A Middleware for Service Deployment in Contributory Computing Systems

Daniel Lázaro Iglesias

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy

Advisor: Dr. Joan Manuel Marquès Puig

DPCS Research Group
Internet Interdisciplinary Institute
Universitat Oberta de Catalunya

Abstract

Cloud computing has recently emerged as a powerful new computing paradigm causing many companies and organizations to move their software to large data centers owned by specialized resource providers. Large data centers are also used to host and support Internet communities, like social networks (e.g. Facebook, Twitter, Flickr, etc), which constitute cloud services offered directly to end users. At the same time, millions of individuals, as well as enterprises, own underutilized computers connected to the Internet. While some models have appeared to use these resources for executing computational tasks, like volunteer computing and desktop grids, these usage models for surplus resources do not fit the needs of cloud consumers, individual service providers or community members. In parallel to these advances, peer-to-peer (P2P) networks have become popular among Internet users and a growing field of research for developing distributed systems with desirable features like decentralization, self-organization, fault tolerance and high scalability among others. However, the P2P paradigm has been limited in practice to applications sharing disk space and network bandwidth, and sharing compute power through P2P networks has not yet become a reality.

We address the lacks of cloud computing, desktop grids and peer-to-peer networks to use non-dedicated resources for *general purpose* computing by proposing the model of contributory computing, where users contribute their resources to be used collectively. A contributory community (aggregation of contributed resources) can be used as a platform to deploy services, as an alternative to large data centers. This services make it possible to use surplus resources in a general way, instead of limiting its use to the execution batch tasks.

This thesis develops the concept of contributory computing by presenting a middleware for building contributory communities and deploying services on the contributed resources. This middleware, called CoDeS, allows resources to be put into the community and services deployed on it, and takes care of keeping the services available using the contributed resources. It is self-managed, decentralized, scalable, fault-tolerant and can deal with heterogeneous resources. Specifically, this thesis will focus on defining an architecture for the middleware and designing some of the main mechanisms that allow the creation of contributory communities. The designed mechanisms are a) a mechanism for service deployment, b) a resource discovery mechanism, and c) an availability-aware resource selection mechanism. These mechanisms are decentralized, scalable and failure tolerant. Specifically, to propose our resource selection mechanism we study resource availability in volunteer computing systems, and use a set of real system traces to develop an availability prediction method. This predictor is used in the design of the resource selection mechanism for service deployment, which can leverage availability information to provide service availability while minimizing the number of migrations. All these contributions serve to prove the technical feasibility of contributory computing. Moreover, we have implemented a prototype of the designed middleware and mechanisms, and have deployed it over PlanetLab, a real distributed testbed. This forms a basis to build real contributory systems.

Resum

El paradigma de computació en núvol ha sorgit recentment amb força, fent que moltes companyies i organitzacions moguin el seu programari a grans centres de processament de dades propietat de proveïdors de recursos especialitzats. També s'utilitzen grans centres de processament de dades per a donar suport a comunitats d'Internet, com les que constitueixen les xarxes socials (com Facebook, Twitter o Flicker, entre d'altres). Podem considerar que aquest suport és un servei de computació en núvol ofert directament a usuaris finals. Al mateix temps, milions d'individus, així com empreses, posseeixen ordinadors infrautilitzats connectats a Internet. Tot i que han sorgit alguns models que utilitzen aquests recursos per executar tasques computacionals, tal com els sistemes de computació voluntària i els anomenats *desktop grids*, aquests models d'ús per recursos sobrers no encaixen en les necessitats dels consumidors de computació en núvol, dels proveïdors de serveis individuals ni dels membres de comunitats basades en Internet. En paral·lel a aquests avenços, les xarxes d'igual a igual s'han fet populars entre els usuaris d'Internet, alhora que s'han convertit en un creixent camp de recerca per al desenvolupament de sistemes distribuïts amb característiques desitjables com ara descentralització, auto-organització, tolerància a fallades i alta escalabilitat, entre d'altres. Tot i així, el paradigma d'igual a igual ha estat limitat a la pràctica a aplicacions que comparteixen espai de disc i ample de banda, mentre que la compartició de capacitat de càlcul a través de xarxes d'igual a igual no s'ha fet encara realitat.

Aquesta tesi adreça les mancances de la computació en núvol, els *desktop grids* i les xarxes d'igual a igual envers l'ús de recursos no dedicats per a computació de propòsit general mitjançant la proposta del model de computació contributiva, en que els usuaris contribueixen els seus recursos per tal de ser usats de manera col·lectiva. Una comunitat contributiva (agregació de recursos contribuïts) es pot utilitzar com una plataforma per a desplegar serveis, sent una alternativa a l'ús de grans centres de processament de dades. Aquests serveis fan possible la utilització de recursos sobrers de manera general, en comptes de limitar el seu ús a l'execució de tasques massives.

