Ad hoc Cloud Computing

Gary Andrew McGilvary

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2014

Abstract

Commercial and private cloud providers offer virtualized resources via a set of co-located and dedicated hosts that are exclusively reserved for the purpose of offering a cloud service. While both cloud models appeal to the mass market, there are many cases where outsourcing to a remote platform or procuring an in-house infrastructure may not be ideal or even possible.

To offer an attractive alternative, we introduce and develop an *ad hoc* cloud computing platform to transform spare resource capacity from an infrastructure owner's locally available, but non-exclusive and unreliable infrastructure, into an overlay cloud platform. The foundation of the *ad hoc* cloud relies on transferring and instantiating lightweight virtual machines on-demand upon near-optimal hosts while virtual machine checkpoints are distributed in a P2P fashion to other members of the *ad hoc* cloud. Virtual machines found to be non-operational are restored elsewhere ensuring the continuity of cloud jobs.

In this thesis *we investigate the feasibility, reliability and performance of ad hoc cloud computing infrastructures*. We firstly show that the combination of both volunteer computing and virtualization is the backbone of the *ad hoc* cloud. We outline the process of virtualizing the volunteer system BOINC to create V-BOINC. V-BOINC distributes virtual machines to volunteer hosts allowing volunteer applications to be executed in the sandbox environment to solve many of the downfalls of BOINC; this however also provides the basis for an *ad hoc* cloud computing platform to be developed.

We detail the challenges of transforming V-BOINC into an *ad hoc* cloud and outline the transformational process and integrated extensions. These include a BOINC job submission system, cloud job and virtual machine restoration schedulers and a periodic P2P checkpoint distribution component. Furthermore, as current monitoring tools are unable to cope with the dynamic nature of *ad hoc* clouds, a dynamic infrastructure monitoring and management tool called the Cloudlet Control Monitoring System is developed and presented.

We evaluate each of our individual contributions as well as the reliability, performance and overheads associated with an *ad hoc* cloud deployed on a realistically simulated unreliable infrastructure. We conclude that the *ad hoc* cloud is not only a feasible concept but also a viable computational alternative that offers high levels of reliability and can at least offer reasonable performance, which at times may exceed the performance of a commercial cloud infrastructure.

Lay Summary

Cloud computing is the ability to run applications and consume computer resources (e.g processors, memory, storage) that are offered from other computers over the Internet, which then can be accessed anywhere with an Internet connection. Well known examples of cloud services are iCloud and Gmail, both served from clouds owned by Apple and Google respectively. In contrast to providing services, many clouds are designed to primarily offer computing resources. This allows users to run tasks in the cloud, as if these were running locally on their own computer, and pay for cloud usage.

Such clouds are typically offered from a set of co-located machines that are dedicated to providing the cloud. While this is an attractive method of offering services and resources, there are many who are unable to employ cloud computing and benefit from its advantages. For example, an organisation may have private data that cannot be run in a public cloud, they may not be able to afford cloud operating costs or have the ability to procure and support their own in-house cloud.

To provide a solution to this problem, we introduce and realize an *ad hoc* cloud computing platform that operates over an organization's or research institution's current set of computers to transform spare resources into a cloud. We outline how the *ad hoc* cloud ensures cloud continuity while operating over a set of potentially unreliable and unpredictable machines. We show that the *ad hoc* cloud is a feasible concept and that it offers high levels of reliability and performance, therefore making the *ad hoc* cloud a viable alternative to other available clouds.

Acknowledgements

Firstly, I would like to thank to my supervisors Malcolm Atkinson, Adam Barker and Ashley Lloyd for their guidance throughout this research. Also, I would like to thank my initial supervisor Jano van Hemert for giving me the opportunity to pursue a PhD. Without their help and effort, this thesis would not have been possible. Furthermore, I thank the EPSRC for their funding over the course of this research.

I also owe thanks to past and present members of the Data Intensive Research group: (Queen) Rosa, David, Chee-Sun, Michelle, Paul, Dave, Iraklis, Ole, Jos, Rob, Donald, Alessandro, Luca, Paolo, and Fan. Thanks for providing a fantastic research environment to work in. I would also like to thank the many visitors to the group, and in particular Miquel and Josep for giving me the opportunity to improve my knowledge of the Spanish language.

I would then like to thank the EPCC for their collaborative and supportive efforts over the last few years. In particular, I owe gratitude to Terry Sloan and the SPRINT team for their work on our joint research papers, and Maciej Olchowik for his support when performing experiments on the EDIM1 cluster. I also thank those who contributed to other joint research papers.

During my PhD, I had the opportunity to work at CERN. While working at CERN was an amazing experience, this was largely due to those I had the chance to meet, in particular: Adam, Juan, Tadek, Bernard, Rocio, Kakia, Vlad, Artem, Alberto, Archit and Utkarsh. May the B club continue and that we all finally meet up one day!

Penultimately, I would like to thank my friends for their support and much needed opportunities for procrastination during the last four years. And a special thanks to my girlfriend Lynsey McIlhone for her patience, especially during the writeup of this thesis. Finally, I would like to thank my family, and in particular, my mother for her support not only during the course of this research, but for the 25 years beforehand making it possible to reach this stage.

Declaration

I confirm that the work submitted is my own, except where work which has formed part of jointly-authored publications has been included. My contribution and the other authors to this work has been explicitly indicated below. I confirm that appropriate credit has been given within the thesis where reference has been made to the work of others. I confirm that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 3 is formed by the following publication that is solely my own work.

Gary A. McGilvary, Adam Barker, Ashley Lloyd and Malcolm Atkinson. *V-BOINC: The Virtualization of BOINC*. Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013).
(http://dx.doi.org/10.1109/CCGrid.2013.14)

Chapter 5 is formed by the following publication. The implementation of the tool was shared between myself and Íñigo Goiri and I solely made further extensions. I also solely wrote the publication.

Gary A. McGilvary, Josep Rius, Íñigo Goiri, Francesc Solsona, Adam Barker and Malcolm Atkinson. *Dynamic Monitoring and Management of Cloud Infrastructures*. Proceedings of the The 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013).
(http://dx.doi.org/10.1109/CloudCom.2013.45)

Appendix A is formed by the following publications. I solely researched and wrote the relevant sections of cloud cost investigation and resource underutilisation.

Ashley Lloyd, Terence Sloan, Mario Antonioletti, Gary A. McGilvary. *Embedded systems for Global e-Social Science: Moving Computation rather than Data*. Future Generation Computer Systems, 2012.
(http://dx.doi.org/10.1016/j.future.2012.12.013)

Michal Piotrowski, Gary A. McGilvary, Terence Sloan, Muriel Mewissen, Ashley Lloyd, Thorsten Forster, Lawrence Mitchell, Peter Ghazal, Jon Hill. *Exploiting Parallel R in the Cloud with SPRINT*. Methods of Information in Medicine, 2012.
(http://dx.doi.org/10.3414%2FME11-02-0039)

GMcGilvary

(*Gary Andrew McGilvary*)

Publications

Parts of this thesis have been published in the following journal and conference papers:

1. **Gary A. McGilvary**, Josep Rius, Íñigo Goiri, Francesc Solsona, Adam Barker and Malcolm Atkinson. *Dynamic Monitoring and Management of Cloud Infrastructures*. Proceedings of the The 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2013). (http://dx.doi.org/10.1109/CloudCom.2013.45)

   The work of this paper is presented in Chapter 5

2. **Gary A. McGilvary**, Adam Barker, Ashley Lloyd and Malcolm Atkinson. *V-BOINC: The Virtualization of BOINC*. Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013). (http://dx.doi.org/10.1109/CCGrid.2013.14)

   The work of this paper is presented in Chapter 3

3. Ashley Lloyd, Terence Sloan, Mario Antonioletti, **Gary A. McGilvary**. *Embedded systems for Global e-Social Science: Moving Computation rather than Data*. Future Generation Computer Systems, 2012. (http://dx.doi.org/10.1016/j.future.2012.12.013)

   The work of this paper is presented in Appendix A

4. Michal Piotrowski, **Gary A. McGilvary**, Terence Sloan, Muriel Mewissen, Ashley Lloyd, Thorsten Forster, Lawrence Mitchell, Peter Ghazal, Jon Hill. *Exploiting Parallel R in the Cloud with SPRINT*. Methods of Information in Medicine, 2012. (http://dx.doi.org/10.3414%2FME11-02-0039)

   The work of this paper is presented in Appendix A

5. **Gary A. McGilvary**, Malcolm Atkinson, Adam Barker, Ashley Lloyd. Optimum Platform Selection and Configuration for Computational Jobs. In All Hands Meeting, York, 2011. (Extended Abstract).

   The work of this paper is not presented however it did provide a foundation for the development of our ad hoc cloud computing platform.

Table of Contents

Chapter 1

Introduction

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [176].

Cloud computing has evidently allowed businesses, research institutions and personal infrastructure users to reduce their capital investment by reducing upfront investment in infrastructure and to convert previously inflexible operating costs such as electricity, cooling, maintenance and security, into costs that are largely incurred only when there is a revenue stream against which to charge them. This model therefore has caused a marked growth in the popularity of cloud computing and fuelled predictions that it will become the dominant computing paradigm.

Despite its popularity, and like cluster and Grid computing, resources and services are offered from a remote and dedicated infrastructure. This thesis focusses on introducing, developing and evaluating an alternative cloud platform, called the *ad hoc* cloud, to fill a gap where computational end-users (i.e users requiring the use of a computational platform) are unable to or are disadvantaged by making use of any dedicated or remote infrastructure.

1.1 Motivation

Nowadays, businesses and personal infrastructure users typically perform computational tasks on in-house private clusters, while many research institutions take advantage of both cluster and Grid computing. This is however changing due to the introduction of public and private cloud computing infrastructures [3, 26, 21, 31, 17] that have revolutionized the way these computational end-users can execute jobs. Cloud computing makes it possible to scale according to the demand, increase collaboration and share information easily, as well as potentially reduce operating expenses, access apparently unlimited computational resources and benefit from the other advantages cloud computing offers.

There are however many situations where these computational models are not suitable for an end-user's requirements:

- *The application or data cannot be moved to the public cloud:* the application's data may be too large to migrate to the remote platform and hence the end-user may find the migration process of little value due to its difficulty. The data may also be sensitive (e.g for medical, commercial or political reasons) and cannot be outsourced for analysis. Furthermore, an application may rely on proprietary licensed software that cannot be migrated to the public cloud easily.

- *The end-user does not want to move to the public cloud:* the migration process may prove costly as well as the cost of application execution over a long period of time. End-users may also require per-month predictable outgoings; a feature that does not currently exist in the public cloud model. By migrating to the public cloud, end-users may also feel they will lose the required control of their data. Furthermore, the issues surrounding public cloud security may deter some end-users from adopting this model.

- *The application is not suited to a public cloud model:* applications that do not have strong performance guarantees or those where execution costs outweigh the value of the actual results, are typically not suitable for the public cloud [80]. The application may also be under development and therefore the number of failed executions may prove costly.

These problems can be alleviated by the procurement of an in-house private dedicated cloud; data will remain local and the unpredictable costs of deploying applications that are not suited to the commercial cloud is avoided. Similarly, an end-user need not worry about variable or unknown demand if they have a suitably sized dedicated infrastructure where per-month costs can be calculated relatively easily. Despite this, a long tail of businesses, research institutions and personal infrastructure users exist that are unable to utilize the private cloud model:

- *The end-user is unable to deploy a private cloud:* this may be due to the lack of required dedicated infrastructure to install a private cloud or limited financial backing to procure and support an internal dedicated private cloud. Such end-users may include internal departments of large world-wide organizations, smaller businesses or universities that have limited budgets, or even users of smaller personal infrastructures.

  It has been shown that approximately 45%, 25% 15% and 15% of the costs of procuring a data centre is attributed to buying servers, power distribution and cooling, electricity and network equipment, respectively [113]. Staff costs to procure, manage and run the infrastructure will also be significant. These costs will of course reduce when one buys a smaller private infrastructure. However the additional costs of buying servers, cooling and equipment as well as continued support may not be financially viable or simply deter an end-user from procuring a private infrastructure.

An end-user who has computational tasks and cannot or will not adopt either cloud model can turn to cluster or Grid computing. However, similar to the potential hindrances of end-user uptake with the aforementioned cloud models, an end-user may not be able to outsource to the Grid or remote cluster or be able to procure and manage an internal cluster. As a consequence, *end-users that are unable to outsource computation or deploy a dedicated platform locally* are left with very few options of how to execute their computational tasks.

1.2 *Ad hoc* Cloud Computing

To offer an attractive alternative, an *ad hoc* cloud computing framework is developed to transform spare resource capacity from an end-user's locally available, non-dedicated and non-exclusive infrastructure into an overlay cloud platform. We define a non-dedicated infrastructure as one where the hosts providing the cloud service are sporadically available and unreliable in nature. Furthermore, we define a non-exclusive infrastructure as one whose hosts are reserved for some other primary purpose, e.g employee workstations running company applications. Examples of such end-users may range from personal infrastructure users with underutilized computers, to startup companies through to large-scale infrastructures. The core of this concept deploys an *ad hoc* cloud on top of the end-user's existing non-dedicated and non-exclusive infrastructure to harvest resources from *ad hoc hosts*; hosts of the infrastructure assigned to the *ad hoc* cloud. These resources are then exposed to other potential end-users within a particular domain as a cloud platform. Figure 1.1 shows a high-level overview of this new cloud computing model.

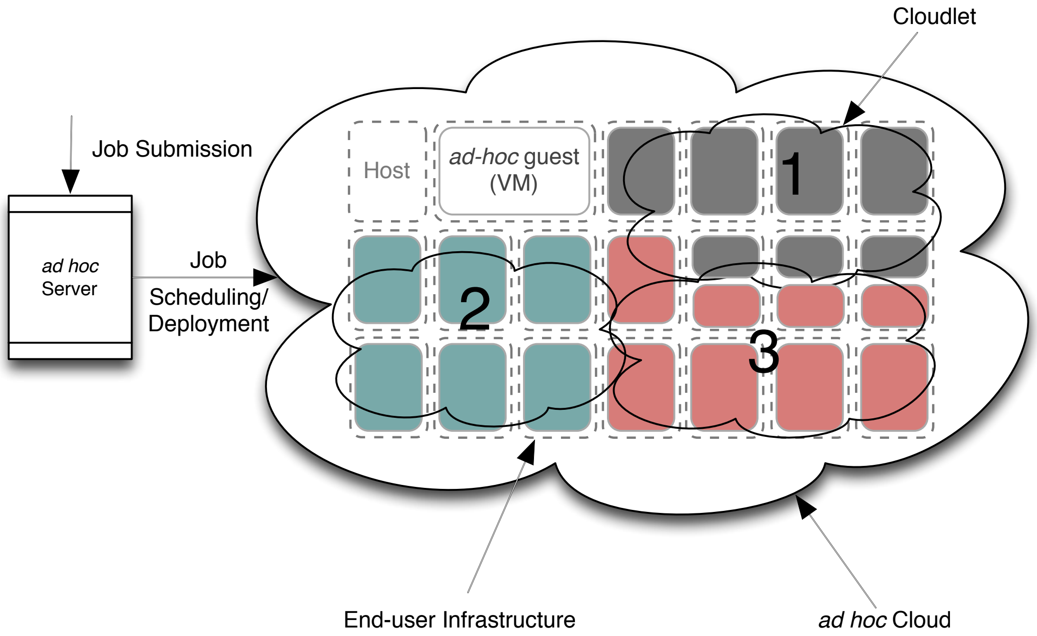


Figure 1.1: *Ad hoc* Cloud Architecture

The end-user's infrastructure, which may be large or small, consists of a number of *ad hoc* hosts that are primarily used for some other tasks; we define these tasks as *host processes*. Each host's spare resources are harvested and are utilized by a series of *ad hoc guests* (guests); virtual machines that execute *cloud jobs* while offering protection for host and guest processes.

Cloud jobs are submitted to the *ad hoc* cloud platform by cloud users and their jobs are then forwarded to the appropriate *cloudlet* for execution; a cloudlet is a set of connected *ad hoc guests* that provide a particular service or execution environment. We see from Figure 1.1 that many cloudlets may exist and can be deployed over multiple *ad hoc* hosts. In the diagram above, cloudlet '1' and '2' may hold the necessary environments for Matlab and BLAST applications to execute respectively. Note that cloudlet members are likely to be distributed across an infrastructure and not co-located as depicted above. However, a host may either be dedicated to a single cloudlet or attached to many to allow applications from different domains to run concurrently upon the same host.

1.2.1 What is Different About *Ad hoc* Cloud Computing?

The *ad hoc* cloud computing model is similar in many ways to cloud, cluster or Grid computing. Likewise, the *ad hoc* cloud is also similar to the volunteer computing paradigms employed by the Berkeley Open Infrastructure for Network Computing (BOINC) [7] and Condor [204] where host resources are harvested and are made available to volunteer computational tasks. Regardless of the computational model chosen, each share common challenges that must be overcome to provide a fully functioning environment to the end-user; the ability to effectively monitor, manage and test the infrastructure are three of the most important.

In order to successfully develop an *ad hoc* cloud, various features must be taken from cloud computing, virtualization and volunteer computing, monitoring, management and testing; we define these as the six founding principles of *ad hoc* cloud computing. Despite being similar to cloud, cluster, volunteer and Grid computing, the *ad hoc* cloud computing paradigm has many key differences. The *ad hoc* cloud model:

- operates over a set of non-exclusive and sporadically available hosts, which may be unpredictable in nature. This is in contrast to offering a service from a dedicated cloud, cluster or Grid infrastructure where each host's resources are fully committed to the service.
- does not assume a level of trust exists between an end-user and the infrastructure provider; a relationship that currently exists between end-users, clouds, clusters and Grids.
- maintains service availability in the presence of host or guest membership churn or failure to ensure job continuity over a set of unreliable hosts.
- does not interfere with executing host processes, especially in cases where these important processes dynamically consume a varying amount of resources at any given time.
- targets a set of more diverse applications such as memory, I/O and disk-intensive tasks as opposed to typical CPU-intensive applications commonly executed by volunteer computing frameworks.

Due to the complexities of operating a cloud platform on a non-exclusive and potentially highly unreliable infrastructure, there are also other subsequent challenges that must be addressed. The *ad hoc* cloud must:

- quickly and accurately determine the presence and status of hosts and guests. This firstly ensures that those available are assigned cloud jobs and secondly, hosts and guests that possess a cloud job but later become non-operational, are detected promptly and the guest that executes the cloud job is migrated and executed elsewhere.
- dynamically cope with a system where the total computational and storage potential of the entire platform changes frequently.
- schedule cloud jobs to near-optimal hosts and guests based on host availability, specifications, reliability and load in order to maximize a job's performance and probability of successfully completing.
- dynamically monitor sporadically available guests and cloudlets to ensure that the system scheduler has the most accurate state information of each host, cloudlet and the entire system to make appropriate scheduling decisions.
- dynamically control sporadically available guests to ensure the infrastructure administrator or end-user has the ability to effectively control the operation of the *ad hoc* cloud platform, dependent on their requirements.
- be simple to download, deploy and utilize. Typically clusters, Grids and clouds require technically minded individuals to make use of the infrastructure however not all cloud users within a particular domain may be highly technical system administrators (e.g biology researchers).

Our research has developed solutions to each of the research challenges above and to our knowledge, no other research in this field has been undertaken on such a large scale to offer these solutions. Therefore, end-users, who were unable to outsource computation or deploy a dedicated cloud or cluster locally are now able to employ *ad hoc* cloud computing to execute their computational tasks. This in turn may act as an intellectual ramp for those who wish to use either the public or private cloud models at a later stage. Furthermore, this alternative platform may also be complimentary to existing pubic and private cloud models.

1.2.2 Is There A Market for *Ad hoc* Cloud Computing?

Nowadays, businesses are able to gain a competitive advantage by improving their analysis of data, running complex models and improving future planning, for example.

In order to achieve this, businesses are increasingly moving towards IT and therefore the proportion of computers available is also growing. This increase of available infrastructure is leading to an increase in resource underutilzation, in turn strengthening the case for the deployment of an *ad hoc* cloud platform.

To determine the potential target market and uptake of an *ad hoc* cloud computing platform, we analyzed UK government statistics to select a subset of UK-only businesses that may suit this paradigm. In order to adopt *ad hoc* cloud computing and gain from the benefits it offers, it is reasonable to assume that a business's current infrastructure size is large enough to allow cloud jobs to continue to execute in the face of failures. Therefore, we assume that the minimum infrastructure size consists of approximately ten hosts, however the maximum size can potentially be limitless. With this in mind and only considering UK businesses with more than ten employees (assuming each has their own host), we approximately calculate the number of businesses that could potentially adopt the *ad hoc* cloud computing paradigm.

We analysed UK government statistics outlining the number and types of private sector businesses in the UK. In the UK alone, approximately 4.8 million private sector businesses exist where 4.5% (214,155) of these have at least 10 employees [104]. However as many businesses will either not be able to employ *ad hoc* cloud computing or simply would not find it useful, the figure can be reduced. The potential business areas that may find *ad hoc* cloud computing useful based on the Standard Industrial Classification (SIC) codes [105] are highlighted in Figure 1.2.

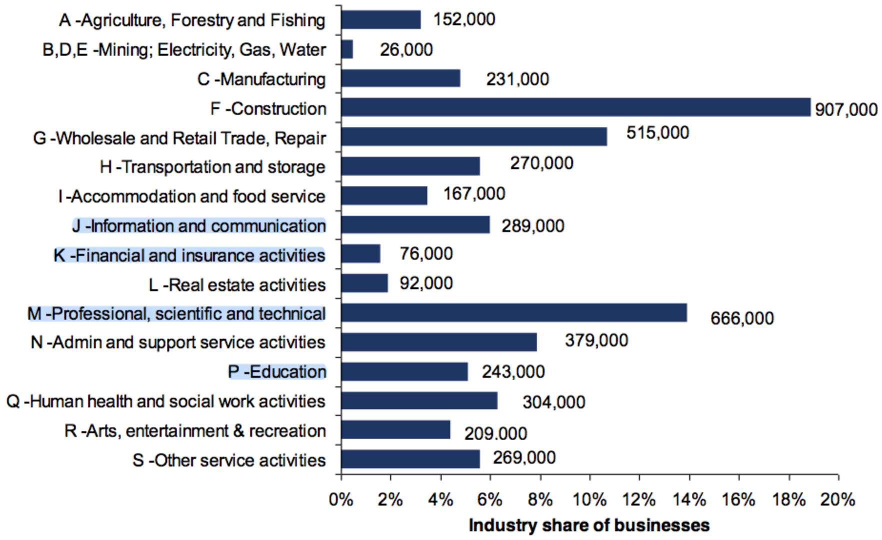


Figure 1.2: UK Private Business Categories [104]

The potential total of UK businesses that may find *ad hoc* cloud computing useful is 1,274,000. However, by omitting those with fewer than 10 employees and assuming that only 4.5% of these businesses could employ *ad hoc* cloud computing, approximately 57,330 UK organizations could potentially employ *ad hoc* cloud computing. At first glance, this figure may seen low, however the UK business statistics study of 2012 [104] does omit key factors. These being non-profit and public organizations (i.e research institutions, universities etc) and organizations that are not registered in the UK but have operations located in the UK.

Many businesses in Europe and indeed the rest of the World could employ *ad hoc* cloud computing who have the difficulties aforementioned. Furthermore, by including research institutions and the 'personal use' market, the possible user-base will be increase substantially. Together with the increase of available infrastructure, as many businesses and research communities increasingly move towards IT, the proportion that could benefit and employ *ad hoc* cloud computing is growing. Hence, a large market does exist for *ad hoc* cloud computing and this market is likely to be many orders of magnitude greater than the figure quoted above.

Therefore, by introducing a cloud platform that is able to take advantage of an existing infrastructure, while offering high levels of performance and reliability, we would expect the level of uptake from the outlined user-base to be high, especially in markets that are driven by decreasing operating costs.

1.3 Research Statement

Based on the need and potential uptake of the *ad hoc* cloud computing paradigm, the objective of this research is to *determine the feasibility, reliability and performance of ad hoc cloud computing infrastructures.* We hypothesise that:

- operating this new cloud concept over unreliable infrastructures is a reliable method to execute cloud applications.
- the performance of the *ad hoc* cloud platform proposed can be at times, comparable to dedicated cloud models, and at others, offers acceptable performance for end-users to effectively run computations.
- the *ad hoc* cloud is a feasible and viable alternative computational platform to commercial or private clouds as well as clusters and Grid infrastructures.

1.4 Terminology

For convenience, a summary of key terms used in this thesis are shown in Table 1.1.

Term	Definition
end-user	a potential user of a computational platform. For example, a cluster, Grid, cloud, volunteer infrastructure, *ad hoc* cloud, etc.
volunteer user	a user of a volunteer computing infrastructure.
volunteer host	a host whose resources are donated to a volunteer computing infrastructure under the instruction of a *volunteer user*.
volunteer application developer	a developer or researcher who implements the volunteer scientific application to be executed on *volunteer hosts*.
BOINC server administrator	an individual who manages a BOINC, V-BOINC or *ad hoc* server.
cloud user	a user of a cloud infrastructure. For example, the *ad hoc* cloud, Amazon EC2, Microsoft Azure and Google AppEngine.
cloud provider	the provider of a cloud service that a *cloud user* may utilize.
ad hoc host	a physical machine whose resources are donated to the *ad hoc* cloud but is used for some other primary purpose.
ad hoc guest	a virtual machine that executes on the *ad hoc* host.
ad hoc host user	a user of the *ad hoc* host, e.g. a company employee.
ad hoc host owner	the owner of the *ad hoc* host. This may be the *ad hoc* host user or another person or entity, for example, a company or research institution.

Table 1.1: Terminology

1.5 Assumptions

For an initial *ad hoc* cloud computing platform to operate successfully, we make a few assumptions relating to the end-user's infrastructure and the applications that can be deployed using our *ad hoc* cloud computing platform.

Firstly, we define 'cloud computing' as various definitions of the term currently exist. We use the definition from NIST which specifies that *"cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [176]. We define 'on-demand' access to an *ad hoc* cloud when the time to acquire resources is similar to or less than the latency to provision a virtual machine on a commercial cloud infrastructure. For example, an Amazon Elastic Compute Cloud instance takes on average, 90 seconds to become available; we report the provision latency of *ad hoc* resources in Chapter 6. We further assume that a cloud job has to be encapsulated in a virtual machine for its management and protection as well as the protection of host processes.

Secondly, we assume that hosts have enough spare resource capacity, e.g CPU cycles, storage, etc to offer to the *ad hoc* cloud computing platform. In cases where the capacity available is sufficiently low and jobs cannot run effectively or at all, we assume that it is possible for an end-user's infrastructure to temporarily outburst to the commercial cloud (or any other available cloud platform); this can be achieved by integrating both the *ad hoc* and remote cloud via the remote cloud provider's APIs. Conversely, applications developed to run on public cloud platforms can be imported with relative ease to the *ad hoc* infrastructure. The method of outbursting is out of the scope of this thesis, however we instead direct you to the following related research for more information [183, 75, 152, 94, 163].

For the sake of protection and easy management of the *ad hoc* cloud platform, we restrict our implementation to private networks; we assume the *ad hoc* cloud paradigm to be predominately deployed on Local Area Networks (LANs). Hence we can be assured that the network and member machines are relatively secure and the network is reasonably fast to operate our reliability mechanisms on. In cases where the private network operates over a Wide Area Network (WAN), we assume that the relevant security protocols are in place to provide adequate protection for both host and guest machines within the infrastructure.

Finally, we also note that our first prototype of an *ad hoc* cloud specifically only deals with the following tested applications: CPU, memory, I/O and disk-intensive applications. We outline in Chapters 6 and 7 which of these applications are suitable for execution on an *ad hoc* cloud. Applications that write to external dependencies (i.e. those located on other virtual machines or physical hosts) may or may not function as expected due to data read and write inconsistencies when a host or guest fails abruptly. Such applications may need further reliability mechanisms incorporated in their implementation to cope with frequent unexpected failures. Furthermore, applications that require high levels of security and isolation from other processes may not be suited to the *ad hoc* cloud.

1.6 Thesis Contributions

The main contributions of this thesis are summarized as follows:

1.6.1 Feasibility

1. The development of a model to run virtual machines over volunteer infrastructures through the virtualization of the Berkeley Open Infrastructure for Network Computing (BOINC), called V-BOINC.

2. The development of a method allowing BOINC project developers and researchers to execute applications with dependencies on volunteer infrastructures through the use of V-BOINC.

3. The development of *ad hoc* client and *ad hoc* server components based on the extension of V-BOINC's equivalents.

4. The development of a BOINC job submission system.

5. The development of a scheduling algorithm to select the near-optimal *ad hoc* host for cloud processes to execute upon as well as select the near-optimal host to relocate a failed cloud job and *ad hoc* guest.

6. The development of an *ad hoc* reliability algorithm ensuring the continuity of cloud jobs.

7. The development of an easy to use platform for use by computer literate but non-system administrator personnel.

1.6.2 Management and Monitoring

8. The development of a dynamic infrastructure monitoring and management tool called the Cloudlet Control and Monitoring System (C2MS).

1.6.3 Cost, Performance and Reliability

9. A method of evaluation for *ad hoc* cloud computing infrastructures.
10. An analysis of the performance and reliability of an *ad hoc* cloud computing platform.
11. A performance analysis between *ad hoc* cloud infrastructures and commercial cloud platforms, such as Amazon EC2.
12. An in-depth cost and performance analysis of the popular commercial cloud provider; Amazon Elastic Compute Cloud (EC2).

1.7 Thesis Structure

The rest of the thesis is structured as follows:

Background: Chapter 2 gives a background of the six founding principles of *ad hoc* cloud computing: virtualization, cloud computing, volunteer computing, infrastructure monitoring, management and testing. Chapter 2 first outlines various virtualization technologies and their evaluated performance followed by a discussion of the different service and deployment models of cloud computing. A detailed overview of Amazon EC2 is also given. Chapter 2 then details volunteer computing, and in particular the volunteer infrastructure BOINC. Finally, the infrastructure monitoring and management tools used later in the thesis as well as the applications used to evaluate our *ad hoc* cloud are described.

V-BOINC: The Virtualization of BOINC: Chapter 3 details how virtualization and volunteer computing are integrated to create a virtualized volunteer infrastructure, which we call V-BOINC, to provide a firm basis for the development of an *ad hoc* cloud platform. Chapter 3 also describes how V-BOINC is able to execute a vast range of applications rather than CPU applications volunteer infrastructures typically execute. The penultimate section of Chapter 3 outlines the performance measurements and overheads of V-BOINC by executing selected benchmarks upon the system. Finally, Chapter 3 discusses what effects virtual machine checkpointing has on the limited storage available on volunteer hosts.

From Volunteer to *Ad hoc* Cloud Computing: Chapter 4 describes the major steps to transform V-BOINC into an *ad hoc* cloud computing platform. Firstly, Chapter 4 outlines related work and secondly, details the *ad hoc* cloud platform architecture and components as well as the interactions between them. Chapter 4 then describes our contributions from BOINC job submission to scheduling cloud jobs and the restoration of *ad hoc* guests on near-optimal *ad hoc* hosts. Chapter 4 also describes one of our primary contributions of how to make an unreliable infrastructure reliable. The penultimate section of Chapter 4 details possible methods to minimize the host process interference caused by cloud processes and finally, the installation and use of our *ad hoc* cloud prototype is discussed.

Monitoring and Controlling Dynamic *ad hoc* Infrastructures: Chapter 5 focusses on the final components required in any *ad hoc* cloud platform; the ability to monitor and manage the dynamic infrastructure. Chapter 5 describes how our monitoring tool, the Cloudlet Control and Monitoring System (C2MS), uses and extends Ganglia to monitor cloudlets of machines whose members dynamically enter and leave frequently. Similarly, Chapter 5 outlines how an infrastructure control component can be integrated with the extended version of Ganglia. Finally, Chapter 5 outlines the performance overheads of the C2MS and how quickly a dynamic infrastructure can be controlled.

Evaluating the *ad hoc* Cloud: Chapter 6 outlines our evaluation of the *ad hoc* cloud. Firstly, Chapter 6 explains the criteria to be measured when evaluating any *ad hoc* cloud and secondly the experimental setup is discussed. Thirdly, the reliability of our *ad hoc* cloud is evaluated when operating over a simulated unreliable infrastructure. Fourthly, the overall performance and overheads associated with the *ad hoc* cloud are evaluated. The latter includes a cloud job's pre- and post-execution overheads, checkpointing overheads and *ad hoc* guest restoration overheads. Chapter 6 also evaluates network performance as well as the performance of our *ad hoc* server. A comparison between the *ad hoc* cloud platform performance and Amazon EC2 is also given.

Conclusions: Chapter 7 gives a summary of the thesis and highlights future work to be undertaken to develop the *ad hoc* cloud computing paradigm further. Finally Chapter 7 concludes by determining whether our research hypothesis was proved correct.

Cloudy Waters: Tapping into the Unknown: Appendix A discusses the background knowledge of cloud computing we acquired before setting out to develop an *ad hoc* cloud platform. While this background is necessary for those researching and developing cloud platforms, it is not fundamental to understand the concepts, development and evaluation of the *ad hoc* cloud. Appendix A begins by describing how a commercial cloud's performance can vary significantly dependent on time of day, the given load of a cloud instance and which type of processor an instance is deployed upon. A method of underutilizing instance processors to increase performance and reduce variabilities is also described. The cost implications of these variabilities are outlined as well as the cost differences seen by an end-user dependent on where in the world a cloud job is submitted from. Appendix A also explores the idea of employing a cost model in *ad hoc* cloud computing infrastructures.

Chapter 2

Background

2.1 Introduction

In this chapter, we provide a background of the literature required to understand the six founding principles of *ad hoc* cloud computing: virtualization, cloud computing, volunteer computing, monitoring, management and testing. This chapter may be skipped at the first reading if each of the founding principles are well known to the reader.

We first give an overview of virtualization and then outline the various technologies and their respective benefits and drawbacks to help determine the most suitable for use in an *ad hoc* cloud computing infrastructure. As virtualization is commonly seen as a major enabler of cloud computing, we are then able to provide an in-depth review of cloud computing and the various service and deployment models clouds typically conform to. We then describe a cloud platform provider in detail for reference later in the thesis.

We then discuss the alternative computing model offered by volunteer computing, the current state of research in that area and give a brief overview of Grid computing as there are many similarities shared between the two models. We then give an in-depth description of a volunteer system that we use within the *ad hoc* cloud.

This is followed by an overview of the importance of infrastructure monitoring and an outline of current infrastructure monitoring and management tools that are used later in the thesis to later show that a functionality gap exists between these tools and those that must monitor and manage *ad hoc* infrastructures. Finally, we describe the applications and benchmarks used during the development and evaluation of our *ad hoc* cloud platform.

2.2 Virtualization

Virtualization is the process of creating an abstract version of a resource or entity, whether it may be a single hardware component or an entire system [218]. Virtualization was first invented by IBM in the 1960s [198] and is used to provide features such as increased utilization, economies of scale, easy management, scalability, agility, reliability and security. These benefits have become an integral part of the cloud computing model where virtualization is commonly seen as its major enabler. We therefore discuss how virtualization is achieved and the types of virtualization that exist. We also outline the major virtualization technologies that are available as well as their merits and downfalls.

2.2.1 Overview

By virtualizing a resource or entire system, multiple and distinct versions of a single physical resource are created, giving the impression that a user has full and exclusive access to the resource. This allows users of virtualized infrastructures to run multiple operating systems on the same piece of hardware. Infrastructure administrators are then able to exploit a one-to-many relationship between hardware and end-user respectively in order to obtain the economic benefits of large-scale resource sharing [212]. This is in contrast to the typical cluster model where one-to-one mappings are commonly exercised. The ability to concurrently share a single piece of hardware is made possible by a hypervisor, or virtual machine monitor, as shown in Figure 2.1.

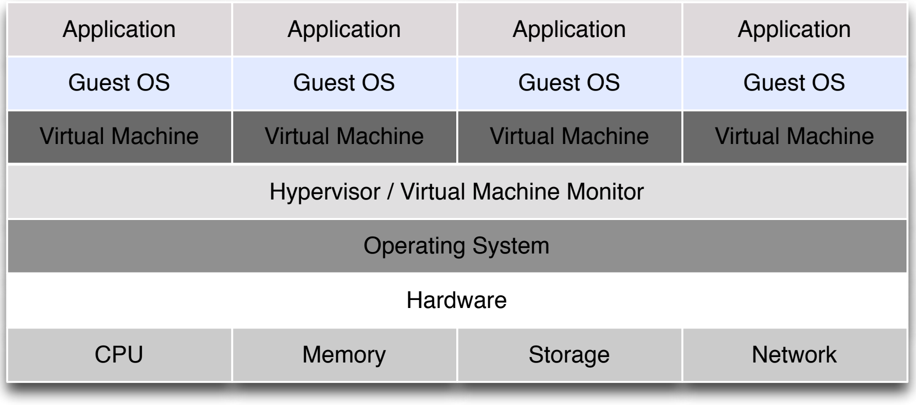


Figure 2.1: Hypervisor: Partitioning Physical and Virtual Resources

The hypervisor resides between the host operating system and guest virtual machines and performs a specific job dependent on the type of virtualization employed: *full virtualization*, *paravirtualization*, or *hardware-assisted virtualization*.

Full virtualization is a virtualization technique that provides a complete virtual environment to simulate the underlying hardware allowing a guest operating system to run on the host without any modifications [207]. Non-sensitive guest code, such as instructions that do not control hardware, typically run directly on the host otherwise the instructions are trapped by the hypervisor, translated by the software and sent to the hardware; requests return via the same route. Due to instruction emulation, the performance of full virtualization technologies can be greatly affected [189], however it is particularly useful for security and isolating users from one another. Note that full virtualization is different from *emulation* where every machine instruction is translated [190].

Paravirtualization is a virtualization technique that provides a software interface used by the guest operating system to execute sensitive instructions [189]. This requires a modification to the guest operating system kernel, where calls to these sensitive instructions are replaced with calls to the hypervisor. Paravirtualization is designed to lower virtualization overhead and increase performance, however operating systems that cannot be modified (e.g Windows XP) are unsupported.

Hardware-assisted virtualization is a virtualization technique that takes advantage specialized instruction sets VT-x and AMD-V on x86 architectures [207]. Guest operating systems do not need to be modified as sensitive instruction calls are trapped by the hypervisor running in privileged mode and can execute the instruction calls directly upon the hardware via VT-x or AMD-V [209]. Hardware-assisted virtualization can offer substantial performance gains, however it is only available on architectures that have the VT-x or AMD-V instructions sets available.

2.2.2 Technologies

Many different virtualization technologies exist, each offering various levels of abstraction and features. We now outline a select number of popular virtualization technologies that have the potential to be used in an *ad hoc* cloud computing platform.

- *QEMU:* is an open source virtual machine emulator that operates on x86, x86-64 and PowerPC architectures and is able to emulate x86, x86-64, ARM, SPARC, PowerPC and MIPS systems [62]. In order to increase the reasonable performance of QEMU, the Kernel Virtual Machine (KVM) component can be used. KVM is a special operating mode of QEMU and takes advantage of hardware-assisted virtualization via the extensions Intel VT-x or AMD-V [103] found on recent Linux kernels.

- *VirtualBox:* is a x86 and AMD64/Intel64 open source virtualization product developed and maintained by Oracle [40]. VirtualBox can be run on all major platforms and supports many guest operating systems. VirtualBox does however have components based on QEMU [160] but offers full virtualization rather than complete emulation. Furthermore, VirtualBox supports hardware-assisted virtualization via Intel's VT-x and AMD's AMD-V.

- *VMware Player:* is a free virtualization suite developed and maintained by the software company VMware [41]. VMware Player is not however open source like the aforementioned virtualization packages. Like VirtualBox, it offers a full virtualization suite for deploying virtual machines but also supports hardware-assisted virtualization.

- *Xen:* is an open source x86 hypervisor allowing multiple virtual machine occupancy that can offer near-native performance [60]. Unlike other virtualization suites where the package is installed upon the host operating system, Xen is implemented as a native hypervisor. Therefore, Xen is installed on the bare-metal and takes control of the physical machine. Xen is an example of paravirtualization but also supports full and hardware-assisted virtualization [223]; the latter is known as a Hardware Virtual Machine (HVM) by Xen.

2.2.3 A Performance Comparison

There have been many studies investigating the affects and overheads of virtualization on both host and application performance, as well as the differences in performance between virtualization technologies. Dominques *et al.* investigate the performance overheads of VMware Player, QEMU and VirtualBox [91]. The authors execute CPU, I/O and network benchmarks on the respective virtual guests and compare the results by executing the same benchmarks on the native host.

The network benchmark that measures the network speed showed that VMware Player gave the greatest network performance when using bridged networking, i.e. the virtual machine is connected to the network using the host's Ethernet adapter. This is followed by QEMU and then VMware Player and VirtualBox when both are configured using Network Address Translation (NAT). The CPU benchmark, which measures floating and non-floating point performance, and the I/O benchmark measuring the read and write performance to and from disk, both showed that QEMU performs slowest with VMware Player performing the best. Other research has found QEMU offers reasonable performance [62]. With the exception of QEMU, Dominques *et al.* quote that between 15% and 35% of an overhead exists when running CPU-bound benchmarks. Other studies quote an overhead of less than 15% [60, 110, 224].

Dominques *et al.* also note an interesting result where the performance of a virtual machine is correlated to the impact on the host OS. Their findings show that as a virtual machine performs better, a host's application performance drops. By fully consuming the virtual CPU and individually running a CPU and memory single-threaded benchmark on the host, little performance overheads were measured for all virtualization technologies. For multi-threaded applications running on the dual-core machine (180% CPU availability including OS overhead), VirtualBox performs best by being able to access approximately 168% of the CPU capacity followed by QEMU and VMware Player at 160% and 120% respectively.

Younge *et al.* investigate the performance of the hypervisors VirtualBox, Xen and KVM in the context of High Performance Computing (HPC) environments. The authors use FutureGrid as their testbed; a geographically distributed set of heterogeneous hosts [18], however they only use a set of four nodes from one location. The benchmark suites HPCC [23] and SPEC [35] were used that contain various benchmarks testing CPU, memory, I/O and disk performance; these may be single or multi-threaded.

Executing the HPCC Linpack benchmark, which measures the floating point rates of execution when solving linear equations, showed that KVM and VirtualBox achieve 51.8 and 51.3 Gflops respectively when compared to a native execution of 73.5 Gflops. Xen managed to achieve 49.1 Gflops on average, however it experienced a high degree of variability. Secondly, a similar benchmark measuring floating point rates of execution when solving complex one-dimensional Fourier Transforms was then used. Without detailing specifics, the results show that all virtualization technologies can in some cases achieve near-native performance. In other cases, KVM and VirtualBox perform well overall however Xen experiences high variability and under-performs.

The authors finally evaluate the technologies using the SPEC OpenMP benchmark that executes shared-memory parallel tests. The results show that KVM almost offers near-native performance with a SPEC score 0.3% lower than the native execution. The performance of Xen and VirtualBox are 0.9% and 0.91% lower respectively.

In summary, we have introduced the background necessary to understand virtualization and analysed previous research outlining the performance differences of currently popular virtualization technologies. Table 2.1 summarizes the performance of the virtualization technologies from the studies of Dominques *et al. (Dom)* and Younge *et al. (Youn)* ranked in order from best performing to least.

Benchmark	**CPU**		**Memory**		**Network**		**IO**	
Author	*Dom.*	*Youn.*	*Dom*	*Youn.*	*Dom*	*Youn.*	*Dom*	*Youn.*
Rank 1	VMw	KVM	NA	KVM	VMw (B)	VBox	VMw	NA
Rank 2	VBox	VBox	NA	XEN	QEMU	Xen	VBox	NA
Rank 3	QEMU	Xen	NA	VBox	VMw (N)	KVM	QEMU	NA
Rank 4	NA	NA	NA	NA	VBox	NA	NA	NA

Table 2.1: Summary of Virtualization Technology Performance

Table 2.1 shows that KVM, VMware Player (VMw: (N)AT, (B)ridged) and VirtualBox (VBox) all perform well, although this is dependent on the host's hardware specifications and the executing application. The performance of Xen has been shown to be lower than the other virtualization technologies outlined, however others find that it offers high performance and also out-performs VMware [60]. Even though the virtualization technologies are ranked, there are cases where the difference between ranks is minimal. We offer our own comparison of virtualization technologies in Chapter 3.

2.3 Cloud Computing

In this section, we outline the various definitions related to cloud computing and then provide a brief overview of cloud computing describing the most important features of the cloud. We then provide an in-depth overview of Amazon EC2 which is required to understand a substantial portion of literature presented later in the thesis. We finally discuss the cloud platform OpenStack and specifically its scheduling component which is modified and used within our *ad hoc* cloud. This also aims to give an alternative view of the available cloud platforms.

2.3.1 Definitions

Ever since the introduction of cloud computing, a single precise definition of the term has not been coined although most definitions reference the delivery of services, applications and resources from a set of remote host servers that are accessed over the commodity Internet [56, 57, 127, 64, 156]. There are however widely agreed definitions that describe the various deployment and service models and we outline those here. Cloud computing platforms exist in many forms and typically conform to common deployment and service models. A cloud may either be deployed as a public, private or hybrid cloud, as shown in Figure 2.2.

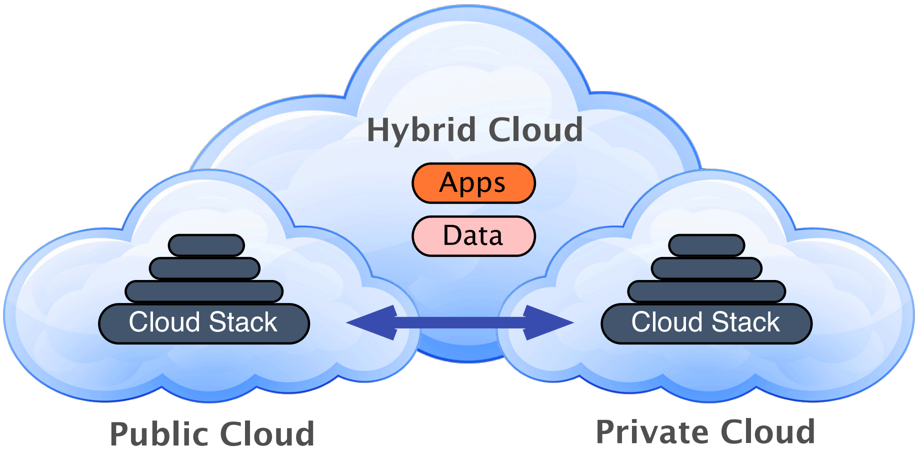


Figure 2.2: Cloud Computing Deployment Models; derived from [42]

Public cloud computing models are made available to the general public and resource use is charged like the pay-as-you-go model for charging electricity. Private cloud computing infrastructures typically reside behind organizational firewalls and are used in accordance to the goals they aim to achieve. Hybrid clouds are private clouds that are extended by *cloudbursting* [55] to the public cloud in order to cope with additional demand or to take advantage of additional services.

Cloud providers also conform to either a single or number of common service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), each of which provide differing levels of environment abstraction [93]. Figure 2.3 depicts these different service models and the level of abstraction offered to the end-user.

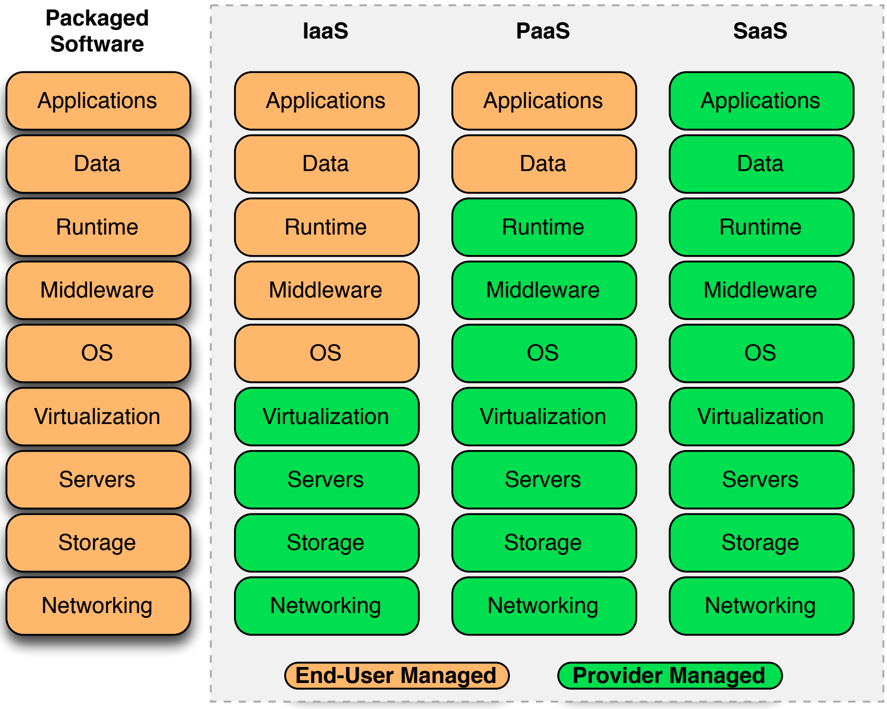


Figure 2.3: Cloud Computing Service Models; derived from [13]

IaaS is a popular service model offered by cloud providers where raw compute, storage and network capabilities are delivered as a service to the end-user. End-users are only able to customize the infrastructure from the Operating System (OS) level upwards to the application. This is in contrast to owning a dedicated cluster, where an administrator would control raw compute, storage and network resources. Within the IaaS model, these resources are available to cloud users via virtualization giving the impression that a single cloud user has full and exclusive access to the resource. Furthermore, virtualization reduces the barrier to entry for many as end-users do not have to make substantial changes to their applications to deploy them on a virtual machine, or instance. A common example of an IaaS cloud infrastructure is Amazon Elastic Compute Cloud (EC2).

PaaS on the other hand delivers an application environment to the end-user who only needs to be concerned with their application and data. The cloud provider manages the underlying infrastructure and can offer various application stacks such as Java and Python for example. This model is particularly useful for those who wish to avoid the complexities of system administration and are only concerned by running applications. Common examples of PaaS cloud infrastructures are Google App Engine (GAE) and Microsoft Azure.

Finally, at a higher layer of abstraction, *SaaS* is a method of delivering applications over the Internet. Like PaaS, the cloud provider manages the underlying infrastructure but applications and associated data are created, managed and delivered by the cloud provider. Common examples of SaaS cloud applications are Google Mail and Microsoft Office 365.

Like *Saas*, both *IaaS* and *PaaS* cloud providers also offer products and services alongside their infrastructure offering. This may be scalable storage (e.g. Amazon S3 and GAE Datastore), relational databases (e.g. Amazon Relational Database Service and SQL Azure), monitoring tools (e.g. Amazon CloudWatch) or even authentication software (Amazon Web Services Identity and Access Management and Azure Active Directory). These services are easily integrated into a cloud user's virtual infrastructure either via web interfaces or simple APIs.

2.3.2 Overview

Cloud computing has become a disruptive yet popular technology due to the advantages it offers both cloud provider and cloud user. By employing virtualization, warehouse scale cluster management reduces the management overhead per customer and the size of the user-base that can be served is much larger when compared to non-virtualized infrastructures. As a result, server utilization has increased, while the overall energy consumed [150] has decreased, in turn promoting greener IT. Increased server utilization often leads to reduced operating expenses and these savings can either be passed to the customers of public cloud infrastructures or manifested in profit.

Cost is one major factor as to why public cloud computing has also become popular in recent years. Cloud computing's 'pay-as-you-go' charging model, where resources are paid for per unit consumed per hour, can often be claimed to be a cheaper model than procuring a dedicated infrastructure [136, 184, 143]. Such claims are typically based on individual cases and should not be generalized.

Furthermore, estimating the true cost of a dedicated infrastructure is not well understood, even by those who currently administer such systems [202]. Additionally, cloud computing is often cited as being more expensive for long term use [202]. However, these additional costs may be offset by taking advantage of recent research to decrease costs; for example, by dynamically auto-scaling virtual instances based on workload information and performance targets [158]. Alternatively, expensive long-term costs may be out-weighted by the other advantages cloud computing offers [56, 57].

One advantage is the scalable and elastic nature of the cloud; the ability to match a workload's resource demand in an autonomic fashion by dynamically adding or removing resources. Elasticity is particularly useful for applications that have sporadic and unpredictable demand. A good example of this is the Animoto use case [5]. Animoto is a software company that creates video slideshows from uploaded photos, music and video clips. The company experienced substantial growth when their application gained popularity and therefore scaled from 40 to 5000 EC2 instances in three days.

Unfortunately, elasticity is not a feature commonly employed by other infrastructures, such a clusters and Grids. Cluster and Grid users must either perform as much work as they can with the available resources or system administrators must procure enough hardware to cope with peak levels of usage if they strive to satisfy demand.

Furthermore, unlike cluster and Grid infrastructures, the cloud is perceived to be a flexible computational model where an apparently 'unlimited' number of resources can be deployed and utilized on-demand from any host with an Internet connection. The cloud also offers high levels of availability. For example, Amazon EC2 commits to provide 99.95% availability as outlined in their Service Level Agreement (SLA) [3]; a contract between cloud customer and provider outlining the standards of service the provider has guaranteed to deliver.

Although levels of performance are not specified within SLAs, there exists a strong notion of trust between cloud user and provider on what levels of performance should be delivered. However it has been shown that actual performance delivered can vary significantly due to resource contention from the use of virtualization and relatively poor network performance [56, 117, 127, 130, 168, 194]. We give further performance measurements of a commercial cloud platform in Appendix A.

We have just given a brief overview of cloud computing as well as a number of its benefits and drawbacks. This however is by no means a full in-depth overview of the topic and aims to serve as a background for further discussion later in the thesis. More information about cloud computing can be found at [72, 131]. We now give an overview of the commercial cloud infrastructure Amazon EC2.

2.3.3 Amazon EC2

Amazon Web Services (AWS) [4] offer their Elastic Compute Cloud (EC2) as a public Infrastructure as a Service (IaaS) cloud where virtual machine instances are provided on top of a bare-bones environment, based on Xen virtualization [212, 60]. Amazon EC2 instances come in many flavours and sizes to give a user a greater choice to correctly fit an instance type to an application's requirements. There are also a large selection of Amazon Machine Images (AMI) available to create and launch instances. An AMI stores the information needed to launch an instance, e.g. the OS, access permissions, storage volumes to attach etc. Cloud users are able to create, publish and share their own AMIs to allow others to launch equivalent instances.

2.3.3.1 Compute

At the time of first using Amazon EC2 for experimentation (approximately May 2011), we used a variety of EC2's Standard On-Demand General Purpose instances; those that charge users for compute capacity by the hour and offer no performance enhancements or extra resources, for example additional storage. Upon subsequent experimentation at a later date, we used instances of the same type and size to remain consistent throughout.

Despite this, Amazon EC2 instances have changed in configuration and cost since 2011. We show the available instance sizes, specifications and costs as of 2011 in Table 2.2. We only describe those that are used during our experiments and are hence mentioned later in the thesis. Full details of EC2 instances can be found on the Amazon EC2 website [3].

Size	vCPU	ECU	RAM (GB)	Storage (GB)	I/O	$/hr
m1.small	1	1	1.7	160	Moderate	0.085
m1.large	2	4	7.5	850	High	0.34
m1.xlarge	4	8	15	1690	High	0.68

Table 2.2: Amazon EC2 Instance Specifications and Costs (2011)

As of 2011, only three Standard On-Demand General Purpose instances sizes existed: small, large and extra large. The number of virtual cores (vCPU), Elastic Compute Units (ECU), virtual resources and costs increase proportionally according to instance size; an ECU provides the equivalent CPU performance of a 1.0-1.2 GHz 2007 AMD Opteron or Intel Xeon processor and is calculated based on Amazon's own unpublished benchmarks. EC2 processor research has shown that the small, large and extra large instances can run on Intel Xeon E5430 4 core 2.66 GHz processors or in some cases (less than 10%), small instances may run on AMD Dual-Core Opteron 2.6 GHz 2218 HE Processors [125]. Since 2011, the available instances sizes, specifications and costs have changed and these changes are reflected in Table 2.3

Size	vCPU	ECU	RAM (GB)	Storage (GB)	I/O	$/hr
m1.small	1	1	1.7	160	Low	0.044
m1.medium	1	2	3.75	410	Low	0.087
m1.large	2	4	7.5	840	Moderate	0.175
m1.xlarge	4	8	15	1680	High	0.350

Table 2.3: Amazon EC2 Instance Specifications and Costs (2014)

In 2012, the medium sized instance was introduced offering approximately twice the amount resources of a small instance for twice the cost. However, these instances also experienced a slight loss of virtual resources when compared to those offered in 2011. A large and extra large instance lost 10 GB's of storage and the level of I/O performance of a small and large instance was reduced to low and moderate respectively. This may be one of the many reasons why costs of the same sized instances are cheaper today than in 2011.

Note that the costs displayed in Table 2.2 and Table 2.3 are specific to the US East *Region* (Northern Virginia) of Amazon's cloud infrastructure. Regions are defined as separate geographical areas whose data centres are completely isolated from those in other Amazon EC2 Regions. At the time of writing, eight Amazon EC2 Regions exist and their approximate locations are depicted (in red) in Figure 2.4.

2.3.3.2 Data Transfer

Within each Amazon EC2 Region, multiple *Availability Zones* exist. Availability Zones are isolated and distinct areas that are designed to be fault tolerant from other Availability Zone failures in the same Region, i.e. if an instance fails in one Availability Zone, a service can still operate provided that it has another instance running in another Availability Zone. As a result of the segregated nature of the infrastructure, each Region has different costs for computation and storage. Transferring data between and within Regions and Availability Zones can also incur different costs due to the different data transfer types:

- Internet Data Transfer (IDT): data transferred from an instance (or any Amazon

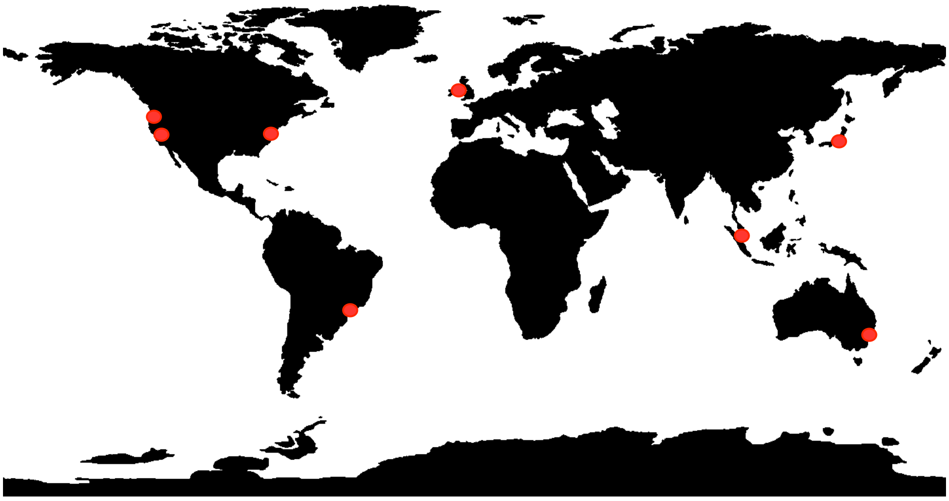


Figure 2.4: Amazon EC2 Regions

Web Service) to the Internet and vice versa. Data transferred between Regions also are classed as IDT.

- Regional Data Transfer (RDT): data transferred between instances in the same Region that are in different Availability Zones.
- Availability Zone Data Transfer (ADT): data transferred between instances in the same Availability Zone.

In order to reference experimental costs of Amazon EC2 in Appendix A, the costs of transferring data in May 2011 are outlined in Table 2.4; 2014 costs are included for completeness.

We see from Table 2.4 that the costs for RDT in both the outward and inward directions are set at $0.01 per GB, while inward IDT is now free compared to 3 years previously where this was set at $0.1 per GB. Furthermore, in 2011 the outward IDT charge was only based on the tiered charge model where users pay based on the amount they transfer. At the time of writing, outward IDT is charged at two different rates: one for transferring data to another Region, currently set at $ 0.02 per GB, and another to the Internet which is based on the cheaper tiered charging model.

2.3.3.3 Storage

Amazon also offer various persistent storage mechanisms such as Amazon Simple Storage Service (S3) and Amazon Elastic Block Storage (EBS); the former was the

Data Transfer Type	Price (May 2011)	Price (March 2014)
Data Transfer In (per GB)		
RDT	$ 0.01	$ 0.01
IDT	$ 0.10	$ 0.00
Data Transfer Out (per GB)		
RDT	$ 0.01	$ 0.01
IDT (to Region)	NA	$ 0.02
IDT (to Internet)		
First 1GB / month	$ 0.00	$ 0.00
Up to 10TB / month	$ 0.15	$ 0.12

Table 2.4: Amazon EC2 Data Transfer Costs (2011/2014)

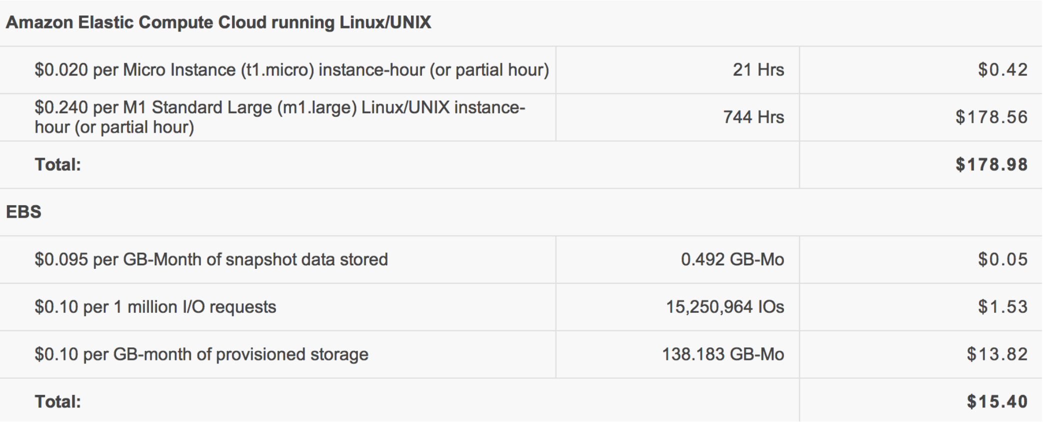
first service AWS publicly offered. Amazon S3 is an Internet-accessible, persistent and scalable storage platform providing limitless storage capabilities [172] that can be used for general persistence, backups, data distribution and sharing [4].

Amazon S3 is based on a simple architecture revolved around objects and buckets. A cloud user's data and metadata are stored as objects and these are placed in buckets. Once stored, objects are copied to multiple locations, by default, to improve data availability. Like any Amazon service, S3 is charged using the pay-as-you-go model based on the number of gigabytes stored as well as the number of requests performed on the data store.

Amazon EBS is a mountable storage, conceptually equivalent to a USB disk, that can be created and attached to a virtual machine [4]. Volumes of up to 1TB can be created and data may or may not be persistent dependent on the cloud user's choice. Amazon EBS offers data access speeds that are typically faster than S3 due to the locality of the volume mounted. Similar to S3, EBS is also charged based on the number of gigabytes stored as well as the number of requests sent to the storage service.

