Cluster environment

Overview

A cluster environment is basically a grid environment at a small scale; i.e., an environment consisting of computer resources situated within the same vicinity. A formal definition would be that a cluster is a team of computers connected via a private, high-speed network, which enables the cluster to be used as a unified computing resource. This permits a flexible

configuration whereby essential components like processors or memory can be hot plugged into the cluster. This preserves prior investments [21]. It should also be mentioned that the concepts related to grids (presented in the previous section) also apply to clusters to a certain extent. There is however a difference in network infrastructure between them. A cluster's interconnection speed nowadays is usually of the order of teraflops [61]. Because a cluster operates at a small scale and uses such high network speed, interconnection issues are relatively small.
Nevertheless, the convenience of such architecture comes at a high price. Here, we focus on another relatively new architecture where resources are natively independent. A cluster of workstations is becoming widely used for exploiting idle clock cycles of relatively inexpensive computers.

Cluster of workstations

A cluster of workstations (COW) is a computer network that connects several computer workstations using special software, forming a cluster. This is contrary to proprietary clusters built around high-end SMP processors and custom network technologies [22, 21]. The increase in PC's microprocessor performance and improvements in local area networks have led to systems capable of performance in the order of tens or hundreds of gigaflops. Therefore, interest in these commodity clusters is rising because such clusters are potential candidates for replacing massively parallel processors (MPP) in high performance and general purpose computing [17]. A price-performance ratio comparison between the two architectures quantitatively justifies this interest. In fact, in consideration of this ratio, the ACM Gordon Bell Prize was awarded to a PC cluster in 2000 [6].
From the description given above, it is clear that a COW represents a cheap replacement to conventional supercomputers. The latter are powerful, costly and often specialized computers that are produced by only a handful of companies [32]. The concept of COWs arose for two main reasons: the presence of under-exploited resources and the obvious need for processing power. By under-exploited resources, we refer to the fact that most desktop machines are actually idle more than 80% of the time [41]. Fortunately, an efficient use of idle workstations can result in an improvement in performance of one's computation.

A popular and successful grid environment present today is the Internet. A considerable part of grid research and its success is now related to web services [67]. There are many

implementations of these web-service based grid systems (e.g., Boinc or Chord) [45]. But, for now, we study the nature of these web services in the next section.

2.3 Cloud Computing

2.3.1 Overview

A cloud is an emerging IT delivery model, taking its place in the industrial scene. It seeks to enable users to gain access to their applications from anywhere, via any connected device [11]. VMware whose vSphere is positioned as the industry's first operating system for building an internal cloud uses this definition: "Cloud computing is the use of networked infrastructure software and capacity to provide resources to users in an on-demand environment" [12, p. 1]. Technically, clouds can be said to be virtual servers over the Internet. Clients can start and stop these servers or use compute cycles only when needed, often paying only for actual usage. Therefore, the cloud stands as a solution to some IT problems in companies by handling issues like security, data security and cost of infrastructure [25]. Cloud computing has both a business value and a technical value within an enterprise. A study in September 2009 performed by the Aberdeen Group found that some companies achieved on average an 18% reduction in their IT costs and a 16% reduction in data centre power consumption by adopting cloud computing [8]. In our research, we focus rather on the technical value by mentioning that it provides a way to increase capacity or add computing capabilities on the fly without investing in new infrastructure.
It is important to note that the concept of cloud computing is not essentially new; it is just the next evolutionary step from the advances in distributed computing, parallel computing, utility computing and virtualization (see Fig. 2.4) [58].

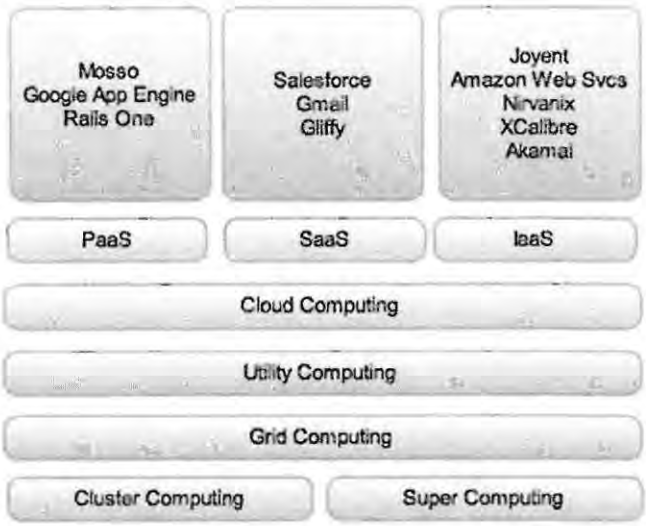


Figure 2.4: Computing stack.

2.3.2 Multi-processing systems

Multi-processing systems rose from the birth of two computational models. As described below, each of these is appropriate to a specific type of computer system structure.

- Parallel computation: this deals with tightly-coupled systems (e.g., mainframes) where multiple processors, often identical, are placed within a single ship. The connection is established at bus level.
- Distributed computation: this deals with loosely-coupled systems (e.g., grids) where single or multi-core processors in stand-alone computers are interconnected via a network connection.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems. But they require greater investments than loosely-coupled systems due to their low level of processor coupling and their lack of flexibility. However, due to the granularity of the problems, it is easier to abstract the complexity of a parallel computation than that of a distributed computation [59]. Distributed systems can scale easily (by simply adding more stand-alone computers) and are relatively cheap to build, whereas multi-core systems do not

scale (the number of cores in a single computer is physically limited) and are still expensive for end-users. This is why we are interested in large scale multi-processing systems (grid environments).

2.3.3 Need for virtualization

In short, virtualization is the abstraction of a computing entity (e.g., software, network, storage). Multiple virtual machines are created in the cloud by abstracting computing resources. Therefore, it is impossible to discuss cloud computing without knowledge of virtualization, which was developed as early as the 1960s at IBM for their mainframes. But, as computers became cheaper and their performance increased, mainframe virtualization was no longer needed. Nevertheless, in recent years, we have witnessed a comeback of virtualization in new areas of use. The implementation has been extended to client, personal and cluster computers to make them more capable than before [19].

Legacy OSs do not often provide appropriate facilities for *many task computing* (this denotes high performance computations comprising multiple distinct activities [55]). Because these traditional OSs focus on other features like intuitive interfaces or rapid development, there is no native support for many task computing (MTC) applications [51]. A definite workaround to this OS limitation is to use a virtualization technology to provide the needed isolation within the OS environment. Multiple applications can share physical resources without affecting one another. In this way, resources within an entity can be reserved for later use or dedicated to a specific use (see Fig. 2.5). This is particularly helpful in utility computing or on-demand computing.
Virtualization support is ubiquitously available in all relevant hardware architectures, which enables new programming and execution models for MTC applications. Since the isolation can be done at hardware level, the technology is well suited for a high performance (HP) or high availability (HA) environment [31].

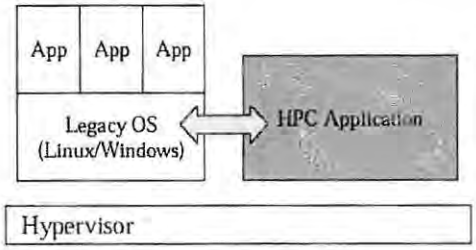


Figure 2.5: Architectural overview of a coexisting HPC application with a legacy OS [51].

Practically, virtualization can be applied in two generic ways. One way is by mapping a single physical resource to multiple logical representations or partitions (logical partitions or virtual machines), called server virtualization. The alternative is to use a number of physical resources to form fewer, yet more capable, virtual resources [11].

Virtualization techniques

There are multiple techniques for virtualization: emulation, full virtualization, para-virtualization, OS level virtualization, library and application virtualization. Here, we briefly describe the most common techniques.

Emulator. This allows a variety of unmodified guest operating systems to run on emulated hardware. A well known example is the open source project named QEMU, which emulates all the necessary hardware and provides libraries that translate machine code for the chosen system [26]. That is, with QEMU, an ARM computer architecture can be emulated inside an Intel x86 computer architecture.

Full virtualization. This technique simulates the complete hardware of the host machines. The guest OSs must be compatible with the host machine hardware, as opposed to emulation. An implementation of this technique is KVM (Kernel Virtual Machine), which provides "bare metal" virtualization on Linux [9].

Para-virtualization. This has a lot in common with full virtualization. The major difference is that the guest OS is allowed to know about the real resources and perform certain operations itself. For example, the guest OS can perform memory handling on its own. This enables para-virtualized machines to run faster. XEN is an implementation of this kind of virtualization [23].

OS level virtualization. This differs quite a lot compared to the previous techniques.

With this technique, there is no guest OS; a single OS manages guest and host applications by creating isolated environments for these applications. This is similar to operations used for client logins and sessions.

Use of Libvirt library

Practically, an external resource consumer is a VM, commonly referred to as a guest machine. Libvirt facilitates VM creation and usage by providing resource management and monitoring library calls [5]. It is a rich library for managing the virtualization capabilities of a computer node. Its API allows a variety of application designs by supporting different types of virtualization techniques (see Fig. 2.6).

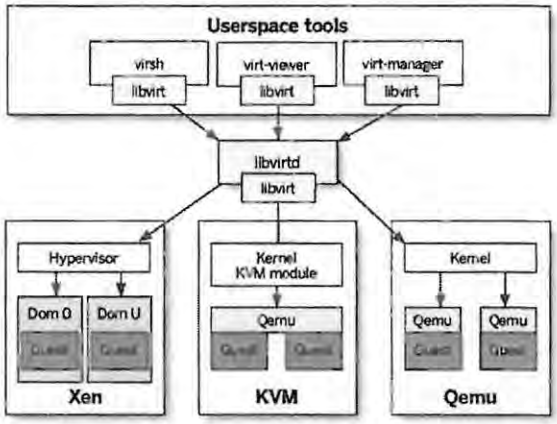


Figure 2.6: Libvirt architecture [14].

2.4 Cloud Services

2.4.1 Nature of cloud services

Based on the previous sections, it is obvious that cloud computing makes use of many existing technologies. One notable technology is utility computing, which raises the service oriented property in the cloud. Because of this property, the authors in [40] suggest that cloud

computing needs a standardised architecture like Service Oriented Architecture (SOA) and the interoperability and collaboration of Web 2.0. The user is a common agent in Cloud services and Web 2.0 services. Cloud designs are becoming more user-centred, thereby offering a ubiquitous web interaction.

2.4.2 Dynamic service composition

The ubiquitous computing trend suggests a dynamic interoperation of distributed services, via automated service provision or runtime creation of new applications. Given a service request, the runtime environment in response has to create a self-compliant solution deployment plan [47]. Of course, this dynamic service composition will depend on the selection of service providers with respect to parameters like availability, performance, load, monetary cost or quality of service (QoS). The existing service composition of P2P systems implements this, but struggles to re-compute a deployment plan during actual runtime execution.
Prinz et al. [54] proposed a service composition framework (SCF), forcing each sub-service to implement four methods that will ensure their execution within the framework: one to execute the service on the passing of parameters, a second to stop it, a third for checkpointing the service's current state, and a fourth to return the state. Peers that participate in the SCF network provide services satisfying these properties.
The SCF network enforces robustness during runtime execution using Logical Service Groups (LSGs). An LSG contains a set of available peers that locally provide a dedicated service. During the service execution, one peer of the LSG executes the service and n other group members monitor its heartbeat (see Fig. 2.7). Failures within the network are compensated by migrating the composite service; i.e., creating and applying a new deployment plan.

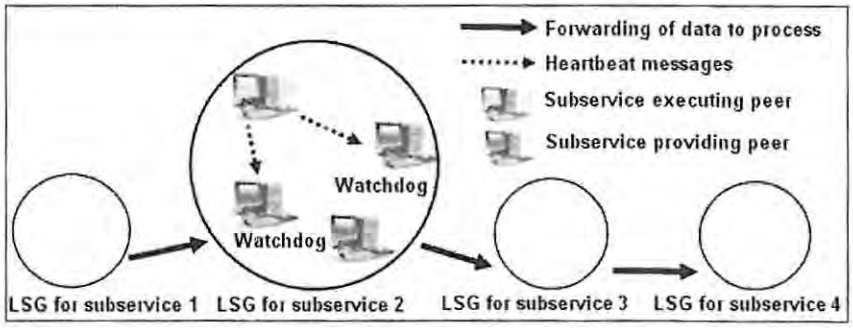


Figure 2.7: Service composition based on LSGs [54].

2.4.3 Type of cloud services

Conceptually, everything in the cloud is assumed to be a service (XaaS) [58]. The "X" reflects the wide variety of service delivery models. As Fig. 2.8 shows, there are several services a cloud can provide.

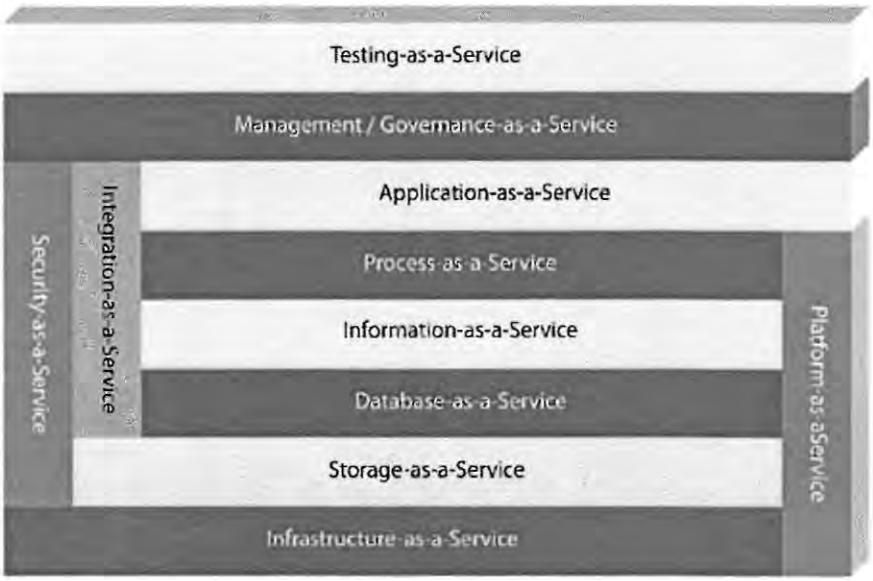


Figure 2.8: Discrete set of services within a cloud architecture [25].

Here we introduce the main ones.

- SaaS (Software as a Service): In this case, the provider allows the customer to use only its applications. Some examples are email services provided by Yahoo or Gmail. There is no need to install software or a server to use these. One only needs an Internet connection to use the email application residing on the providers' servers. Looking at cloud computing from this perspective, we realize that it has existed for longer than we think.
- PaaS (Platform as a Service): A complete platform is provided as a service; that is, a set of software and development tools hosted on providers' servers. Popular applications of this service are programming and database platforms. The user can only run services supported by the platform. An example is Google Application Engine, which provides a set of Python APIs that enable developers to create their applications conventionally.

- IaaS (Infrastructure as a Service): Also called data centre as a service or hardware as a service, this is a coarse grained cloud service. This service provides the ability to remotely access computing resources. The latter constitutes the user's data centre where the desired services or applications can be hosted. A famous example is the Amazon Elastic Cloud (EC2), which provides virtual servers with unique IP addresses and blocks of storage on demand [53].

Traditionally, cloud infrastructures are massively scalable data centres where computational resources can be dynamically provisioned and shared. The strength of a cloud resides in its ease of management through operations like automatic provisioning, re-imaging, workload re-balancing or monitoring [11]. Consequently, cloud systems are designed in a client-server model, whereby they take advantage of this centralized control over the infrastructure. Moreover, they make computing services available to the common user (with minimum knowledge of computer science required).

2.4.4 Overview of the Amazon Elastic Cloud

The Amazon Elastic Cloud (EC2) is a cloud platform that allow users to create virtual computers on which they run their own applications. The platform is based on Xen virtualization. With this, the EC2 provides the following features: automated scaling, Amazon CloudWatch (realtime monitoring of VMs) and elastic IP addresses (an IP address can be mapped to any virtual machine instance) [46]. In this respect, Amazon uses an IaaS delivery model, which is part of the Amazon Web Services (AWS) stack. It is considered the biggest cloud for housing IaaS services. Amazon EC2 has a simple web service interface that allows users to obtain and configure their computing resources. Two web APIs, SOAP and REST, are also available to the developer to interact with its services.
The EC2 implements utility computing; users pay only for the computing capacity they actually use. Billing is hourly per virtual computer. Amazon charges $0.027/hour for the smallest virtual machine running Linux. The EC2 is mostly used by developers to deploy their web applications.

2.5 P2P Computing and the Cloud

Recent developments have given birth to a new distributed computing paradigm: P2P computing, which reflects the sharing of computing resources in a P2P environment. A better term for this is volunteer computing, which represents a type of distributed computing in which computer owners donate computing resources (e.g., CPU cycles and storage) [57]. We have noted the wide use of P2P systems for storage or information sharing (e.g., Napster). Such systems belong to the set of high availability computing (HAC) systems. Another set of computing systems in which P2P computing is involved is the set of high performance computing (HPC) systems. With respect to this, P2P computing can be regarded as applying P2P techniques (described in Section 2.1) for HPC or HAC. In other words, it simply refers to using a P2P platform to solve a computing problem [69]. This is how desktop grids are built. In this case, peer nodes are worker nodes, which solve independent tasks (see Fig. 2.9). Of course, it is possible for these individual tasks to be working together on a global problem.

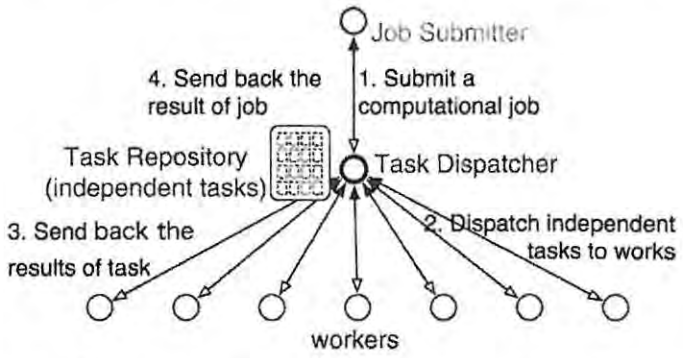


Figure 2.9: General architecture of P2P computing platforms [69].

The P2P paradigm has emerged as an alternative way to build scalable grids (grids with many sites) [20, 65]. It enables the creation of a grid with the capability to provide more computing power than any supercomputer or cluster. Moreover, such a grid does not have single point of failure (elements fail independently), which provides a "fail safe" potential in the system. Now, it has been stated that if a grid can be built, there is the potential for offering cloud services (see Fig. 2.4). We look at the feasibility of this.

2.5.1 Bridging P2P computing and cloud computing

In contrast to traditional P2P systems, clouds required a well managed infrastructure in order to provide suitable services. Popular P2P computing systems like [1] are embarrassingly parallel and aim to solve a single computational problem [35]. This explains their success in scientific computations or grid computations. In these computations, the peers never communicate with one another, but only with a server. So the true power of P2P is never truly harnessed. Therefore, many researchers [2, 43] believe that we can intelligently bridge the two computing paradigms of P2P and cloud to get the best of both worlds. This combination could lead to user hosted clouds that are cheap to operate and use and could encourage a free exchange of resources.
Cloud computing shares a few similarities with P2P computing specifically in terms of providing dynamically scalable "over-the-Internet" resources. So, a hybrid system should provide a consistent, well managed, yet scalable computing environment. Let us examine the essence of this idea through its potential advantages and disadvantages.

Advantages

- Improved reliability over the traditional (client-server) cloud, in the sense that it is natively an on-demand platform.
- Cost effective, because it removes the need to build expensive data centres.
- Easy scalability.
- People who offer their idle CPU times may get to benefit commercially and an ecosystem of P2P node providers will develop around every providers infrastructure [64].

Disadvantages

- Security and privacy issues are magnified in this situation.
- Ensuring the predictability of nodes is a problem.
- Traditional network routing may need to be re-conceived to handle the extra traffic.

- Management is another difficult problem.
- Lack of control compared to the client-server architecture could make suitable services difficult.
- Regulatory issues could be a big deterrent as the P2P cloud grows [64].

Possible difficulties

From these a priori advantages and disadvantages, we discern the issues to be addressed in this research.

Transparency. We have already mentioned the importance of seamless service creation and convenient service interaction. The service should be easily accessible ("point and click" service access). Through the term transparency, we emphasize the fact that when a user runs a task remotely through our service provision system, s/he should be comfortable doing so. We identify two steps here.

1. In a best effort, find a suitable host for the intended service (cloud/grid computing).
2. Provide a workspace abstraction for the user (ubiquitous computing).

Reliability. Although issues like bottlenecks or single points of failure are not of great concern in P2P systems, the permanent node churn induces a notable risk for hosting any external computation. The way this risk is dealt with defines the reliability of a service, and the system as a whole. We identify two related factors.

- Fault tolerance: accidental failures may occur on a host.
- Volunteer based networking: nodes can join or leave the system freely.

A recent attempt has been made by Graffi et al. to tackle these issues. They suggest an algorithm for long term resource reservation [43]. This has been proven to be successful for long term leases (e.g., 100 days), but loses its efficiency for short term leases. The algorithm is based on experimental research documented in [63], which proves that older nodes are more reliable in a P2P network.

On the another hand, the authors in [57] claim to provide a solution for the reliability problem through a simple predictive algorithm, which is used for deploying Java based services on a P2P grid architecture. Basically, this algorithm uses the *availability data* of a group of nodes to predict the availability of a node in that group.

2.5.2 Self provisioned clouds

A self provisioned cloud is another way of expressing the nature of a P2P cloud. Self provisioned clouds are those that are self-sustainable. They manage the needed computing infrastructure on their own. In addition to this, they perform an automated provision of computing services. The next two sections present, respectively, what the service provision process and the infrastructure organization entail.

Service oriented computing (SOC)

In [45], the key aspects of SOC are defined as follows.

- Service publication: phase in which a service description is created and sent to others.
- Service discovery: the process of discovering services that meet certain requirements.
- Scalability: how well the infrastructure behaves in a bigger scale, with many service providers and consumers.
- Resiliency: reflects the reliability of the infrastructure for the publication, discovery and provision of services. The infrastructure should be resilient to failures.

A survey on P2P architectures for SOC is given in [45], while Table 2.1 summarises the findings.

	Semi Centralised	Unstructured	Super Node	Structured
Service Publication	**GOOD** *Reliable service location.*	**POOR** *Services may fall out of range.*	**POOR** *Services may fall out of range.*	**EXCELLENT** *Efficient and reliable service publication.*
Service Discovery	**GOOD** *Search via indexing server.*	**GOOD** *Inherent discovery through flooding search.*	**GOOD** *Inherent discovery through flooding search.*	**NONE** *No support for service discovery.*
Scalability	**VERY POOR** *Requires big, costly servers.*	**POOR** *Flooding search does not scale.*	**GOOD** *Improved, but un-scalable.*	**EXCELLENT** *Efficient, scalable lookup.*
Resilience	**VERY POOR** *Failure of indexing server affects all nodes.*	**EXCELLENT** *Resistant to failure and attack.*	**GOOD** *Less resistant to failure of supernodes.*	**EXCELLENT** *Resistant to failure and attack.*

Table 2.1: P2P architectures for service oriented computing [45].

Virtual organization clusters

In [52], a novel architecture is presented for overlaying dedicated cluster systems on existing grid infrastructures, named Virtual organization clusters (VOC). These are basically virtual clusters constructed to satisfy the need of a particular organization. An organization is a group of people with a shared mission. Virtual clusters are clusters built using VMs.
The concept of VOC is a high level solution to the problem posed by implementing cloud like services on P2P like systems. The idea is to permit transparent service deployment by making use of existing grid middleware. VOC achieves this by establishing a separation of virtual administrative domains, where each domain is subject to user (organization) needs. Thus, it represents an independent policy workspace where decisions made by the administrators of the domain are not subject to external influence. The virtual administrative domain (VAD) is implemented on top of a physical administrative domain to provide a homogeneous interface suitable for grid work. The VAD consists of a set of virtual machine images for a single virtual organization (VO). With this abstraction, the needed transparency is clearly established (see Fig. 2.10).

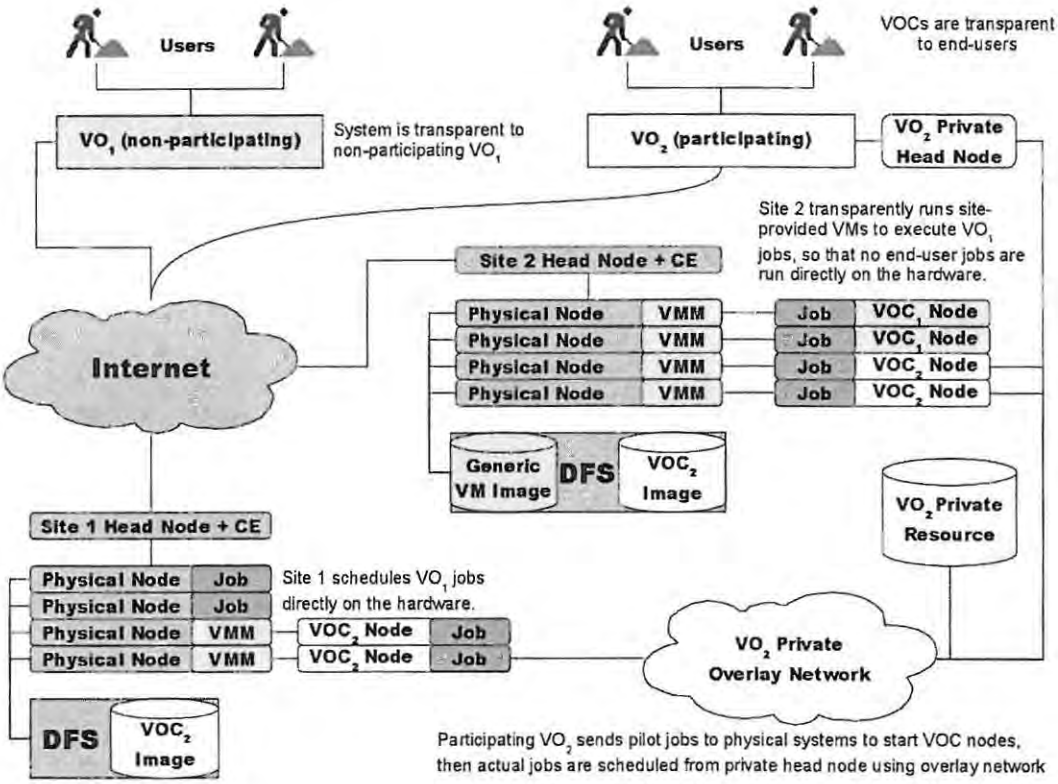


Figure 2.10: Virtual organization clusters on a grid system [52].