Aquesta tesi desenvolupa el concepte de computació contributiva presentant un *middleware* per a la construcció de comunitats contributives i el desplegament de serveis en els recursos contribuïts. Aquest middleware, anomenat CoDeS, permet afegir recursos a la comunitat i desplegar serveis en ells, encarregant-se de mantenir els serveis disponibles utilitzant els recursos contribuïts. És auto-gestionat, descentralitzat, escalable, tolerant a fallades i pot tractar amb recursos heterogenis. Específicament, aquesta tesi es centrarà en definir una arquitectura per al *middleware* i en dissenyar alguns dels mecanismes principals que permetran la creació de comunitats contributives. Els mecanismes dissenyats són a) un mecanisme per al desplegament de serveis, b) un mecanisme de descobriment de recursos, i c) un mecanisme de selecció de recursos basat en la disponibilitat. Aquests mecanismes són descentralitzats, escalables i tolerants a fallades. En concret, per tal de proposar el nostre mecanisme de selecció de recursos hem estudiat la disponibilitat de recursos en sistemes de computació voluntària i hem usat traces d'un sistema real per desenvolupar un mètode de predicció de disponibilitat. Aquest predictor l'utilitzem en el disseny d'un mecanisme de selecció de recursos per al desplegament de serveis, que pot aprofitar informació sobre la disponibilitat per proveir disponibilitat dels serveis tot minimitzant el nombre de migracions. Aquestes contribucions serveixen per a demostrar la viabilitat, des d'un punt de vista tècnic, del model de computació contributiva. A més d'això, hem implementat un prototipus del *middleware* i dels mecanismes dissenyats, i l'hem desplegat sobre PlanetLab, un banc de proves real per a sistemes distribuïts. Tot plegat forma una base per a la construcció i el desplegament de sistemes contributius reals.

Acknowledgments

Primer de tot, vull agrair a en Joan Manuel Marquès haver-me donat l'oportunitat d'entrar en el món de la recerca, fet que m'ha permès arribar fins aquí. També li he d'agrair tot el suport, l'orientació i l'ajuda que m'ha ofert durant aquests anys del doctorat. Aquesta tesi no hauria estat possible sense ell.

També vull donar les gràcies a tots els companys del grup DPCS, especialment a en Xavi Vilajosana per tota la seva ajuda, des que vam començar com a becaris de doctorat fins encara ara; a l'Angel Juan, pel seu suport que ha estat vital per poder acabar aquesta tesi; i a en Josep Jorba, per haver contribuït als meus primers treballs en el món de la recerca de sistemes distribuïts. També vull donar les gràcies a l'Esteve Verdura i l'Adrià Navarro pel seu ajut en aconseguir desplegar CoDeS a PlanetLab.

Gràcies també a tota la gent de l'IN3, començant pels companys becaris i investigadors, i continuant per tot el personal de gestió i de direcció que han fet possible que em dediqués a realitzar aquesta tesi. D'entre els investigadors vull agrair especialment en Joan Melià, amb qui he compartit despatx durant la major part d'aquest temps i encara continuo, i també els altres companys de despatx incloent l'Edgar Gómez, Jonatan Castaño, Biel Company, Adolfo Estalella, Ruth Pagès, Leticia Armendáriz, així com els actuals, Ivan Serrano, Ana Titus, Josep Maria Marco i Maria Pérez-Mateo. També he de mostrar el meu agraïment al Marc Domingo, el Jordi Llosa, el Guillem Cabrera i a la resta del petit grup de tecnòlegs de l'IN3.

I am also heartily indebted to Derrick Kondo, for welcoming me at INRIA and giving me his support, making my stay at Grenoble a great experience, and also for his help and guidance in the research of availability. He has contributed greatly to this thesis. I also thank Sangho Yi and Issam Al-Azzoni for being such nice office mates, Lucas Schnorr for his invaluable help in trace visualization, Annie Simon for her administrative support, Christian Seguy for his rapid solution of technical problems and Eric Heien, Pedro Velho and Thiago Presa among others for some nice conversations.

Finalmente, gracias a mi familia, especialmente a mis padres y a mi hermana Eva, cuyo apoyo y cariño me han acompañado durante estos años del doctorado y todos los anteriores. Gracias a Eva por aguantar mis pormenorizadas explicaciones sobre sistemas distribuidos y por todo lo demás, que es mucho. Gracias a mis amigos, especialmente a Fernando, Dani, Toni, Javi, Laura y Cris, por darme gran parte de los buenos momentos que he pasado durante estos años. Y gracias a un montón de gente más que de un modo u otro me han ayudado en este doctorado, o, ya puestos, en mi vida, y que serían demasiados para mencionarlos a todos aquí.