2.3.3.4 Billing and Usage Reports

Amazon EC2 charges for compute, data transfer and storage by monitoring the consumption rates for each of these resources and bills the cloud user at the end of each month; an example section of an EC2 bill is shown in Figure 2.5



Amazon Elastic Compute Cloud running Linux/UNIX		
$0.020 per Micro Instance (t1.micro) instance-hour (or partial hour)	21 Hrs	$0.42
$0.240 per M1 Standard Large (m1.large) Linux/UNIX instance-hour (or partial hour)	744 Hrs	$178.56
Total:		**$178.98**
EBS		
$0.095 per GB-Month of snapshot data stored	0.492 GB-Mo	$0.05
$0.10 per 1 million I/O requests	15,250,964 IOs	$1.53
$0.10 per GB-month of provisioned storage	138.183 GB-Mo	$13.82
Total:		**$15.40**

Figure 2.5: Example Amazon EC2 End-of-Month Bill

Figure 2.5 shows the breakdown of costs for each of the instances used during one month, as well as the associated charges relating to Amazon EBS. The cloud user also has the ability to check the amount of resources they consumed during a specified period by downloading a copy of their Usage Report. The Usage Report is an XML or CSV file that displays the amount of resources consumed per service; an example section of a Usage Report is shown in Figure 2.6.

```
<OperationUsage>
      <ServiceName>AmazonEC2</ServiceName>
      <OperationName>InterZone-Out</OperationName>
      <UsageType>DataTransfer-Regional-Bytes</UsageType>
      <StartTime>10/05/11 20:00:00</StartTime>
      <EndTime>10/05/11 21:00:00</EndTime>
      <UsageValue>7512592218</UsageValue>
</OperationUsage>

<OperationUsage>
      <ServiceName>AmazonEC2</ServiceName>
      <OperationName>InterZone-In</OperationName>
      <UsageType>DataTransfer-Regional-Bytes</UsageType>
      <StartTime>10/05/11 20:00:00</StartTime>
      <EndTime>10/05/11 21:00:00</EndTime>
      <UsageValue>7756651392</UsageValue>
</OperationUsage>
```

Figure 2.6: Example AWS XML Usage Report

Figure 2.6 shows the number of RDT bytes transferred inwards and outwards from an instance. Note that Usage Reports can be large in size as they detail the resources consumed per hour for each AWS service. The Usage Report is updated each hour to reflect the cloud user's current resource usage, however costs may be updated up to one day after a particular resource has been consumed by the cloud user's instance.

We have now explored all of the concepts of Amazon EC2 that are necessary for understanding the experiments performed and results obtained that are outlined later in the thesis.

2.4 Volunteer Computing

The number of privately owned devices such as desktops, laptops, tablets and smartphones for example, are estimated to account for one billion of the computational devices currently within the digital consumer market [48]. Crin *et al.* assume that a typical PC consists of 4GB RAM, 1TB storage and has a 10Mbps network connection and therefore outline that a theoretical infrastructure exists that has the computational capability of 100 ExaFLOPS, 10 Exabytes of storage and can achieve a bandwidth of 1 Petabit per second [79].

Though many devices may be switched off or disconnected at any time, the available resource pool will still be large. While many will be concurrently in use, the potential spare resource capacity available is of a great magnitude as PC devices typically remain idle for large periods [111]. This therefore makes this theoretical infrastructure potentially one of the most powerful distributed systems on the planet [142].

In this section, we give an overview of volunteer computing and how even a subset of computational and storage resources can be utilized from this theoretical infrastructure. We then discuss Grid computing; a form of distributed computing that is closely linked to the volunteer computing model. We also describe its differences in relation to volunteer computing and cloud computing as well as its benefits and drawbacks. Finally we describe the Grid middleware platform BOINC.

2.4.1 Overview

Volunteer computing is a form of distributed computing where members of the public are able to offer computational and storage resources to scientific research projects [52]; or indeed to other projects that capture the public imagination. Introduced in 2006 by Luis Sarmenta [191], volunteer computing became popular with the SETI@Home project [47]. SETI@Home allows distributed volunteer users to offer computational and storage resources from their commodity devices to help in the search for extraterrestrial intelligence.

Nowadays, a wide range of scientific projects are available from various scientific fields such as computational biology, climate prediction and high-energy physics [142]; members of the general public are even able to create their own scientific project if they have the technical skills to do so. The latest known figures show that 900,000 volunteer users donate their computational and storage resources to 60 scientific projects [48].

Volunteer computing typically conforms to a basic master-slave architecture where slaves request jobs, the job is executed and the results are returned to the master for analysis. The simplicity of the approach is one reason why volunteer computing has been a popular computational model. However, volunteer computing comes with additional challenges that are as common in cluster and Grid computing.

Volunteer users cannot be trusted to return valid results, reliability protocols must ensure each task is completed in the face of volunteer host churn and tracking all available volunteer hosts and tasks within the system must be performed accurately. The scientific application must also be engineered for the context. Furthermore, dealing with host heterogeneity, maintaining scalability and minimizing overheads are also great technical challenges showing the complexities volunteer systems must overcome in order to successfully contribute to science.

While these systems are technically successful, there are however unavoidable downfalls of volunteer computing, in particular for those who donate resources. For example, volunteer users may observe a decrease in performance when volunteer tasks are executing. This in turn may also cause CPU fans to spin faster and increase noise; CPU overheating may occur if the fans are unable to cope. Furthermore, an increase in power consumption will occur if a CPU is executing tasks when it otherwise would be idle. Security is also another issue where volunteer users must trust the volunteer project does not distribute malicious or untrustworthy applications or that certified scientific tasks do not behave abnormally.

The technical aspects of volunteer computing have received much attention since its introduction however attracting and retaining volunteers has proved one of the most difficult challenges. Volunteer users may be attracted to a particular project to help advance research or be engaged in the social interaction that volunteer computing offers. Furthermore, volunteer computing platforms typically offer credits when volunteer resources are utilized hence in turn, competition and recognition of achievement also provides volunteer users incentives to donate resources [79]. Nov *et al.* aimed to determine why volunteer users donate resources and their level of resource contribution by investigating the popular SETI@Home project [174]. Their findings show that enjoyment and reputation do not significantly impact on contribution however the goals and values of the project do.

Darch *et al.* classify volunteer users either as *Super Crunchers*, *Lay Public* and *Alpha-Testers* [87]. *Super Crunchers* compute a large quantity of scientific data, the *Lay Public* make a smaller contribution and *Alpha-Testers* are recruited by volunteer projects to perform early testing. The authors find that *Super Crunchers* contribute due to the credit system and praise they receive for contributing lots of resources. The *Lay Public* may also find the credit system a major pull factor as well as contributing to science and society. The *Alpha-Testers* are typically engaged by the reputation they receive by testing the early features of volunteer projects.

2.4.2 The Grid

Due to the similarity between volunteer and Grid computing, the Grid model also offers similar advantages but equal challenges to overcome. Grid computing is a form of distributed computing that combines geographically distributed resources to create a high throughput computing infrastructure [108]. This global infrastructure facilitates the sharing of resources and access to a large-scale computational platform that would otherwise be unavailable. Without the advances of networking technology in the mid-1990s, the Grid would not have been able to provide an effective collaborative data sharing and analytical infrastructure for use by researchers [217].

The Grid infrastructure is typically composed of geographically distributed and voluntary resources provided by an organization, for example a university or business. Within these institutional boundaries, an organization has the responsibility of providing a scalable, flexible and secure environment for researchers [217]. An organization must conform to a set of open standards and protocols when developing Grid solutions. For example, the Open Grid Services Architecture (OGSA) exists to define policies of how to share data between various institutional boundaries [30].

Due to the distributed nature of the Grid, the infrastructure is inherently heterogeneous, loosely coupled and dynamic [63]. However, a shared global platform also fosters global collaboration between organizations to execute tasks which solve problems to reach common goals; a primary reason why Grid computing was born [157]. Grid infrastructures can execute a variety of tasks from many research project areas such as high-energy physics, bioinformatics and chemistry, for example. These applications may be also be distributed in nature and require high-throughput or fast data processing capabilities. A well cited use of Grids is the work being performed at CERN to help analyze and store the vast amounts of data produced from the Large Hadron Collider (LHC) experiments [217, 109, 157]. Data is transferred over the World LHC Computing Grid (WLCG) [44] from Geneva to various organizations called *Tiers*; an example of the scale of data transfers is shown in Figure 2.7.

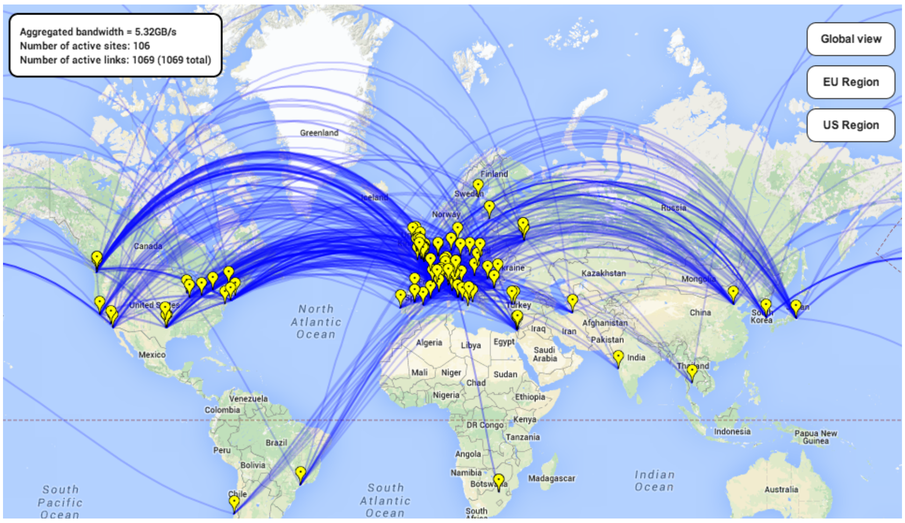


Figure 2.7: WLCG Data Transfers [43]

Three types of tiers exist: Tier-0, Tier-1 and Tier-2 [114]. The Tier-0 centre consists of the shared infrastructure available at CERN and has the purpose of data recording, performing initial analyses and distributing data from their experiments. Tier-1 centres store data, perform large-scale preprocessing and store subsequent results. Tier-2 centres are typically universities and organizations who can store and analyze small amounts of data. Figure 2.7 shows that at the time of writing, the aggregated bandwidth usage was 5.32 GB/s and this data was being sent from CERN to 106 sites over 1069 links. The coordination of data and distributing computations over the Grid infrastructure are a few of the many technical challenges that must be overcome to successfully operate over the Grid.

Many of these challenges are solved by Grid middleware such as Globus [78, 107], gLite [149, 159] and Condor [154, 204]. However these middleware frameworks can be complex and target the use of an organization's dedicated resources within the Grid. There are however substantial idle resources within organizations that could be utilized, hence Grid middleware such as BOINC [47] and Xtermweb [100], as well as Condor aim to take advantage of these resources to create a Desktop Grid.

While conceptually similar to volunteer computing, Desktop Grids are composed of a more homogeneous set of resources and are either under the same ownership or owners agree to common management policies to achieve a goal. On the other hand, volunteer computing is composed of a wide range of heterogeneous resources dispersed worldwide that are unreliable in nature. We now focus on one Desktop Grid middleware and volunteer computing system that is core to our *ad hoc* cloud computing model.

2.4.3 BOINC

The Berkeley Open Infrastructure for Network Computing (BOINC) is an open source client-server middleware system created to allow projects with large computational requirements, usually set in the scientific domain, to utilize a technically unlimited number of volunteer machines distributed over large physical distances [47]. Created in 2002, BOINC has become one of the most popular volunteer computing middleware systems.

2.4.3.1 Overview

The success of BOINC can be attributed due to its simplicity and ease of use from a volunteer user's perspective as well as its architecture in general. BOINC follows a basic client-server model. Volunteer users must download BOINC and select or enter their desired project in order to obtain tasks from the appropriate BOINC server; there are very few actions that must be performed afterwards and BOINC can execute indefinitely without user intervention. The BOINC architecture is shown in Figure 2.8.

Two important components are depicted: the BOINC client and the BOINC server. The BOINC client is an application that is installed on the volunteer host and has the purpose of communicating with the server, attaching the client to single or multiple projects, organizing the computation and returning results. The BOINC client is composed of four components as shown in Figure 2.8 [49]:

- The core client: communicates with the server, attaches clients to projects, organizes the computation, executes the application and returns the result.
- The *boinccmd* API: a command-line interface for controlling the core client. It is able to obtain new tasks, suspend computations, upload results, reset the project, etc.

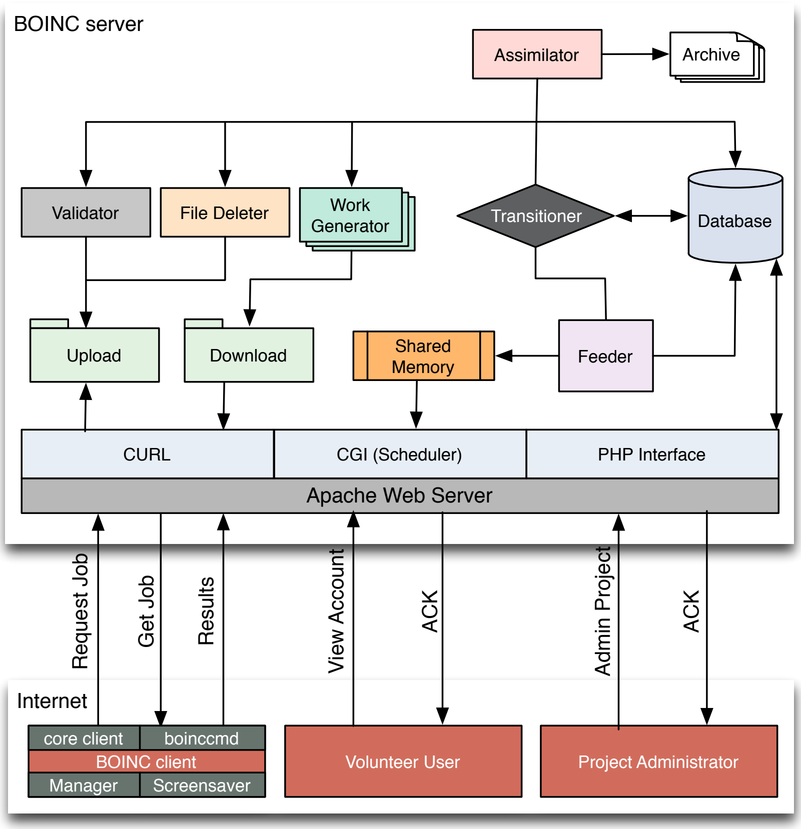


Figure 2.8: BOINC Architecture; derived from [98]

- The BOINC Manager: a Graphical User Interface (GUI) representation of the *boinccmd* API. The Manager also shows all attached projects, current downloads, computational progress, etc.

- Screensaver: a project specific screensaver displaying graphics of a running task; whether a screensaver exists is project dependent.

2.4.3.2 The BOINC Process

Upon running the BOINC client for the first time, a series of benchmarks are executed to determine the true speed of a host's CPU. The total resource capacities and available disk space are also recorded. Once connected to a scientific project, the BOINC client will receive an application from the BOINC server to execute.

The application itself typically consists of an application executable, that has previously been compiled on the target host type, and a series of input and output files [79]. The application must have checkpointing measures in place to allow the computation to continue if BOINC is quit by the user or the host terminates or fails [49]. During the execution of an application, the BOINC client records the amount of work performed by the volunteer host and issues credits to the user which are published on-line. Credits are calculated by multiplying the application's CPU time by benchmark scores.

Conceptually the BOINC client is a simple application however much of the system's complexity resides on the BOINC server. The BOINC server has the main purpose of hosting the scientific project (e.g SETI@Home) and creating, distributing, collecting, storing and validating Results from many clients [111]. Results are instances of a particular BOINC Work Unit (i.e. a particular scientific task) regardless if the Work Unit has been completed or not. To store and distribute these Results, the server uses MySQL for data storage, while Apache and PHP are used for web access; for example, to allow a volunteer user to modify project preferences or a project administrator to configure the project.

The BOINC server is underpinned by a set of running daemons that create and coordinate entities related to the project [51]. A set of default daemons are provided however additional daemons can be added dependent on the project characteristics and functionality required. After an application developer has created their scientific project, and as shown in Figure 2.8, the *work generator* daemon begins creating project Work Units and stores these in the 'Download' folder. The *transitioner*, whose task it is to manage the state transitions of Work Units and Results, then generates multiple Results from a single Work Unit and stores these in the database.

The *feeder* periodically extracts these Results and enters them into a shared memory region. The *scheduler*, which has the purpose of communicating with client using XML messages, coordinates outbound Results from the shared memory region to clients while concurrently dealing with completed Results. Received Results are placed in the 'Upload' folder and the *transitioner* is informed. The *validator* is then instructed to validate the received Results. BOINC does this by adopting replication where each job is executed on multiple hosts. By comparing the Results received from different clients, BOINC ensures that host errors or security breaches have not influenced the Results. Credits are issued to hosts only if the Result is deemed as valid [51].

Once a Result has been validated, a Canonical Result is created; a Result which is the simplest and best of those validated. Optionally, the *assimilator* may perform an administrator-defined action such as archiving Canonical Results to long-term storage. In order to reduce storage space consumption on the server, the *file deleter* removes Work Unit data files and Results that are no longer required.

2.4.3.3 The Performance of BOINC

In order to attract and retain volunteer users, the performance of both the BOINC server and client must be acceptable. The BOINC client should not cause significant overheads or slowdown of the volunteer host and the processes currently running on it. This however is managed by the volunteer user via project preferences.

The volunteer user is able to control aspects of the job by adjusting these preferences via their account hosted on the BOINC server [49]. They can control the minimum interval between checkpoints, the maximum utilization of a processor, the total disk, memory and network usage allowed or even the time of day the volunteer host can be used; many other options exist but for brevity, are not explained here. We show in Chapter 4 that no significant overheads exist while executing an application using BOINC.

As most of the complexity of the BOINC system resides on the server, achieving good performance is critical to meet the demands of BOINC clients. As the number of volunteer hosts can range from tens of volunteers to potentially hundreds of millions [51], it is especially important that the server scales well when there is an increase in the number of client requests for work or Results uploaded.

Anderson in 2005 [51] performed an analysis of the BOINC server performance and found that an inexpensive computer (2GB RAM, 2 x 2.4 GHz processors and 480 GB storage) hosting the server can distribute approximately 8.8 million tasks per day. Excluding file upload and download, which is project dependent, a network offering approximately 8.2 Mbps would be needed to cope with this number of tasks. The main performance bottleneck was the CPU which reduced database performance and limited the number of tasks per day that could be distributed.

Amdahl's Law dictates that CPU speeds double approximately every 18 months, hence nowadays, we would expect the BOINC server to process and distribute more tasks per day. Hence, based on these measurements, it is reasonable to assume that any modification to the BOINC server would have little effect on performance, which we show in Chapter 6, and the ability to achieve 8.8 million tasks per day.

However, as the trend in CPU speed progresses, the available network bandwidth will become the bottleneck, especially in the future as applications become more data-intensive. The number of volunteer resources may also have to increase substantially in order to cope with large storage demands if disk technology and performance does not keep pace with the increase in CPU speed.

2.4.4 Grids, Clouds and Volunteer Infrastructures

Now that we have described all computational models that are either directly or indirectly related to *ad hoc* cloud computing, we offer a brief comparison between Grid, cloud and volunteer computing.

Grid computing has provided cloud computing with fundamental aspects of distributed computing enabling it to thrive in recent years. They both share common entities such as being available via an Internet connection and offer geographically distributed resources [108, 56]. They are also scalable, created for multi-tenancy and both are trusted to provide reasonable performance and security.

Clouds and Grids are however different in many ways. Perhaps the most significant difference is the use of virtualization in cloud infrastructures for reasons previously mentioned. Furthermore, cloud mandates the use of virtualization whereas Grid permits it but does not require it. Therefore applications do not need to be modified for use on the cloud as a virtual machine can be modelled on an end-user's OS and local resources. On the other hand, Grid does require applications to be modified and submission scripts must be created to execute the application.

Another significant difference is that Grid computing components are owned by a consortium of organizations who agree to conform to a common implementation operation and use model, whereas cloud computing infrastructures are owned and maintained by a single organization that chooses its own model. Although a degree of trust exists between both the users of cloud and Grid infrastructure providers, the core concept of Grid builds upon stronger levels of trust between organizations in order to foster collaboration. This was highlighted when Ashley *et al.* were the first to connect organizations from three continents into a single large-scale research Grid in the Asia-Pacific region [155].

As Grid computing was built on the premise of data sharing and collaboration, cloud computing is typically known to offer a commercialized version of this computational model and is targeted at businesses rather than researchers. This has resulted in cloud infrastructures becoming service orientated for business requirements, whereas Grids that offer service-based functionality, are typically aimed at scientific research or large-scale computations; for example, the WLCG. Furthermore, HPC applications are less well suited to running effectively on the cloud, however they are suitably matched to the Grid. The location of data is also unknown when utilizing commercial and private cloud infrastructures.

Two features that are core to the cloud computing model but are typically omitted from Grid infrastructures are on-demand resources and elasticity. The high throughput nature of the Grid as well as the large number of computations to be performed, typically results in well populated queues containing applications waiting to access resources. Therefore obtaining resources on-demand within a Grid infrastructure is uncommon, however on-demand access to Grid resources could be possible if the Grid were underutilized. Despite this, the static provisioning of resources typically employed by batch job submissions also prevents resources from being available almost instantly, as well as being elastic. However as computational demand increases, the elastic nature of cloud computing may decrease as resources become stretched.

Volunteer and cloud computing are two completely different computational models and their differences have previously been outlined in Chapter 1. However in addition, unlike cloud computing, volunteer computing does require that an application is modified for it to be executed on volunteer resources; these volunteer resources may be from cloud or Grid infrastructures. Volunteer and Grid computing are closely linked where resources are harnessed and offered to scientific research, however these resources are typically not available on-demand. For example, volunteer resources may be unavailable and Grid resources may only be accessed through a batch system and subsequent queue. Although the potential resources available to volunteer infrastructures are of great magnitudes, Grid computing should offer better performance due to the co-location of compute resources and the reliability of resources.

Volunteer computing does however offer access to computational resource at a much lower cost compared with an organization that purchases and maintains resources for the Grid. Volunteer infrastructures do have a greater host churn and failure rate and volunteer resources cannot be trusted; this is in direct contrast to Grid computing. Grid computing is however able to execute a larger range of applications whereas volunteer infrastructures mainly execute embarrassingly parallel CPU-intensive tasks. The similarities outlined are in no way a complete list and opinions may differ. However, although the aforementioned similarities may appear minute in some cases, the research and technical challenges enabling each of these computational models is substantial.

2.5 Monitoring and Management

Server monitoring and management are critical components of infrastructure administration allowing the verification of resource use, observation of server performance, identification of failures and control of servers. These components are especially useful in complex systems where administrators are not able to understand and control their infrastructure directly; this is especially true in the case of *ad hoc* clouds.

In this section, we give a brief overview of the basics of server monitoring and management; we define the latter as the ability to control individual or multiple servers concurrently to acquire a desired state. We then discuss two existing monitoring tools, Ganglia and Nagios, to help understand the literature presented in Chapters 5 and Chapter 6 respectively. Finally, we outline various infrastructure management tools that have the potential to be used in an *ad hoc* cloud and how they are able to offer infrastructure control.

2.5.1 Overview

The rise of high throughput computing posed many challenges on how to operate such large and dynamic infrastructures efficiently and successfully. As the number of hosts increased and the nature of the system became more distributed, system monitoring and management became an important yet complex task, without which computational platforms such as the Grid may not have become successful.

The technological advances of the last decade have increased the functionality of modern monitoring and management software. Hence, there exists a large number of Grid and HPC monitoring tools such as: MapCenter [68], GridICE [53], R-GMA [83], GridRM [58] and Supermon [201], to name a few. This is in addition to smaller tightly integrated command-line tools for monitoring specific aspects of a system; for example *iperf* [24] and *tcpdump* [38]. However, as new tools are created or further technological advances are made, we must still consider the *why*, *what*, *how* and *when* of monitoring and managing systems.

2.5.1.1 Why monitor and manage?

The reasons to monitor and manage an infrastructure are typically well known and have been previously mentioned. For example, Grids require monitoring for performance analysis, tuning and prediction as well as for scheduling and fault detection [59]. However some may require monitoring and management in order to maintain high availability or cut costs by dynamically powering down underutilized machines. Management is also a critical component to ensure an infrastructure requires less skilled effort to operate and is under administrator control at all times.

2.5.1.2 What to monitor and manage?

Past and present monitoring software typically monitors a set of common metrics such as CPU, memory, network and disk usage. Nowadays, some monitoring software adds administrator alerting and autonomic response to events or failures [27]; others allow system administrators to extend the set of metrics [162].

There are however two entities that can be monitored: the entire infrastructure or a particular service running upon the infrastructure [153]. Infrastructure-based monitoring gathers performance metrics relating to the entire infrastructure. Service-based monitoring measures metrics related to a specific service running on an infrastructure. Management software can also conform to either of these models, however if system administrators have to the ability to control hosts, they also have basic control of the services running on these hosts, e.g. restart, stop and start.

2.5.1.3 How and when to monitor and manage?

Various monitoring software packages have different approaches of how to monitor an infrastructure or service effectively. This depends on why monitoring is being addressed and what are the entities to be measured.

A large number of monitoring tools follow a standard model of how to monitor an infrastructure; the Grid Monitoring Architecture (GMA) [205] developed by the Global Grid Forum [29] (now the Open Grid Forum) offers a good overview of this standard model. This has provided many monitoring software developers with a foundation to build on. We briefly describe the standard architectures employed by many current monitoring tools whose architecture is similar to that proposed by the GMA.

Many system monitoring tools subscribe to the model where both a *producer* of data and a *consumer* of data [225] work together to provide a monitoring service. The *producer* executes on the host to be monitored or the host that executes a service, dependent on the type of monitoring employed. The *producer* of data periodically polls the underlying system for data at a rate set by the system administrator. This data is then either pushed to a *consumer* and/or is pulled by a *consumer* running on a different host; this is dependent on the data transfer model implemented and this must be tolerant towards host failures.

The *consumer* process typically resides on a dedicated server [213] that stores and displays the monitoring data in a human-presentable format to a system administrator. This data can also be analyzed periodically by applications that check quality of service delivered, check for intrusion and compute cumulative operational statistics. Some of these are made inspectable by the system's users and all are available to the providers of the infrastructure.

As many monitoring software packages utilize the standard monitoring model, these packages are also advertised as being scalable, reliable, available, highly accurate and having low network overheads. However, studies have found many tools do not offer the features as advertised. For example, Volk *et al.* note that scalability is not well implemented in current monitoring tools and that data visualization will no longer be adequate as data grows [210]. Furthermore, monitoring software is inflexible and unable to cope with dynamic infrastructures [82, 213, 220] due to the static way these tools are configured.

Management software is also known to face similar problems of inflexibility and in many ways have challenges similar to monitoring software packages due to the architecture it employs. Like monitoring tools, the core of a system management software package typically resides on dedicated servers for centralized control, displaying changes in system state and ensuring local and remote host security is managed appropriately. Similarly, infrastructure control may only be possible if the tool is also installed on the hosts to be managed. Alternatively, many tools do not need software installed on the monitored hosts as exchanging SSH keys can be used to allow remote control via password-less login [61].

2.5.2 Infrastructure Monitoring Tools

We now discuss the Ganglia and Nagios monitoring tools, both of which are described to understand the literature presented in Chapters 5 and 6 respectively.

2.5.2.1 Ganglia

Ganglia is a scalable and distributed monitoring system designed for high performance computing infrastructures such as clusters and Grids [162]. Ganglia is designed to monitor infrastructures by using a hierarchical approach where multicast messages are distributed within a cluster to disseminate the current state of the system to every other host within the system [188]. Like the standard monitoring model aforementioned, Ganglia is composed of two daemons as shown in Figure 2.9.

The *gmond* daemon collects information about the host it runs upon in an XML format and sends periodic heartbeat messages, via a UDP multicast protocol, to the

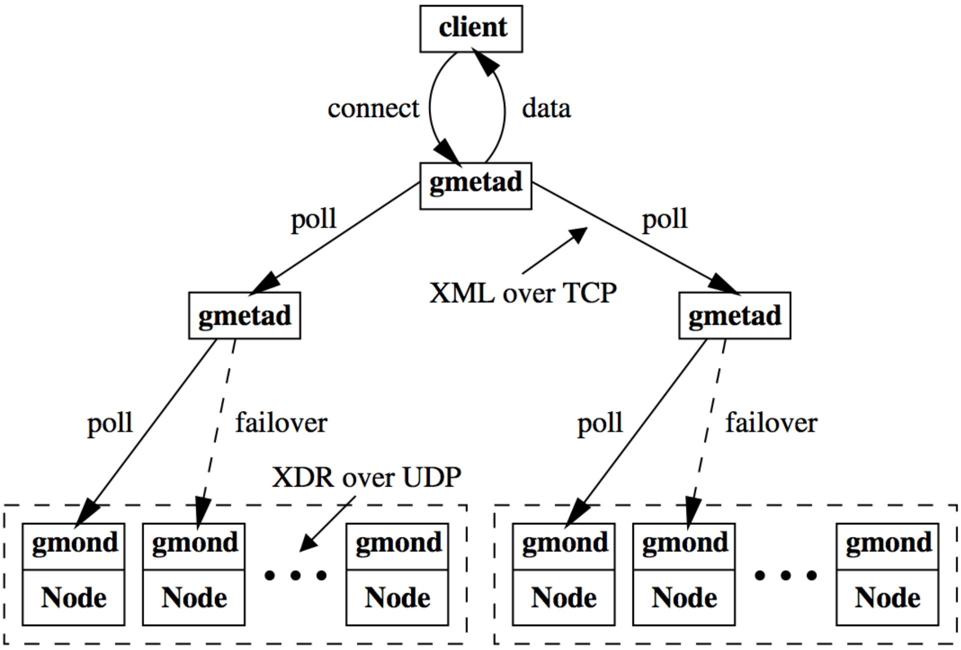


Figure 2.9: Ganglia Architecture [162]

entire cluster. Figure 2.9 shows that two clusters exist which may be within the same local network or be geographically distributed over a Wide Area Network (WAN).

In order to specify which cluster a node belongs to, the file */etc/ganglia/gmond.conf* can be manually edited to include the name of the cluster; further parameters can be set but are not described here. Within each cluster, heartbeat messages are sent via the External Data Representation (XDR) format; this is used for data transfer efficiency. Due to the distributed and hierarchical nature of the approach, the monitoring mechanisms in a cluster are highly decentralized and fault tolerant.

The data in each cluster is periodically polled by the *gmetad* daemon which collects and aggregates the resulting XML containing the data; this daemon can be configured by manually editing the */etc/ganglia/gmetad.conf* file. If any cluster node fails to respond to a request, the daemon selects the next cluster node in the hierarchy to request data from. Upon a client's request (e.g. via the Ganglia web interface), the *gmetad* daemon will export the aggregated XML data from multiple clusters to the PHP web interface for viewing.

Ganglia provides these graphical representations by integrating RRDtool (Round Robin Database) [34]. This is an open source tool for data logging and graphing historical data between a time series selected by the viewer. Figure 2.10 shows a typical graph that RRD can display to the infrastructure administrator. This graph shows a

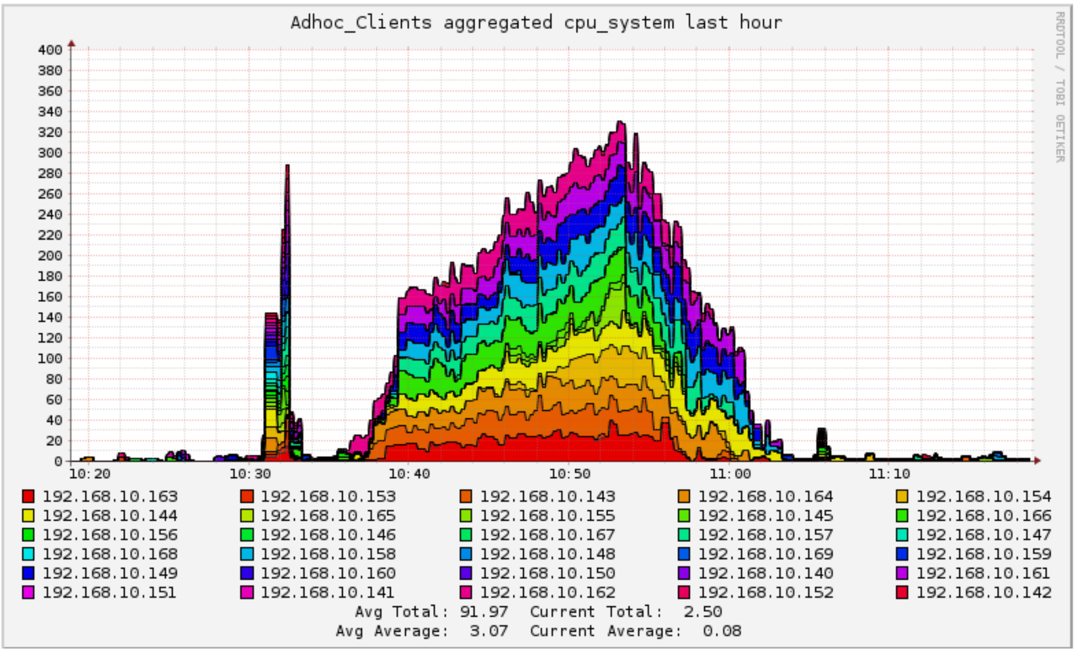


Figure 2.10: RRDtool Graph Example

stacked graph of CPU utilization for each node over a period of one hour; each node is labelled and depicted using distinct colours. Although it may be difficult to distinguish one node from another in this example, RRD allows administrators to inspect graphs in further detail and display the features they would like to view.

Discussion: Massie *et al.* in their initial Ganglia proposal outline the local overheads of the *gmond* and *gmetad* daemons. The authors show that the *gmond* overhead is low when executing on 102 nodes of PlanetLab [162]. CPU utilization is less than 0.1% and physical and virtual memory use 0.9 MB and 15.2 MB respectively. No disk I/O overhead is incurred as *gmond* daemons only maintain soft state. Massie *et al.* also outline the overheads of running the data aggregation daemon *gmetad*. CPU utilization was again less that 0.1% and physical and virtual memory was 2.4 MB and 96.2 MB respectively. Despite these values being low, the I/O overhead of *gmetad* was significant by using 15564.8 kbits/s.

The global overhead of the system is described as being low where monitoring and aggregation nodes use 6 Kbits/s and 272 Kbits/s of network bandwidth respectively; the latter amounts to sending 19.15 GB of monitoring data over the network per week. In contrast, other studies have found that Ganglia introduces considerable overheads [225, 213]. Wei *et al.* realize that many problems exist with Ganglia, such as reliability and that Ganglia may reduce the performance of a network that is performing poorly

[214]. The authors propose an alternative ring structure between aggregation nodes as opposed to the tree structure currently implemented. The authors also propose sending a reduced but common set of metric data around the network as well as limiting how deeply the XML data is recursively displayed. These methods aim to reduce bandwidth consumption and increase the responsiveness of the Ganglia interface for a large number of nodes.

2.5.2.2 Nagios

Nagios is "*the industry standard in IT infrastructure monitoring*" [27] which monitors system metrics, network protocols, servers, network infrastructure and services. Nagios is also scalable to thousands of nodes, stores historical data and is open source; some however dispute its claims of scalability [220]. The centralized Nagios server that hosts the web interface and monitoring core, polls each monitoring source to acquire data.

Despite being primarily used as monitoring tool, Nagios also implements administrator alerting and can respond to failures. In such situations, administrators are notified of any infrastructure problems and can define event handlers that respond to these problems automatically. Like Ganglia, Nagios is also configured by modifying a number of complicated configuration files [129], inherently making it a statically configured monitoring tool. We refer to Nagios in Chapters 5 and 6 during our discussions of how monitor an *ad hoc* cloud effectively and our evaluation of the *ad hoc* cloud, respectively.

2.5.3 Infrastructure Management Tools

Reviewing all existing infrastructure management tools is out of the scope of this thesis, however we describe a select number of infrastructure management tools that are required to understand the literature presented in Chapter 5. The tools investigated are: Webmin, Capistrano and *cexec*.

Webmin is a browser-based system administration tool for Unix [76]. Webmin allows remote command and configuration changes to be performed through a web interface rather than the traditional approach of manually editing configuration files. The design of Webmin has evolved around the concept of modules where a basic service is provided to the administrator and functionality can be extended by adding downloadable modules. For example, modules exist to allow File Manager browsing of remote hosts and others modules are available to control a variety of servers (e.g Apache, MySQL). Webmin requires that software is installed on both the server and clients to allow the former to control the latter.

Capistrano is an open source remote host automation tool that can execute scripts and commands concurrently over multiple hosts [9]. Capistrano does not require any installation of software on remote hosts as it assumes SSH keys have been exchanged between the central server and remote hosts to allow password-less login. This is in contrast to Webmin that uses the software installed on remote hosts to create tunnels to send instructions over. Remote hosts can be controlled by editing the Capistrano configuration file called a *capfile*. An example of this file is shown in Figure 2.11.

```
role :hosts, "129.215.90.83", "129.215.90.84"
task :upload_and_execute, :roles => :hosts do
    set :default_shell, "bash"
    set :user, "ubuntu"
    set :home_dir, "/home/ubuntu/"
    system("cd /home/fedora/ && tar cvzf ’test.tar.gz’ Test/")
    upload("/home/fedora/test.tar.gz","#{home_dir}",:via => :scp)
    run("cd #{home_dir} && tar xvzf test.tar.gz")
end
```

Figure 2.11: Example Capistrano Capfile

This file must contain a *task* which is executed by Capistrano. The task’s name (e.g upload_and_execute) and the hosts the task will be executed upon are specified; the latter is expressed via a *role*. Capistrano will perform this task when the following command is executed:

```
cap upload_and_execute -f /path/to/capfile
```

In this example, the shell is set to ‘bash’ and the remote host user and directory are set to ‘ubuntu’ and ‘/home/ubuntu/’ respectively. The *system* command executes local commands where *upload* copies a file to the remote hosts. The *run* directive executes a command on each of these remote hosts. In this example, a local folder is compressed, uploaded to each remote host and then decompressed; all of these operations are performed concurrently over all remote hosts.

cexec is a cluster tool that simply executes commands over multiple hosts concurrently [132]. Like Capistrano, *cexec* does not require any software installation on target machines as it assumes SSH keys are exchanged between hosts. To execute an instruction concurrently over a set of hosts, the following command is executed from the command line:

```
cexec -f hosts.txt '/etc/init.d/gmond restart'
```

In this example, *cexec* takes two arguments: a configuration file listing the hostname or IP address of the hosts and the command to execute, i.e. restart the Ganglia monitoring daemon.

In this section, we have outlined the *why*, *what*, *how* and *when* of infrastructure monitoring and management and the basics of these topics. We gave Ganglia and Nagios as two examples of monitoring software. The former is used within our C2MS monitoring tool and the latter's output is used to create an experiment to help simulate a number of hosts that an *ad hoc* cloud platform may operate on; both are described in Chapters 5 and 6 respectively. We then outlined the three simple infrastructure management tools Webmin, Capistrano and *cexec* and how they are able to offer basic control of an infrastructure. These tools are analysed in further detail in Chapter 5.

2.6 Platform Testing and Evaluation

The introduction of any new computational method or system involves performing an in-depth evaluation to determine its relative benefits and downfalls. This section describes the benchmarks used to determine the performance, overheads and reliability of an *ad hoc* cloud computing infrastructure.

2.6.1 Overview

Ad hoc cloud computing infrastructures are significantly different in terms of the underlying implementation when compared to established cloud platforms and traditional clusters. As the *ad hoc* cloud is a unique computational model that has not yet been realized, previous work has been unable to determine the performance, overheads and reliability of *ad hoc* cloud computing infrastructures. Various studies however have managed to analyse some performance aspects of the six founding principles of *ad hoc* cloud computing.

We built upon this previous work by confirming the accuracy of those results and then evaluated the performance, overheads and reliability of an *ad hoc* cloud prototype. The wealth of research available outlining the performance and unique characteristics of commercial clouds also allowed us to make a comparison with *ad hoc* clouds.

We have used four benchmark applications to determine the CPU, memory, I/O and disk performance and overheads of various subcomponents of the *ad hoc* cloud platform. They are also used to test the performance and reliability of the platform as a whole. The benchmarks used are the *stress* workload generator, *Primes* and *CreateGB* and *SPRINT*, each of which are described below.

2.6.2 Stress Workload Generator

The *stress* workload generator is a simple benchmarking tool for POSIX systems [37]. The tool has the capability to simulate a variety of workloads such as those that are either CPU, memory, I/O and disk-intensive. Furthermore, the tool tests workloads that are resource-bound in many ways, for example an application that is both CPU and I/O-intensive.

The *stress* workload generator is also able to simulate both single and mulit-threaded applications. These features, as well as its extremely easy to use command-line interface, was why this benchmark suite was chosen. In order to stress a number of resources, a workload can be created using the command-line interface. For example, to simulate a workload that is CPU and I/O and memory-bound, the following command-line arguments can be specified:

```
stress --cpu 2 --io 1 --vm 1 --vm-bytes 128M --verbose
```

This command creates two CPU-bound processes and one I/O-bound process. A further memory process is created that allocates 128 MB of memory to the process; this figure can be increased to achieve the utilization level desired.

The current operation of *stress* executes a particular workload indefinitely however we have modified the tool to fully utilize a resource up until a specified number of iterations. We therefore can obtain a completion time that can be used across different platforms to aid a comparison. The number of iterations for each resource vary for different experiments and the reasoning behind this, as well as the actual values will be outlined during the description of each experiment in later Chapters.

Despite the *stress* benchmark covering a wide range of workloads, we select other benchmarks to obtain in greater detail the affect of a varying set of other resource-intensive applications that will cover a large class of applications that may run on an *ad hoc* cloud.

2.6.3 Prime Number Calculator

Primes is a CPU intensive application used to calculate prime numbers up to a specified value. We use this benchmark to primarily test the performance of V-BOINC. We display the code used to calculate a set number of primes in Figure 2.12. The limit on the number of primes to be calculated are outlined during the description of each experiment in later chapters.

```
#!/bin/sh
start_time=$(date +%s)
echo 2
j=3
rng=300
while test $j -le $rng
do
i=2
x=`expr $j - 1`
while test $i -le $x
do
if [ `expr $j % $i` -ne 0 ]
then
i=`expr $i + 1`
else
break
fi
done
if [ $i -eq $j ]
then
echo $j
fi
j=`expr $j + 1`
done
end_time=$(date +%s)
diff=$((end_time - start_time))
echo 'Runtime: $((diff/60)) minutes and $((diff % 60)) seconds'
```

Figure 2.12: Primes Benchmark Source Code

2.6.4 CreateGB

CreateGB is a memory and I/O intensive function used to create a file of a specific size using the Linux function *dd* [20]. The function *dd* is used to covert and copy files and displays the read and write speed once the function has completed. We use *dd* to read and write to and from varying sized files dependent on the experiment being run. We give example commands for both of these scenarios.

```
dd if=/dev/random of=1GBFile bs=512M count=2 &> write.txt
```

```
dd of=/dev/zero if=1GBFile bs=512M count=2 &> read.txt
```

The former command takes random numbers generated from */dev/random* and creates a 1 GB file with a block size of 512 MB (i.e 512 MB must be written at a time) and writes two blocks to the file '1GBFile'. The read and write speeds are then written to a file for analysis. Reading a file of 1 GB has similar arguments, however the input is re-directed to the data sink */dev/zero*. Similar to other experiments, the parameters of *CreateGB* are outlined during the description of each experiment in later chapters.

2.6.5 SPRINT

The Simple Parallel R INTerface (SPRINT) is a package providing parallel functions of the statistical package R, allowing data to be analysed over multiple processors rather than being performed on a single node [120, 177]. SPRINT was selected because it is a widely used tool in the biomedical community therefore allowing us to obtain real data on how our system copes with a real application.

Due to the ever increasing data set sizes from the biomedical community, many bioinformatics computing infrastructures are being stretched to their computational limits, where performing genomic analyses has now become a lengthy process. In order for such a community to use HPC resources, a researcher may have to learn the HPC infrastructure, as well as parallel programming to take full advantage of the resources. SPRINT allows the researcher to focus on their task at hand by reducing the HPC knowledge required and eliminates the need for users to write parallel programs.

By loading SPRINT onto every computational node via the R programming language, a master node controls each worker node via a task farm approach. When a parallel function is encountered, the work is distributed between the worker nodes using MPI [106]. Figure 2.13 shows an architectural diagram of this approach.

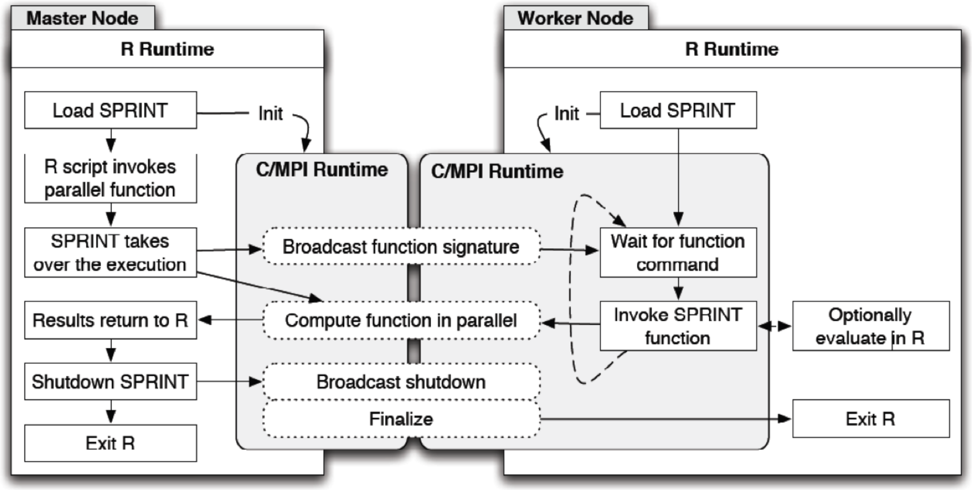


Figure 2.13: The SPRINT Master and Worker Node Relationship [36, 90]

In a large number of cases, both the master and worker nodes will execute SPRINT with equivalent hardware and software specifications, however SPRINT has the ability to run over heterogeneous environments allowing each end-user to tailor their use of SPRINT to their own infrastructure or requirements. Currently SPRINT offers many parallel functions from R, however we only concern ourselves with *pcor* and *pmaxT*.

2.6.5.1 pcor

The function *pcor* is the parallel version of the R serial function *cor* and is a CPU and memory-intensive function. As its name may suggest, it performs correlation on a given data set; Pearson's correlation is used by default. Rather than running this on a single node, *pcor* implements a master-slave approach where the master node coordinates the rest of the processors. Each processor takes a row of the data to correlate with all others in the data set and this occurs on a first-come first serve basis. When a processor completes its task, it is given another until all rows have been correlated. This achieves appropriate load balancing even with heterogeneous worker nodes. Both the serialized and parallel correlation functions take the same input arguments (in a simple case):

pcor(x, y=x), where *x* and *y* are the data to be correlated.

Data may include a matrix containing differing genes each with different samples for different subjects. If this were the case, because R by default calculates pairwise correlations between columns (samples), we instead use:

pcor(t(data))

This transposes the matrix, ensuring that we determine the relationship between different samples from the same gene rather than different samples from different genes. This is the configuration we have used for our experiments and we outline the number of genes and samples used in each experiment later in the thesis.

2.6.5.2 pmaxT

The function *pmaxT* is the parallel version of the serial function *mt.maxT* from the R multtest package and is a CPU-intensive function. Both the serial and parallel versions of *pmaxT* perform a permutation test to determine the statistical significance of the data, expressed through p-values [177]. The p-value is the probability that a gene's expression level is statistically significant between different conditions, e.g. two types of cancerous tumours. It is common place that a p-value lower than 5% (0.05), signifies that the gene expression value in one condition, when compared to another, can occur by chance alone [122]. To calculate this significance, we call:

pmaxT(X=data, classlabel=classlabel, B=15000)

The variable *data* is the data set and *classlabel* gives an index that separates the data into equal chunks to be assigned to available processes. By dividing and assigning a portion of data to each process, once the necessary number of permutations is complete, each process will have a partial observation of the data. These observations are then sent to the master node where, once all observations are received, the adjusted p-values are calculated.

The latter argument *B* is the number of permutations to be performed, either *complete* or *random*, dependent on the data input sizes. If the data size is small, one can afford to perform complete permutation testing, i.e. testing all possible permutations. Complete permutations of a large data set may take a significant amount of time, hence a set number of random permutations will be more appropriate. We outline the number of genes, samples and permutation count used in each experiment later in the thesis.

2.7 Summary

In this chapter, we have discussed the six founding principles of *ad hoc* cloud computing: virtualization, cloud computing, volunteer computing, monitoring, management and testing. Firstly we outlined the basics of virtualization and detailed a select number of virtualization technologies that have the potential to be used within an *ad hoc* cloud computing infrastructure. An analysis of current research showed that most virtualization technologies perform well, however this is dependent on the underlying hardware and the application executing on the virtual machine. We then gave an overview of cloud computing and the benefits and drawbacks other studies have exposed. This led to an analysis of Amazon EC2, its architecture and the typical costs and performance one may expect.

The background then focussed on volunteer and Grid computing as well as the important research that has led to the success of these computational models. By outlining the difference between the two models, we were able to distinguish which model is suited to *ad hoc* clouds. We then gave a detailed overview of the BOINC volunteer system and analyzed previous research to determine its benefits and drawbacks; in particular its performance-related aspects. BOINC however was shown to perform well overall.

Our overview then discussed the current state of a subset of infrastructure monitoring and managements tools. In particular, we focussed on Ganglia, its architecture and performance. Other studies found that Ganglia has a high overhead in relation to the amount of data it transfers over the network. We also gave a brief description of Nagios and the logs it produced. We then outlined three infrastructure management tools called Webmin, Capistrano and *cexec* and showed how they are able to offer concurrent command execution over a set of hosts.

In order to evaluate many of the *ad hoc* cloud computing founding principles above as well as the *ad hoc* cloud as a fully functioning platform, we selected a number of applications, namely the *stress* workload generator, Primes, CreateGB and SPRINT. We also use these applications to test the reliability and performance of the *ad hoc* cloud as a fully integrated system.

Chapter 3

V-BOINC: The Virtualization of BOINC

3.1 Introduction

In this chapter, we discuss how two of the six founding principles of *ad hoc* cloud computing are integrated to provide the basis of our *ad hoc* platform; these are volunteer computing and virtualization. Volunteer computing systems, and in particular BOINC, deal with many of the complexities surrounding non-dedicated hosts. BOINC also provides an infrastructure where computational jobs can be created, sent to volunteers, executed and returned for analysis.

By integrating virtualization into BOINC, we not only create an initial platform that can be extended to solve our research challenges outlined in Section 1.2.1 of Chapter 1, but we can also solve many of BOINC's drawbacks. The drawbacks of BOINC relate to running applications in the user space of the volunteer machine; the portion of system memory where user processes execute. These drawbacks are:

- Project developers are required to port their application to every target machine architecture.
- Project developers need to provide application-level checkpointing to ensure job progress is not lost upon host termination or failures.
- Project developers are limited to creating applications that have no dependencies.
- Users of BOINC must trust that project servers they attach to, will not distribute malicious or untrustworthy applications.

By virtualizing BOINC, an application developer only needs to port an application to a single virtual machine architecture, host security is addressed by sandboxing therefore protecting the host from third party applications and system-level checkpointing is available. Applications with dependencies can also easily execute where dependencies may be pre-installed or attached to a virtual machine. This enables application developers to create more complex applications to obtain results of more value. These challenges are solved by our implementation of virtual BOINC, or V-BOINC.

The foundation of our approach relies on sending lightweight virtual machine images to volunteer clients allowing BOINC applications to run in the virtual machine itself rather than in the user space of the host. This is implemented by installing a BOINC client within the virtual machine image to fetch applications for a user specified project. This is in addition to the BOINC client installed on the user's host to download the virtual machine image.

Our approach to virtualization within BOINC allows V-BOINC to execute applications from typical BOINC projects such as SETI@Home and future projects with applications that have dependencies. This will in turn increase the number of potential applications volunteer infrastructures are able to execute. The use of V-BOINC therefore aims to enable access to computations that could not otherwise be performed, enabling more science, design and business to be done.

In this chapter, we first give an overview of related research describing other studies that have attempted to incorporate virtualization into volunteer infrastructures. This is followed by our own comparison of virtualization technologies to determine which is best suited for V-BOINC as well as *ad hoc* cloud platforms. We then outline the architecture and internal operational processes of V-BOINC while describing its implementation details. This includes how we introduce, distribute and operate virtual machines as well as how to ensure virtual machine sizes are kept as small as possible and how to perform automatic checkpointing. This is followed by an evaluation of our V-BOINC platform. Firstly we determine the performance differences between V-BOINC and regular BOINC. We then show the performance of V-BOINC when executing SPRINT. Finally, we explore the affects of virtual machine checkpointing on volunteer hosts dependent on the class of scientific application executing.

3.2 Related Work

Several other research projects have added virtualization to BOINC. This section reviews this research while paying specific attention to the differences between our own approach and others.

Ferreira *et al.* [102] aim to provide solutions to BOINC's downfalls; namely porting applications to all participant machines and security. The authors employ a virtualization approach to create a BOINC middleware component for use with VMware and VirtualBox called *libboincexec*. Their implementation shows the virtualization approach increases the execution time of an application by 196 seconds for VMware Player [41] and 229 seconds for VirtualBox [40] on average, when compared to running the same applications via the BOINC framework.

While the authors achieve good results, their implementation assumes a virtual machine image is already present and is configured correctly on the volunteer machine and no application dependencies exist. The authors show that in order to run a job within the virtual machine, the application must first be transferred to the host machine and copied to the virtual machine. Similarly, output data must be transferred to the BOINC server via the host machine. This method may however introduce security weaknesses where an application and data can be corrupted before they are copied to the virtual machine and vice versa. Furthermore, when an application and its data are large in size, transferring these to the host and then to the virtual machine will increase the job pre-execution time significantly.

The authors implementation also breaks the BOINC policy of being transparent to the user where many changes are required to the host due to the external dependencies of *libboincexec*. Also the effects of virtual machine checkpointing, for example, the time to create a snapshot and the storage requirements on a volunteer host are not explored; we cover these items in the following sections.

González *et al.* [111] realize that running interpreted applications in BOINC (e.g R, Matlab, Java etc) is difficult when firstly, an application will have lots of dependencies and secondly, it is not possible to send an application environment such as Matlab to a host; licensing issues may however prevent Matlab being installed locally if the package does not already exist. Currently, a BOINC Wrapper exists that allows legacy applications to be run within BOINC, however the authors go further and create a *starter* tool that detects whether the correct environment is present for the application to run successfully and if not, detects missing parts and downloads them. The environment is then deleted after the computation has finished. A problem may occur however if URLs of packages change overtime.

The authors also realize that interpreted applications do not have application-level checkpointing and introduce virtualization via VMware Player to provide system-level checkpointing. By using VMware Player, users of the authors' system will be presented with the virtual machine, violating the BOINC policy of being transparent to the user. In our case, we use VirtualBox to allow headless virtual machines; virtual machines that do not display a window at runtime. Despite the authors use of virtual machine checkpointing, they do not explore its effects on a volunteer host.

González *et al.*s' method may be useful for typical scientific BOINC applications with no dependencies, however our approach also targets applications with dependencies and we also try to customize and open up BOINC so that researchers and organizations can make use of V-BOINC easily and effectively. Similarly, developers at LHC CERN have developed the CernVM that runs data analyses from LHC experiments [71]. The virtual machine image is available to run on many hypervisors such as VirtualBox, KVM, VMware, Xen and Hyper-V Server.

The CernVM/VBoxWrapper Test Project [10] is similar to our project, where virtual machine images can be downloaded to execute computations, however the framework is not customizable to the point where users are able to select the project they would like to join; only LHC computations can be performed. Their server implementation is also not available however V-BOINC's is publicly available [39] and the V-BOINC virtual machine image size is smaller than the CernVM, in turn reducing the transfer time between the server and clients belonging to the general public.

Recently, BOINC offered virtual machine functionality [6] via its *vboxwrapper* program that acts as an interface between the BOINC client and VirtualBox. This program as well as the application and its data are stored in a shared folder between the host and guest, where the computation is then executed. Our approach differs as virtual machine images can be automatically downloaded to the host and execute applications from any BOINC project.

3.3 Virtualizing BOINC

V-BOINC is the virtualized version of BOINC allowing users to avoid the drawbacks of BOINC and take advantage of virtualization [165]. We chose BOINC as our underlying volunteer computing platform not only because it is the most popular and easiest to use volunteer platform, but it also has features that are useful for developing an *ad hoc* cloud computing infrastructure.

We considered other volunteer computing platforms such as Condor, however for example, Condor assumes volunteer hosts are constantly connected and can be trusted [52]; this is commonly seen in Grid environments which Condor was designed for.

BOINC does not make these assumptions and nor does our concept of the *ad hoc* cloud. The architecture of BOINC is also easier to adjust to develop the features we need as part of our *ad hoc* cloud. Furthermore, Anderson *et al.* find that BOINC is able to scale up to two times greater than Condor [52]. Other volunteer computing platforms were analysed and were either deemed to be unfit for our purposes or did not provide enough functionality.

In order to transform regular BOINC into V-BOINC, some complex additions are required. Namely the V-BOINC server distributes virtual machine images, from a BOINC project named V-BOINC, as opposed to distributing scientific applications while the V-BOINC client not only controls the host's BOINC core client but also the virtual machine and it's inner BOINC client. These components are relatively difficult to create and hence they can be downloaded alongside their source code on the V-BOINC page at [39]; this chapter discusses the latter's implementation.

3.3.1 Virtualization Technologies

In order to integrate virtualization into a volunteer computing infrastructure, we must firstly define the characteristics we require of the virtualization software package. These requirements are listed in Table 3.1 alongside the three most relevant virtualization technologies and whether they satisfy our conditions.

Requirement	**QEMU/KVM**	**VirtualBox**	**VMware Player**
Unique IP Address Allocation	✔[1]	✔	✔
Headless VM	✔	✔	✔[1]
Image Size < 235MB (tar.gz)	✔	✔	✔
Boot Time < 20s	✔[2]	✔	✔
Basic VM Control	✔	✔	✔[1]
Remote Command Execution	✗	✔	✔[1]
Checkpointing	✔	✔	✔
Portability (Mac & Linux)	✔[3]	✔	✗

[1] additional configuration and/or installation required on host

[2] only when used with KVM enabled

[3] KVM component not available on Mac OS X

Table 3.1: Virtualization Technology vs V-BOINC Requirements

The software packages chosen are QEMU/KVM, VirtualBox and VMware Player. Other technologies were analysed and were either deemed to be unfit for our purposes or did not provide enough functionality. For example, Xen was not included in our comparisons as it is primarily a bare-mental hypervisor.

We require that these virtualization technologies allow bridged networking to give the virtual machine a unique and Internet accessible public IP address, enabling the virtual machine's inner BOINC client to directly receive jobs and return results to a BOINC project server. The chosen package must also adhere to the BOINC policy of being unobtrusive to the user, offer API's for basic virtual machine control (e.g. start, stop, etc) and allow command execution on the virtual machine.

Furthermore virtual machine checkpointing must be available and the chosen package must be portable to both Linux and Mac OS X machines; the platforms V-BOINC targets; future work will include Windows platforms. We also specify that the virtual machine image must boot within 20 seconds; this is significantly faster the startup times of an instance on Amazon EC2 [175, 127, 194]. Finally we require that the size of the virtual machine image file while compressed is less than 235 MB, therefore being smaller than the current size of the CernVM; the project most similar to ours.

We see from Table 3.1 that QEMU/KVM satisfies the majority of our requirements, however it does not offer an API for executing commands upon the guest. Furthermore, to obtain a unique IP address, QEMU/KVM requires configuration changes and additional installations on the host that are unreasonable to ask a volunteer user to undertake. The resulting virtual machine does satisfy our boot time requirement by instantiating in 11 seconds, however this is only when the KVM component is enabled to increase performance; a component that is not available on Mac OS X. Without the use of KVM, the performance of the virtual machine would decrease significantly.

As VirtualBox is based on many QEMU components, VirtualBox also satisfies our boot time requirement by instantiating the same image on the same host in approximately 13 seconds. Most importantly, the major advantage of VirtualBox is the ability to easily start the virtual machine image with a Network Bridge Adapter via Ethernet or wireless giving the machine a unique IP address and identity. The VirtualBox API called VBoxManage, also simplifies the task of controlling virtual machines where QEMU's equivalent provides less relevant options and remote commands can be executed upon the guest via the *guestcontrol* function.

VMware Player satisfies most of our requirements but the virtualization technology is not available on both Mac OSX and Linux. VMware Player only offers headless virtual machines, basic control and remote command execution if the VIX API is installed.

To avoid additional installation of packages and configuration on the volunteer host and with ease of use for the volunteer user in mind, the most suitable candidate for use within V-BOINC, and therefore the *ad hoc* cloud, is VirtualBox; other studies have shown that VirtualBox is a suitable choice for virtualizing HPC [223] and volunteer [102, 6, 10] environments.

V-BOINC currently supports the VirtualBox versions 4.1.8, 4.2.18 and 4.3.6; however later versions should also work but they remain untested. In the future, V-BOINC will support the above hypervisors to increase the user base of this volunteer computing paradigm. We now give an overview of how V-BOINC operates and runs computational jobs.

3.3.2 Methodology Overview

The foundation of V-BOINC relies upon five components each shown in Figure 3.1:

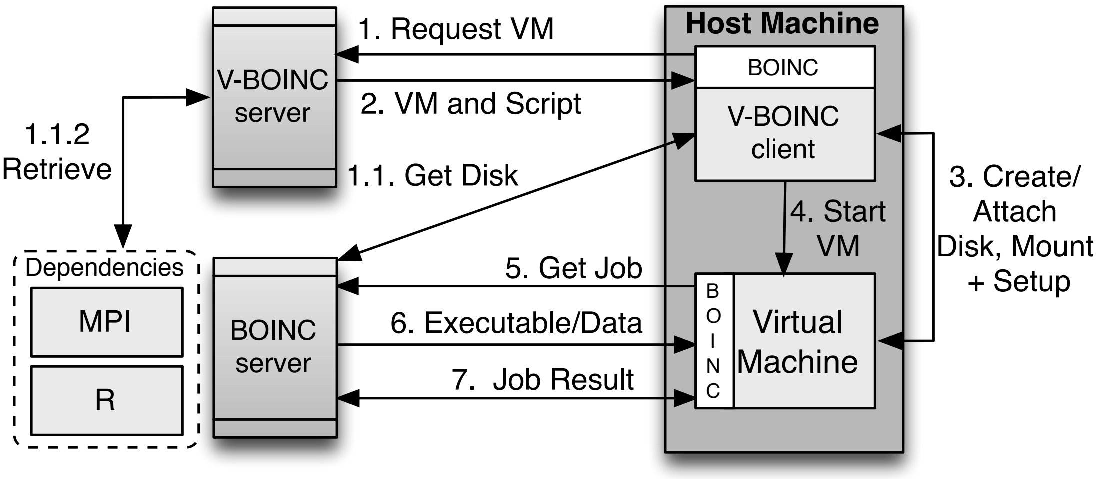


Figure 3.1: V-BOINC Implementation Overview

- **V-BOINC server:** A modified BOINC server distributing virtual machine images, as opposed to scientific applications, to attached volunteer hosts.
- **V-BOINC client:** A downloadable package encapsulating a modified BOINC client and a GUI with the purpose of communicating with the V-BOINC and BOINC Servers as well as the host virtualization hypervisor.

- **The virtual machine (VM):** The platform the BOINC scientific application will execute on. The V-BOINC virtual machine uses the Ubuntu Server 11.04 OS and runs on a VirtualBox Virtual Disk Image (VDI). A single OS is currently used for initial deployment of the project to the volunteer user community, however we envisage an extensive variety in the future. By default, the V-BOINC virtual machine is set to use at most 1 GB of RAM and 1 processor however this can be changed by the volunteer user.

- **BOINC server:** A typical BOINC project server that provides scientific applications to attached volunteer hosts.

- **Dependency Disks (DepDisk):** A separate VDI containing the application's dependencies.

We see from Figure 3.1 that after a volunteer user submits the details of the BOINC scientific project they wish to attach to via the V-BOINC client (e.g the BOINC project server URL and their BOINC project weak account key), the host BOINC client is instructed to request a virtual machine image (1). Concurrently, the V-BOINC client probes the BOINC server to determine if any dependencies exist for the specified project (1.1). If dependencies exist, a VDI (or *.vdi*) file containing the dependencies is retrieved (1.1.2) and transferred to the V-BOINC client via *curl*.

In the event that multiple BOINC projects exist on the same BOINC server, the BOINC project's URL can be used as a unique identifier to allow the V-BOINC project to correctly determine whether dependencies exist for the volunteer user's chosen BOINC project. This ensures the V-BOINC client always receives the correct dependencies for the application that will execute in the virtual machine. We assume that developers of BOINC projects who wish to deploy applications with dependencies are prepared to create a VDI file containing the dependencies and make this publicly available on the BOINC server to allow the V-BOINC client to determine whether a DepDisk needs to be downloaded.