For high throughput computing (HTC) and MTC applications, VOCs can provide cloud environments. These clouds are autonomously managed by middleware. This implies that software agents manage, configure, and repair the virtual clusters without human involvement. Of course, the VOC will have to be customized for this. A few resource reservation approaches, like [52, 43] have been developed, which could complement the VOC.

To provide its functionality, VOC assumes a grid as the computing bed and uses IP Over P2P (see Section 2.1) for its clustering middleware. Because of the objectives of VOC as set out by its authors, its implementation and hence its overall approach differ from those we present later in this thesis. However, though their approach does not fully embrace the P2P realm, it is a step towards the type of P2P cloud computing bridge defined in Section 2.5.1.

2.5.3 Clouds@Home

As stated before, some research has been done on the idea of a P2P cloud. Clouds@home is the latest proposal in this respect. It suggests an ecosystem of clouds by involving open (e.g., OpenNebula) and closed clouds (see Fig. 2.11). Our idea is in complete agreement with that presented in [2]. But at this point, Clouds@home is a specification, not an implementation. As the previous section suggests, the reliability criterion is critical. Appropriate clustering (node and resource advertisement) and scheduling (resource discovery) algorithms are needed. The chosen algorithms need to be practically discussed and implemented. This is lacking in the Clouds@home proposal.

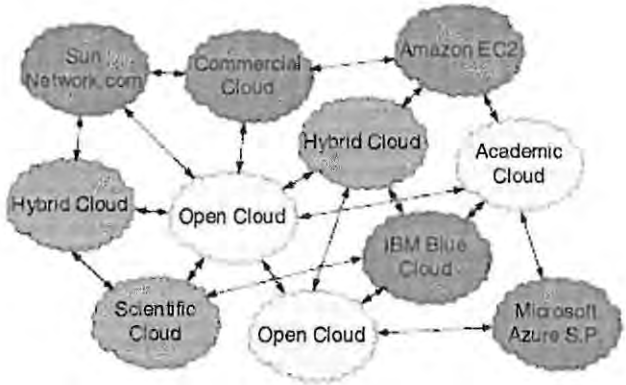


Figure 2.11: Clouds@Home scenario [2].

2.5.4 Boinc

There is an active system implementation that relies on volunteer computing to provide a usable computing platform, namely, the Boinc computing system. It achieves its potential computing power by exploiting the CPU cycles of thousands of machines that are connected to the Internet. An example of a project run by Boinc is SETI@home. The latter performs a distributed computation on astral data to find signs of extraterrestrial intelligent life [18]. Boinc's current implementation only supports embarrassingly parallel applications and follows a master/worker model, with no communication or coordination between clients.
Recently, Costa et al. [35] proposed a cloud system implemented using the Boinc middleware: MapReduce over the Internet (BOINC-MR). MapReduce is a software framework for parallel

data intensive computations. It also embraces applications that can be reduced to map and reduce operations. BOINC-MR is a cloud environment for MapReduce job execution.

2.6 Summary

To obtain knowledge of the tools required for this research, we carried out a critical study on clouds and P2P systems. In this chapter, we touched on specific topics, identifying issues that should be considered for any P2P cloud implementation. This chapter essentially covered three main sections: P2P computing, Cloud computing and the combination of both. Techniques used in P2P systems (e.g., clustering methods) and those used in clouds were presented separately. This allowed us to pinpoint the area of interest in each paradigm and to suggest a bridge between them. Examples of systems like Boinc were mentioned to illustrate our discussions.

With this background study, we have the necessary knowledge to build and present our work. We start by presenting a framework for P2P clustering.

Chapter 3

Prototyping an Autonomous P2P Framework

An autonomic P2P system is a self sustainable and dependable system that allows seamless peer interactions or collaborations. Towards our implementation of a P2P cloud, a framework is designed to build such a system. The framework ensures that a P2P system exhibits some basic properties such as failure recovery and minimal bandwidth overhead, necessary for using a dynamic network. A P2P network is dynamic in the sense that it changes in size and topology over time due to the churn of its nodes. These nodes are interconnected via a certain transport protocol (Wi-Fi or Ethernet), using pre-established networking structures (e.g., IP routing). In other words, the network is essentially ubiquitous. Hence, a P2P framework needs to be potentially implementable and applicable with generic networks. For instance, a resource querying protocol would practically operate at a higher level (above the transport layer) in the OSI stack. So, the algorithms developed within the framework should be indifferent to the transport layer protocol and be able to execute without any assumptions thereof. In this way, ubiquitous network access is supported in an application that implements the framework. We touch on some of the network assumptions in the next section.

3.1 Design Considerations

To represent a network at hand, most algorithms assume a graph G consisting of a set of nodes C and a set of vertices V. Then they derive a dominating set S based on G such that S$\in$ G (see Section 2.1.1). But, the hostility of the network (indifference to the transport layer as described previously) makes it practically impossible to ensure the correctness of graph G. For instance, network routes are not rigid in IP networks; hop counts between two nodes may vary. Effectively, with non-sensor networks, there is no means to predetermine exactly how nodes are interconnected, or the distances (hop count) between those nodes. As a result, an algorithm that builds a dominating set by assuming a graph representing the complete network cannot be considered.

In our approach, we stipulate that every node has unique qualifying properties and that a root node, in addition to these properties, has an identifier describing its position as the root. The root position is of course the highest position in the hierarchy (tree), and any node is potentially capable of occupying this position. The root node is referred to as the super node in some algorithms. Note that it is possible to use multicast messages to find such a node. However, this is not recommended for two reasons: *n-casting* may not be supported and flooding should be avoided in the network. So, as suggested, we rather use a global variable that can uniquely identify the root node throughout the network. Because of the adhoc nature of this network, the value of this variable (e.g., the root IP address) is obviously subject to change. This is why in our design, dynamic DNS (DDNS) capabilities are leveraged. DDNS allows DNS entries (e.g., DNS name to IPv4 address mapping) to be dynamic. These entries can be changed by sending DNS update packets to the DNS server. In this design, the global root variable is represented by a DNS name. The choice of DNS is also supported by the fact that it is a well established and widely used protocol.

In our review of P2P frameworks (see Section 2.1), we identified some key aspects common to these frameworks. To cope with the effects caused by the dynamic nature of P2P networks and consequently to achieve our aim, a P2P cluster must satisfy the following properties:

1. Every ordinary node has at least a site head as a neighbour (dominance property).
2. Every ordinary node is affiliated with the neighbouring site head with the greatest relevance.

3. No two site heads can be neighbours (independence property) [24].

In the next section, we propose a clustering algorithm and show how it facilitates resource querying in an unstable network. To complement the framework, the scheduling algorithm used for resource discovery is also presented.

3.2 Clustering Component

The clustering method is mainly subject to the node dynamics in the network it sets out to cluster. This infers the need for a reactive algorithm to cope with such an environment. Nevertheless, to improve the speed of the overall clustering process, the clustering algorithm needs to perform proactive operations. Connected dominating sets (CDS) are convenient for this approach. Though they have be proven only to be successful in adhoc sensor networks, we believe that the resilience a CDS infers in the network structure can be leveraged. The CDS represents an overlay networking infrastructure on the underlying P2P network. Each node in the CDS has a table of information about the nodes it supervises. This table is updated autonomously with respect to changes in the network.

3.2.1 Proposed clustering algorithm: H-DCA

There is neither a fixed (or predictable) topology, nor central administration in an adhoc system. Nevertheless, these are ideal features for a dependable system. A best-effort algorithm can be used such that the adhoc system experiences the advantages of a centralized system. The algorithm is executed distributively, running independently on individual nodes to tolerate node churn and to enforce intelligent behaviour in the network. In this section, we explain the H-DCA (Hierarchical Distributed Clustering Algorithm) used to construct a hierarchical CDS in order to achieve the desired behaviour. The algorithm is essentially *weight based* and *zonal based.*

Weight based A weight is associated with each node; the bigger the weight, the greater is the chance of the node being a site head. This is a simple criterion to quantitatively

distinguish nodes belonging to the same site. Because stability is an important factor for routing, one way to weight a node is to use its *uptime* value (how long it has been online).

Zonal based Geographical or localisation information about a node is relevant to the algorithm. The H-DCA divides the network into zones, by organizing them in a DNS-protocol like hierarchy. The DNS-protocol partitions the network into zones and subzones to localize nodes and to process DNS lookups accordingly. Our hierarchical approach is inspired from this well defined and widely implemented concept. A zone is defined as the subnet in which a node resides. However, a subnet itself may not be practically well defined. Depending on its size, the subnet may contain other subnets (sub-subnets). In this case, defining a zone is not straightforward. Though this concern is raised here, it really only matters at the implementation stage.

For simplicity, the term "zone" is replaced by "site" from now on. A site denotes a group of nodes in a particular geographical location.

For each site, the H-DCA determines a site head, which is the *dominator* for that site (according to the *weight based* criterion). A collection of site heads (head nodes) constitutes a dominating set, the backbone of the network (see Fig. 3.1).

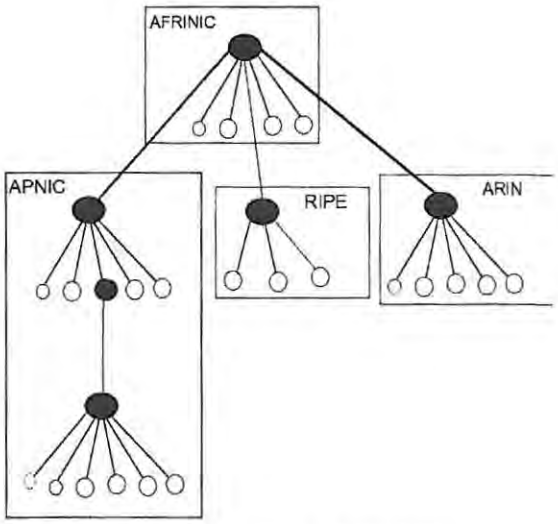


Figure 3.1: Tree of clusters.

It should be noted that the H-DCA proceeds in a top-down manner; i.e., it uses iterative deepening. A new node makes first contact with the root node (using its globally known identifier). Then the latter, as a parent node, replies with information on relevant child nodes. Then, the new node contacts these nodes (now considered as its parent nodes) and the process repeats. Because the algorithm is also mostly proactive, new nodes follow a pre-defined tree structure and only update branches relevant to themselves. During this construction of the cluster tree, convenient rules are applied as presented in subsequent sections.

Contrary to other algorithms, those given in [15, 68, 24] for instance, a node does not know about its neighbours unless it is a head node. This is done to reduce the communication traffic in the network. That is, node information is only stored and updated where needed, thereby reducing bandwidth usage, the number of messages, and time complexity. As far as each node is concerned, it is:

- responsible for itself; i.e., it is self-managed;
- aware of its parent node (its supervisor); and
- responsible for nodes beneath it; it possesses knowledge (e.g., shared resources) about them.

These are the qualitative properties that define a peer node in the framework. From this introduction of the H-DCA, we also identify some quantitative properties, which are described in the next section.

3.2.2 Constraint variables

In the following, we describe the basic features that each node possesses and shares in order to join the cluster tree correctly.

Closeness Nodes that are close to one another should cluster together. In this respect, the closeness variable represents localisation information about a node (e.g., network range to which the node belongs). We define two nodes with the same network range to be neighbours (i.e., they belong to the same subnet or site).

Centrality degree This is the number of nodes a site head supervises. It is initialized to one, counting the node itself. This value may be used to quantify the responsibility of a node. Nodes with a higher degree of centrality are more important to the network. Effectively, this characterizes the value of the knowledge contained in the site head. The degree of centrality can help to appreciate the site head churn effect on the global network. One could suggest that where this effect is substantial, the knowledge should be duplicated or further distributed to capable nodes.

Uptime This is a time value defining how long a node has been active. Determining nodes with higher *uptime* values helps to ensure stability in the network. It is initialized to zero, as the node would have just joined the network.

Algorithm 1 is used for the initialisation and update of the constraint variables.

Algorithm 1: Initialize and update constraint variables.

Input: Closeness, Degree, Uptime
Let $Count()$ returns the number of nodes being supervised by me.
Let $T_0 = Time(now)$ the starting time of node.
PROCEDURE $Init$;
Closeness = $NetworkRange()$;
Degree = 1 ;
Uptime = 0 ;
PROCEDURE $UpdateVar$:
begin
  if $Closeness == NetworkRange()$ **then**
    Degree = $Count()$;
  end
  else
    Closeness = $NetworkRange()$;
    Degree = 1 ;
  end
  Uptime = $Time(now) - T_0$;
end

3.2.3 Algorithm workflow

The H-DCA is message-driven; that is, it uses an exchange of messages (requests or responses) between nodes to drive its execution. Considering node dynamics within the network, this is

appropriate. The exchange of messages is typically found in three procedures: *WhoIsHead, BeHead, Ping*, which are used, respectively, to request information from a parent node, to inform a parent node about a node's role, and to verify the link to a given node. These procedure calls are used in *Algorithms 2, 3, and 4*. It should be noted that *Ping* is run as a permanent process to ensure realtime link failure detection. It operates with "keep alive" packets that are sent periodically to determine the link's state. A link is represented by a branch in the cluster tree.

The next two sections present the two phases into which the algorithm can be logically separated. These phases can be inclusive (i.e., one phase uses the other during its execution).

3.2.4 Clustering setup (*Algorithm 2*)

This is the initial phase of the clustering process. A node joins the cluster tree in this phase. That is, the phase completes when a suitable position in the tree is found for the new node to occupy. The algorithm exhibits proactive actions by providing heuristics to ensure completeness is achieved. It can also be used when a re-clustering (full or partial) is necessary. In the case of a partial re-clustering, variable i in Algorithm 2 is not equal to zero. The starting point for the clustering setup is as follows: once online, a node executes *Procedure Init* and then makes contact with the root node. Next, it applies the following rules:

- **Rule 1** A node that has newly joined the network cannot and will not have more relevance that older ones (parent nodes). Consider an arbitrary node v. If there is no response to a request made from v to a site head, then v becomes the site head for that site. Node v inherits the global root identifier if it is the first node in the tree.
- **Rule 2** Consider an arbitrary node n and depth i in the tree. If a site head exists for i, then n requests the address of the next head in the branch relative to its position (see closeness constraint in Section 3.2.2). The rule is executed iteratively from the root level moving down the tree. This iterative deepening process was mentioned earlier, in the H-DCA introduction.

Though procedures *WhoIsHead and BeHead* are not defined in Algorithm 2, their roles are contextually explained below. Procedure *TraverseTree* is also described.

Algorithm 2: Clustering setup.

Input: Closeness
Let n_i denote a node in the tree,
where i is its depth in the tree. E.g., n_0 is the root node.
PROCEDURE *WhoIsHead*;
PROCEDURE *BeHead*;
$i = 0$;
PROCEDURE *TraverseTree* :
begin
  while 1 **do**
    if $(n_{i+1} = WhoIsHead(n_i, Closeness)) == NULL$ **then**
      BeHead(n_{i+1});
      break;
    end
    else
      $i++$;
    end
  end
end

- *WhoIsHead:* This is used to identify the head node at a given depth i in the network tree. If $i = 0$ then the head node is the root node. In this case, the procedure uses a DNS lookup query to resolve the root DNS name (this is the global root variable introduced at the beginning of this chapter). Otherwise, the procedure requests information about a relevant head node at depth i, with $i > 0$, from the head node at depth $i - 1$. Of course, the node at depth i, with $i > 0$, is a child to a node at depth $i - 1$.

- *BeHead:* The procedure is used by a node to claim a head position in the tree. Since the first position ($i = 0$) is the root position, a node claims it by using a DNS update query. The other positions ($i > 0$) are claimed by sending a *join* message to the previous ($i - 1$) head site. The *join* message carries the clustering constraint variables of its sender.

- *TraverseTree:* This makes use of the two previous procedures. As shown in Algorithm 2, using *WhoIsHead*, this procedure traverses the cluster tree to find a suitable position for a node. Then, the position is claimed using *BeHead*.

Fig. 3.2 depicts three iterations of the execution of the clustering setup algorithm.

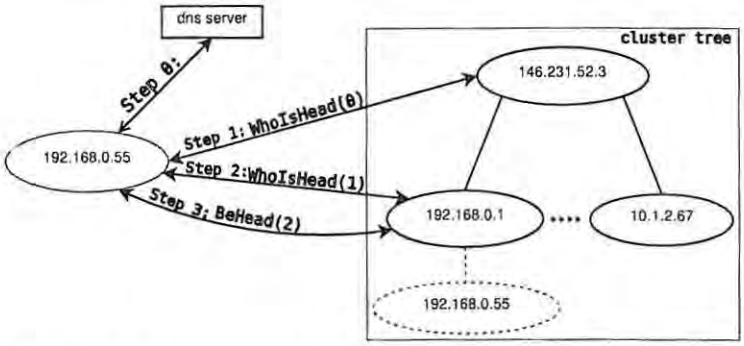


Figure 3.2: A new node joins the cluster in three steps.

3.2.5 Clustering maintenance (*Algorithms 3 and 4*)

The maintenance phase is used in two situations.

- To improve the connectivity and stability of the network tree. The stability of the cluster tree is improved by changing site heads with more suitable (reliable) ones whenever possible.
- To fix the tree in case of link failure (required re-clustering). Consider the case of link failure whereby the state of a node changes (i.e., typically changing its position in the tree). The algorithm accommodates the outage by partially re-clustering where the change has occurred. The effects may escalate to a larger scale depending on the node's position in the network tree at the time of failure.

Due to events related to these situations, the clustering maintenance algorithm is composed of reactive actions. Here, we assume that the algorithm has the means to detect a failure in the network. That is, the *Ping* procedure (introduced in Section 3.2.3) is only a prototype. However, a *Ping* algorithm is proposed later in Section 3.5.3.

This phase complements the first one (clustering setup phase) and is defined by the following rule set.

- **Rule 1** Nodes, whose variables are older than a predefined period, are disregarded; i.e., information about the node is no longer valid. After joining the tree, a node should contact its site head periodically to prove its availability and/or to update its information.

- **Rule 2** If an arbitrary node v loses its link with its site head then it is responsible for recreating a new one, possibly with the new site head (Algorithm 3 reflects this rule). This can be done by contacting an ancestor node in the branch and traversing the tree from that node. This obviously requires the node to cache the ancestor nodes in its branch. This can be done during the clustering setup phase. If there is no available ancestor node then the re-clustering phase is equivalent to the clustering setup phase.

Algorithm 3: Link failure handling.

Input: Closeness, Degree, Uptime
Let d be the depth of an arbitrary node in the tree.
while *(Ping(H_{d-1}) == FALSE) AND ($d > 0$)* **do**
    $d--$;
end
UpdateVar;
TraverseTree(d-1);

- **Rule 3** A node can challenge its site head and takes its position if its *uptime* value is significantly larger; i.e., it is more stable (Algorithm 4 reflects this rule). The former site head hands over its knowledge to the new head and informs all its site nodes about the change. This rule is rarely applied because it is mainly used to cope with race conditions that may arise with *BeHead* messages.

Algorithm 4: Site head update.

Let j be the depth of an arbitrary node in the tree.
while 1 **do**
    if *($j! = 0$) AND (Challenge(H_{j-1}) == TRUE)* **then**
        BeHead(H_{j-1});
        $j--$;
    end
    else
        break;
    end
end

3.2.6 Theorem

Given an adhoc network, the H-DCA will act on it such that adhoc network properties as listed in Section 3.1 [24] are satisfied.
Proof: We show from the nature of the algorithm (definitions) and its properties as described in the previous section that the adhoc clustering properties are always satisfied.

1. If a node shares the same *closeness* value (same network address) with its parent node (i.e., they are neighbours) then it follows from the structure of the tree that the latter is its site head. Else, the node is its own site head as it does not have a neighbour by definition (see *closeness* description in Section 3.2.2). The dominance property is therefore guaranteed.

2. Consider a set S of neighbouring nodes, where $P \in S$ is the parent node. From Rule 1 in Section 3.2.4) which assigns a site head and Rule 3 in Section 3.2.5 which allows a site head replacement, it follows that any $v \in S$ only communicates with the neighbour of highest relevance; P in this case.

3. From the hierarchical based structure of the cluster tree, two parent nodes with network ranges T and R, respectively, such that $T = R$ do not exist. That is, there cannot be two or more parent nodes that are neighbours. This satisfies the independence property.

3.3 Scheduling Component

The scheduling component's role is to perform resource information management within the framework. This component is crucial as knowledge in a P2P system is natively distributed. Indeed, a P2P system is characterized by its capability to aggregate distributed resources for a given task. Irrespective of the resource provider, each resource should be meaningful to the system at any time. That is, the system needs to possess updated knowledge about all the resources at its disposal in order to make informed decisions about those resources. This is facilitated by the overlay structure established with the clustering component. Effectively, the cluster structure created by the H-DCA is robust by nature, making a P2P network open to generic peer resource management techniques. For example, DHT algorithms can

be implemented. But the scheduling technique needed in the framework is intended for computer resource service provision. So, adequate scheduling tools need to be developed and used accordingly. Resource advertisement and querying are the basic tools involved. These are presented in the following subsections.

3.3.1 Resource advertisement

Resource advertisement allows the framework to collect and update resource information in the P2P system. It is a distributed operation whereby nodes individually advertise their shared resources. The operation is executed to inform the network about a newly available resource or about changes to a known resource. In this way, the system is kept globally informed. Note that information is represented by an advert. So, an advertisement is contextually defined as the process during which a node sends its resource information to its parent node (supervisor). A protocol called $IPPROTO\_IPP$ was created to perform this, where IPP stands for information policy protocol. Details on the implementation of this protocol are provided in the next chapter.
In the protocol, the advertisement is carried out by sending a message. The payload of the message has the information on the shared resource(s). This payload is therefore the advert. The latter is created when a resource is available or has changed properties. For example, a text that says "ram: 1024" is an advert for shared random access memory. On receipt of this advert, the head node responsible for the advertising node creates or updates the corresponding resource information. This information may be represented as a table entry in a database.
Basically, $IPPROTO\_IPP$ defines how an advert is sent between a node and its head node. Fig. 3.3 illustrates the skeleton of an $IPPROTO\_IPP$ message.

3.3.2 Resource discovery

First, it is important to remember that resource discovery is performed in a decentralized P2P platform, represented by a cluster tree. So, requests for resources need to be dispatched conveniently for the exploration of the P2P network. A discovery is practically achieved with the aid of a query (or request) initialized by a peer node. The query is satisfied when the

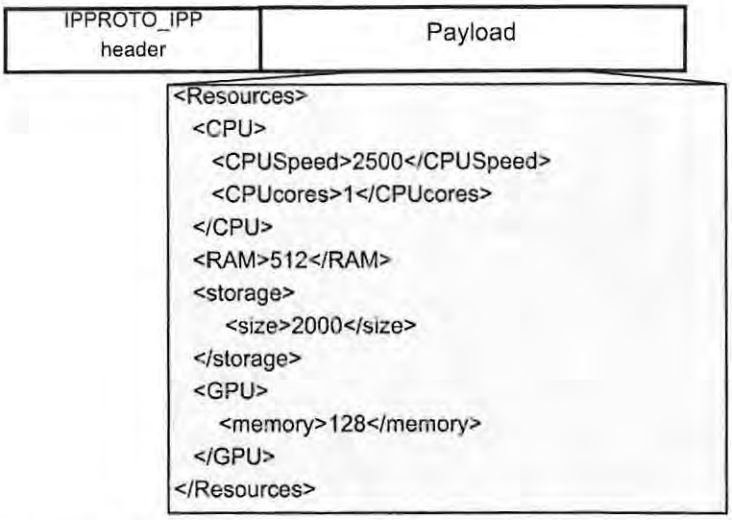


Figure 3.3: Sample advert message. XML is used in the payload for readability. Our implementation of *IPPROTO_IPP*, however, does not use XML.

resource description it contains is matched. Peer nodes listen for incoming query messages and contribute to their broadcast across the network by forwarding them to selected neighbours. Note that each node publishes its resource information on its head node, except the root node (it is its own head). For local access, each node also keeps a copy of the same resource information. Therefore, a node may provide information about one set of resources (e.g., local resources) or multiple sets of resources (e.g., all resources shared by the nodes it supervises).

Let us contextualize the resource discovery operation for computing resources. The discovering process used in this work is essentially a capacity-based search. That is, our discovery mechanism aims to find a node capable of hosting a particular computing service. With this in mind, we define some basic constraints.

Constraint variables

Resource queries have multiple attributes. One set of attributes is the computing resource requirements. These are constraints on the resources needed for a prospective computing service. But here, we discuss a different set of attributes that are native to our framework.

Uptime constraint The *uptime* variable is explained in Section 3.2.2. The constraint stip-

ulates merely that nodes with higher *uptime* values have preference; i.e., their resources are favoured. In fact, Graffi et al. [43] show that older nodes have lower probabilities of failure. They state that the probability density of the availability of nodes follows a Weibull distribution with parameters: *scale* $\lambda = 169.5385$ and *shape* $k = 0.61511$. The Weibull distribution is a continuous one. That is, if the variable of the distribution is a "time-to-failure", the failure rate is proportional to the power of time [70].

Timestamp constraint We introduce a *timestamp* variable to age resource information. The constraint that stems from this variable stipulates that resource information should not be outdated. For example, a *timestamp* value which is 100 seconds behind the current time is an outdated value. This suggests obviously that the corresponding information is no longer valid and cannot be used. Indeed, outdated information is caused by the fact that the responsible node cannot be reached (either it is offline or there is a link failure).

Algorithm workflow (*Algorithm 5*)

The discovery algorithm is a best effort one in that it returns the first node that satisfies the requirements in a query. Nodes formulate queries with respect to user needs. If a node cannot satisfy a query locally, it contacts other nodes in the tree by forwarding the query appropriately (upwards and/or downwards depending on the depth of the tree and the role of the node). For instance, if the originator of a query is the root node, the tree traversal is similar to an iterative deepening process. The algorithm execution is greedy; a query is forwarded through the most "healthy" edge, where "healthiness" is related to the *uptime* constraint.
If a node has information matching a forwarded request, it sends back the response through the forward path. Note that the response is not and cannot be sent directly to the node from which the request originated because each node in the forward path acts as a proxy. In other words, the forwarding node rewrites the query header and sends the query on behalf of the preceding node. This is enforced by the hierarchical structure of the cluster tree.

Algorithm 5 shows the discovery process when a query is received. For conciseness, procedures *Node, Match and Dispatch* are not defined in the algorithm. Their roles are however explained below.

- *GetNode:* This procedure returns the address of the corresponding i^{th} node. $i = 0$ and -1 are reserved values, where $i = 0$ refers to the node itself and $i = -1$ refers to its site head. All $i > 0$ indexes refer to child nodes. These nodes are ordered by priority using their *uptime* value. For instance, $GetNode(i)$ with $i = 1$ returns the most available (highest *uptime* value) of the child nodes.

- *Match:* this is responsible for processing a query. It completes successfully when the query can be satisfied by a specific peer node. The address of this node is returned.

- *Dispatch:* this implements a distributed tree traversal (exploration). Essentially, it re-routes queries to other nodes in the cluster tree. Obviously, it checks that the query does not take the same route twice; i.e., the traversal is acyclic.

Algorithm 5: Resource discovery.

Input: query
Let d be the depth of an arbitrary node in the tree., and c its centrality degree.
Let $i = 0$ be the index of a node its supervises.
Let *host* be a variable to hold a node's address.
PROCEDURE *GetNode*, *Match*, *Dispatch*;
if *(host = Match(query))* $!=$ *NULL* **then**
    return GetNode(i);
end
else
    $++i$;
    while $i < c$ **do**
        if *(host = Dispatch(query, GetNode(i)))* $!=$ *NULL* **then**
            return host;
        end
    end
end
if *(d!=0) AND ((host = Dispatch(query, GetNode(-1)))* $!=$ *NULL)* **then**
    return host;
end
return *NULL*;

3.3.3 Resource reservation

After being discovered, resources are reserved for hosting a prospective service. Of course, the reservation agreement expires if the corresponding resources are not claimed in time. The expiration time is defined by the type of reservation. The latter is itself defined by the type of service. That is, the reservation depends on a pre-defined start-up time for the service. With respect to this, there are short term and long term reservations. Our implementation of the framework focuses only on short term resource reservations. It deals with services that start immediately. Small reservation times are used to deal with node dynamics and in fact, these help avoid race conditions (e.g., a resource is allocated to multiple services simultaneously, because the corresponding requests were processed concurrently).

Nonetheless, an implementation for long term reservations can be realized by using the work discussed in [43]. Details on how this work can be leveraged are provided in Section 3.5.4.

3.4 Formal Analysis of the Framework

The techniques used in the framework were chosen due to their efficiency. For instance, the H-DCA was developed to reduce information update overhead, and optimize use of the network bandwidth. In this section, we carry out a formal analysis of the framework by showing the time and message complexity of the various algorithms.

3.4.1 Join time complexity

Given a time t, where d_t is the depth of the tree and N_{d_t} is the number of nodes at that depth, the time complexity is linear, of order $O(d_t N_{d_t})$.

Proof Assume that procedures *WhoIsHead* and *BeHead* take, respectively, t_1 and t_2 time units to complete. Then, the algorithm completes execution after $(i * t_1 + 1 * t_2)$ steps for each node, where i is the node's depth. Taking into account all the nodes, the total execution time is $\sum_{i=0}^{d_t} N_i(i * t_1 + 1 * t_2)$.

However, due to the simplicity of procedures *WhoIsHead* and *BeHead*, we consider that they take unit time to complete, i.e., $t_1 = 1, t_2 = 1$. So, the time complexity is: $O(\sum_{i=0}^{d_t} N_i(i+1)) = O(d_t N_{d_t} + d_t) = O(d_t N_{d_t})$.

3.4.2 Join message complexity

Similarly, given a time t, where d_t is the depth of the tree and N_{d_t} is the number of nodes at that depth, the message complexity is linear, of order $O(d_t N_{d_t})$.

Proof This proof is similar to the previous one. Considering a request and a response as two different steps, the algorithm requires 2(i+1) messages to complete the process at each node, where i is the node's depth. Following the logic of the previous proof, we obtain $O(2 * d_t N_{d_t})$, equivalent to $O(d_t N_{d_t})$.

3.4.3 Discovery time complexity

We determine the complexity of the two types of discovery processes involved in our framework. Note that the discovery algorithms are in fact search algorithms. As such, we do not provide any mathematical proof as the complexities of search algorithms are known and straightforward.

Node discovery complexity. This is of order $O(log(N))$, where N is the total number of nodes and gives the time taken by the algorithm to find any node in the network.

Proof For simplicity, assume that our cluster tree is a B-tree (sites have the same number of nodes). Let T be the number of site nodes. By analogy to a binary search, the complexity is of order $O(log_T(N))$.

But because our tree is not balanced, we identify the lower and upper bound of the complexity. Let t_{min} be the number of nodes in the smallest site and t_{max} be the number of nodes in the largest site. The complexity is bound between $O(log_{t_{max}}(N))$ and $O(log_{t_{min}}(N))$. This makes the node discovery complexity (average) logarithmic.

Resource discovery complexity. The complexity is of order $O(N)$ where N is the total number of nodes. Indeed, in a worst case search, the resource information of all nodes is parsed.

However, this time complexity can be reduced by classifying ("sorting") resources at each site. This causes the local (within a site) resource search to be logarithmic ($O(log(T))$), and the global resource discovery complexity to be of order $O(Slog(T))$, where S is the number of sites and T the number of site nodes. Since $S = N/T$, resource discovery takes less than linear time ($O(Slog(T)) < O(N)$).

3.5 Possible Optimizations

As previously explained, our P2P framework provides basic clustering and scheduling algorithms. However, this was conceived with runtime efficiency in mind and is open to optimization. For example, based on the *centrality degree*, the network tree can be structured symmetrically (i.e., sites are created logically to have the same number of nodes) for a fair distribution of knowledge. In this section, we show how the framework can be optimized by presenting a few ideas to improve the algorithms. Remember that each node in the framework has a table that contains its personal information (clustering variables) and the resource information of nodes it manages (including itself).

3.5.1 Optimizing root

The root position is critical to the resilience of the P2P cluster, as it is the only entry point to the cluster. Therefore, this position can be the cause of bottleneck situations in a very large network. Additionally, a failure of the root node splits the cluster until another node claims its position. In this case, new requests could theoretically timeout. To avoid these undesirable situations, a load balancing algorithm is needed. Since a DNS name is used to identify the root node, it follows logically that DNS optimizations can be leveraged.

1. The cluster tree can have multiple super (root) nodes ordered by priority. The priority is established with Rule 1 in Section 3.2.4. The oldest node has the highest priority and acts as the root node. The other nodes are backups and will take the root position (if the acting root node fails) according to their priority values.
   This solution can be applied using DNS service (SRV). DNS SRV is an extensive specification mainly used for mapping hosts to particular services. But, specifically, it enables

DNS entries to have priorities. So, multiple SRV records can be configured for capable root nodes in a network. Here is an example:

```
#service.name  TTL    class  SRV  priority  weight  port  target
root.ucloud.   86400  IN     SRV  0         10      80    192.168.2
root.ucloud.   86400  IN     SRV  1         10      80    10.10.1.9
root.ucloud.   86400  IN     SRV  2         10      80    169.254.5.15
```

2. Independent sites can be created. This occurs when the site head does not report to a superior node, thereby acting as the root node for that particular site. The site heads may only communicate for administrative reasons; this will form a federated cluster. This solution is particularly relevant for large sites. One approach to implement it requires the usage of local DNS resolvers. Another approach is to leverage the *weight* property in a DNS SRV record, which was introduced for load balancing.

3.5.2 Optimizing site head

Failures of site heads (creating holes in the cluster tree) affect resource discovery in the network. The presence of "holes" degrades the resilience of the cluster. When a site head fails, resources of nodes under its supervision cannot be discovered. This is an undesirable situation that persists till the broken links are detected. Though this is normally only for a short time, it could cost a critical query a response. This time is determined by the *Ping* procedure (see Section 3.5.3 below). To cover the holes, backup site heads can be used. In this case, a twin copy of the acting site head table is kept on capable nodes (i.e., potential site heads). The latter are the backup nodes to which site nodes duplicate their resources information. However, this redundancy method does increase the message complexity of the clustering maintenance.

A better method is to use a helper (partial) backup node for the site head. The helper node is not a replacement for its site head and thus there is no need to duplicate resource information. The helper node only keeps addresses of existing site nodes. If a failure occurs, it provides temporary query handling in place of the site head until the site nodes re-cluster. Note that this solution increases K-consistency, which means that the clustering algorithm will take longer to stabilize.

3.5.3 Optimizing *Ping*

As mentioned earlier, *Ping* is the watchdog timer for edges (node links) in the cluster tree. It is armed with "keep alive" messages sent periodically to verify and maintain links between nodes. The used period (*keepalive* time) is constant. This is judged to be inefficient because the churn rate of nodes should affect the *keepalive* time. The *Ping* algorithm can learn from previously sent messages and modify its period accordingly. By analogy with the probability of a node failure decreasing with time (postulated by Graffi et al. [43]), the probability of a link failure also decreases with time. Therefore, we suggest a learning function that increases the period as the link persists in time. Equation (3.1) is a simple example of a learning function while Algorithm 6 gives the corresponding *Ping* algorithm. This is inspired from the TCP connection retry algorithm where the retry time is a polynomial function [29]. But in our implementation, for a slower increase rate, we use a logarithmic function: the Fibonacci function.

$$F(i) = \begin{cases} (i-1) * F(i-1) & \text{if } i > 1 \\ Period & \text{if } i = 1. \text{ Period is a constant.} \end{cases} \tag{3.1}$$

3.5.4 Optimizing discovery constraints

For developing a convenient resource discovery algorithm for our framework, we introduced two constraints: *uptime* and *timestamp*. These constraints can be optimized as described below.

Optimizing *uptime* constraint

Site heads are chosen based on the criterion (see "Uptime constraint" in Section 3.3.2) that they do not fail as often as ordinary nodes. So, in a semi dynamic network, root and site head optimizations may not be needed. Consequently, the associated overhead is non-existent. However, to increase the P2P platform resilience, necessary for long term resource reservation, one can use an approach inspired from Refs. [43] or [57]. In short, the idea is to anticipate a node failure and to predefine a backup node accordingly. This redundancy improves resource discovery and reservation. In fact, a proof is given in [43] showing that a reservation can be

Algorithm 6: Ping.

Let *period* be the keep alive time and *ping* the keep alive message.
Let $t = t_0 = Time(now)$.
PROCEDURE *Send*, *Receive*;
$i = 0$;
while *1* **do**
  if $(t - t_0) > period$ **then**
    Send(ping, GetNode(-1)); //Recall that $GetNode(-1)$ returns the site head address.
    if *Receive(GetNode(-1))* $==$ *TRUE* **then**
      period = F(i++);
      $t_0 = t = Time(now)$;
      continue;
    end
    else
      break;
    end
  end
  else
    $t = Time(now)$;
  end
end

fully guaranteed under churn with very low traffic overhead. The cumulative probability of the nodes involved in the reservation is 1.

Equation (3.2) is the probabilistic equation used. The probability $P_{fail}(p, t_R)$ that a peer p will go offline in the next t_R minutes is calculated as the difference of the offline probabilities at $t_{on}(p)$ and $t_{on}(p)+t_R$ in relation to the probability of having survived until $t_{on}(p)$. $F(t, k, \lambda)$ is the cumulative distribution function for the Weibull distribution with time variable t, shape parameter k, and scale parameter λ.

$$P_{fail}(p, t_R) = \frac{F(t_{on}(p) + t_R, k, \lambda) - F(t_{on}(p), k, \lambda)}{1 - F(t_{on}(p), k, \lambda)} \tag{3.2}$$

With this solution, the resource discovery process returns multiple resource providers. Then, the probability function in Equation (3.2) is used to select the most suitable provider.

Optimizing *timestamp* constraint

The *timestamp* constraint can be adjusted to be a function of the traffic latency in the operating network. That is, the *timestamp* of information should be increased appropriately when node interconnection is slow. This is to avoid valid information expiring too quickly. Indeed, a short expiry period requires unnecessary additional information updates in an already slow network.

3.6 Summary

In this chapter, we presented a new approach to the adhoc clustering problem. We used the concept of CDS to structure an adhoc network, making it dependable and reliable. The proposed algorithm (H-DCA) combines both weight (*uptime*, *centralitydegree*) and location criteria with ease of implementation. Note that, if necessary, additional criteria can easily be incorporated into the algorithm. For example, *load level* and *centrality degree* of nodes can be used to determine site heads. Nevertheless, the H-DCA succeeds in cluster formation, and guarantees the possibility of inter and intra cluster communications. We prove the node discovery time complexity to be logarithmic, thereby matching the efficiency of popular algorithms such as Chord [45] and IPOP [28]. K-consistency in the H-DCA cluster is always 1, ensuring the network is fully connected. The site heads (members of the dominating set) work together to ensure the resilience of the whole network. Knowledge is therefore distributed.
Though the algorithm works for an adhoc network of any degree of mobility, it is practically more efficient for semi adhoc networks (lower degrees of mobility). Once the tree is formed through clustering, using it is straightforward. The clustering technique employed is suitable for capacity-based queries, which were introduced during the presentation of the scheduling algorithms.
In addition, various complexity analyses were performed to formally describe our framework. With this framework as the basic tool for our cloud system, we can proceed with the provisioning of computing services.

Chapter 4

Computing Service Composition

A computing service is an on-demand computing platform suitably set up to satisfy user needs and to exhibit ease of use. So, computing service composition is the process by which computing resources are made available in a usable form. Traditionally, computing services are built on top of legacy operating systems (OSs), which in turn interact with available computing resources (hardware devices). These types of services are termed desktop services and are offered through desktop applications. But to adhere to the scope of our research, we are interested in how to build services on generic computing platforms while maintaining resource isolation, guaranteeing ubiquitous service access, and ensuring service usability and scalability. These types of services are termed cloud services. Multiple users can have access to a cloud service or conversely, a single user can have access to multiple cloud services. Most legacy OSs do not natively support the creation and execution of such services. To build this support, key techniques for the composition of cloud services are leveraged, of which virtualization techniques are the main ones. With the application of these techniques, we introduce a computing power service (CPS). The CPS, the cloud service being offered, is a ready to use service that responds to a user's computing power needs. The term computing power is used to stress the provision of computing resources such as processing power (e.g., CPU) and processing memory (e.g., RAM). CPSs are set up in the form of virtual machines (VMs). To help understand the architecture of such services, this chapter starts by exploring the tools involved in their composition.

4.1 Enabling Technologies

Depending on the granularity of the computing service to be offered, two virtualization concepts can be used: hardware or software infrastructure virtualization. Those concepts refer to the creation of a virtualization platform at the hardware or software level, respectively. The following sections give more details.

4.1.1 Complete virtualization

As shown in Fig. 4.1, complete virtualization is performed almost at the physical level. This includes full and para virtualization techniques introduced in Section 2.3.3. Computing devices are abstracted by isolating shared physical devices and making them available for an OS. This OS (guest OS) is run and contained within the native OS (host OS) of a node. In this case, the computing service is simply the provision of a complete virtual computer on top of which the user can boot a fresh or self installed OS. This is perceived as a *hardware IaaS*. The service offers a generic hardware platform open to a wide variety of user computing experiences (through a variety of guest OSs). We see complete virtualization as a trivial solution for building a CPS. Shared resources are made available directly to the user without further customization. In other words, any software related operation is left to the discretion of the user.

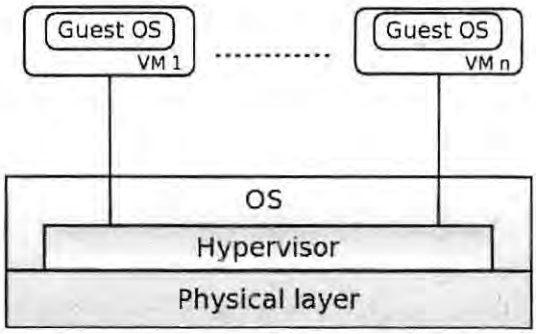


Figure 4.1: Complete virtualization architecture.

- Advantages: This setup can run virtually any operating system.

- Disadvantages: A VM within this category has a longer boot time and is more memory expensive. Additionally, the transfer of guest OS images during VM creation contributes to the service initiation latency.

4.1.2 Selective virtualization: containers

Selective virtualization is done at the software level. Resources are isolated for particular software applications, and computing devices are still virtualized. Selective virtualization, however, differs from complete virtualization in that the virtualized infrastructure is specific to the application(s). Particularly, the kernel OS available on the worker node can be used to run applications for an external user. An isolated environment is created within the kernel to host these applications. With selective virtualization, a set of applications are instantiated and/or automated, depending on a user request. This is useful for rapid instantiation of a service and to save disk space and memory. In this case, the provided service is perceived as a *software IaaS*. The service provides an application-specific and ready execution environment. This operation is sometimes referred to as *sandboxing*.

Sandbox This is a container which is appropriately customized and isolated for security and safety reasons. A container is a private environment in which a software application is executed. Container environments have separate namespaces, set up conveniently during the application's lifetime. An application is given its own IP address, and its own file system, and shares the actual hardware resources with other applications.
For the sake of generality, a container with running applications is also referred to as a virtual machine.

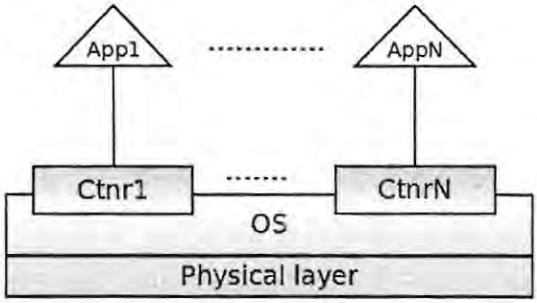


Figure 4.2: Selective virtualization architecture.

- Advantages: It offers a lightweight VM workload (potential lightweight VM creation and execution).
- Disadvantages: It requires a compatible runtime environment for any prospective VM.

Table 4.1 summarises the different features of complete and selective virtualization systems. Considering that the choice of either virtualization type depends on user preference, this chapter discusses both of these.

Features	Complete	Selective
Multiple Kernels	✓	✗
Administrative power	✓	✓
Checkpoint & Resume	✓	✓
Live migration	✓	✓
Live system update	✗	✓

Table 4.1: Feature comparison of complete and selective virtualization systems [62].

4.1.3 Storage virtualization

Although not discussed previously, *storage* is a required computing resource that participates actively in the CPS. A storage unit is needed to accompany the computing service. The definition of this unit is not explicit because it depends on the nature of the service. A storage unit can be created either temporarily to ensure the successful instantiation of a VM, or as a file storage for the user, or both. Either way, a cloud-ready disk storage is created that is non-pervasive. We define two types of storage.

Stand-alone storage An empty image disk is created to handle a user's storage needs. Depending on these needs, the disk is set up appropriately. For example, a user may need the disk to contain a bootable OS.

Shared storage This is file storage that can be shared between a VM and another requestor. When needed (e.g., for faster transfer), the file storage is compressed into a disk image.

Disk imaging suggests the creation of a virtual file system (VFS). To perform this, we used the *libguestfs* library in [10]. Though other methods such as *qemu-img* [26]) exist, they do not offer a well defined API. Libguestfs is a mature project and is specifically designed for virtualization solutions. Our cloud service also benefits from the performance improvements in this library.

4.2 CPS Architecture

At this point, it is clear that the basic tool for creating a CPS is virtualization. Evidently, the first step in guaranteeing the hosting of any CPS is to perform resource isolation and allocation on worker nodes. Virtualization appears to be the native solution for this. It provides us with a virtual desktop solution represented by a VM. But, this solution is only meaningful to a user who has physical access to the "desktop". So, the virtual desktop solution model needs to be transformed into a cloud-specific solution. There is still a long way between a VM and a ready-to-use cloud service; especially when attributes of a P2P environment have to be considered. With respect to this, we define a service as a computational response to a task description based on a running VM.
From the previous chapter (Chapter 3), we understand the role of peer nodes to be resource providers. These nodes may have different potential and will have different load cycles, thereby continuously providing a variety of resources. A service provision methodology should cater for this. The service itself needs to be a mobile agent and should be designed to exhibit this property. Each prospective service provider implements a cloud containing a stack of modules, thereby elevating a node from resource provider status to that of service provider. Fig. 4.3 depicts the structure of a CPS as defined in the cloud.

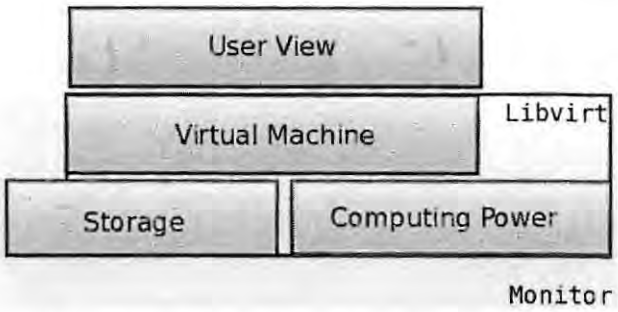


Figure 4.3: Service provision components.

The layers depicted in Fig. 4.3 represent the components needed on a node to establish a service provisioning back-end. The descriptions of these layers are given below.

4.2.1 User view

The user view layer constructs the portal (front-end) of a service. Naturally, an interface is needed for a remote user to access its service. The nature of this interface is inferred from the nature of the service itself. Since the CPS provides computing resources in the form of VMs, a desktop abstraction is established at the user view layer. To guarantee a familiar user interaction, a remote desktop technology is employed. Instead of creating a new user interface for VMs, we use currently available and well designed GUIs, thereby providing a perfect desktop integration. So, the user view layer implements a remote desktop protocol. Each VM is attached to a VNC (Virtual Network Computing) server, exporting its desktop. Therefore, users can remotely control or monitor their jobs, running through VM(s) as if they were running locally.

Though we chose for user comfort a remote desktop interface, a text based (serial) interface is also possible. The latter is obviously less bandwidth expensive but may only suit technical users.

4.2.2 Virtual machine and computing power layers

These layers perform resource isolation and allocation. Necessary computing resources are dedicated to a particular VM. Using the task attributes obtained from the user, a VM description containing details of its resources is created. With this description, operations in a VM lifecycle, as shown in Fig. 4.4, are pipelined. Libvirt is the API used to implement these operations. Details of this implementation are provided later in Section 4.3.

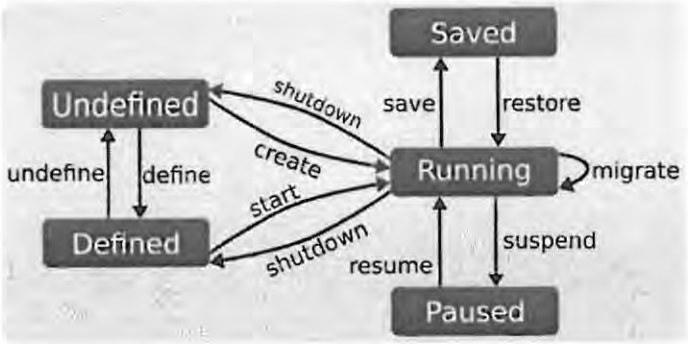


Figure 4.4: VM lifecycle.

Through virtualization, a service is potentially mobile. It can be saved, migrated and restored on different hardware. The only attribute that may change during a mobile operation is the service portal. This is taken care of in the user view layer, which acts as a proxy during such an operation. That is, the user view only switches to a different node when the service is fully migrated and is usable. As such, mobility of the service is transparent to the user.

4.2.3 Storage layer

As introduced in Section 4.1.3, a temporary storage is created for convenience. This will usually be an empty disk. But it can be *bootstrapped* (i.e., made bootable) depending on the user's request. This layer makes use of the techniques developed in Section 4.1.3. Once a VM is terminated, its disk image is discarded; the user has of course, the choice to retrieve a (compressed) copy thereof.

4.2.4 Monitor layer (Watchdog)

Because nodes can join and leave the network freely, availability of a started service is the system's responsibility. The role of this layer is to oversee service execution and trigger events accordingly. It basically makes sure that the different components involved start, run, and terminate successfully. In case of complete failure, which means failure of the monitor itself, the user should resubmit the task.

Caching protocols are also implemented at this level. The *check-pointing or save* virtualization facility available through the Libvirt API is leveraged to make snapshots of a running VM. With this snapshot and service attributes such as *username*, a service is cached and can be restored later on a different node if necessary.

4.3 CPS Implementation

Libvirt is the core API for this implementation. The API uses a generic word for the term virtual machine, namely, domain, in order to embrace other types of virtualization such as containers. But for consistency, we continue to use the term virtual machine. It should be noted that the Libvirt API is quite extensive. Even though the CPS implementation as presented in this section only partially uses the API facilities, it does not infer a limitation to the variety of CPSs that can be built. That is, the implementation can be further developed to the full extent of the library. For instance, Libvirt can be used to interact with public clouds such as the EC2.
So for the sake of simplicity, we present a partial implementation in this section.

4.3.1 CPS definition

Naturally, the CPS attributes are defined on receipt of a user request. The user is expected to provide values for the following attributes in the request.

- Computing attributes: number of processing cores, processing memory (RAM) and disk space.
- User credentials: user name and password.

These attributes identify users and reflect their computing needs. But, they only partially define a CPS. To complete the CPS description, state and portal attributes are added. The state attribute shows the position of a service in its lifecycle. Possible states are: instantiated, running, paused or terminated. Portal attributes contain the necessary values for a user to access the service through a remote connection. So, portal attributes are: IP address and port

number. With these complementary attributes, a CPS is fully defined and can be instantiated. In addition, a corresponding unique ID is created during the CPS definition.
In brief, a CPS is uniquely characterised by the tuple (u, p, c, r, d) where u is the user name, p is the password, c is the number of cores, r is the memory in MB, and d is the disk space in MB.

The Libvirt API requires the description of a VM to be done using the Extensible Markup Language (XML). So, to conform with the API, the sub-tuple (c,r,d) which represents VM related attributes, is defined using the language. Fig. 4.5 illustrates the template for a virtual machine XML description.

```
<domain type='qemu' id='2'>
 <name>VMid</name>
 <memory>r</memory>
 <vcpu>c</vcpu>
 <os> ..... </os>
 <devices>
  <emulator>.....</emulator>
  <disk> ..... </disk>
   <filesystem> ..... </filesystem>
  <video>
    <model type='cirrus' vram='9216' heads='1'/>
  </video>
 </devices>
</domain>
```

Figure 4.5: XML template.

The empty tags in the XML represent entries specific to the virtualization driver used. In fact, the XML description will vary slightly for each driver (e.g., added tags). A virtualization driver is the software required on a peer node for it to exhibit a particular virtualization capability. Examples of such drivers are the XEN, KVM, and OpenNebula drivers which are all available in a Linux environment. In this implementation, two virtualization drivers are used: the QEMU and LXC drivers.