Concurrently while a DepDisk is downloading, the virtual machine image and an executable script are downloaded to the volunteer host's BOINC client (2); both download processes must complete before proceeding to the next step. The V-BOINC client either attaches the DepDisk, if the application is found to have dependencies, or alternatively creates an empty disk and mounts this to the virtual machine image (3). The virtual machine image is then started (4) allowing it to request (5) and receive (6) BOINC jobs and return job results (7).

3.3.3 Lightweight, Flexible and Robust VMs

The purpose of attaching/creating mountable DepDisks above (1.1/3) is well justified for a number of reasons. We do not want to rely on volunteer host dependencies where packages must be present and in a specific location on a volunteer machine. This limits the number of hosts available to a specific project due to the many different host configurations possible. Instead we use virtual mountable disks making it an easy and effective method for applications with dependencies to run.

An alternative model would involve storing application dependencies on the virtual machine before sending it to the volunteer host. However due to the potentially large number of updates BOINC developers may make to their dependencies, additional bandwidth would be consumed by transferring VDIs from the BOINC developer to the V-BOINC server, in turn frequently triggering the re-build process of virtual machines. Note that software packages (e.g MPI, R, Java, etc) could be transferred and utilized via regular BOINC, however one would not be able to benefit from advantages of virtualization.

To reduce the bandwidth consumed by transferring V-BOINC virtual machine images to volunteer hosts, the virtual machine has been stripped of all unnecessary components such as Linux swap space and nonessential packages. As a consequence, no extra disk space exists, hence virtual mountable disks are required, not only for adding application dependencies, but for adding disk space for applications to use. Where no dependencies are required, a fresh disk is locally created on the volunteer host and mounted; the default disk size is 8 GB however this value can be adjusted by the volunteer user. In both cases, Linux swap space is re-established to ensure the performance of the virtual machine is not degraded. As a result of distributing stripped virtual machines, no bandwidth is wasted by transferring these images with unused disk capacity.

In order to create the smallest usable virtual machine image possible, our virtual machine uses the VirtualBox Fixed Disk Image (FDI) type as opposed to the Dynamic Disk Image (DDI). The former is of fixed size and the latter has the capability to grow according to how much is stored on it, up to a specified maximum; this image however does not decrease in size when items are removed from a virtual machine, therefore making it difficult to keep the image as small as possible. For example, an FDI file with the OS components installed could reach 681MB, however with the same components installed, a DDI could be 700MB. It is important to keep the size of the virtual machine VDI at an absolute minimum to reduce the data transferred and stored on the host. The current size of our virtual machine VDI is 649 MB uncompressed and 207 MB compressed.

On the other hand, DepDisks use the DDI type to minimize the initial storage required on the host. For example, when the virtual machine image is downloaded and the DepDisk attached, the minimal storage possible is consumed due to the combination of different disk types used. By essentially partitioning a virtual machine over two VDI files, we ensure that when a user attaches to another BOINC project, a new DepDisk need only be mounted to the virtual machine as opposed to downloading both a new virtual machine image and a DepDisk.

3.3.4 Taking Control

After the virtual machine image has been transferred to the volunteer machine via the host BOINC client; an operation that would only take 3 minutes assuming that the current average UK bandwidth of 9 Mbps [28] applies; it must be unpacked, configured and started; a process that is performed both by the instantiation script downloaded in step (2) of Figure 3.1 and the V-BOINC client. The instantiation script simply:

- Decompresses the virtual machine image tar file.
- Signals the V-BOINC client to take control of the instantiation process.

The V-BOINC client is then responsible for the continued operation of both the virtual machine and the job executing on it. Afterwards the V-BOINC client must:

- Register the virtual machine image with VirtualBox.
- Create a fresh VDI or attach a pre-created DepDisk to the virtual machine.
- Start the virtual machine image.
- Take periodic checkpoints once the virtual machine is running.
- Wait for the virtual machine process to finish. This firstly shows to the volunteer user that the virtual machine process is still running if they use the BOINC Manager and that any virtual machine errors are caught during execution, which can then be uploaded to the server for debugging.

Once the virtual machine process is running, further complexities are introduced as a second BOINC client located on the virtual machine needs to be controlled from the host to execute typical BOINC commands, such as requesting tasks and uploading results; this is performed by using the *boinccmd* command line tool through the V-BOINC Client GUI. Figure 3.2 shows how the V-BOINC client GUI must interact with both BOINC clients and the VirtualBox API.

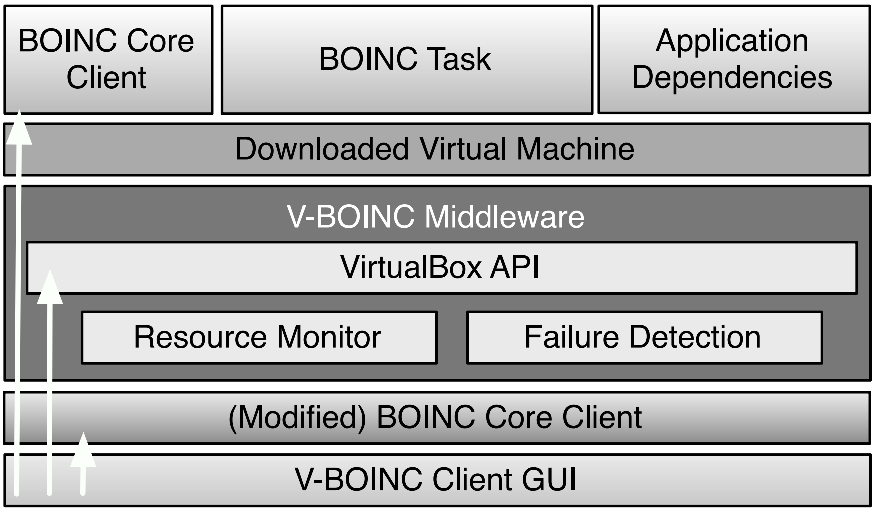


Figure 3.2: V-BOINC Volunteer Host Components

The V-BOINC client GUI provides a similar interface to that of the BOINC Manager, offering options to control either BOINC client's state to either running, suspended and halted via the *boinccmd* component. For example, if a volunteer user wishes to suspend a job running on the virtual machine, one has to use the *suspend* directive via the *boinccmd* tool on the virtual guest. BOINC offers other command options such as: *reset*, *detach*, *update*, *resume*, *nomorework* and *allowmorework*. These commands must be passed to the V-BOINC Middleware component which wraps them in a VirtualBox API method call to the *guestcontrol* function and executes them on the virtual machine; the virtual machine has Guest Additions installed to allow this.

These commands will control a BOINC job's execution within the virtual machine process, however controlling the virtual machine itself is more complex as the host BOINC Client cannot (easily) control separate non-BOINC processes. For example, the above *boinccmd suspend* command would not suspend the virtual machine process if executed locally on the host. Commands such as these must be performed via the VirtualBox API by calling the *controlvm* component. Additionally, the Middleware component also provides resource monitoring and virtual machine failure detection to inform the user in real time, the current state of V-BOINC.

3.3.5 Checkpointing and Recovery

Currently, BOINC project developers must ensure application-level checkpointing mechanisms are in place to allow an application to continue to run in the face of volunteer host failures or termination. When the volunteer host returns to a steady state after such an event occurs, the BOINC client restores the application from the last checkpoint. In order to replace the requirement of application-level checkpointing implemented by BOINC application developers, V-BOINC implements periodic virtual machine checkpointing, with the interval between checkpoints chosen by the volunteer user.

After recovering from volunteer host's termination or failure, the V-BOINC client instead restores the virtual machine that was previously executing the application, instead of restoring the application itself. This not only makes the task of application development easier for BOINC project developers, but it also allows volunteer users of the V-BOINC client to take checkpoints when they wish. Therefore, the project developer and/or research scientist can be confident this functionality is an improvement on the existing application checkpointing mechanisms employed by BOINC.

Periodically the V-BOINC client will make a call to the *snapshot* function of the VirtualBox API located in the V-BOINC Middleware (see Figure 3.2). By executing this command for a particular virtual machine on a volunteer host, VirtualBox will pause the virtual machine and take the checkpoint. The appropriate checkpoint files in placed in the *Snapshots* folder located in the virtual machine's configuration directory; a folder containing the virtual machine settings and the VDI images of both the virtual machine and DepDisk. The files created when checkpointing via VirtualBox are:

- A copy of the virtual machine settings. These settings include the hardware configuration, such as the memory allocated to the machine, as well as any attached disks.

- The current state of all VDI's attached to the virtual machine. VirtualBox implements this by storing differencing images; images that store all write operations after a checkpoint is taken.

- The current state of memory if a checkpoint is taken while the virtual machine is running. This memory state file can be quite large — up to the memory size allocated to the machine — and is dependent on the application memory usage. Allocating less memory, limits the size of the memory dump file but reduces application performance for those dependent on memory.

To restore a checkpoint, the correct differencing image is activated and the current checkpoint/virtual machine state is deactivated. To reduce the storage space consumed on the host, previous stale checkpoint files that are not required are deleted by V-BOINC; for example the memory state file of all previous checkpoints.

3.4 Experiments and Results

We now outline the experiments performed to firstly show the achievable resource performance of V-BOINC when compared to regular BOINC and secondly, to outline our use case showing that the V-BOINC platform can be used for computations requiring dependencies. Thirdly we show what affect periodic system-level checkpointing has on the valuable storage space reserved for BOINC jobs and on the BOINC job itself.

All experiments were performed on a Dell OptiPlex 790 host that has two Intel i3-2100 Core 3.10 GHz processors and 3.8 GB of memory. By default, the V-BOINC virtual machine is set to use the hardware-assisted virtualization instruction sets VT-x/AMD-V and the default values of using 1 GB of RAM and 1 processor are increased to the maximum VirtualBox allows; 2 processors and 2.9 GB of RAM.

3.4.1 BOINC vs V-BOINC

To evaluate V-BOINC, we measured the performance of V-BOINC when compared to regular BOINC. This was performed by running a series of benchmarks and a use case application and collecting their execution times.

3.4.1.1 Benchmark Performance

We used six benchmarks each with different resource usage demands to demonstrate the performance of a range of workloads. Each benchmark was executed ten times in a variety of configurations described below. Upon completion, each benchmark would output its own wallclock execution time which were then stored for analysis. The average execution times of each benchmark were then calculated and plotted in Figure 3.3. We display 95% confidence intervals to show that in most cases, the true mean will lie within the specified range.

Primes calculated the first 300 prime numbers while **CreateGB** created and wrote to a file of 5 GB. **CPU, Memory, I/O** and **Disk** strained each of the resources up

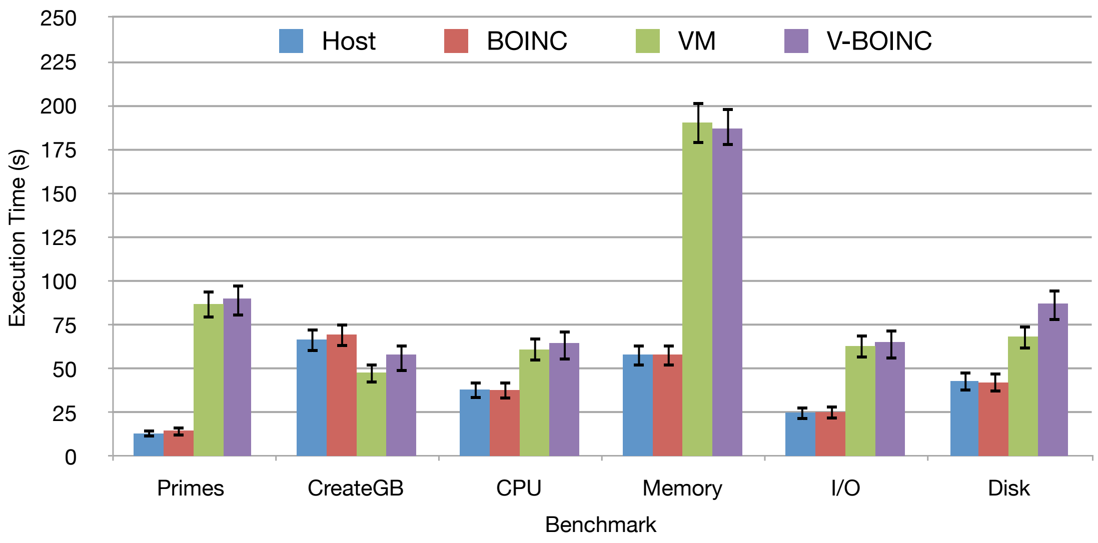


Figure 3.3: V-BOINC Benchmark Execution Times

to a specified number of iterations via a single *stress* process. Note that the *Memory* benchmarks allocates 2.5 GB of memory, e.g. *–vm-bytes 2560M*; details of these benchmarks can be found in Section 2.6 of Chapter 2. Each benchmark is then run over four different platform configurations:

1. execution just on the *Host* without the use of BOINC.
2. execution on the *Host* using BOINC.
3. execution just on the VirtualBox virtual machine without the use of V-BOINC.
4. execution on the V-BOINC virtual machine using V-BOINC.

Figure 3.3 shows the execution times obtained by running the benchmarks in each of the computational environments above. Firstly we see that the overhead of BOINC is negligible when comparing cases (1) and (2). Secondly and most importantly, we see that in most cases V-BOINC is slower than traditional BOINC with the exception of the CreateGB benchmark. Thirdly we see that the implementation of V-BOINC introduces little overhead when comparing cases (3) and (4). This shows that the performance difference between BOINC and V-BOINC is introduced by VirtualBox and not the implementation of V-BOINC.

Although the performance overhead of virtualization is often quoted as being up to 35% when executing CPU-bound applications [91], the performance overhead of our CPU benchmark is almost double. This is expected as overheads depend on a number of factors such as the virtualization technology chosen, the application, the configuration of the virtual environment, the available resources to the virtual machine and the physical hardware of the host.

This slowdown is caused by many factors relating to the virtual machine settings and hypervisor. When a virtual machine image is registered with VirtualBox, one must specify the memory, number of CPU's to use as well as a CPU execution cap, i.e only use 90% of the processor for example. However because the virtual machine is not able to use the full amount of memory and processing power available to the host machine, it is predictable that V-BOINC would perform slower; as only 2.9 GB of RAM could be allocated to the virtual machine, the memory-intensive benchmarks performed much slower. This is a problem facing full and para-virtualization technologies; only bare-mental hypervisors such as Xen may be able to overcome such problems.

Our memory benchmark execution time above uses 2.5 GB of memory; approximately 66.9% and 85.2% of the total available host and virtual machine memory respectively. If we normalize the percentage of memory used to 66.9% for each host, the execution time difference reduces from 190 seconds to approximately 160 seconds, showing the true virtualization overhead. However, the remaining memory intensive benchmark CreateGB shows that not all applications may run slower when using virtualization and this is dependent on the internal components of the application. We can only assume that the hypervisor's caching strategy is better than that of the underlying system as both the versions of *dd* are equal on the virtual machine and underlying host.

Similar to the memory deficit, the processing power available to the virtual machine is also lower than the total available to the underlying host. This is caused by the resources used by processes running and supporting the hypervisor on top of those running the OS. Hence the performance differences between host and virtual machine executions can be partly attributed to the maximum settings VirtualBox allows for any particular virtual machine but also the performance of VirtualBox itself, where others have found the performance difference much slower than execution upon the host [102, 223, 91].

3.4.1.2 Case Study: SPRINT-R

To illustrate that V-BOINC can not only execute standalone applications and to also show the performance achieved for a real use-case application, we execute the Simple Parallel R INTerface (SPRINT) [178]; as previously mentioned in Section 2.6.5 of Chapter 2, SPRINT has MPI and the statistical package R as dependencies.

For our experiment, we ran SPRINT's *pcor* function that performed parallel correlation on a randomly generated data set with 11,000 genes (rows) and 321 samples (columns) using two processes. Upon completion, each function would output its own wallclock execution time, which were then stored for analysis. The function was executed five times and the average execution times of each benchmark were then calculated and plotted in Figure 3.4. We display 95% confidence intervals to show that in most cases, the true mean will lie within the specified range. Again we provide a comparison between running the application via a variety of configurations, i.e Host, BOINC, VM and V-BOINC.

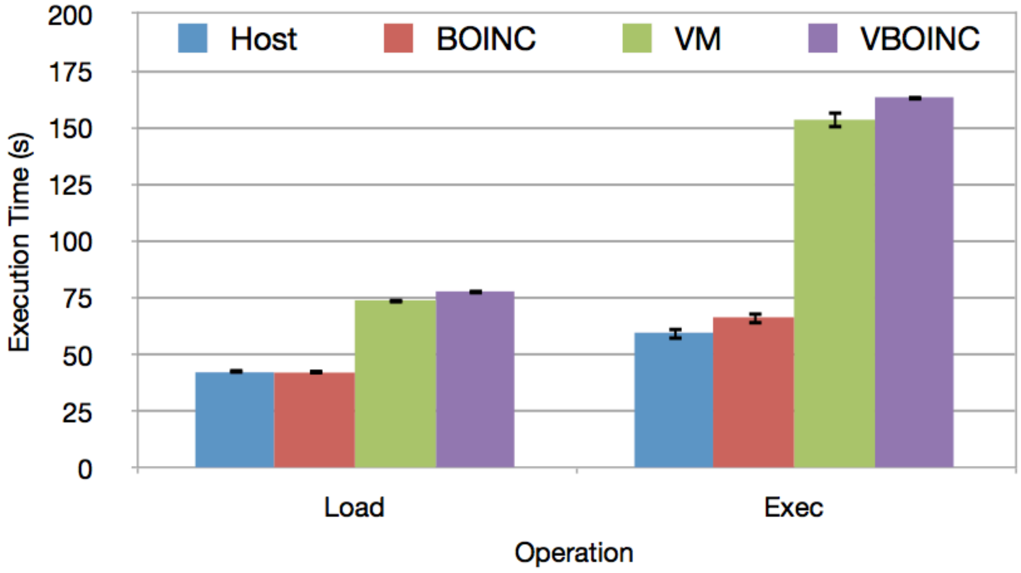


Figure 3.4: SPRINT Data Load and Execution Times

Figure 3.4 depicts three results similar to those outlined in Figure 3.3, however in order for *pcor* to perform the analysis, data must be loaded (Load) into R and then executed (Exec). We see that running SPRINT on BOINC shows little or no overhead when compared to running SPRINT on the host itself. The overhead of the V-BOINC implementation is also minimal where little difference can be seen when running SPRINT via the cases (3) and (4) above. Most importantly, we see the performance difference between V-BOINC and regular BOINC. This is caused by the overhead of virtualization, as shown by the green columns, therefore increasing the time for loading and execution to approximately double and triple respectively in these cases; a fact that must be accepted when using virtualization to solve other problems.

3.4.2 The Effect of Checkpointing

To enable BOINC project developers to omit application-level checkpointing from their code, V-BOINC offers periodic checkpointing. However, because the total disk capacity BOINC is permitted to use is potentially limited by the volunteer user's specified preferences, storage space is extremely valuable. To determine the likely storage space consumed by our system-level checkpointing approach, we executed a series of benchmarks representing different workloads while taking checkpoints at one minute intervals over a ten minute period.

We recorded the time to take a checkpoint, the size of the memory dump file and the size of the resulting differencing images of the DepDisk and virtual machine. The time to take the checkpoint was measured by executing the *snapshot* function of the VirtualBox API in conjunction with the Unix-based function *time*. The differencing image sizes were obtained by using a Java program we created to list file sizes. Each benchmark was performed five times on the V-BOINC virtual machine with an attached 8 GB DDI DepDisk containing experiment files and the necessary dependencies for SPRINT. The average checkpoint capture times and checkpoint file sizes were then calculated and the results are shown in Table 3.2.

Benchmark	Snapshot Time (s)	Memory Size (MB)	DepDisk Snapshot Size (KB)	VM Snapshot Size (KB)
CPU	1.1779	86.9	36	8
Memory	1.7142	56.76	36	8
I/O	0.9425	43.57	36	8
Disk	24.6023	1126.4	54374.4	8
Primes	1.2153	98.1	36	8
SPRINT	31.4665	2334.72	36	8

Table 3.2: Snapshot Sizes and Times per Benchmark

In four of the six resource intensive benchmarks (CPU, Memory, I/O and Primes), the average snapshot time is approximately one second. In these cases, the memory dump file size is lower than 100 MB and the differencing DepDisk and virtual machine images also remain small at 36 KB and 8 KB respectively; the lowest possible snapshot sizes for these two disks in this configuration. This shows that the differencing images are not written to during execution as only CPU, memory and I/O resources were used.

The remaining Disk and SPRINT benchmarks show different results where checkpoint times and memory dump file sizes are larger. This is caused by a large amount of memory consumed in both cases and a large amount of writes to disk in the former. In these cases, the largest memory dump file recorded was 2.28 GB using SPRINT and 1.1 GB using the disk-intensive benchmark; this benchmark also has the largest DepDisk snapshot VDI size of approximately 53 MB.

These results show that applications that do not write to disk, or perform lots of memory operations (e.g cache writes, etc), are unlikely to consume large amounts of storage space on the volunteer host when periodic checkpoints are taken. However applications that intensively perform memory or disk operations are likely to produce larger memory dump and checkpoint files. This is reassuring as typically BOINC applications tend to be CPU intensive operating over little data (e.g SETI@Home uses about 10MB per host [50]) therefore the checkpointing process should be quick and consume very little storage space.

3.4.3 The V-BOINC Server

One common problem of running a BOINC server is the difficulty of initially installing the server. This is due to the complex tasks a BOINC server administrator must perform, as well as the lack of documentation on such procedures [66, 187]. We provide documentation [164] on how to install a regular BOINC server to make the installation process easier for BOINC administrators as well as increase the end-user uptake of BOINC.

By extending a regular BOINC server in order to create a V-BOINC server, we have naturally made the installation process more difficult due to the incorporation of additional functionalities and complexities. To solve this problem, we have created a deployment script named *configure* that automatically performs all of the operations to successfully install the server.

For example, the script creates the V-BOINC project, copies pre-created files to the appropriate locations (e.g. the virtual machine to the BOINC *download* folder; see Section 2.4.3 of Chapter 2), configures the BOINC daemons and modifies permissions. This process usually takes one minute to complete however this may take longer depending on how long the BOINC project takes generating encryption keys. Afterwards, the V-BOINC server is ready to serve virtual machines to volunteer hosts.

Similar to the performance degradation caused by virtualization on the volunteer host, we expect the performance of the V-BOINC server to be less than that of a regular BOINC project server deployed on the same host. Previous research has shown that a BOINC server hosted on a single inexpensive computer can distribute up to 8.8 million tasks per day with the CPU and network bandwidth being the main bottlenecks [51].

In the case of V-BOINC, we expect that the number of tasks per day the server can distribute will be significantly lower than that of a regular BOINC server, with the network bandwidth being the major bottleneck when volunteer BOINC clients request a virtual machine image to be downloaded on their machine. We investigate this further is Chapter 6 when we evaluate the performance of our *ad hoc* cloud prototype.

3.5 Summary

V-BOINC is a tool providing solutions to the drawbacks of regular BOINC by allowing project developers to port their application only to the V-BOINC virtual machine and omit application-level checkpointing from their code. Developers with applications that have dependencies can easily utilize V-BOINC, where users of regular BOINC cannot (easily) run such applications. Finally end user worries relating to security and untrustworthy applications are also solved via the sandbox environment of virtual machines. Note that V-BOINC does not currently deal with providing correct credit to BOINC users nor does it accurately adhere to volunteer user-based preferences.

The design and implementation of V-BOINC plays a major role of how regular BOINC applications and those with dependencies can easily be run upon V-BOINC. In the former case, our inner virtual machine BOINC client allows regular BOINC applications to be run in the virtual presence without modification and furthermore, four stage transfers between the virtual machine and host do not occur compared with other approaches [102, 160].

Applications with dependencies are able to run using V-BOINC due to its attachable disk mechanism that automatically mounts dependencies. Furthermore system-level checkpointing is available in order to allow BOINC project developers to omit application checkpointing code from their BOINC application. In the same way regular BOINC restores applications when volunteer hosts return to an available and usable state, V-BOINC instead restores the previous virtual machine checkpoint allowing the BOINC application to continue executing.

We have also shown how the performance of V-BOINC compares with regular BOINC and how the implementation of V-BOINC introduces a negligible overhead when compared to VirtualBox. As expected, the performance of regular BOINC is better than that of V-BOINC due to the virtual machine overhead. The actual overhead caused by the implementation of V-BOINC is however negligible when compared with running the same application on a standalone virtual machine. Therefore one must weigh up the advantages of V-BOINC compared to the increased performance of traditional BOINC and whether the performance cost from virtualization is acceptable for volunteer computing. Investigating this with real volunteer users, application communities and different hypervisors, such as QEMU/KVM and VWware Player, are worthy of extensive future investigation.

This chapter has shown how it is possible to successfully integrate volunteer computing and virtualization. Many users within the volunteer community have taken advantage of V-BOINC; approximately 200 users have downloaded V-BOINC since its introduction. A large subset of these actively used our Amazon EC2 V-BOINC on-line service that acted as a V-BOINC server by distributing virtual machines to the general public for those who wished to run BOINC applications in a secure environment. Currently this service is unavailable due to the lack of credits available from our previously acquired Amazon EC2 research grant. We aim to restore this service in the future; information of how V-BOINC can be downloaded and used can be found at [39].

By developing V-BOINC, we have not only solved many of the issues surrounding volunteer computing infrastructures, but most importantly, we have also implemented a platform that can be developed further to ultimately transform it into an *ad hoc* cloud computing platform. We discuss how this is achieved in the following Chapter.

Chapter 4

From Volunteer to *ad hoc* Cloud Computing

4.1 Introduction

In this chapter, we outline in detail how each of our contributions are implemented that relate to transforming our virtualized volunteer computing infrastructure into an *ad hoc* cloud computing platform. We first give an overview of the related research in the field of *ad hoc* cloud computing by outlining two distinct and important studies that introduce the topic and provide a foundation for similar research, including our own, to build on. We also discuss similar studies that aim to transform volunteer computing models into cloud computing infrastructures.

This is followed by outlining the architecture and implementation of our *ad hoc* cloud prototype as well as how a volunteer infrastructure, normally controlled by distributed volunteer hosts, is converted into a centrally controlled infrastructure. We then describe how an *ad hoc* cloud user is able to submit a job to BOINC running on a modified V-BOINC server and the processes involved when scheduling a virtual machine to a near-optimal *ad hoc* host.

An in-depth overview of the key enabler of *ad hoc* cloud computing is then described, i.e. introducing reliability into the unreliable infrastructure V-BOINC. This involves periodic checkpointing, distribution, scheduling and restoration. Penultimately, we outline potential measures to reduce the level of interference an *ad hoc* cloud user's task has on the *ad hoc* host and finally, we describe the effort required to install and operate the components of the *ad hoc* cloud.

4.2 Related Work

The topic of *ad hoc* computing encompasses a variety a topics ranging from *ad hoc* networks and protocols [151] to more recently, *ad hoc* cloud computing. Up until 2009, research primarily involved the former, however the concept of creating a distributed infrastructure over non-exclusive and sporadically available hosts and devices has grown in popularity. This section will focus solely on the latter.

Firstly we outline the two most important studies of *ad hoc* and volunteer cloud computing as well as the current state of research in this area. We omit low-level network and protocol details as this is out of the scope of this thesis. We do however assume that by incorporating the outcomes of *ad hoc* network and protocol research, our *ad hoc* cloud computing platform will benefit both in terms of performance and relevance to the potential user community.

4.2.1 The Two Pillars

The concept of *ad hoc* cloud computing was first introduced and discussed by Kirby *et al.* [141]. Those authors propose the concept of the *ad hoc* cloud within enterprise settings to harness unused resources to improve overall utilization, reduce net energy consumption and allow enterprises to take advantage of operating their own in-house cloud.

Their work focusses on the major research and implementation challenges to realize the *ad hoc* cloud computing concept and describes one approach on how to do so. The primary challenges relate to coping with sporadically available hosts and minimizing the impact on non-cloud processes to an acceptable level; these are analogous to the contributions described in Chapter 1. Additionally the authors list further challenges, for example:

- What are the architectural requirements for an *ad hoc* cloud infrastructure?
- How can the membership of the set of machines in an *ad hoc* cloud be controlled?
- In which situations should an *ad hoc* cloud be scaled out or contracted?
- To what extent can planning decisions be improved using measurements and predictions of previous, current and future workloads?

These research challenges are analogous to our secondary contributions listed in Chapter 1. With comparable challenges to be solved, our proposed approach has some similarities to that proposed by Kirby *et al.* The approach taken in this thesis also involves creating groups of hosts called *cloudlets* that contain the necessary environment or services, termed *cloud elements*, for a task to run on any abstract class of host, e.g. virtual machine, Java virtual machine etc.

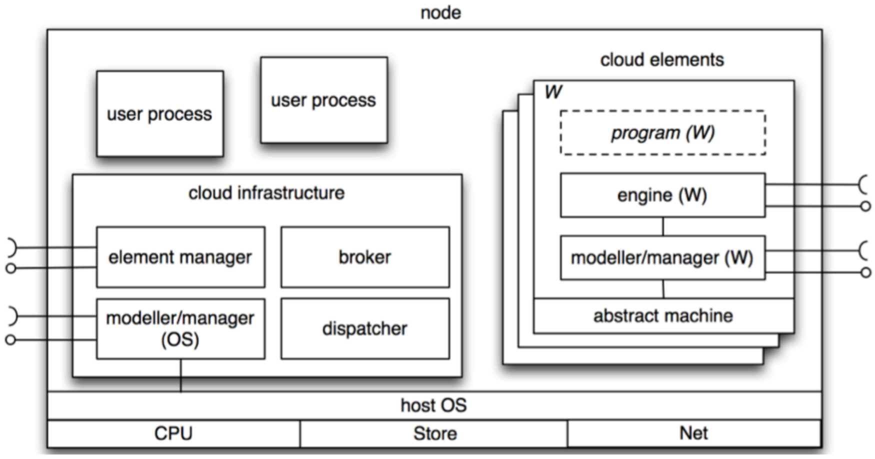


Figure 4.1: Kirby *et al.* Node Structure [141].

Figure 4.1 depicts their proposed architecture, where each cloud element contains a modeller/manager component that is installed on the abstract machine. This interacts with the host OS to monitor the host resource usage and performance. Its counterpart installed on the host itself, monitors the effect of the cloud element on the host. The *Broker* and *Dispatcher* pattern deploys tasks to appropriate nodes based on program characteristics and specified Quality of Service (QoS) targets. These are components which we believe are fundamental to the *ad hoc* cloud model and as such, are adopted within our methodology.

The work performed by Kirby *et al.* has therefore provided our research with a foundation on which to build. However, with the exception of these similarities, the differences between the proposed approaches are substantial; namely scheduling policies, QoS guarantees and how to introduce reliability into an unreliable infrastructure. We describe these differences further while describing our own approach later in this chapter.

Although the term *ad hoc* cloud computing was proposed by Kirby *et al.* in 2010, research into creating clouds from voluntary resources had already been started earlier by Chandra *et al.* Like Kirby *et al.*, Chandra *et al.* also outline the challenges of creating clouds, termed Nebulas, from unreliable and sporadically available resources. Chandra *et al.* also outline a possible approach to realize the concept [80, 81]. The differences between the ideas of Kirby *et al.* and Chandra *et al.* are minimal as both aim to solve the same class of problems but in different ways. These studies also outline the importance of taking features from both volunteer and cloud computing to enable a dynamic infrastructure, such as an *ad hoc* cloud or Nebula, to be developed.

Chandra *et al.* outline the problems with current cloud models, such as applications may not suit the dedicated cloud model or end-users may not or cannot take advantage of such platforms; these are similar to the challenges we identified in Chapter 1 and aim to solve. For example, an application may not require strong performance guarantees or be experimental in nature. Furthermore, the cloud may be too expensive to migrate to, especially in cases when an application relies on large amounts of distributed data; preferably the computation should be moved to these data sets and not vice versa [112]. A Nebula and an *ad hoc* cloud offer an alternative computational platform to solve such problems, however Chandra *et al.* identify further challenges to be solved, namely how to:

- maintain the state of cloud job, volunteer nodes and concurrent user requests,
- deal with a high level of heterogeneity between volunteer hosts,
- provide robustness to localized failures and be self-recovering,
- provide easy management for both cloud users and those who donate resources,
- provide protection for the volunteer resources against malicious actions,
- calculate the performance tradeoff between computation and data placement.

With the exception of the latter, our approach offers a solution to each of these challenges. The approach of Chandra *et al.* offers potential solutions to a few of these points, namely how to handle host heterogeneity, failures and data-compute locality. As heterogeneity has a direct impact on the performance of an application, the authors note that resource scheduling must occur to deploy the correct application onto a set of suitable resources. For example, larger applications can be deployed on faster hosts to mitigate the effect the slowest host has on the overall performance; these hosts must also be selected based on reliability.

Failure handing is an important component of operating any task over an unreliable infrastructure and Chandra *et al.* propose two solutions: employ replication by executing a job on multiple hosts concurrently or perform aggressive application or virtual machine checkpointing and restore these checkpoints upon any host failures; the latter is a feature we have incorporated into our *ad hoc* cloud platform. We do not employ task redundancy as we imagine that the *ad hoc* cloud will typically run on small-medium scale infrastructures where the number of hosts are limited. Therefore by employing task redundancy, the number of available *ad hoc* hosts that can execute new cloud jobs is reduced by a factor of the set redundancy value.

To minimize the potential performance degradation caused by sub-optimal data and compute locations, Chandra *et al.* proposed either to calculate the network distance between these entities or estimate network performance. Although we do not consider data and compute locality, by incorporating these features in our *ad hoc* cloud platform as part of future work, we would expect further performance improvements.

The Nebula concept was then partially-realized by Sundarrajan *et al.* who discuss their early experiences with a prototype of the system [216]. Those authors re-iterate current cloud unsuitability when executing dispersed data-intensive applications on centralized infrastructures. They therefore use a data-intensive blog-analysis use case to test their prototype distributed over the global research testbed PlanetLab [32] to aid in a comparison. In order to analyse the blog data set on PlanetLab, the data is distributed over data nodes and in order to analyse the data, execution nodes request the data of a particular data node under the instruction of a centrally managed master node. This architecture is shown in Figure 4.2.

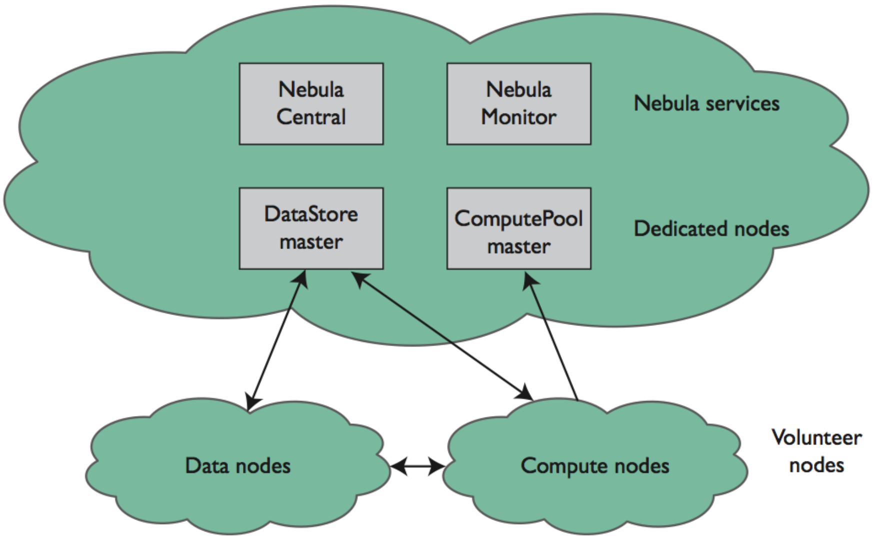


Figure 4.2: Nebula System Architecture [81]

The Nebula master contains the central node; a front-end interface allowing users to join the cloud and administrators to create and upload applications as well as monitor and manage the system. The DataStore and ComputePool components manage a number of volunteer nodes that offer storage and compute respectively.

When analysing an increasing number of blogs, Nebula achieved time savings of 53% on average and data transfer savings when compared to a running on a centralized cloud emulator. This analysis assumed no host failures occurred, however, it has been shown that Nebula is able to outperform the centralized cloud model when a small number of failures occur. Nebula implements a reactive approach to fault tolerance by using task replication and task re-execution.

4.2.2 Volunteer Systems and Cloud Computing

The previous work outlined has highlighted the research and implementation challenges surrounding *ad hoc* cloud computing as well as initial results showing the promise of such a paradigm.

Andrezejak *et al.* explore the idea of allowing a web-service provider, for example DropBox, to use resources from the non-dedicated hosts they serve [54]. This aims to reduce the number of dedicated servers a web-service provider owns or provisions from cloud services, in turn reducing costs. The authors restrict the use of their proposed approach to the web service domain, where non-dedicated hosts are primarily used for their processing capabilities. This is due to the limited bandwidth and sporadic availability of non-dedicated hosts.

Andrezejak *et al.* identify that the availability of non-dedicated resources is a primary problem, however they assume that a web service has fault tolerance and redundancy mechanisms in place to cope with highly volatile non-dedicated hosts; we develop and present our solution to this problem later in this chapter. Those authors focus on a number of other research challenges.

Firstly Andrezejak *et al.* review techniques for predicting short-term availability of non-dedicated hosts in order to help predict their long-term availability. Secondly, they outline a method to identify optimal combinations of dedicated and non-dedicated hosts to reduce costs or reduce the number of migrations performed when non-dedicated hosts fail. Processes previously running on a failed non-dedicated host are restored by migrating the process's data to another non-dedicated host. Unlike our *ad hoc* cloud platform, this is performed periodically and not when the failure occurs;

the details of this migration and restoration process are not described. Finally, the authors investigate the trade-offs between using a greater number of dedicated hosts or non-dedicated hosts.

In order for a web-service provider to utilise resources from a non-dedicated host, the host's availability is first calculated based on monitoring data collected over several weeks. Well known Machine Learning predictors such as Last Value, Naïve Bayes and Gaussian predictions are used to group the hosts according to their predicted short-term availability; a non-dedicated host is however deemed to be available if 100% of its CPU is free, which in many cases will not occur.

Non-dedicated hosts in a group with the lowest rank (i.e. those that are likely to be available) are used first by the web-service provider before lower groups are exploited in descending order; the authors consider the case where non-dedicated hosts are ranked in groups ranked from 1 to 4. By combining simple prediction mechanisms with host ranking, Andrezejak *et al.* claim this allows accurate long-term predictions to be made. Their results show that their average highest and lowest error rates of long-term availability prediction are approximately 21% and 14% respectively. Furthermore, as the number of non-dedicated hosts increases, the probability of meeting availability guarantees decreases, and vice versa.

Also by increasing the level of data redundancy, the number of dedicated hosts required to meet availability guarantees decreases. Our implementation of the *ad hoc* cloud does not utilise redundancy but instead takes a reactive approach to deal with non-dedicated host failures. We assume that by incorporating task redundancy into the *ad hoc* cloud, the success rate of task completion will increase further.

Andrezejak *et al.* then propose an optimisation method to either reduce costs for the web-service provider or reduce the number of migrations performed. The authors assume a web-service provider's dedicated resources are served from a cloud provider such as Amazon EC2 and therefore incur costs of 10 US cents per hour for each dedicated host; this is comparable to a particular Amazon EC2 instance. Similarly data transferred between non-dedicated and dedicated hosts is charged at a rate of 10 US cents per GB.

Andrezejak *et al.* find that for a group of rank 1 non-dedicated hosts, the optimal number of dedicated hosts required to meet availability guarantees while also reducing costs is 25 dedicated hosts from a set of 55. For a group of rank 4 non-dedicated hosts, the total number of hosts a web service must use must increase to 62 in order to keep costs as low as possible. By only considering rank 1 non-dedicated hosts, 44 dedicated hosts from a possible set of 52 are required to minimise the number of migrations. In the case of rank 4 non-dedicated hosts, the migration rate is lowest when more dedicated hosts are used, in particular, 49 dedicated hosts from a set of 52.

As our concept of the *ad hoc* cloud only involves one dedicated host and a potentially unlimited number of non-dedicated volunteer hosts, Andrezejak *et al.* show that volunteer hosts have a great potential to perform tasks that are typically executed on dedicated hosts. Those authors' work presents an approach that complements our own, particularly regarding the calculation of a host's short and long-term availability as well as reducing the number of migrations between volunteer hosts. We detail our solutions to these problems as well as a method to handle volunteer host failures later in this chapter.

Mori *et al.* discuss their sophisticated *ad hoc* cloud computing environment, named SpACCE, that is tailored for application sharing and distributed collaboration [170]. Their idea is based on creating a cloud environment by offering services from an *ad hoc* server, called CollaboTray, that may at any time, migrate to another node in the network. An example service outlined is Microsoft Powerpoint. The server may migrate if the node currently hosting the server has an increase in utilization or will reduce the performance of the service delivered to the clients. If an application requires more capacity to execute effectively, other clients can be converted into servers to avoid the total server resource capacity from diminishing.

Due to the *ad hoc* nature of their project, our goals are similar; namely how to effectively co-exist with user processes, deal with dynamic hosts and the migration of components between hosts. Their results show that large performance latencies can occur if the server does not have 40% of the CPU available to use. This means that applications that are resource intensive will be unable to utilise CollaboTray. In order to migrate CollaboTray, it is first closed, its state is then transferred to another node and finally it is restarted; a similar process we use to migrate and restore virtual machines between hosts.

However CollaboTray does not use virtualization, hence the security of the system is questionable if the server is migrated to an untrustworthy node. There is also no concept of host reliability which will result in poor application performance if the server is migrated to an unreliable node. Our implementation of the *ad hoc* cloud provides solutions to the downfalls mentioned as well as additional features such as effective monitoring and scheduling.

Cunsolo *et al.* argue that cloud computing is a computational model directed towards businesses, therefore restricting its usefulness for scientific purposes [85]. Those authors propose an alternative to the data centre model where individual users are able to donate their resources to form a unified cloud. As this is similar to volunteer computing, they name this Cloud@Home. By merging volunteer and cloud computing, users may either offer their resources for free to an OpenCloud or buy and sell resources from a cloud called HybridCloud. These two cloud models are then able to exist independently, link with one another, or link to other public and private cloud computing platforms.

In their proposal, the authors identify that resource management, security, reliability and Quality of Service (QoS) are some of they key challenges to overcome. Resources are managed centrally and security is provided by virtualization, data encryption and secure transmission protocols. Reliability is however based on negotiations with volunteer users specifying their contribution; a volunteer host could however leave at any time and no mechanisms for recovery are proposed. In turn, no QoS guarantees could be made. Furthermore, the authors do not specify, among other things, the volunteer system to be used and how this could be transformed into a cloud platform.

Wu *et al.* create a private cloud based on BOINC for the purpose of executing parallel and distributed simulation tasks [221]. Much of this focus is on scheduling tasks to nodes within the system by using BOINC as a dispatcher according to the authors own load-balancing algorithms. Although no reference is made to how their architecture is in fact a cloud or how BOINC is part of their architecture, the authors do note that scheduling and infrastructure monitoring are important components within private clouds.

4.2.3 Mobile *ad hoc* Cloud Computing

Despite the relatively few successful studies of *ad hoc* cloud computing and the merging of volunteer and cloud computing, the field of mobile computing has shown more promise and has been popular since 2009 [186].

Mobile devices are well known to be ‘resource poor’ where the compute capacity, memory size and storage space is extremely limited [193, 101]. They also are limited by battery life and network connectivity. However, there are cases where offloading to another remote mobile device or computational platform is useful; for example, to render a high quality image when power is low. Most studies focus of whether it is feasible to execute applications within mobile device clouds and whether any performance gains can be achieved. There are generally varied success stories on the matter [101, 208] and benefits are perceived to be dependent on the application [146].

Applications that are suited to remote clusters or clouds typically cannot be offloaded effectively due to the high latencies of WANs [208, 101]. However some dispute these claims and show that offloading computation to Amazon EC2 is feasible as well as desirable for latency-tolerant applications [169]. To take advantage of locality, an independent group of mobile nodes, called *cloudlets*, are proposed to allow devices within a cloudlet, to offload tasks to other members [171, 208, 101, 193].

Satyanarayanan *et al.* outline their proposal for a mobile cloud computing environment that closely matches our own approach with non-mobile devices [193]. The authors propose to use VirtualBox virtual machines upon mobile devices; some studies however find that VM-based approaches are ineffective [208]. Satyanarayanan *et al.* focus a significant part of their research on how to minimize virtual machine sizes and how to transfer them effectively between devices.

These authors' approach of hosting pre-configured virtual machines on devices and only transferring *overlays* (checkpoints) over the network matches ours. Satyanarayanan *et al.* do not consider scheduling, monitoring or how to deal with mobile churn and task restoration. However, it is encouraging that their results show that the transmission of virtual machine overlays to offload computation between mobile devices performs well; we expect greater performance on a 'resource-rich' platform.

4.3 Architecture of the *ad hoc* Cloud

In this section, we first outline an initial conceptual architecture of the *ad hoc* cloud describing the high-level components required to create such a platform. This is then followed by a detailed architecture of our implemented prototype.

4.3.1 Conceptual Architecture

The conceptual architecture of any *ad hoc* cloud should be primarily composed of components taken from the *ad hoc* cloud computing founding principles previously introduced. We believe an *ad hoc* cloud should contain core elements from the following high-level components shown in Figure 4.3. Due to the challenges and complexities an *ad hoc* cloud poses, the *ad hoc* cloud should be based on:

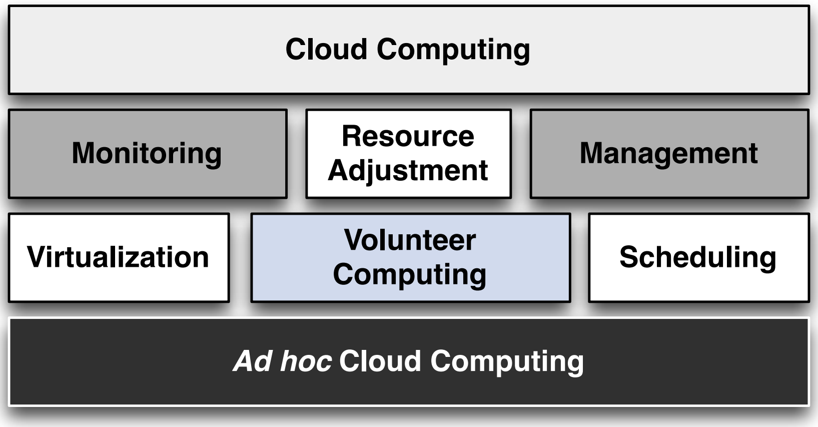


Figure 4.3: Conceptual Architecture of the *ad hoc* Cloud

- *Volunteer computing*: the foundation of an *ad hoc* cloud should rely on a component that is able to control and coordinate a potentially large set of distributed, heterogeneous and unpredictable sporadically available resources. Furthermore, the *ad hoc* cloud should rely on a component that realizes the importance of the host user having access to their resources when needed. Volunteer computing infrastructures, in particular BOINC, offer these functionalities and therefore we chose BOINC as the core of our *ad hoc* cloud computing prototype.

- *Virtualization*: the potentially untrustworthy nature of volunteer resources requires that host resources and processes as well as a cloud job running on an *ad hoc* host are protected. We employ virtualization to overcome this challenge and to allow easy management of the general infrastructure; the importance of virtualization was shown in the previous chapter describing V-BOINC. In order to extend a volunteer infrastructure and provide a reliable environment for cloud jobs to continuously operate in the face of host failure or churn, the features of virtualization such as checkpointing must be exploited.

- *Scheduling*: due to the unreliable nature of an *ad hoc* cloud as well as the need for cloud jobs to execute as quickly as possible, additional scheduling methods must be created to take into account host availability, resource specification, resource load and reliability.

- *Monitoring*: additional scheduling mechanisms require additional monitoring mechanisms, above those that are provided by volunteer computing infrastructures, to provide data for such scheduling decisions. Advanced monitoring is also required to enable cloudlet-based monitoring allowing cloudlet resources to be expanded or contracted dependent on administrator-defined goals.

- *Management*: infrastructure management allows cloudlet-based host migrations to be performed as well as giving the administrator of the *ad hoc* cloud the capability to control hosts in the event of any problems or allow necessary tasks to be performed over a group of *ad hoc* hosts.

- *Resource adjustment:* the ability to minimize host process interference caused by cloud processes will determine the success and up-take of the *ad hoc* cloud computing paradigm. While we do not introduce such functionality in our prototype, the underlying virtualization technology or various open-source tools can be used to provide this feature.

- *Cloud computing*: the concepts from both public and commercial cloud platforms play important parts in defining the *ad hoc* cloud and as such there are many similarities between the models. For example, the *ad hoc* cloud operates as a PaaS cloud, permits multi-tenancy, must obtain resources on-demand, be easy to use, strive to provide an adequate level of QoS, etc. Developing an *ad hoc* cloud which is similar to popular cloud platforms will play a key role in the success of *ad hoc* cloud computing.

We now discuss how each of these components are implemented and incorporated into our *ad hoc* cloud computing prototype.

4.3.2 Prototype Architecture

In order to develop our *ad hoc* cloud computing platform, we have used our virtualized volunteer infrastructure V-BOINC as a foundation to build on and extend. Therefore we inherit many of the functionalities V-BOINC has to offer as well as an initial client-server architecture.

We now give an architectural overview of the *ad hoc* cloud while focussing on the differences when compared to the V-BOINC server and client components. We then outline how these components interact and give an overview of how the *ad hoc* cloud operates. These descriptions are then used in subsequent sections to describe in greater detail the implementation and features of the *ad hoc* cloud.

4.3.2.1 The *ad hoc* Server

The *ad hoc* server is an extension of the V-BOINC server previously described in Chapter 3. While an *ad hoc* and V-BOINC server share one primary purpose of distributing virtual machines to volunteer hosts, the *ad hoc* server is able perform more complex operations. Unlike a V-BOINC or regular BOINC server, the *ad hoc* server is able to:

- allow *ad hoc* cloud users to submit jobs to BOINC,
- schedule cloud jobs and virtual machine migrations to near-optimal *ad hoc* hosts based on host availability, specifications, resource load (i.e. the current utilization of a resource) and reliability,
- send instructions to *ad hoc* hosts for execution,
- monitor and manage the state of the system easily.

These additional functionalities help transform our V-BOINC infrastructure into one half of an *ad hoc* cloud computing platform. We are able to provide these functionalities by creating two BOINC projects: VM Service and Job Service. This is in contrast to our V-BOINC server where volunteer users are served from a single BOINC project named V-BOINC. The architecture of the *ad hoc* server is shown in Figure 4.4.

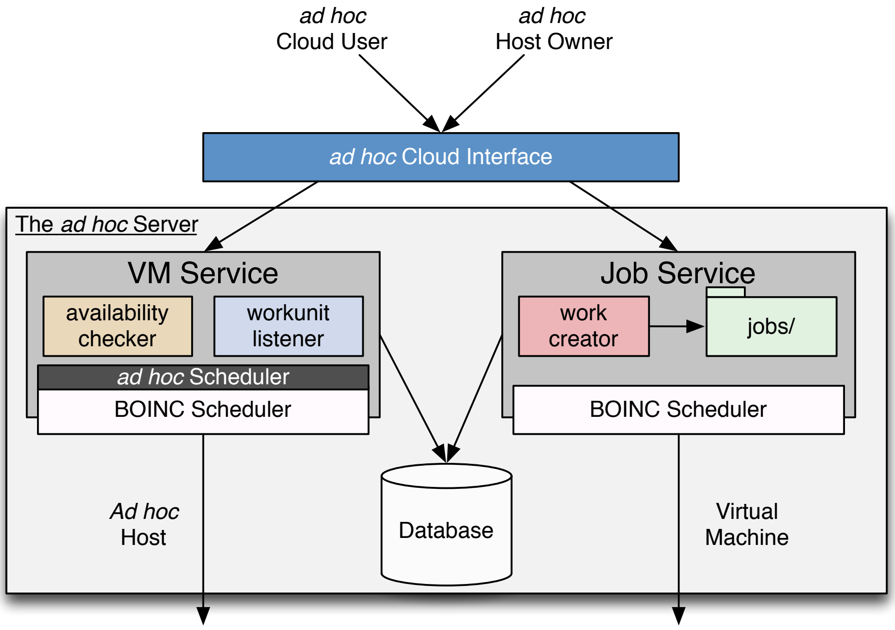


Figure 4.4: The *ad hoc* Cloud Server Architecture

The Job Service project has the purpose of receiving cloud jobs from *ad hoc* cloud users, via the *ad hoc* Cloud Interface, and registering these jobs with BOINC. Once registered, the Job Service has the task of informing the VM Service that the cloud job is ready to be executed on an *ad hoc* guest running on an *ad hoc* host. The *ad hoc* Cloud Interface also allows *ad hoc* cloud users and *ad hoc* server administrators to manage their respective BOINC accounts.

The VM Service project, conceptually similar to the V-BOINC project that runs on a V-BOINC server, has the task of distributing virtual machines to *ad hoc* hosts. Additionally, the VM Service project schedules jobs to near-optimal *ad hoc* hosts and virtual machine migrations, sends instructions to both *ad hoc* hosts and *ad hoc* guests and monitors and controls the entire system state. Ideally it would be beneficial for development and management purposes if both cloud jobs and virtual machines could be served from a single BOINC project however this is not possible as we need to distinguish between the two entities to allow the former to execute on the latter. Although the *ad hoc* server is substantially different to a regular BOINC server and offers a greater number of features, we have ensured that architectures of both have remained similar; a comparison can be made between Figures 4.4 and 2.8. We describe how the *ad hoc* server is able to offer the outlined features in greater detail later in the chapter.

4.3.2.2 The *ad hoc* Client

The *ad hoc* client is an extension of the V-BOINC client previously described in Chapter 3. While an *ad hoc* and V-BOINC client share the primary purpose of executing volunteer applications that run volunteer host virtual machines, the former has a greater number of responsibilities, in particularly to help provide a reliable environment for job execution. Unlike a regular BOINC or V-BOINC client, the *ad hoc* client is able to:

- receive instructions from the *ad hoc* server to be executed,
- periodically take checkpoints of the virtual machine,
- schedule and send checkpoints to a near-optimal number of *ad hoc* hosts,
- receive virtual machine checkpoints from other *ad hoc* hosts,
- restore virtual machine checkpoints sent from previously terminated or failed *ad hoc* hosts or guests,
- effectively monitor both *ad hoc* hosts and guests.

Due to the large number of features integrated into the V-BOINC client to create the *ad hoc* client, the architectures of both are significantly different with the latter being more complex. The architecture of the *ad hoc* client is shown in Figure 4.5.

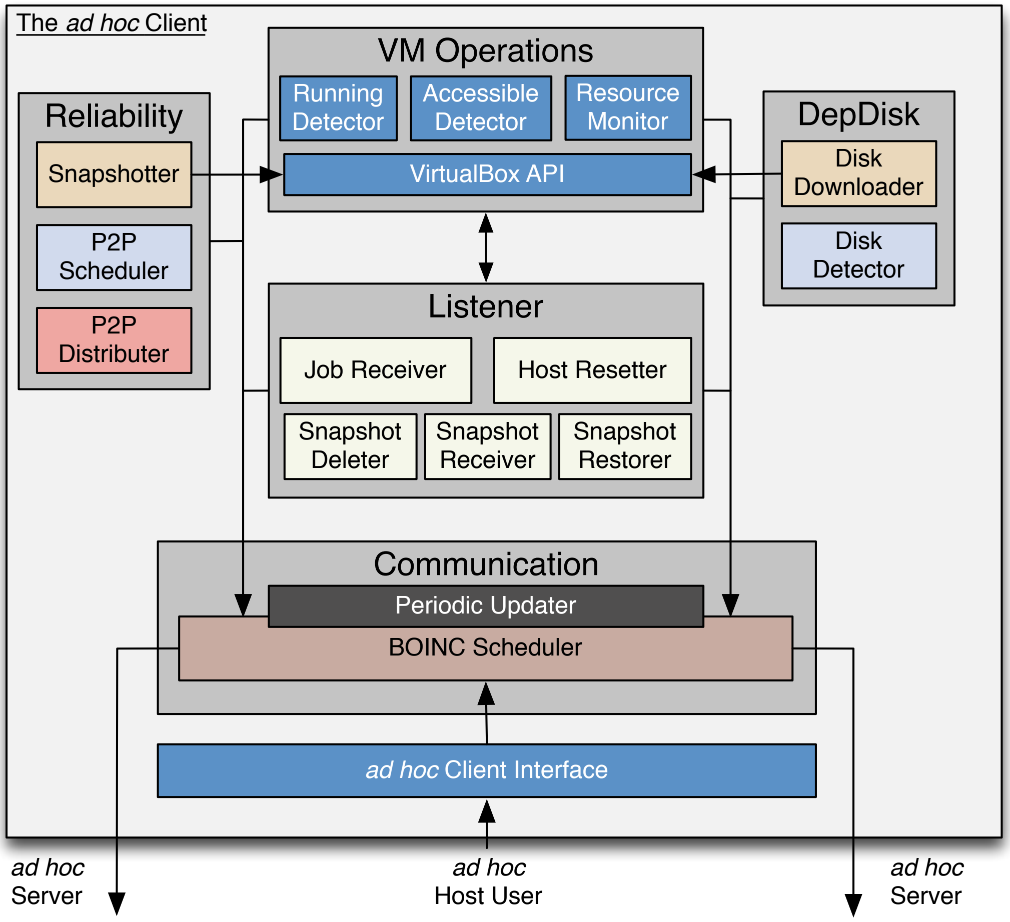


Figure 4.5: The *ad hoc* Cloud Client Architecture

We see that the *ad hoc* client is composed of six major components: the *ad hoc* Client Interface, Communication, Listener, VM Operations, DepDisk and Reliability. The *ad hoc* Client Interface provides a GUI, similar to the BOINC Manager (see Section 2.4.3 of Chapter 2), to control the *ad hoc* host's membership within the *ad hoc* cloud. The Communication component interacts with the *ad hoc* server while the Listener listens for any instructions sent from the server to be executed. This may include performing operations on the virtual machine via the VM Operations component which has the responsibility of dealing with all aspects related to the interactions between VirtualBox and the virtual machine.

The DepDisk component checks the *ad hoc* server for dependency disks and downloads the correct DepDisk for an application if it requires it. Finally and most importantly, the Reliability component ensures the continuity of cloud jobs by periodically taking virtual machine checkpoints and distributing these in a P2P fashion within a cloudlet; a set of connected *ad hoc* guests that provide a particular service or execution environment, i.e. those that possess the same application dependencies and DepDisk. We describe how the *ad hoc* client is able to offer the aforementioned features via the six major client components in greater detail later in the chapter.

4.3.3 Client-Server Interaction

The *ad hoc* server and client interact through the communication mechanisms BOINC already provides. However modifications have been to made to BOINC and V-BOINC allowing these *ad hoc* components to transfer customized data between them and the server. This is required to instruct an *ad hoc* client or to allow an *ad hoc* client to update the server on the current status of the virtual machine, for example.

BOINC implements client-server communication by exchanging XML messages which are then parsed locally allowing the receiving entity to determine the appropriate actions to subsequently take. Figure 4.6 shows an excerpt of a message sent from the client to the server. In this example, we see that a message is composed of many XML elements that are grouped according to the information they provide. In reality, BOINC server or client messages are composed of many information groups and messages are much larger in length; on average messages are approximately 10 KB [51].

Figure 4.6 shows that important information about the host is passed to the server. For example, the *authenticator* uniquely identifying the host, the *ad hoc* host's designated *hostid*, the version of BOINC running and the type of platform. User-based preferences specified on the host rather than via the BOINC server are also sent; for example, we see that the volunteer user restricts BOINC to use at most 90% of the available memory when the volunteer host is idle. A message also provides details of currently executing or completed jobs. For example, Figure 4.6 shows that the volunteer host was executing a V-BOINC job that completed in approximately 60 minutes.

```
<scheduler_request>
   <!-- Host and BOINC Information -->
   <authenticator>2_9fec55ffe011154092f0bde825</authenticator>
   <hostid>11</hostid>
   <core_client_major_version>7</core_client_major_version>
   <core_client_minor_version>0</core_client_minor_version>
   <platform_name>x86_64-pc-linux-gnu</platform_name>
   <!-- Local User Preferences -->
   <working_global_preferences>
      <global_preferences>
         <source_project>http://IPADDR/VBOINC/</source_project>
         <run_if_user_active>1</run_if_user_active>
         <ram_max_used_idle_pct>90.000000</ram_max_used_idle_pct>
      </global_preferences>
   </working_global_preferences>
   <!-- Details of the Executing Job -->
   <result>
      <name>VBOINC_1384477492_0_0</name>
      <final_elapsed_time>3600.236996</final_elapsed_time>
      <state>5</state>
   </result>
   <!-- Virtual Machine Elements  -->
   <vm_ip_addr>129.215.90.90</vm_ip_addr>
   <vm_has_job>true</vm_has_job>
   <vm_failure>false</vm_failure>
   <!-- Ad hoc Client Elements -->
   <cwd>/home/boincadm/BOINC</cwd>
   <received_snapshot_ip>129.215.92.92</received_snapshot_ip>
   <received_snapshot_id>13</received_snapshot_id>
   <restored_snapshot_ip>129.215.91.91</restored_snapshot_ip>
   <restored_snapshot_id>12</restored_snapshot_id>
   <deleted_snapshot_id>29</deleted_snapshot_id>
   <snapshots_to_hosts>129.215.92.92</snapshots_to_hosts>
   <sizes_to_hosts>85.76</sizes_to_hosts>
   <send_times_to_hosts>11.04</send_times_to_hosts>
   <time_sent_to_hosts>1397679933</time_sent_to_hosts>
</scheduler_request>
```

Figure 4.6: Example BOINC Client Request

In order to allow custom messages to be sent between the *ad hoc* client and server, we have modified the BOINC scheduler to allow our own subset of *ad hoc* XML elements and values to be entered. Figure 4.6 shows these additional XML elements that either describe the state of the virtual machine or the *ad hoc* host. In the former case, additional elements include the virtual machine's IP address, whether the virtual machine has been assigned a cloud job and whether the virtual machine has failed.

The elements describing the state of the *ad hoc* host include the directory of the *ad hoc* client and whether the *ad hoc* host has just received, restored or deleted a virtual machine checkpoint; the ID and IP addresses uniquely identify these checkpoints. Conversely, the *ad hoc* client also describes whether it has sent any virtual machine checkpoints to other *ad hoc* hosts as well as the size of checkpoint, the time when it was sent and the estimated transfer time. We describe how the additions made to the BOINC communication mechanisms are used in subsequent sections of this chapter.

4.3.4 Operational Overview

In this section, we give a high level overview of how the *ad hoc* cloud operates in comparison to the operations and features of V-BOINC shown in the V-BOINC client-server architecture of Figure 3.1. The differences include the tasks performed by the *ad hoc* client and server and the communication mechanisms between them. The client-server architecture of the *ad hoc* cloud computing platform is shown in Figure 4.7.

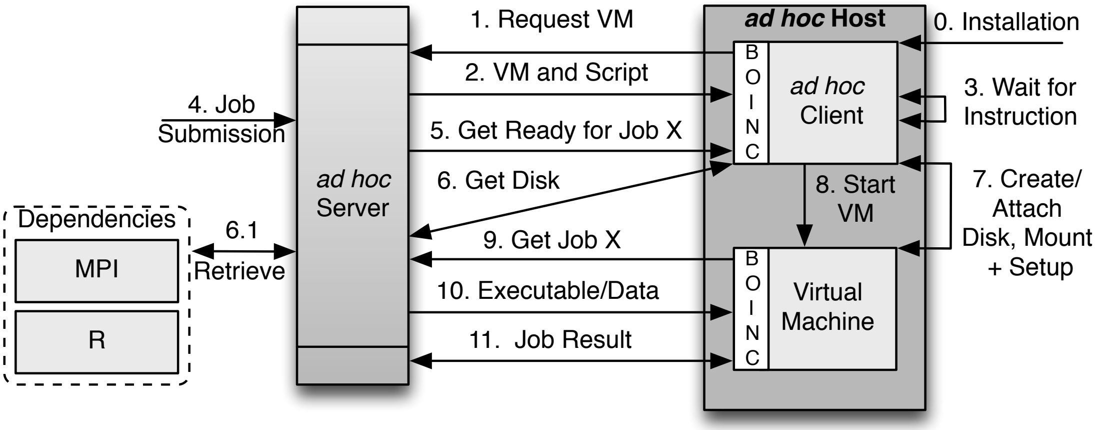


Figure 4.7: The *ad hoc* Cloud Client-Server Architecture

Firstly, an *ad hoc* host owner installs and instantiates the *ad hoc* client on the *ad hoc* host (0) which then triggers the *ad hoc* client to automatically request a virtual machine from the *ad hoc* server (1). A virtual machine image and an executable script used to decompress the virtual machine are then sent to the *ad hoc* host (2).

These initial steps are similar to the those performed by V-BOINC, however V-BOINC assumes that a virtual machine is only downloaded when it has a job to execute. This is in contrast to the *ad hoc* cloud where a virtual machine is instantly installed when an *ad hoc* host connects to an *ad hoc* server and waits on an instruction to start (3). On a number of occasions where the number of cloud jobs is less than the available *ad hoc* hosts, some virtual machine images may never be used. However, we argue that by initially downloading the virtual machine image, the reduced time taken to begin executing a cloud job is necessary in order to reduce the overall time cloud users wait for their results to materialize. New virtual machine images are only downloaded when the *ad hoc* host user or owner deletes the virtual machine image.

A job is then submitted to the *ad hoc* server by the *ad hoc* cloud user (4) and the *ad hoc* client is then instructed to prepare for executing a job (5). This message will include whether the job has any dependencies and if so, the correct DepDisk uploaded by the *ad hoc* cloud user during job submission, is downloaded from the *ad hoc* server (6/6.1). Subsequent processes follow the operations performed by V-BOINC. The DepDisk is either attached or if a DepDisk does not exist, a fresh virtual disk is created and attached (7). The virtual machine is then started (8) and instructed to ask for the job it has been prepared for (9). The cloud job and data are then sent to the virtual machine (10), executed and upon completion, the results are returned to the server (11) for the *ad hoc* cloud user to view and download.

Despite the relatively small differences between the client-server architectures of V-BOINC and the *ad hoc* cloud, the individual client and server components, as well as the communication mechanisms between them, are substantially different. The *ad hoc* cloud offers a vast array of features that both BOINC and V-BOINC cannot offer; these features are what has transformed the virtualized volunteer infrastructure V-BOINC into an *ad hoc* cloud computing platform.

We now describe in detail how the additional features offered by both the *ad hoc* client and server are implemented and integrated with one another to create a successfully operating *ad hoc* cloud computing platform. We order our discussion starting from the processes involved when an *ad hoc* cloud user first submits a job until the moment the user receives their results.

4.4 BOINC Job Submission

To use the resources available in the *ad hoc* cloud, an *ad hoc* cloud user must submit a job to the *ad hoc* server. However in the case of both BOINC and V-BOINC, applications that volunteer hosts execute are statically created before the server begins distributing these applications. Both BOINC and V-BOINC applications are first complied for the target architectures the applications will execute on and are then uploaded to the *ad hoc* server and registered with BOINC. Therefore, dynamically allowing *ad hoc* cloud users to arbitrary submit applications at any time is not a trivial task. Other studies have investigated how to effectively enable job submission to BOINC.

4.4.1 Overview of BOINC Submission Systems

WS-PGRADE/gUse was one of the first tools to allow job submission to BOINC via the CancerGrid [135]. WS-PGRADE/gUse enables a large number of communities to access Grid, Desktop Grid and cloud infrastructures without having to spend a significant amount of time creating scripts to deploy their application on the infrastructure [134]. WS-PGRADE/gUse provides a workflow-based GUI created from the open source Liferay portal [25] served from an Apache Tomcat server. This may either be installed on a host local to an end-user's infrastructure or on a remote host such as the public gUse service located at [22]. In order to provide job submission capabilities for a large number of infrastructures, WS-PGRADE uses DCI-Bridge; a service that offers standardized access to a variety of computational infrastructures [144].

One infrastructure WS-PGRADE is compatible with is BOINC, however WS-PGRADE/gUse is not a solution that is integrated into BOINC but is rather a service that interacts with BOINC. Despite providing standardized access to multiple computational platforms as well as being popular in the scientific communities, the effort required to integrate WS-PGRADE/gUSE with the *ad hoc* cloud would be too great for this type of platform.

For example, we would be required to install WS-PGRADE/gUSE locally and create a customized web-portal environment for job submission to the *ad hoc* cloud. This requires the installation of further libraries and packages, e.g. the portal project Liferay. Furthermore, as WS-PGRADE/gUse is a framework providing many functionalities, many of which we do not require, the computational overhead of using the framework may also be larger than required. WS-PGRADE is also used for submitting workflows to computational infrastructures, however many of the jobs submitted to the *ad hoc* cloud will not be workflow-based. However it is not unreasonable to assume that in the future, the *ad hoc* cloud platform will become another infrastructure WS-PGRADE and DCI-Bridge are able to submit jobs to.

Rios *et al.* also have a similar goal of reducing the barriers of use to BOINC and therefore have created a tool called Legion to generate web portals to perform a variety of tasks; one of those is submitting workflows to BOINC [187]. This is performed by creating a web interface that interacts with a Legion web service that is able to interact with the BOINC server via SOAP.

However in order to interact with BOINC tasks, Legion stores additional information about these tasks within the BOINC database, however BOINC already creates and stores this information within the database by default, and therefore a degree of unnecessary data redundancy occurs. Furthermore, Legion requires additional libraries to allow job submission to BOINC.

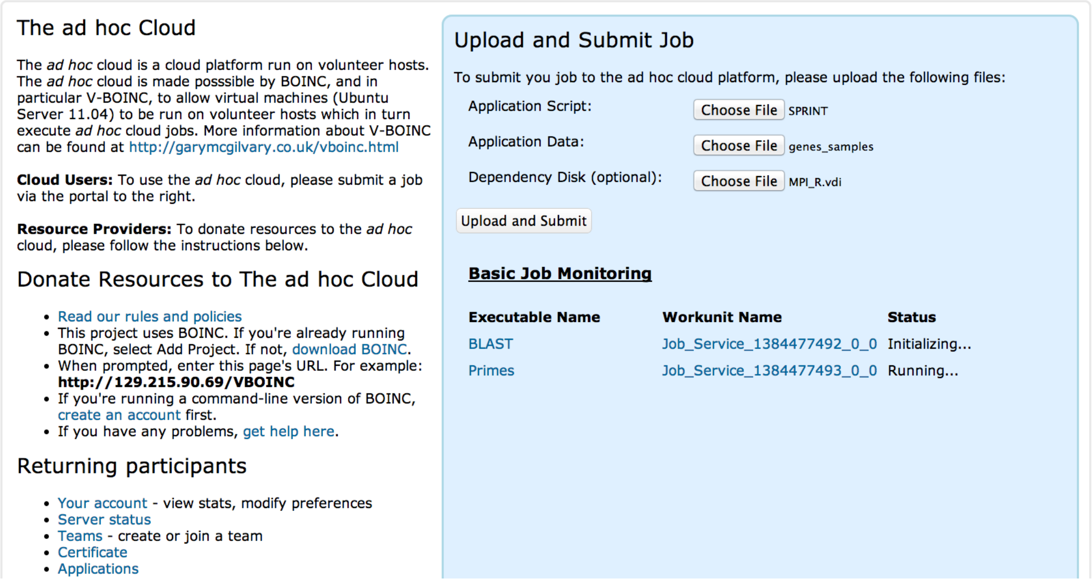
For similar reasons to why we didn't adopt WS-PGRADE/gUSE to submit jobs to BOINC, Legion also requires too much effort to integrate into the *ad hoc* cloud platform and may also generate an additional computational overhead due to the variety of other tasks Legion performs.

After investigating other available frameworks claiming to allow job submission to BOINC, we found that they were either deemed unfit for our purposes, for reasons similar to above, or did not offer the simple functionality we require. We therefore decided to implement our own job submission system that is integrated into BOINC. Our system is composed of two components: the *ad hoc* Cloud Interface and the *work creator*, both of which were depicted in Figure 4.4 showing the *ad hoc* server architecture.

4.4.2 *Ad hoc* Cloud Interface

The *ad hoc* Cloud Interface is a modified version of the default user interface BOINC provides to its volunteer users. This gives users instructions about how to use BOINC as well as access to their on-line account allowing volunteer user-based preferences to be modified or the state of current or previously run tasks to be viewed. The default volunteer host-user interface BOINC provides is shown on the left-hand side of Figure 4.8.

To enable job submission using the BOINC default interface, we have modified the interface to allow *ad hoc* cloud users to upload an application executable and optional data to be analysed as well as a DepDisk if the application requires it. As mentioned in



The ad hoc Cloud

The *ad hoc* cloud is a cloud platform run on volunteer hosts. The *ad hoc* cloud is made posssible by BOINC, and in particular V-BOINC, to allow virtual machines (Ubuntu Server 11.04) to be run on volunteer hosts which in turn execute *ad hoc* cloud jobs. More information about V-BOINC can be found at http://garymcgilvary.co.uk/vboinc.html

Cloud Users: To use the *ad hoc* cloud, please submit a job via the portal to the right.

Resource Providers: To donate resources to the *ad hoc* cloud, please follow the instructions below.

Donate Resources to The ad hoc Cloud

- Read our rules and policies
- This project uses BOINC. If you're already running BOINC, select Add Project. If not, download BOINC.
- When prompted, enter this page's URL. For example: **http://129.215.90.69/VBOINC**
- If you're running a command-line version of BOINC, create an account first.
- If you have any problems, get help here.

Returning participants

- Your account - view stats, modify preferences
- Server status
- Teams - create or join a team
- Certificate
- Applications

Upload and Submit Job

To submit you job to the ad hoc cloud platform, please upload the following files:

Application Script:	Choose File	SPRINT
Application Data:	Choose File	genes_samples
Dependency Disk (optional):	Choose File	MPI_R.vdi

Upload and Submit

Basic Job Monitoring

Executable Name	Workunit Name	Status
BLAST	Job_Service_1384477492_0_0	Initializing...
Primes	Job_Service_1384477493_0_0	Running...

Figure 4.8: The *ad hoc* Cloud Interface with Job Submission

Section 1.5 of Chapter 1, applications that contain an executable and data are the types of applications we have only tested on the *ad hoc* cloud, however other applications may be able to execute successfully; testing additional application types is left for future work. Furthermore, we only test applications that are able to execute on an Ubuntu Server 11.04 OS, however a large selection of Operating Systems can potentially be selected by and provided to the *ad hoc* cloud user to allow them to execute a vast array of different applications on different environments.

The modified interface also shows basic monitoring of jobs that have been submitted by the *ad hoc* cloud user; detailed job information can be viewed by browsing BOINC's default task pages. After an *ad hoc* cloud user uploads and submits their application and optional data and DepDisk to the *ad hoc* server, these files are placed into a numbered directory in a folder named *jobs/* in the Job Service project; this is shown in *ad hoc* server architecture of Figure 4.4.

For example, the third application to be submitted can be found in the directory *jobs/3*. Within each numbered directory, the application and related files are labelled dependent on which argument they were uploaded using the interface. For example, the application *SPRINT* and its data *genes_samples* are relabelled to *SPRINT.app* and *genes_samples.data* respectively.

4.4.3 Creating Work

In order to register the application and optional entities with BOINC, we have created and added a daemon to BOINC called *work creator*, as shown in Figure 4.4. This daemon periodically checks the *jobs/* directory to determine if any new applications have been submitted and if so, the application and the optional entities are segregated into application and input files. This information is used to create a BOINC input template file; a file describing the properties of a job in XML, such as the application and its input files. A output template file is also created describing the application's output files; we assume that an application's results are written to a single output file for testing. These templates are then used to register the application with BOINC using BOINC's C++ *create_work* function. An example method call is shown in Figure 4.9.

```
int create_work(
 DB_WORKUNIT& workunit,
 const char* input_template,            // input template contents
 const char* output_template_filename,  // output template name
 const char* output_template_filepath,  // output template path
 const char** infiles,                  // array of input files
 int ninfiles                           // number of input files
 SCHED_CONFIG& config
);
```

Figure 4.9: BOINC Workunit Creation

The function *create_work* takes a *DB_Workunit* object as the first argument describing the various features of the workunit (i.e.the cloud job) to be created such as the maximum disk or memory it is allowed to consume during execution. As the cloud job will execute in a virtual machine which itself has a restricted level of resources it is able to consume from the *ad hoc* host, we allow the workunit to consume 100% of each virtual resource. The following three arguments are the input template contents as well as the name and path of the output template. The *infiles* array stores references to the application's input files, e.g. the data to be analysed. Note that a DepDisk is not included as an input file as it must be sent to the *ad hoc* host before the virtual machine starts. This is shown in Figure 4.7 where an *ad hoc* host is told to prepare for a job (5) before the job is executed on the virtual machine (10).

Finally a *SCHED_CONFIG* object that contains various items about the BOINC project (e.g the Job Service project), such as paths to the project and download folders, must be passed to *create_work*. Once the function has successfully completed, a BOINC workunit is created and is automatically stored in the database of the Job Service project. The workunit's metadata is also stored such as the workunit ID, creation time and current state. The workunit then waits in the Job Service database to be distributed to an *ad hoc* guest.

4.5 Job Scheduling

After the successful submission of a job to BOINC, the selection of a near-optimal *ad hoc* host now takes place, a decision that is made by the VM Service *ad hoc* Scheduler shown in Figure 4.4. However as the VM Service and Job Service projects are independent, the former does not know when a job has been submitted to the latter.

Therefore, to determine whether a cloud job has been submitted to the Job Service, we have added a *workunit listener* daemon to the VM Service that checks the Job Service database for new workunits. As cloud jobs may be submitted at any time, the *workunit listener* periodically checks the Job Service database. On the discovery of a new Job Service workunit, the *workunit listener* notifies the *ad hoc* Scheduler; in effect, this daemon acts as a cross-project *feeder* daemon.

When the *ad hoc* Scheduler knows how many cloud jobs are awaiting to be distributed to *ad hoc* guests, it begins the process of selecting a near-optimal *ad hoc* host on which to instantiate the *ad hoc* guest and begin executing cloud jobs. This decision is based on a combination of factors: *ad hoc* host availability, specification, resource load and reliability. We assume that a number of *ad hoc* hosts are available in the *ad hoc* cloud to allow cloud job scheduling to occur. In cases where no *ad hoc* hosts are present or available, cloud jobs will remain in the Job Service project database.

As developing a complex job scheduler is out of the scope of this research, we outline our proposal for a simple *ad hoc* Scheduler below. We build on the large number of previous scheduling studies in the HPC, Grid and cloud computing fields while noting the many improvements that can be made to our scheduler.

4.5.1 Overview of Relevant Schedulers

The regular BOINC server scheduler is a simple scheduler that follows a 'bag of tasks' approach where a job is only sent to a volunteer host that has enough memory and disk space and can complete the job within its deadline [51, 52]; the estimated memory usage, disk usage and deadline are specified via the *DB_Workunit* object when using regular BOINC. Additional scheduling mechanisms are available to cope with the heterogeneous environments of volunteer hosts such as dealing with varying cores, numerical variability and the job size, for example. To ensure a job is successfully executed as well as ensuring volunteer hosts return valid results, BOINC executes a single task on multiple volunteer hosts and compares the results.

The RIDGE system is a reliability-aware platform that takes into account a host's previous performance, behaviour and reliability [70]. RIDGE has been developed on top of BOINC to determine the optimal job redundancy level for each BOINC job. This is in contrast to the regular BOINC server scheduler that statically sets the redundancy level for each job. By dynamically adjusting redundancy levels according to the current state of the volunteer infrastructure as well as scheduling jobs to reliable and well performing volunteer hosts, RIDGE is able to outperform the regular BOINC scheduler in terms of task throughput and can also reduce a job's execution time; RIDGE was tested under a variety of reliability conditions on PlanetLab [70, 200].

Although the *ad hoc* cloud does not employ job redundancy to achieve reliability, the methods of scheduling according to a host's past performance and reliability are similar; a host's reliability is calculated in a similar manner. Reliability or redundancy-based schedulers, such as RIDGE, can be complimented with algorithms that also take into account volunteer host reputation to ensure groups of volunteer hosts do not collude to upload incorrect results [77].

Reputation-based schedulers have also been found to increase the probability of a task being correctly executed [199], while others may also simultaneously minimize the completion time [118, 115]. Furthermore, reliability schedulers may also predict the future availability of volunteer hosts [54, 180, 181] to determine whether volunteer tasks should execute on a host any time soon.

Where possible, volunteer tasks can decomposed into smaller variable sized sub-tasks that can then be dispersed over many volunteer hosts. By matching the capability and performance of each volunteer host to the resource requirements of the sub-task, the overall completion time of the whole task can potentially be reduced in a large number of cases [206]. For data-intensive applications running on volunteer infrastructures, volunteer hosts can be ranked based on their estimated download time of a piece of data from the volunteer server [138, 139, 140]. The volunteer host that is predicted to offer the best download time is then chosen to execute the volunteer task.

Scheduling may also be based on whitebox [167] or blackbox [197] methods where either a large or small amount of information is known about the application before execution respectively. Jobs can also be scheduled to near-optimal hosts based on the job's predicted resource requirements. This is achieved by comparing the job to all others that have previously executed [67, 45, 182]. A near-optimal host can then be chosen if it has the required resources available.

The requirements of an end-user can also be a factor during scheduler decisions, for example, a required completion deadline or cost budget [226, 74, 73]. Others schedulers may aim to minimize the computation time [95, 96], strike a balance between cost and performance [88, 119], increase the profit of the service provider [179] or ensure that provider-specified SLAs are fulfilled [182, 222, 46].

This is by no means a complete overview of the current state of scheduling in computational environments. The studies outlined above are a small subset of relevant material to show the potential improvements that could be made to our *ad hoc* Scheduler described below. By incorporating a number of these scheduling methods, the accuracy of our own scheduling proposal would increase the success rate of cloud jobs running on the *ad hoc* cloud, decrease their overall completion time and improve task throughput.

4.5.2 Host Filtering

We model our job scheduling mechanism on the virtual machine scheduler of OpenStack. OpenStack is an open source and scalable operating platform for building public and private clouds [31]. Its virtual machine scheduler, called *nova-scheduler*, calculates the near-optimal host to deploy a virtual machine on. This decision is based on host availability, specification and resource load. The *nova-scheduler* has two phases: filtering and weighing. Filtering determines if a host is eligible for a virtual machine to be dispatched to it.

Commonly applied filters are *CoreFilter*, *RamFilter*, *DiskFilter* that determine if a host has enough processors, memory and storage space respectively. This eligibility list is then passed to the weighing phase where hosts are ordered according to administrator-defined weights to determine the best hosts for a virtual machine to be deployed upon. We discuss how cloud jobs are scheduled according to *ad hoc* host availability, specification, resource load and reliability based on a modified version of the OpenStack *nova-scheduler*.

4.5.2.1 Availability

In order to select an available *ad hoc* host, the *ad hoc* server maintains a list of available hosts determined via the *availability checker* daemon added to the VM Service project of the *ad hoc* server shown in Figure 4.4. To ascertain whether an *ad hoc* host is available, the *availability checker* periodically queries the VM Service database to determine when an *ad hoc* client last polled the server. If the *ad hoc* client polled the server within the last two minutes, the *ad hoc* host is deemed available for use.

Currently, regular BOINC clients only contact the server to obtain a job, return results or when a volunteer host explicitly instructs the BOINC client to contact the server. Therefore in most cases, a BOINC client will not poll the BOINC server for long periods of time despite being still available to execute applications. To solve this, we have added a Periodic Updater component to the *ad hoc* client, as shown in Figure 4.5, that polls the *ad hoc* server every minute; this to similar in the case of OpenStack where compute nodes (i.e. those that run virtual machines) periodically signal to the compute service that they are still available. The Periodic Updater is implemented as a *pthread* which is created when the *ad hoc* client is instantiated; POSIX threads, or *pthreads*, is a standard for threads in the Portable Operating System Interface (POSIX) family of standards [196].

Upon each poll from an *ad hoc* client, the *ad hoc* server stores the contact time in the VM Service project database. This allows the *availability checker* to determine whether the *ad hoc* client has indeed polled in the last two minutes. Those who have not polled within this time period are set to unavailable. The *ad hoc* Scheduler queries the VM Service database to obtain a list of all available *ad hoc* hosts.

4.5.2.2 Host Hardware Specifications

Available *ad hoc* hosts are then analyzed to determine if they physically have enough resources available to execute both an *ad hoc* guest and cloud job. Although we do not know the amount of resources a cloud job, and consequently an *ad hoc* guest will use before execution, we assume that both require a reasonable amount of resources to execute effectively. We therefore assume that each *ad hoc* host has at least 1 CPU core, 1 GB of RAM and 20 GB of storage space.

It is possible to monitor and store the execution times and resource usage levels of previously executed cloud jobs or benchmarks and predict a newly submitted cloud job's execution time and resource usage levels based on the similarity. While there are many studies that outline the process and value of employing this approach [95, 96, 145, 128, 67, 45], the difficultly of determining whether a cloud job, before it has even been executed, shares characteristics with those previously run is an extremely difficult task and is worthy of being investigated in a new course of research.

As previously mentioned in Section 2.4.2 of Chapter 2, a BOINC client automatically records the amount of resources the volunteer host has when it is first run. However volunteer user-based preferences limit both the BOINC client's and volunteer application's use of these resources. Based on both these data sets, the *ad hoc* Scheduler analyses the amount of resources an *ad hoc* guest and cloud job could potentially access. *Ad hoc* hosts that do not satisfy the resource criteria above are removed from the list of potential cloud job execution candidates. This is similar to the operations performed by the OpenStack *nova-scheduler* that calculates suitable hosts for virtual machine placement based on the filters *CoreFilter*, *RamFilter* and *DiskFilter*.

4.5.2.3 Resource Load

The resource load of the remaining *ad hoc* hosts is then retrieved. This is made possible by incorporating Ganglia (see Section 2.5.2 of Chapter 2) into the *ad hoc* client, which is depicted as the Resource Monitor in Figure 4.5. Upon installing the *ad hoc* client, an *ad hoc* host user or owner therefore does not need to install Ganglia separately; we discuss the installation of the *ad hoc* cloud components in Section 4.8 of this chapter.

The Ganglia *gmond* daemon runs locally on the *ad hoc* host and collects CPU and memory load as well as disk consumption and network usage. While network usage may be useful to determine which cloud jobs are best suited to a particular *ad hoc* host, we omit network usage from our scheduling calculations and leave this for future work. The Ganglia *gmetad* daemon runs upon the *ad hoc* server and collects the monitoring data from the *ad hoc* hosts. As previously mentioned in Chapter 2, data collected by Ganglia are stored in *rrd* files. To enable the *ad hoc* Scheduler to read the stored values, the *rrd* files for each *ad hoc* host are queried to obtain the latest resource loads. Resource loads can be obtained by using the following command:

```
rrdtool fetch cpu_system.rrd AVERAGE -r 120 -s -120
```

This *rrdtool fetch* command fetches the average CPU loads calculated for each 15 second period over a total of two minutes. By default, Ganglia averages monitoring data over each 15 second period, however we average the load over each two minute period to smooth the fluctuations of real-time monitoring data and get a good indication of the current load.

If an OpenStack scheduler was integrated into the *ad hoc* Scheduler, at this point the *nova-scheduler* would begin the weighing process and then reserve an *ad hoc* host that is available, has enough hardware to exploit and has the least memory usage; the latter can be modified to filter and weigh according other metrics. However, we assume that for the *ad hoc* cloud to offer reasonable performance to cloud jobs, *ad hoc* host processes should not utilize more than 70% of the CPU and have at least 512 MB of memory available when the cloud job is executing. The output from the above command is passed to the *ad hoc* Scheduler which decides if the current load is acceptable for *ad hoc* guest and cloud job execution. *Ad hoc* hosts that have an average greater than the values specified are removed from the list of potential execution candidates. These average resource usage values are stored alongside the potential *ad hoc* hosts database entries that could be used to execute currently awaiting cloud jobs.

In summary, an *ad hoc* host must have the hardware specifications previously mentioned and have enough of these resources available to offer reasonable performance. For example, although an *ad hoc* host with a total of 768 MB of RAM (i.e. less than our 1GB requirement) could be frequently underutilized, therefore meeting our minimum available amount of memory set at 512 MB, the lack of potential access to more resources does not give the cloud job the opportunity to perform better when it requires more resources. Therefore this is why the *ad hoc* Scheduler filters *ad hoc* hosts based on both hardware specifications and resource load.

4.5.3 Calculating Host Reliability

Due to the uncertain nature of *ad hoc* cloud computing where hosts may leave or fail at any moment, the reliability of *ad hoc* hosts must be taken into account. An *ad hoc* host's reliability is based on five factors:

1. the total number of cloud jobs previously assigned to the *ad hoc* host,
2. the total number of cloud jobs previously completed by the *ad hoc* host,
3. the number of *ad hoc* host failures,
4. the number of *ad hoc* guest failures,
5. the current resource load of an *ad hoc* host.

Examples of *ad hoc* host failures include host termination or any hardware or OS failure that causes the *ad hoc* client to stop operating; for example, kernel panic. Examples of *ad hoc* guest failures include failures related to virtual machine configuration, instantiation, execution, and shutdown. The reliability factors (1)-(3) are monitored by the *ad hoc* server. For each *ad hoc* host, the number of assigned and successfully completed cloud jobs by default are recorded in the Job Service database by BOINC. However as the *ad hoc* Scheduler is part of the VM Service project, it must query the Job Service database to obtain these figures. The number of *ad hoc* host failures can be monitored by the VM Service's *availability checker* daemon which will set an *ad hoc* host to terminated or failed after two minutes of inactivity.

The reliability factors (4)-(5) are monitored by the *ad hoc* client when a cloud job is executing. Any failure relating to the *ad hoc* guest is detected by either the Running Detector or Accessible Detector components shown in Figure 4.5. Virtual machine configuration error, such a failure during registration with VirtualBox or a DepDisk not attaching, are detected by timeouts. Similarly, an *ad hoc* guest is deemed failed if it has not instantiated or shutdown within a certain time period. To determine whether an *ad hoc* guest is still executing, it is periodically polled every ten seconds using VirtualBox's *VBoxManage* API; this ensures that a non-operational *ad hoc* guest is detected quickly with minimal resource overheads. The *runningvms* function outputs a list of running virtual machines and is parsed to determine if the virtual machine is still running.

The current resource loads of an *ad hoc* host can be monitored either by Ganglia or via BOINC's basic monitoring mechanisms. The regular BOINC client monitors the total CPU usage of non-BOINC processes to determine when to suspend BOINC if non-BOINC processes exceeded a threshold specified by the volunteer user. Regardless of the monitoring mechanism employed, an *ad hoc* host's current resource usage may affect reliability when the host becomes heavily utilized by host processes, i.e. those executed on behalf of the *ad hoc* host user, for long periods of time. Therefore the performance of the cloud job will suffer and may take a substantial time to complete; this is unacceptable when running tasks on any cloud platform. Although the current resource loads of an *ad hoc* host may affect the probability of a cloud job completing, we do not incorporate this resource load functionality into our reliability calculations and therefore leave it as future work.

Upon the detection of an *ad hoc* guest failure, the *ad hoc* client informs the *ad hoc* server by inserting the failure type into the <*failure*></*failure*>XML element of the modified BOINC communication mechanism as shown in Figure 4.6. The *ad hoc* server could also be informed of any *ad hoc* host performance issues in the same way. Based on the data sent from the *ad hoc* client and the data collected on the *ad hoc* server, the server is then able to calculate the reliability of each *ad hoc* host using the following formula:

$$host\_reliability = \begin{cases} 0 & \text{if } NF = CA \\ 100 & \text{if } NF = 0 \\ (CC/CA) * 100 & \text{otherwise} \end{cases}$$

where,

NF = the total number of *ad hoc* host and guest failures,
CA = the total number of cloud jobs assigned to the *ad hoc* host,
CC = the total number of cloud jobs completed by the *ad hoc* host.

An *ad hoc* host's reliability is calculated after cloud job has completed, the *ad hoc* guest has become non-operational or the *ad hoc* host has not polled within the last two minutes. The calculated reliabilities are then stored in the VM Service project database alongside the information of each candidate *ad hoc* host. This reliability calculation gives an estimate of the *ad hoc* host's behaviour for the entire time the host is part of the *ad hoc* cloud. This calculation could however be improved to reflect an *ad hoc* host's recent reliability, e.g over the last few hours. Furthermore, daily or weekly patterns could also be detected to determine whether or not to assign a cloud job to the *ad hoc* host. We leave the investigation and potential incorporation of these possible additions as future work.

4.5.4 Making a Decision

The *ad hoc* server now has a filtered list of potential execution candidates based on *ad hoc* host availability, hardware specifications and current resource load. This list is then sorted in descending order according to the reliability of each *ad hoc* host. Table 4.1 shows an example candidate list.

To schedule a cloud job to a single *ad hoc* host, the *ad hoc* scheduler selects the head element of the list, i.e. the *ad hoc* host that is most reliable. Similarly, the first x candidates are selected when scheduling a batch of x cloud jobs. This ensures that reliable *ad hoc* hosts that are able to offer reasonable resources to a cloud job always have a job to execute. This is a simple scheduler and many improvements could be

Reliability	Host ID	CPU Free	Memory Free	Disk Free
99	12	40%	678 MB	160 GB
82	89	88%	2 GB	850 GB
68	17	95%	6 GB	200 GB
44	2	45%	1.1 GB	20 GB

Table 4.1: Example Scheduling Candidate List

made to optimize the scheduling process. For example, it may be better to schedule a single cloud job to *ad hoc* host 89 which is still reliable but offers more resources for the cloud job to consume. As developing a complex job scheduler is out of the scope of this research, we leave the evaluation and possible incorporation of these improvements as future work.

4.5.5 Preparing and Executing a Cloud Job

After an appropriate *ad hoc* host has been selected to execute a cloud job, the host is instructed to perform the necessary steps to allow the cloud job to begin executing in the virtual machine; this is step (5) shown in Figure 4.7 depicting the high-level *ad hoc* cloud client-server architecture. However as both regular BOINC and V-BOINC are volunteer infrastructures and are therefore controlled by the volunteer host user, both implementations do not typically allow require server-initiated communication functionality. Hence BOINC clients typically do not receive messages from a server unless they initiate a request.

To solve this problem, we developed five Listeners, shown in Figure 4.5, to allow the *ad hoc* server to communicate with *ad hoc* clients without waiting for clients to initiate the communication. The Listeners available are: the Job Receiver, Host Resetter, Snapshot Deleter, Snapshot Receiver and Snapshot Restorer. The *ad hoc* server instructs *ad hoc* clients by appending additional XML elements to the default BOINC server message sent to BOINC clients; this typically includes information about the volunteer host such as its host identifier, the projects it is attached to and BOINC tasks it possesses, for example. Each Listener then parses the appropriate section of the message to determine whether to perform any actions. The following four XML elements, shown in Figure 4.10, are appended to a BOINC server message to instruct an *ad hoc* client to begin preparing the *ad hoc* host for executing the cloud job.

```
<job_service_url> http://129.215.90.11/Job_Service</job_service_url>
<job_service_auth>2_9fec55ffe011154092f0wef24f</job_service_auth>
<job_service_depdisk>MPI_R.vdi</job_service_depdisk>
<job_service_job_id>235</job_service_job_id>
```

Figure 4.10: Section of job Preparation Instruction

The Job Receiver Listener, whose task it is to start the preparation of the *ad hoc* host, will store the parsed values and begin downloading the DepDisk *MPI_R.vdi* from the *ad hoc* server (Figure 4.7 step 6). Similar to the V-BOINC process, the DepDisk will attach to the already downloaded virtual machine and the virtual machine will then be started. The Job Receiver will then instruct the inner BOINC client installed in the virtual machine, to attach to the Job Service project located at the URL *http://129.215.90.11/Job_Service* using the weak authenticator supplied (Figure 4.7 step 9); the *ad hoc* client uses the *guestcontrol* function of the VirtualBox API to allow commands to be executed in the virtual machine.

Although the *ad hoc* server knows the correct workunits to supply to each *ad hoc* guest, the *ad hoc* guest relays the workunit ID back to the server when attaching to the Job Service project. This confirms that the correct *ad hoc* guest will receive the correct cloud job for the environment prepared.

4.6 Making the Unreliable Reliable

After a cloud job has been downloaded to the *ad hoc* guest and begins executing, it relies on the successful operation and availability of both the *ad hoc* host and guest. By executing a cloud infrastructure over an unreliable set of volunteer hosts, cloud jobs will however be greatly affected by the premature termination and failures of *ad hoc* hosts. There have been many studies researching how to introduce reliability into unreliable infrastructures.

4.6.1 Overview of Fault Tolerant Computing

Three common fault tolerance recovery measures used in Grid infrastructures are [86]: *checkpointing*, *replication* and *rescheduling*.

Checkpointing allows the state of an executing task to be periodically saved. Grid tasks that fail can either be restarted from a previous checkpoint on the same host

[124], or migrated to another host for continued execution [148]. For a large number of scientific applications that use MPI to achieve parallelism, various studies show that it is possible to implement coordinated checkpointing within MPI [219].

Replication provides fault tolerance by allowing many replicas of a single task to be distributed to multiple hosts with the hope that one succeeds. Systems such as RIDGE [70], BOINC [51] and other studies [200] employ task replication. Replication not only benefits the Grid user but can also benefit the Grid provider in order to fulfil their defined SLAs [147]. Replication is also common in cloud computing environments to provide reliable storage service; Amazon S3 is an example of such a service [4].

Rescheduling restarts failed tasks on different hosts; this is contrast to checkpointing where a task can begin executing from a previous state. This is particularly useful to reduce the overheads associated with checkpointing and replication [86]. Task rescheduling has found be a reasonable approach for providing reliability to a Grid user [123] however for long-running processes, it may introduce significant overheads.

Weissman investigates whether checkpoint-recovery or wide-area replication can introduce fault tolerance measures for Single Program Multiple Data (SPMD) parallel applications distributed over WANs [215]. Checkpointing measures are implemented within the application code and periodically the data the code operates on is saved to either an NFS-mounted disk or a parallel array of local disks. Wide-area replication is performed by the Gallop scheduler, which is primarily used to select the best execution sites for applications. Weissman finds that for SPMD parallel applications in the configurations mentioned, checkpointing may offer a lower performance overhead for small applications when fast local disks are present. Wide-area replication is more suited to applications on a much larger scale. It is noted however that one fault tolerance approach may not be acceptable for all classes of application. Furthermore, this study does not investigate the overheads of recovery and the optimal fault tolerant method for a variety of environments, e.g the Grid versus the *ad hoc* cloud for example.

Fault tolerance measures have also been introduced into volunteer or Desktop Grid infrastructures. Saramenta introduces fault tolerant measures to ensure the validity of results while reducing the overheads of redundancy-based fault tolerance [192]. The author combines redundancy with spot-checking, where a spot-check job is sent to a volunteer host for execution but the result of that job is already known beforehand. If the volunteer host returns a bad result, the server (e.g. the BOINC server) knows not to trust that particular host and can invalidate the results from all previous workunits this host has executed or blacklist the host to ensure it never computes a task again.

Domingues *et al.* propose a novel technique that aims to identify malicious hosts by employing task redundancy with checkpoint-based verification [92]. Before a task is started, a specific set of future checkpoints are selected by the server (e.g. the BOINC server). The task is then instructed to execute and while doing so, it periodically checkpoints. Upon reaching a server-specified checkpoint (e.g. number 24), each redundant task computes the hash of the checkpoint and sends this to the server upon the next communication. The server is then able to compare checkpoints to determine the validity of the currently executing redundant tasks.

The use of virtualization also makes it possible for applications to execute in a reliable and fault tolerant manner. Nagarajan *et al.* propose a method where Xen virtual machines executing MPI tasks are migrated from a source host to a target host when the former is determined to have substantially high temperatures, fan speeds and voltage usage [173]. The target host is selected based on the least CPU load as monitored by Ganglia. The authors use live migration; a method of transferring a virtual machine from one host to another without affecting the availability of the virtual machine.

Note that in order to enable live migration, Xen requires that firstly, the source and target hosts have the same hardware and have equal CPU specifications and secondly, shared storage such as NFS is used. In the event a source host fails without any warning and before the virtual machine is migrated to another host, Nagarajan *et al.* dictate that the virtual machine is simply restored on the source host from its last checkpoint.

Nagarajan *et al.* claim that their working prototype minimizes the transfer times and overall downtime experienced by live migrating the virtual machine executing MPI tasks. However, the perceived primary contribution of the work is not evaluated, i.e. an investigation into the effectiveness of their health monitoring algorithm that decides when to migrate a virtual machine. Therefore the results obtained are similar to those from benchmarking Xen's live migration feature when MPI tasks are executed.

Cully *et al.* propose a similar method where virtual machines are live migrated using Xen to ensure reliable application execution [84]. The authors employ an aggressive checkpointing approach where checkpoints are taken at very high frequencies, for example, one checkpoint every 25ms. Their architecture is based on a primary host that performs checkpointing and these are then replicated on a backup host. The system state is not available or is perceived not to have been modified until the checkpoint has successfully been sent to the backup host. If the primary host fails, the virtual machine stored in memory on the backup host can begin execution once the failure has been detected. In the event both the primary and backup host fails, the virtual machine can be restored from a mirror of the backup host's disks elsewhere.

The aggressive checkpointing approach does however introduce a variety of problems. The speed in which checkpoints are taken may place extreme resource demands on the source host. Furthermore, each checkpoint must stop the virtual machine for a brief amount of time and the application output and network packets must be buffered until a checkpoint has been committed to the backup host. Afterwards, these they can be replayed to the virtualization user. Cully *et al.* find that their outlined solution is able to offer high-availability in the face of failures, however their system can introduce 50% and 25% performance penalties when executing general-purpose and network-dependent tasks respectively; the latter is caused by buffering network packets until checkpoints are committed.

4.6.2 P2P Reliability Algorithm

The *ad hoc* cloud, and in particular the working relationship between the *ad hoc* client and server, provides fault tolerance by also employing checkpointing of the *ad hoc* guest. However in contrast to similar research outlined above, we have developed a P2P reliability algorithm where the *ad hoc* host periodically checkpoints an *ad hoc* guest throughout its execution and distributes these checkpoints in a P2P fashion to a near-optimal number of *ad hoc* hosts, preferably in the same cloudlet. As previously mentioned, a cloudlet is a set of connected *ad hoc* guests that provide a particular service or execution environment, i.e. those that posses the same application dependencies and DepDisk.

In the event of an *ad hoc* host prematurely terminating or failing, or the *ad hoc* guest simply fails, the *ad hoc* server instructs one of the *ad hoc* guest's checkpoints to be restored on another *ad hoc* host elsewhere. An example of our P2P reliability algorithm is shown in Figure 4.11. We describe the implementation details in subsequent sections.

In this example, fourteen *ad hoc* hosts, each with an *ad hoc* guest labelled from A to N, either run and execute a cloud job on the guest or are await instruction to configure and boot the guest. Firstly, *ad hoc* guest A receives and begins the execution of a cloud job. During the *ad hoc* guest's execution, it is periodically checkpointed and the resulting checkpoints are distributed to a select number of *ad hoc* hosts based on their reliability; in this example, the checkpoint is sent to *ad hoc* hosts' B, E and K. However after a period of time, *ad hoc* host A prematurely terminates therefore inter-

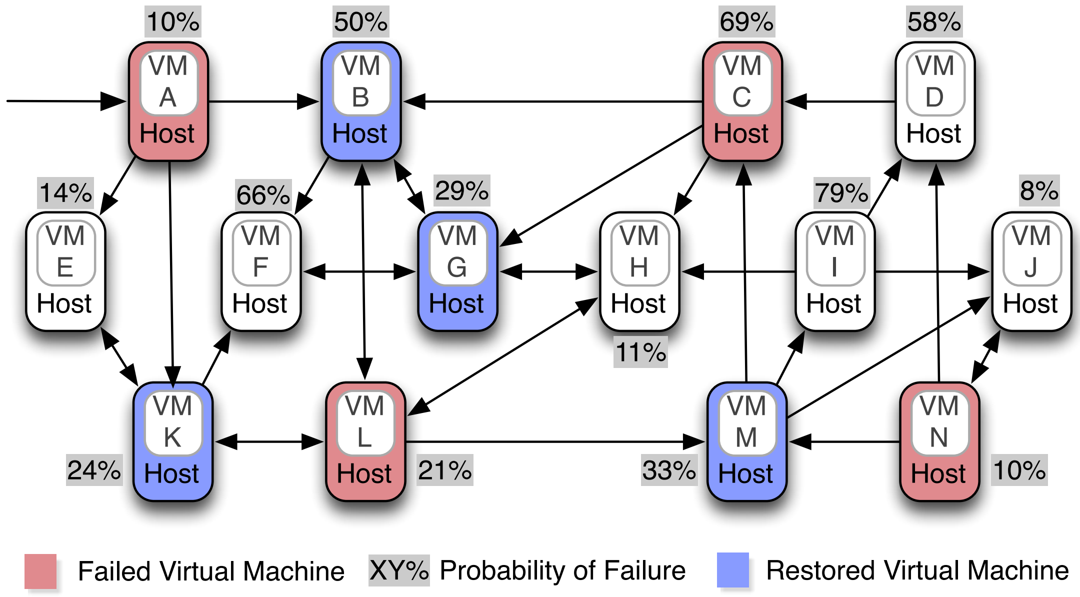


Figure 4.11: P2P Reliability Snapshot Overview

rupting the execution of *ad hoc* guest A and its cloud job. This failure is detected by the *ad hoc* server and the process of restoring the *ad hoc* guest's checkpoints is started. The *ad hoc* server then selects a near-optimal *ad hoc* host, in this case *ad hoc* host K, and instructs its *ad hoc* client to restore the previously interrupted *ad hoc* guest A's checkpoint. The following sections describe exactly how this P2P reliability algorithm is implemented.

4.6.3 Periodic Checkpointing

After a cloud job begins executing on an *ad hoc* guest, the Snapshotter component of the *ad hoc* client, shown in Figure 4.5, begins taking periodic checkpoints of the *ad hoc* guest. By default checkpoints are taken every five minutes, i.e. 12 per hour; the reasoning behind this figure is described in Chapter 6. The *ad hoc* client is able to instruct VirtualBox to take a live checkpoint when the *ad hoc* guest is running by issuing the following command to the VirtualBox API *VBoxManage*:

```
VBoxManage snapshot vboinc_vm take snapshot001 --pause
```

This command calls the *snapshot* function passing as arguments, the name of the *ad hoc* guest, the operation to perform, the name of the checkpoint and an instruction to pause the *ad hoc* guest when taking the checkpoint. In our experience, omitting the latter can cause the checkpointing process to fail. In this example the name of the *ad hoc* guest is *vboinc_vm*, the operation is *take* and the name of the checkpoint is *snapshot001*. Upon the successful completion of the *snapshot* function, a copy of the virtual machine's settings, new differencing VDI images for each virtual disk and a memory state file are placed within the *ad hoc* guest's *Snapshots/* folder; see Section 3.3.5 of Chapter 3 for a more detailed description of VirtualBox checkpoints.

4.6.4 Checkpoint Scheduling and Distribution

After each checkpoint is taken, the files in the *Snapshots/* folder are checked by the P2P Distribution component (shown in Figure 4.5) to determine whether they are needed in the future. Differencing images, which store write operations between checkpoint intervals, are required to sequentially build the correct state of the *ad hoc* guest during a *restore* operation. Memory state files of previous checkpoints are however not required as the contents of an *ad hoc* guest's memory is restored from a single memory state file rather than from multiple sequentially linked memory state files.

Due to the potentially large size of memory state files (i.e. up to the memory size allocated to the virtual machine), previous memory state files are deleted. Following the removal of extraneous files from the *Snapshots/* folder, the *ad hoc* host's IP and host ID are extracted from the underlying BOINC client and the *Snapshots/* folder is then compressed as a *.tar.gz* file conforming to the naming convention *IP_ID.tar.gz*; we explain why this is required later in this chapter.

4.6.4.1 Scheduling

The P2P Scheduler component of the *ad hoc* client then begins the process of deciding which *ad hoc* hosts, or potential checkpoint receivers, the compressed checkpoint file should be sent to; the scheduling process is executed after each checkpoint is taken. It does this by selecting the available and most reliable potential checkpoint receivers in the same cloudlet. By selecting members in the same cloudlet, an *ad hoc* guest can be restored quickly due to the locally available DepDisk. In the event that there are no other members in the same cloudlet, a cloud job's dependencies must be downloaded to the checkpoint receiver before the checkpoint can be restored.

The *ad hoc* client is able to determine the cloudlet membership, availability and reliability of all other *ad hoc* hosts by polling the *ad hoc* server. As previously mentioned, each *ad hoc* client periodically polls the *ad hoc* server, via the Periodic Updater component, to signify its availability to the *ad hoc* cloud. Upon each poll, the *ad hoc* server returns the cloudlet membership, reliability value, IP address and the working directory of the remote *ad hoc* clients of other available *ad hoc* hosts to the checkpoint sender's BOINC Scheduler. This data is then parsed and passed to the P2P Scheduler component to begin the scheduling process. The P2P Scheduler:

1. filters the list of potential checkpoint receivers based on whether they are in use, i.e. running an *ad hoc* guest and cloud job.

2. filters the list of potential checkpoint receivers based on sender's cloudlet membership,

3. orders the list in descending order based on the reliability of the potential checkpoint receivers,

4. selects the first n hosts that have less than a 5% chance of all n failing.

Ad hoc hosts that are in use are not chosen as they cannot restore the checkpoint if the checkpoint sender's guest becomes non-operational. It is possible to execute two virtual machines concurrently on an *ad hoc* host, however the performance overheads would be significantly larger. In the event no *ad hoc* hosts are free, the P2P Scheduler will schedule checkpoints to *ad hoc* hosts that are currently in-use with the hope that one becomes available. To solve the problem where other *ad hoc* hosts instead become available, it is possible to migrate the checkpoint from the in-use *ad hoc* host to those that are available and then perform the restoration; this however is not currently implemented and is left for future work.

The P2P Scheduler then schedules according to which cloudlet the potential checkpoint receivers are members of. The checkpoint sender is able to determine which cloudlet it belongs to based on the name of the DepDisk attached to the *ad hoc* guest. If the cloud job was not submitted with a DepDisk, the *ad hoc* host belongs to the default cloudlet. If a potential checkpoint receiver does not belong to the same cloudlet as the checkpoint sender, it is removed from the list of potential checkpoint receivers; this newly generated list is then order according the reliability. However, in the event that there are no other potential checkpoint receivers in the same cloudlet, the P2P Scheduler will not filter any *ad hoc* hosts, therefore leaving all available *ad hoc* hosts as potential checkpoint receivers.

The P2P Scheduler then selects a number of reliable hosts that have a less than 5% chance of all selected hosts failing. A target that we aim to achieve is to ensure a cloud job will successfully complete 95% of the time. We believe it is unreasonable to assume that an *ad hoc* cloud could successfully complete a cloud job 100% of the time due to its unpredictable and volatile nature. When an *ad hoc* guest must be restored on another *ad hoc* host, the 95% success rate requirement can only be met if at least one of the checkpoint receivers still possess the *ad hoc* guest's checkpoint and the combined probability of all of these receivers failing is 5% or less. This is because the presence of a checkpoint is directly related to the future success of an application in the event an *ad hoc* guest must be migrated and restored for whatever reason.

The combined probability of a selected number of all *ad hoc* hosts failing can be calculated by multiplying the respective failure probabilities (or reliabilities) of each host. For example, Figure 4.11 shows that the probability of a cloud job never completing when running on virtual machine A, is approximately 1.7%; the multiplication of the failure probabilities for the *ad hoc* hosts B, E, and K. The P2P Scheduler assumes that at least three checkpoint receivers should be selected regardless whether the combined failure probability of one or two receivers is less than 5%. For example, as each *ad hoc* host is assumed to be 100% reliable when it first joins, an *ad hoc* host may fail to complete a single cloud job therefore reducing its reliability to zero. By ensuring that a checkpoint is sent to at least three other *ad hoc* hosts, we can be reassured that the *ad hoc* guest can be restored on another *ad hoc* host in most cases.

This scheduling method does however mean that reliable destinations may end up storing many checkpoints. However, the maximum host storage that can be used by the *ad hoc* cloud (e.g. the *ad hoc* client, checkpoints, etc) can be specified by the *ad hoc* host user via regular BOINC, if the host user wished to limit disk consumption. In the event an *ad hoc* host reaches its maximum storage preference limits, the *ad hoc* server does not send the details of that host to polling *ad hoc* clients, therefore ensuring further checkpoints are not sent to the host. Furthermore, reliable hosts may at times become busy by continuously receiving checkpoints from others. This is especially inconvenient for cloud jobs that require acceptable levels of network performance; the exact extent of the performance overheads caused by checkpoint distribution is outlined in the following Chapter.

Scheduling checkpoints to reliable *ad hoc* hosts does however favour *ad hoc* clients that have to send large checkpoints to other *ad hoc* hosts. In this scenario, the reliability-based scheduler by default sends these checkpoints to the fewest but most reliable hosts, in turn reducing the total bandwidth used during the P2P reliability algorithm. There are however improvements that can be to our P2P Scheduler.

For example, checkpoints that are small (e.g. under 100 MB) could be sent to a larger number of unreliable hosts while still meeting the completion target. This would leave the fewer but the most reliable *ad hoc* hosts to be used only for storing larger checkpoints. Furthermore, the P2P Scheduler does not know the storage capacity available on other *ad hoc* hosts and whether the remote host can actually store the checkpoint. Although we leave this addition for future work, this information is not vital as if a transmission of a checkpoint fails, for example due to the lack of storage space, the failure is detected and at least one other potential checkpoint receiver is selected to ensure the 95% success rate requirement is satisfied. We leave the evaluation of these features for future work.

4.6.4.2 Distribution

Once the checkpoint receivers have been selected, the compressed file containing the checkpoint is concurrently distributed to the selected *ad hoc* hosts. To achieve this, we use the tool *pscp*; a program for performing parallel file transfers using the Secure Copy Protocol, or *scp* [33]. Figure 4.12 shows the number of arguments passed to *pscp*.

```
pscp -h hosts.txt ./slots/0/129.125.96.96_543.tar.gz.REMOVE
/home/user/adhoc_client/host_snapshots/
```

Figure 4.12: Example *pscp* Command

Firstly, a *hosts.txt* file is given that contains the IP addresses of the selected checkpoint receivers; this information is collected from the data sent to the *ad hoc* host from the *ad hoc* server. Secondly, the relative or absolute path to the compressed checkpoint file is then given; we append *.REMOVE* to the file for reasons explained later. Finally, the remote directory where the checkpoint should be stored on the checkpoint receiver is given; we specify that a checkpoint should be sent to the *adhoc_client/host_snapshots/* folder of the remote *ad hoc* client.

The *ad hoc* client then executes the *pscp* program and parses its output to determine if any copies of the checkpoint were not successfully transmitted. As previously mentioned, in the event a copy is not transmitted successfully, the *ad hoc* client will select at least one other potential checkpoint receiver ensuring that the combined probability of all of receivers failing is 5% or less. After all of the checkpoint copies are successfully sent, the *ad hoc* client sends an empty file named *IP_ID_complete* to each of the receivers to specify that the transfer operation has been completed.

The *ad hoc* client then informs the *ad hoc* server which *ad hoc* hosts have been sent a copy of the checkpoint by sending the IP address of each checkpoint receiver, the size of the compressed checkpoint file, the estimated total transfer time, and the wallclock time when the file was transferred; these are expressed via the latter four XML elements shown in Figure 4.6. Upon receiving this data, the *ad hoc* server stores each entity in a temporary storage table in the VM Service project database.

The Snapshot Receiver component of the receiving *ad hoc* client periodically checks the *adhoc_client/host_snapshots/* folder to detect if any new checkpoints have been received. If a compressed checkpoint file and *_complete* file exist with equal IP addresses and host IDs, the *ad hoc* client knows that the compressed file has been successfully transferred from an *ad hoc* host with the extracted IP address and host ID. The *ad hoc* client can then remove the additional *.REMOVE* part previously added to the *.tar.gz* file signifying the file can be used to restore an *ad hoc* guest, upon instruction from the *ad hoc* server.

In the common event that multiple checkpoints are received from the same *ad hoc* host, consequently a new *.tar.gz.REMOVE* file will be renamed to *.tar.gz* once the file has been fully received. This simple mechanism aims to save storage space as well as the amount of work an *ad hoc* host has to perform as it does not need to identify and delete previous checkpoints from each host; by renaming, we instead overwrite a previous checkpoint.

Once a checkpoint has been successfully received and is renamed, the *ad hoc* client then sends the checkpoint sender's host IP address and ID, taken from the checkpoint file name, to the *ad hoc* server. The *ad hoc* server then matches and stores the data from the temporary storage table into a table recording all checkpoint transfers between *ad hoc* hosts confirming the fact that a checkpoint has been successfully sent from one *ad hoc* host to another. These reliability measures outlined will continue to execute in the background until the virtual machine completes its assigned cloud job or prematurely terminates.

4.6.5 Checkpoint Restoration

While periodic checkpointing and distribution are important to help introduce reliability into an unreliable infrastructure, the ability to restore *ad hoc* guests in an effective and near-optimal fashion are equally important. The restoration procedure only begins when it is assumed or is known that an event has occurred on an *ad hoc* host that has halted the operation of the *ad hoc* guest and executing cloud job. As previously mentioned, events may include premature termination or failures as well as failures related to the *ad hoc* guest or client. Furthermore, these failures are either determined by the *ad hoc* server's *availability daemon* when an *ad hoc* host has not polled within the last two minutes or the *ad hoc* client informs the *ad hoc* server that a failure has occurred.

In the event of the *ad hoc* server detecting or being informed of a *ad hoc* host, guest or client failure, the server follows a set procedure to restore the *ad hoc* guest on another *ad hoc* host. The *ad hoc* server first retrieves the IP address and host ID of the *ad hoc* host that executed the halted *ad hoc* guest. This information is then passed to the *ad hoc* Scheduler. In the same way the scheduler selected a near-optimal *ad hoc* host to instantiate an *ad hoc* guest, which in turn executes the cloud job, the *ad hoc* Scheduler also selects a near-optimal *ad hoc* host, from those that posses the checkpoint, based on the same scheduling features: host availability, hardware specifications, resource load and reliability.

As a consequence of the P2P Scheduler only distributing an *ad hoc* guest's checkpoints to only those in the same cloudlet, the *ad hoc* Scheduler by default only instructs the restoration to be performed within the same cloudlet. The selected *ad hoc* host is instructed to restore the failed *ad hoc* guest's checkpoint by appending an additional <*snapshot_to_restore*> XML element to the BOINC server-client message. In order for the *ad hoc* client to receive such messages, the Snapshot Restorer component shown in Figure 4.5, checks each BOINC server message to determine if the additional XML element is present. This element takes as an argument the IP address and host ID of the *ad hoc* host that possessed the failed guest.

After the receiving *ad hoc* client extracts the IP and host ID, it is then able to search for appropriate checkpoint in its *adhoc_client/host_snapshots/* folder. When found, the compressed checkpoint file is decompressed and moved to the folder containing the already downloaded virtual machine; in our case this is the initial folder regular BOINC uses to execute scientific applications, i.e. *slots/0*. As each virtual machine and attached disks have the same unique identifier, no problems exist when transferring checkpoints from one virtual machine to another. The *ad hoc* client then re-registers the virtual machine to pick up the addition of the checkpoint and restores the virtual machine to allow the cloud job to continue executing; both the re-registration and restoration processes are performed by interacting with the VirtualBox API.

It may be the case that due to errors that persist between migrations, an *ad hoc* guest may continue to fail and be migrated continuously. To avoid this scenario, the *ad hoc* server limits the number of consecutively failed restorations to five. Furthermore, we also give the cloud user the ability to view how many times their cloud job has been migrated allowing them to decide when it is best for their cloud job and its assigned *ad hoc* guest to terminate; the latter functionality can also be used when a cloud user simply wishes to stop their cloud job.

Once the *ad hoc* guest is found to be successfully running, via the Running and Accessible Detector components, the *ad hoc* client informs the *ad hoc* server that the guest has been successfully restored. This is performed by relaying the received checkpoint IP address and host ID back to the *ad hoc* server in the <*restored_snapshot_id*> and <*restored_snapshot_ip*> XML elements, as shown in Figure 4.6. In order to save as much space on each *ad hoc* host as possible, the *ad hoc* server then instructs all *ad hoc* hosts to delete checkpoints they received from the previously failed *ad hoc* guest or for a guest that has successfully completed its cloud job. An instruction to delete an unnecessary checkpoint is received and performed by the Snapshot Deleter component shown in Figure 4.5.

4.7 Minimizing Host Process Interference

We now discuss possible methods of how to reduce the interference experienced by *ad hoc* host processes caused by processes related to the operation of the *ad hoc* cloud. Host processes may include web browser, text editor, command line or development software processes that are initiated by the *ad hoc* host user. Processes and operations of the *ad hoc* cloud include the execution BOINC, the operations performed by the *ad hoc* client and the processes created by VirtualBox to execute and manage a virtual machine. As the resources of the *ad hoc* cloud are donated, executing host processes should have priority over executing *ad hoc* cloud operations and cloud jobs.

4.7.1 Suspending Tasks

Fortunately, BOINC does provide one solution to this problem. By default, if the total CPU utilization consumed by host processes of non-BOINC related processes exceeds 25% of the CPU, then the BOINC client will suspend the execution of the scientific application; this default limit can be changed by the volunteer user via user-based preferences. However, in the case of the *ad hoc* cloud and V-BOINC, the BOINC client does not execute scientific applications, but instead executes virtual machines controlled by VirtualBox. As the VirtualBox virtual machine is executed as a standalone process, the BOINC client is not able to control or interact with this process in anyway. Therefore the BOINC client is not able to suspend the virtual machine if the total CPU utilization of host processes exceeds the volunteer user's specified limit.

We solve this problem by allowing our *ad hoc* client to suspend virtual machines if the set CPU utilization level is exceed. This is similar to suspending a regular BOINC task in memory if the volunteer user allows this; this option is also set via volunteer user-based preferences. However in order to suspend the virtual machine at the correct moment, the *ad hoc* client must determine which processes are BOINC related and those that are not and whether they exceed the *ad hoc* host user's specified limits.

By modifying the regular BOINC client to create our *ad hoc* client, any overheads introduced by the implementation of the *ad hoc* client are encapsulated within the original processes of BOINC. However as VirtualBox virtual machines are executed as standalone processes, the *ad hoc* client will classify these processes as host processes. However, these are in fact processes related to the operation of the *ad hoc* cloud. To solve this problem, we monitor the CPU utilization of the VirtualBox processes *VBoxHeadless*, *VBoxManage*, *VBoxXPCOMIPCD* and *VBoxSVC* as well as the *ad hoc* client, by periodically parsing the usage levels output from the UNIX-based command *top*.

The remaining percentage of CPU utilization can therefore be assumed to be consumed by non-BOINC processes. Therefore, if this value exceeds the *ad hoc* host user's limits, the virtual machine is suspended. Like BOINC, the virtual machine is resumed when the CPU usage of non-BOINC processes drops below the specified limit. In our development and evaluation of the *ad hoc* cloud, we set such limits to 100% allowing all the CPU to be used to avoid frequent suspending and resuming of the *ad hoc* guest.

Furthermore, BOINC offers no options to suspend and resume scientific applications when memory, storage or network usage levels are exceeded. We leave it as future work to incorporate these features in relation to suspending and resuming virtual machines. We also intend to migrate a virtual machine when it is suspended frequently and for long periods, as well as when the performance of the *ad hoc* host becomes poor.

4.7.2 Dynamic Resource Use Adjustment

Suspending a virtual machine, and therefore a cloud job, when resource use exceeds an *ad hoc* host user's preferences is a useful technique to ensure that host processes experience little interference from cloud processes. This however assumes that an *ad hoc* host user correctly sets the resource use limit, i.e. the user knows that their own host processes will not consume more than its specified share of resources. An *ad hoc* host user is however unlikely to know the resources their processes would ideally like to consume before or during execution.

Therefore we propose an alternative method of minimizing host process interference by dynamically adjusting the maximum level of resources that the *ad hoc* client and virtual machine are able to consume dependent on current utilization rates of host processes. Figure 4.13 gives an example.

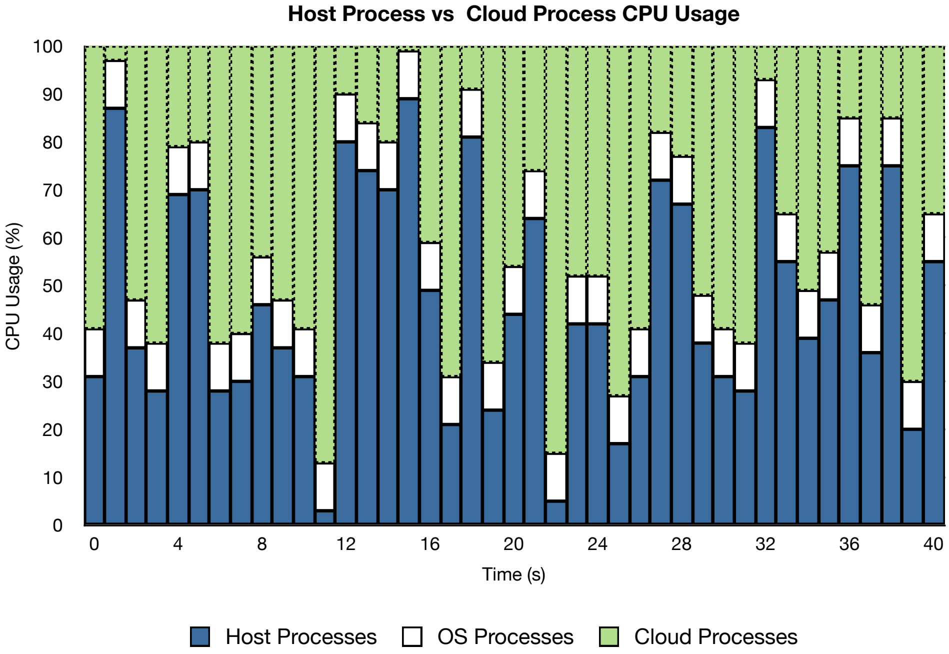


Figure 4.13: Dynamic Resource Use Adjustment Example

The example graph above shows the CPU utilization levels of cloud processes (green) based on the utilization level of the host processes (blue); we assume a 10% utilization rate for OS processes (white). For example, at time *t=20*, we see that the host processes and OS overhead consume approximately 55% of the CPU. This therefore restricts CPU use of cloud processes to the remaining 45%. At *t=21*, the total percentage of the CPU used by host processes and the OS increases to approximately 75%, only leaving

25% of the CPU available for cloud process execution. By dynamically adjusting the maximum CPU utilization percentage for cloud processes, we can in theory allow host processes to execute with no interference.

4.7.3 Potential Solutions

Although we have not developed a solution to provide this functionality, we give a brief outline of tools and techniques that can minimize the interference host processes experience. The dynamic behaviour we regard as future work could potentially be achieved by VirtualBox where the number of physical CPUs and a maximum CPU utilization can be set before a virtual machine is started. However, it would be unreasonable to restart the virtual machine if new resource levels were required. VirtualBox does however support CPU hot-plugging [40]; the ability to add or remove virtual CPUs.

The major advantage of CPU hot-plugging is that virtual CPUs can be added or removed during the execution of a virtual machine. Therefore, as the behaviour and CPU consumption rates of host processes change over time, it is possible to add and remove virtual CPUs dependent on host process CPU utilization. For example, if an *ad hoc* host has one CPU but the virtual machine is assigned four virtual CPUs, where each provide 25% of the available CPU capacity, a single virtual CPU could be removed if the total CPU usage of the host processes increase by 25%.

This solution however would not allow cloud processes to fully utilize the resources not consumed by host processes as blocks of CPU resources are being added or remove at any time. As VirtualBox or any other virtualization technology does not allow maximum resource usage limits to be set during a virtual machine's execution, introducing dynamic resource adjustment is likely to be difficult. Despite this, we envisage that this functionality will become available in the future, either as part of hypervisor or via open source solutions such as those outlined previously. If fully implemented, the *ad hoc* client would only have to interact with the VirtualBox API to achieve this.