4.3.2 Libvirt: QEMU-KVM

Recall from Section 2.3.3 that QEMU is a full hardware emulator. It allows the implementation of full virtualization-based services as explained in Section 4.1.1. Such a service defines a hardware infrastructure according to a user request. QEMU is an appropriate tool for this type of service. Capabilities of QEMU such as portability are leveraged through its driver (see [26] for QEMU tools and optimizations). For example, to emulate a x86 virtual machine, the driver requires the XML tags given in Fig. 4.6 to be included in a VM description. In fact, Fig. 4.6 completes the XML template given in Fig. 4.5.

```
<os>
  <type arch='x86_64' machine='pc-0.12'>hvm</type>
  <boot dev='cdrom'/>
</os>
<devices>
  <emulator>/usr/bin/qemu-system-x86_64</emulator>
  <disk type='file' device='cdrom'>
    <driver name='qemu'/>
    <source file='/home/ubuntu-8.04-desktop-i386.iso'/>
    <target dev='hdc' bus='ide'/>
    <readonly/>
  </disk>
  <graphics type='vnc' listen='127.0.0.1' port='6000' passwd='p'/>
</devices>
```

Figure 4.6: Partial description of a x86_64 virtual machine in Libvirt.

4.3.3 Libvirt: LXC

LXC stands for Linux container and enables virtualization at the kernel level. LXC is mainly used to run isolated daemon processes in the Linux kernel, but can also be used for interactive processes. The description for a LXC VM is done appropriately for such processes. The LXC driver allows us to implement services that require selective virtualization. Recall that these services are called *software IaaS*. As explained in Section 4.1.2, a VM on selective virtualization will run a container and not an entire OS. In addition to the VM description, a file (script) describing the start-up of the container is required. This script sets up the appropriate namespaces needed for applications to run inside the container. For interactive processes, the script will also define a VNC server (the service portal). Indeed, the latter

is defined separately in the script because LXC does not provide native support for such processes. A sample bash script is shown in Fig. 4.7.

```
#!/bin/sh\n
touch /etc/fstab
mount -t proc /proc /proc
mount -t sysfs /sys /sys
mount -b /mnt/userfilesystem/ /
export PATH=$PATH:/usr:/usr/local:/usr/bin:/usr/sbin DISPLAY=:0
/usr/bin/Xvnc :0
/usr/bin/gedit  #launch a user application
```

Figure 4.7: Example of LXC bash script.

The container filesystem can be constructed from either the host filesystem or the user filesystem (uploaded files), or a combination of both.

A *software interposer* is used to complement a container. It is created for each service instance. An interposer is the implementation of the sandbox features presented in Section 4.1.2. For instance, to get files dynamically from the user filesytem, we write an interposer for the *open* system call of the Linux kernel. This interposer is a shared library, which is linked with the container applications during runtime. This is possible because the Linux kernel uses dynamically linked libraries for system calls.

4.3.4 Implementation diagram

Each CPS carries at its core a VM. So, most CPS operations are extensions of VM operations. For now, we describe only the core implementation of a CPS. A complementary description is provided in the next chapter.
The diagram in Fig. 4.8 shows the UML representation used for the CPS implementation. The two key components in this diagram are presented.

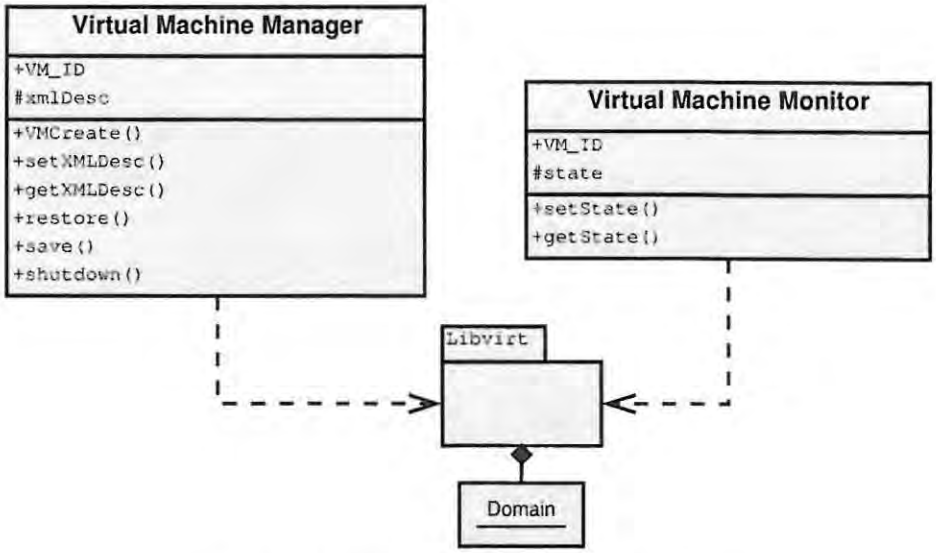


Figure 4.8: Partial CPS implementation.

Virtual Machine Manager. The VM manager is responsible for performing VM related actions (e.g., start-up, shutdown). For instance, it instantiates a VM (VM creation and start-up) using the VM attributes described in Section 4.3.1. All the operations involved in a VM lifecycle (see Fig. 4.4) are defined in the VM manager. The latter is implemented as a C++ module, based on the Libvirt API. This is how the Libvirt methods are leveraged. Note that the Libvirt API is natively written in C. The UML diagram in Fig. 4.8 shows the VM manager module with prototypes of its main methods.

Virtual Machine Monitor. The VM monitor is implemented as a method that is threaded for each VM instance. The role of this thread is to generate or handle VM events. The state attribute described in Section 4.3.1 is involved at this point. Evidently, an event occurs each time the VM changes state. These changes are easily observed through Libvirt. Corresponding events (e.g., VM shutdown) can therefore, be generated.

The VM monitor does not have the capability to handle every type of event by itself; it may just engage a distinct operation. For example, it can initiate a VM migration as a result of certain events (e.g., "host shutdown"). The migration is then handled by a separate operation because a VM migration infers a service migration.

4.4 Summary

Within this chapter, we discussed how to build computing services in a computing node. Particularly, virtualization was used to facilitate resource abstraction, isolation and provision. With this, two types of services were described: selective and complete virtualization services. The core utility provided through these services is the physical infrastructure; hence the term computing power service (CPS). Software related decisions are left to the discretion of the user.

Note that, the description of a CPS presented in this chapter does not define it as a full cloud service. So far, the CPS is a stand alone and isolated service, but not a mobile agent. It will be fully understood as a cloud service in the next chapter.

Chapter 5

μCloud Architecture and Implementation

Cloud services can be automated in a hostile environment; i.e., cloud solutions can be transparently integrated into a running computing network. For instance, a cloud model can be applied to P2P networks to provide on-the-fly cloud solutions. In response to this statement, we propose an on-demand service provision system. However, we are aware that there are issues with hosting a service on a peer node (the worker) and issues with making it available to another peer node (the client). Specific tools need to be implemented to transform a P2P computing service into a cloud service. The solutions proposed in previous chapters are combined to form a P2P cloud platform named μCloud. It establishes a lightweight and autonomic computing sharing environment. This chapter details the processes used in the μCloud architecture and implementation.

5.1 Towards a RESTful Cloud Service

A formal description or standard is needed to guide the design of μCloud. Several standards are available for cloud solutions. The nature of intended services in the cloud defines the service delivery model to be employed, which in turn defines the architecture of the cloud itself. To develop a cloud and cloud services, there is a new idea inspired from SOA. It is

named SOI (service oriented infrastructure). SOA is related to the application architecture, whereas SOI is related to the infrastructure architecture. So, it is sensible to understand SOI properties, specially because a cloud is merely an IT infrastructure for delivering services.

5.1.1 Service oriented infrastructure

The Open Group responsible for the SOA standard provides the definition: "SOI is an architecture for describing IT infrastructure in terms of services". This encompasses all aspects of the service lifecycle, including service construction, deployment, and operation. These aspects define a delivery infrastructure. Naturally, an infrastructure management framework is needed to plan, build, and run the IT resources and services in accordance with the business logic and service level requirements. An SOI suggests a service centric design for this framework. It requires key service related components at each level of the architecture (see Fig. 5.1).

- Service provisioning engine: implements the delivery model.
- Service monitoring engine: monitors services throughout their lifecycle.
- Service management engine: performs actions on services once they are activated (e.g., re-allocation).

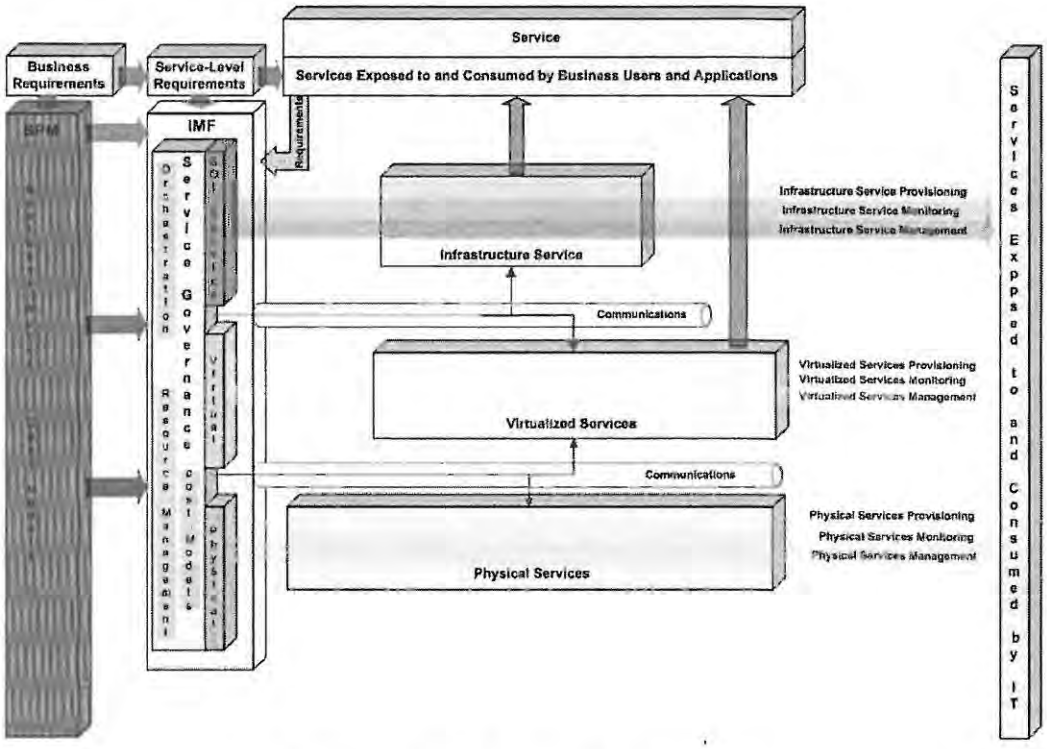


Figure 5.1: SOI reference model [13].

The model in Fig. 5.1 depicts the building blocks for the SOI. To implement an infrastructure, these conceptual blocks must be instantiated for each type of service. Infrastructure services typically use and/or provide a virtualized pool of shared resources (servers, network, storage, infrastructure software) that are deployed and managed in a highly automated way. This is true for the CPS (presented in the previous chapter) and is, in fact, the service provided by μCloud. Therefore, the latter must follow an SOI reference architecture.

Note that the SOI standard is still being drafted by the Open Group.

5.1.2 RESTful service

This section describes a cloud service delivery model, REST, chosen due to its relevancy to SOI. REST is an architecture style for designing web applications. It describes a client-server architecture with clear separation of concerns so as to allow components to evolve

independently. As presented above, this is important for an SOI since it consists of distinct building blocks. REST uses the common term "server" to refer to such an infrastructure. Client-server communications are required to be stateless. For instance, each request must contain all of the information necessary to process it, and cannot depend on any stored context on the server. So, a service state is kept on the client side.

The μCloud services are RESTful with respect to their management. For example, a CPS construction, deployment, and execution are performed RESTfully. But, CPS applications (e.g., a running VM) will need a user interaction oriented architecture (e.g., a Remote Frame Buffer protocol). Such interaction cannot be RESTful. Though it was originally created for distributed hypermedia systems, REST is judged convenient for μCloud due to its P2P property. A stateless service provision model with well defined constraints (REST) is required for a scalable P2P cloud platform. Actually, with more cloud implementations being developed by vendors and open source communities, RESTfulness is usually mentioned as an important property [46].

5.2 μCloud Overview

μCloud is a cloud service infrastructure based on P2P. A peer node can either be a client (requester) or a worker node (provider). The set of worker nodes forms the computing unit. A client benefits from the computational capabilities of a worker node through on-demand service composition, provided that the node can host related VM(s). This primarily depends on the node's self-specified restrictions on its resources (e.g., amount of shareable memory), established beforehand (i.e., prior to joining the system). Moreover, each node is only liable for its own execution and may not be aware of the existence of others, unless absolutely necessary (see Section 3.2.5). The desired outcome is to have each node operate independently, automating its service provision to satisfy both host and guest(s) computations. No latency should appear on the host, as it accepts a guest's job only if it is capable of completing it.

From the description above, we have identified two distinct architectural views in μCloud: the client experience and the computing unit. Taking these views into account, a simple diagram depicting the μCloud system is presented in Fig. 5.2. In fact, the μCloud system consists of three types of participants, namely, service requesters (clients), service providers

(computing nodes), and service brokers (coordinators). One of the coordinators (represented by the cloud in Fig. 5.2) is chosen to orchestrate all initial interactions between clients and the computing unit.

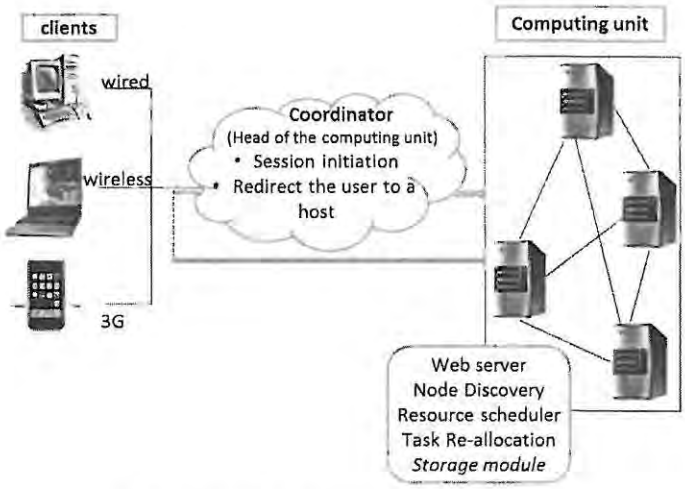


Figure 5.2: Overview of μCloud.

By aggregating the solutions in Chapter 3 and Chapter 4, we found that the following components need to be implemented by each computing node in μCloud: Web server, Task allocation, Task re-allocation, Information policy. From this, we draw a simple diagram in Fig. 5.3, showing how the main components of μCloud are connected. Details of these components are depicted in subsequent diagrams.

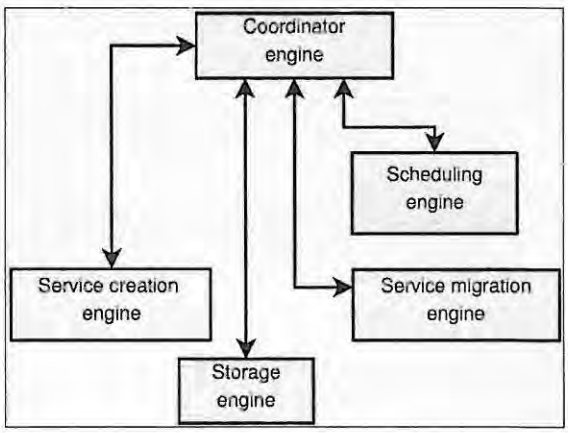


Figure 5.3: μCloud main components.

To further our conceptual understanding of μCloud, we provide brief descriptions of the two architectural views just introduced.

5.2.1 Client experience: request processing

The issues from a client point of view are usability, desirability, and productivity. The client experience in a system is particularly characterised by the way its requests are processed. So, from a client point of view, μCloud implements the following design decision. All user interaction is done via a web browser, for compatibility and portability reasons. User requests are captured through this interface and processed accordingly (see Fig. 5.4). For instance, a submitted user task description (represented by the CPS tuple) is transformed into a VM description, a VM is then scheduled to start on a node (thereby executing the task), and access (control) of this VM is given back to the user. These steps are carried out as part of the cloud service and were introduced in the previous chapter.

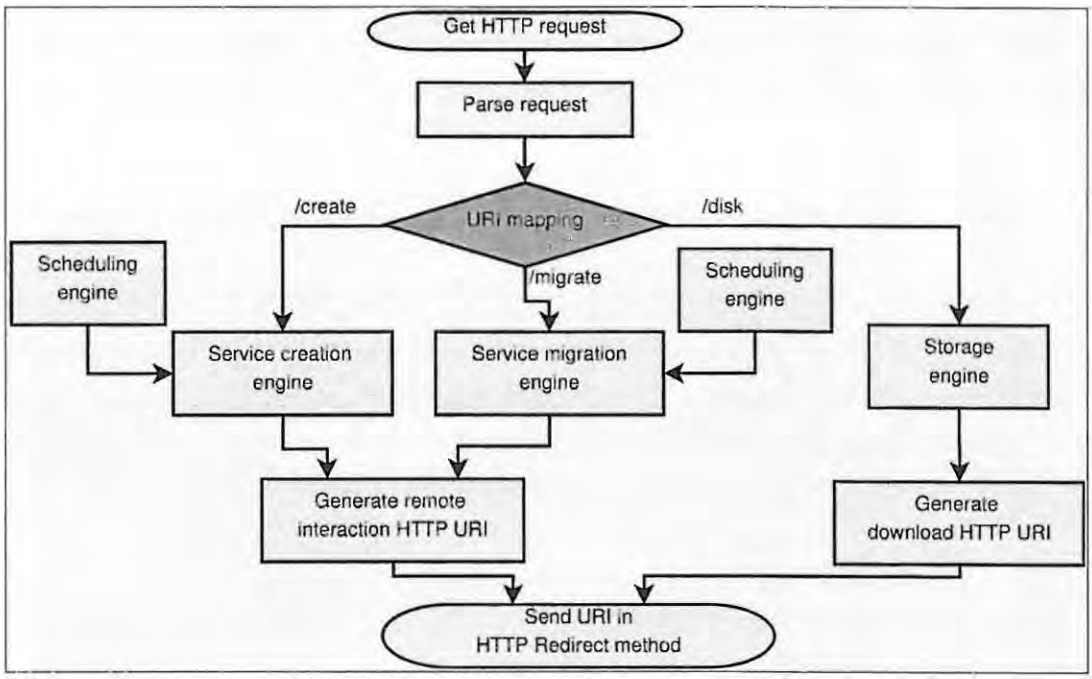


Figure 5.4: Request workflow in the Coordinator engine.

5.2.2 Computing unit: response handling

The computing unit is a collection of computing nodes (i.e., peer nodes sharing computing resources). Each acts as an independent cloud service provider. We also stipulate that every computing node initially has the same potential. That is, there is no weighted priority system (pre-established) other than the one used by μCloud. Note that the shared computing resources are defined by the administrator of the computing node.

From the computing unit view, the nodes are responsible for handling responses. A node responds to a client request by creating and hosting a CPS accordingly. We have previously identified the issues with setting up a host-ready environment for a CPS. P2P networking solutions presented in Chapter 3 are used to solve these issues.

Basically, response handling is done in three main phases: find a node to host the CPS, allocate the CPS job to the node to start its execution, and monitor this node for failure handling. These phases are achieved, respectively, by using the following methods:

- On-the-fly node discovery: We recognise that a subscription-based method, using DHTs (Distributed Hash Tables), as described in Section 2.1.4, has shown great success. However, the CPS needs a P2P infrastructure without a DHT since its overhead cannot be tolerated. Nevertheless, a fully distributed node discovery model is still needed. So, we use a reactive protocol that locates suitable nodes when there is a pending task.

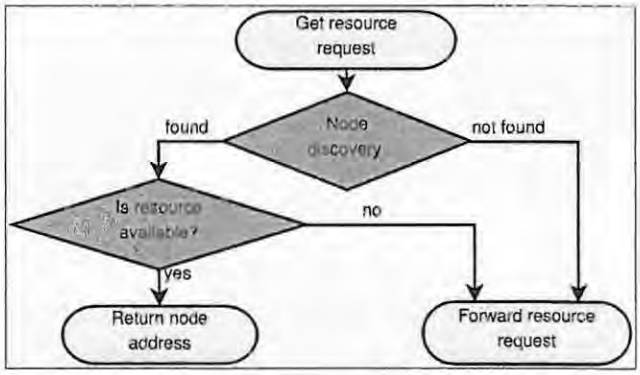


Figure 5.5: Scheduler engine.

A solution was proposed in Section 2.1.4; a resource discovery methodology was conceived to find capable nodes. The methodology is basically a capacity-based search,

and it is involved in the node detection process as depicted in Fig. 5.5.

- Service creation: When a node is prepared to host a service, the process for creating and starting that service is initiated. Fig. 5.6 is a simple diagram that depicts the μCloud *engine* responsible for handling service creation operations.

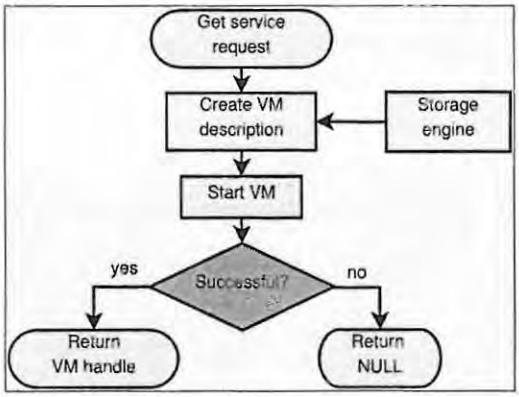


Figure 5.6: Service creation engine.

In addition, a storage media is required in the CPS. Fig. 5.7 is a diagram showing the operations in the μCloud storage engine.

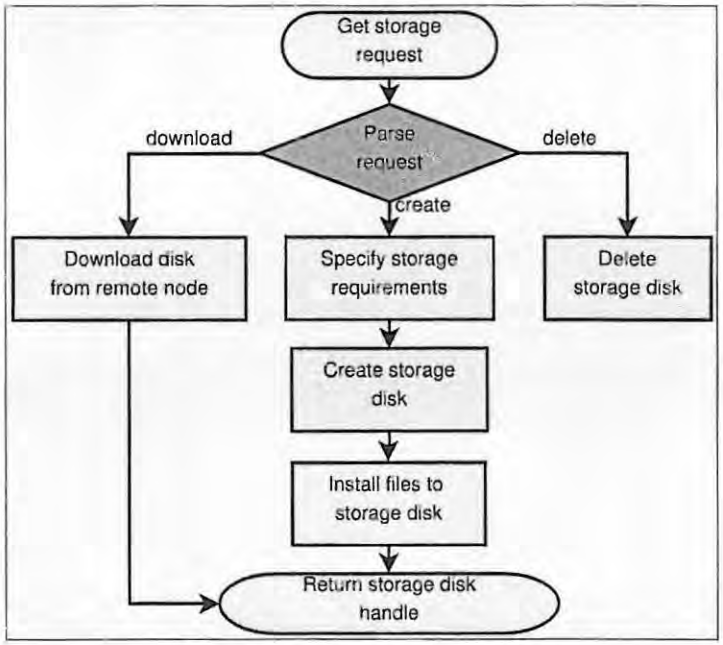


Figure 5.7: Storage engine.

- Failure handling: When a peer node has an overload or is not able to continue hosting running services (VMs), the extra load should be migrated to a different node; a load sharing mechanism is implemented to achieve this. In the case of complete failure, the task is resubmitted. Fig. 5.8 is a simple diagram representing the service migration process.

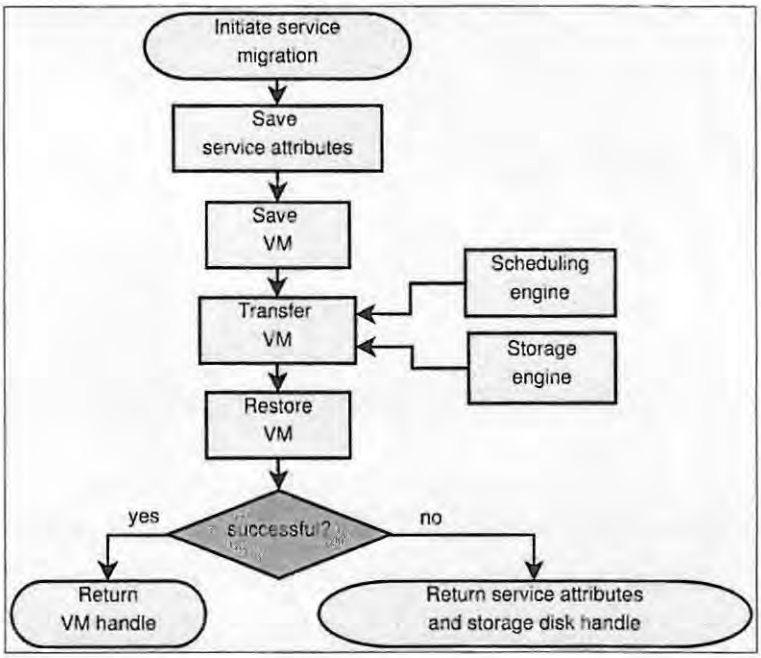


Figure 5.8: Service migration engine.

5.3 μCloud Coordinator

As suggested by the SOI, the μCloud architecture needs an infrastructure management framework (IMF). Considering the P2P characteristics of μCloud, it follows logically that its IMF must be distributed. Indeed, the computing unit can exhibit emergent behaviour allowing its nodes to be self-managed. That is, every service provider in μCloud has a corresponding service broker. In other words, each computing node runs its own management engine, called the *coordinator*. Note that coordinators have different roles at different times due to the hierarchical structure established by the H-DCA on the P2P system. For instance, the coordinator of the root node naturally represents the main coordinator of the cloud and is

the entry point for clients. Apart from this extra property, all coordinators are functionally the same. The role of a coordinator is to mediate the service agreement between the client and a computing node. In addition, coordinators of head nodes have the following functions: switcher and job tagging. This is imposed by their status as resource supervisors in the cluster tree.

Coordinator functions are divided into specific categories as shown in Fig. 5.10. These categories are hinted at by the SOI reference model (see Fig. 5.1).

Figure 5.10: Service lifecycle.

5.3.1 Service discovery and instantiation

The service provider discovery is mostly based on a host discovery algorithm (this is equivalent to the resource discovery algorithm). The latter suggests a host capable of handling a given request. During this phase, a node with the minimum computing power requirements is detected among the pool of connected nodes. This node will become a service provider by implementing an adequate service, with respect to the model explained in Section 5.1. This is the first phase in a service lifecyle.

In the service instantiation phase, the service provider first builds (i.e., implements, assembles, and/or orchestrates) the service and then prepares it for deployment and execution. The operations performed include preparation and configuration of virtual storage for the VMs that constitute the service as well as specification of dependencies among the different service components. The service provider may actually automate the VM initialisation to have it ready for use when it is started (e.g., LXC based CPS).

5.3.2 Service deployment

This includes two steps: scheduling and starting a service instance. A scheduler is invoked to allocate the computing resources required for the service to execute (i.e., start related VMs). During the deployment phase, the coordinator ensures that the service has started correctly and the user has access to the VM. In case this fails, the coordinator re-deploys the service on another node. The worker node is found this node using a *best-effort* scheduling algorithm, which merely looks for an available node satisfying the minimum computing power requirements in the shortest possible time. We consider this algorithm to be the same as the host discovery algorithm.

5.3.3 Service maintenance

This relates to the re-allocation of a service instance; i.e., service migration. A computing node needs to perform autonomous actions to, e.g., consolidate and redistribute service workloads, replicate data sets, etc. The service maintenance module implements the fault tolerance property in the cloud. When a fault occurs on a host, its tasks are re-allocated, based on the scheduling algorithm just introduced. A fault will either be a host decision to stop external jobs, or its limitation to pursue their execution. To detect faults, the module interacts with the monitor layer described in Section 4.2.4. Its core implementation uses virtualization capabilities such as suspend/resume and migrate available in Libvirt. Practically, the service migration process is equivalent to that of a new service creation. It is only the state of the service that differs; i.e., it is set to "paused" instead of "new".

5.4 μCloud Implementation

So far, we have covered the specifications and algorithms needed to build a P2P cloud platform. In this section, we present an implementation of μCloud. To reflect the flexibility and RESTfullness of the μCloud infrastructure, a modular implementation is used. The system modules are written in C++, because of its speed and the management and web modules are written in Ruby, because of its interpreted nature. The implementation is interoperable

and requires no systematic change on a peer's OS (host OS). Although Linux was used as the host OS, it can be ported to other OSs provided that they support the required libraries.

Fig. 5.11 shows a class diagram depicting the main components (see Fig. 5.2) required to implement μCloud on a computing node. These components are explained in subsequent sections.

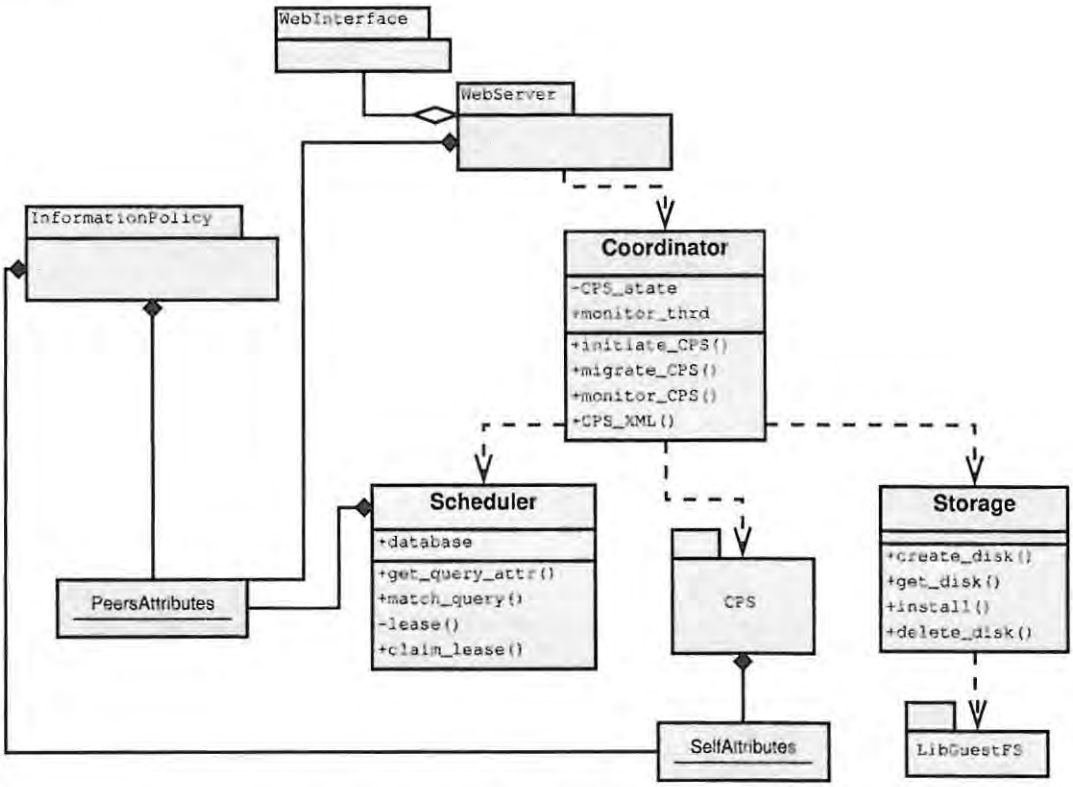


Figure 5.11: μCloud class diagram.

5.4.1 *SelfAttributes* object

This object stores up-to-date information about the node itself. This information is represented as attributes which completely describe an active μCloud node. As shown in Listing 5.1, a node has both clustering and computing resource attributes. The former attributes are initialized as specified by Algorithm 1 in our P2P framework. On the other hand, the attributes for shared resources are initialised with to respect to the administrator of the

computing node. All these attributes are stored in an IPC shared memory using the Unix System V API. In fact, the *SelfAttributes* object is shared between the *CPS* and the *InformationPolicy* modules. Obviously, the *CPS* module needs to modify the attributes of shared resources during service creation (lease of resources) or destruction (release of resources). The *InformationPolicy* notifies the node's supervisor of these modifications.

Listing 5.1: *SelfAttributes* object structure.

```
#define Mbytes *1024
#define maxi_time_out 130  //expiry timeout for attributes

struct resource_attributes
{
        unsigned int nb_of_cores;
        unsigned int nb_of_gpu_cores;
        unsigned int ext_ram;          //in Mbytes
        unsigned int gpu_ram;          //in Mbytes
        unsigned int disk_size;          //in Mbytes
        char availability[18];
};

struct node  //contains clustering properties
{
        char ID[16];    //IPaddr
        unsigned short node_depth;
        struct _weight
        {
                int degree;
                double uptime;
        } weight;
        char networkRange[36];
};
```

5.4.2 *PeersAttributes* object

μCloud uses a SQL database to represent its knowledge. That is, SQL tables are used to store and classify information about nodes. For a given node, the information comprises the complete attributes (as defined in Listing 5.1) of nodes under its supervision and the clustering attributes of its supervisor. The *InformationPolicy* module is responsible for updating this information. It modifies relevant database entries appropriately to reflect changes on given node's properties.

For simplicity and a lightweight design, SQLite was chosen to implement the knowledge representation. The *PeersAttributes* object is therefore a SQLite file with tables describing relevant peer nodes (head node, child nodes), including the node itself. Two tables named *computing_info* and *meta_info* are defined, respectively, to contain computing resource information and clustering information. Listing 5.2 defines the structure of these tables.

Listing 5.2: *PeersAttributes* SQL tables.

```
CREATE TABLE computing_info \
        (IPaddr varchar(11) PRIMARY KEY,\
         nb_of_cores int,\
         ext_ram int,\
         nb_of_gpu_cores int,\
         gpu_ram int,\
         disk_size int\
        );
CREATE TABLE meta_info \
        (IPaddr varchar(11) PRIMARY KEY,\
         node_depth int,\
         degree int,\
         uptime int DESC,\
         networkRange varchar(23),\
         timestamp int\
        );
```

5.4.3 *InformationPolicy* module

This is an important component of μCloud as it enables it to express its P2P functionality. The *InformationPolicy* module is used by a node to collaborate with its peers. In fact, it is an implementation of the P2P framework presented in Chapter 3. The algorithms related to the clustering and scheduling components are developed as specified by the framework. Recall that these components use messages to drive their execution. With respect to this, two messaging protocols have been conceived: *IPPROTO_DCP* and *IPPROTO_IPP*.

IPPROTO_DCP This is a distributed clustering protocol (DCP) based on TCP/IP. It defines how a node handles or generates messages related to clustering operations. Basically, there are two scenarios in which the protocol is involved: emission and reception of clustering messages. The first scenario occurs when a node is joining the cluster tree. It needs to emit *IPPROTO_DCP* packets to interact with other nodes.

The second scenario applies to head nodes. Such nodes have the responsibility to guide new nodes into a position in the cluster tree. They receive, process and respond to *IPPROTO_DCP* packets. Listing 5.3 shows how a *IPPROTO_DCP* packet is constructed.

Listing 5.3: *IPPROTO_DCP* packet.

```
#define IPPROTO_DCAP   string("DCAP")
ostringstream pckt_payload;
pckt_payload     << IPPROTO_DCP << '\n'
                 << "request_head:" << '\t'
                 << Clustering::thisNode_ID.ID << '\t'
                 << Clustering::thisNode_ID.node_depth << '\t'
                 << Clustering::thisNode_ID.weight_degree << '\t'
                 << Clustering::thisNode_ID.weight_uptime << '\t'
                 << Clustering::thisNode_ID.networkRange ;
```

IPPROTO_IPP This protocol was introduced in Section 3.3.1. It is an information policy protocol (IPP) based on TCP/IP. Once a node is part of the cluster, it is active and expected to perform resource advertisement operations. It must advertise its resource information as well as changes to this information. These changes are detected by probing the shared memory hosting the *SelfAttributes* object. Listing 5.4 shows how a *IPPROTO_IPP* packet is constructed.

Listing 5.4: *IPPROTO_IPP* packet.

```
#define IPPROTO_IPP        string("IPP")
ostringstream pckt_payload;
pckt_payload     << IPPROTO_IPP << '\n'
                 << NODE_OPT(H_OPTS_SHARED)->nb_of_cores << '\t'
                 << NODE_OPT(H_OPTS_SHARED)->ext_ram << '\t'
                 << NODE_OPT(H_OPTS_SHARED)->nb_of_gpu_cores << '\t'
                 << NODE_OPT(H_OPTS_SHARED)->gpu_ram << '\t'
                 << NODE_OPT(H_OPTS_SHARED)->disk_size;
```

The protocols above define how a node interacts with its peers at an informative level. This concludes the operational overview of the *InformationPolicy* module. Now, we need to perform a structural overview of the module. Fig. 5.12 shows the building blocks of this structure. They are implemented as *C++* classes, which are briefly described below.

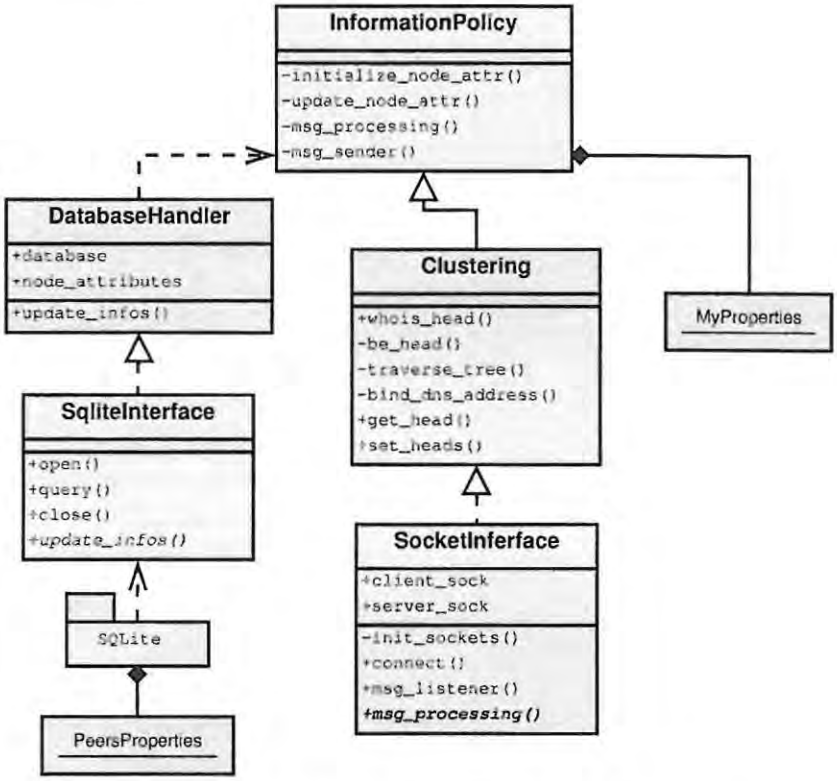


Figure 5.12: *InformationPolicy* class diagram.

InformationPolicy This is the main class of the module. It implements the clustering maintenance algorithm to keep the node active in the cluster tree (see Section 3.2.5). But first, it needs to initiate a clustering operation for the node to join this tree. We need to mention that the Fibonacci sequence was used as the "keep alive" period function. This means that during the maintenance phase (which involves *Ping*), the "keep alive" time increases logarithmically. An upper limit is set to avoid infinite increases.

Clustering This simply implements the clustering set up algorithm; i.e., a node uses this class to join the cluster tree (see Section 3.2.4). During this process, $IPPROTO\_DCP$ is used for the exchange of messages. The $Net :: DNS$ library in Perl is used to generate and send dynamic DNS update packets.

SocketInterface This class implements the needed network sockets using the Linux socket libraries. Specifically, it leverages the *epoll* feature present in Linux kernels to improve

efficiency. The *epoll* API differs from other APIs in that it does not pass the set of socket descriptors back and forth to the kernel. It sends (polls) the set to the kernel only once and use the *epoll_wait*() function to keep the set permanently (one can set a timeout) in the kernel and to wait for events. Since two μCloud nodes only exchange messages briefly, an event-based connection is recommended.
A P2P node has two self explanatory types of sockets: client and server sockets. An abstract class (*msgProcessing*) is prototyped to process packets from the listening socket (server). For reliability, these sockets are TCP based. It is of course possible to use UDP sockets instead.

DatabaseHandler As shown in Fig. 5.12, this class is based on a *C++* implementation of the SQLite library. It is used inside the *InformationPolicy* module to perform SQL operations on the *NodeProperties* object.

In brief, the *InformationPolicy* module runs as three processes. The first one is the parent process which executes an instance of the *Clustering* class. After the node becomes part of the cluster tree, it spawns the other processes and sleeps. The second process uses *IPPROTO_IPP* to advertise the information from the *SelfAttributes* object. It also implements the *Ping* algorithm for link failure detection. If a failure is detected, the parent process wakes up (for re-clustering) and the child processes are killed. The third process is a listener which differentiates and handles requests (e.g., *IPPROTO_IPP* or *IPPROTO_DCP* requests) on reception.

5.4.4 *Scheduler* class

This is a simple class that performs local resource discovery and reservation. It only operates with resources within the scope of the node. That is, the implemented resource discovery is limited to resources under the node's supervision. The class uses the *PeersAttributes* object to discover an appropriate resource provider. The discovery operation is presented, in its simplest form, in Listing 5.5. After this operation, relevant resources are leased for the intended CPS. This was explained in Section 3.3.3. To enforce reliability, the *Scheduler* class confirms the availability of a resource before it is leased.

Listing 5.5: Basic resource discovery using SQL. c, r, d are elements of the CPS tuple.

```
SELECT C.IPaddr, C.ext_ram FROM computing_info AS C NATURAL JOIN meta_info AS M
WHERE  C.nb_of_cores >= c AND \
       C.disk_size >= d AND \
       C.ext_ram >= r AND \
       M.timestamp >= Time(now)-time_out \
ORDER BY C.ext_ram DESC;
```

5.4.5 *Storage* class

μCloud manages disk images with the *Storage* class. Indeed, any storage related operation is performed by this class. It leverages the Ruby wrapper of the Libguestfs library to create and manage storage units. By fully harnessing the capabilities of the library, one can automate the installation of operating systems and miscellaneous applications on a disk image.
The *Storage* class is also capable of transferring storage units across the network. This is necessary during a service migration, for example.

5.4.6 *Coordinator* class

The expected functions of this class were presented in Section 5.3. The *Coordinator* manages services provided by a node, from their creation to their destruction. It is implemented as shown in the diagram in Fig. 5.11 and works closely with the *webServer* module. The latter forwards a request to the *Coordinator* when a CPS needs to be created in response to that request. The CPS is therefore configured at this point. Note that the *Coordinator* can initiate a service migration for the CPS if required.
Evidently, the system needs a means for migrating services. A transferring mechanism is implemented to accompany the *Coordinator*. This enables the system to safely transfer services between hosts while maintaining integrity and availability and at the fastest possible speed. Here, the VM operations involved are: save, migrate, and resume.

5.4.7 *WebServer* module

Recall that a Web server is used in μCloud to facilitate client to computing node interactions; i.e., a recent Web browser is the only necessary tool on the user side. The *webServer* module

is a Rack compliant implementation of the web server needed in μCloud. Rack is middleware for Ruby web servers [16]. It allows the *webServer* implementation to be ported across several web containers effortless. We chose *Thin* as the default web container because of its speed [36]. It is event-based, suitable for web (short) connections. Listing 5.6 shows the configurations in the *webServer* module.

Listing 5.6: *webServer* URI mappings. *Thin* can be replaced with any server supported by Rack.

```
thin = Thin::Server.new($myIP, $myPORT,{:signals => false}) do
  use Rack::CommonLogger
  use Rack::ShowExceptions

  map "/" do run WebPortal.new end
  map "/initiate" do run VMinitiate.new end
  map "/interact" do run VMinteraction.new end
  map "/download/disk" do run DiskDownloadHandler.new('Storage/users') end
  map "/upload" do run NonCachingUploadHandler.new end
  map "/include" do run Rack::File.new('webServer/noVNC/include') end
  map "/images" do run Rack::File.new('webServer/html/images') end
end
```

It should be mentioned that *Thin* is forced to handle the "/initiate" URI in a thread-based manner. This URI involves an actual service creation, which means that μCloud takes longer to formulate a response.
Structurally, the *webServer* module uses two distinctive classes:

RequestHandler All client requests are handled by this class. In fact, Listing 5.6 is is part of the code for this class. A notable type of request is that for CPS creation (the first three mappings in Listing 5.6). A *Coordinator* instance is created for a new CPS request. With the aid of the *Coordinator*, the *RequestHandler* decides which node is going to serve the client. If the node itself is capable (it has been chosen by the *Scheduler*), the *Coordinator* starts the operations for CPS provision. Otherwise, the client request is dispatched to other nodes.

RequestDispatcher This class implements the distributed resource discovery algorithm presented in Chapter 3. Client queries are forwarded appropriately to other peers to discover the required resources. *RequestDispatcher* uses the *NodesProperties* object to obtain information on nodes in order to perform the *dispatch* as defined by the discovery algorithm (e.g., nodes with high *uptime* value are preferred).

5.4.8 *WebInterface* package

This package contains files that construct the client interface. An overview is provided in Fig. 5.13. As mentioned earlier, any user-to-service interaction is done through HTTP. This is why the *webInterface* package is composed of *HTML* and *Javascript* files. Most of the *Javascript* files are used to build an HTTP remote desktop client. In fact, we use an HTTP to VNC proxy based on *Javascript*, called noVNC [50]. The latter uses the WebSockets API provided by the HTML5 specification to implement the RFB (remote frame buffer) protocol, which is the protocol used in VNC. It is recommended that the client use a recent browser to ensure support for HTML5.
The *.html* files are created for the clients to make and submit their requests. For instance, the CPS tuple is represented in an HTTP POST request constructed through the *index.html* file. Note that *∗.rhtml* files have HTML content that can be parsed by Ruby for editing. The *erb* library available by default in Ruby is used for this.

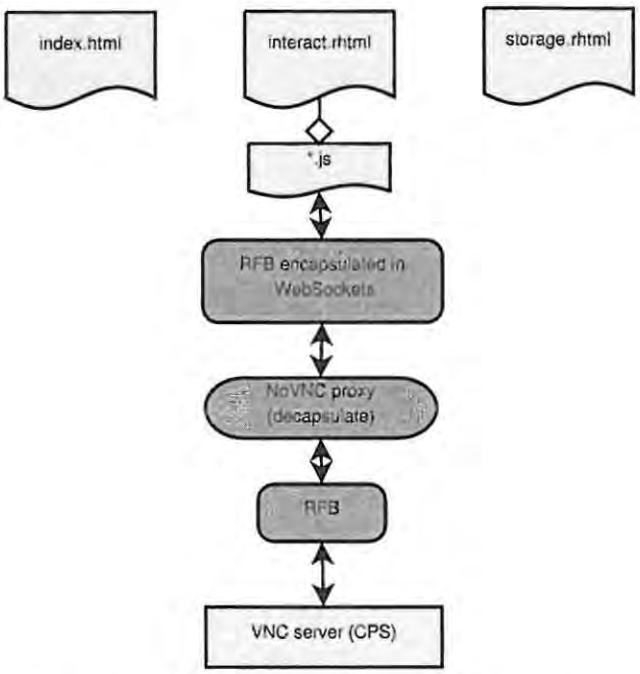


Figure 5.13: *WebInterface* package content.

In the next page, Fig. 5.14 shows screenshots of the *webInterface* in action.

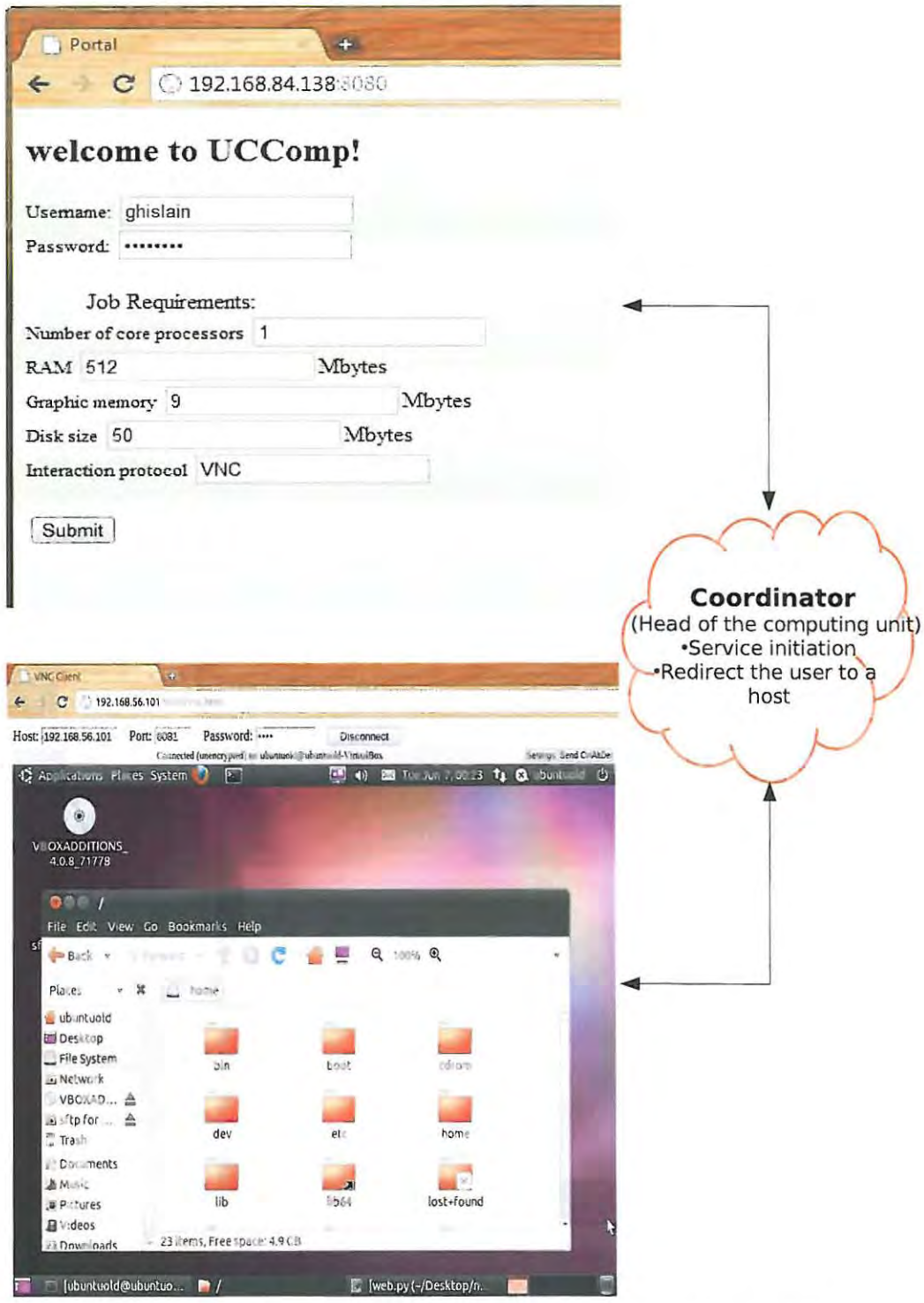


Figure 5.14: Client experience through the *webInterface* package.

5.5 Summary

This chapter presented a novel method for creating cloud services. To facilitate and enhance user computer experiences, a cloud is built within a P2P network. Peers form a cluster of potentially powerful computers (workers) on which external computations are virtually executed. The cloud, μCloud, merely provides the CPS system. This cloud has built-in intelligence to enable the dual property of unity (from a user point of view) and component independence (from a peer point of view). Peer nodes implement a set of modules that enable them to be considered as a unit, a computing unit; this is emergent behaviour. The computing unit has a single point of entry: the root node, which acts as the main coordinator of the unit and handles clients' initial interactions. Other client interactions are done directly from the client nodes to the worker nodes, thereby harnessing the power of P2P.
Note that most of the source code for modules comprising μCloud are not given in this thesis. To access the full implementation of μCloud, refer to the thesis web portal at `http://www.cs.ru.ac.za/research/g09f5474`.

Chapter 6

Simulations and Results

One research area in which actual physical testing is challenging is that of cluster systems (coarse grained architectures). The scale required to test such a system practically can only be achieved during mass deployment. In other words, we cannot perform a practical test of the required scale in a lab environment. Fortunately, simulation testing provides the means whereby large scale tests can be run in a repeatable manner without the need for physical deployment [33, 30]. Experiments of cluster systems can be effected using a few commodity workstations or PCs through simulations. For instance, using these commodity resources, a network simulation can be performed to provide a fair abstraction of a real network and related applications.

In relation to our research, our P2P framework is an important component of μCloud and needs to be evaluated separately on a large scale. The cost of creating and maintaining a cluster tree in a P2P network needs to be assessed. As such, we designed a testbed that involves simulating multiple nodes in multiple networks using a group of lab PCs.

6.1 Network Simulator

To test our clustering algorithms, we designed a simple and lightweight distributed network simulator, called μnesim (micro network simulator). Using VLAN technology, μnesim creates multiple virtual networks on a single machine. Doing this on multiple machines, we are able

to generate different network topologies with a variable number of nodes. Contrary to popular simulators, which have their own TCP/IP stack implementation, our simulator uses the native operating system TCP/IP stack. This means that our P2P framework implementation in μCloud (the *InformationPolicy* module) can be tested without changing any part of its code. Although, due to this, μnesim may require more computing resources than certain free simulators, it has the advantage of ease of use.
It should be mentioned that the network latency that occurs in real networks is not simulated by μnesim. This is because network latency is not relevant to the evaluation of our P2P framework.