Although using CPU hot-plugging is one potential option to achieve this, there are publicly available open-source tools that could also be integrated into the *ad hoc* client. *cpulimit* is a tool that aims to limit the CPU usage of any executing process [11]. The program takes the process name or ID to be limited, as well as the maximum percentage of the CPU the process is allowed to consume. The tool works by frequently suspending and resuming the specified process at appropriate moments by sending SIGSTOP and SIGCONT signals respectively [11] .

By integrating *cpulimit* into our *ad hoc* client, a potentially viable solution can be created. The *ad hoc* client would have to periodically monitor the CPU usage of host processes and dynamically set the maximum levels cloud processes are allowed to consume. For example, if the CPU usage of non-BOINC processes (including *cpulimit*) rises from 55% to 75%, the processes related to VirtualBox as well as the BOINC process would be now limited to consuming 25% of the CPU as opposed to 45% previously. There are however challenges related to how often processes should be 're-limited'. For example, performing this frequently may reduce the interference experienced by host processes but the performance overheads of doing this may be large.

We have outlined some potential solutions that would allow an *ad hoc* client to dynamically adjust the maximum resources cloud processes could consume dependent on the resource usage levels of host processes. As virtualization technologies do not support this, we believe this shows the difficultly of adding this functionality to the hypervisor, however current open source tools may be provide a viable solution if combined and integrated properly. As the development of such a solution is future work, it is important that this dynamic behaviour is achieved not only for CPU resources, but also for memory, disk and network resources.

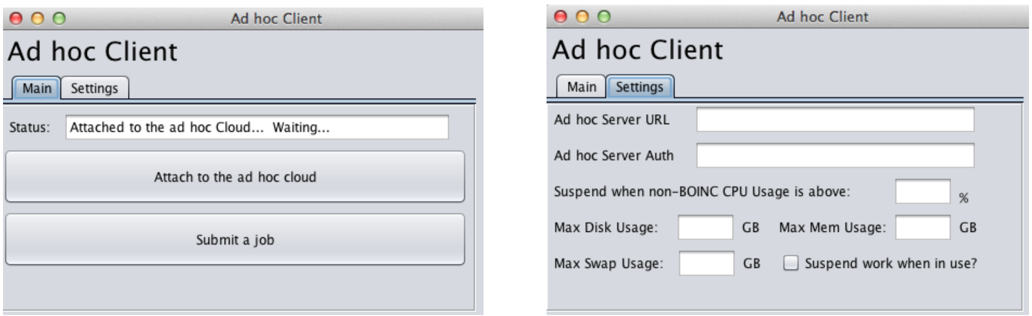
4.8 Installation and Ease of Use

We now discuss how the *ad hoc* cloud computing platform is installed and whether it is easy to use in comparison to installing and using regular BOINC. We first outline the installation and ease of use of the *ad hoc* client followed by the *ad hoc* server.

4.8.1 The *ad hoc* Client

If an *ad hoc* or volunteer host user can install the regular BOINC client, they can install the *ad hoc* client. Installing regular BOINC on UNIX-based hosts involves either installing the BOINC client via a repository (e.g *apt-get install boinc-client*) or downloading a *.zip* or *.sh* file that is either decompressed or executed respectively, to extract the contents of a standalone BOINC client. By opening the BOINC Manager, a volunteer host is able to attach to a BOINC project and interact with BOINC in many simple ways. For example, reset, suspend and abort the project. This shows the simplicity of installing and using BOINC; one of the key reasons why a large number of volunteer hosts currently participate in volunteer projects.

Installing and using the *ad hoc* client is just as easy. As the *ad hoc* client is integrated into a regular BOINC client, the installation process is exactly the same; an *ad hoc* host user or owner can simply download and decompress a *tar.gz* file and begin using the *ad hoc* client instantly. Instead of providing a BOINC Manager, we offer a much simpler GUI interface implemented in Java as shown in Figure 4.14.



Status: Attached to the ad hoc Cloud... Waiting...

(a) (b)

Figure 4.14: The *ad hoc* Client Interface

The *ad hoc* client interface simply offers the *ad hoc* host user a choice of whether to attach to or detach from the *ad hoc* cloud, as shown in Figure 4.14(a); this feature is provided by a single button. This option to control the host's membership is all that is required to allow *ad hoc* host users to add or remove their host from the *ad hoc* cloud; all the aforementioned features of the *ad hoc* cloud are hidden from the user. The *ad hoc* host's state in relation to its contribution to the *ad hoc* cloud is shown via a simple status bar. Furthermore, we have also added a button that links to the job submission portal (see Figure 4.8) to allow *ad hoc* host users to submit cloud jobs if they wish to do so. As previously mentioned, the process of submitting a cloud job to the *ad hoc* cloud is also simple.

Figure 4.14(b) shows a reduced number of settings an *ad hoc* user can set locally, however we provide two almost equivalent GUI settings interfaces for two deployment scenarios. The first deployment scenario is when regular volunteer users join the *ad hoc* cloud. These users are in control of their own host's resources and as such can specify how the *ad hoc* client, as well as the *ad hoc* guest use them. In this scenario, the *ad hoc* host user is able to join the *ad hoc* cloud they wish to join and specifiy basic preferences without having to direct their web browser to their BOINC account stored on the *ad hoc* server.

The second deployment scenario is when an organization who owns the *ad hoc* hosts (i.e. the host owners) wish to create an *ad hoc* cloud from the hosts being used by their employees, for example. In this scenario, it is likely the organization will want to specify how much resources are to be allocated to the *ad hoc* cloud, for example, to perform an analysis to meet an upcoming deadline. The *ad hoc* host owners would therefore like to set values that cannot be changed by the *ad hoc* host users. Therefore we offer a second GUI interface that does not allow values to be entered in the setting fields; these values are set by the *ad hoc* server and are decided by the organization. User-based preferences located on the *ad hoc* server that can also by accessed by *ad hoc* host users are also unmodifiable. In either scenario, using the *ad hoc* client is also just as simple as using the regular BOINC client, if not even easier. It is however possible to modify the BOINC Manager to include such features and we leave it to future work to achieve this.

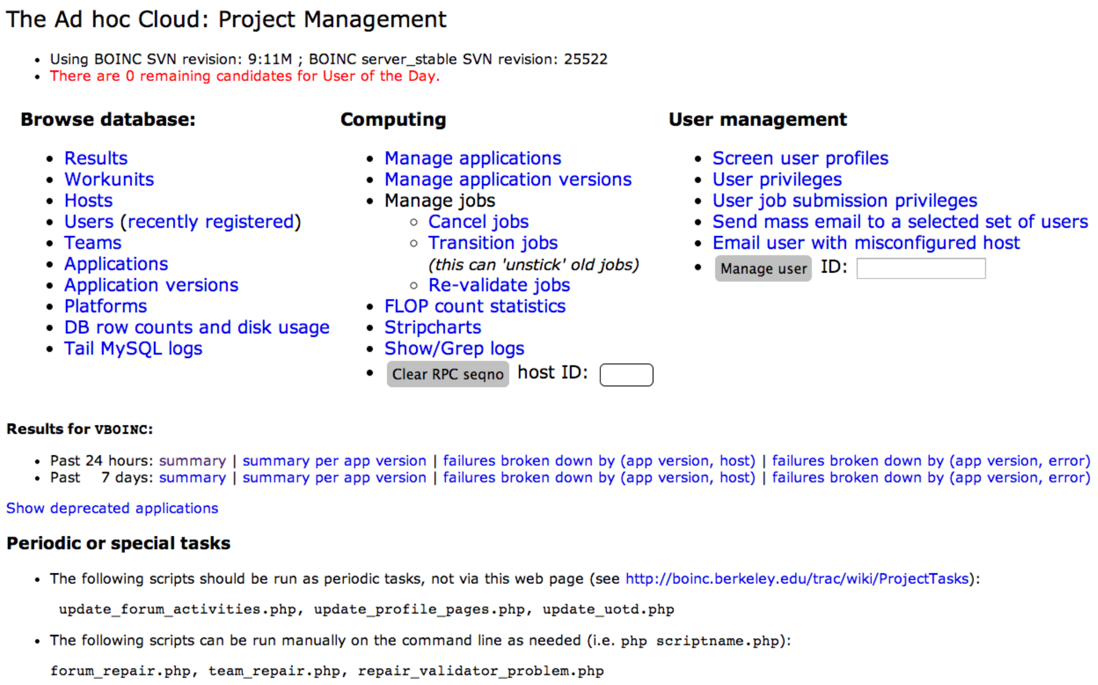
4.8.2 The *ad hoc* Server

The regular BOINC computational model conforms to an architecture where scientific applications are sent from a centrally managed server to volunteer hosts. However the installation of a BOINC server is known to be extremely difficult [66, 187] due to the manual effort required and lack of documentation on the process. We aim to ease the process of installing a BOINC server by providing detailed installation documentation [164].

However the manual effort still required may be difficult for those who are not system administrators. Therefore, to solve this problem and similar to our solution regarding installing a V-BOINC server (see Section 3.4.3 of Chapter 3), we have also created a deployment script named *configure* to automatically perform all of the operations required to successfully install the *ad hoc* cloud server. For example, the script creates both the VM Service and Job Service projects, copies pre-created files to the appropriate locations (e.g. the virtual machine to the BOINC *download* folder, daemons to a *bin/* folder, etc), configures the BOINC daemons and modifies permissions. This process usually takes one minute to complete however this may take longer depending on how long the script takes generating encryption keys. Afterwards, the *ad hoc* server is ready to perform the tasks outlined in this chapter.

As a BOINC, V-BOINC or *ad hoc* server must be installed on a centrally managed host, difficulties may arise when the underlying host fails causing the server to become unavailable. However, due to the design of the BOINC, multiple BOINC servers are able to operate concurrently on multiple hosts and individual server components such as BOINC daemons can also be distributed over multiple hosts. Therefore by replicating and distributing a BOINC server over multiple hosts, issues such as reliability and availability should be of little concern in an *ad hoc* cloud computing infrastructure.

The management of an *ad hoc* server is also easy due to the default web interface BOINC provides with its regular BOINC server. Figure 4.15 shows the BOINC management console.



The Ad hoc Cloud: Project Management

- Using BOINC SVN revision: 9:11M ; BOINC server_stable SVN revision: 25522
- There are 0 remaining candidates for User of the Day.

Browse database:

- Results
- Workunits
- Hosts
- Users (recently registered)
- Teams
- Applications
- Application versions
- Platforms
- DB row counts and disk usage
- Tail MySQL logs

Computing

- Manage applications
- Manage application versions
- Manage jobs
  - Cancel jobs
  - Transition jobs
    (this can 'unstick' old jobs)
  - Re-validate jobs
- FLOP count statistics
- Stripcharts
- Show/Grep logs
- Clear RPC seqno host ID:

User management

- Screen user profiles
- User privileges
- User job submission privileges
- Send mass email to a selected set of users
- Email user with misconfigured host
- Manage user ID:

Results for `VBOINC`:

- Past 24 hours: summary | summary per app version | failures broken down by (app version, host) | failures broken down by (app version, error)
- Past 7 days: summary | summary per app version | failures broken down by (app version, host) | failures broken down by (app version, error)

Show deprecated applications

Periodic or special tasks

- The following scripts should be run as periodic tasks, not via this web page (see http://boinc.berkeley.edu/trac/wiki/ProjectTasks):

  `update_forum_activities.php, update_profile_pages.php, update_uotd.php`

- The following scripts can be run manually on the command line as needed (i.e. `php scriptname.php`):

  `forum_repair.php, team_repair.php, repair_validator_problem.php`

Figure 4.15: The *ad hoc* Server Dashboard

This console allows the *ad hoc* server administrator to view the current state of the VM Service and Job Service databases as well view and manage the state of each cloud job. Furthermore, an *ad hoc* server administrator is also able to view and manage all *ad hoc* hosts within the cloud; for example block specific *ad hoc* hosts due to their malicious actions. As previously mentioned, we assume that cloud jobs produce an output file which is then returned to the *ad hoc* server after successful completion. We make this output file viewable and downloadable to the cloud user through the server's web interface. Therefore by making the installation of the server relatively simple and adopting BOINC's current web interfaces for server management, deploying and using an *ad hoc* server is now easy.

4.9 Summary

In this chapter we have outlined how to transform our virtualized volunteer computing infrastructure V-BOINC into an *ad hoc* cloud platform.

Firstly we gave a literature review of the current state of research in the *ad hoc* cloud computing field. Two exemplary research projects were shown to be the major enablers of this new computational model by viewing the concept in two different lights and proposing an approach to each. Research from the mobile device cloud field showed promising results. However with the exception of the major enablers of *ad hoc* cloud computing-like platforms, studies that focussed on merging cloud computing and volunteer computing showed less promising results despite being a popular research topic for the last five years. We believe this is because many research projects either fail to identify or address key issues related to *ad hoc* cloud computing. Other than the work we have presented in this chapter, little technical realization and evaluation has so far been reported. This could be attributed to the difficulty of integrating volunteer systems with features taken from the cloud computing field.

We then gave an architectural overview of the *ad hoc* server and *ad hoc* client and the interactions that exist between them. We also gave an overview of the processes involved from job submission, to cloud job execution and retrieval of results. We showed how cloud users are able to easily submit cloud jobs to BOINC using our submission system, especially in comparison with other studies that would introduce unnecessary overheads when performing such a simple task. Next we focussed on how a cloud job is scheduled to a near-optimal *ad hoc* host based on host availability, hardware specifications, resource load and reliability, with the latter being the dominant factor in selecting a host. We then described how the chosen *ad hoc* host is prepared to receive the cloud job and indeed how the job is executed.

This was followed by describing the implementation of our major contribution of making an unreliable infrastructure reliable. We then outlined our P2P reliability algorithm that periodically takes and distributes virtual machine checkpoints to other *ad hoc* hosts in the *ad hoc* cloud. Similar to cloud job scheduling, checkpoint scheduling was also primarily based on the reliability of other *ad hoc* hosts. In order to achieve a reliable system, we outlined our methods of how to restore virtual machine checkpoints on other *ad hoc* hosts when an *ad hoc* guest fails or is believed to have failed. Similarly, checkpoint scheduling also was key in determining the near-optimal *ad hoc* host to restore a checkpoint on.

Penultimately, we described a possible method intended to limit the interference of host processes caused by executing *ad hoc* clients and *ad hoc* guests. We have postulated that this feature may be based on dynamically adjusting the number of virtual CPUs assigned to the *ad hoc* guest or through the use of external tools, both of which are based on the current load of executing host processes. Although a necessity in an *ad hoc* cloud, we believe either our chosen virtualization technology VirtualBox will implement this feature in the near future or open-source tools will become available to work towards a solution to this problem.

Finally, we showed that like the BOINC client, the *ad hoc* client is extremely easy to install and use, in turn potentially allowing those who are not technically skilled to donate their resources to the *ad hoc* cloud. In a similar fashion, we showed that the *ad hoc* server, unlike the BOINC server, is easy to install due to our single deployment script that can create a ready-to-use *ad hoc* server in a matter of minutes.

By outlining how to transform V-BOINC into an *ad hoc* cloud computing platform, we have provided solutions to many of the research challenges outlined in Section 1.2.1 of Chapter 1.

Chapter 5

Monitoring and Controlling Dynamic *ad hoc* Infrastructures

5.1 Introduction

In this chapter, we discuss how it is possible to monitor and manage dynamic groups of hosts where hosts frequently migrate between groups or are members of multiple groups; these dynamic groups are synonymous to the cloudlet as part of the *ad hoc* cloud. However this chapter may be skipped at the first reading if the reader only wishes to focus on the core concepts of the *ad hoc* cloud.

Monitoring dynamic groups of hosts is useful in a number of settings. Monitoring the state of each cloudlet in an *ad hoc* cloud could help to make a number of scheduling decisions such as determining the set of near-optimal *ad hoc* hosts that should execute specific applications, e.g. SPRINT applications, on specific *ad hoc* guests. Cloudlet-based monitoring could also be used to shape the available resources of *ad hoc* hosts to the resource demands of each cloudlet application as well as determine when to move hosts from underutilized cloudlets to those that are overloaded.

Group-based monitoring is not only useful in *ad hoc* clouds but also for those who need to monitor and manage dynamic groups of hosts in other computational infrastructure such as Grids, clusters and clouds. For instance, an organization may employ server clustering to ensure high availability, scalability and easier management of their infrastructure. Server clustering is the grouping of a set of hosts based on administrator-defined characteristics, for example, servers may be clustered according to the service they provide. In order to optimize performance, manage load and maintain availability, hosts may migrate from one group to another.

As group-based monitoring is useful in a number of scenarios other than the *ad hoc* cloud, we refer to these dynamic groups of hosts as cloudlets regardless of whether group-based monitoring is used in the *ad hoc* cloud or other computational infrastructures. Subsequently, we also do not make a direct reference to the *ad hoc* cloud but we note that a host and server are synonymous with an *ad hoc* host and *ad hoc* server respectively. Furthermore, an infrastructure administrator is synonymous with an administrator of the *ad hoc* cloud.

Throughout this thesis we have assumed that cloudlet-based monitoring is a trivial task in relation to the frequent migration of hosts between cloudlets or as those that are part of many cloudlets, however many challenges exist when monitoring cloudlets. Typically monitoring tools are statically configured hence any change of a host's membership, requires an administrator of the infrastructure to:

- manually change the monitoring tool's configuration file to specify the new cloudlet the host belongs to. System configuration tools could be used, however this increases the complexity and effort needed to monitor cloudlets,

- restart the monitoring processes upon the host being monitored. This is required for the host to adopt the changes. In the event configuration files reside on a remote server, the following action applies,

- restart the data collection process (located on a central server); a process that may take several minutes for the changes to take effect. This therefore restricts the use of the monitoring tool during this period.

Within a large dynamic infrastructure where hosts frequently migrate between cloudlets, the manual effort to perform these tasks upon each membership change would be significant. Many monitoring tools exist for hosts that need little or no configuration changes over a host's lifetime, however none are designed to monitor dynamically changing groups of hosts.

To solve the aforementioned problems, we introduce the Cloudlet Control and Monitoring System (C2MS) [166] which extends Ganglia to allow infrastructure administrators to create cloudlets to be monitored and managed and hosts to be migrated between cloudlets. Furthermore, the C2MS prevents the re-configuration of servers and restart of monitoring processes when hosts migrate between cloudlets. Infrastructure administrators are then able to define and monitor the overall state of cloudlets independently without explicitly reconfiguring and restarting the monitoring tool, in turn making it easy to view monitoring data of cloudlets. The C2MS is an innovation that overcomes the time-consuming limitations of previous monitoring tools in turn freeing administrators of large-scale systems to focus on operational challenges with improved information. We have also made a series of extensions/improvements to Ganglia with the aim of making infrastructure monitoring and management easier for administrators. We introduce:

- further metrics than those provided by Ganglia that are commonly monitored nowadays; these are power usage and CPU temperature monitoring,
- a management element on top of Ganglia to give administrators the ability to quickly take control of individual hosts or entire cloudlets by issuing administrator commands. This may be used for ensuring *ad hoc* hosts behave as expected, upgrading existing software (e.g. the *ad hoc* client) or installing new software over many hosts, for example.

A large number of server monitoring and management tools exist independently, however very few provide both of these functions. As such, we also reduce the effort required for installation and maintenance of these independent packages by combining these features into a single tool. The C2MS can be used on a number of infrastructures such as clusters, clouds and Grids and is available to download online at [8]. The aforementioned features and the implementation of this tool were primarily undertaken by Íñigo Goiri, Josep Ruis and I [166].

In this chapter, we discuss how the C2MS offers the aforementioned features in detail and how they are implemented. We first discuss related research showing how current monitoring tools are unable to monitor dynamic groups of hosts. This is followed by a system overview of our tool. We then describe how the C2MS is implemented to solve the aforementioned problems as well as introduce power usage and CPU temperature monitoring and infrastructure management. Finally we evaluate the C2MS in comparison with Ganglia and how well it can manage large-scale infrastructures.

5.2 Related Work

The number of system monitoring and management tools are plentiful however none are able to monitor dynamically changing groups of hosts without the need for explicit manual reconfiguration upon any group membership changes. We outline some of the current leading monitoring and management tools in the field while paying specific attention to the different features offered by our work.

The C2MS uses Ganglia as its foundation for infrastructure monitoring due to its popularity, easy installation process, easy to use web interface and its extensibility. Ganglia monitors different groups of hosts, which Ganglia terms ‘clusters’, by allowing the infrastructure administrator to define the cluster name within the host’s configuration file; see Section 2.5.2 of Chapter 2 for more information. This allows Ganglia’s PHP web interface located on a central server to display the aggregated data for this group. Any host changing to an alternative cluster requires manual reconfiguration and the restart of the *gmond* and *gmetad* daemons on the host and central server respectively. The C2MS offers an abstract layer built on top of Ganglia allowing group membership changes without the need for reconfiguration or daemon restarts upon any host.

The monitoring tool Nagios does however allow groups of static hosts to be monitored but this is only to simplify the configuration of these hosts and to make navigation via the Nagios GUI easier; this is not for monitoring dynamic groups of hosts. Nagios is also configured by modifying a number of configuration files, however these are located on a central server rather than on the remote hosts like Ganglia. The configuration files define all hosts to be monitored as well as the operations to be performed, for example, check availability, monitor host resource usage, etc. As a result, any modifications to the configuration files requires Nagios to be restarted. Therefore the statically configured Nagios makes monitoring sporadically available hosts within a cloudlet difficult; the C2MS provides this functionality.

Wright *et al.* outline their view of a dynamic cloud management and monitoring system tailored towards services, e.g. a web server [220]. The authors note the lack of dynamic tools for cloud environments; others have also noted the lack of tools for rapidly changing environments, particularly for cloud environments [153, 213]. Wright *et al.* therefore have created the Cloud Management System (CMS). Their CMS is similar to the C2MS and Ganglia in many ways, however their implementation monitors services running on cloud instances rather than the instances themselves. Furthermore, the CMS is only a proposal of a potential monitoring service and no prototype or implemented system exist yet to our knowledge.

Birman *et al* outline their distributed self-configuring monitoring and adaptation tool called Astrolabe [65]. Astrolabe works like any other monitoring tool by observing the state of an infrastructure where the tool is installed. However it differs by essentially creating a virtual system-wide hierarchical relational database based on a peer-to-peer protocol meaning no central server needs to exist to collect monitoring data. By performing distributed data analysis, Astrolabe can create performance summaries of *zones* — machines typically grouped based on the shortest latency between pairs or simply an administrator-specified group — by data aggregation; a method we use to create graphs of cloudlets.

A major advantage of Astrolabe is its ability to adapt to configuration changes without the need of restarting the monitoring tool, however administrator-specified groups and the resources to be monitored need to be manually configured in their *configuration certificate* file; an infeasible task for a large dynamically changing cloud infrastructure.

5.3 System Overview

The C2MS consists of three major components: the *Monitoring*, *Cloudlet Creator* and *Control* components shown in Figure 5.1. These components relate to the web pages that an infrastructure administrator can navigate to in order to exploit the functionality of the C2MS.

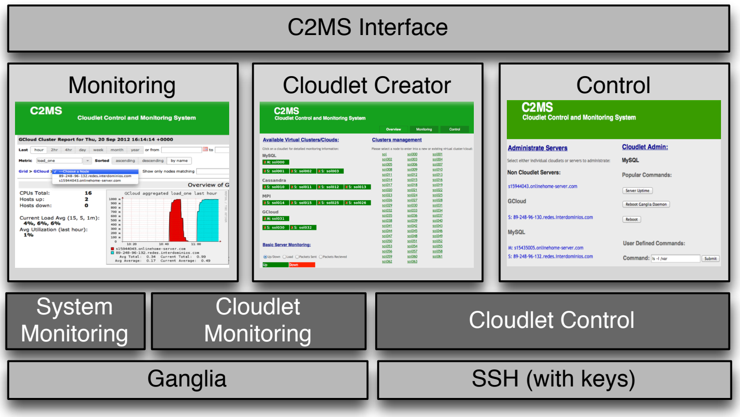
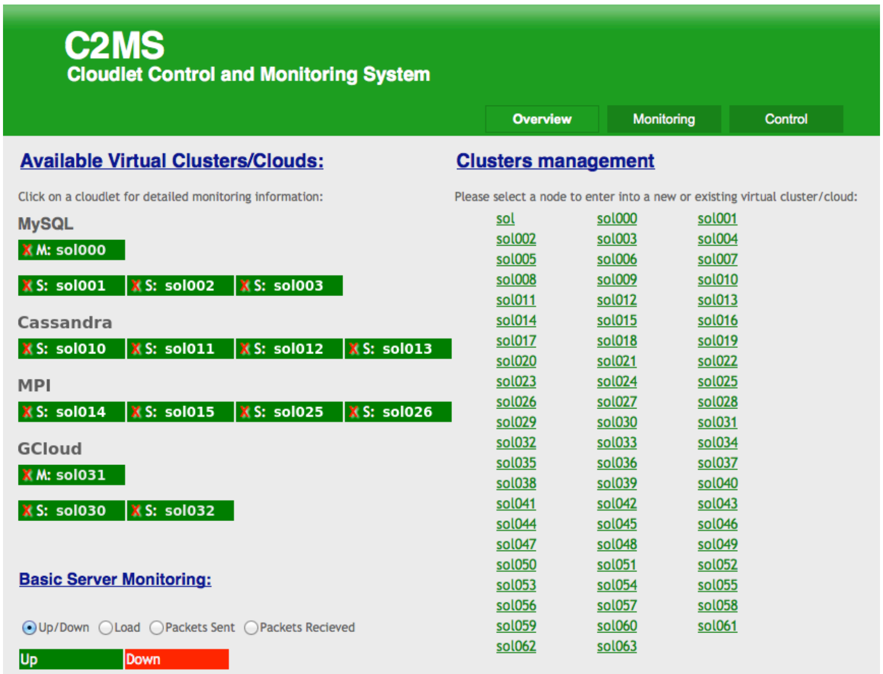


Figure 5.1: The C2MS Architecture

The Monitoring component is a modified version of Ganglia that allows individual, cloudlet or entire system monitoring, where the former and latter are provided by Ganglia by default. The Control component gives administrators the ability to control either single hosts or entire cloudlets via SSH. Both components use the output from the Cloudlet Creator; a component allowing administrators to create cloudlets to be monitored. For example, an administrator of the *ad hoc* cloud could create three cloudlets to monitor *ad hoc* hosts that execute MySQL, BLAST and Matlab applications respectively. In order to interact with these components, the C2MS interface displaying the three tabs Overview, Monitoring and Control are shown in Figure 5.2.



C2MS
Cloudlet Control and Monitoring System

Available Virtual Clusters/Clouds:

Click on a cloudlet for detailed monitoring information:

MySQL
X M: sol000
X S: sol001 X S: sol002 X S: sol003

Cassandra
X S: sol010 X S: sol011 X S: sol012 X S: sol013

MPI
X S: sol014 X S: sol015 X S: sol025 X S: sol026

GCloud
X M: sol031
X S: sol030 X S: sol032

Basic Server Monitoring:

● Up/Down ○ Load ○ Packets Sent ○ Packets Recieved

Up Down

Clusters management

Please select a node to enter into a new or existing virtual cluster/cloud:

sol sol000 sol001
sol002 sol003 sol004
sol005 sol006 sol007
sol008 sol009 sol010
sol011 sol012 sol013
sol014 sol015 sol016
sol017 sol018 sol019
sol020 sol021 sol022
sol023 sol024 sol025
sol026 sol027 sol028
sol029 sol030 sol031
sol032 sol033 sol034
sol035 sol036 sol037
sol038 sol039 sol040
sol041 sol042 sol043
sol044 sol045 sol046
sol047 sol048 sol049
sol050 sol051 sol052
sol053 sol054 sol055
sol056 sol057 sol058
sol059 sol060 sol061
sol062 sol063

Figure 5.2: C2MS PHP Interface Overview Page

The Overview tab displays all hosts in the system that have the *gmond* daemon running and allows cloudlets to be created for combined monitoring and control. Administrators can create cloudlets by selecting a host from the list of all those available (right) and entering the desired cloudlet name on a pop-up text field; hosts are added to existing cloudlets in the same way however the cloudlet name can be selected. Figure 5.2 shows a number of example cloudlets such as MySQL and MPI cloudlets each with four member hosts. The administrator is then able to view cloudlet or system specific graphs by clicking on the cloudlet name or via the Monitoring tab.

To remove a host from a cloudlet, an administrator is required to click on the ‘X’ marked besides the host name. Entire cloudlets can also be deleted or member hosts can be migrated from one cloudlet to another. Furthermore, administrators are able to view basic monitoring characteristics (bottom left) showing whether each host is up/down and the CPU and network load based on the check button selected; this information is displayed via color-coding the hosts.

The Control tab provides a similar page, that is shown and explained in detail later in this chapter, allowing administrator-defined commands to be executed either on individual hosts or over entire cloudlets. Typically, Ganglia allows public users to view a host's resource usage data, however because this tool is intended for private use (i.e., administrators only) due to the Cloudlet Creator and Control components, we provide a login page with changeable credentials to prevent public users accessing the system and performing malicious tasks. We leave it to the administrator to provide additional security measures if required.

5.4 Implementation

We now explore how the functionalities behind the three interface components are implemented and integrated to create the C2MS.

5.4.1 Creating Cloudlets

In order to monitor and view the state of an entire cloudlet, the C2MS must be aware that a cloudlet exists and which hosts belong to the cloudlet. Upon installing and configuring Ganglia, an infrastructure administrator simply needs to modify each host's *gmond* configuration file to include the hostname or IP address of the Ganglia interface located on a central server and specify the Ganglia cluster as *Initial*. This allows the *gmetad* daemon running on a central server to receive monitoring data that is perceived to be from a single group of hosts.

Upon receiving this data, the C2MS will register that each of the monitored hosts are present and display them to the user as shown in the right hand side of Figure 5.2; as hosts enter and leave the Ganglia monitoring system, the C2MS dynamically adjusts those that are available. By giving each host the same Ganglia cluster name, we can virtually partition this group of *Initial* hosts at a higher level to allow cloudlets to be created. Details of how to configure Ganglia can obtained from [19] and instructions on how to setup the C2MS are presented in the C2MS downloadable [8].

When an administrator creates a cloudlet via the C2MS interface, the host and cloudlet name specified is recorded in a file named *clusters* within the */etc/ganglia/* folder; this file contains a list of cloudlets and their member hosts. We use this file to record cloudlet membership changes to minimize the additions made to Gangalia as well as minimize overheads associated of performing this simple a task. For example, a MySQL database could be integrated into Ganglia, however the effort and computational overhead would be unnecessary.

The C2MS interface then displays cloudlets by reading and parsing the *clusters* file. However at this point, Ganglia will not be able to display cloudlet-based monitoring data as it is unaware a cloudlet or a number of them exist. To enable cloudlet based monitoring, Ganglia requires that each Ganglia cloudlet has a folder present in */var/lib/ganglia/rrds/.* This cloudlet folder contains directories for each of the cloudlet's member hosts which themselves contain monitoring data (*.rrd* files) for the host as discussed in Section 2.5.2 of Chapter 2.

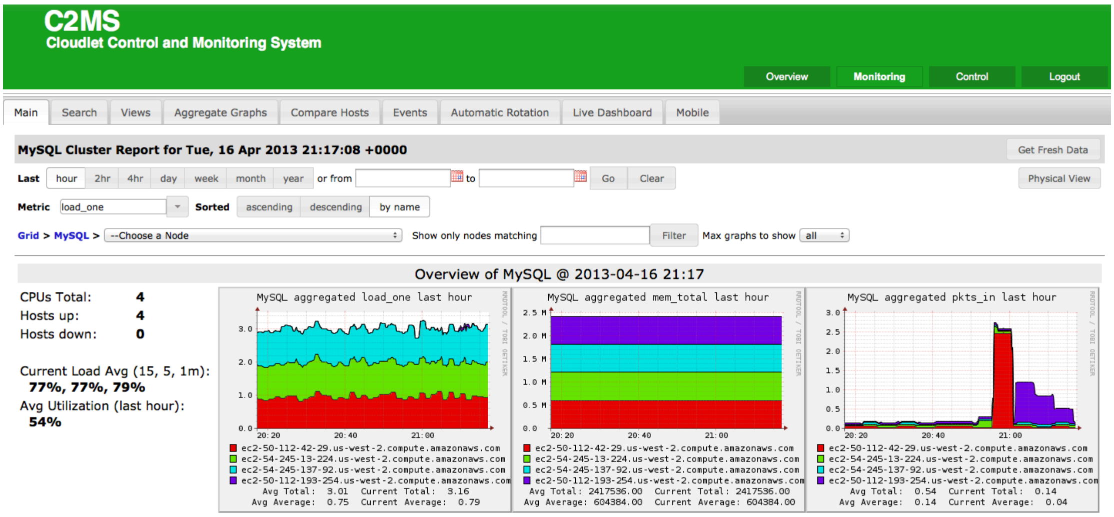
Upon cloudlet creation, the C2MS creates the appropriate cloudlet folder within the */var/lib/ganglia/rrds/* directory and links to the original *.rrd* files of each host within the *Initial* folder created by Ganglia. We therefore do not need to replicate any data, which would in turn introduce overheads. Hence with the creation of a new cloudlet, Ganglia is lead to believe that it has received monitoring data from a new cluster which contains the hosts listed in the */var/lib/ganglia/rrds/cloudlet_name* directory.

In the event of cloudlet creation, deletion or a change of a host's cloudlet membership from one to another, only modifications to configuration files linked to the C2MS and the cloudlet folders within */var/lib/ganglia/rrds/* are needed; this allows us to avoid restarting the Ganglia daemons upon any changes. For example, if an *ad hoc* host migrates from one cloudlet to another, the C2MS modifies the */etc/ganglia/clusters* file to reflect the changes on the C2MS interface. Symbolic links are then created from the new cloudlet directory in */var/lib/ganglia/rrds/cloudlet_name* to the original host data present in */var/lib/ganglia/rrds/Initial/host_name*. These configuration changes are obscured from the administrator and are automatically performed by the C2MS interface.

5.4.2 Monitoring Cloudlets

The information we are interested in displaying to the administrator is the entire state of multiple cloudlets via summary graphs for each Ganglia metric; this data can also be fed into the *ad hoc* Scheduler (described in Section 4.5 of Chapter 4). Each page displaying monitoring data of a cloudlet allows users to either view a summary of the current cloudlet state or select individual hosts to examine their resource usage in more detail. Figure 5.3 shows both these features which are inherited from Ganglia.

Firstly, we see that four hosts exist within the 'MySQL' cloudlet, both from the number of 'hosts up' and the total of CPUs. The graphs shown are only specific to the 'MySQL' cloudlet with colours making the distinction between individual hosts



MySQL Cluster Report for Tue, 16 Apr 2013 21:17:08 +0000

Metric: load_one

Max graphs to show: all

Overview of MySQL @ 2013-04-16 21:17

CPUs Total: 4
Hosts up: 4
Hosts down: 0

Current Load Avg (15, 5, 1m):
77%, 77%, 79%
Avg Utilization (last hour):
54%

```
MySQL aggregated load_one last hour
Avg Total:    3.01   Current Total:    3.16
Avg Average:  0.75   Current Average:  0.79

MySQL aggregated mem_total last hour
Avg Total:  2417536.00   Current Total:  2417536.00
Avg Average: 604384.00   Current Average: 604384.00

MySQL aggregated pkts_in last hour
Avg Total:    0.54   Current Total:    0.14
Avg Average:  0.14   Current Average:  0.04
```

Figure 5.3: Monitoring Data of a Cloudlet

present in the cloudlet. To create cloudlet summary graphs, data aggregation is used and this is apparent in the graphs above, where data from one host is stacked upon another, in turn displaying the total resource use for the selected cloudlet; different cloudlets can be selected via the ‘Overview’ page of Figure 5.2.

The depicted graphs automatically change when hosts are added to or removed from the cloudlet. To create aggregated graphs dynamically, Ganglia calls its */var/www/ganglia-web/stacked.php* file when the page is viewed; a default file of the Ganglia implementation. We have modified this file to only create stacked graphs for hosts present in a cloudlet rather than an entire system as regular Ganglia would do; the same has been applied to the number of ‘hosts up’, ‘hosts down’, and ‘CPUs Total’. The PHP file returns PNG files of the created graphs and these are displayed via the C2MS interface.

Graph data aggregation can be easily achieved through the use of RRDtool. We implement this through PHP calls to RRDtool, however this can be easily explained by the use of *rrdtool*’s graph function shown in Figure 5.4.

```
rrdtool graph agg_graph.png --imgformat=PNG
DEF:one=host1_metric.rrd:sum:AVERAGE
AREA:one#00CF06::STACK
DEF:two=host2_metric.rrd:sum:AVERAGE
AREA:two#CC0000::STACK --start timeX --end timeY
```

Figure 5.4: Stacked Graphs using RRDtool

First we define variables, one for each of the host's *.rrd* files to be aggregated (e.g. *one* and *two*). The data *sum* is then plotted using the average utilization for each 15 second period, as performed by Ganglia by default. We use the AREA shape to plot the variable values with different colours and in the form of a STACK, where one dataset is placed on top of another. We also enter a start and end time specified by the administrator via the C2MS interface to allow historical cloudlet monitoring data to be accessed. Other arguments are omitted here for clarity that relate to the appearance of graphs such as the width, height and labels.

5.4.3 Additional Metrics

The C2MS not only measures basic resource usage such as CPU, memory, etc, but by installing additional modules, one can also monitor power consumption and temperature. Monitoring temperature requires a host's CPU(s) to possess built-in temperature monitoring capabilities such as those found in Intel Core based processors and others [185]. The C2MS collects temperature data by adding a monitoring module to the *gmond* daemon of every host, which periodically polls the CPU's Digital Thermal Sensor to obtain temperature data. This data is then available to the *gmetad* daemon which in turn can display this information. Similarly, this information can be aggregated to show data for single cloudlets or for single hosts. Figure 5.5 shows one other method we use for displaying this data where servers are presented as a heat map in the rack format.

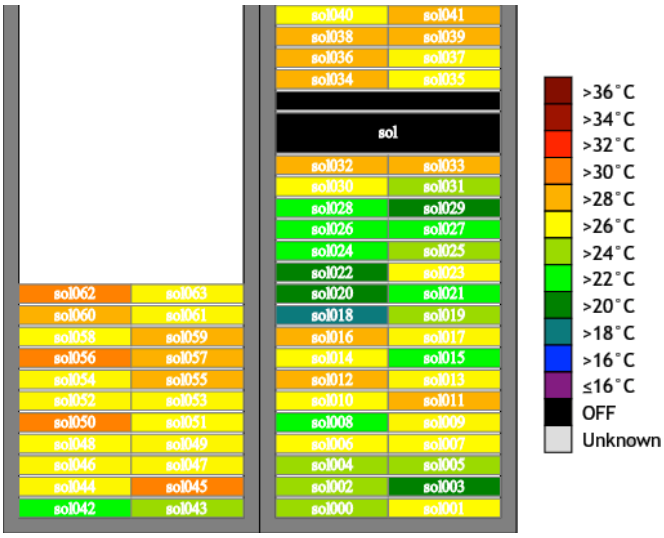
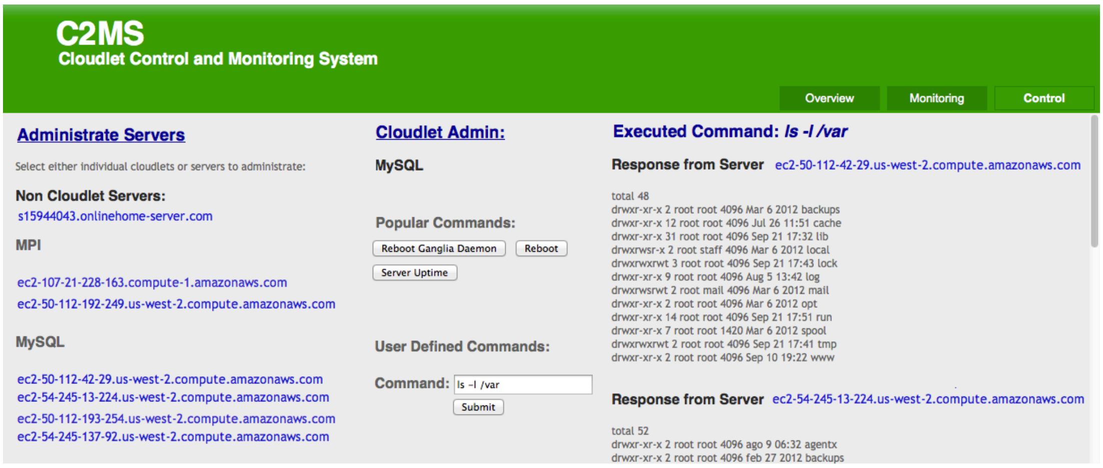


Figure 5.5: CPU Temperature Data Output

Similar to temperature monitoring, power observation requires the appropriate power monitoring hardware or a Power Distribution Unit (PDU). The data recorded by the PDU is then periodically queried and stored in RRD files following the Ganglia RRD structure. These are then exported to graphs and added to the Ganglia interface for viewing. We also allow power consumption to be monitored for cloudlets and the data is also exported using graph aggregation. To distinguish power usage for hosts connected to the same PDU, the administrator must identify each PDU and its connected hosts in a file accessed by the C2MS. These details include the host name, the MAC address, the PDU identifier and the outlet the host is connected to. In the context of an *ad hoc* cloud, it is highly unlikely PDUs will be available and connected to *ad hoc* hosts. Therefore this functionality is reserved for monitoring cloudlets in dedicated cloud, cluster and Grid environments.

5.4.4 Controlling Cloudlets

Our final contribution incorporates a host management component into the C2MS. Administrators are not only able to control individual hosts but can issue specified instructions over cloudlets.



Administrate Servers

Select either individual cloudlets or servers to administrate:

Non Cloudlet Servers:
s15944043.onlinehome-server.com

MPI

ec2-107-21-228-163.compute-1.amazonaws.com
ec2-50-112-192-249.us-west-2.compute.amazonaws.com

MySQL

ec2-50-112-42-29.us-west-2.compute.amazonaws.com
ec2-54-245-13-224.us-west-2.compute.amazonaws.com
ec2-50-112-193-254.us-west-2.compute.amazonaws.com
ec2-54-245-137-92.us-west-2.compute.amazonaws.com

Cloudlet Admin:

MySQL

Popular Commands:

User Defined Commands:

Command: ls –l /var

Executed Command: *ls -l /var*

Response from Server ec2-50-112-42-29.us-west-2.compute.amazonaws.com

```
total 48
drwxr-xr-x 2 root root 4096 Mar 6 2012 backups
drwxr-xr-x 12 root root 4096 Jul 26 11:51 cache
drwxr-xr-x 31 root root 4096 Sep 21 17:32 lib
drwxrwsr-x 2 root staff 4096 Mar 6 2012 local
drwxrwxrwt 3 root root 4096 Sep 21 17:43 lock
drwxr-xr-x 9 root root 4096 Aug 5 13:42 log
drwxrwsrwt 2 root mail 4096 Mar 6 2012 mail
drwxr-xr-x 2 root root 4096 Mar 6 2012 opt
drwxr-xr-x 14 root root 4096 Sep 21 17:51 run
drwxr-xr-x 7 root root 1420 Mar 6 2012 spool
drwxrwxrwt 2 root root 4096 Sep 21 17:41 tmp
drwxr-xr-x 2 root root 4096 Sep 10 19:22 www
```

Response from Server ec2-54-245-13-224.us-west-2.compute.amazonaws.com

```
total 52
drwxr-xr-x 2 root root 4096 ago 9 06:32 agentx
drwxr-xr-x 2 root root 4096 feb 27 2012 backups
```

Figure 5.6: Controlling a Cloudlet

Figure 5.6 shows the members of each cloudlet (left) and by selecting the cloudlet name, an administrator is able to issue a pre-populated or self-defined (middle) command over the set of hosts. The results of command execution of each host are also shown (right). In order to introduce control functionality, we investigated a number of popular tools to determine whether they satisfied our requirements. Such a tool must:

1. allow the grouping of hosts and concurrent command execution upon these groups,
2. not require the installation of software on remote hosts within cloudlets,
3. be easy to integrate into the C2MS.

The tools we investigated were: Webmin, Capistrano and *cexec* (see Section 2.5 of Chapter 2). Webmin allows the grouping of hosts into cloudlets and commands to be executed per-cloudlet. However Webmin requires the installation of software on remote hosts and the integration process of Webmin into the C2MS would not be simple as the underlying core of Webmin would have to be modified. For example, the creation of a cloudlet via the C2MS interface would have to be reflected in the GUI interface of Webmin to avoid administrators creating a cloudlet twice on both interfaces. Capistrano also allows the grouping of hosts by simply specifying these groups in their configuration *capfile*. This file can be easily accessed and modified by the C2MS. Furthermore, Capistrano does not require any installation of software on remote hosts due to its use of SSH keys between hosts.

Finally, *cexec* is also able to execute commands over a set of hosts. *cexec* requires that a configuration file exists listing the hostname or IP address of the hosts in a cloudlet alongside the cloudlet name; multiple cloudlets can exist allowing the administrator to specify the cloudlet to execute the command over. Like Capistrano, we can automatically generate this file by entering the hostnames of the cloudlet members, taken from the */etc/ganglia/clusters* file, into *cexec*'s configuration file, making integration into the C2MS easy. Furthermore, *cexec* does not require any software installation on target hosts. The C2MS currently uses *cexec* as its control component. This is based on the simplicity of the tool as well as its performance as explored in the following section.

5.5 Evaluation

Ganglia is commonly used in the HPC and Grid communities where clusters, like cloud infrastructures and potentially *ad hoc* clouds, typically contain a large number of hosts. We now investigate how effectively the C2MS can monitor such systems by determining whether our implementation introduces any additional overhead above that already introduced by regular Ganglia. We then determine the optimal method of host management and if the C2MS can execute administrator commands over a large number of machines quickly. We perform these experiments on Amazon EC2 with the C2MS interface running on a Large Ubuntu 12.04 instance and hosts running on Micro instances of the same type.

5.5.1 Monitoring Performance

Ganglia is well known for its scalable implementation hence the modifications we have made must also be able to cope with an increase in the number of hosts. We use at most 130 hosts; the maximum number of instances we could instantiate on Amazon EC2. First we tested whether our method of graph aggregation and the operations that underpin it introduce any overheads when compared with regular Ganglia. To test this, we split the experiment into two parts: one to record the page load times of both systems and another to determine the impact on the Apache server displaying the data via the Ganglia and C2MS interfaces.

5.5.1.1 Page Load Times

We first explore whether viewing a cloudlet's monitoring output via the C2MS takes additional time to load when compared with regular Ganglia. For example, if we view the monitoring output of a 50 host regular Ganglia cluster, does the C2MS introduce any overhead when we view a cloudlet of the same size? We compare the page load times of an increasing Ganglia cluster and C2MS cloudlet size. By adding a simple PHP page load counter to the page displaying monitoring data, loading each page 15 times and taking the average value, we obtain the results shown in Figure 5.7

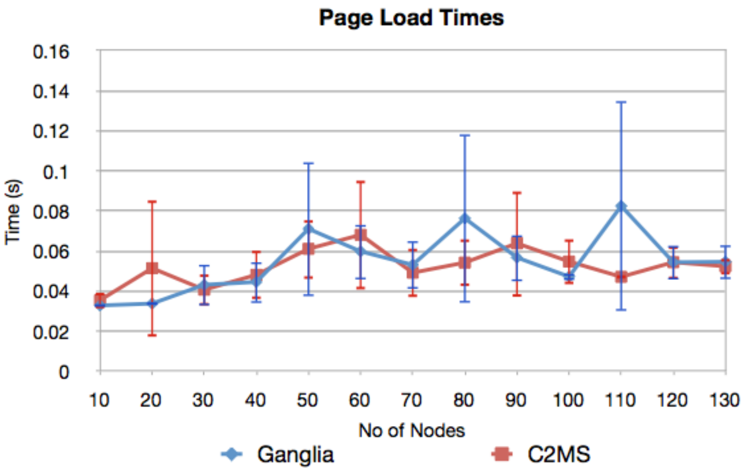


Figure 5.7: Page Load Time Comparison between Ganglia and the C2MS

We initially see that the page load times are small using both Ganglia (blue) and the C2MS (red) with data being displayed in a matter of milliseconds. By using 95% confidence intervals, we see that any system can potentially produce equal page load times. Variations of recorded page load times can be seen, however this is expected due to the unit we measure in, i.e. milliseconds. We also see that as the number of hosts increase in the cloudlet, the page load times increase slightly, however with the exception of the recorded times at 110 servers when using Ganglia. This can be attributed to host performance variation as the C2MS data point at 20 nodes also provides a larger variation than expected. We see that no major time differences of page loads exist between the two tools and this can be attributed to avoiding data replication upon cloudlet creation and linking to original host's data as well as the low overhead of our additional code to achieve cloudlet-based graph aggregation.

5.5.1.2 Apache Load

Secondly we determine if viewing a cloudlet's monitoring output via the C2MS places additional load on the Apache server displaying the data in comparison to regular Ganglia. Upon loading a page 5 times, we recorded the total CPU load placed on the Apache server; this is performed three times (i.e, 15 page loads in total) for each cloudlet size and the average value is taken. We used Apache's Server Status module to obtain the data values recorded [133]. Figure 5.8 shows our results.

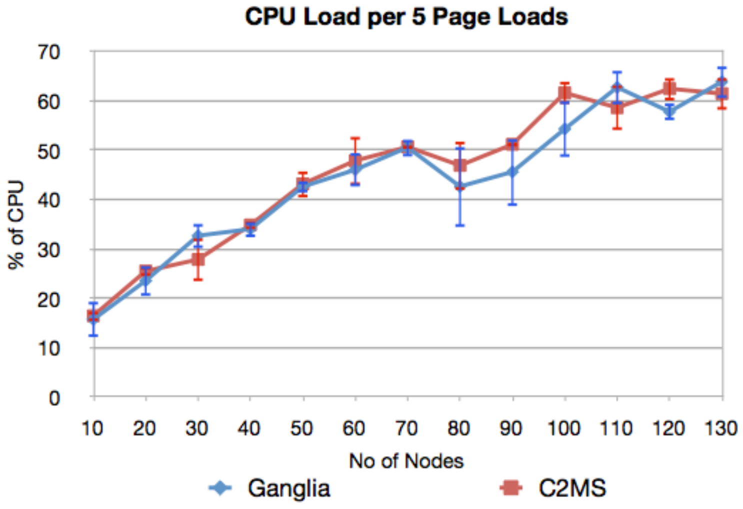


Figure 5.8: Apache Load Comparison between Ganglia and the C2MS

As expected, with a greater amount of data to display, a greater percentage of the CPU is used. Both tools show similar results until 80 nodes, after which the C2MS uses slightly more resources after a dip in utilization. The average results of the two tools then oscillate from 100 nodes onwards where the differences are then negligible. We also display 95% confidence intervals to show that in most cases, readings from both tool executions will provide similar results. Hence the overhead introduced by the C2MS and our modified version of the */var/www/ganglia-web/stacked.php* file is in most cases is negligible. Hence administrators familiar with Ganglia should see no additional latencies when using the C2MS on relatively large cloud infrastructures. As such, the performance differences between the C2MS and other monitoring tools will be similar to the differences between Ganglia and these tools, therefore we need not conduct a performance comparison between the C2MS and other monitoring tools.

5.5.2 Control Performance

We now investigate whther the C2MS can execute administrator-specified commands quickly over a large set of hosts. Currently two versions of the control component are available: serial and parallel SSH command execution. The cluster management tools Capistrano, *cexec* and Webmin were tested for providing concurrent command execution functionality. We expect the parallel version to outperform serial execution but which management tool offers the best performance? We investigated the time taken for parallel tools to execute a simple *uptime* command over an increasing number of hosts. Each command is run 5 times per method and the results are averaged as shown in Figure 5.9.

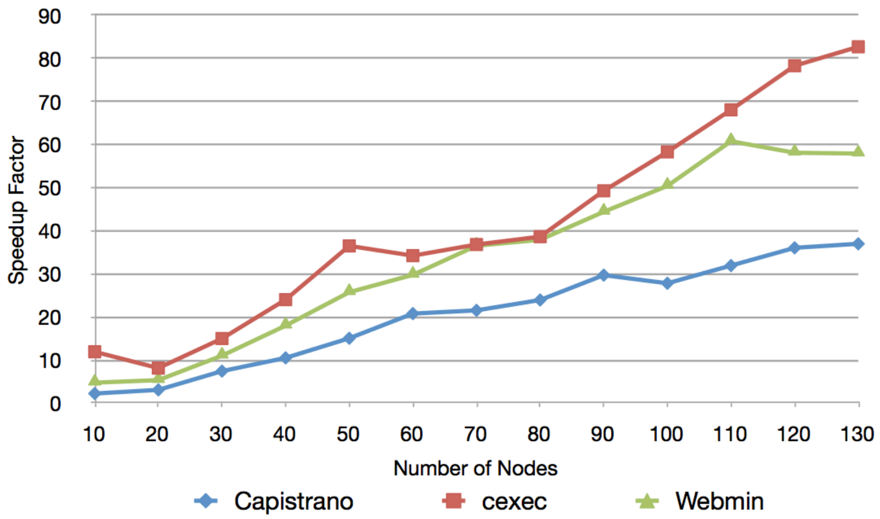


Figure 5.9: Speedup Comparison of Management Tools

Figure 5.9 displays the speedup achieved for each parallel execution tool while Figure 5.10 shows the execution times of each of these tools with 95% confidence intervals; due to the small variation between runs for the *cexec* and Webmin tools, these are difficult to view.

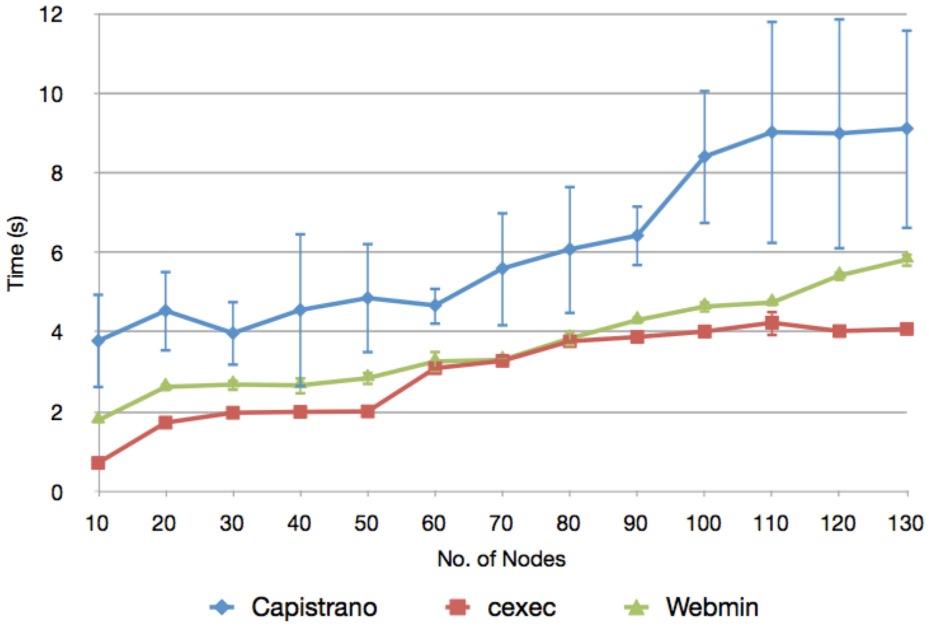


Figure 5.10: Parallel Execution Time Comparison of Management Tools

We see that the *cexec* tool offers the lowest execution time and greatest speedup when executing a command over up to 130 hosts. Webmin follows closely where execution times and speedup equal that of *cexec*'s at some stages. Capistrano being the slowest of the three still offers fast parallel command execution over 130 hosts taking only approximately 9 seconds but with greater variability.

Although the greatest speedup achieved is approximately 3.6 times below the ideal, the speedup of 82 at 130 hosts is a vast improvement on the serial version originally employed; a gap of 328 seconds exists when executing the same command over 130 hosts via the serial execution in comparison to Capistrano. By using *cexec* we achieve the greatest performance and least variability of execution times over a varying number of hosts; further experimentation would be required to determine the upper limits of *cexec* as well as the other control tools.

5.6 Summary

In this chapter, we have outlined the C2MS; a dynamic host monitoring and control tool designed specifically for those who need to effectively monitor a dynamic group of hosts which we call cloudlets. Cloudlet-based monitoring is especially useful in *ad hoc* clouds where administrators and *ad hoc* cloud scheduling mechanisms can determine the near-optimal *ad hoc* hosts that should execute a particular class of applications or assign additional *ad hoc* hosts to cloudlets that are more important than others.

Furthermore, organizations that employ server clustering will also find cloudlet-based monitoring useful to help maintain availability and scalability as well as to make infrastructure management and planning easier. This is especially useful when seeking to understand individual cloudlet demand and to judge whether to increase or decrease a cloudlet's capacity. However in order to provide this functionality, the limitations of monitoring tools must be overcome. Current monitoring tools tend to be static meaning that any change in an infrastructure's configuration, requires hosts to be reconfigured and monitoring processes restarted to allow the monitoring tool to successfully adopt the new setup; an unreasonable task to undertake on medium to large-scale infrastructures. We solve this challenge by developing the C2MS.

We have shown how the C2MS allows administrators to define cloudlets as well as easily add and remove hosts to and from these groups via our easy to use C2MS web interface. Administrators can then easily monitor individual cloudlets by viewing dynamically aggregated graphs for the many metrics that Ganglia offers as well as cloudlet power usage and CPU temperature monitoring data available by adding the appropriate module. All operations related to the C2MS are hidden from the administrator meaning regular Ganglia users will have no problems using the C2MS.

From the experiments we have performed, we have shown that the C2MS offers quick control of hosts as well as effective monitoring with little or no overhead compared with the tool it is built on. It is important to note that the C2MS can not only be used on clouds but any platform where Ganglia can be installed, for example, clusters, Grids, personal infrastructures, etc. Infrastructure administrators who wish to download the tool, can do so at [8].

Originally, the C2MS was created for use in a software production company but it has also successfully been used to monitor hosts within our *ad hoc* cloud computing environment during our evaluation of the platform. We detail the use of the C2MS in our evaluation of the *ad hoc* cloud in the following Chapter.

Chapter 6

Evaluating the *ad hoc* Cloud

6.1 Introduction

In this chapter, we evaluate our *ad hoc* cloud computing prototype to determine its feasibility, reliability and performance when run on a realistically simulated unreliable infrastructure. Based on our results, we argue that the *ad hoc* cloud is not only a feasible concept but also a viable computational alternative that offers high levels of reliability and can at least offer reasonable performance, which at times may exceed Amazon EC2. We measure the success of these criteria according to our evaluation model that specifies which aspects of any *ad hoc* cloud should be evaluated.

We first outline the computational platform used to evaluate our *ad hoc* cloud prototype in Section 6.3 and then conduct our evaluation in two parts. The first investigates the reliability of our *ad hoc* cloud when operating on a simulated unreliable infrastructure run on our chosen computational platform (Section 6.4). We explain how this is performed by obtaining host availability data from an operational infrastructure and replaying the events to simulate a set of sporadically available *ad hoc* hosts.

The second part of our evaluation measures the performance and overheads of our *ad hoc* cloud (Section 6.5). We execute a series of benchmarks representing a wide range of workloads to determine typical cloud job completion times. However, we show that a variety of overheads unique to the *ad hoc* cloud can increase the execution time of a cloud job. This includes pre- and post-execution overheads, periodic checkpointing overheads and virtual machine restoration overheads. We also investigate what affect our major contribution of P2P checkpoint distribution has on the network and whether a large number of checkpoints can be concurrently sent between *ad hoc* hosts.

In order to determine whether the measured *ad hoc* cloud performance including all associated overheads is acceptable, we execute the same set of benchmarks on Amazon EC2 and find that an *ad hoc* cloud can offer comparable performance with an instance that has comparable resources. We finally investigate whether the *ad hoc* server at the core of the *ad hoc* cloud can scale well or whether it is a performance bottleneck.

6.2 Evaluation Model

We now outline our evaluation model defining the criteria that should be measured when evaluating any implementation of any *ad hoc* cloud. These criteria are:

1. *Reliability:* determines whether the reliability of the *ad hoc* cloud is sufficient enough to complete cloud jobs. We define an *ad hoc* cloud as reliable when it successfully completes cloud jobs 95% of the time assuming the underlying infrastructure is unreliable and that the host(s) a cloud job runs upon, may fail a number of times throughout the job's lifetime. We also assume that the cloud job is not prevented from completing, for example, due to errors in development.

   The reliability of the *ad hoc* cloud can be measured by running a series of benchmark applications either over an actual unreliable or realistically simulated infrastructure expressing a range of reliability conditions and measuring the average application success rate.

2. *Platform performance:* determines whether the overall performance of the *ad hoc* cloud is at least reasonable for executing cloud jobs. We define an *ad hoc* cloud as offering reasonable performance when the difference of job execution times between the *ad hoc* cloud and a commercial cloud infrastructure are minimal, i.e. all cloud jobs take less than 25% longer to complete on average when no host failures occur and that each host failure must not increase all cloud job completion times by more than a further 10% on average. The *ad hoc* cloud job completion times take into account, the time for job submission, queuing, host scheduling, job execution and data transfer while the time to instantiate a virtual machine as well as data transfer is taken into account for commercial cloud infrastructures.

   The performance of the *ad hoc* cloud can be measured by benchmarking and comparing results from running the benchmarks on other computational infrastructures, for example, commercial cloud infrastructures.

3. *Component performance:* determines whether the performance of each individual component is sufficient to be included in an *ad hoc* cloud. Those that have the greatest affect on performance are:

    - *virtualization*: does the chosen virtual technology have low overheads?
    - *monitoring:* does monitoring all hosts in an *ad hoc* cloud have low overheads?
    - *scheduling:* are scheduling components efficient and accurate?
    - *host process interference:* are host processes greatly affected by *ad hoc* cloud operations and processes?

4. *Usability:* determines whether the *ad hoc* cloud software is easy to install and use for *ad hoc* cloud host users as well as the cloud's system administrators. This includes the time for each user to learn their respective interfaces, the actions they can perform and the ability to operate with little support. These can be measured by performing usability trials for all sets of *ad hoc* cloud users and determining whether the *ad hoc* cloud has similar interfaces and operations that current commercial cloud and volunteer users are familiar with.

In this chapter, we primarily evaluate the reliability and platform performance of our *ad hoc* cloud computing platform according to metrics outlined above. We define an *ad hoc* cloud as successful when all the above success criteria have been satisfied. Component performance and the likely usability of our *ad hoc* cloud computing platform have been evaluated in Chapters 3, 4 and 5.

6.3 Evaluation Platform

The majority of our experiments outlined later took place on the Edinburgh Data Intensive Machine 1 (EDIM1) [161, 14]. EDIM1 is a cluster primarily used by the Data Intensive Research (DIR) group [12] and associated partners to analyse large amounts of data; the cluster is hosted and managed by the EPCC [16]. The architecture of EDIM1 is shown in Figure 6.1. EDIM1 consists of 120 backend nodes distributed over three racks and has three frontend nodes *dir0*, *dir1* and *dir2*. Each backend node has:

- one 1.60 GHz Intel Atom 300 dual core processor with hyperthreading,
- 4 GB memory,

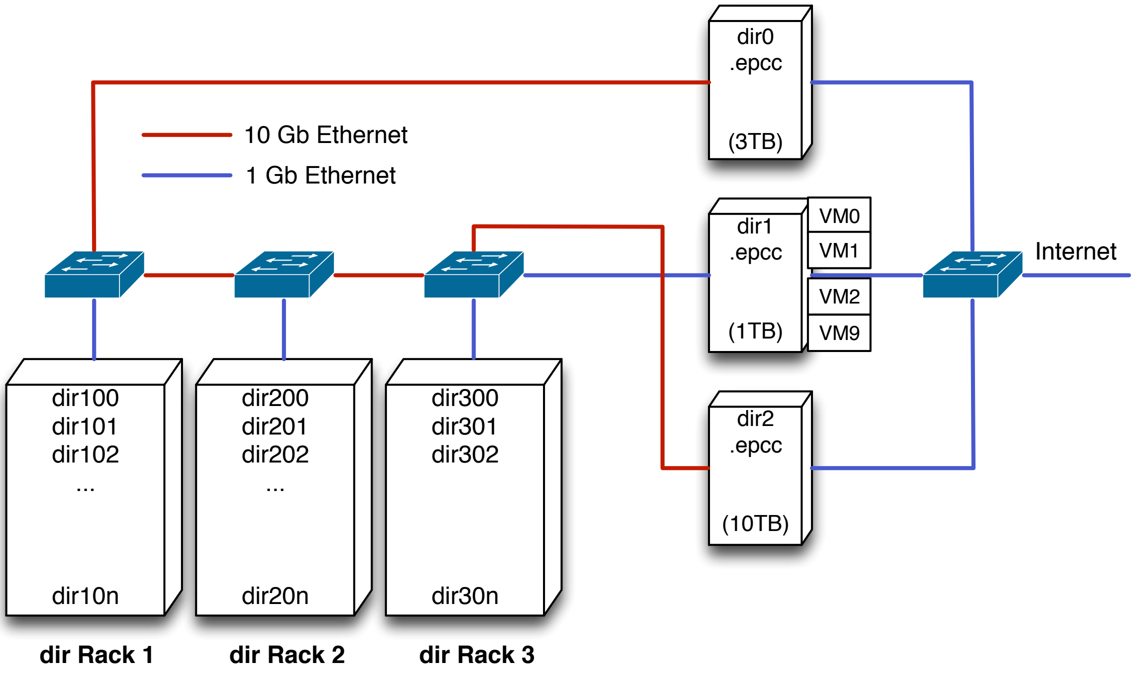


Figure 6.1: EDIM1 Architecture [15]

- 3 2TB SATA disks and 1 250GB SSD disk,
- 1Gb Ethernet interconnect.