6.1.1 Apparatus

This section presents the hardware and software environments used to run the simulator.
The Department of Computer Science at Rhodes University administers a number of computer labs. One of these labs, with a capacity of 10 PC machines, was made available for our research during non lecture periods. Each PC has a quad core 2.6 GHz Intel CPU, 4 GB RAM and 500 GB hard disk. These computing capabilities were harnessed for our simulations.

In our setup, one PC is used as the server. The others (9 in total) netboot from this server PC, which contains an image of the Fedora 14 operating system they need to install. This image is customized for a PC to run μnesim together with instances of the μCloud *InformationPolicy* module at start up. In fact, the *InformationPolicy* module can be compiled as a stand alone component of μCloud. This is possible because it only interacts with other components via system shared objects (refer to the μCloud diagram in Fig. 5.11). In addition, the server PC runs a set of software servers (PXE, TFTP, DNS and DHCP servers) to ensure that other PCs are configured appropriately when they boot. It also runs a MySQL server that collects data during the simulations.

6.1.2 Methodology

As mentioned before, the role of μnesim is to create multiple network interfaces to simulate a network of multiple nodes. In other words, each PC run several virtual nodes. Each

of these nodes belongs to a particular virtual network (virtual site) and is identified by its virtual IP address. These virtual nodes are called netbots. Basically, the simulating methodology is to create and automate these netbots so that they act in the same way that physical nodes running μCloud would. A Ruby script is used to implement this. The script repeatably creates a virtual network interface with an instance of the *InformationPolicy* module running on it. This constitutes a netbot. Its execution is automated to start and end at given times, to simulate node churn.

Node churn

Poisson distributions are used to create node dynamics in the simulated environment. That is, the simulator implements a Poisson distribution with a given Poisson rate, which is used to determine the churn behaviour of a node in the system (*join* and *leave* times). To ensure this behaviour varies for different simulations, the following Poisson rates can be used: 2, 4, 10, 19. Fig. 6.1 depicts the probability mass function for each of these rates, where λ denotes the given rate. μnesim uses a Ruby library developed in [49] to implement the Poisson distributions.

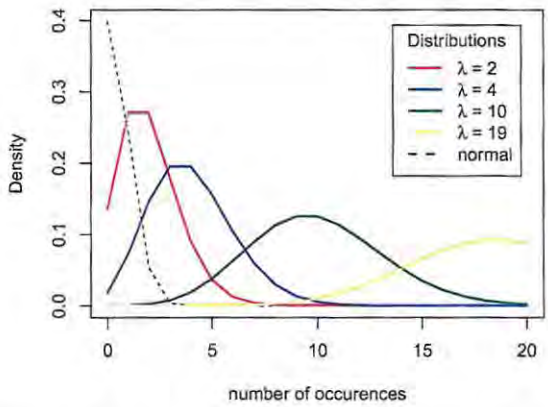


Figure 6.1: Probability mass functions for different Poisson distributions.

6.2 Data Collection

An important part of an experiment is the actual collection of data. Indeed, after a netbot has been created, its behaviour needs to be observed and recorded. Evidently, the nature of the

records depends on the nature of the experiment. Since we are testing a P2P framework, the data was collected in accordance with the Failure Trace Archive (FTA) schema [4]. The FTA is a centralised public repository of traces of parallel and distributed systems. It provides a SQL schema for these traces. We changed a few entries in the schema to correspond with our tests. For example, the *event_type* attribute in the *event_trace* table is used for identifying clustering and re-clustering events. Our modified FTA schema is presented in Appendix B.1.

So, for our simulations, μnesim notices critical events (e.g., failures) during a netbot's lifetime and records them in a SQL database. Recall that this database is hosted on the server PC.

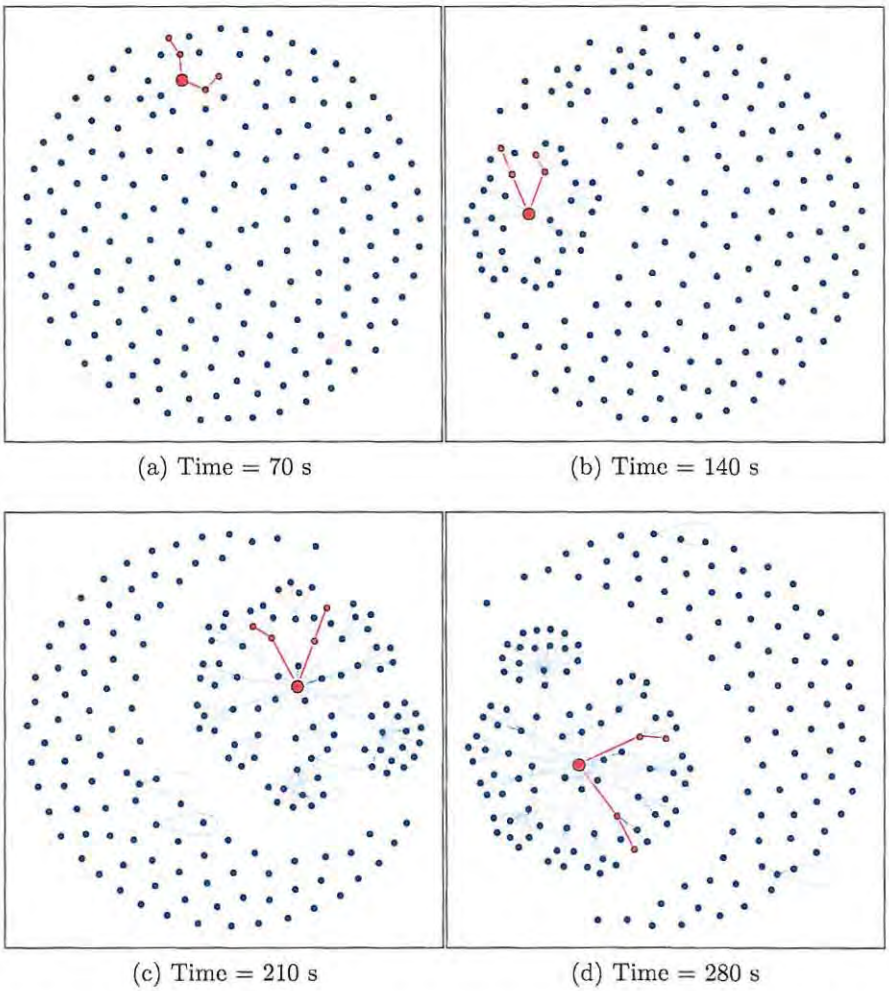
6.3 Results and Discussions

The set of data collected during the simulations were represented using the statistical programming language R. The language provides a wide range of facilities for visualising data, which are used to produce meaningful graphs, reflecting the nature of results obtained. This section uses descriptive and relative statistics to analyse these results. The observations and conclusions from this analysis are presented in subsequent discussions.

6.3.1 Cluster topology

We begin with an exploratory analysis of the hierarchical distributed clustering algorithm, H-DCA. The latter treats a network of nodes in a particular way to ensure clustering. So for a given simulation, we take snapshots of the network periodically and represent it as understood by the H-DCA. The *igraph* library available in *R* is leveraged to depict the structure of the network [37]. The snapshots in Fig. 6.2 show particularly how the H-DCA reacts to join events. Nodes with no edges are offline and trigger a join event when they become alive.

Discussion (Fig. 6.2) It is relevant to mention that the graphs were plotted using a class of the force-directed algorithm known as "Fruchterman Reingold". Effectively, nodes within the same site cluster together like physical objects in the same force field. In other words, site heads dominate the network graph by attracting nodes within their vicinity. The structure



(a) Time = 70 s (b) Time = 140 s

(c) Time = 210 s (d) Time = 280 s

Figure 6.2: Topologies of the network cluster at 4 different times.

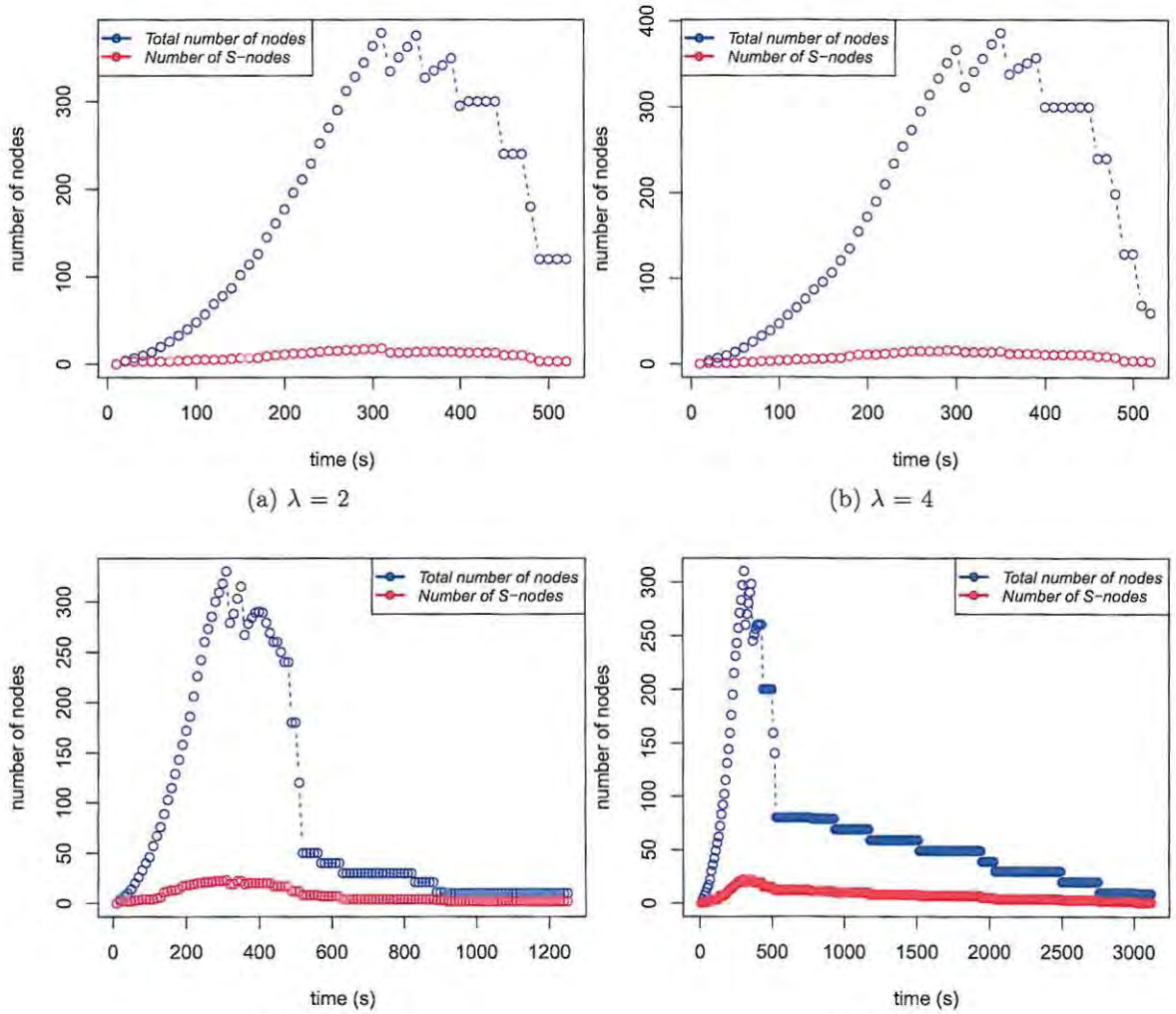
of the graphs depicted in Fig. 6.2 is therefore a sign that the clustering protocol operates as intended. The sequence of these graphs shows the development of the cluster tree over time. As more nodes decide to join the network, the tree evolves by accommodating these nodes. During this process, the tree maintains its structure, thus preserving its scaling property.
We also observe that a K-consistency of 1 is ensured. The cluster is "always" fully connected; there exists at least one path from any node to every other node in the network. The red line in each graph depicts its diameter. This shows the longest path a node discovery query would take to traverse the graph. We deduce from the length of this path that the H-DCA enables the "low stretch" property in the cluster. In Fig. 6.2d, the diameter of the cluster is

4 "edges long", which is 2 times the depth of the cluster.
Additionally, a vertex with a loop edge represents a node that supervises itself. This should be true only for the root node. But, we observe that there are multiple and/or isolated root nodes in the third graph of Fig. 6.2. These nodes appear because of race conditions at the root position. This situation is of course temporary as the nodes will realize their status and re-cluster. Some isolated clusters are also observed in the first two graphs. This stems from race conditions that occur when plotting the graph during a high join rate.

6.3.2 Cluster statistics

For different values of λ (the churn rate), we recorded the different states of nodes in the network. There are basically two states for a node: site head or site node. We examine how the two types of nodes compare statistically in different networks, with different node dynamics. In the following, consider T-nodes to denote the site nodes and S-nodes the site heads. Figs. 6.3 and 6.4 plot the total number of nodes N (T-nodes and S-nodes) and the number of S-nodes in the network at different time intervals for experiments with λ set to 2, 4, 10, and 19.

Discussion (Fig. 6.3) With $\lambda_{join} = \lambda_{fail} = \lambda$, the difference between the number of T-nodes against S-nodes over time indicates that the clustering protocol is effective and stable. As shown in Fig. 6.3, the H-DCA maintains a minimal number of S-nodes (a minimal dominating set) as the network scales. The size of the set stays constant for high values of N. This is because at such values, all the available sites have been discovered. Note that these observations are true for varying values of λ. The H-DCA is able to maintain or stabilise a large set of nodes under any given rate of the node dynamics.

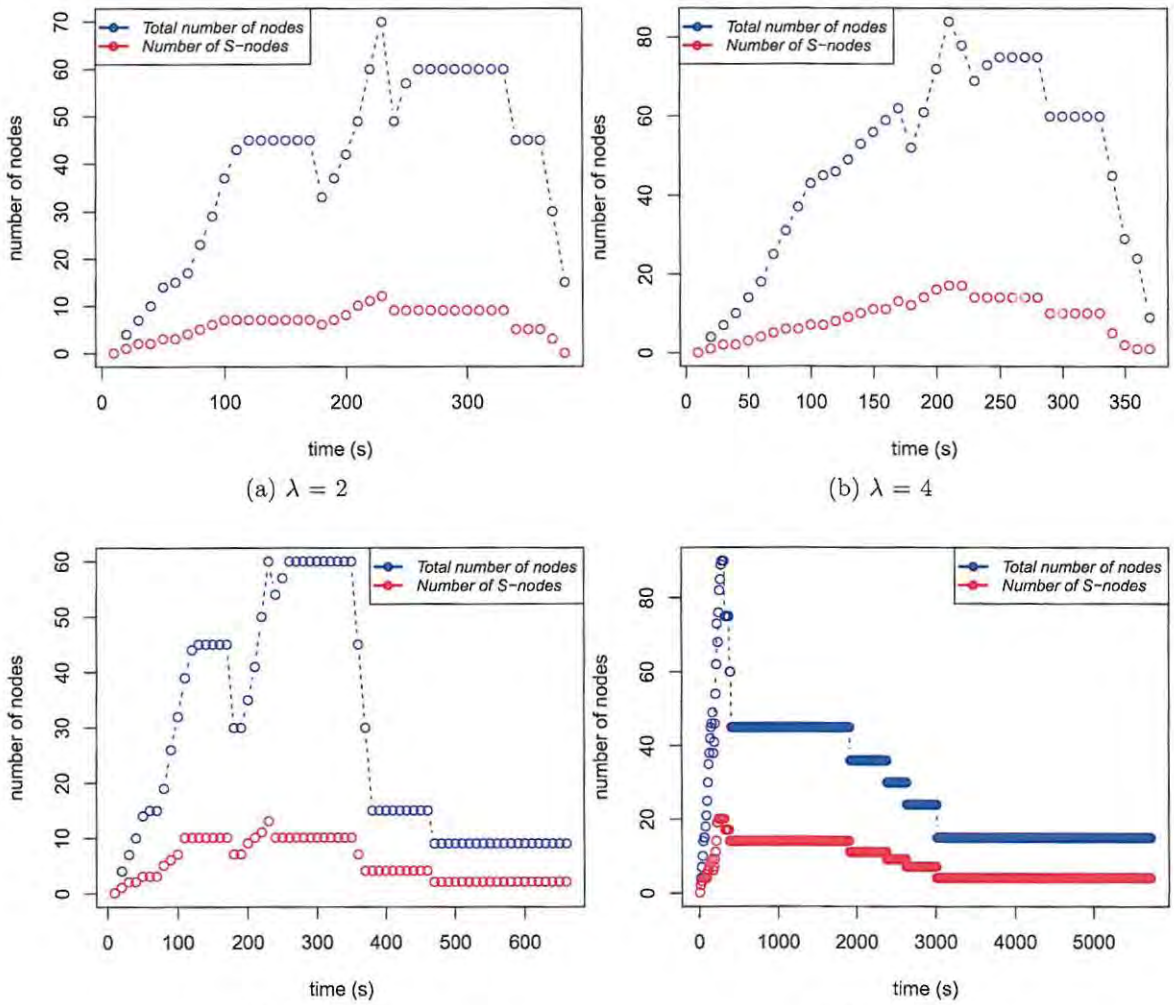


(a) $\lambda = 2$ (b) $\lambda = 4$

(c) $\lambda = 10$ (d) $\lambda = 19$

Figure 6.3: Number of nodes and S-nodes in the network over time. The maximum total number of nodes is 450 with a maximum of 20 nodes per site.

Discussion (Fig. 6.4) This figure is based on a smaller population of nodes. This is necessary to show how the number of S-nodes grows with respect to the total number of nodes. Because N is smaller, changes in its value are clearly noticeable in the graph depicting S-nodes. Indeed, most of the nodes in the network are S-nodes. In addition, more T-nodes become S-nodes (the two curves converge) for smaller numbers of N (this is observed in both Figs. 6.3 and 6.4).



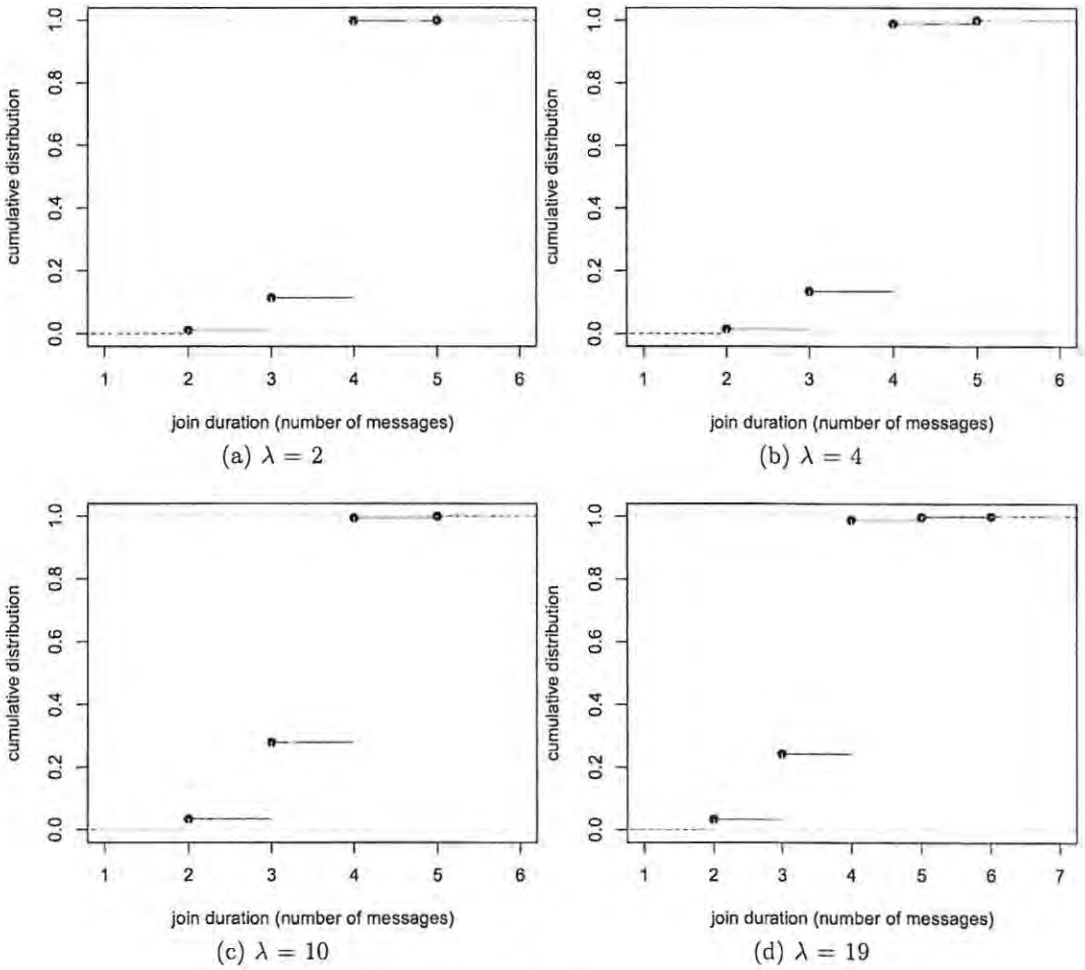
(a) $\lambda = 2$ (b) $\lambda = 4$

(c) $\lambda = 10$ (d) $\lambda = 19$

Figure 6.4: Number of nodes and S-nodes in the network over time. The maximum total number of nodes is 90 with a maximum of 5 nodes per site.

6.3.3 Join duration

The join duration is the amount of time that has elapsed between a new node starting its clustering process and becoming an S-node or T-node. Because a network simulator is used for the tests, the join duration is expressed as the number of *clustering messages* required for a node to join the cluster tree.



(a) $\lambda = 2$ (b) $\lambda = 4$

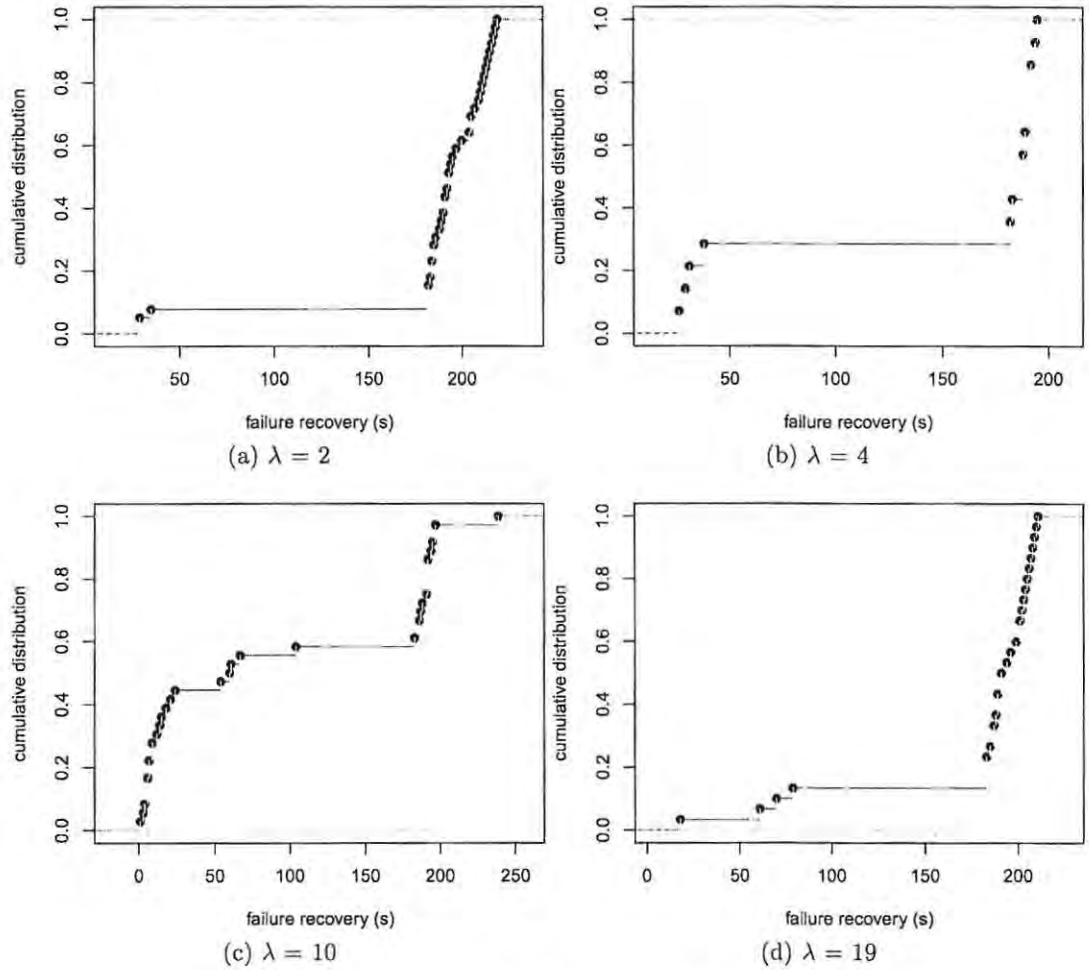
(c) $\lambda = 10$ (d) $\lambda = 19$

Figure 6.5: Cumulative distributions of join duration.

Discussion (Fig. 6.5) The average join duration is 3, which is the depth of the cluster tree. The reason for this is that the majority of nodes are T-nodes, situated at $leaf$ positions in the tree and it is expected from the hierarchical structure of the cluster. We can also notice from the graphs that the join duration is affected by the failure rate. The cumulative distribution for $\lambda > 4$ is greater, showing that join duration increases as the failure rate increases. This is because as nodes withdraw from the network more frequently, a new node needs to send more messages to perform a stable join.

6.3.4 Failure detection

During the experiments, we also observed how the network reacts to node failures. Fig. 6.6 analyses the durations between node failures (holes in the tree) and the reaction to these failures (a new node filling the empty position).



(a) $\lambda = 2$ (b) $\lambda = 4$

(c) $\lambda = 10$ (d) $\lambda = 19$

Figure 6.6: Cumulative distributions of failure recovery.

Discussion (Fig. 6.6) For every node failure, we observe an acceptable value for recovery time. Recall that failure events are detected with respect to the *keep alive* timeout. This value should be chosen appropriately for minimal network overhead. Using a period of 25

s in our *keep alive* timeout function (see Equation (3.1)), we observe that average recovery times for λ equal to 2, 4, 10, and 19 are 185 s, 144 s, 94 s, and 178 s, respectively. This is considered an improvement on algorithms such as in [69], which have a minimum failure detection period of 150s. This also suggests that one can increase the timeout period for *Ping* in a bigger network to reduce the overhead.