As the primary purpose of EDIM1 is to analyse and store large amounts of data, storage capacity is favoured over compute resources. EDIM1 is securely operated from behind University firewalls therefore these resources can be accessed via a terminal session from the frontend login node *dir0*. However, if a publicly visible service is required, a virtual machine can be instantiated on *dir1* allowing the compute and storage resources to be exposed to the Internet. The frontend node *dir2* gives data a temporary storage location before the data is migrated to the cluster for analysis. Each of the frontend nodes are connected to the Internet by 1Gb Ethernet links while only *dir1* is connected to the backend nodes via a 1 Gb link; both *dir0* and *dir2* are connected to the backend nodes via a 10 Gb Ethernet link.

In order to test our prototype of an *ad hoc* cloud, we managed to obtain access to 30 backend nodes distributed over rack's one and two; a larger set of nodes was not available as the remaining nodes were being revised to have improved caching therefore leaving only 30 available non-virtualized nodes at that time. Each backend node was configured using Cobbler and Puppet and had the Scientific Linux 6.4 OS installed. The *ad hoc* client was installed on each of the nodes and henceforth they are referred to as the *ad hoc* hosts. The *ad hoc* server was installed on a publicly accessible virtual machine located on *dir1* which used 2 CPUs and 2 GB of memory from a machine that has a total of 32 CPU cores and 32 GB of memory available; the OS installed was Ubuntu 12.04.

The virtual machine also had 100 GB of attached block storage which the server was installed and run on. Note that we did not use *dir0* and *dir2* in the course of our experiments. We chose EDIM1 to evaluate our *ad hoc* cloud prototype as firstly, we required access to potentially a large number of available and non-virtualized nodes and secondly, in the event of any difficulties, support was available from the EPCC.

6.4 Reliability

In this section, we describe how we evaluate the reliability of our *ad hoc* cloud computing platform deployed on EDIM1. As we do not have access to an organizational infrastructure to test the reliability of our platform in a realistic setting, we describe how we accurately simulated such an environment by controlling the behaviour of 30 *ad hoc* hosts. We then detail our results showing that the *ad hoc* cloud is a reliable platform in the face of host failure or churn.

6.4.1 Simulating *Ad hoc* Host Behaviours

The primary aim of this experiment is to find the potential reliability an *ad hoc* cloud could offer in a realistic setting. Therefore we accurately simulated an unreliable infrastructure by obtaining Nagios monitoring data over a period of 36 months from 650 hosts in The School of Informatics at The University of Edinburgh. An example of the data we received is shown in Figure 6.2.

```
[1294472199] HOST ALERT: host256 UP SOFT 1 PING CRITICAL
[1294472210] HOST ALERT: host259 UP SOFT 1 PING CRITICAL
[1294472220] HOST ALERT: host174 DOWN SOFT 3 PING WARNING
[1294473745] HOST ALERT: host271 UP SOFT 1 PING CRITICAL
[1294473756] HOST ALERT: host259 DOWN HARD 1  PING CRITICAL
```

Figure 6.2: Nagios Example Output

The example output displays various forms of information. However the three most important entities are the timestamp when an event occurred, the hostname that the event relates to and the host state. For example, the first line of Figure 6.2 shows that *host256* became available on 8/1/2011 at 7:36:39 AM (i.e. the epoch time 1294472199) and this new state was determined using *ping*.

We parsed this large amount of monitoring data by creating a Nagios data tool that calculated the host activity for every hour, i.e the number of unique UP and DOWN state events for each host. Despite finding that hosts within Informatics are highly reliable and rarely fail or become unavailable, there were times when groups of hosts did become sporadically available over short periods of time. In one of the most active hours, at least 30 hosts acted in a sporadic manner. We therefore used this set of hosts to simulate the behaviour of an *ad hoc* cloud; this hour was between approximately 04:45 and 05:40 am on the 13th of September 2012. Therefore by using monitoring data from an actual infrastructure, we can determine the reliability of an *ad hoc* cloud as if it was operated over the selected set of hosts at the selected time.

Figure 6.3 shows the availability map depicting the group of selected host's behaviour during the selected hour. A host is initially assumed to be available until a red marker signifies the host has become unavailable or has failed. A green marker signifies that the host has now become available and the downtime can be calculated between the time of the two events, depicted by the dark grey area between the two. We include the time and date when each event occurred on the left hand side of the availability map with each blue area showing the number of events that occurred in each ten minute period.

Most importantly, we also show the reliability of each host. This was calculated as the ratio of the total number of downtime seconds over the total number of available seconds the host was available, from when monitoring records began until the beginning of the selected hour. We also show the last three digits of the IP address assigned to the *ad hoc* guest that runs cloud jobs; we define this as the *VM ID*. These virtual machines may run on any EDIM1 host depending on the order the installed *ad hoc* clients register with the *ad hoc* server.

In order to simulate the behaviour outlined in the availability map, we created and added a *simulator* daemon to the VM Service project. This daemon takes the Nagios monitoring data for the selected hour and replays the UP and DOWN state events for each of the *ad hoc* hosts on the EDIM1 infrastructure. Simulating an *ad hoc* host could be performed by instructing each EDIM1 node to shutdown and boot up when a DOWN and UP event occur respectively, however this solution is impractical and difficult to manage remotely.

Instead we informed the *ad hoc* server of the infrastructure state changes by modifying the VM Service project database. For example, when a DOWN event occurs, the respective *ad hoc* host and EDIM1 node is set to unavailable by setting the host's availability value to false. This then triggers the *ad hoc* server to initiate the virtual machine migration process when it detects the *ad hoc* host, that previously was executing a cloud job, is now unavailable. Similarly, when an UP state event occurs, the *ad hoc* host's availability value is set to true allowing the *ad hoc* host to receive cloud jobs or to restore virtual machine checkpoints. By replaying availability data from an infrastructure that an *ad hoc* cloud could have been deployed on, we can reasonably gauge the reliability of our prototype in similar realistic settings.

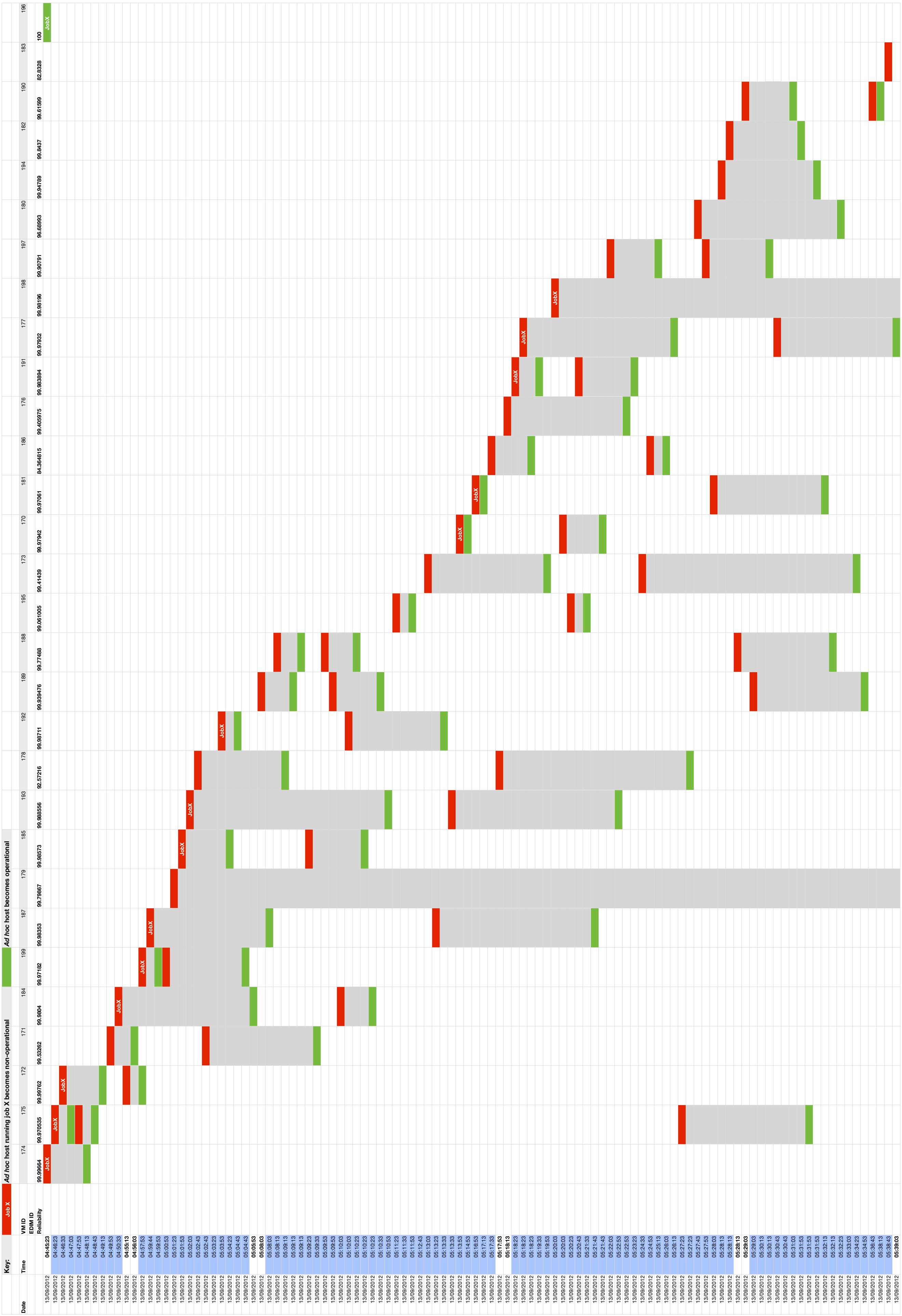


Figure 6.3: Informatics Host Activity Availability Map

6.4.2 Experiment Design

In order to test the reliability of our *ad hoc* cloud prototype, we submitted 15 CPU cloud jobs to the Job Service project via the mechanisms previously mentioned. Although this may not represent a large number of typical loads deployed on an *ad hoc* cloud, we decided not to utilize more than 50% of the available infrastructure so that in the event of any failures, a cloud job is not queued until another *ad hoc* host becomes available or the job does not run on an *ad hoc* host that currently executes another cloud job; both of which will increase the completion time of a cloud job. As we were only interested in finding the potential reliability an *ad hoc* cloud could offer at this point, we restricted the number of jobs submitted during our experiments to 15.

These jobs are then distributed by the *ad hoc* server to the most reliable *ad hoc* hosts as other scheduling criteria are inherently satisfied; the host is available, has appropriate hardware specifications and all are lightly loaded. Figure 6.3 also shows that the *ad hoc* server correctly distributes cloud jobs to the most reliable *ad hoc* hosts available; each *ad hoc* host assigned a job is labelled 'Job' in their first state event. Once each *ad hoc* host has been prepared for receiving a cloud job, the cloud jobs begin executing. Periodic checkpointing occurs once per minute and other *ad hoc* client operations that were described in Chapter 4, concurrently execute. The *simulator* daemon is then started to simulate the behaviour of 30 Informatics hosts on the 13th of September 2012 approximately between the times 04:45 and 05:40 am.

During the simulation, we recorded many aspects related to the reliability of the platform; our observed performance metrics are discussed later in the chapter. We record the source and target *ad hoc* hosts for each checkpoint sent in the platform, the source and target *ad hoc* hosts during an *ad hoc* guest migration and the number of jobs that were successfully completed when cloud job results were returned to the Job Service project. This experiment was performed three times to determine the overall reliability of running an *ad hoc* cloud on the Informatics infrastructure at the select time and date; we were restricted to running the experiment three times due to time constraints, however we are confident that upon further experimentation the results obtained would be similar, if not equal, to those outlined in the following section.

6.4.3 Results and Analysis

Our results from the three experimental runs show that under the environmental conditions simulated, the *ad hoc* cloud can offer a high level of reliability. Table 6.1 shows the summary of the cloud job success rates, the number of completed jobs and virtual machine migrations as well as and the type of failures experienced during each experimental run.

Experimental Run	**Success Rate**	**No. of Jobs Completed**	**No. of VM Migrations**	**Failure Types**
1	86.6	13	11	VM and BOINC
2	80	12	10	VM and BOINC
3	93.3	14	11	BOINC

Table 6.1: Reliability and Failure Statistics of the *ad hoc* Cloud

We see that during the first experimental run, 13 out of a possible 15 cloud jobs are successfully completed despite the unreliability of the simulated *ad hoc* cloud; this equates to 86.6% of the jobs being completed. The number of VM migrations that were triggered by the simulation was 11, therefore at least 4 cloud jobs did not experience any *ad hoc* host or guest failures. Similarly, experimental runs 2 and 3 showed that 80% and 93.3% of the respective submitted cloud jobs completed successfully.

Experiment two also showed that one less virtual machine migration occurred due to better selection of an *ad hoc* host to restore the checkpoint on. The failures that terminated cloud jobs were caused by either virtual machine or BOINC errors. Virtual machine errors were caused by the virtual machine becoming inaccessible to the *ad hoc* client when tested by the Accessible Detector component, or the virtual machine failed to restore properly. BOINC errors were caused by a failure to upload the cloud job's results despite the cloud job being completed.

We now show the virtual machine migration traces for each experimental run in Figures 6.4, 6.5 and 6.6 respectively. These traces firstly show the *ad hoc* hosts selected to initially execute each cloud job, labelled 'Job' on their first DOWN state event and the series of an *ad hoc* guest migrations from one *ad hoc* host to another. This is depicted by the coloured transition paths between hosts which also indicate the identifier of the job being migrated. For example, Figure 6.4 shows that *Job13*'s *ad hoc* guest with the VM ID 184, is migrated from EDIM1 host 159 to EDIM1 host 163 that has a reliability of 99.90791%; reliability values are adjusted after each failure, completed cloud job and state event.

We see that in Figure 6.4, the most virtual machine restoration activity takes place during the first 25 minutes and we also see in some cases, after an *ad hoc* guest has been migrated, it may have to migrate once again if the underlying *ad hoc* host becomes non-operational; a series of virtual machine migrations between *ad hoc* hosts are depicted by the same colour of transition paths. For example, *Job1* of Figure 6.4 is first migrated from EDIM1 host 155 to 167 and then onward to EDIM1 host 152 a short time later. The restoration fails on the next virtual machine migration to EDIM1 host 144 indicated by the error message state event.

A failure during restoration is caused by an unsuccessful restoration procedure by VirtualBox. The virtual machine either simply does not restore or the virtual machine is not accessible after the restore operation; features that we hope are fixed in future versions of VirtualBox. The second failure of the same experimental run is caused by BOINC not uploading the results of *Job8* even though the *ad hoc* server is operational and reachable. Figure 6.4 also shows that *ad hoc* guests are restored on the most reliable *ad hoc* hosts and that *ad hoc* guests can be restored on *ad hoc* hosts that have successfully completed their previously assigned job (e.g EDIM1 host 165); we assume all cloud jobs run to completion unless explicitly specified with the 'Complete' state event.

The virtual machine migration trace of Figure 6.5 is similar to the previous experimental run in terms of the *ad hoc* hosts chosen to restore an *ad hoc* guest, however the significant difference is that three cloud jobs do not successfully complete. The *ad hoc* guests executing the cloud jobs *Job3* and *Job12* failed to restore and the BOINC client executing *Job10* did not upload the cloud job's results; coincidently, *Job3* and *Job10* fail to successfully complete on the same *ad hoc* host. The single cloud job *Job5* did not return due to a result upload failure in the third experimental run shown in Figure 6.6 by the 'Upload Error' state event.

For all experimental runs, the average cloud job successful completion rate is 86.6% despite our aim to successfully complete 95% of all cloud jobs. Although in the success rate of 93.3% for experimental run three is close, we fail to meet this criterion in our experiments. However, it is important to note that the failures reducing the overall reliability of the *ad hoc* cloud were not caused by failures within our prototype. We hope that future releases of both VirtualBox and BOINC will provide solutions to their unrecoverable failures and increase the likelihood of a virtual machine restoring or a result being uploaded.

Therefore it is encouraging that the implementation of our prototype can indeed perform well and that by executing 15 cloud jobs over an unreliable simulated infrastructure, the *ad hoc* is still able to successfully complete cloud jobs more than 85% of the time; a figure that may increase with future improvements made to the *ad hoc* cloud development and the technologies it uses. Furthermore, we assume that the sporadic behaviour of the Informatics infrastructure at the aforementioned date and time, accurately simulates a typical infrastructure an *ad hoc* cloud will be deployed on, however there will be many cases when operational infrastructures are more unpredictable and unreliable. Only by deploying our *ad hoc* cloud computing prototype on a number of operational infrastructures with a wider range of workloads will we determine the true reliability of the *ad hoc* cloud.

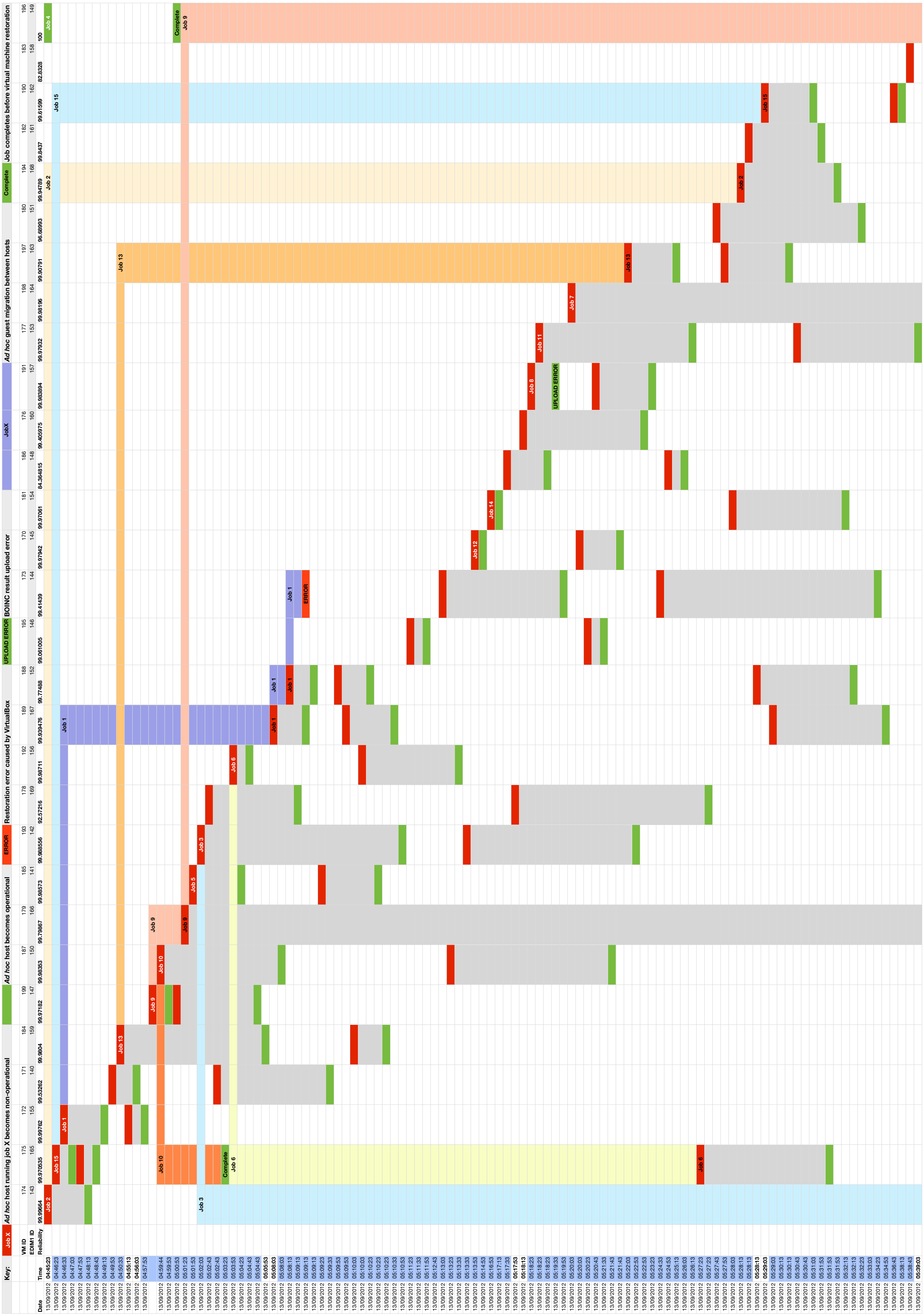


Figure 6.4: Simulated Host Failures and Job Relocations for Experimental Run 1

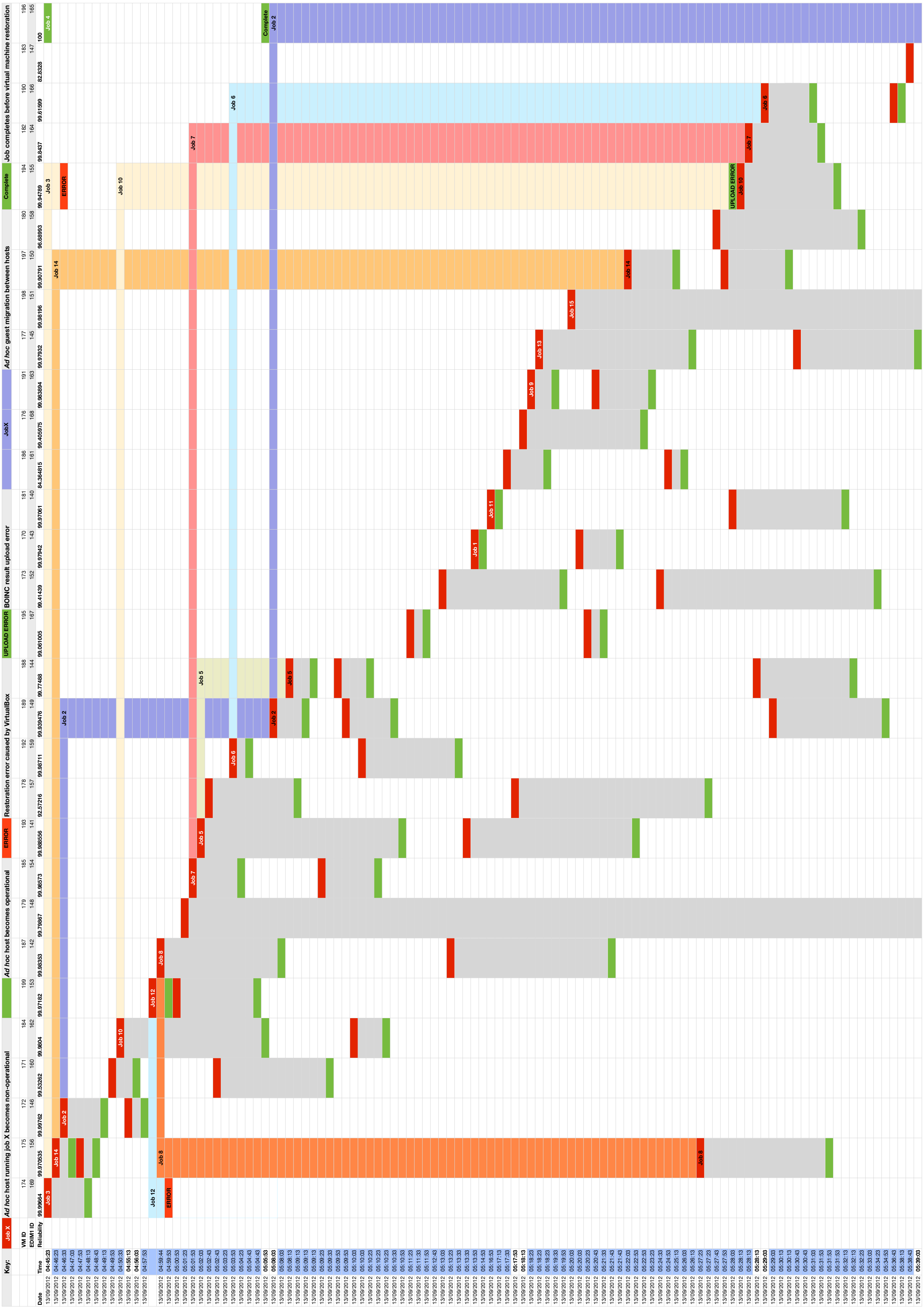


Figure 6.5: Simulated Host Failures and Job Relocations for Experimental Run 2

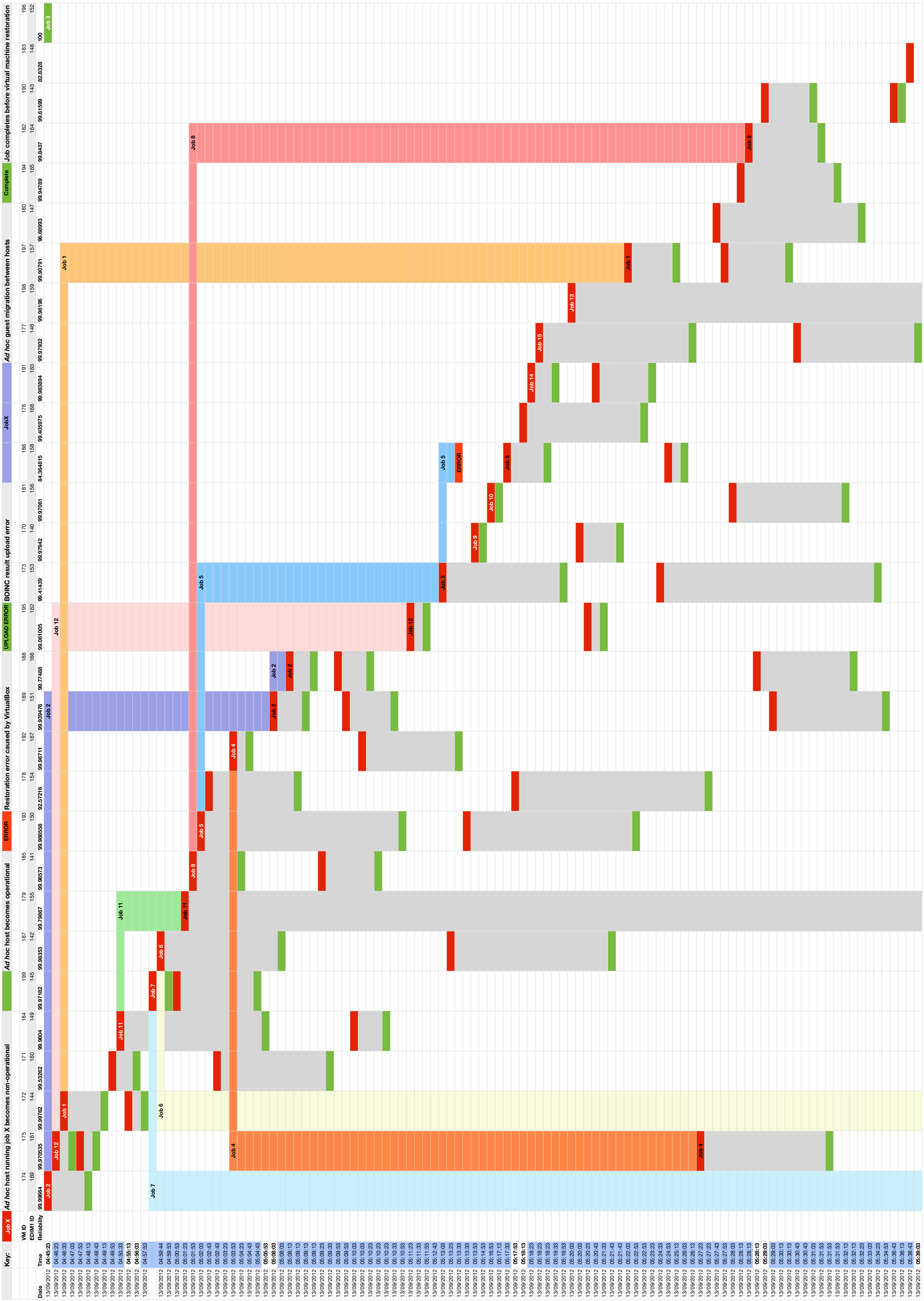


Figure 6.6: Simulated Host Failures and Job Relocations for Experimental Run 3

6.5 Platform Performance

We now investigate the platform performance of our *ad hoc* cloud prototype deployed on EDIM1. Firstly, we evaluate the performance of the *ad hoc* cloud by benchmarking our prototype deployed on EDIM1. Secondly we discuss the pre- and post-execution overheads unique to the *ad hoc* cloud that can ultimately increase the completion time of a cloud job. We then explore the affects of periodic checkpointing and checkpoint distribution have on both the cloud job as well as the sending and receiving *ad hoc* hosts and the network. This is followed by discussing virtual machine restoration overheads and we then give a comparison between the performance of the *ad hoc* cloud with Amazon EC2. Finally, we discuss the performance of our *ad hoc* server.

6.5.1 Benchmarking the *ad hoc* Cloud

We now investigate the performance of running cloud jobs in variety of different configurations on EDIM1 in order to determine the extent of overheads introduced by the *ad hoc* client. Similar to the V-BOINC experiments outlined in Section 3.4 of Chapter 3, we ran the *CPU*, *Memory*, *I/O* and *Disk* benchmarks in the following configurations:

- native execution on the EDIM1 node,
- execution on a virtual machine (VM) on an EDIM1 backend node. The virtual machine has 2 CPUs and 2 GB of memory,
- execution via V-BOINC and the V-BOINC client. The V-BOINC virtual machine also has 2 CPUs and 2 GB of memory,
- execution via the *ad hoc* cloud. A single *ad hoc* client's components and threads execute while periodic 1 minute checkpoints are distributed to three other *ad hoc* hosts,
- execution on Amazon EC2. A single m1.medium instance has 1 vCPU with 2 ECUs and 3.75 GB of memory.

Each benchmark was executed five times in each of the configurations above and on its completion, each benchmark would output its execution time, i.e the time it spent executing; we define this as the *cloud job execution time*. Note that each *stress* benchmark was executed as a single process and that the *Memory* benchmark allocated 2 GB of memory, e.g. *–vm-bytes 2048M*. In the case of running each benchmark via the *ad hoc* client, the cloud job execution time does not include other factors that may affect the total time a cloud job is present in the *ad hoc* cloud. For example, the time a job awaits to be scheduled to an *ad hoc* host, the time to configure and instantiate a virtual machine, the time taken by periodic checkpointing or the time caused by virtual machine migration; we discuss such overheads later in the chapter. The average cloud job execution times of each benchmark are shown in Figure 6.7. We display 95% confidence intervals to show that in most cases, the true mean will lie within the specified range.

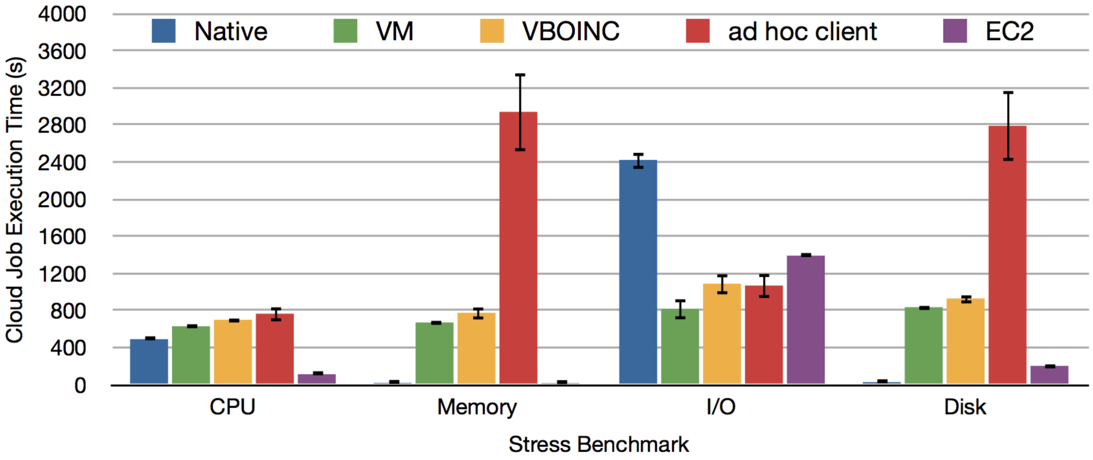


Figure 6.7: EDIM1 Benchmark Results and *ad hoc* Client Overhead

Figure 6.7 shows the cloud job execution times of the benchmarks running in their respective configurations. As expected, we see that running each benchmark natively typically offers the lowest execution time with the exception of the *I/O* benchmark. Similar to the results obtained by evaluating V-BOINC, this phenomenon may be caused by the virtualization technology having better caching mechanisms than the native host. We also see that running a virtual machine on EDIM1 introduces significant virtualization overheads. This is shown by the difference of execution times between the Native and VM configurations; the greatest virtualization overhead occurs when the *Memory* benchmark takes almost half an order of magnitude longer to complete when using virtualization. The virtualization overheads experienced on EDIM1 are significant and are due to the use of Intel Atom CPUs in the backend nodes where hardware virtualization is not supported.

As expected, we also see that the cloud job execution time is typically lower on Amazon EC2 than any configuration that uses virtualization on EDIM1, with the exception of the *I/O* benchmark. As outlined in Chapter 4, the overhead between the VM and V-BOINC configurations is minimal, but we see that executing the same benchmarks via the *ad hoc* client, the overhead is substantially greater for the *Memory* and *Disk* benchmarks.

At first glance, it would reasonable to assume the additional functionality of an *ad hoc* client in comparison to a V-BOINC client would explain the additional overheads. For example, periodic checkpointing, compressing and distribution may consume memory and disk space that otherwise would be used by the executing cloud job. Furthermore, the additional operations that occur in the background, such as the Resource Monitor, Running Detector, Accessible Detector and many of the Listener components waiting on a specific event to occur, may also consume resources required by the virtual machine and cloud job.

Although these additional functionalities do introduce overheads, they are negligible in comparison to the additional and significant overheads introduced by executing the *Memory* and *Disk* benchmarks via the *ad hoc* client on EDIM1. We hypothesise that during these benchmarks, the combination of a large number of instruction calls to the hypervisor (which must be trapped by the hypervisor, translated by the software, sent to the hardware and the request returned via the same route), as well as the execution of the virtual machine, the internal cloud job and the *ad hoc* clients are large enough to introduce the significant slowdown. However, in the event that a workload demands more resources from the *ad hoc* host, the overhead of the *ad hoc* client will not increase as the virtual machine will only be offered resource capacity that is not used by host processes or by BOINC and the *ad hoc* client. It is worth noting that if insufficient resources are available for the *ad hoc* client to consume, the execution times for workloads may increase, however by migrating the virtual machine to a more viable host, workload performance reductions can be minimized.

However, in order to prove the *ad hoc* cloud does not introduce significant overheads when executing both of these benchmarks, we executed the same benchmarks on four other hosts each with different hardware specifications. We define these hosts as *general purpose hosts* as they have hardware specifications comparable to standard models of laptops and Desktops; the class of hosts we believe host owners are likely to deploy an *ad hoc* cloud on. The specifications of these hosts are shown in Table 6.2.

Table 6.2 shows the variety of hardware specifications of our general purpose hosts, however most importantly, hardware-assisted virtualization is available on all of these hosts to help increase the performance of the hypervisor. In order to confirm that our *ad hoc* client does not introduce significant overheads, we executed the *stress* benchmarks on an *ad hoc* cloud deployed on our general purpose hosts and recorded the cloud job

Name	Processor Type	Memory	VT-x/AMD-V
MacBook Pro 2007	2.2 GHz Intel Core 2 Duo	2 GB	Yes
MacBook Pro 2010	2.4 GHz Intel Core 2 Duo	8 GB	Yes
Dell Optiplex 755	3 GHz Intel Core 2 Duo	4 GB	Yes
Dell Optiplex 790	3.1 GHz Intel Core i3-2100	4 GB	Yes

Table 6.2: General Purpose Host Benchmark Results

execution times. As the results obtained were similar for each host, we only outline the results obtained from an *ad hoc* cloud consisting of a server and client operating on the Dell Optiplex 790 and 755 hosts respectively. We do not display results for the *CPU* benchmark as it failed to complete on a virtual machine running on our general purpose hosts. Our results are shown in Figure 6.8.

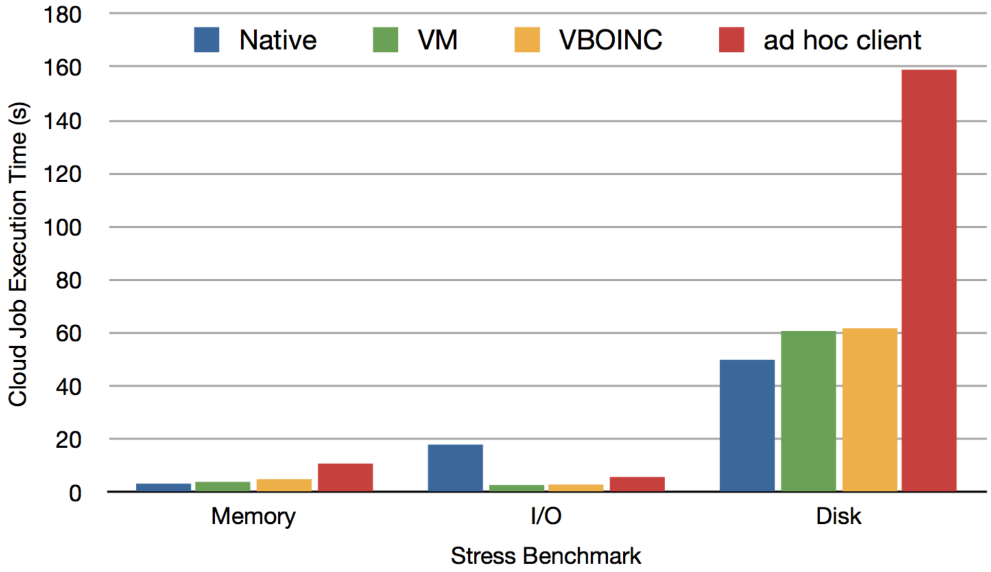


Figure 6.8: Dell Optiplex Benchmark Results and *Ad hoc* Client Overheads

Firstly we see that by executing these benchmarks on a general purpose host, the cloud job execution time is substantially reduced. For example, executing the *Memory*, *I/O* and *Disk* benchmarks via the VM configuration on the Dell Optiplex 755 takes approximately 11, 13.4 and 12.8 minutes less respectively.

Secondly, we see that in comparison to EDIM1, the performance overheads introduced when executing the *Memory* benchmark via the *ad hoc* client on a general purpose host is lower; this overhead is reduced from a cloud job taking 4.5 times longer to complete on EDIM1 to 2.5 on the general purpose host. Similarly, we also see that the performance overheads introduced when executing the *Disk* benchmark via the *ad hoc* client on a general purpose host is lower in comparison with running the same benchmarks on EDIM1; this overhead is reduced from a cloud job taking 3.3 times longer to complete on EDIM1 to 2.5 on the general purpose host.

Conversely, the overhead increases when executing the *I/O* benchmark via the *ad hoc* client on the general purpose host; this overhead previously took 1.3 times longer to complete on EDIM1 but now takes 4.2 times longer. Note that the execution time of this benchmark on an *ad hoc* client is still lower than the native execution on the general purpose host.

Despite the more realistic overheads observed from a general purpose host, the cloud job execution times for each benchmark are lower when executing on a general purpose host in comparison to executing the same benchmarks on an Amazon EC2 m1.medium instance that has similar resources to the Dell Optiplex 755. The *ad hoc* client running on this host executes the *Memory*, *I/O* and *Disk* benchmarks, approximately 68%, 98% and 21% faster than on the m1.medium instance.

By executing the series of *stress* benchmarks on EDIM1 and a number of hosts that are likely to be used by a large majority who employ *ad hoc* cloud computing, we have shown that the overheads introduced by the *ad hoc* client are minimal for a cloud platform of this nature. Furthermore, we have shown that an *ad hoc* client running on a general purpose host is able to perform better than executing on an Amazon EC2 instance with comparable resources.

We investigate whether the *ad hoc* cloud is still able to outperform Amazon EC2 later in the chapter, after we outline the affects of the many unique overheads associated with the *ad hoc* cloud have on a cloud job's execution time.

6.5.2 Pre- and Post-Execution Overheads

We now know the true execution overheads introduced by virtualization and the *ad hoc* client, as well as the likely cloud job execution times when these jobs are run on EDIM1 or hosts with similar hardware specifications to the general purpose hosts aforementioned. However, the time between an *ad hoc* cloud user submitting their job and the user receiving their results will be greater than the execution time of the cloud job running on the *ad hoc* cloud; we define this as the *total completion time*. Other than the execution time of a cloud job, the total completion time of a job includes the times associated with performing the following tasks:

- the *ad hoc* server to detect a submitted cloud job as well as create and queue the workunit,

- the *ad hoc* server to select a near-optimal *ad hoc* host to assign the cloud job to. A job may have to wait until an *ad hoc* host becomes available,

- the *ad hoc* client to prepare the *ad hoc* host for receiving the cloud job. This includes the time for downloading or creating a DepDisk, disk attachment, virtual machine boot up and configuration.

- the *ad hoc* client to request and download the cloud job as well as upload any results after completion.

We investigate what affects each of these pre- and post-execution tasks have on the total completion time of a cloud job; we discuss other overheads that occur during the execution of a cloud job in the next section. Our experimental setup was as follows: we submitted a Primes job to the *ad hoc* cloud and measured the wallclock execution time for each of the above tasks. To measure these times, we modified our *ad hoc* client by adding a simple counter that recorded the start and end times when the task started and ended respectively. The *ad hoc* client then calculated the difference and output the task completion times to file which were then analysed.

The experiment was performed on both EDIM1 and the general purpose *ad hoc* clouds hosts five times and the results were averaged. In the latter case, as the results obtained were similar for each host, we only outline the results obtained from an *ad hoc* cloud consisting of a server and client operating on the Dell Optiplex 790 and 755 hosts respectively.

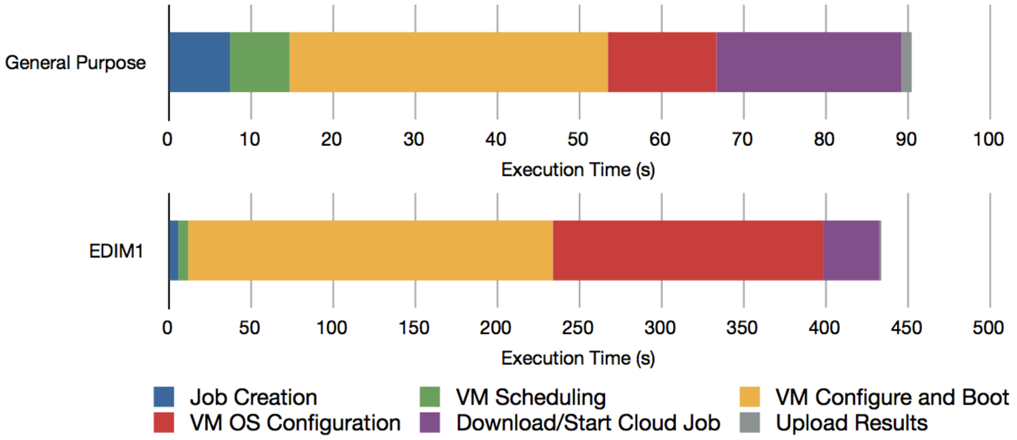


Figure 6.9: Pre- and Post-Execution Overheads

We see from Figure 6.9, that the pre- and post-execution overheads by running Primes on both the general purpose host and EDIM1 differ significantly. In the former case, the contribution to the total completion time is minimal; approximately 90 seconds. This overhead is comparable to the acquisition of a single virtual machine on Amazon EC2 [175, 126].

On the other hand, the pre- and post-execution overheads observed when Primes is executed on EDIM1, are in some cases substantial; note the difference of y-axis scales. The time to configure and boot the virtual machine are approximately 5.8 and 12.6 times greater than performing the same tasks on a general purpose host. These large overheads can be attributed to the lack of support for hardware-assisted virtualization on EDIM1s Intel Atom processors as well as their relatively low processing capacity.

The overheads of performing the remaining tasks on both a general purpose host and EDIM1 are small. Both *ad hoc* servers are able to detect a cloud job, create a workunit and schedule a workunit to an *ad hoc* host in approximately 13 seconds; this assumes that an *ad hoc* host is available, otherwise a workunit may have to wait to be scheduled. Furthermore, both *ad hoc* clients are able to download and prepare the Primes application for execution in approximately 22 and 34 seconds on a general purpose host and EDIM1 respectively.

However note that the download time of a job is directly related to its size and the available bandwidth between the *ad hoc* server and client. The time to upload the job in both cases takes approximately 1 second after which the *ad hoc* cloud user is able to view and download their results. Note that there are no other overheads related to downloading a new virtual machine image for every cloud job as the virtual machine can be reset and re-used after the cloud job completes or the virtual machine fails or terminates abruptly. By considering the results from the types of general purpose hosts that are likely to be used by a large majority who employ *ad hoc* cloud computing, we see that the pre- and post-execution overheads introduced by the *ad hoc* server and client minimally contribute to the total completion time of a cloud job.

6.5.3 Checkpointing Overheads

Pre- and post-execution overheads may minimal, however there are other overheads that may increase the total completion time of a cloud job during its execution on the *ad hoc* guest. We perceived that such overheads existed by comparing the job execution time (see Figure 6.7) and the completion time recorded by BOINC system as part of the Job Service project; we define the latter as the *BOINC execution time*. Note that the former, measures only the time the job spent executing and not the wallclock time between the start and end time of the cloud job.

By default, BOINC records the time when it sends a job to an *ad hoc* guest and the time when it receives the job's the results. The difference between the start and end times is therefore the BOINC execution time and this can obtained either by querying the Job Service database or through the project's administration web interface.

In order to determine the performance overheads of our *ad hoc* client while an *ad hoc* guest executes a cloud job, we submitted each benchmark to our EDIM1 *ad hoc* cloud five times and collected their BOINC execution times once completed. During the execution of the benchmark, the *ad hoc* client was set to periodically checkpoint once per minute and distribute the compressed checkpoint to three other *ad hoc* hosts. The results of all experiments were averaged and these are shown in Table 6.3.

Benchmark	**BOINC Execution Time (s)**	**Job Execution Time (s)**	**Execution Overheads (s)**	**Unaccounted Overheads (s)**
CPU	829	763.92	65.08	30.08
Memory	3341	2938.96	402.04	367.04
I/O	1156.60	1067.49	89.11	54.11
Disk	2918.60	2788.34	130.26	95.26

Table 6.3: Overheads Unaccounted for when Executing Cloud Jobs

We see that by subtracting the cloud job execution time from the BOINC execution time, we obtain the time introduced by the overheads associated with the *ad hoc* client. The BOINC execution time does however include the time for the *ad hoc* server to send the cloud job to the *ad hoc* guest and for the guest's BOINC client to return the results. We know from Figure 6.9 that the time to perform these operations on EDIM1 is 34 seconds and 1 second respectively. Therefore we can update the additional time introduced by the *ad hoc* client to exclude these overheads. This leaves an unaccounted period of time introduced by other *ad hoc* client overheads.

Table 6.3 also shows that these overheads differ dependent on the type of cloud job running. Although an *ad hoc* client is composed of many threads concurrently executing various tasks, we can rule out this as a primary factor of variable overheads between each benchmark. The only feature of an *ad hoc* client that could introduce such overheads is periodic checkpointing. We investigate what affect periodic checkpointing has on the total completion time of a cloud job.

6.5.3.1 Checkpoint Downtime

Periodic checkpointing is one of the many important processes used improve the reliability of the inherently unreliable *ad hoc* cloud model. While we show later in the chapter that this is an effective method of providing reliability, periodic checkpointing does however have one potentially significant limitation; during each checkpointing operation, the virtual machine must be paused therefore suspending the executing cloud job; we define this as the *checkpoint penalty*.

Previously in Section 3.4.2 of Chapter 3, we investigated what affects of taking checkpoints each minute had on the storage space of a volunteer host that uses V-BOINC. We found that for *CPU*, *Memory* and *I/O stress* benchmarks, the average time to take a checkpoint was approximately 1.5 seconds while the same operation took the *Disk* benchmark on average 25 seconds to complete; the exact figures can be found in Table 3.2.

For the respective benchmarks, the downtime incurred while taking checkpoints is low, however taking per minute checkpoints may not be optimal for both the cloud job or the *ad hoc* host. For example, as a cloud job consumes a larger portion of storage space or memory, or the checkpointing frequency decreases, the size of the checkpoint may increase, in turn increasing the time to take the checkpoint.

In order to determine the potential checkpoint times for a range of checkpointing frequencies, we varied the frequency over the period of one hour for each benchmark. This was performed by instantiating our V-BOINC virtual machine with 1 GB of memory on our MacBook Pro 2007 general purpose host and running each of our *stress* benchmarks individually while taking either 1, 2, 3, 4, 6, 12, 30 or 60 equally spaced checkpoints per hour.

For example, we started by taking 60 checkpoints per hour while the *CPU* benchmark was executing. In the second hour, the same benchmark was executed while 30 checkpoints were taken, and so on; this process was repeated for each benchmark. After each hour, the virtual machine was terminated and a new one was instantiated for the following hour to ensure each experiment was independent and that no virtual machine or host caching affected the results of the subsequent experiment.

Checkpoint times were obtained by the *ad hoc* client via the UNIX-based *time* command when executing the VirtualBox *snapshot* function. Note that as this experiment was performed on a different host which had a different virtual machine configuration from the V-BOINC checkpoint experiment (see Section 3.4.2) , the per minute checkpoint times for each benchmark are slightly different to those outlined in Chapter 3. Table 6.4 shows the minimum and maximum checkpoint times experienced when executing the benchmarks while a varying number of checkpoints were taken each hour.

Benchmark	**Min. Checkpoint Time (s)**	**Max. Checkpoint Time (s)**
CPU	2.3 (CF 60)	13.21 (CF 1)
Memory	5.83 (CF 4)	16.46 (CF 2)
I/O	2.37 (CF 60)	13.05 (CF 4)
Disk	43.4 (CF 30)	54.71 (CF 4)

Table 6.4: Checkpoint Frequency and Time Relationship

Table 6.4 only displays the minimum and maximum checkpoint times as there is no strict correlation between the checkpointing frequency and checkpoint time; we indicate which checkpoint frequency (CF) produces the minimum and maximum checkpoint times in Table 6.4. We sometimes see that a pattern can emerge where lower checkpoint frequencies produce higher checkpoint times and vice versa. For example, we see that the maximum checkpoint times are produced when the checkpoint frequency is 4 or lower, however the checkpoint frequencies that produce the minimal checkpoint times are not consistent for each benchmark.

We see that a checkpoint frequency of 60 produces the minimal checkpoint times for the *CPU* and *I/O* benchmarks while a checkpoint frequency of 4 produces the minimal checkpoint time for the *Memory* benchmark. Therefore the differences between the minimum and maximum checkpoint times are partially dependent on the checkpoint frequency, but not for all classes of application.

Despite the checkpointing frequency employed, we can be reassured that the checkpoint downtime penalties of periodic checkpointing are minimal for the *CPU*, *Memory* and *I/O* benchmarks. However, we see that the *Disk* benchmark has a high checkpoint downtime penalty of at most 54.71 seconds per checkpoint. If per minute checkpoints were taken while the *ad hoc* guest executes a disk-intensive benchmark, we would see that for every minute, the cloud job could potentially be suspended for the following 54.71 seconds.

6.5.3.2 Checkpoint Size

During the same experiment, we also recorded the checkpoint sizes dependent on the checkpoint frequency used. The results are shown in Table 6.5. Note that as this experiment was performed on a different host which had a different virtual machine configuration from the V-BOINC checkpoint experiment (see Section 3.4.2), the per minute checkpoint sizes for each benchmark are slightly different to those outlined in Chapter 3.

	Benchmark Checkpoint Size (MB)			
Snapshots per Hour	**CPU**	**Memory**	**I/O**	**Disk**
60	37.2	56.6	36.9	89.5
30	37.3	43.5	37.7	89.7
12	37.8	44.0	39.0	90.1
6	38.2	45.0	38.5	92.2
4	38.6	43.3	41.2	91.0
3	38.2	43.3	37.0	2370.9
2	36.5	57.8	38.4	1574.8
1	37.0	44.3	36.9	1393.3

Table 6.5: Checkpoint Interval and Size Relationship

We see that in most cases, a decrease of the checkpoint frequency does not necessarily increase the size of the checkpoint; this is particularly true for benchmarks that do not write to disk or consume large amounts of memory. Surprisingly, the *Memory* benchmark also follows this pattern and produces checkpoint sizes of approximately 44 MB despite utilizing over 90% of the virtual machine's memory. Unsurprisingly, as the *Disk* benchmark continues to write data to the virtual disk, the checkpoint size increases. A dramatic increase is observed when the checkpoint frequency is three per hour or fewer; we attribute this to the behaviour of the benchmark. We can therefore be reassured that if a checkpoint frequency of 4 per hour or higher is used by each *ad hoc* client, the checkpoint sizes and downtime penalties will be relatively small.

6.5.3.3 The Near-Optimal Checkpoint Frequency

This therefore introduces the problem of how to decide which checkpoint frequency is best with the aim of reducing both the checkpoint time and size; we define this as the *near-optimal checkpoint frequency*. However, the near-optimal checkpointing frequency is not only a factor of checkpoint sizes and capture times, but also the desired reliability of the *ad hoc* cloud and the load placed on the *ad hoc* host's CPU and network by checkpoint compression and distribution respectively.

A high checkpoint frequency will produce a greater number of checkpoints that must be distributed to other *ad hoc* hosts. This in turn will increase the likelihood of a cloud job completing if the job is interrupted by *ad hoc* host or guest terminations or failures. This will however introduce greater CPU and network loads. A low checkpoint frequency will produce fewer checkpoints, which in some cases may be larger in size, but the reliability of the *ad hoc* platform would suffer. Calculating the near-optimal checkpoint frequency for each application submitted to the *ad hoc* cloud would be extremely difficult, therefore we only outline a possible solution to provide a rough estimate of the near-optimal frequency for each class of *stress* benchmark. We base our assumptions on the data presented in Tables 6.4 and 6.5.

Our analysis first begins by discarding checkpoint frequencies that are 3 or less for the *Disk* benchmark due to sudden increase of checkpoint size; it would be unwise to distribute this amount of data over the network periodically. Furthermore, we also discount checkpoint frequencies 4 or less for all benchmarks as it is unreasonable to assume that if the cloud job fails and its *ad hoc* guest is restored on another *ad hoc* host, the *ad hoc* cloud user would have to wait an additional 15 minutes while the cloud job returns to the previous state before the *ad hoc* guest failed; we define this as the *re-computation overhead*.

As checkpoint frequencies greater than 4 per hour result in similar checkpoint sizes and capture times, reducing network load and achieving a desired reliability level are the only entities that must now be factored into determining the near-optimal checkpoint frequency; we omit CPU load in this decision making process as we assume that compressing checkpoints no greater than 100 MB consumes little CPU resources. We also assume that by compressing checkpoints as a *.tar.gz* file with a low compression ratio of 5:4, produces an 80 MB compressed file, we can estimate the amount of likely checkpointing traffic sent over the network to other *ad hoc* hosts per hour. Figure 6.10 shows our estimations.

We see that if 60 checkpoints are taken per hour and each is sent to one other *ad hoc* host, 4.6 GB will be distributed every hour until the benchmark completes. Similarly, in the case when 30, 12 and 6 checkpoints are taken every hour, the total amount of data distributed to a single *ad hoc* host would be 2.3 GB, 0.9 GB and 0.4 GB respectively.

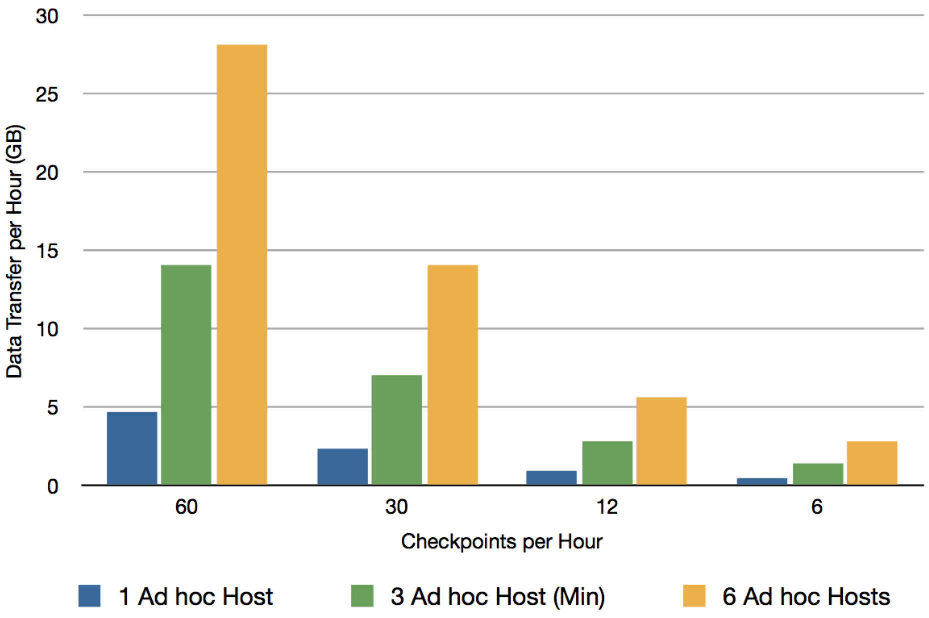


Figure 6.10: Estimated Checkpoint Data Transfer Per Hour

However, our implementation specifies that the P2P Scheduler selects at least three other *ad hoc* hosts to receive a single checkpoint to provide a reliable environment for cloud job execution. Therefore an *ad hoc* host must transfer a total of 14.06 GB, 7.03 GB, 2.81 GB or 1.4 GB if 60, 30, 12 or 6 checkpoints are taken every hour, respectively.

We believe that in most cases, an *ad hoc* host will not have to send checkpoints to more than six other *ad hoc* hosts at a time due our *ad hoc* scheduler's scheduling rule that the most reliable hosts are selected first to receive checkpoints. For example, in the event that the six most reliable *ad hoc* hosts each have a probability of 60% of failing or terminating at any time, each *ad hoc* host in the *ad hoc* cloud must send every checkpoint to these six *ad hoc* hosts in order to satisfy the 95% successful completion rate for cloud jobs.

In this possible worst case scenario, we see that if the checkpoint frequency is 60 or 30 per hour, an *ad hoc* host must transfer a total of 28.12 GB and 14.06 GB respectively per hour to other *ad hoc* hosts. This is in contrast to a total 2.81 GB of data transferred per hour when the checkpoint frequency is 6 per hour. With the aim of striking a balance between desired reliability and reducing network load, we believe that taking and distributing 60 or 30 checkpoints per hour results in a significant amount of data being transferred over the network. Furthermore, we assume that a checkpointing frequency of 6 per hour (i.e. 1 checkpoint every 10 minutes) does not provide enough reliability and that the re-computation overhead is still quite high.

Therefore we believe that a reasonable estimation of the near-optimal checkpointing frequency based on checkpoint size, capture time, desired reliability for a job and the reduction of load for CPU and network resources is 12 checkpoints per hour for our particular benchmarks. It is important to note however, that this estimation is based on the results from our benchmarks, which may be atypical in terms of checkpoints sizes compared with normal workloads and have significantly different resource demands or usage patterns. Furthermore, we realize that our chosen static near-optimal checkpoint frequency may not be optimal for small cloud jobs that complete in under five minutes, for example. We also realize that the network of EDIM1 may be significantly different from those of commodity networks, however as mentioned in Section 1.5 of Chapter 1, we do assume that in most cases the *ad hoc* cloud will be deployed on a LAN that offers reasonable performance, e.g organization and research institution networks.

Currently, the chosen near-optimal checkpoint frequency is a static value set within our *ad hoc* cloud implementation however we aim to dynamically adjust this frequency based on the reliability of the *ad hoc* host executing the cloud job. For example, as the reliability of the *ad hoc* host increases, fewer checkpoints can be taken and subsequently distributed, and vice versa; we leave the addition of this functionality as future work.

6.5.4 Network Performance

Our chosen near-optimal checkpoint frequency of 12 per hour results in at least 2.81 GB of data being transferred per hour from each *ad hoc* host if the policy that checkpoints must be distributed to a minimum of three *ad hoc* host receivers is enforced. In our possible worst case scenario where six *ad hoc* hosts receive a checkpoint from a checkpoint sender, the sender must transfer 5.6 GB of data per hour. A commodity network could easily handle one *ad hoc* host distributing this amount of data per hour, however we expect an *ad hoc* cloud to have many available *ad hoc* hosts each executing an *ad hoc* guest. Therefore, as the number of executing *ad hoc* guests and cloud jobs increase, the network may become a bottleneck if many concurrently distribute checkpoints.

We therefore investigate whether the network of EDIM1, which offers similar bandwidth rates to commodity cloud networks when data is exchanged between racks, is able to cope with the concurrent distribution of large number of checkpoints between *ad hoc* hosts. We assume that the size of a checkpoint when compressed is 80 MB and that each of the 30 EDIM1 *ad hoc* hosts concurrently transfer a checkpoint to the other 29 *ad hoc* hosts. We determine whether the EDIM1 network is capable by recording the total completion time for all *ad hoc* hosts to successfully distribute their checkpoint to the other 29 hosts.

This experiment was performed by a bash script that concurrently logged in to each *ad hoc* host and executed the *pscp* command (see Section 4.6.4) that triggered the checkpoint distribution to begin. On each of the *ad hoc* hosts, the compressed checkpoint is placed in a directory which is referenced by the *pscp* command as well as a *hosts.txt* file specifying the *ad hoc* hosts the checkpoint should be sent to. The completion time of each *pscp* command is recorded by the UNIX-based *time* command and sent back to a central location for analysis. This experiment was performed five times and the completion times for all *ad hoc* hosts to successfully distribute their checkpoint are shown in Figure 6.11.