Additionally, the overall cost of the recovery is low. A failure affects at most all of the immediate neighbours (one hop nodes). Other nodes are not affected because they are not even aware of the failure due to the hierarchical structure of the cluster tree.

Furthermore, Fig. 6.6 shows the relation between failure recovery duration and failure rates. The recovery process is longer for high failure rates. This is expected as the failure recovery process depends on the failure detection duration and the join duration (a node fills the hole in the cluster tree by triggering a join event). We showed in the previous section that join duration increases with failure rate. This causes a longer recovery duration for a higher failure rate.

However, the graph in Fig. 6.6a with a low failure rate has a relatively high recovery duration. This is because the failure detection in the H-DCA reacts slower as the network becomes more stable (low failure rate).

6.4 Summary

Simulated experiments were used to evaluate the clustering component of μCloud. They allowed us to observe the behaviour of μCloud in networks with variable scales. To simplify the interpretation of the results, the observations for the cluster topology, node types, join duration and failure recovery were graphically represented in Figs. 6.2, 6.3, 6.4, 6.5, 6.6, respectively. From these, we analysed different networking properties of our algorithms. The analyses were sufficient to prove that these algorithms work as expected. The cluster is stable, scalable to a large N, and ensures a K-consistency of 1.

To complete the observations presented here, an evaluation of different networks of nodes running μCloud is performed in the next chapter.

Chapter 7

Experiments and Results

In Chapter 5, we discussed and implemented a novel P2P cloud system: μCloud. We postulated that it is dependable, scalable, and effective in hosting computing services. The validity of this statement is evaluated in this chapter. The goal of the evaluation is to assess the functionality and performance of the operations involved in μCloud. Specifically, the creation of services and handling of node failures are observed. To do this, we set up an environment to run appropriate experiments. This environment is used to conduct tests and record measurements for the validation of the techniques employed by the μCloud system. Effectively, an empirical analysis is performed based on these measurements.

7.1 Experimental Design

This section presents the environment used as the testbed and explains how relevant data was collected. The design used for the experiments is similar to the one presented in the previous chapter.

7.1.1 Testbed architecture

Among others, the Department of Computer Science at Rhodes University administers a lab of 46 PC machines, which is available for research during term vacations. Each PC has a

quad core 2.6 GHz Intel CPU, 4 GB RAM and 500 GB hard disk. These PCs are used as peer nodes in the testbed architecture. An external PC is used as the server. The peer nodes netboot from this server PC, which contains an image of the Fedora 14 operating system they need to install. This image is customized for a PC to run μCloud at startup. In addition, the server PC runs a set of software servers (PXE, TFTP, DNS, and DHCP servers) to ensure that other PCs are configured appropriately when they boot. It also runs a MySQL server that collects data during the simulations.
To summarise this, Fig. 7.1 depicts the diagram for the test environment.

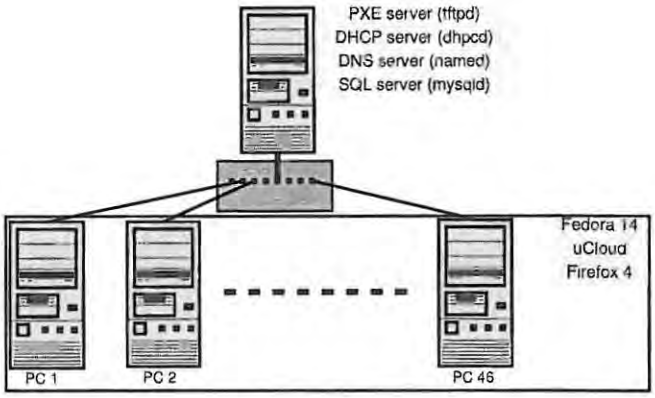


Figure 7.1: Testbed architecture.

7.1.2 Methodology

Realtime experiments, with multiple client requests, are carried out to analyse the behaviour of μCloud. This is achieved with three distinct components. The first component is responsible for creating random behaviour in each node (e.g., random *join* time). The second component is responsible for automating client requests and the third component simply observes the behaviour in each node; i.e., data collection.

Node dynamics

This is similar to the approach employed in μnesim. It involves using Poisson distributions to randomize the churn behaviour of nodes in the network (random join and leave times). In every experiment, a Poisson rate is chosen randomly for each node. But in addition to the

churn behaviour, the shared resources for a node are also configured with random values. In other words, the number of cores and amount of memory shared by a node are determined randomly. With this, different node parameters (e.g., node lifetime, node shared resources) are established for different experiments.

Automating requests

A Ruby script was written to simulate a client's behaviour. It generates service requests automatically and interacts with the service when the request is successful. This aims to put the μCloud system under regular task workloads, with varying task requirements and task execution times.
The script is based on the Watir (Web Application Testing in Ruby) toolkit. Watir (pronounced water) is a free, open-source functional testing tool for automating browser-based tests of web applications [7]. This means our script conducts activities on a browser the same way people would. The Watir-WebDriver extension supports Chrome, Firefox, Internet Explorer, Opera. Firefox 4 was chosen for the experiments.

Data collection

During experiments, each node records events related to μCloud (e.g., CPS creation) in the server PC (MySQL server). As in Section 6.2, the records were collected according to our modified Failure Trace Archive (FTA) schema (see Appendix B.1). Records for each test were kept in separate SQL databases. The full set of databases can be found at `http://www.cs.ru.ac.za/research/g09f5474/experiments.zip`.

7.2 Results and Discussions

With our experimental setup, tests were repeated for variable periods and consequently, various results were obtained. Table 7.1 gives a summary of the results of all the experiments we conducted.

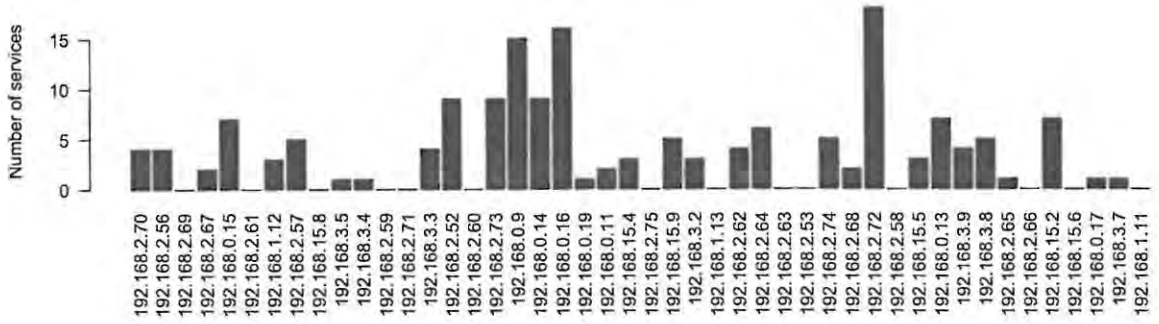
Experiment	Duration (s)	Number of nodes	Median node lifetime(s)	Number of queries	Number of CPS created	Number of CPS migrated	Average CPS creation latency(s)	Average CPS migration latency(s)
Exp1	34269	46	2574.031	1168	192	23	64.55660	194.34783
Exp2	60082	46	2246.833	774	167	12	65.24000	189.75000
Exp3	64963	46	1421.512	1514	375	51	72.07971	207.29412
Exp4	88045	46	1171.987	1737	459	44	62.27423	201.68182
Exp5	24981	46	1090.761	1547	357	19	62.34771	177.36842

Table 7.1: Summary of results for different experiments.

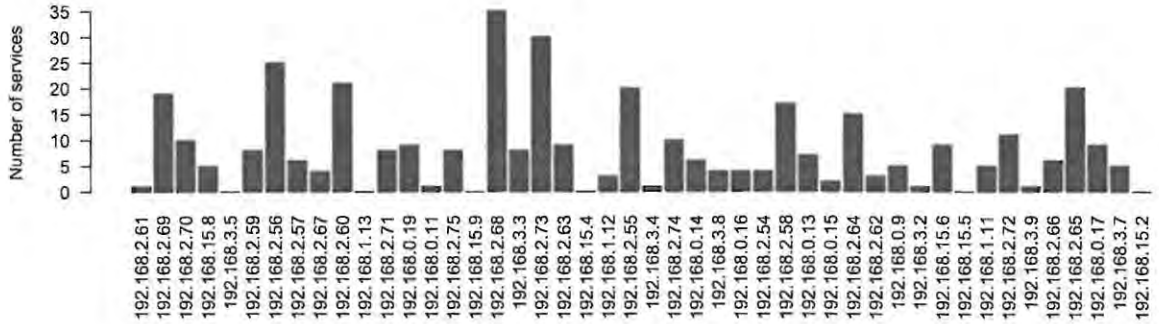
As in the previous chapter, we analysed the results obtained using the R language toolkit. The observations inferred from the graphs produced with the toolkit are presented in the following subsections.

7.2.1 Node workloads

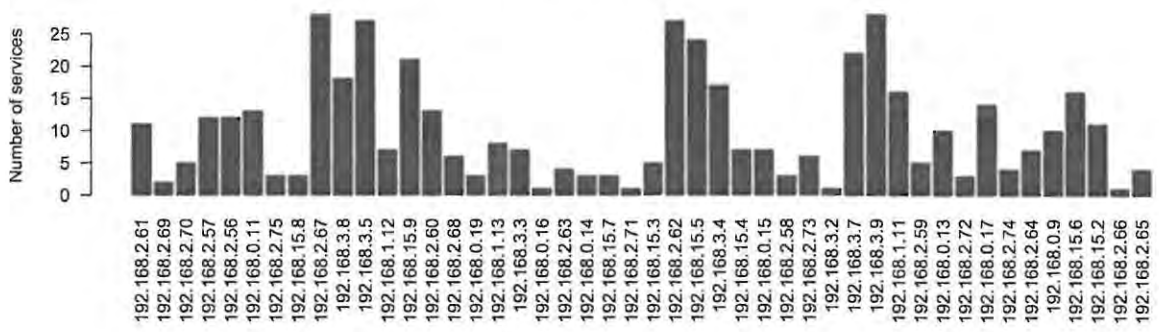
It is relevant to generate graphs that summarise the participation of nodes during the experiments. Nodes participate in μCloud in different ways (e.g., resource query routing). The most notable action of a node is the successful creation of a computing service. So, the total number of services created on each node during a test run is determined and represented in the bar plot in Fig. 7.2.
In addition, for various experiments, the median node lifetime and the median number of services provisioned in each node are determined and represented in Fig. 7.3.



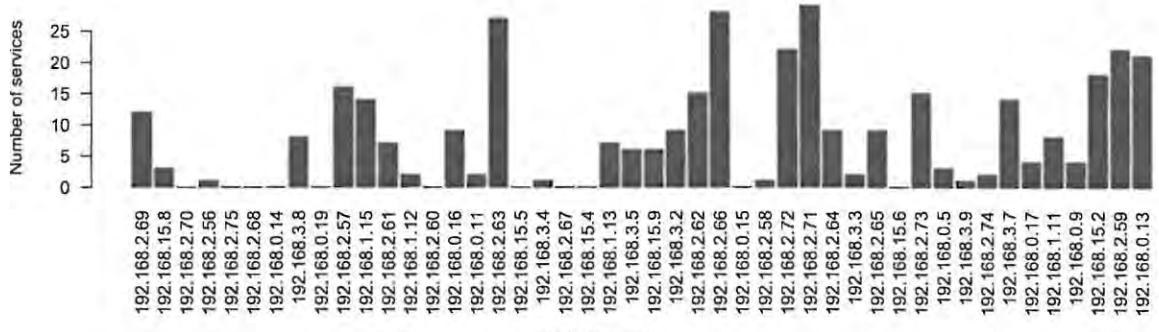
(a) Exp. 2



(b) Exp. 3



(c) Exp. 4



(d) Exp. 5

Figure 7.2: Number of services provisioned on each node for different experiments.

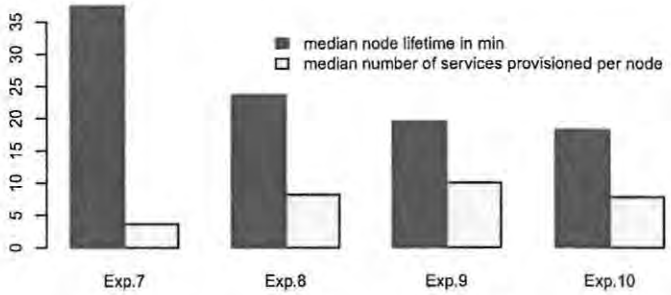


Figure 7.3: Median node lifetime and median number of provisioned services per node.

Discussion (Fig. 7.2, 7.3) The graphs show the distributions of workload among nodes for four of the experiments we conducted. These distributions prove two things. Firstly, they show that the setup used for testing μCloud operates as intended. Almost all the nodes involved in the experiments perform a service provision and the operation is recorded. Secondly, the distribution of multiple services across different nodes proves that μCloud operates distributively. It manages all the computing nodes at its disposal.
Note that nodes also participate in μCloud by performing resource discovery operations. These operations need to be performed for service provisions to occur.

7.2.2 Service provision success rate

Here, we evaluate a basic property of μCloud: its ability to leverage computing nodes to create computing services. For this, Figs. 7.5 and Fig. 7.4 are produced. The number of "rejected jobs" in the pie charts stands for the number of requests that could not be satisfied due to lack of sufficient resources. Conversely, the number of "accepted jobs" denotes the number of requests that μCloud could potentially satisfy. As these variables are only affected by the availability of shared resources during service requests, μCloud capabilities cannot be questioned based on their values.

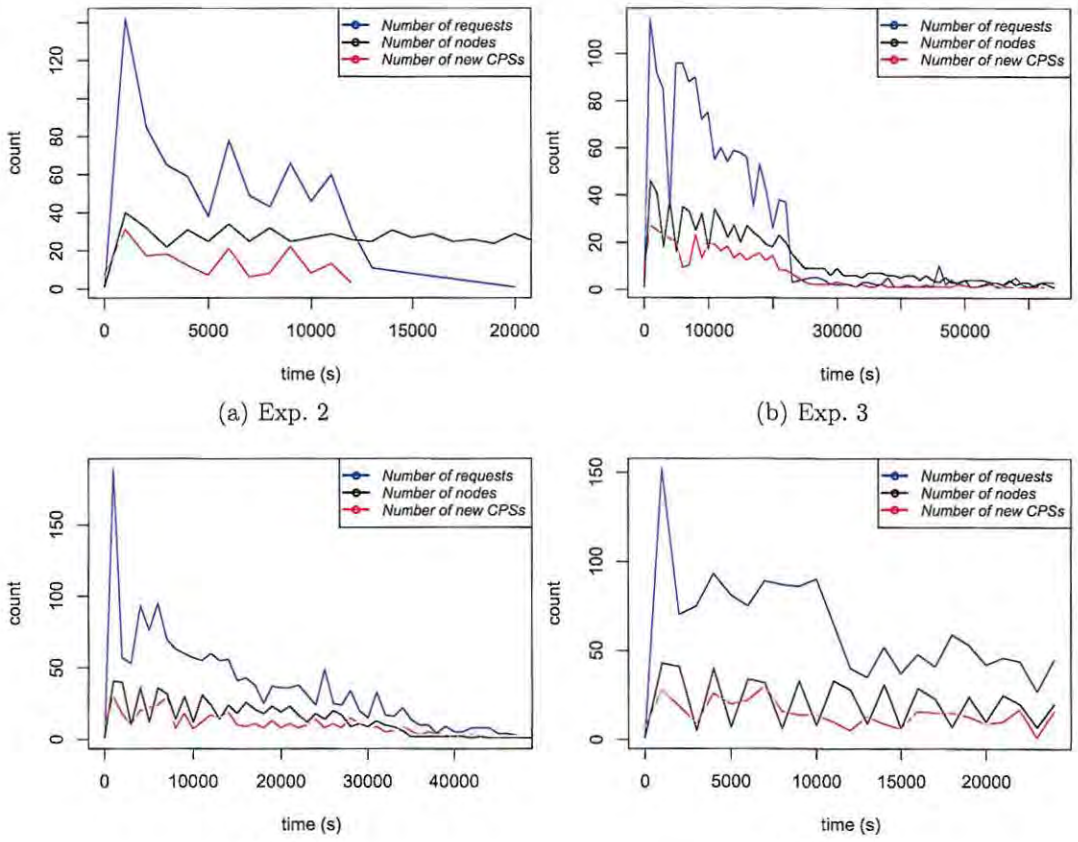
(a) Exp. 4

(b) Exp. 5

Figure 7.4: Pie charts of CPS creation related operations.

Discussion (Fig. 7.4) The charts in the figure need little explanation. They merely show the overall success of μCloud in creating services with respect to the demand. The latter is the number of client requests. As mentioned before, a request is accepted when appropriate resources are available else it is rejected. After a request is accepted, a service creation follows in order to run client jobs. However, there are instances whereby services are not successfully created ("failed jobs" in the charts). The best explanation we have for this involves Libvirt being overloaded with VM creations. Nevertheless, a maximum percentage success of 96.2% ("started jobs" over "accepted jobs") is achieved in Exp. 5.

Discussion (Fig. 7.5) For different time intervals, the number of requests, number of nodes, and number of services created are computed. This leads to two deductions. Firstly, the variance of the graphs for different numbers of nodes and requests in different experiments proves the success of our testbed in creating diverse test environments. This enables us to perform a good evaluation using the results. Secondly, by observing the graphs for the number of nodes and number of new CPSs, we realize that they converge in every experiment. This is an obvious expectation as new services depend on available resources.



(a) Exp. 2

(b) Exp. 3

(c) Exp. 4

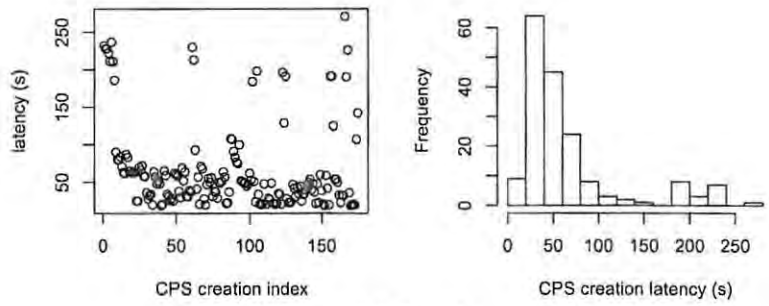
(d) Exp. 5

Figure 7.5: Number of CPSs created in different time intervals.

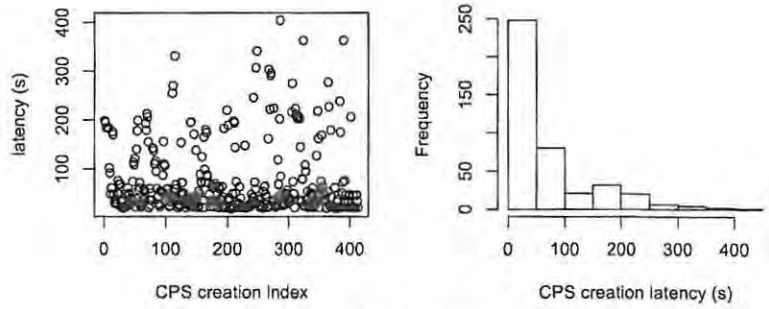
7.2.3 Service creation latency

For the experiments, we customized the CPS to run the same software (e.g., same kernel and filesystem) for all client requests. All the CPSs use the same disk space. The only varying attributes are the number of cores and the amount of RAM used by a CPS, which are specified by the client. This allows us to discard the "disk space" metric in the forthcoming discussion. It is obvious that CPSs with higher disk requirements would take longer to be created. So by using the same disk space for all the CPS instances, we analyse other metrics that affect the latency of a CPS creation.
The graphs in Fig. 7.6 represent the time taken by μCloud to successfully create CPSs in response to client requests. With these graphs, the latency between a job submission and its execution (start of the corresponding VM) is examined.

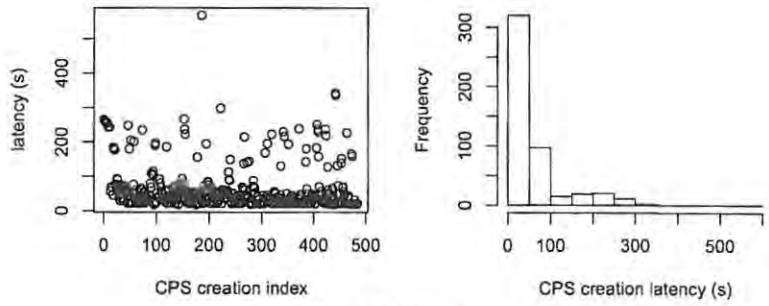
Discussion (Fig. 7.6) A noticeable observation is the high frequency of low CPS creation latencies. In fact, the creation latency is mostly affected by the operation through which the disk space required for a CPS is reserved. μCloud performs this reservation using the *guestfs* toolkit, which is implemented with a "cache" optimization. Therefore, a few CPSs have high creation latencies because μCloud reserved their required disk space by calling *guestfs* in a "cold cache" state. But once *guestfs* is executed, it caches related disk operations and results to perform faster in future executions.
With this inherited disk optimization, the minimal, average, and maximum CPS creation durations are respectively calculated to be 22s, 62.2s, 340s in Exp. 4. Considering the frequency of low duration values, it can be said that μCloud generally provisions services with minimal latency.
We also need to mention that we could not identify any variation in service creation latency with respect to number of nodes present in the computing unit. This is expected since μCloud operates as a P2P platform.



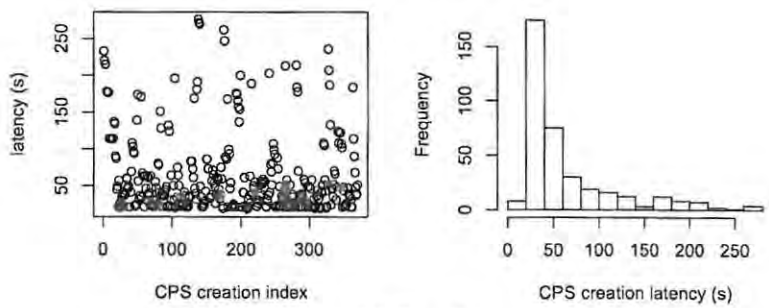
(a) Exp. 2



(b) Exp. 3



(c) Exp. 4

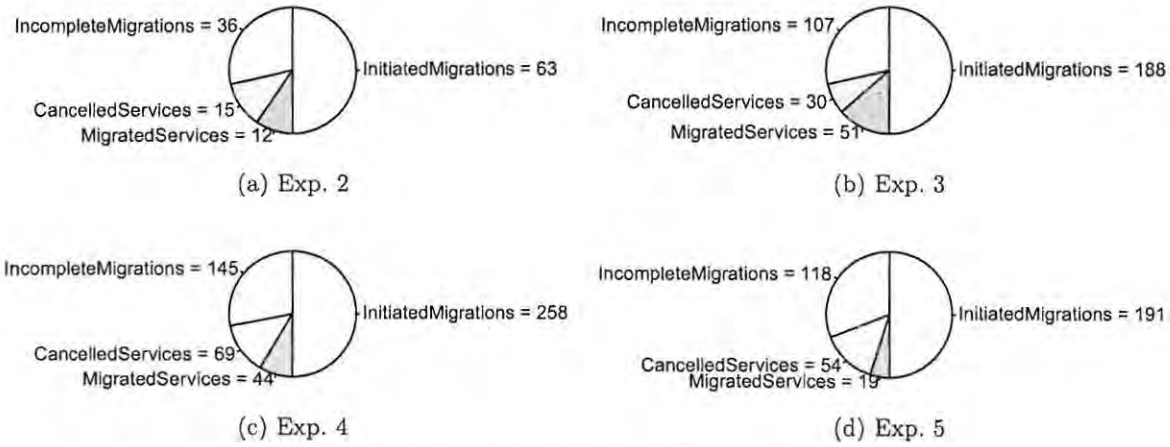


(d) Exp. 5

Figure 7.6: CPS creation latencies and corresponding histograms.

7.2.4 Failure handling success rate

Node failures create holes in the network, at least temporarily. Here we assess the reaction of μCloud in response to these failures. It is expected that it will attempt to save services hosted by a node when this node fails (due to a shutdown request). A "save" operation is carried out by executing a service migration. Note that the service migrations mentioned here are performed in realtime. That is, services are migrated from failing nodes ubiquitously such that the effect on client interactions is minimal (seamless service migration).



(a) Exp. 2 (b) Exp. 3

(c) Exp. 4 (d) Exp. 5

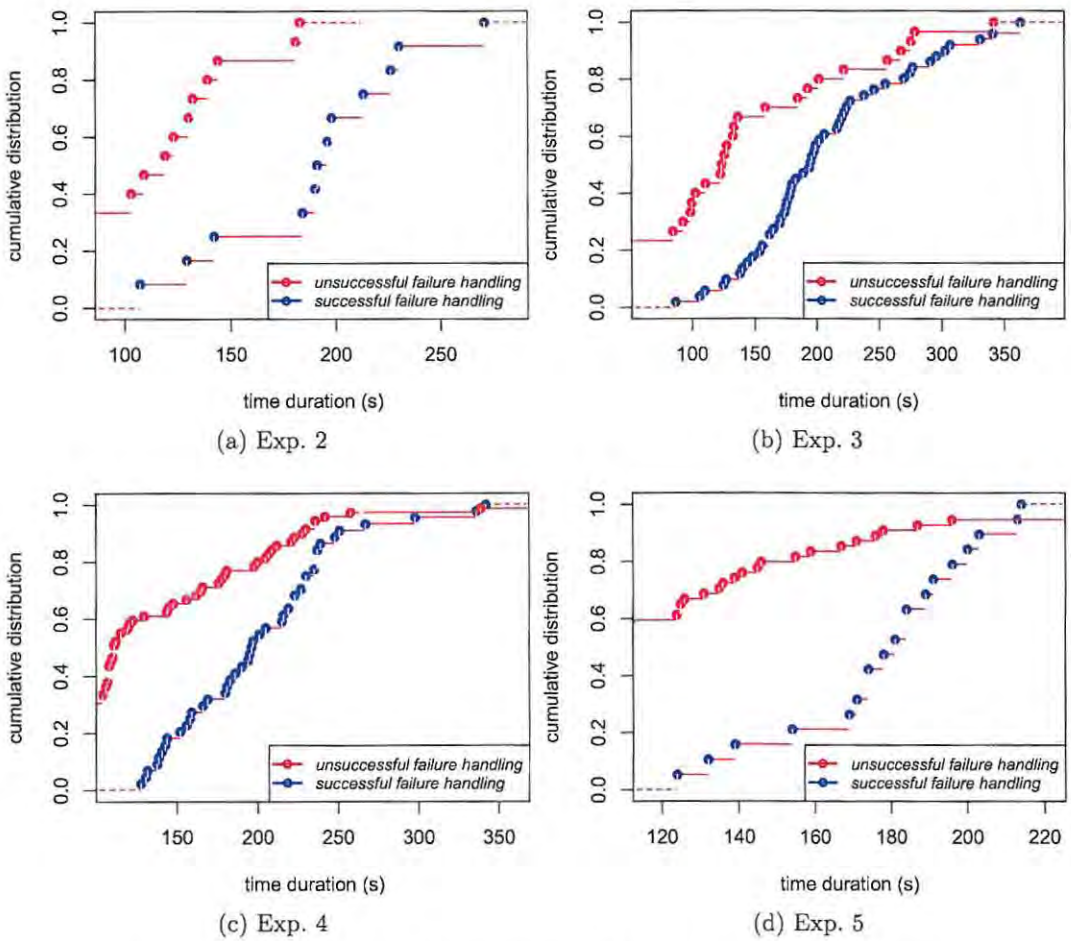
Figure 7.7: Pie charts of CPS migrations.