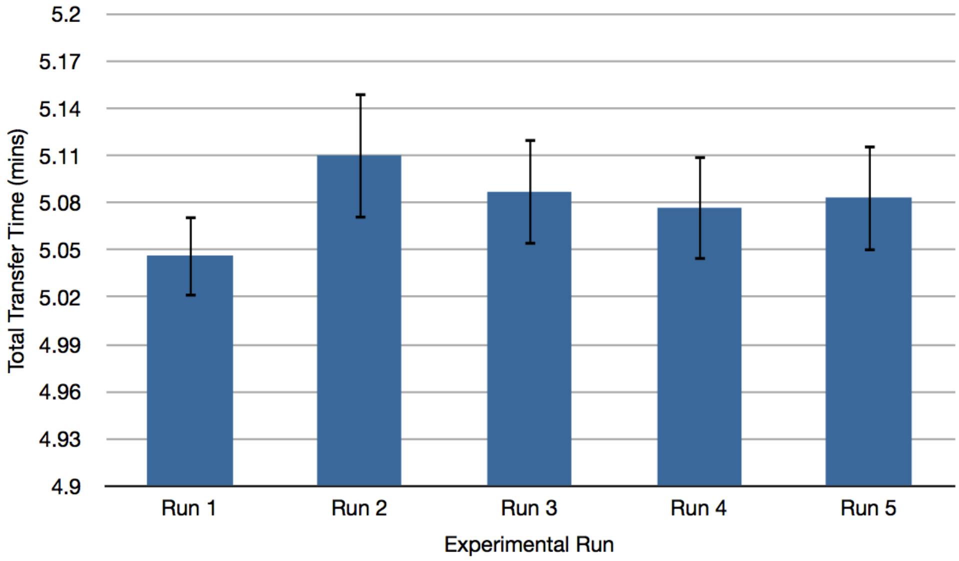


Figure 6.11: Concurrent Checkpoint Distribution

We see that each experimental run on average completes in 5.7 minutes where the differences between runs is minimal; 95% confidence intervals show that in most cases, the true mean will lie within the specified range. By concurrently sending an 80 MB checkpoint from each of the 30 EDIM1 *ad hoc* hosts to the remaining 29 *ad hoc* hosts, 870 checkpoints totalling to 67.9 GB of data are being concurrently sent at a single moment. This also shows that *ad hoc* hosts have the ability to concurrently send and receive a large number of checkpoints. By sending 870 checkpoints per hour, this equates to 290 *ad hoc* hosts each concurrently sending a checkpoint to 3 other *ad hoc* hosts or 145 *ad hoc* hosts each concurrently sending checkpoints to 6 other *ad hoc* hosts.

Based on these results, we can be assured that if an *ad hoc* cloud has a large number of *ad hoc* guests executing cloud jobs, the *ad hoc* cloud should be able to quickly distribute checkpoints and scale well assuming the network is not heavily utilized. Furthermore, it is also reasonable to assume that a commodity network should be able handle a reasonably large number of concurrently distributed checkpoints; this of course is dependent on the network and the amount of traffic however. In order to reduce the load imposed on the network, we can take advantage of recent virtual machine migration developments. For example, the size of checkpoints can be reduced by techniques such as data deduplication between the source and destination *ad hoc* hosts [89], delta compression ensuring only dirty memory pages are transferred to the destination host [203] or pre-copying disk access requests to the destination host [69], for example; we leave the incorporation of these techniques into the *ad hoc* cloud as future work.

It is important to note that in our reliability analysis experiment described in Section 6.4 and the results from our performance experiments shown in Figures 6.7 and 6.8 as well as Table 6.3, we employed per minute checkpoint frequencies and not the near-optimal checkpoint frequency outlined above during this network performance analysis. A per-minute checkpoint frequency was selected to firstly determine potential upper limits of reliability of our EDIM1 *ad hoc* cloud and secondly, remain consistent with previous V-BOINC experiments. This also allowed us to determine that the EDIM1 network can handle much higher checkpointing traffic.

6.5.5 Virtual Machine Restoration

Checkpoints that are successfully distributed between *ad hoc* hosts must be instantiated at any moment after an instruction from the *ad hoc* server. We have previously shown that the performance overheads associated with periodic checkpointing and that a commodity network could potentially handle the traffic generated by our P2P Reliability Algorithm. We now outline the performance overheads associated with virtual machine restoration which also affect the total completion time of a cloud job.

We consider the situation where an *ad hoc* client detects that an *ad hoc* guest is non-operational and informs the *ad hoc* server that the cloud job is no longer running. As previously mentioned, the *ad hoc* scheduler then selects a near-optimal *ad hoc* host to restore the *ad hoc* guest on. After receiving the instruction to restore the appropriate checkpoint, the *ad hoc* client then decompresses the checkpoint, re-registers the virtual machine with VirtualBox and performs the restoration. In order to determine the affects that these operations have on the total completion time of a cloud job, the maximum execution times of performing these tasks are measured and are shown in Figure 6.12.

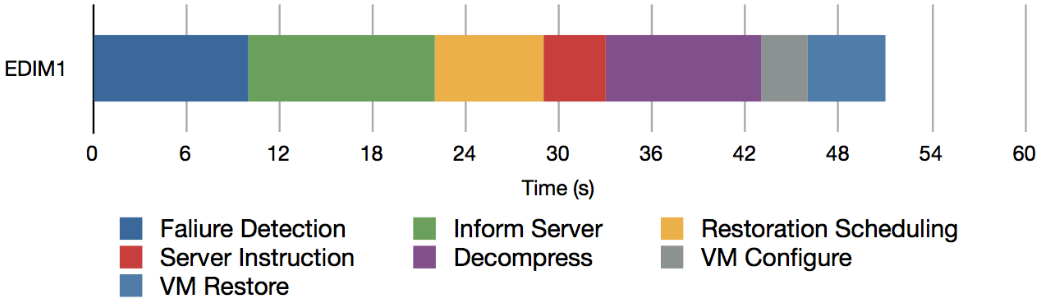


Figure 6.12: *Ad hoc* Guest Restoration Overheads

We see that from the time an *ad hoc* client detects an *ad hoc* guest is non-operational, to the time the *ad hoc* guest is restored on another *ad hoc* host, the entire process is likely to take under one minute. We measured the total time for this process during our reliability analysis experiment and found that on average, the restoration process took 24.4 seconds. In a scenario where the *ad hoc* host or client fails, meaning the *ad hoc* server cannot be informed of any failures, the time for the server to detect the failure increases the time of the restoration process by approximately two minutes; the *ad hoc* server classifies that an *ad hoc* host has failed when it has not polled the *ad hoc* server during the last two minutes.

Furthermore, we also assume that at least one *ad hoc* host is available for restoration scheduling, otherwise the time until an *ad hoc* host becomes available lengthens the total process of checkpoint restoration. As mentioned in Section 4.6.5 of Chapter 4, the *ad hoc* client of the selected *ad hoc* host to restore the checkpoint, must inform the *ad hoc* server that the restoration process has completed successfully, allowing the server to instruct other *ad hoc* hosts who received the same checkpoint to delete it.

We do not display the timings for these tasks as they have no effect of the total completion time of a cloud job, however we estimate these tasks to take at most 40 seconds based on the shared tasks above. For example, the *ad hoc* client informs the *ad hoc* server that the checkpoint has been restored, the server then performs *ad hoc* host selection and issues an instruction; this only leaves the time to delete the checkpoint to be estimated, which we believe takes a short period of time. Note that as these results were obtained from running our *ad hoc* cloud platform on EDIM1, it is likely that some of the task execution times shown in Figure 6.12 would be slightly lower. In either case, it is encouraging that if a cloud job is interrupted in any way, the job can potentially begin executing again within one minute.

6.5.6 The *Ad hoc* Cloud vs Amazon EC2

In this section, we give a comparison between the performance achieved by executing cloud jobs on our *ad hoc* cloud computing prototype, deployed both on EDIM1 and our general purpose hosts, to executing the same cloud jobs on Amazon EC2. In particular, this evaluation incorporates the additional times introduced by the overheads of the *ad hoc* cloud that were omitted in earlier experiments. This includes pre- and post-execution, checkpointing and virtual machine restoration overheads.

We calculate the total completion time of a cloud job running on an *ad hoc* cloud by adding the times associated with each overhead giving the wallclock time from job submission to results retrieval. The checkpoint frequency for this experiment was set to 60 per hour and in order to incorporate virtual machine restoration overheads, we assume that an *ad hoc* guest has been migrated to one other *ad hoc* host during its execution; an operation that took 24.3 seconds to complete. Executing a cloud job on EC2 does not involve many of the overheads associated with the *ad hoc* cloud. However a cloud user must wait for an EC2 instance to boot and then transfer their job to the instance before it can be executed, much in the same way an *ad hoc* cloud job waits on an *ad hoc* guest booting which is then transferred to the guest for execution; we incorporate these pre-execution overheads in our use of the EC2 instance. We assume that an instance takes approximately 60 seconds to boot and that uploading the same job used in previous experiments (see Figure 6.9) takes 22.5 seconds, as the network performance of both EDIM1 and Amazon EC2 are similar. Amazon EC2's total cloud job completion times are calculated by adding the times introduced by these overheads, to the cloud job execution times displayed in Figure 6.7.

We executed our *Memory*, *I/O* and *Disk* benchmarks on the EDIM1 *ad hoc* cloud as well as the *ad hoc* cloud run over our general purpose hosts; in the latter case, the *ad hoc* server and client ran on the Dell Optiplex 790 and 755 hosts respectively. The same benchmarks were executed on a m1.medium EC2 instance that has similar resources to the Dell Optiplex 755 host. Our results do not show the execution of the *CPU* benchmark due to reasons outlined previously. Figure 6.13 shows the total cloud job execution time differences between the *ad hoc* cloud and Amazon EC2.

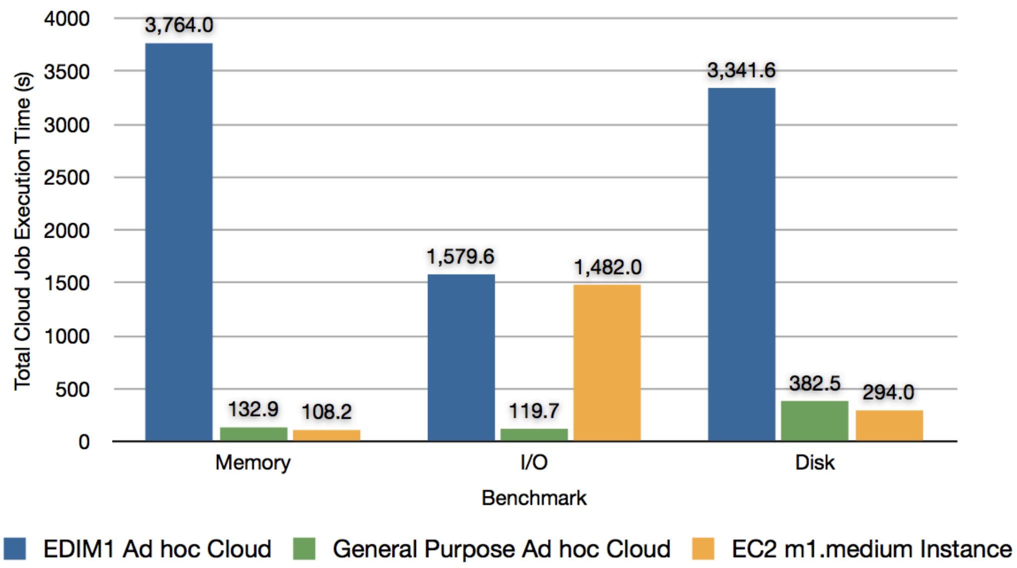


Figure 6.13: *The Ad hoc* Cloud Performance vs Amazon EC2

As predicted, we see that there is a significant difference between the total cloud job completion times when executing the cloud job on the EDIM1 *ad hoc* cloud to the same cloud job executing on an m1.medium EC2 instance. Again we attribute such differences to the lack of hardware-assisted virtualization of EDIM1 processors and their relatively low processing capacity in comparison with the EC2 instance. Despite this, only an 6.5% increase of the completion time is observed when executing the *I/O* benchmark on EDIM1.

Most importantly, the differences between the total cloud job completion times when executing a cloud job on an *ad hoc* cloud deployed on general purpose hosts and Amazon EC2, is minimal; these differences would decrease by 24.3 seconds if one virtual machine migration was not assumed to occur. Conversely, a larger number of migrations will increase the total completion time accordingly. We see that if one migration is performed on the *ad hoc* cloud, the *Memory* and *Disk* benchmarks approximately take 25% and 30% longer to execute on our general purpose host *ad hoc* cloud respectively, while the *I/O* benchmark outperforms EC2 by over an order of magnitude. However, if no migrations occur, we see that the *Memory* benchmark produces a similar execution time to Amazon EC2.

Despite the slight increase of a cloud job's total completion time in some cases when executing on the general purpose *ad hoc* cloud, users of the *ad hoc* cloud can be reassured that their job will continue to execute (in most cases) in the face of host failure or churn. If an instance fails on Amazon EC2, the current state of the instance is lost unless strict state-saving measures are employed. Therefore, whether zero or many virtual machine migrations occur during the operation of an *ad hoc* cloud, we can be confident that the platform will offer reasonable and comparable performance to an EC2 instance with equivalent resources. This is encouraging especially as the *ad hoc* cloud has to operate on an unreliable infrastructure in contrast to offering the cloud service from a dedicated infrastructure.

6.5.7 *Ad hoc* Server Performance

So far in this chapter, we have primarily evaluated our *ad hoc* client and the overheads of the operations it performs. We now focus on the performance of our *ad hoc* server and more specifically, how well the server operates in our realistically simulated *ad hoc* cloud environment. Throughout our simulated *ad hoc* experiments, the C2MS monitored the *ad hoc* server's CPU, memory, storage and network load.

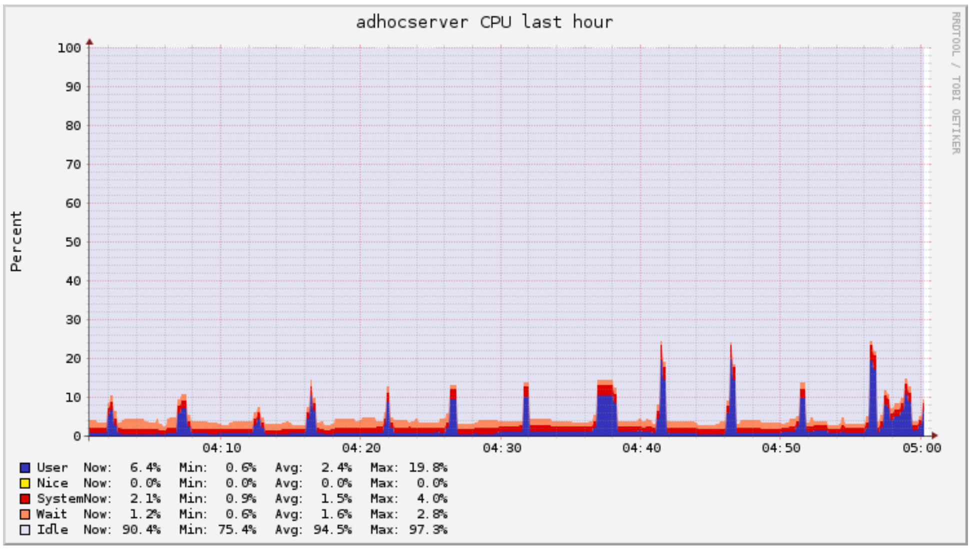


Figure 6.14: *Ad hoc* Server CPU Utilization

Figure 6.14 shows that during the one hour of our reliability analysis experiment, the 2 cores available to the *ad hoc* server are at most 25% utilized. The periodic spikes of CPU usage correlate to the periodic nature of BOINC daemons from both the VM Service and Job Service projects as well as the periodic polls from *ad hoc* clients.

Figure 6.15 shows that during the same hour, 1.95 GB of memory is consistently utilized from the *ad hoc* server's available 2 GB of memory. This is due to the storage of outbound BOINC workunits from the VM Service project; we specify that the number of workunits containing the virtual machines to be distributed to *ad hoc* hosts is

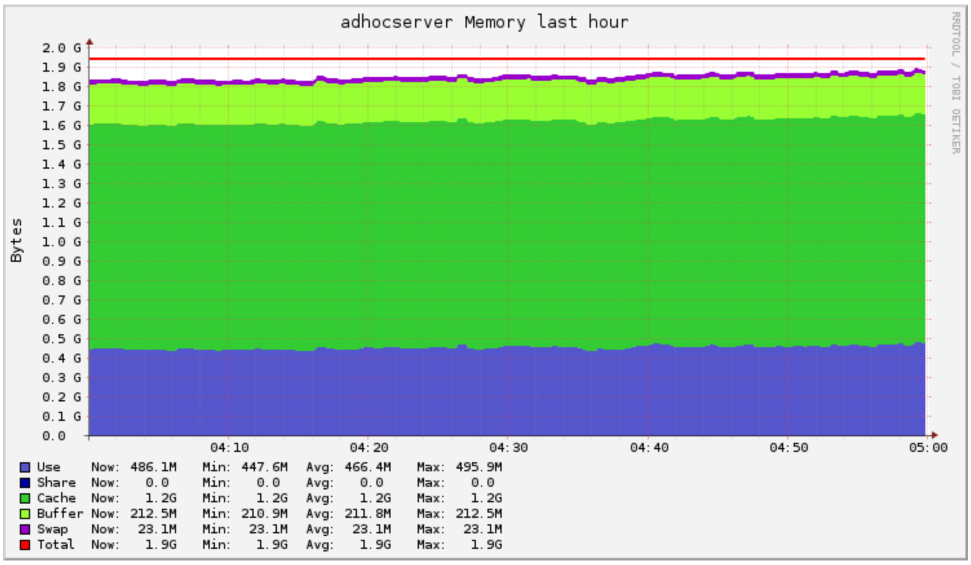


Figure 6.15: *Ad hoc* Server Memory Utilization

limited to 30. The BOINC daemons of both the VM Service and Job Service projects and the subsequent and frequent database accesses, will also consume a small portion of the *ad hoc* server's memory.

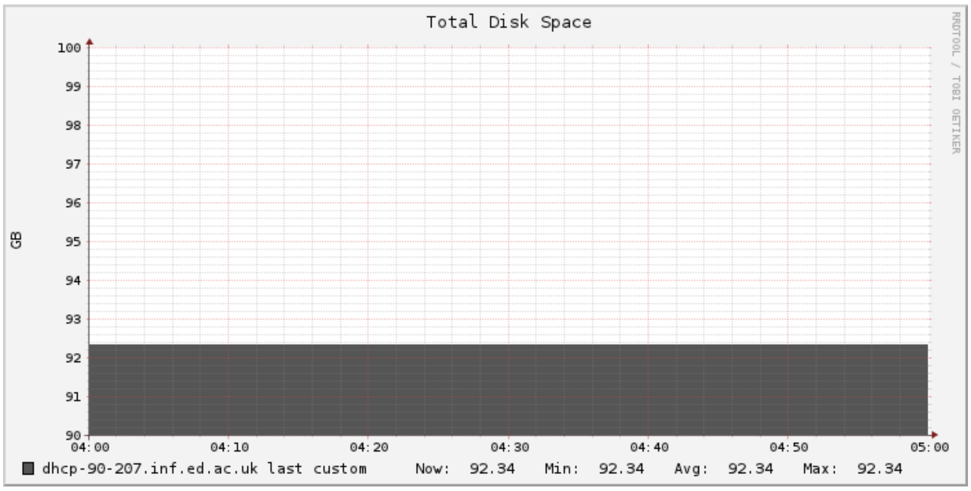


Figure 6.16: *Ad hoc* Server Disk Utilization

Figure 6.16 shows that the disk space consumed by the *ad hoc* cloud implementation is consistent and also relates to the physical disk space consumed by outbound BOINC workunits, but a large amount of storage space is also consumed by the VM Service and Job Service projects as well as the BOINC server installation.

Finally, Figure 6.17 shows the network traffic to and from *ad hoc* clients. A large spike of outward traffic is initially shown when the *ad hoc* server instructs 15 *ad hoc*

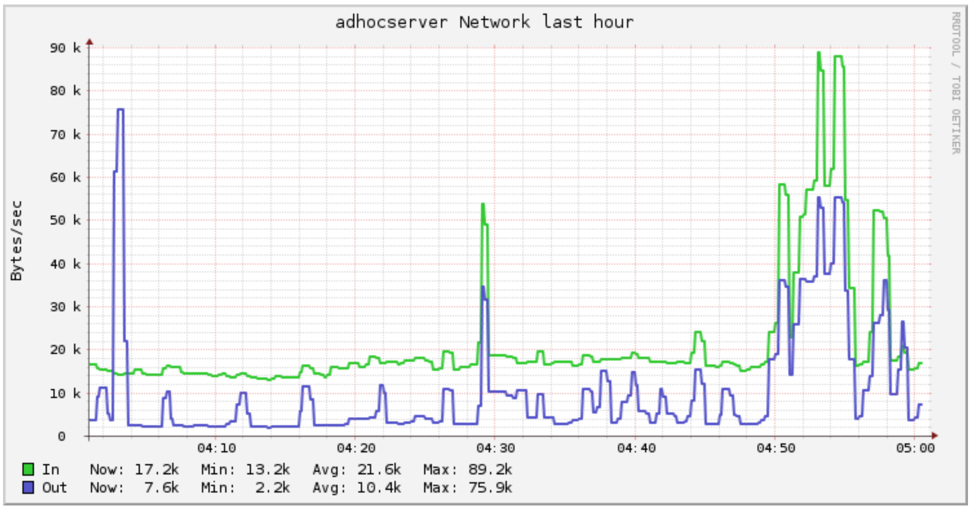


Figure 6.17: *Ad hoc* Server Network Utilization

clients to configure their *ad hoc* host and download the job. The latter network usage spikes are contributed by the web interface and command line analysis of results.

Figure 6.17 does not however show the network bandwidth consumed by the 15 *ad hoc* clients each downloading an initial virtual machine before the experiment began. While the *ad hoc* server is capable of concurrently serving 15 virtual machine requests at any given moment, if the number of requests increase substantially, the bandwidth available to the *ad hoc* server will eventually become a bottleneck.

A number of solutions exist to overcome this problem. Firstly, in the event the server is overloaded with requests, the exponential backoff algorithm BOINC employs can be used to instruct *ad hoc* clients to perform the virtual machine request at a random time later in the near future. In most cases, this will be acceptable as downloading a virtual machine is not a time critical operation unless the *ad hoc* cloud is heavily loaded and requires more resources.

Secondly, the size of the virtual machine image can be reduced to accommodate a larger number of requests to be satisfied. As previously mentioned, our V-BOINC virtual machine has been stripped of all unnecessary components and therefore it is small as it could possibly be. However, different virtual images can be used which will result in different virtual machine image sizes. For example, the mini desktop Linux OS Damn Small Linux is approximately 50MB and would allow a substantially larger amount of virtual machine images to be served concurrently.

Thirdly, an image's size can be reduced further by employing high-ratio compression algorithms, for example, 7-zip [1]. Penultimately, due to the architecture of the BOINC server, the *ad hoc* server can be replicated multiple times and placed on different networks to load balance the distribution of virtual machine image requests. Finally, it may be more efficient to distribute virtual machine images from one *ad hoc* host to another in a P2P fashion. This could be achieved by determining the physical proximity and network bandwidth available between each possible sender and the receiving *ad hoc* host and then selecting a sender based on the near-optimal connection; we also aim to investigate the effectiveness of this method in the near future.

In our experiments, we have shown that during the operation of an *ad hoc* cloud with 30 *ad hoc* hosts, half of which execute cloud jobs, the consumption of the *ad hoc* server's resources is minimal at any given time. Based on these results, we are confident that the *ad hoc* server is scalable and could easily handle larger infrastructures as regular BOINC is designed to do; the limits of the *ad hoc* server's capabilities in the current configuration is left as future work.

6.6 Summary

We have now evaluated the reliability and performance of our *ad hoc* cloud deployed on EDIM1. Firstly, we showed that by deploying our *ad hoc* cloud prototype on EDIM1 and accurately simulating a currently operational infrastructure, the reliability of our *ad hoc* cloud was found to be high and could successfully complete up to 93.3% of cloud jobs.

We are encouraged by the fact that the remaining cloud jobs did not complete due to errors in the underlying technologies of BOINC and VirtualBox and not our implementation of our prototype; though many things could be improved to increase the reliability further. Therefore, we are confident that the reliability of our *ad hoc* cloud development can increase in a range of other scenarios when solutions to the aforementioned errors are implemented.

Secondly, we outlined the performance of running our set of *stress* benchmarks on EDIM1 in a variety of configurations to determine the cloud job execution times and overheads unique to EDIM1. Due to the low performance of the platform's CPUs and lack of hardware-assisted virtualization, the cloud job execution times and overheads were substantial in comparison to a number of general purpose hosts each with hardware-assisted virtualization and standard hardware specifications. We then showed that by executing cloud jobs on an *ad hoc* cloud deployed on general purpose hosts, cloud job execution times in many cases were lower than executing the same jobs on Amazon EC2.

Thirdly, we discussed the pre- and post-execution performance overheads of the *ad hoc* cloud such as cloud job registration, *ad hoc* host job scheduling and preparation as well as result uploading to the BOINC project. We showed that these overheads are significant when an *ad hoc* cloud is deployed on EDIM1, again due to the low performance of the platform's CPUs and lack of hardware-assisted virtualization. In contrast, pre- and post-execution overheads were low when deploying an *ad hoc* cloud on general purpose hosts.

Fourthly, we outlined the performance overheads introduced when a cloud job executes on an *ad hoc* guest. We showed that such overheads are primarily caused by virtual machine suspension due to periodic checkpointing. This was followed by a discussion of calculating the near-optimal checkpoint frequency to minimize checkpointing overheads but also to minimize network bandwidth usage and increase reliability. We found that for our particular classes of applications, the near-optimal checkpoint frequency was 12 checkpoints per hour. This value may however not be applicable to the large number of different applications submitted to the *ad hoc* cloud.

We then investigated what affect periodically distributing checkpoints between *ad hoc* hosts in a P2P fashion has on network performance. We showed that if 870 checkpoints were distributed at any given moment, the network of EDIM1 would be able to easily handle the generated traffic. Furthermore, this substantiates our claim that the *ad hoc* cloud could operate over a large number of *ad hoc* hosts and is able to scale. This was followed by outlining the overheads associated with virtual machine restoration and we found that when an *ad hoc* guest or host fails that executes a cloud job, the *ad hoc* guest can be restored elsewhere in under one minute.

Penultimately, we compared the performance of executing a cloud job on an *ad hoc* cloud, deployed both on EDIM1 and a set of general purpose hosts, to executing the same cloud job on an Amazon EC2 instance with equivalent resources. We showed that while an EDIM1 *ad hoc* cloud typically does not match the performance of EC2, an *ad hoc* cloud deployed on our general purpose hosts can offer similar performance even in the event of one or multiple *ad hoc* guest failures. Furthermore, cloud jobs operating on an *ad hoc* cloud are by default protected from any *ad hoc* host or guest failures or terminations, however this is not the case when utilizing the EC2 infrastructure.

Finally we showed the affects our experimental setup has on the *ad hoc* server. Due to the efficiency of BOINC, the underlying V-BOINC platform and the additional server components, a host with standard hardware specifications can successfully act as an *ad hoc* server without degrading the performance of the entire *ad hoc* cloud;

we believe that deploying the *ad hoc* server on a host with state of the art processors and large amounts of memory and disk space would have little affect on the overall performance of the *ad hoc* cloud due to the low resource utilization levels seen during our experiments.

Overall, our evaluation has shown that our *ad hoc* cloud prototype is a feasible and reliable platform that can offer at least reasonable performance to cloud jobs, in some cases which may be better than running the same job on Amazon EC2. We believe that the simulated behaviour of an unreliable infrastructure will represent typical infrastructures an *ad hoc* cloud runs on, however there will be many other cases when the underlying infrastructure is more unpredictable and unreliable. We believe the *ad hoc* cloud will have the ability to overcomes these untested challenges.

We also believe that by testing a range of workloads, we show that CPU, memory and I/O-intensive applications are well suited to the *ad hoc* cloud, particularly when the cloud operates over a set of general purpose hosts that have hardware support for virtualization. For these workloads, the performance overheads introduced by the *ad hoc* client are minimal and the checkpoint sizes are typically small, resulting in less bandwidth being consumed by our P2P checkpointing approach.

Disk-intensive workloads are less suited to the *ad hoc* cloud as checkpoint sizes and capture times are large, resulting in greater checkpoint penalties and a greater amount of data being distributed over the network. In the former case, a greater checkpoint penalty results in less checkpoints being taken overall, ultimately affecting the reliability provided to the job via the P2P checkpointing approach. In the latter case, commodity networks may not be able handle such large traffic volumes therefore affecting the performance and reliability of other cloud jobs.

For similar reasons, data-intensive workloads are currently limited by the local network accessible to the *ad hoc* cloud. While we do not explicitly investigate such workloads, Kijsipongse *et al.* do however show that performing data-intensive analyses by using a modified version of V-BOINC that supports Hadoop, data-intensive workloads can effectively run on volunteer infrastructures [137], and therefore an *ad hoc* cloud. The benchmarks we have used to test the performance and overheads of our *ad hoc* cloud will represent a large number of cloud jobs submitted to the cloud, however the investigation of data-intensive workloads and many other applications that have different characteristics, for example CPU-memory-intensive applications, are left for future investigation.

While we believe that our *ad hoc* cloud can operate under a variety of unreliable conditions, as well as effectively execute a vast range of different workloads, these extrapolations are based on assumptions that need to be tested by deploying the *ad hoc* cloud on a live operational infrastructure with real workloads in the near future.

Chapter 7

Conclusions

In this chapter, we conclude this thesis by giving a summary of the work presented. This is followed by outlining the possible future research to be undertaken to improve the *ad hoc* cloud computing paradigm. Finally, we end this thesis by offering our concluding remarks in relation to our research hypothesis.

7.1 Summary

This thesis has outlined our proposal for an *ad hoc* cloud computing platform to allow end-users who are unable to outsource computation or deploy a dedicated computational platform locally, to take advantage of a new cloud computing paradigm that uses the spare capacity from their non-exclusive and unreliable infrastructure.

This use of an *ad hoc* cloud may be due to their inability to migrate their applications and data to a remote infrastructure, the unsuitability of that computational platform to the application's or end-user's requirements. Or it may be economically impractical for them to procure and operate an internal dedicated computational platform. Infrastructure owners who simply wish to improve the utilization and return on investment of their current infrastructure are also able to take advantage of *ad hoc* cloud computing.

We hypothesised in Chapter 1 that the concept of an *ad hoc* cloud is a feasible and reliable and platform that is able to at least offer reasonable levels of performance. We argue that this hypothesis is correct based on the evaluation of our *ad hoc* cloud prototype. Our prototype is the based on the combination of the six founding principles of *ad hoc* cloud computing: virtualization, cloud computing, volunteer computing, monitoring, management and testing. We discussed each principle in Chapter 2 and the subsequent chapters detailed how each principle was sequentially integrated with the others to create an *ad hoc* cloud computing platform; this allowed the case to be made and development of an effective *ad hoc* cloud computing platform to be prototyped. This prototype is built on an extended version of our virtualized volunteer infrastructure V-BOINC, therefore transforming it into an *ad hoc* cloud computing platform.

7.1.1 V-BOINC

V-BOINC, or the virtualized version of BOINC, is perhaps the important enabler of the *ad hoc* cloud; this is our first listed contribution of Section 1.6 in Chapter 1. Described in Chapter 3, V-BOINC takes advantage of both the features of BOINC, such as the ability to run tasks on volunteer heterogeneous hosts, and also virtualization where security issues between host and guest are inherently addressed; the latter also provides easier management of the infrastructure.

V-BOINC builds on BOINC's strengths and overcomes many of the disadvantages of regular BOINC. For example, BOINC project developers must port their application to each target host architecture, implement application-level checkpointing and are limited to deploying applications that have no external dependencies. Furthermore, BOINC users must trust the BOINC project they attach to.

V-BOINC overcomes these challenges by allowing applications to be ported to a single host and a wider range of applications to be executed; unlike typical CPU-intensive applications BOINC executes (Section 1.6, contribution 2). Furthermore system-level checkpointing is available and security issues are inherently addressed by the sandbox environment. The solutions to these disadvantages, combined with the ability to transfer and interact with virtual machines, makes V-BOINC a fundamental component of an *ad hoc* cloud.

Our evaluation shows that the implementation of V-BOINC introduces negligible overheads above those introduced by virtualization alone when executing a wide range of applications. This also is the case for applications that require external dependencies such as MPI or R, for example. With the exception of disk-intensive applications, the checkpointing functionality of V-BOINC also consumes little storage space and the checkpoint penalty is low. Although the development of V-BOINC is necessary for creating a successful *ad hoc* cloud, as a standalone service, V-BOINC is popular in the volunteer community where approximately 200 users have downloaded the package as well as made use of the on-line V-BOINC service.

7.1.2 *Ad hoc* Cloud Prototype

In order to transform V-BOINC into an *ad hoc* cloud computing platform, we first identified many of the research challenges involved of creating such an infrastructure in Chapter 4. A summary of challenges that are addressed by our *ad hoc* cloud prototype are how to:

- operate a cloud over a set of non-exclusive, untrustworthy and sporadically available hosts that are unpredictable in nature and where the total computational and storage potential of the cloud changes frequently,
- maintain service availability in the presence of host or guest churn or failure to ensure the job continuity,
- minimize the affect cloud processes have on host processes,
- schedule cloud jobs to near-optimal hosts and guests,
- monitor and control dynamic groups of hosts,
- develop a platform that is simple to download, deploy and use.

We then explored the current state of research into *ad hoc* cloud computing or similar computational infrastructures and found two important studies by Kirby *et al.* and Chandra *et al.* that provided the basis for all subsequent *ad hoc* cloud research to build on. Many other studies investigated the concept of merging volunteer and cloud computing to create an *ad hoc* cloud, however most fail to identify the important research challenges to be solved and little technical realization was reported. We attribute this to the difficulty of integrating features taken from both volunteer and cloud computing platforms. Studies from the field of mobile clouds did however show more promising results, but aim to solve a different set of problems.

By building on previous research and identifying the key issues to be addressed, we outlined the extensions required to transform V-BOINC into an *ad hoc* cloud computing platform in Chapter 4. This primarily involved introducing new functionalities into the V-BOINC server and client in order to create the *ad hoc* cloud equivalents (Section 1.6, contribution 3). This included a simple BOINC job submission system (Section 1.6, contribution 4), an *ad hoc* cloud scheduler allowing cloud jobs to be executed and *ad hoc* guest checkpoints to be restored on near-optimal *ad hoc* hosts (Section 1.6, contribution 5) and a P2P checkpoint distribution mechanism (Section 1.6, contribution 6), respectively.

The latter is our primary contribution that introduces reliability into an unreliable infrastructure. This works by distributing periodic checkpoints of an *ad hoc* guest to other *ad hoc* hosts in the *ad hoc* cloud to allow the guest to be restored elsewhere in the event the original becomes non-operational. We then outlined possible solutions to minimize the affect cloud processes have on host processes followed by showing that, unlike regular BOINC, the *ad hoc* client and server components are easy to install and use due to the modifications we have made (Section 1.6, contribution 7). V-BOINC and all of the subsequent extensions aforementioned, realize our concept of the *ad hoc* cloud and make it a feasible computational alternative to commercial or private clouds, clusters and Grids.

A key facet of any computational infrastructure is resource monitoring, either for individual hosts or the entire infrastructure. However due to the architecture of the *ad hoc* cloud, where *ad hoc* hosts may migrate between cloudlets, it becomes difficult to monitor highly dynamic hosts within these groups; we believe no current monitoring tool is able to offer cloudlet-based monitoring due to being statically configured.

As cloudlet-based monitoring is not only useful in an *ad hoc* cloud setting, but also organizational settings that employ server clustering, we developed the Cloudlet Control and Monitoring System (Section 1.6, contribution 8). By extending Ganglia to allow cloudlet-based monitoring, as well as introducing additional metrics and a infrastructure management component, we showed that the C2MS does not introduce any overheads above those of Ganglia and can execute administrator-specified commands over a large infrastructure quickly.

7.1.3 Prototype Evaluation

We define an evaluation model that all *ad hoc* cloud prototypes should be measured against (Section 1.6, contribution 9). Primarily, we evaluated the reliability and performance of the *ad hoc* when all the aforementioned components are integrated and deployed on EDIM1, as well as a number of general purpose hosts (contribution 10).

Our experimental results showed that the *ad hoc* cloud was capable of successfully completing up to 93.3% of cloud jobs in the face of realistically simulated host churn or failure. Despite failing to meet our target of successfully completing at least 95% of all cloud jobs due to the errors introduced by BOINC and VirtualBox, we are confident that a cloud job would exceed our specified success threshold if these errors were remedied.

The performance of our *ad hoc* cloud deployed on EDIM1, as well as the overheads introduced by our implementation in comparison to V-BOINC, were found to be excessive. This was caused by the lack of hardware-assisted virtualization and the relatively low performance of the processors used in the cluster. Due to the large number of instruction calls to the hypervisor and the executing *ad hoc* guest and cloud job all under command of the executing *ad hoc* client, the low processing capacity available was not enough to satisfy computational demand. To prove that this is unique to EDIM1, we ran our *ad hoc* cloud in the same conditions on a number of general purpose hosts. This showed that both the performance and overheads are substantially reduced to acceptable levels.

Our evaluation then focussed on the features of the *ad hoc* cloud that could increase the total completion time of a cloud job. Pre- and post-execution overheads were shown to be large on EDIM1, which are unique to the platform, and low when an *ad hoc* cloud operates on general purpose hosts. Checkpointing overheads, namely the checkpoint penalty, are also low for compute, I/O and memory-intensive applications but can be high for disk-intensive applications. Similarly, the checkpoint size for the latter is found to be high when the checkpoint frequency is lower than 4; conversely checkpoint sizes were low for all other resource-intensive applications.

This therefore posed the question of how to determine the near-optimal checkpoint frequency based on minimizing the checkpoint penalty, the amount of data distributed over the network and increase reliability. We found that based on our benchmarking results, the near-optimal checkpoint frequency is 12 checkpoints per hour.

We then investigated whether the checkpointing traffic generated by this checkpoint frequency would degrade the performance of the EDIM1 network. Our results showed that the network could easily cope with 870 checkpoints being distributed at a time. Therefore is it reasonable to assume that commodity networks could also distribute a large number of checkpoints, if not already heavily utilized. The final overhead investigated was the affect of virtual machine restoration. Our results showed that if an *ad hoc* guest running a cloud job is detected as non-operational, it is possible to restore the guest elsewhere in under one minute.

By combining the affect of all overheads associated with an *ad hoc* cloud that increase the total completion time of a cloud job, we were able to offer a fair comparison between the performance of an *ad hoc* cloud and Amazon EC2 (Section 1.6, contribution 11). Our results found that despite all associated overheads, the performance of the *ad hoc* cloud can be similar, if not better, than Amazon EC2. During the course of our experiments, we evaluated the resource loads imposed on our *ad hoc* server deployed on a host with standard hardware specifications. While many experiments were performed over a long period of time, the server's resources were typically underutilized, proving that this single and centrally managed host would not become a bottleneck if the number of *ad hoc* hosts, guests and cloud jobs increased.

7.2 Future Work

Throughout the course of this research, we noted many improvements that could have been made to our approach of solving the aforementioned research challenges, the functionality offered by our *ad hoc* cloud and our evaluation methodology. We discuss these improvements in the following sections.

7.2.1 Approach

Our initial research into the feasibility of *ad hoc* cloud computing came at a time when open source private cloud computing platforms were in their early stages of development. However, the requirements of an *ad hoc* cloud were found to be similar to those present in volunteer computing and therefore, our chosen approach was to transform a volunteer infrastructure into a cloud platform. Nowadays, that open source private cloud platforms are more mature, it would be interesting to investigate a contrasting approach by transforming a cloud platform into an *ad hoc* cloud computing platform and then evaluate the merits and drawbacks of each approach.

After regular use of OpenStack [31], the conceptual architecture and many of its components now have similar purposes to those as part of an *ad hoc* cloud computing platform. The development of the latter via the modification of OpenStack therefore may be less difficult than our proposed approach and many of OpenStack's features would be exposed by default. For example, the authentication service Keystone, the virtual machine image service Glance and the networking service Quantum.

Whether modifying OpenStack to create an *ad hoc* cloud is a more efficient and an advantageous approach would have to be investigated further. For example, the system and software features complexities as well as the maintenance and performance overheads of the two approaches could be compared. Furthermore, the effort required to introduce reliability and its subsequent success could also be measured.

7.2.2 Additional Features

Throughout this thesis, we have outlined the additional features that should be added to our *ad hoc* cloud prototype to increase its usability, acceptability, reliability and performance. The features we would like to include in our research and implementation are:

- *Virtualization:* currently our *ad hoc* cloud computing prototype only allows one type of virtual machine to be distributed to the hosts within the infrastructure. We propose that *ad hoc* cloud users should have the ability to upload their own virtual machine, in turn allowing their submitted applications to run in an environment they choose.

  Due to the well known existence of virtualization performance overheads, which in some cases may be large, we propose to evaluate the performance overheads of other virtualization technologies for possible inclusion into both the V-BOINC and *ad hoc* cloud implementations. Furthermore, we propose to investigate methods of how to accurately suspend virtual machines based not only on non-BOINC CPU usage, but also memory, disk and network usage. This provides a basis to migrate *ad hoc* guests to other *ad hoc* hosts when the number of virtual machine suspensions is frequent signifying that performance of the host is poor. Currently, our *ad hoc* cloud implementation is only available for installation on Unix-based hosts; we plan to create platform installable for Windows platforms at a later date.

- *Performance and reliability:* the success of an *ad hoc* cloud deployment is primarily based on the performance and reliability of the implementation. We have shown that overheads caused by our *ad hoc* client can have an affect on the performance of the executing cloud job. Therefore it is vital that our implementation is optimized to reduce these overheads. Furthermore, we also plan to develop solutions to the errors caused by BOINC and VirtualBox that affected the success rate of cloud jobs during our experiments.

  The performance and reliability of our *ad hoc* cloud may be enhanced if task redundancy is also employed, therefore we intend to investigate whether both approaches can compliment one another. We also propose to investigate whether our current host reliability formula could be improved by taking into account an *ad hoc* host's recent reliability, instead of the calculating the host's reliability based on its behaviour since it first became a member of the *ad hoc* cloud; recent reliability may be a better indicator of an *ad hoc* host's true reliability.

- *Cloud job and checkpoint scheduling:* we propose to include an *ad hoc* host's current network usage into scheduling decisions that determine whether the host is suitable to execute cloud jobs. Furthermore, we also propose to include an *ad hoc* host's current load into scheduling decisions that determine whether the *ad hoc* host should receive a particular checkpoint.

  To improve the accuracy of scheduling checkpoints to near-optimal *ad hoc* hosts, we believe that it is important to factor in the checkpoint size and estimated transfer time. The reliability of the *ad hoc* cloud could be increased and the total bandwidth usage reduced by sending smaller checkpoints to a larger number of unreliable hosts and larger checkpoints to fewer more reliable hosts, while still satisfying the 95% cloud job success rate property. We believe that this checkpoint scheduling approach would improve reliability and reduce bandwidth usage between *ad hoc* hosts, however this hypothesis would have to be investigated.

  Furthermore, we propose to investigate the effectiveness of monitoring and storing previous cloud job execution times and resource usage patterns in order to predict the estimation time and resource use for a current cloud job based on similarity. This would allow cloud jobs to execute on suitable *ad hoc* hosts with the enough resources and that are usually available for the duration of a cloud job's predicted runtime. We believe this would substantially improve the accuracy of scheduling decisions and ultimately reduce the total completion time of a cloud job.

- *Network performance:* in order to ensure checkpoint transfer speeds are as low as possible, we intend to employ a number of reduction mechanisms such as checkpoint transfer deduplication, delta compression or checkpoint pre-copying, for example. We would have to investigate which mechanism effectively reduces checkpointing traffic and whether a combination of approaches could be used to further reduce the amount of checkpoint traffic distributed in an *ad hoc* cloud.

  Furthermore, we propose to dynamically adjust the checkpoint frequency during the execution of a cloud job in order to reduce the load on the network. We aim to firstly modify the checkpoint frequency not only based on the checkpoint size, capture time, desired reliability for a job and the reduction of load for CPU and network resources, but also the reliability of the *ad hoc* host executing the virtual machine. For example, the initial checkpoint frequency could be set proportional to the bandwidth available and the predicted checkpoint sizes. These could then be dynamically adjusted based on the number of jobs a particular host completes or fails. In such cases the checkpoint frequencies could be decreased and increased respectively, for example, by a factor of two until a future event occurs. An investigation into possible algorithms to adjust checkpoint frequencies would need to be undertaken.

Also, the current network load, monitored by the *ad hoc* server, can influence which *ad hoc* host's should be allowed to transfer a checkpoint at any given time. For example, if the network is congested, the *ad hoc* server may instruct only a select number of unreliable *ad hoc* hosts to distribute checkpoints until the network congestion has eased. Note that the checkpoint frequency can also be dynamically adjusted to increase reliability, reduce re-computation overheads or improve the performance of the *ad hoc* host.

- *Minimizing cloud process interference:* an important component of an *ad hoc* cloud, mentioned in a number of other studies, is how to effectively reduce the interference cloud processes have on the host processes. Previously we outlined possible methods to overcome this problem. We propose to include the use of the tool *cpulimit* to dynamically adjust the CPU resources available to cloud processes dependent on host process CPU usage. An investigation into other tools or methods that successfully control memory, network and storage consumption must be performed.

- *Usability:* Although installing, using and managing the *ad hoc* cloud was proven to be extremely simple, further improvements can be made. The familiarity of the BOINC Manager in the volunteer community makes it reasonable to suggest that our own GUI interface components should be integrated into the current BOINC Manager. We could also take advantage of popular platform interfacing tools such as WS-PGRADE/gUse and SCI-BUS [134]. In the event the *ad hoc* cloud becomes resource limited for short periods of time, we also propose to include mechanisms to outburst to another private or commercial clouds, provided they have a publicly accessible API.

7.2.3 Evaluation

Additional experimentation would need to be performed in the event the above features are integrated into our *ad hoc* cloud prototype. There are however further improvements that can made to our current evaluation methodology.

In order to determine the true benefits and drawbacks of our *ad hoc* cloud, a greater range of applications must be executed. Our research concentrated on executing simple applications, however those that require parallelism, write to external databases or simply have different resource usage patterns, for example, must be analysed when running on the *ad hoc* cloud. Furthermore, a greater number of platforms must be tested when operating an *ad hoc* cloud in order to provide a comparison between the platforms outlined in this thesis and others. This analysis would also determine whether the performance and overheads recorded on EDIM1 is a unique occurrence. Our simulated infrastructure also did not simulate CPU, memory, I/O and network resources being consumed by *ad hoc* hosts. We propose to execute random workload generators on each *ad hoc* host and determine the reliability and performance differences.

However, the true success of our *ad hoc* cloud prototype can only be measured by deploying the cloud on an unreliable but operational infrastructure, such as one within an organization or research institution, for a long period of time. By default, this would allow the *ad hoc* cloud to execute a wide range of applications and experience different situations involving host churn or failure. Furthermore, the feedback from the *ad hoc* cloud system administrators as well as host users and cloud users, would be invaluable to help improve the acceptability, usability and performance of our implementation.

7.3 Concluding Remarks

This thesis has proposed the concept and development on an *ad hoc* cloud computing platform. We initially hypothesised that the *ad hoc* cloud was a feasible, yet reliable alternative computational platform. Furthermore, we also hypothesised that the *ad hoc* cloud could offer at least reasonable performance, especially in comparison with commercial cloud infrastructures.

Throughout this thesis we have shown that the *ad hoc* cloud is a reliable platform even when operating over an unreliable infrastructure. We expect the reliability of our prototype to increase when the underlying technologies used as well as our own implementation improves over time. We have also shown that the performance offered by the *ad hoc* cloud is comparable to an Amazon EC2 instance with similar resources, despite operating on an unreliable infrastructure and with the unavoidable overheads associated with *ad hoc* cloud computing.

We found that CPU, memory and I/O-intensive applications are well suited to the *ad hoc* cloud, however disk and data-intensive workloads may not be, due to the large checkpoint penalties, ultimately affecting the reliability of the job. Checkpoint sizes were also typically found to be large resulting in a large amount of data being distributed over the network to ensure job continuity; this in turn may affect the performance and reliability of other *ad hoc* cloud jobs.

Therefore, based on an extensive investigation of the research issues, a complete implementation of an experimental prototype and hitherto unprecedented evaluation of the *ad hoc* cloud, *the ad hoc cloud is not only a potentially worthwhile form of computational provision, but also a viable platform* that provides a computational alternative to commercial or private clouds as well as clusters and Grid infrastructures.

Appendix A

Cloudy Waters: Tapping into the Unknown

This accompanying chapter sets out the additional background knowledge of cloud computing we acquired before development and evaluation of the *ad hoc* cloud.

A.1 Introduction

Introducing a new cloud computing paradigm first requires an in-depth analysis of the research surrounding cloud computing. This is especially important for our research as both the *ad hoc* and commercial cloud computing models share similar features. For example, users and their applications share the same hardware and resource contentions may arise; the *ad hoc* cloud does however try to minimize the affect of the latter.

In this chapter we analyse and benchmark Amazon EC2 to obtain a greater understanding of cloud computing and to determine whether the *ad hoc* cloud computing paradigm is a feasible concept when compared with commercial clouds; contribution 12 listed in Section 1.6 of Chapter 1. We chose Amazon EC2 as it is a popular IaaS cloud provider and offers Unix-based virtual machines.

Firstly we provide an overview of related work from the scientific community outlining the benchmarks performed on Amazon EC2. We then briefly discuss the cost-related difficulties when employing cloud computing in scientific settings. Next we describe our own benchmarking approach that aims to investigate the performance and cost variabilities of the cloud dependent on resource contention, time of day and the physical processor an instance uses.

This is followed by a discussion on how to improve cloud performance via instance underutilization. Finally we determine whether it would be possible to charge cloud users for using resources from the *ad hoc* cloud and discuss the challenges and likelihood of doing so.

A.2 Science on the Cloud

Due to the obscure nature surrounding commercial cloud providers and their underlying infrastructure and software solutions, there exists a large research pool that has investigated the cost and performance of these infrastructures.

The common finding from most cloud benchmarking research is that commercial clouds need to improve, especially for those in the HPC and scientific communities. Some studies have found that Amazon EC2 is capable of running certain scientific applications better than on high performance clusters [64] while others show that only a single EC2 instance offers comparable performance [168]. Other studies show that while this is unlikely, Amazon EC2 is able to match the performance offered by a local commodity cluster [99].

A large portion of research does however challenge these claims given that commercial clouds are designed for commercial use [117]. Furthermore, commercial clouds are known not to satisfy many of the performance requirements of scientific applications [117, 211, 121, 175, 127]. For example, Amazon EC2 is quoted as being up to 6 times slower than a commodity cluster and twenty times slower than a high performance cluster [130]. Resource contention may cause such performance differences [130]. Armbrust *et al.* find that CPU and memory can be shared surprisingly well [56] while others find this is not the case [125], especially when a cache is shared by multiple virtual machines [121].

Scientific applications typically rely on being able to achieve good disk I/O performance however Amazon EC2 is known to be poor in this respect [56] despite disk I/O performance levels varying between Regions and Availability Zones, due to the difference disks employed [194]. Virtualization overhead also may [175] or may not [117] reduce the performance of executing scientific applications. These applications are typically parallel in nature [99] and individual tasks must communicate and/or transfer data between each other. The poor network performance of Amazon EC2 [56, 117, 127, 130, 168, 194], which is comparable to a commodity Ethernet network [121], therefore significantly reduces the performance of scientific applications.

Studies have shown that MPI-based applications experience significant latencies of the order of one or two magnitudes slower than traditional clusters [99, 168, 194]. Until high performance interconnects are widely adopted within Amazon EC2, HPC applications will not be able to run effectively within Amazon EC2; this is especially true as applications become increasingly compute or data-intensive.

There are however cases where small-scale HPC applications are able to effectively run on Amazon EC2 [121, 99]. In order to potentially increase the performance of a scientific application running on Amazon EC2, it is possible to tune the application for use on the target platform [175]. Despite this, surprising some studies question the reliability of Amazon EC2 for scientific applications. For example, instances may fail to launch or network instabilities may cause instances to crash [168]. A cloud user's defined network configuration may also fail to take effect and instances may hang. A single error occurs enough that running scientific applications on Amazon EC2 can become difficult [130]. The single or combined effects of these performance and reliability issues may deter scientific researchers from migrating to the cloud.

Recently however, Amazon EC2 has attempted to address the concerns of scientific researchers and those that require increased CPU, memory, disk and network performance. In addition to Amazon EC2's general purpose instances, they now offer compute, memory and storage-optimized instances that are tailored towards offering a better service for applications; these instances typically have more of one resource but less of others [3].

For example, a compute-optimized instance offers slightly more CPU resources than a general purpose instance of the same size but has approximately half of the memory. Although this instance providers five times more computational resources than a small instance, studies have shown it is not five times faster [99]; it does however offer double the I/O performance. Despite the potential performance downfalls of these types of instances, they are designed to help a cloud user appropriately fit an instance to an application's ideal resource use. The situation is also similar with memory and storage-optimized instances. To increase the disk I/O performance and reduce the affects of resource contention, Amazon offer EBS-optimized instances which provide a dedicated throughput to an EBS volume at a chosen rate between 500 and 2,000 Megabits per second. Amazon have also recognized the network performance issues relating to scientific applications and have recently introduced Enhanced Networking.

Enhanced Networking introduces a higher packet per second performance, lower latencies and lower network jitter by using Single Root I/O Virtualization (SR-IOV).

SR-IOV allows a network adapter to appear as separate devices in order to be accessed effectively by many data sources [195]. Enhanced networking can only be used with HVM AMI compute and storage-optimized instances that have an appropriate driver installed and that are part of an Amazon Virtual Private Cloud (VPC); a logically isolated area of AWS that allows a cloud user to take advantage of having control of their own virtual network [3].

Similarly, some instances are able to take advantage of Cluster Networking where instances are grouped into a logical cluster to provide high-bandwidth and low latency connections between cluster instances [3]. This is different from Amazon's VPC where instances are operated within a segregated platform that allows virtual networking. Instances within a Cluster Network are run in a single flat network shared with other cloud users. However, perhaps the most important development to help those that require higher CPU, memory, disk and network performance is the introduction of Dedicated Instances. These instances are run on hardware dedicated to a single cloud user therefore eliminating the resource contention and security issues that arise from resource sharing. The per hour costs of these instances are slightly higher than Amazon's general purpose instances plus an additional $2 'Per Region' fee per hour.

At the time of writing, the performance and costs of using these cloud offerings has largely been untested by the scientific community however we would expect that some scientific applications would benefit from these features. Furthermore, as much of the related research mentioned above is a snapshot of the Amazon EC2 taken from over the last few years, we expect that as the number of technological developments will increase, a greater percentage of scientific applications will be able to run effectively on the cloud.

One such improvement that is vital to solve is the issue surrounding networking. Technological solutions such as Enhanced Networking will only delay the inevitable reduction in network performance during a time where applications are becoming increasingly data-intensive and the number of cloud users is increasing. Only an upgrade of the physical network will partially solve problems faced when executing scientific or HPC applications on the cloud. While these developments are aimed at encouraging more scientific users to the cloud, there is one other aspect that needs improving to achieve that goal; the issues surrounding recurrent costs and capital funding from UK Research Councils. Currently, the funding policies of UK Research Councils do not suit the typical cost model of cloud computing [97]. As the research costs of using the cloud as not known beforehand, UK Research Councils are unable to allocate funding to researchers and projects. Furthermore, it is not yet possible to charge research costs based on recurrent spending or through an individual researcher's credit card [116].

Cloud computing providers may be improving their infrastructures for scientific applications, however we will only see a substantial growth in the number of scientific researchers using cloud computing when the financial relationship between the researcher and funding body improves. Until such a time, the number of studies comparing the performance of the cloud to local commodity or high performance clusters will increase and so too will the differences of results obtained from these studies. Therefore it is important to note that similar comparisons of performance and cost should also be performed before researchers decide whether to migrate to the cloud.

A.3 Cloud Performance Variations

We have shown that there are sufficient studies relating to the benchmarking of Amazon EC2 and outlined the common results obtained from each evaluation. A few of these studies suggest that moderate to extreme performance variabilities exist when running applications on the cloud. This may be due to contention for non-virtualized resources [194, 99], the physical processors that instances must use [127, 130], the scheduling of virtual machines to different physical servers [194], or even a cloud user's application. At a time when cloud benchmarking was popular (2010/2011), we also measured the extent of these performance variabilities to determine those that might be shared when running applications on the *ad hoc* cloud where hardware is shared and resource contention may occur.

A.3.1 Resource Contention and The Time of Day

We now outline the experiment performed to determine if existing performance and cost variabilities are caused by either the time of day or resource contention.

Our experimental setup was as follows: we instantiated a small Standard On-Demand General Purpose instance within the *us-east-1a* Availability Zone which has the AMI ID *ami-a6f504cf* (this is now unavailable but appears to have changed to *ami-e2f67bd2*); this AMI has the OS Ubuntu Maverick i386 server installed. SPRINT's *pcor* function (see Section 2.6.5 of Chapter 2) was executed 13 times at various times throughout a day with the number of genes and samples set to 11,000 and 321 respectively. The function's execution time was output on completion and stored for analysis.

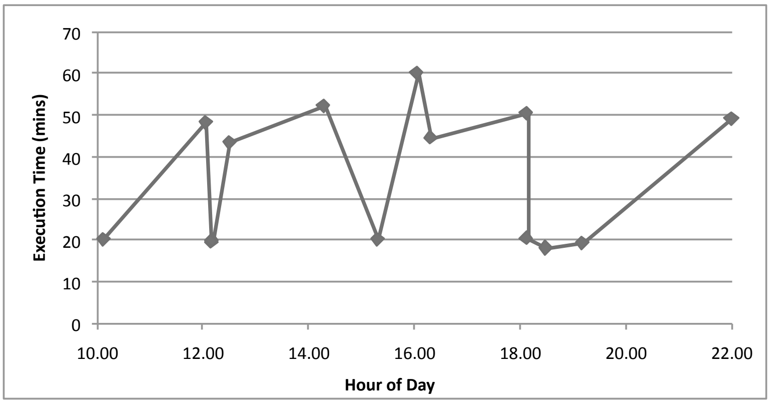


Figure A.1: Execution Time vs Time of Day (m1.small)

Figure A.1 shows that an application's execution time can vary significantly. In our case, we experienced a minimum and maximum execution time of approximately 19 minutes and 61 minutes respectively when executing SPRINT over a 12 hour period. This performance gap is not only inconvenient but also has cost implications for the cloud user when instances are charged per hour. A researcher executing SPRINT at 16:00 on that particular day, would have had to wait 61 minutes for the job to complete and despite only requiring the instance to be available for one more minute extra, would have been charged double when compared to executing the same job at other times throughout the day.

Iakymchuk *et al.* also conduct a similar investigation but in greater detail [125]. The authors execute the DGEMM application, which calculates the product of double precision matrices, on an extra large compute-optimized instance (c1.xlarge) in an unspecified Availability Zone. Their results are shown in Figure A.2.

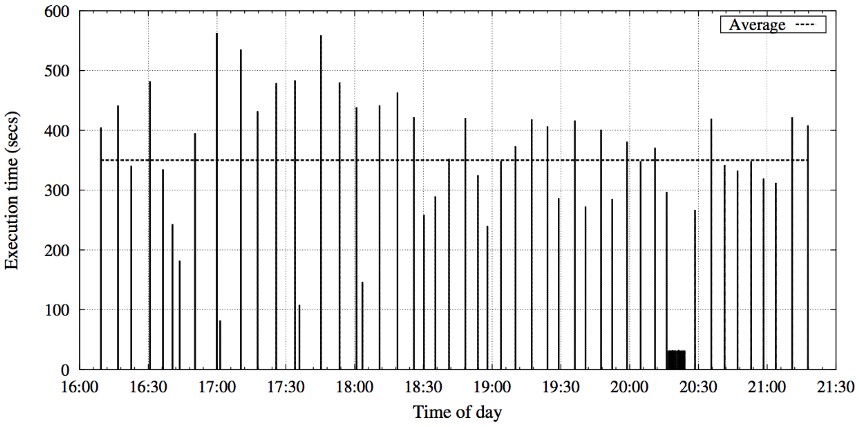


Figure A.2: Execution Time vs Time of Day (c1.xlarge) [125]

Those authors results, as well as our own, show significant performance and subsequent cost variabilities that are seen when executing an application multiple times throughout the day. This also shows that such variabilities are not only limited to one single instance but are common to different instance types, and indeed all general purpose instance types.

Similarly, these variabilities are not only a unique characteristic of SPRINT and can affect the performance of any application running on Amazon EC2. We must note that the results from Iakymchuk *et al.* and our own do not imply that an application's execution time is equal at the same time each day; they vary significantly dependent on a number of factors that are out of the cloud user's control. In fact, it is likely that the performance delivered is independent of the time of day itself; although there may be times where the cloud is more busy than others which in turn will have a slight effect on performance [194, 127]. The performance variabilities exhibited when running SPRINT are likely to be caused by CPU, memory and disk I/O resource contention on the physical host; negligible network bandwidth was consumed by SPRINT in our experiment.

Therefore we have shown that the performance of a single application contained on a single EC2 instance can suffer significant performance variabilities; in many cases this will increase the costs charged to a cloud user's monthly bill. This investigation also gives us a better understanding of what type of applications are not suited to the commercial cloud, for example, those that need to be completed before a strict deadline. Such applications may also not be suited to the *ad hoc* cloud where minimal resource contentions arise and the performance can be greatly affected by the termination or failures of *ad hoc* hosts and *ad hoc* guests.

A.3.2 Instance Processors and ECUs

Amazon EC2 introduced the concept of an Elastic Compute Unit (ECU) to provide standardized and consistent CPU performance for EC2 instances. As a reminder, an ECU provides the equivalent CPU performance of a 1.0-1.2 GHz 2007 AMD Opteron or Intel Xeon processor. For example, if two instances each have 2 ECUs of compute capacity and are deployed onto two different types of CPU, in theory they should offer equivalent performance if no other resource contentions arise. We outline an experiment to determine whether an ECU offers equivalent performance regardless of the different types of physical processor an instance uses.

We first determined the number and types of processors used when running a large Standard On-Demand General Purpose instance (m1.large) in the *us-east-1d* Availability Zone. Note that a cloud user is not able to select the processor their instance uses hence this was performed by trial and error. In order to determine the physical CPU type of a running instance, we ran the following command on the instance:

```
cat /proc/cpuinfo | grep "model name"
```

We found that a large instance uses two physical processors of the same type. As a reminder, a large instance has two virtual cores each with 2 ECUs of compute capacity, i.e. 2.0-2.4 GHz per core. Therefore one physical processor, or a portion of it, is equivalent to one virtual core of a large instance. We also found that the types of physical processor can be different on a per-instance basis; a number of studies describe other processors used in Amazon's infrastructure [130, 194]. The three processor types commonly used when deploying large instances in the *us-east-1d* Availability Zone are shown in Table A.1.

CPU Type	Min. Usage	Max. Usage
Intel Xeon E5507 2.27 GHz	88.1%	100%
Intel Xeon E5645 2.4 GHz	83.3%	100%
Intel Xeon E5430 2.66 GHz	75.1%	90.2%

Table A.1: Large Instance Physical CPU types and Per-Core Usage Levels

We see that a large instance may use either two Intel Xeon E5507, E5645 or E5430 processors each with varying cycle-per-second frequencies. Table A.1 also shows the calculated minimum and maximum utilization rates of each processor to deliver 2 ECUs of compute capacity per core. We calculate these rates using basic mathematics. For example, in order for a virtual core to at least achieve the minimum performance of 2 ECUs (i.e. 2 GHz), 88.1% of the E5507 2.27 GHz processor should be utilized when the instance requires it. Similarly, in order for a virtual core to achieve the maximum performance of 2 ECUs (i.e. 2.4 GHz), all of the E5507 2.27 GHz processor should be utilized; these utilization rates are calculated for each of the other processor types.

In the case of the E5430 2.66 GHz processor, we see that it offers a compute capacity greater than the maximum limit of 2 ECUs (i.e. 2.4 GHz). Therefore a maximum of 90.2% of the processor must be utilized. In order to ensure the minimum and maximum utilization rates of each processor are adhered to, Amazon's EC2 infrastructure steals CPU cycles from processors. The percentage of cycles stolen can be verified by running the *top* command on UNIX-based instances [99]. In our case, the steal percentage *%st* was as we expected. To determine whether a virtual machine offers consistent performance regardless of which physical processor it uses, we ran a benchmark over the various processor types and analysed its execution times and CPU loads.

Our experimental setup was as follows: we instantiated three Standard On-Demand General Purpose large instances which had the AMI ID *ami-a6f504cf* (this is now unavailable but appears to have changed to *ami-e2f67bd2*); this AMI has the OS Ubuntu Maverick i386 server installed. These instances were run in the *us-east-1d* Availability Zone and each had three different physical processors selected from those in Table A.1. The instances executed the SPRINT functions *pcor* and *pmaxT*. The former processed a randomly generated dataset consisting of 11,000 genes and 321 samples and the latter processed a dataset consisting of 1000 genes and 50 samples with the number of permutations set to 150,000.