Discussion (Fig. 7.7) The charts summarise the measurements of service migration operations in μCloud. "Cancelled services" are those that could not be successfully migrated due to a lack of resources. "Incomplete migrations" represent services that were unsuccessfully migrated because there was not enough time for the operation; nodes failed too quickly. "Migrated services" represents the effort of μCloud to save services in realtime. With this, we observed that a minimum percentage of 56.2 % for failed migrations ("incomplete migrations" over "initiated migrations") is achieved in Exp. 4. This percentage value increases with the workload across nodes. Obviously, it is difficult to successfully migrate services in a busy computing unit. However, note that services are checkpointed and stored on the client side when migration is unsuccessful.

7.2.5 Failure handling latency

The amount of time taken by μCloud to handle node failures in different experiments are determined and represented in Fig. 7.8. The amount of time for unsuccessful failure handling ("cancelled services") and successful failure handling ("migrate services") are both plotted. Recall that the response of μCloud to a node failure consists of two phases: service checkpointing and service migration. The addition of the time taken by both these phases constitutes the failure handling latency.

Discussion (Fig. 7.8) The distributions in each graph converge with longer time durations.

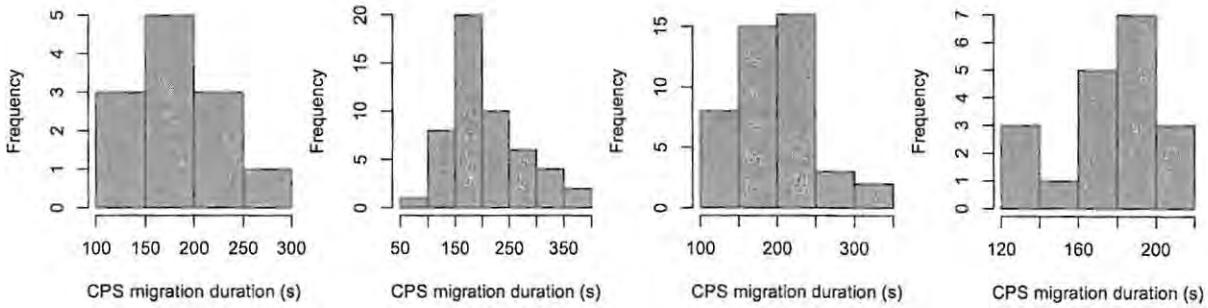


(a) Exp. 2 (b) Exp. 3

(c) Exp. 4 (d) Exp. 5

Figure 7.8: Cumulative distributions of CPS migration durations.

This means that service migrations that take longer have the same probability of being successful or cancelled. Indeed, a longer service migration duration means difficulties were faced by μCloud while performing that migration. These difficulties stem from the overall state of the computing unit and can cause migrations to fail. With respect to this, we identify two obvious possible issues in a computing unit: low average node lifetime and high average node workload.



(a) Exp. 2 (b) Exp. 3 (c) Exp. 4 (d) Exp. 5

Figure 7.9: Histrograms of CPS migration durations.

Discussion (Fig. 7.9) We find that the minimum of the average latencies for successful failure handling is 177 s in Exp. 5. In other words, on average, a μCloud node needs just over 3 mins to successfully migrate a hosted service to another node. This implies that the administrators of these nodes need to allow μCloud at least 3 mins before withdrawing their resources. The CPSs use this time to release their resources.

7.3 Summary

Different properties of μCloud were assessed in this chapter. We set up experiments to test the capability of μCloud to leverage shared computing resources and provision services according to client requests.

The graphs used in some of the discussions in this chapter were limited to certain experiments. These experiments were chosen because the graphs enabled us to highlight specific points in the discussions. Nevertheless, summaries of all the experiments conducted were generated and are presented in Table 7.1.

Chapter 8

Conclusion

8.1 Summary

In the introduction to this research, we set ourselves three main goals. These goals involved leveraging the potential of P2P systems for computing service provision. To facilitate the use of enhanced computational resources, we built a cloud using a network of volunteer nodes. Indeed, this thesis presented a novel cloud delivery model by explaining how cloud services can be created and provisioned in a P2P network. The contributions of this work are summarised in the following points.

- The principal contribution is the development of an approach to cluster nodes in a pyramid structured hierarchy, for appropriate cooperation. This cooperation leads to the emergence of a uniform and robust resource provisioning platform. Indeed, this platform relies on a cluster of potentially powerful computers (workers) to virtually execute external computations.

  Our algorithm clusters an adhoc network, making it dependable and reliable. The proposed algorithm (H-DCA) combines both weight (uptime, centrality degree) and location criteria to offer a simple implementation. It succeeds in cluster formation and guarantees fast inter-node communication to facilitate node discovery (this discovery is proven to be logarithmic in Section 3.4.3). By implementing the H-DCA, peer nodes operate as a computing unit; this is emergent behaviour. Note that even though the

H-DCA works for an adhoc network with any degree of churn, it is practically more efficient for semi adhoc networks (with a lower degree of churn).

- One of the goals in this thesis was to develop a service that would enable users to benefit from the computing capabilities of a peer node. The computing power service (CPS) presented in Chapter 4, was implemented for this purpose. Based on virtualization, the CPS provides a ready-to-use and remotely accessible desktop environment. Through this service, users can interact seamlessly with their computing environment.

- From the previous points, it is apparent that a cloud can be built to leverage a P2P computing unit and provide CPS services adequately. Such a computing unit has built-in intelligence that gives it the dual property of unity (from a user point of view) and component independence (from a peer point of view). This research provides an implementation of a P2P cloud platform called μCloud, which is a distributed CPS provisioning system based on P2P nodes. Users interact with this system Rest-*fully*, by submitting their computing requirements (through HTTP POST requests) and obtaining an interface to a relevant CPS. This system has the following advantages:

  High availability By clustering many computers, we can design an environment that provides reliability and high availability, which is key to applications like web servers or databases.

  High performance A high performance infrastructure is needed in many institutions (e.g., research institutions, hospitals). Performance can be achieved through the potential parallelism offered by the system.

- To validate our work, experiments were conducted to evaluate the μCloud system. This evaluation was carried out in two phases: large scale simulations to observe the clustering behaviour of μCloud nodes and practical tests to observe the capability of μCloud to provision CPSs. The results obtained indicate that μCloud optimally leverages the computing potential of nodes to successfully provide CPSs based on user requests. This P2P potential was exploited successfully, while maintaining some client-server properties like administration/management of resources.

8.2 Future Work

A noticeable limitation of μCloud is its capability to "live migrate" services (in realtime) when a node fails. We feel this capability can be improved somewhat, even though μCloud's success in "live" service migration depends, naturally, on the availability of computing resources. Moreover, a service migration depends on the service itself. Resource intensive services are difficult to migrate in realtime. So, in general, when migration is not successful, μCloud has to queue services and save them on the client side. These services can be restarted at a later stage, when the required resources are available.

In addition, there are a few issues relevant to a P2P cloud that we did not address in this research. Below, we explain how studying these issues can help to improve μCloud.

- Promoting collaboration in the P2P network. A rewarding system can be used to encourage collaboration between nodes. Active peers are rewarded for sharing their resources. For instance, let us define a "donation" as the amount of resources shared for a specified amount of time. The rewarding system will have to determine the cost/value for each donation and then reward the donor accordingly. A simple example of a reward is giving a former donor priority to benefit from others' donations. This will promote fairness in the P2P system.

- Add support for long term resource reservation. For a better service to clients, μCloud should allow computing resources to be leased "long" (weeks) before their actual use. This naturally requires a good prediction engine in the μCloud scheduler.

- μCloud can be extended to implement VOCs (virtual organization clusters) for better administrative management of its computing unit, which is a P2P cluster. This suggests a well federated P2P cloud. Moreover, in this research, we did not openly attempt to solve cloud related security issues. A trusted environment may be needed to ensure confidentiality for certain client operations.

References

[1] Berkeley open infrastructure for network computing. Online. Available from: http://boinc.berkeley.edu.

[2] Clouds@home: Cloud computing over unreliable, shared resources. Online. Available from: http://clouds.gforge.inria.fr/pmwiki.php.

[3] Computing as a Service (CaaS). Online. Available from: http://www.verizonbusiness.com/Products/it/cloud-it/caas/.

[4] Failure Trace Archive. Online. Available from: http://fta.inria.fr.

[5] Libvirt: The virtualization API. Online. Available from: http://www.libvirt.org.

[6] SC2000 Awards. Online. Available from: http://www.sc2000.org/awards/index.htm.

[7] Web Application Testing in Ruby (Watir). Online. Available from: http://watir.com/.

[8] Business Adoption of Cloud Computing: Reduce Cost, Complexity and Energy Consumption. Online, September 2009. Available from: http://www.aberdeen.com/Aberdeen-Library/6220/RA-cloud-computing-sustainability.aspx.

[9] Kvm: Kernel based virtual machine. Red Hat, Inc., 2009.

[10] Library for accessing and modifying virtual machine images. Online, 2009. Available from: http://http://www.libguestfs.org/.

[11] Seeding the clouds: Key infrastructure elements for cloud computing. White paper, IBM Corporation, February 2009.

[12] Server virtualization and cloud computing: Four hidden impacts on uptime and availability. White paper, Stratus Technologies, June 2009.

[13] Service oriented infrastructure reference framework. Draft Technical Standard, The Open Group, April 2009.

[14] How-to: Linux and Windows virtualization with KVM and Qemu. Online, January 2010. Available from: `http://tuxradar.com/content/howto-linux-and-windows-virtualization-kvm-and-qemu`.

[15] AKBARI TORKESTANI, J., AND MEYBODI, M. R. Clustering the wireless ad hoc networks: A distributed learning automata approach. *J. Parallel Distrib. Comput. 70*, 4 (2010), 394–405.

[16] ALLEN, R. Rack: a Ruby Webserver Interface. Online. Available from: `http://rack.rubyforge.org/`.

[17] AMNON BARAK, AVNER BRAVERMAN, I. G., AND LADEN, O., Eds. *Performance of PVM with the MOSIX Preemptive Process Migration Scheme* (1996).

[18] ANDERSON, D. P., COBB, J., KORPELA, E., LEBOFSKY, M., AND WERTHIMER, D. Seti@home: an experiment in public-resource computing. *Commun. ACM 45* (November 2002), 56–61.

[19] ANDERSSON, H., AND SVENSSON, J. *Virtualization in a Mobile Environment: an Introduction to ParaVirtualization with XenARM.* PhD thesis, Lund University, March 2010.

[20] ANDRADE, N., BRASILEIRO, F., CIRNE, W., AND MOWBRAY, M. Automatic grid assembly by promoting collaboration in peer-to-peer grids. *Journal of Parallel and Distributed Computing 67*, 8 (2007), 957 - 966.

[21] BAKER, M. Cluster computing white paper. *Computing Research Repository cs.DC/0004014* (2000).

[22] BARAK, A., AND LA'ADAN, O. Performance of the MOSIX parallel system for a cluster of PCs. In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking* (London, UK, 1997), Springer-Verlag, pp. 624–635.

[23] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev. 37* (Oct. 2003), 164–177.

[24] BASAGNI, S. Distributed clustering for ad hoc networks. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks* (Washington, DC, USA, 1999), ISPAN '99, IEEE Computer Society, pp. 310–.

[25] BECKMAN, M. Cloud computing deep dive. Tech. rep., InfoWorld Media Group, September 2009.

[26] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 41–41.

[27] BETTSTETTER, C., AND KÖNIG, S. On the message and time complexity of a distributed mobility-adaptive clustering algorithm in wireless ad hoc networks. In *Proc. European Wireless* (Florence, Italy, February 2002), pp. 128–134.

[28] BOYKIN, P. O., BRIDGEWATER, J. S. A., KONG, J. S., LOZEV, K. M., REZAEI, B. A., AND ROYCHOWDHURY, V. P. A symphony conducted by brunet. *CoRR abs/0709.4048* (2007).

[29] BRADEN, R. Requirements for internet hosts - communication layers. RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298.

[30] BUYYA, R., RANJAN, R., AND CALHEIROS, R. N. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. *CoRR abs/0907.4878* (2009).

[31] CACHEIRO, J. L., FERNANDEZ, C., FREIRE, E., DIAZ, S., AND SIMON, A. Providing grid services based on virtualization and cloud technologies. In *Proceedings of the 2009 international conference on Parallel processing* (Berlin, Heidelberg, 2010), Euro-Par'09, Springer-Verlag, pp. 444–453.

[32] CAMPBELL, P. Securing an OpenMosix cluster. SANS Institute, January 2004.

[33] CASANOVA, H., LEGRAND, A., AND QUINSON, M. Simgrid: A generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference*

on Computer Modeling and Simulation (Washington, DC, USA, 2008), IEEE Computer Society, pp. 126–131.

[34] CHEN, Y. P., AND LIESTMAN, A. L. Approximating minimum size weakly-connected dominating sets for clustering mobile ad hoc networks. In *In Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)* (2002), pp. 165–172.

[35] COSTA, F., SILVA, L., AND DAHLIN, M. Volunteer cloud computing: Mapreduce over the internet. In *Proceedings of the Fifth Workshop on Desktop Grids and Volunteer Computing Systems - PCGrid'11* (2011), pp. 1850–1857.

[36] COURNOYER, M.-A. Thin: A fast and very simple Ruby web server. Online, January 2008. Available from: `http://macournoyer.wordpress.com/2008/01/03/thin-a-fast-and-simple-web-server/`.

[37] CSARDI, G., AND NEPUSZ, T. The igraph software package for complex network research. *InterJournal Complex Systems* (2006), 1695.

[38] DIKKEN, L. DynamicPVM: Task migration in PVM. Tech. rep., Shell Research, November 1993.

[39] DUBHASHI, D., MEI, A., PANCONESI, A., RADHAKRISHNAN, J., AND SRINIVASAN, A. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 717–724.

[40] ESTEVES, R. A taxonomic analysis of cloud computing. 1st Doctoral Workshop in Complexity Sciences ISCTE-IUL/FCUL, June 2011.

[41] FERREIRA, L., BERSTIS, V., ARMSTRONG, J., KENDZIERSKI, M., NEUKOETTER, A., TAKAGI, M., BING, R., AMIR, A., MURAKAWA, R., HERNANDEZ, O., MAGOWAN, J., AND BIEBERSTEIN, N. *Introduction to grid computing with Globus*, first ed. IBM Corp., Riverton, NJ, USA, 2003.

[42] GANGULY, A., AGRAWAL, A., BOYKIN, P. O., AND FIGUEIREDO, R. IP over P2P: Enabling self-configuring virtual IP networks for grid computing. In *Proceedings of*

20th International Parallel and Distributed Processing Symposium (IPDPS-2006 (2006), pp. 1–10.

[43] GRAFFI, K., STINGL, D., GROSS, C., NGUYEN, H., KOVACEVIC, A., AND STEINMETZ, R. Towards a P2P cloud: Reliable resource reservations in unreliable P2P systems. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2010), ICPADS '10, IEEE Computer Society, pp. 27–34.

[44] HAN, B., AND JIA, W. Clustering wireless ad hoc networks with weakly connected dominating set. *J. Parallel Distrib. Comput. 67*, 6 (2007), 727–737.

[45] HUGHES, D., AND WALKERDINE, J. A survey of peer-to-peer architectures for service oriented computing. *Network* (2010), 1–19.

[46] INC, A. *Amazon Elastic Compute Cloud (Amazon EC2).* Amazon Inc., http://aws.amazon.com/ec2/, 2008.

[47] KONSTANTINOU, A. V., EILAM, T., KALANTAR, M., TOTOK, A. A., ARNOLD, W., AND SNIBLE, E. An architecture for virtual solution composition and deployment in infrastructure clouds. In *Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing* (New York, NY, USA, 2009), VTDC '09, ACM, pp. 9–18.

[48] LAM, S. S., AND LIU, H. Failure recovery for structured p2p networks: protocol design and performance evaluation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2004), SIGMETRICS '04/Performance '04, ACM, pp. 199–210.

[49] LEIGHTON, J. Poisson Distribution Ruby Class. Online, September 2008. Available from: `http://poisson.rubyforge.org/`.

[50] MARTIN, J. VNC client using HTML5 (Web Sockets, Canvas) with encryption (wss://) support. Online. Available from: `https://github.com/kanaka/noVNC`.

[51] MERGEN, M. F., UHLIG, V., KRIEGER, O., AND XENIDIS, J. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev. 40*, 2 (2006), 8–11.

[52] MURPHY, M. A., ABRAHAM, L., FENN, M., AND GOASGUEN, S. Autonomic clouds on the grid. *Journal of Grid Computing 8*, 1 (2010), 1–18.

[53] PANDEY, A., PANDEY, A., TANDON, A., MAURYA, B. K., AND KUSHWAHA, U. Cloud computing: Exploring the scope. *ArXiv e-prints* (May 2010).

[54] PRINZ, V., FUCHS, F., RUPPEL, P., GERDES, C., AND SOUTHALL, A. Adaptive and fault-tolerant service composition in peer-to-peer systems. In *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems* (Berlin, Heidelberg, 2008), DAIS'08, Springer-Verlag, pp. 30–43.

[55] RAICU, I. *Many Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing.* PhD thesis, the Faculty of the Division of the Physical Sciences, University of Chicago, Illinois, March 2009.

[56] RAJARAMAN, R. Topology control and routing in ad hoc networks: A survey. *SIGACT News 33* (2002), 60–73.

[57] RAMACHANDRAN, K., LUTFIYYA, H., AND PERRY, M. Decentralized resource availability prediction for a desktop grid. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (Washington, DC, USA, 2010), CCGRID '10, IEEE Computer Society, pp. 643–648.

[58] RIMAL, B. P., JUKAN, A., KATSAROS, D., AND GOELEVEN, Y. Architectural requirements for cloud computing systems: An enterprise cloud approach. *Journal Grid Computing 9* (March 2011), 3–26.

[59] ROTHBERG, E., SINGH, J. P., AND GUPTA, A. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), pp. 14–25.

[60] ROY, S., HALDER, S., AND MUKHERJEE, N. A multi-agent framework for performance tuning in distributed environment. *ArXiv e-prints* (May 2010).

[61] SIBAI, F. N. Design and evaluation of low latency interconnection networks for realtime many-core embedded systems. *Comput. Electr. Eng. 37* (Nov. 2011), 958–972.

[62] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev. 41* (March 2007), 275–287.

[63] STEINER, M., EN NAJJARY, T., AND BIERSACK, E. W. Analyzing peer behavior in KAD. *Institut Eurecom* (2007), 1–28.

[64] SUBRAMANIAN, K. Rethinking The Cloud: From Client-Server To P2P. Online, July 7 2009. Available from: `http://www.cloudave.com/1899/rethinking-the-cloud-from-client-server-to-p2p/`.

[65] TRUNFIO, P., TALIA, D., PAPADAKIS, H., FRAGOPOULOU, P., MORDACCHINI, M., PENNANEN, M., POPOV, K., VLASSOV, V., AND HARIDI, S. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems 23*, 7 (2007), 864 – 878.

[66] TSAI, M.-J., AND HUNG, Y.-K. Distributed computing power service coordination based on peer-to-peer grids architecture. *Expert Syst. Appl. 36*, 2 (2009), 3101–3118.

[67] VINTER, B., ANDERSEN, R., REHR, M., BARDINO, J., AND KARLSEN, H. *Towards a Robust and Reliable Grid Middleware.* Nova Science Publishers, Inc., 2007.

[68] VOLKER TURAU, B. H. A self-stabilizing algorithm for constructing weakly connected minimal dominating sets. *Elsevier Science* (2010).

[69] WANG, H., TAKIZAWA, H., AND KOBAYASHI, H. A dependable peer-to-peer computing platform. *Future Generation Computer Systems 23*, 8 (2007), 939 – 955.

[70] WU, S.-J. Estimations of the parameters of the weibull distribution with progressively censored data. *Japan Statistics Society 32*, 2 (September 2002), 155–163.

Appendix A

Grid Discovery Systems

Table A.1: Qualitative comparison of grid discovery systems based on unstructured P2P architectures.

System	Architecture	Resource indexing	Query resolution
Iamnitchi et al. [23]	Flat P2P overlay network, including one or more peers per VO.	Each peer provides information about one or more resources.	Queries can be forwarded using different strategies: random walk, learning-based, best-neighbour, learning-based, and best-neighbour.
Talia et al. [54]	Flat P2P overlay network, including one OGSA-compliant Peer Service per VO.	Within each VO, a hierarchy of index services provides information about local resources.	Discovery messages are routed across Peer Services using a modified Gnutella protocol. Message buffering and merging techniques are used to reduce web service overhead.
Mastroianni et al. [35]	Within each organization, one or more nodes act as super peers.	A super-peer maintains metadata of all nodes in the local organization.	The set of super-peers to which a query is forwarded is determined on the basis of statistical information about previous discovery tasks.
Puppin et al. [42]	Nodes are grouped into clusters, where each cluster may include one or more super-peer nodes.	On each node, an agent publishes information about resources. The information is broadcast to all super-peers in the cluster.	The hop count routing index is used to select the neighbour super-peers with the highest probability of success.
Marzolla et al. [34]	Nodes are organized in a tree structured overlay network.	Each node maintains a condensed description of the resources present in the sub-trees rooted in each of its neighbouring nodes.	A multi-attribute query is decomposed into a set of sub-queries. The sub-queries are matched against the routing indexes and routed only to those neighbours whose indexes satisfy all the sub-queries.

Table A.2: Qualitative comparison of grid discovery systems based on structured P2P architectures.

System	Architecture	Basic protocol	Multi-attribute resource registration	Multi-attribute query resolution	Range query resolution	Load balancing
MAAN [8]	One DHT per attribute	Chord	Each attribute is registered in the appropriate DHT	Each sub-query is resolved separately and the results are intersected at the querying node. Single attribute dominated routing	Sequential	Uniform, locality preserving hash function. Value distribution is known in advance
Andrzejak et al. [6]	One DHT per attribute	CAN	Each attribute is registered in the appropriate DHT	Each sub-query is resolved separately, and results are intersected at the querying node	Flooding	Simple neighbour load exchange
SWORD [39]	Each attribute is assigned a different subregion of a common DHT	Bamboo	Each attribute is registered in the appropriate region of the common DHT	The query is sent to the sub-region of the most selective attribute, or an attribute chosen at random	Tree-like	Leave rejoin protocol. Customized hash functions
XenoSearch [50]	One DHT per attribute	Pastry	Each attribute is registered in the appropriate DHT	Each sub-query is resolved separately and the results are intersected at the querying node	Tree-like	None
Mercury [7]	One DHT per attribute	Symphony	All attributes are registered in every DHT	Lookup on the DHT of the attribute with the smallest range	Sequential	Periodical network sampling to find load-imbalances (leaverejoin protocol)
Schmidt et al. [48]	One DHT for all attributes	Chord	Point query to register the attribute	Ranged query contains unknown bits. Each step forwards query to neighbour with an additional common prefix bit. Forward twice for each unknown bit	Tree-like	Exchange of load between neighbours
Ratnasamy et al. [44]	A range dividing tree per attribute. All trees mapped in a single DHT	Any	Each attribute is registered in the appropriate tier	Each sub-query is resolved separately and the results are intersected at the querying node	Tree-like	Uniform hash and attribute range subdivision

Appendix B

SQL Schema of Experiment Records

Table B.1: Modified FTA schema. The entries that were modified are highlighted in bold.

platform	
platform_id	unique number identifiying this platform. Allows one to differentiate or pool together different platforms.
platform_name	name of the platform (e.g. "Berkeley NOW Lab Fall 1998")
platform_location	location name of the trace platform source (e.g. "Berkeley NOW Lab - Soda Hall 2nd Floor, USA, Planet Earth")
platform_type	type of the trace source (from single_machine, cluster, multicluster, grid, desktop_grid, volunteer_computing)
misc_notes	other notes relating to the trace platform

node	
platform_id	id of the parent trace description table
node_id	unique ID for this node
node_name	name of node
node_ip	IP address
node_location	location of the node (e.g. country, geographic coordinates)
timezone	time zone of the resource (second offset from GMT)
proc_model	processor name, model, version number
os_name	name and version of the resource OS
cores_per_proc	number of cores per processor
num_procs	number of processors for this resource
mem_size	number of bytes of memory
disk_size	number of bytes of disk space

node_perf	
metric_id	unique ID for performance metric (benchmark)
node_id	unique ID for this node
platform_id	ID of measurement platform that is the node parent
i_val	integer
f_val	float
s_val	string

component	
component_id	unique ID for this trace description platform
node_id	ID of the parent resource corresponding to this trace description
platform_id	ID of measurement platform that is the node parent
creator_id	ID of creator of this component trace data
node_name	node name
trace_start	when the trace event first appeared (UNIX time). Can be NULL if component was always available.
trace_end	when the trace event last appeared (UNIX time). Can be NULL if component was always available.
resolution	resolution of the traces in seconds

creator	
creator_id	ID of creator of this component trace data
component_id	unique ID for this component trace data
node_id	ID of the node corresponding to this trace
platform_id	ID of platform that is the node parent
creator	name(s) of the person(s) who recorded the event traces
cite	citation (bibtex, etc) for using the data from the event traces
copyright	copyright statement about how data set can be used

event_trace	
event_id	unique ID of event
component_id	unique ID for this trace description platform
node_id	ID of the parent resource corresponding to this trace description
event_state	state of event (for example, 0% - 100% for CPU availability)
node_name	name of node
event_type	**type of event (1 – > clustering, 2 – > re-clustering)**
event_start_time	start of this event (UNIX time)
event_end_time	end of this event (UNIX time)
event_end_reason	**reason the event type or state changed at the end of this trace (HTTP response codes are used)**

event_state	
event_id	unique ID of event state
component_id	unique ID for this trace description platform
node_id	ID of the parent resource corresponding to this trace description
platform_id	ID of measurement platform that is the node parent
i_val	integer
f_val	float
s_val	string