Both functions spawned two processes to utilize each core of the instance. During the execution of each function, we measured their execution times as well as the average and peak CPU utilization rates for the various processors. The CPU load was measured by polling the CPU for usage information every second by capturing the output from the command *top* and these values were confirmed by Ganglia. The average and peak utilization rates for each processor were then calculated. This experiment was performed five times and the results were averaged. We display 95% confidence intervals to show that in most cases, the true mean will lie within the specified range; some confidence intervals may not appear due to the small variations between runs. Furthermore, the experiment was performed multiple times on different days to ensure the overall conclusions were valid and not specific to a certain day and time.

Figure A.3 shows the execution times of the SPRINT functions dependent on the underlying physical processor and both the average and peak CPU loads each function was able to achieve. Our results clearly show that the execution times of the SPRINT functions are dependent on which processor an instance is set to use. Iosup *et al.* also find that one factor causing performance variability is the underlying processor [127] while Schad *et al.* and Jackson *et al.* find that Intel Xeon processors offer the best performance when compared to AMD Opteron processors [194, 130]. Our results show that the E5645 2.4 GHz processor offers the best performance for *pcor* which on average used 90% of both CPU cores and was able to reach a peak utilization of 96%; executing *pcor* on this processor met the minimum usage levels set out in Table A.1.

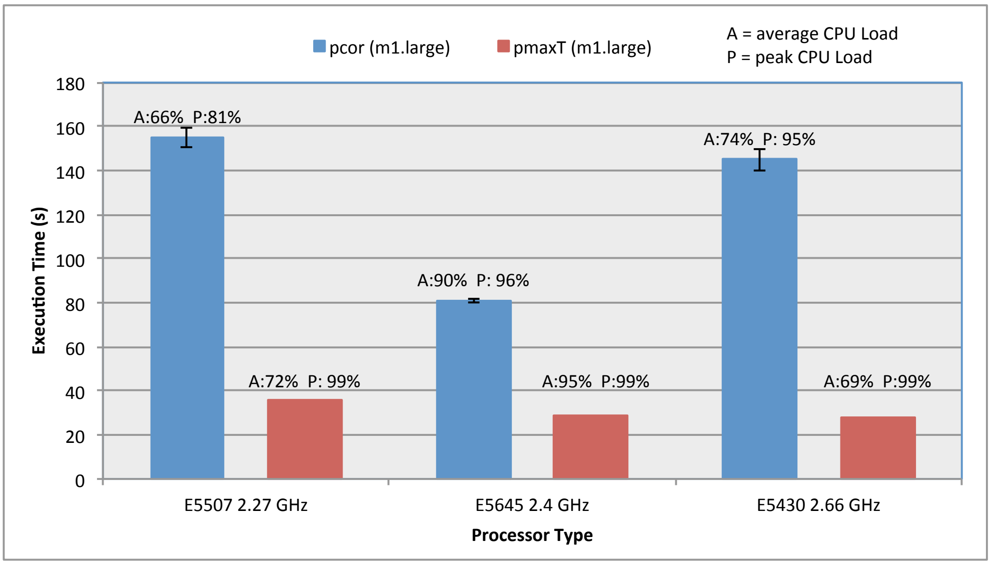


Figure A.3: Average and Peak CPU Loads Achieved by *pcor* and *pmaxT*

The E5507 2.27 GHz processor offered the poorest performance where *pcor* was only able to achieve an average CPU utilization of 66% and a peak utilization of 81%; both of which are lower than the minimum usage levels to offer 2 ECUs per core. The E5430 2.66 GHz processor offered reasonable performance, however *pcor*'s average utilization did not meet the required minimum usage of 75.1%. The function did manage a peak utilization of 95% showing that when the CPU is under-utilized, a higher share of the CPU can be used when available [194].

Intuitively it is reasonable to assume that the fastest processor will offer the best performance and conversely, the slowest processor will offer the least performance. However due to the CPU stealing mechanisms employed by Amazon, the E5507 2.27 GHz and E5430 2.4 GHz processors have more CPU cycles stolen from them when executing *pcor*. Interestingly, the least performing processor has the greatest number of CPU cycles stolen. This is shown by the relative average and peak CPU loads *pcor* was able to achieve on the E5507 2.27 GHz processor. One would expect a slower processor to be fully utilized if it were to offer 2 ECUs per core as Amazon specify.

The execution of *pmaxT* shows similar results, however the differences in performance between the processors are much less. Once again, the E5507 2.27 GHz processor offers the least performance with an average CPU utilization rate of 72%; a figure that suggests the physical processor does not offer 2 ECUs to the virtual machine. Both the E5430 2.4 GHz and E5430 2.66 GHz processors complete the *pmaxT* function in approximately 30 seconds and have an average CPU utilization rate of 95% and 69%

respectively; the latter does not meet the minimum usage target set out in Table A.1 All processors in this case are however able to achieve a peak CPU utilization of 99%.

These results not only show that the underlying processor an instance uses can effect the completion time of an application but the application may also be suited to a particular processor. Our results show that *pcor* is suited to the E5430 2.4 GHz processor and *pmaxT* is suited to either of the E5430 2.4 GHz or E5430 2.66 GHz processors. Furthermore, we see the small variation of execution times for the processors and particularly the E5645 2.4 GHz processor when executing both *pcor* and *pmaxT*; this shows that it is possible to achieve consistent application performance on Amazon EC2. Based on our results, it is reasonable to suggest that the E5430 2.4 GHz processor performs best overall when executing the *pcor* and *pmaxT* functions. This experiment was performed on other sets of m1.large and m1.xlarge instances and our findings were the same each time.

Due to the large infrastructure Amazon offers to its cloud service as well as the dynamic conditions the infrastructure is faced with, we are unable to select a processor that offers the greatest performance for all sets of applications in our selected Availability Zone. Therefore, in order for a cloud user to get the best performance for their application, a cloud user must perform preliminary experiments, such as those outlined above, to determine the processor that best suits their application. We believe this places a high expectation on cloud users to firstly have the expertise to perform such tasks and to secondly contribute a large amount of time and effort to find the best cloud configuration for their application.

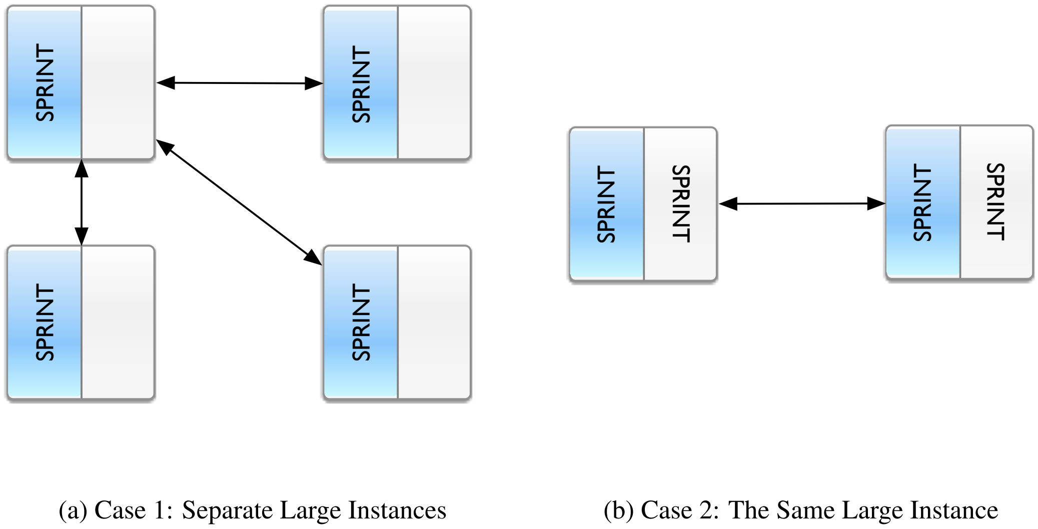
A.3.3 Instance Underutilization

We have shown that both resource contention, perhaps influenced by the time of day, and the processor used by an instance can affect the performance on an application. These are problems that cloud users are typically unaware of or are initially unable to address. We discuss one approach to increasing an application's performance by using instance underutilization. We define this as *reserving a larger instance than required and only using a small percentage of the available resources* [178].

The concept of underutilization is based on reserving a larger instance than required into order to reduce the interference caused by other instances resident on the same physical server. Reserving more resources than required may however introduce additional costs for cores that are not needed. One important study has shown that instance underutilization is an effective method to increase performance and reduce costs [125], however another was unable to see any improvements [121]. We investigate this concept further and offer our own perspective on using underutilization to increase performance and potentially reduce costs.

Our experimental setup was as follows: we executed the SPRINT functions *pcor* and *pmaxT* on a varying number of large Standard On-Demand General Purpose instances (m1.large) located in the *us-east-1d* Availability Zone. The instances had the AMI ID *ami-a6f504cf* (this is now unavailable but appears to have changed to *ami-e2f67bd2*); which has the OS Ubuntu Maverick i386 server installed. The function *pcor* processed a randomly generated dataset consisting of 11,000 genes and 321 samples while *pmaxT* processed a dataset consisting of 1000 genes and 50 samples; the function performed 150,000 permutations.

To test the effectiveness of underutilization, the experiment was split into three parts. First SPRINT was executed five times using two cores but spread over a different number of instances. The remaining two parts also involved executing SPRINT five times when using four and eight cores but again spread over a different number of instances. We give an example of how we test the effectiveness for underutilization in Figure A.4.



(a) Case 1: Separate Large Instances

(b) Case 2: The Same Large Instance

Figure A.4: SPRINT Running on Large Instance 4 Cores

For example, if four SPRINT processes are to be executed (i.e. one on each core), we can test underutilization by executing SPRINT over both four and two large instances. In the former scenario, SPRINT consumes one core on each of the four large instances as shown in Figure A.4(a). In the latter scenario, SPRINT consumes all four cores available from the two large instances as shown in Figure A.4(b). In order to test underutilization, the number of instances to use can be expressed by the following formulas:

- Underutilization: *No. of Large Instances = Number of Cores Required*
- Full utilization: *No. of Large Instances = 0.5 * Number of Cores Required*

By instantiating an equal number of processes and cores in both scenarios, we can test whether reserving more resources than required can reduce an application's completion time as well as potentially reduce costs. In this experiment, we ran the SPRINT functions using between two and eight cores; each run was performed five times and the average values taken. Furthermore, the experiment was performed multiple times on different days, as well as in different Availability Zones, to ensure the overall conclusions were valid and not specific to a certain day and time of set of physical servers. We show the results for both *pcor* and *pmaxT* in Figures A.5 and A.6 respectively.

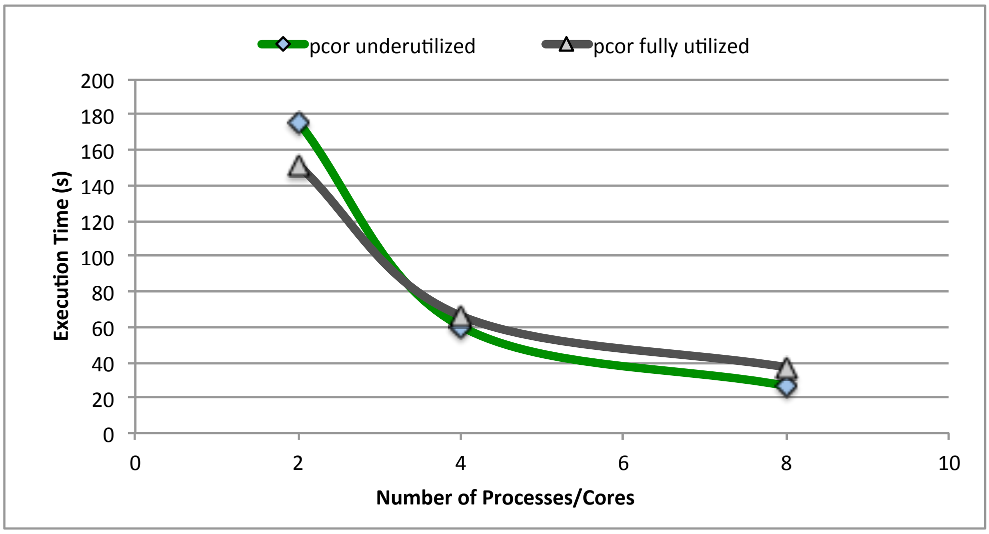


Figure A.5: *pcor* Underutilization

At a first glance, we see from Figure A.5 that there is little difference between *pcor*'s completion times regardless of whether underutilization is employed or not. Underutilization actually increases the execution time when running two *pcor* processes over two large instances when compared to using the same number of cores on two large instances. This is attributed to the network communication involved between the instances but also Amazon's poor network performance. As the number of cores and processes increase, underutilization slightly reduces *pcor*'s completion time by approximately 11 seconds when eight cores are used.

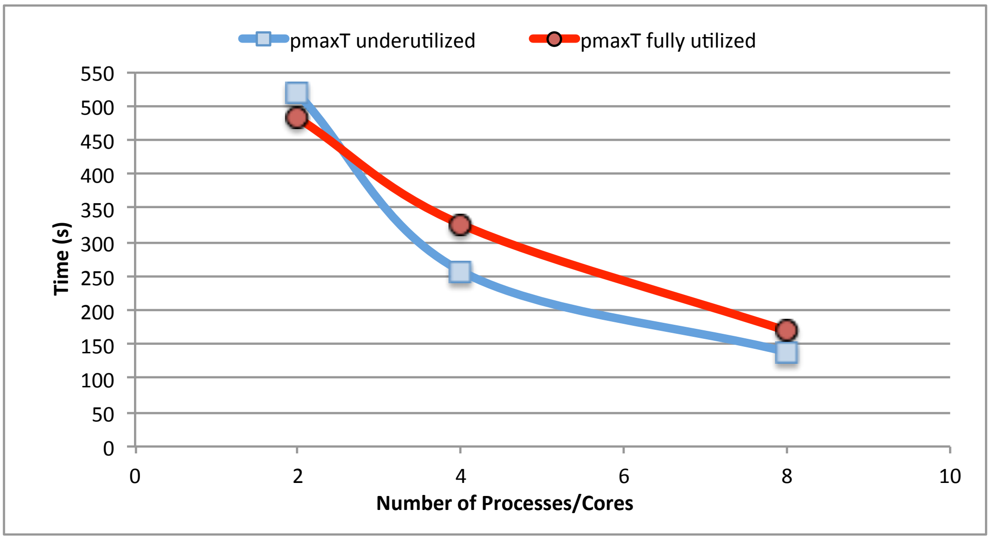


Figure A.6: *pmaxT* Underutilization

On the other hand, we see from Figure A.6 that the execution of *pmaxT* shows more promising results. Similar to the execution of *pcor*, employing underutilization initially increases the completion time of the function, however, underutilization does allow *pmaxT* to execute much faster when the number of cores is greater than two. The performance gap is most noticeable when four *pmaxT* processes are executed using both underutilization and full utilization where the former completes the *pmaxT* function approximately 70 seconds faster. Similarly, by employing underutilization, *pmaxT* completes approximately 30 seconds faster when 8 cores are used.

It is possible the performance gaps mentioned may save the total instance hours needed therefore reducing costs. Over a long period, these savings may turn out to be substantial. For example, the 70 second time difference between the four core scenario of running *pmaxT* in the two different configurations, could increase running costs by $0.68 per execution if underutilization is not employed. If this job were repeated multiple times per day each day, the money potentially saved via underutilization could be substantial after the period of one year. For small cash-flow sensitive businesses or research institutions, reducing the time and money consumed while running jobs, should be of great importance.

Iakymchuk *et al* found that underutilization can reduce an application's execution time by two orders of magnitude when running a series of benchmarks on Amazon

EC2's compute-optimized extra large (c1.xlarge) instances [125]. This performance gap will no doubt allow cost savings to be made when using Amazon EC2. Hence, this raises interesting research questions on whether cloud users should spend more time, effort and money to find their optimal job configuration to lower costs overall, over a longer time period.

We have shown that it is possible to use underutilization to obtain greater application performance, which in turn may reduce the amount charged to the cloud user. Although commercial cloud infrastructure details are unpublished, the effects of employing underutilization are likely to be caused by the reduction in resource contention. The benefits of underutilization are likely to increase when the size of the instance increases where more of the physical server is occupied, however this remains as future work to determine whether this hypothesis is correct.

A.4 A Pay As You Go *Ad hoc* Cloud?

Previous studies have proposed a cloud model where clouds are created from volunteer resources and cloud users are charged for utilizing these volunteer resources [85]. We now investigate whether the *ad hoc* cloud computing could also employ the 'pay-as-you-go' charging model. We do this by determining the difficulties and subsequent cost variations of the charging model offered by Amazon EC2.

A.4.1 Charging for Data Usage

Amazon EC2's 'pay-as-you-go' charging model charges for instance use per hour and storage and data transfer per GB. We expect charging for instance and storage use to be relatively simple when compared to charging for data transfer; instance hours and the amount stored on Amazon's infrastructure can be counted per account.

Charging for data transfer however is slightly more complex as this involves accurately monitoring the number and size of packets sent from an instance, either directly to another instance or to a customer's web browser. Data transfer monitoring and metering is further complicated by the various types of data transfers between instances in different Regions and Availability Zones; for example, IDT and RDT as previously mentioned in Section 2.3.3 of Chapter 2. We test whether the data transfer charging model implemented by Amazon EC2 can accurately monitor and charge cloud users for the amount of data they transfer.

Our experimental setup was as follows: we instantiated a single small Standard On-Demand General Purpose instance (m1.small) in the Availability Zone *us-east-1b* which has the AMI ID *ami-a6f504cf* (this is now unavailable but appears to have changed to *ami-e2f67bd2*); this contains the OS Ubuntu Maverick i386 server. We then sent small amounts of data to the instance from various sources and measured whether Amazon EC2 could correctly determine the amount of data transferred; we assume that large amounts of data transferred are easy to record and hence are accurately charged for. The volumes of data sent to the instance was on the order of MBs and we used the installation of SPRINT as our example. The packages required to run SPRINT on the instance are taken from both the EC2 Ubuntu repository and our own local machine. The instances therefore receivies 84.3 MB of RDT and 3.6 MB of IDT data respectively; this gives a total of 87.9 MB being transferred to the instance.

These packages were sent to the instance every hour and after each hour, we were able to determine the amount Amazon thought was transferred via the Usage Report. This was compared with the data transfer measurements taken from *tcpdump* [38] and Amazon CloudWatch [2]. The former is a command line packet analyser for monitoring server communication and the latter is Amazon's own implementation for monitoring AWS services. Our results are shown in Figure A.7.

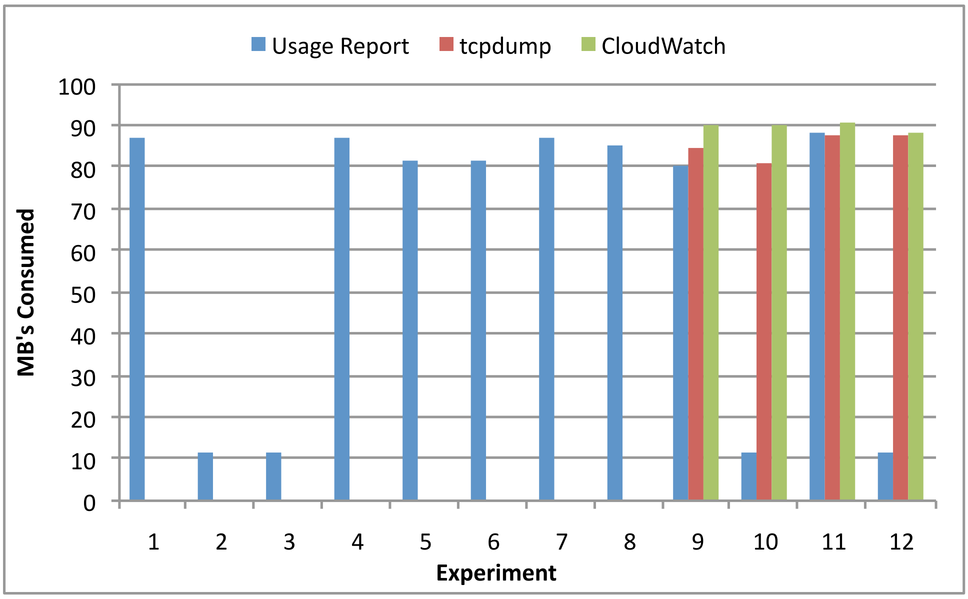


Figure A.7: Data Transfer Measurement: EC2 Usage Report, *tcpdump* and CloudWatch

Figure A.7 shows twelve equal data transfers taking place. The first eight of those display the number of MBs Amazon thought were transferred to the instance; these figures were obtained from the EC2 Usage Report. Despite transferring the same amount of data in each experimental run, we see that Amazon does not correctly record the volume of data transferred in a lot of cases. For example, Amazon's billing mechanisms only records that the instance only receives 11 MB instead of 87.9 MB of data.

The variations seen over the first eight runs then prompted us to use *tcpdump* and Amazon CloudWatch to determine whether the figures from the Usage Report were correct. For example, the discrepancies of results could have been a result of some caching mechanisms employed by Amazon. By using both *tcpdump* and CloudWatch, we see that the data is actually received by the instance therefore ruling out any reasons regarding caching. For example, experimental runs ten and twelve show that Amazon EC2's charging mechanisms do not record 87.9 MB of data being transferred however the dedicated CloudWatch service and *tcpdump* do.

The recorded values from both of these services also fluctuate, however in the case of *tcpdump*, the fast arrival rate of packages meant it was unable to record all data incoming to the instances. Based on the size and number of dropped packets, at least a total of approximately 87 MB would have been received. CloudWatch on the other hand records greater than 87.9 MB being transferred to the instance in some cases; this is likely to be due to data being transferred from an SSH connection initiated from our local client.

We also examined the accuracy of the data recorded by Amazon EC2 when transferring data out from the instance; the results were also similar where KBs or MBs were unaccounted for. This not only occurs for IDT transfers but also for RDT transfers between Availability Zones. As predicted, we were only charged for the data transfers that Amazon EC2 recorded and displays in their EC2 Usage Report. Therefore cloud users can take advantage of potentially cheaper data transfers if they have applications that periodically transfer small amounts of data to other instances within Amazon EC2 or beyond it.

A.4.2 The Effects of End-User Location

Regional cost differences, dependent on where an instance is deployed, are known to exist in Amazon EC2. For example, instantiating two instances in different Amazon Regions will be charged at two different rates. We, however, chose to test whether differences exist dependent on a user's job submission location and whether they are significant enough to make it advantageous for an end-user to submit a job to Amazon

EC2 from one location rather than any other. To examine such differences, we submitted a job from two widely separated and distinct locations in the world as shown in Figure A.8; the UK and Thailand.

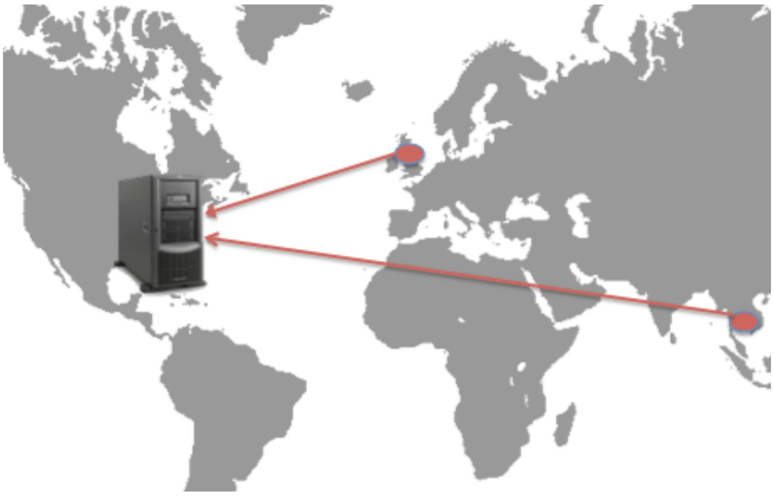


Figure A.8: Experiment Setup

Our experimental setup was as follows: we ran our the SPRINT function *pcor* over two large Standard On-Demand General Purpose instances (m1.large); the function processed a randomly generated dataset consisting of 11,000 genes and 321 samples. The instances had the AMI ID *ami-9b9091ef* and were located in the US East Region within the *us-east-1b* Availability Zone. In order to ensure a fair and consistent experiment was performed by both submission locations, a collection of scripts was created to automatically instantiate instances, setup the experiment, run the experiment and teardown instances.

In collaboration with Dr Sornthep Vannarat, Head of the Large Scale Simulation Laboratory from the National Electronics and Computer Technology Center (NECTEC), the experiment could be performed in Thailand. Once the computation was complete in both locations, the Amazon invoices and the AWS Usage Reports were obtained. To ensure the results were valid, confirmation was required that Dr Vannarats Amazon EC2 account was tied to an address in Thailand, otherwise if not, different charges could be seen. We show the costs and resource usage details in Table A.2.

We can see that the total cost of running two large instances for the same period of time, including other associated costs such as data transfer and I/O requests, is more expensive when the job is submitted from the UK than in Thailand. This is caused by the level of taxation in the two countries where at the time of writing, UK charges Value

Location	Cost	Data In	Data Out	Storage	I/O Req.
Scotland	$2.52	0.274 GB	0.008 GB	0.151 GB	46,523
Thailand	$2.10	0.205 GB	0.007 GB	0.151 GB	84,103

Table A.2: Difference in Resource Usages across Experiments

Added Tax (VAT) at 20%. This explains why running the job from the UK increases the costs by in $0.42, whereas Thailand charges no taxes. For small and cash-flow sensitive businesses and research institutions, this difference may have a significant impact on growth, for example, a business spending $4000 on cloud costs per month. For one year of use, the contribution to VAT at 20% would be $9600; more than two months of cloud usage.

In the case of Amazon EC2, these taxes are calculated based on the address of a users account allowing the cloud user to outsource computation to a tax-free region in order to reducing the direct cost of the final service. Taxes are not the only aspect that can affect cost variations. Location affects currency exposure, and as currencies vary in relation to each other, this changes the final price upon payment to Amazon in American Dollars; a process of conversion that is also charged for by the bank.

In addition to cost variations, Table A.2 also shows variations in the levels of Data In, Data Out and I/O Requests. Submitting the SPRINT job from the UK incurred 70MB's of extra data transferred inwards to the US Region, accounting for an addition of $0.01 compared to the Thailand run. This is either caused by data retransmissions when transferring data to the instances or further proves that Amazon does not correctly record data transfer.

We also see that there are 37580 fewer recorded I/O requests from the UK, accounting for $0.01 less than its counterpart therefore levelling the costs incurred due to resource use variation; the costs in Table A.2 therefore show only the differences due to tax. Why the number of I/O requests differ significantly is likely a result of EC2's underlying storage reading and writing mechanisms however experimentation to uncover the exact cause of this variability is future work.

Furthermore, by submitting the job from Thailand, *pcor*'s execution time took on average 79 seconds longer to complete. We attribute this to the relative differences in the cloud load at the times the experiments were performed.

A.5 Summary

In this section, we have shown that performance variabilities exist due to resource contention, perhaps the time of day the cloud is used and the processor that an instances uses. Performance can be enhanced by employing instance underutilization however this is likely to be dependent on an application's resource usage. The results from the three investigations show the many uncertainties that exist surrounding cloud computing.

If a cloud user of Amazon EC2 wishes to achieve near-optimal application performance, he or she must have in-depth knowledge of their application and hope that the EC2 instance scheduler selects a near-optimal physical server, both in terms of available hardware and resource contention. Furthermore, the cloud user must then determine and select the most appropriate processor and level of underutilization. This therefore places a huge workload upon cloud users and especially scientific application users, if they want to achieve the greatest levels of performance from their application.

Additionally, the results from the other studies make this task extremely difficult where the instance choice and overall cloud configuation are critical. For example, large instances can outperform an extra large instance for MPI applications [121] or performance variability can differ dependent on the Availability Zone used [194]. Therefore it is extremely unlikely that a cloud user is able to knowingly achieve optimal performance for their application.

Unfortunately, this is a problem that is likely to plague other commercial cloud infrastructures as well as the *ad hoc* cloud; resource contention, the time of day and the underlying processor a virtual machine uses will also affect performance in these cases. Therefore, as these downfalls are not unique to the *ad hoc* cloud, it therefore has the potential to offer equivalent performance to commercial clouds if the research challenges outlined in Chapter 1 have been overcome.

We have also shown that various cost variations can exist on Amazon EC2. Cost variations may be caused by data transfer not being accurately recorded and therefore correctly charged or by the location of a cloud-user and the local tax rates. These cost variations may benefit the cloud user if they reside in a tax-free Region or have applications that send small amounts of data over the network which may or may not be recorded by Amazon EC2.

However it is highly unlikely that a charging model could be integrated into an *ad hoc* cloud computing infrastructure. The fact that cost variations are present clearly show the difficulty of monitoring and metering cloud resources. Furthermore, charging end-users of an *ad hoc* cloud may not be possible due to the concerns surrounding the security of local resource monitoring systems that could be modified to inflate charges. The situation would be further complicated when an *ad hoc* cloud is distributed over countries with different tax systems. Although the concept of a ‘pay-as-you-go’ *ad hoc* cloud is appealing, many research challenges would have to be overcome hence we leave this for future work.

Bibliography

[1] 7-Zip. http://www.7-zip.org/. Accessed: July 2014.

[2] Amazon EC2 Cloudwatch. http://aws.amazon.com/cloudwatch/. Accessed: April 2014.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/. Accessed: April 2014.

[4] Amazon Web Services. http://aws.amazon.com/. Accessed: April 2014.

[5] Animoto Amazon EC2 Use Case. http://aws.amazon.com/solutions/case-studies/animoto/. Accessed: April 2014.

[6] BOINC VBoxApps. http://boinc.berkeley.edu/trac/wiki/VboxApps. Accessed: April 2014.

[7] BOINC Website. http://boinc.berkeley.edu/. Accessed: April 2014.

[8] C2MS Demo and Installable. http://garymcgilvary.co.uk/c2ms. Accessed: May 2014.

[9] Capistrano. http://capistranorb.com/. Accessed: April 2014.

[10] CernVM/VBoxWrapper Test Project. http://boinc.berkeley.edu/vbox/. Accessed: April 2014.

[11] cpulimit. http://cpulimit.sourceforge.net/. Accessed: April 2014.

[12] Data Intensive Research Group. http://research.nesc.ac.uk/. Accessed: May 2014.

[13] David Chou Microsoft Architect. http://blogs.msdn.com/b/dachou/archive/2009/01/13/cloud-computing-and-the-microsoft-platform.aspx. Accessed: April 2014.

[14] EDIM1. https://www.epcc.ed.ac.uk/facilities/other-facilities/edim1-data-intensive-machine. Accessed: May 2014.

[15] EDIM1 Network Diagram. https://www.wiki.ed.ac.uk/display/DIRC/Home. Accessed: May 2014.

[16] EPCC. https://www.epcc.ed.ac.uk/. Accessed: May 2014.

[17] Eucalyptus. https://www.eucalyptus.com/. Accessed: April 2014.

[18] FutureGrid. https://www.futuregrid.org/. Accessed: April 2014.

[19] Ganglia. http://ganglia.sourceforge.net/. Accessed: May 2014.

[20] GNU dd. https://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html. Accessed: April 2014.

[21] Google AppEngine. http://code.google.com/appengine/. Accessed: April 2014.

[22] gUse Public Portal. http://guse.hu/. Accessed: April 2014.

[23] HPCC Benchmarks. http://icl.cs.utk.edu/hpcc/. Accessed: April 2014.

[24] iperf. http://iperf.fr/. Accessed: April 2014.

[25] Liferay. http://www.liferay.com/. Accessed: April 2014.

[26] Microsoft Azure. https://www.windowsazure.com/en-us/. Accessed: April 2014.

[27] Nagios. http://www.nagios.com/. Accessed: April 2014.

[28] Ofcom. http://www.ofcom.org.uk/. Accessed: April 2014.

[29] Open Grid Forum. http://www.ogf.org/. Accessed: April 2014.

[30] Open Grid Services Architecture. http://toolkit.globus.org/ogsa/. Accessed: April 2014.

[31] OpenStack. https://www.openstack.org/. Accessed: April 2014.

[32] PlanetLab. http://www.planet-lab.org/. Accessed: April 2014.

[33] pscp. http://www.theether.org/pssh/. Accessed: April 2014.

[34] RRDTool. http://oss.oetiker.ch/rrdtool/. Accessed: April 2014.

[35] SPEC Benchmarks. http://www.spec.org/. Accessed: April 2014.

[36] SPRINT. http://www.r-sprint.org/. Accessed: April 2014.

[37] Stress Workload Generator. http://people.seas.harvard.edu/~apw/stress/. Accessed: April 2014.

[38] tcpdump. http://www.tcpdump.org/. Accessed: April 2014.

[39] V-BOINC Information. http://garymcgilvary.co.uk/vboinc.html. Accessed: April 2014.

[40] VirtualBox. https://www.virtualbox.org/. Accessed: April 2014.

[41] VMWare Player. http://www.vmware.com/products/player/. Accessed: April 2014.

[42] VMware vCloud Connector. http://www.vmware.com/products/vcloud-connector. Accessed: April 2014.

[43] WLCG. http://dashboard.cern.ch/. Accessed: April 2014.

[44] World LHC Computing Grid. http://wlcg.web.cern.ch/. Accessed: April 2014.

[45] Arshad Ali, Ashiq Anjum, Julian Bunn, Richard Cavanaugh, Frank Van Lingen, Muhammad Atif Mehmood, Harvey Newman, Conrad Steenberg, and Ian Willers. Predicting the Resource Requirements of a Job Submission. In *In Proceedings of the Conference on Computing in High Energy and Nuclear Physics*, page 273, 2004.

[46] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In *Network Operations and Management Symposium*, pages 204–212. IEEE, 2012.

[47] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004.

[48] David P. Anderson. Volunteer Computing: The Ultimate Cloud. *Crossroads*, 16:7–10, 2010.

[49] David P. Anderson, Carl Christensen, and Bruce Allen. Designing a Runtime System for Volunteer Computing. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[50] David P. Anderson and Gilles Fedak. The Computational and Storage Potential of Volunteer Computing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[51] David P. Anderson, Eric Korpela, and Rom Walton. High-Performance Task Distribution for Volunteer Computing. In *Proceedings of the First International Conference on e-Science and Grid Computing*, E-SCIENCE '05, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[52] David P. Anderson and Kevin Reed. Celebrating Diversity in Volunteer Computing. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, HICSS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[53] Sergio Andreozzi, Natascia De Bortoli, Sergio Fantinel, Antonia Ghiselli, Gian Luca Rubini, Gennaro Tortone, and Maria Cristina Vistoli. GridICE: A Monitoring Service for Grid Systems . *Future Generation Computer Systems*, 21(4):559 – 571, 2005. High-Speed Networks and Services for Data-Intensive Grids: the DataTAG Project.

[54] Artur Andrzejak, Derrick Kondo, and David P. Anderson. Exploiting Non-Dedicated Resources for Cloud Computing. In *Network Operations and Management Symposium*, pages 341–348. IEEE, 2010.

[55] Nikos Antonopoulos and Lee Gillam. *Cloud Computing: Principles, Systems and Applications*. Springer, 2010.

[56] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing.

Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, February 2009.

[57] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[58] Mark Baker and Garry Smith. GridRM: An Extensible Resource Monitoring System. In *CLUSTER*. IEEE Computer Society, 2003.

[59] Zoltn Balaton, Peter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. Comparison of Representative Grid Monitoring Tools. Technical report, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 2000.

[60] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, October 2003.

[61] Daniel J. Barrett and Richard E. Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[62] Daniel Bartholomew. QEMU: A Multihost, Multitarget Emulator. *Linux Journal*, 2006(145):3, May 2006.

[63] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[64] G. Bruce Berriman, Ewa Deelman, Gideon Juve, Moira Regelson, and Peter Plavchan. The Application of Cloud Computing to Astronomy: A Study of Cost and Performance. In *e-Science in Astronomy Conference*, December 2010.

[65] Kenneth P. Birman, Robbert van Renesse, James Kaufman, and Werner Vogels. Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation. In *Active Middleware Services'03*, pages 4–13. IEEE, 2003.

[66] Michael Black and Gregory Bard. SAT Over BOINC: An Application-Independent Volunteer Grid Project. In *Proceedings of the 2011 IEEE/ACM*

12th International Conference on Grid Computing, GRID ’11, pages 226–227, Washington, DC, USA, 2011. IEEE Computer Society.

[67] M. Bohlouli and M. Analoui. Grid-HPA: Predicting Resource Requirements of a Job in the Grid Computing Environment. *International Journal of Electrical and Computer Engineering*, 3(2):68–75, 2008.

[68] Franck Bonnassieux, Robert Harakaly, and Pascale Primet. MapCenter: An Open Grid Status Visualization Tool. In *Proceedings of ISCA 15th International Conference on Parallel and Distributed Computing Systems*, pages 2–3, 2002.

[69] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live Wide-area Migration of Virtual Machines Including Local Persistent State. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE ’07, pages 169–179, New York, NY, USA, 2007. ACM.

[70] Krishnaveni Budati, Jason Sonnek, Abhishek Chandra, and Jon Weissman. RIDGE: Combining Reliability and Performance in Open Grid Platforms. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC ’07, pages 55–64, New York, NY, USA, 2007. ACM.

[71] P. Buncic, C. Aguado Sanchez, J. Bloomer, L. Franco, S. Klemer, and P. Mato. CernVM A Virtual Software Appliance for LHC Applications. In *Proceedings of the XII. International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, 2008.

[72] Rajkumar Buyya, James Broberg, and Andrzej Goscinski. *Cloud Computing Principles and Paradigms*. Wiley, 2011.

[73] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurreny and Computation: Practice and Experience (CCPE)*, 14(13):1175–1220, 2002.

[74] Rajkumar Buyya, Manzur Murshed, and David Abramson. A Deadline and Budget Constrained Cost-Time Optimisation Algorithm for Scheduling Task Farming Applications on Global Grids. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[75] Rajkumar Buyya and Rajiv Ranjan. Federated Rresource Management in Grid and Cloud Computing Systems. *Future Generation Computer Systems*, 26(8):1189–1191, October 2010.

[76] Jamie Cameron. *Managing Linux Systems with Webmin*. Prentice Hall Professional, 2004.

[77] Louis-Claude Canon, Emmanuel Jeannot, and Jon B. Weissman. A Scheduling and Certification Algorithm for Defeating Collusion in Desktop Grids. In *ICDCS*, pages 343–352. IEEE Computer Society, 2011.

[78] Ian Foster Carl and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[79] Christophe Cerin and Gilles Fedak. *Desktop Grid Computing*. Chapman and Hall/CRC, 2012.

[80] Abhishek Chandra and Jon Weissman. Nebulas: Using Distributed Voluntary Resources to Build Clouds. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[81] Abhishek Chandra, Jon Weissman, and Benjamin Heintz. Decentralized Edge Clouds. *IEEE Internet Computing*, 17(5):70–73, September 2013.

[82] R.G Clegg, S. Clayman, G. Pavlou, L. Mamatas, and A. Galis. On the Selection of Management/Monitoring Nodes in Highly Dynamic Networks. *IEEE Transactions on Computers*, 62(6):1207 – 1220, June 2013.

[83] A. W. Cooke, A. J. G. Gray, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Cordenonsi, R. Byrom, L. Cornwall, A. Djaoui, L. Field, S. M. Fisher, S. Hicks, J. Leake, R. Middleton, A. Wilson, X. Zhu, N. Podhorszki, B. Coghlan, S. Kenny, and J. Ryan. The Relational Grid Monitoring Architecture: Mediating Information about the Grid. *Journal of Grid Computing*, 2, 2004.

[84] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual

Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[85] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Volunteer Computing and Desktop Cloud: The Cloud@Home Paradigm. In *Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications*, NCA '09, pages 134–139, Washington, DC, USA, 2009. IEEE Computer Society.

[86] Christopher Dabrowski. Reliability in Grid Computing Systems. *Concurr. Comput. : Pract. Exper.*, 21(8):927–959, June 2009.

[87] Peter Darch and Annamaria Carusi. Retaining Volunteers in Volunteer Computing Projects. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 368:41774192, 2010.

[88] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The Cost of Doing Science on the Cloud: The Montage Example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[89] Umesh Deshpande, Brandon Schlinker, Eitan Adler, and Kartik Gopalan. Gang Migration of Virtual Machines Using Cluster-wide Deduplication. In *CCGRID*, pages 394–401. IEEE Computer Society, 2013.

[90] Bartosz Dobrzelecki, Amrey Krause, Michal Piotrowski, and Neil Chue Hong. *Managing and Analysing Geomic Data using HPC and Clouds*, chapter 13, pages 261–277. Springer, January 2011.

[91] Patricio Domingues, Filipe Araujo, and Luis Silva. Evaluating the Performance and Intrusiveness of Virtual Machines for Desktop Grid Computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[92] Patrcio Domingues, Bruno Sousa, and Lus Moura Silva. Sabotage-tolerance and Trust Management in Desktop Grid Computing. *Future Generation Comp. Syst.*, 23(7):904–912, 2007.

[93] Dave Durkee. Why Cloud Computing Will Never Be Free. *Communications of the ACM*, 8(4):20–29, April 2010.

[94] Erik Elmroth and Lars Larsson. Interfaces for Placement, Migration, and Monitoring of Virtual Machines in Federated Clouds. In *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, GCC '09, pages 253–260, Washington, DC, USA, 2009. IEEE Computer Society.

[95] Erik Elmroth and Johan Tordsson. A Grid Resource Broker Supporting Advance Reservations and Benchmark-based Resource Selection. In *Lecture Notes in Computer Science*, pages 1061–1070. Springer-Verlag, 2005.

[96] Erik Elmroth and Johan Tordsson. A Standards-based Grid Resource Brokering Service Supporting Advance Reservations, Coallocation, and cross-Grid Interoperability. *Concurr. Comput. : Pract. Exper.*, 21(18):2298–2335, December 2009.

[97] EPSRC. Research Councils Workshop on Cloud Computing for Research. EPSRC, July 2010.

[98] Trilce Estrada, Michela Taufer, and David P. Anderson. Performance Prediction and Analysis of BOINC Projects: An Empirical Study with EmBOINC. *Journal of Grid Computing*, 7(4):537–554, 2009.

[99] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazons EC2. In *In The 1st Workshop on Cloud Computing and its Applications (CCA)*, 2008.

[100] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Cappello. XtremWeb: A Generic Global Computing System. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 582–587, 2001.

[101] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu. Dynamic Mobile Cloud Computing: Ad Hoc and Opportunistic Job Sharing. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 281–286, Washington, DC, USA, 2011. IEEE Computer Society.

[102] Diogo Ferreira, Filipe Araujo, and Patricio Domingues. libboincexec: A Generic Virtualization Approach for the BOINC Middleware. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1903–1908, Washington, DC, USA, 2011. IEEE Computer Society.

[103] J Fisher-Ogden. Hardware Support for Efficient Virtualization. Technical report, University of California, San Diego, 2006.

[104] Department for Business Innovation and Skills (BIS). Business Population Estimates for the UK and Regions. In *Business Population Estimates 2012*, 2012.

[105] Office for National Statistics. UK Standard Industrial Classification of Economic Activities 2007 (SIC 2007). In Lindsay Prosser, editor, *Office for National Statistics*. Palgrave Macmillan, 2007.

[106] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[107] Ian Foster. Globus Online: Accelerating and Democratizing Science through Cloud-Based Services. *IEEE Internet Computing*, 15(3):70–73, 2011.

[108] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[109] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001.

[110] Laura Gilbert, Jeff Tseng, Rhys Newman, Saeed Iqbal, Ronald Pepper, Onur Celebioglu, Jenwei Hsieh, and Mark Cobban. Performance Implications of Virtualization and Hyper-Threading on High Energy Physics Applications in a Grid Environment. In *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[111] Daniel Lombrana Gonzalez, Francisco Fernandez de Vega, L. Trujillo, G. Olague, M. Cardenas, L. Araujo, P. Castillo, K. Sharman, and A. Silva. Interpreted Applications within BOINC Infrastructure. In Fernando Silva, Gaspar

Barreira, and Ligia Ribeiro, editors, *IBERGRID 2nd Iberian Grid Infrastructure Conference Proceedings*, pages 261–272, Porto, Portugal, 12-14 May 2008.

[112] Jim Gray. Distributed Computing Economics. *Queue*, 6(3):63–68, May 2008.

[113] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[114] LHCb Computing Group. LHCb Computing Model. Technical report, CERN, 2005.

[115] M. Hakem and F. Butelle. Reliability and Scheduling on Systems Subject to Failures. In *International Conference on Parallel Processing*, page 38. IEEE, 2007.

[116] Max Hammond, Rob Hawtin, Lee Gillam, and Charles Oppenheim. Cloud Computing for Research. Technical report, Curties and Cartwright, June 2010.

[117] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case Study for Running HPC Applications in Public Clouds. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 395–401, New York, NY, USA, 2010. ACM.

[118] Yi He, Zili Shao, Bin Xiao, Qingfeng Zhuge, and Edwin Sha. Reliability Driven Task Scheduling for Heterogeneous Systems. In *The International Conference on Parallel and Distributed Computing and Systems*, pages 465–470. ACTA Press, 2003.

[119] Thomas A. Henzinger, Anmol V. Singh, Vasu Singh, Thomas Wies, and Damien Zufferey. Static Scheduling in Clouds. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, Berkeley, CA, USA, 2011. USENIX Association.

[120] Jon Hill, Matthew Hambley, Thorsten Forster, Muriel Mewissen, Terence M. Sloan, Florian Scharinger, Arthur Trew, and Peter Ghazal. SPRINT: A New Parallel Framework for R. *BMC Bioinformatics*, 9(1):558+, 2008.

[121] Z Hill and M Humphrey. A Quantitative Analysis of High Performance Computing with Amazon's EC2 Infrastructure: The Death of the Local Cluster? In

10th IEEE/ACM International Conference on Grid Computing, pages 26 – 33, 2009.

[122] Will Hopkins. A New View of Statisics. Internet Society for Sport Science, 2000.

[123] Eduardo Huedo, Rubén S. Montero, and Ignacio M. Llorente. Evaluating the Reliability of Computational Grids from the End User's Point of View. *J. Syst. Archit.*, 52(12):727–736, December 2006.

[124] Soonwook Hwang and Carl Kesselman. A Flexible Framework for Fault Tolerance in the Grid. *Journal of Grid Computing*, 1(3):251–272, 2003.

[125] Roman Iakymchuk, Jeff Napper, and Paolo Bientinesi. Improving High-performance Computations on Clouds Through Resource Underutilization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 119–126, New York, NY, USA, 2011. ACM.

[126] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011)*, 2011.

[127] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, June 2011.

[128] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. An Empirical Study into Adaptive Resource Provisioning in the Cloud. In *IEEE International Conference on Utility and Cloud Computing (UCC 2010)*, page 8, Chennai/ India, 12 2010. IEEE.

[129] C Issariyapat, P Pongpaibool, S Mongkolluksame, and K Meesublak. Using Nagios as a Groundwork for Developing a Better Network Monitoring System. In *Proceedings of Technology Management for Emerging Technologies*, 2012.

[130] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *IEEE Second Inter-*

national Conference on Cloud Computing Technology and Science (CloudCom), 2010.

[131] Kris Jamsa. *Cloud Computing*. Jones & Bartlett, 2013.

[132] Zoltan Juhasz, Peter Kacsuk, and Dieter Kranzlmuller, editors. *Distributed and Parallel Systems: Cluster and Grid Computing*. Springer, 2004.

[133] Mohammed J. Kabir. *Apache Server Bible*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[134] Peter Kacsuk. *Science Gateways for Distributed Computing Infrastructures: Development Framework and Exploitation by Scientific User Communities*. Springer Verlag, August 2014.

[135] Peter Kacsuk, Zoltan Farkas, Miklos Kozlovszky, Gabor Hermann, Akos Balasko, Krisztian Karoczkai, and Istvan Marton. WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities. *J. Grid Comput.*, 10(4):601–630, December 2012.

[136] Ali Khajeh-Hosseini, David Greenwood, and Ian Sommerville. Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 450–457, Washington, DC, USA, 2010. IEEE Computer Society.

[137] Ekasit Kijsipongse and Suriya U-ruekolan. Scaling Hadoop Clusters with Virtualized Volunteer Computing Environment. In *Proceedings of the 11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 146–151. IEEE, May 2014.

[138] Jinoh Kim, Abhishek Chandra, and Jon B. Weissman. Exploiting Heterogeneity for Collective Data Downloading in Volunteer-based Networks. In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 275–282, 2007.

[139] Jinoh Kim, Abhishek Chandra, and Jon B. Weissman. Accessibility-based Resource Selection in Loosely-coupled Distributed Systems. In *Proceedings of the 28th International Conference on Distributed Computing Systems*, pages 777–784, 2008.

[140] Jinoh Kim, Abhishek Chandra, and Jon B. Weissman. Using Data Accessibility for Resource Selection in Large-Scale Distributed Systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(6):788–801, June 2009.

[141] Graham Kirby, Alan Dearle, Angus McDonald, and Alvaro Fernandes. An Approach to Ad-Hoc Cloud Computing. University of St Andrews, Whitepaper, 2010.

[142] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D.P. Anderson. Cost-Benefit Analysis of Cloud Computing versus Desktop Grids. *IEEE International Symposium on Parallel Distributed Processing*, pages 1 –12, May 2009.

[143] Donald Kossmann, Tim Kraska, and Simon Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[144] M Kozlowszky, K Karoczkai, I Marton, A Balasko, A Marosi, and P Kacsuk. Enabling Generic Distributed Compuuting Infrastructure Compatibility for Workflow Management Systems. *Computer Science*, 13(3):61–78, 2012.

[145] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating Computation Times of Data-Intensive Applications. *IEEE Distributed Syst. Online*, 5, 2004.

[146] Hyun Jung La and Soo Dong Kim. A Self-Stabilizing Process for Mobile Cloud Computing. In *Proceedings of the 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, SOSE '13, pages 454–462, Washington, DC, USA, 2013. IEEE Computer Society.

[147] C Lac and S Ramanathan. A Resilient Telco Grid Middleware. In *11th IEEE Symposium on Computers and Communications*, pages 306–311, June 2006.

[148] G Lanfermann, G Allen, T Radke, and E Seidel. Nomadic Migration: Fault Tolerance in a Disruptive Grid Environment. In *2nd IEEE/ACM International Symposium om Cluster Computing and the Grid*, 2002.

[149] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall,

M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006.

[150] Craig A. Lee. A Perspective on Scientific Cloud Computing. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 451–459, New York, NY, USA, 2010. ACM.

[151] Ricardo Lent and Javier Barria. *Towards Reliable Mobile Ad hoc Networks*, chapter 6. InTech, 2011.

[152] Wubin Li, Johan Tordsson, and Erik Elmroth. Modeling for Dynamic Cloud Scheduling Via Migration of Virtual Machines. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 163–171, Washington, DC, USA, 2011. IEEE Computer Society.

[153] Maik Lindner, Fermn Galán, Clovis Chapman, Stuart Clayman, Daniel Henriksson, and Erik Elmroth. The Cloud Supply Chain: A Framework for Information, Monitoring, Accounting and Billing. In *2nd International ICST Conference on Cloud Computing (CloudComp 2010)*. Springer Verlag, 2010.

[154] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[155] Ashley D. Lloyd and Terence M. Sloan. Intercontinental Grids: An Infrastructure for Demand-Driven Innovation. *J. Grid Comput.*, 9:185–200, June 2011.

[156] Wei Lu, Jared Jackson, and Roger Barga. AzureBlast: A Case Study of Developing Science Applications on the Cloud. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 413–420, New York, NY, USA, 2010. ACM.

[157] Frederic Magoules. *Fundamentals of Grid Computing: Theory, Algorithms and Technologies*. Chapman & Hall/CRC, 2010.

[158] Ming Mao, Jie Li, and Marty Humphrey. Cloud Auto-scaling with Deadline and Budget Constraints. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48, October 2010.

[159] Cecchi Marco, Capannini Fabio, Dorigo Alvise, Ghiselli Antonia, Gianelle Alessio abd Giacomini Francesco, Maraschini Alessandro, Molinari Elisabetta, Monforte Salvatore, and Petronzio Luca. The gLite Workload Management System. *Journal of Physics*, 219, 2010.

[160] Attila Csaba Marosi, Péter Kacsuk, Gilles Fedak, and Oleg Lodygensky. Sandboxing for Desktop Grids Using Virtualization. In *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 559–566, Washington, DC, USA, February 2010. IEEE Computer Society.

[161] P Martin, M Atkinson, M Parsons, A Carter, and G Francis. Edim1 progress report. Technical report, EPCC, 2011.

[162] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation And Experience. *Parallel Computing*, 30:2004, 2003.

[163] Michael Mattess, Christian Vecchiola, and Rajkumar Buyya. Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 180–188, Washington, DC, USA, 2010. IEEE Computer Society.

[164] Gary McGilvary. *How to Create a BOINC Project*. The University of Edinburgh, http://garymcgilvary.co.uk/papers/How_to_create_a_boinc_project.pdf, April 2012. Accessed: May 2014.

[165] Gary McGilvary, Adam Barker, Ashley Lloyd, and Malcolm Atkinson. V-BOINC: The Virtualization of BOINC. In *CCGrid 2013*, Delft, The Netherlands, May 2013.

[166] Gary A. McGilvary, Josep Rius, Íñigo Goiri, Francesc Solsona, Adam Barker, and Malcolm Atkinson. C2MS: Dynamic Monitoring and Management of

Cloud Infrastructures. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science, CloudCom*, volume 1 of *CLOUDCOM '13*, pages 290–297, Washington, DC, USA, 2013. IEEE Computer Society.

[167] Shean T. McMahon and Isaac D. Scherson. Selection of Optimal Computing Platforms through the Suitability Measure. In *ISPDC*, pages 236–241, 2007.

[168] Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. Performance Evaluation of Amazon EC2 for NASA HPC Applications. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, pages 41–50, New York, NY, USA, 2012. ACM.

[169] Chonglei Mei, Daniel Taylor, Chenyu Wang, Abhishek Chandra, and Jon Weissman. Sharing-Aware Cloud-Based Mobile Outsourcing. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 408–415, Washington, DC, USA, 2012. IEEE Computer Society.

[170] Tatsuya Mori, Makoto Nakashima, and Tetsuro Ito. A Sophisticated Ad Hoc Cloud Computing Environment Built by the Migration of a Server to Facilitate Distributed Collaboration. In Leonard Barolli, Tomoya Enokido, Fatos Xhafa, and Makoto Takizawa, editors, *AINA Workshops*, pages 1196–1202. IEEE, 2012.

[171] Abderrahmen Mtibaa, Khaled Harras, and Afnan Fahim. Towards Computational Offloading in Mobile Device Clouds. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, 2013.

[172] James Murty. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, 2008.

[173] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 23–32, New York, NY, USA, 2007. ACM.

[174] Oded Nov, David Anderson, and Ofer Arazy. Volunteer Computing: A Model of the Factors Determining Contribution to Community-based Scientific Research.

In *Proceedings of the 19th international conference on World Wide Web*, WWW '10, pages 741–750, New York, NY, USA, 2010. ACM.

[175] Simon Ostermann, Ru Iosup, Nezih Yigitbasi, and Thomas Fahringer. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *ICST International Conference on Cloud Computing*, 2009.

[176] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2009.

[177] Savvas Petrou, Terence M. Sloan, Muriel Mewissen, Thorsten Forster, Michal Piotrowski, and Bartosz Dobrzelecki. Optimization of a Parallel Permutation Testing Function for the SPRINT R Package. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 516–521, New York, NY, USA, 2010. ACM.

[178] M Piotrowski, GA McGilvary, T Sloan, M Mewissen, AD Lloyd, T Forster, L Mitchell, P Ghazal, and J. Hill. Exploiting Parallel R in the Cloud with SPRINT. *Methods of Information in Medicine*, 52:80–90, 2013.

[179] Florentina I. Popovici and John Wilkes. Profitable Services in an Uncertain World. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 36–, Washington, DC, USA, 2005. IEEE Computer Society.

[180] Karthick Ramachandran, Hanan Lutfiyya, and Mark Perry. Decentralized Resource Availability Prediction for a Desktop Grid. In *CCGRID*, pages 643–648. IEEE, 2010.

[181] Karthick Ramachandran, Hanan Lutfiyya, and Mark Perry. Decentralized Approach to Resource Availability Prediction Using Group Availability in a P2P Desktop Grid. *Future Generation Comp. Syst.*, 28(6):854–860, 2012.

[182] Gemma Reig, Javier Alonso, and Jordi Guitart. Prediction of Job Resource Requirements for Deadline Schedulers to Manage High-Level SLAs on the Cloud. In *Proceedings of the 2010 Ninth IEEE International Symposium on Network Computing and Applications*, NCA '10, pages 162–167, Washington, DC, USA, 2010. IEEE Computer Society.

[183] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Em-

merich, and F. Galán. The Reservoir Model and Architecture for Open Federated Cloud Computing. *IBM J. Res. Dev.*, 53(4):535–545, July 2009.

[184] Arnon Rosenthal, Peter Mork, Maya Hao Li, Jean Stanford, David Koester, and Patti Reynolds. Cloud computing: A New Business Paradigm for Biomedical Information Sharing. *J. of Biomedical Informatics*, 43(2):342–353, April 2010.

[185] E Rotem, J Hermerding, A Cohen, and H. Cain. Temperature Measurement in the Intel CoreTM Duo Processor. In *12th International Workshop on Thermal investigations of ICs*, 2006.

[186] Ruay-Shiung-Chang, Jerry Gao, Volker Gruhn, Jingsha He, George Roussos, and Wei-Tek Tsai. Mobile Cloud Computing Research- Issues, Challenges and Needs. In *IEEE 7th International Symposium on Serivce-Orientated System Engineering*, pages 442–453, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[187] Genghis Ros, Pablo Fonseca, and Oscar Daz. Legion: An Extensible Lightweight Framework for Easy BOINC Task Submission, Monitoring and Result Retrieval using Web Services. In *Proceedings of the Latin American Conference on High Performance Computing*, 2011.

[188] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide Area Cluster Monitoring with Ganglia. In *CLUSTER'03*, pages 289–289, 2003.

[189] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In *Proceedings of the 2010 Second International Conference on Computer and Network Technology*, ICCNT '10, pages 222–226, Washington, DC, USA, 2010. IEEE Computer Society.

[190] Jerome H. Saltzer and M. Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.

[191] Luis Sarmenta. *Volunteer Computing*. PhD thesis, Massachusetts Institute of Technology, 2001.

[192] Luis Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18:561–572, 2002.

[193] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009.

[194] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3:460–471, September 2010.

[195] Greg Schulz. *The Green and Virtual Data Center*. CRC Auerbach Publications, 2009.

[196] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. John Wiley & Sons, 2005.

[197] Yogesh Simmhan, Emad Soroush, Catharine van Ingen, and Deb Agarwal an Lavanya Ramakrishnan. BReW: Blackbox Resource Selection for e-Science Workflows. In *5th Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2010.

[198] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. *Systems and Virtualization Management. Standards and New Technologies*, chapter Web Service Distributed Management Framework for Autonomic Server Virtualization, pages 61–71. Springer Berlin Heidelberg, 2008.

[199] J. D. Sonnek and J. B. Weissman. A Quantitative Comparison of Reputation Systems in the Grid. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 242–249, Washington, DC, USA, 2005. IEEE Computer Society.

[200] Jason Sonnek, Abhishek Chandra, and Jon Weissman. Adaptive Reputation-Based Scheduling on Unreliable Distributed Infrastructures. *IEEE Trans. Parallel Distrib. Syst.*, 18(11):1551–1564, November 2007.

[201] M.J Sottile and R.G Minnich. Supermon: A High-Speed Cluster Monitoring System. In *IEEE International Conference on Cluster Computing*, 2002.

[202] Amy Spellman, Richard Gimarc, and Mark Preston. Leveraging the Cloud for Green IT: Predicting the Energy, Cost and Performance of Cloud Computing. In *Proceedings for the CMG*. CMG, February 2010.

[203] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines. *SIGPLAN Not.*, 46(7):111–120, March 2011.

[204] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[205] B Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A Grid Monitoring Architecture. Technical report, GGF Performance Working Group, 2000.

[206] Rahul Trivedi, Abhishek Chandra, and Jon Weissman. Heterogeneity-Aware Workload Distribution in Donation-Based Grids. *Int. J. High Perform. Comput. Appl.*, 20(4):455–466, November 2006.

[207] Fred van der Molen. *Get Ready for Cloud Computing: A Comprehensive Guide to Virtualization and Cloud Computing*. Van Haren Publishing, 2010.

[208] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the Cloud to the Mobile User. In *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, MCS '12, pages 29–36, New York, NY, USA, 2012. ACM.

[209] VMWare. Understanding Full Virtualization Paravirtualization and Hardware Assist. Technical report, 2007.

[210] Eugen Volk, Jochen Buchholz, Stefan Wesner, Daniela Koudela, Matthias Schmidt, Niels Fallenbeck, Roland Schwarzkopf, Bernd Freisleben, Gtz Isenmann, Jrgen Schwitalla, Marc Lohrer, Erich Focht, and Andreas Jeutter. Towards Intelligent Management of Very Large Computing Systems. In *International Conference on Competence in High Performance Computing*, pages 191–204, 2010.

[211] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. *Login*, 33:18–23, October 2008.

[212] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th Confer-*

ence on Information Communications, INFOCOM'10, pages 1163–1171, Piscataway, NJ, USA, 2010. IEEE Press.

[213] Jonathan Stuart Ward and Adam Barker. Semantic Based Data Collection for Large Scale Cloud Systems. In *Proceedings of the 5th International Workshop on Data-Intensive Distributed Computing*, DIDC '12, pages 13–22, New York, NY, USA, 2012. ACM.

[214] Wenguo Wei, Shoubin Dong, Ling Zhang, and Zhengyou Liang. An Improved Ganglia-Like Clusters Monitoring System. In *Second International Workshop on Grid and Cooperative Computing*, 2003.

[215] Jon Weissman. Fault Tolerant Wide-Area Parallel Computing. In *IPDPS 2000 Workshop*, volume 1800, pages 1214–1225. Springer Berlin Heidelberg, 2000.

[216] Jon B. Weissman, Pradeep Sundarrajan, Abhishek Gupta, Matthew Ryden, Rohit Nair, and Abhishek Chandra. Early Experience with the Distributed Nebula Cloud. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing*, DIDC '11, pages 17–26, New York, NY, USA, 2011. ACM.

[217] Barry Wilkinson. *Grid Computing: Techniques and Applications*. Chapman & Hall/CRC Computational Science, 2009.

[218] Chris Wolf. *Virtualization: From the Desktop to the Enterprise*. Apress, 2005.

[219] Namyoon Woo, Heon Y. Yeom, and Taesoon Park. MPICH-GF: Transparent Checkpointing and Rollback-Recovery for Grid-enabled MPI Processes. *IEICE Transactions on Information and Systems*, 87:1820–1828, 2004.

[220] Trevor Wright, Scott Fuller, and Ashwin Reddi. Dynamic Cloud Management System for Monitoring and Managing Services. George Mason University, Whitepaper, July 2010.

[221] Yihua Wu, Jian Cao, and Minglu Li. Private Cloud System Based on BOINC with Support for Parallel and Distributed Simulation. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 1172–1178, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[222] Kaiqi Xiong and Harry Perros. SLA-based Resource Allocation in Cluster Computing Systems. In *Proceedings of the IEEE IPDPS*, 2008.

[223] Andrew J Younge, Robert Henschel, James Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C. Fox. Analysis of Virtualization Technologies for High Performance Computing Environments. In *The 4th International Conference on Cloud Computing (IEEE CLOUD 2011)*, Washington, DC, 07/2011 2011. IEEE.

[224] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for hpc systems. In *Proceedings of the 2006 International Conference on Frontiers of High Performance Computing and Networking*, ISPA'06, pages 474–486, Berlin, Heidelberg, 2006. Springer-Verlag.

[225] Serafeim Zanikolas and Rizos Sakellariou. A Taxonomy of Grid Monitoring Systems. *Future Gener. Comput. Syst.*, 21(1):163–188, January 2005.

[226] Qian Zhu and Gagan Agrawal. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. In *HPDC '10: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 304–307, New York, NY, USA, 2010. ACM.