¡Gracias a todos!

Gràcies a tots!

Thank you everyone!

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Cloud computing has recently emerged as a powerful new computing paradigm and is rapidly establishing itself as a business model. Many companies and organizations are moving their software to the cloud in order to reduce the costs of infrastructure while gaining unlimited computational capacity and storage and the possibility to access their applications from the web. The main new achievements of cloud computing [16, 17] are the appearance of infinite computing resources, the lack of an up-front investment by cloud users and the ability to pay for use of resources measured in fine grain.

However, the cost of cloud computing is not negligible. On the one hand, for cloud providers, the cost of the required large data centers is huge, when considering the sum of power consumption, air conditioning, network capacity, hardware, space and maintenance [47]. Moreover, large data centers also cause a huge environmental impact [100]. On the other hand, the cost for the cloud consumers is adequate for small and medium projects, but for large projects it may be simply too high [89]. Moreover, some services that could be deployed in a cloud may have weak performance and robustness requirements [40], and therefore obtain no benefit of the additional guarantees offered by cloud providers.

Additionally, the cloud computing paradigm is moving data and applications to centralized computing data centers spread around the world. This means that a handful of large companies control all the computation and data of a huge amount of users and enterprises [100], creating a relation of absolute dependance. Moreover, the cloud computing model has also other limitations like the absence of Service-Level Agreements, lock-in, performance instability and network limits [77], that make it also inadequate for many mission-critical applications. This has led to the rise of cloud-based infrastructures, called private clouds, where local clusters are managed with cloud techniques. This can leverage some of the advantages that virtualization provides in the form of flexibility. However, it does not offer the other achievements of cloud computing.

Large data centers are also used to host and support Internet communities, like social networks (e.g. Facebook, Twitter, Flickr, etc). These are cloud services offered directly to end users. However, these users might also have concerns about the dependance created by trusting their computation and storing their data in a privately owned data center, just as enterprises. Moreover, these services might not have a commercial purpose and/or profitability, making them inadequate for private data center hosting. Such non commercial services might be provided by users or organizations that want to share their applications as a public service [40], and both the upfront commitment of traditional resource provisioning and the constant cost of cloud computing might be more than what the providers are willing to pay.

While large data centers are built at a great cost to gather computational resources in a centralized location, millions of individuals, as well as enterprises, own underutilized computers connected to the Internet. Some models have appeared to use these resources for executing computational tasks, specifically enterprise desktop grids and volunteer computing. The former use the surplus capacity of resources inside an enterprise, partially used by its workers (i.e. each employee's PC), to make computations. The latter uses resources voluntarily contributed by individuals to be used by an entity in an specific project (e.g. SETI@Home, Folding@Home, etc). Both models use a centralized server, or a number of them, maintained by the entity that makes use of the resources, to coordinate the participant machines and store the results.

Volunteer computing, in particular, has been able to draw a lot of resources from individuals around the world. SETI@home has more than 200,000 active users [1], and Folding@home has more than 450,000 active CPUs [2], counting desktop computers, GPUs and Playstation 3.

However, these usage models for surplus resources do not fit the needs of cloud consumers, individual service providers or community members. One reason is that they still rely on centralized resources provided by a third party entity, so dependency concerns are not solved. Another reason, even more important, is that these resources are only used for executing jobs, that run for a limited time and return a result. This limitation of both paradigms, which could be jointly called the desktop grid paradigm, prevents the use of non-dedicated resources in a more general manner, being in this respect inferior to cloud computing.

In parallel to these advances, another important paradigm has become popular among Internet users around the world: peer-to-peer (P2P) networks. Popularized through applications for sharing files (mainly media like music and movies) between end users (peers), this kind of network has become a growing field of research for developing systems with desirable features like decentralization, self-organization, fault tolerance and high scalability among others.

However, the P2P paradigm has been limited in practice to applications sharing disk space and network bandwidth. Some reasons for this might be its initial appearance as a way to share data between users, or the intention to keep P2P applications as non-intrusive to computer users as possible. In any case, although some applications not related to file-sharing (like Skype's voice call and video conferencing) have appeared, sharing compute power through P2P networks has not yet become a

[1]238,556 on February 11th, 2011. http://boincstats.com/stats/project_graph.php?pr=sah

[2]460,157 on February 11th, 2011. http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats

reality.

We address the lacks of cloud computing, desktop grids and peer-to-peer networks to use non-dedicated resources for *general purpose* computing by proposing the model of contributory computing, where users contribute their resources to be used collectively. A contributory community (aggregation of contributed resources) can be used as a platform to deploy services, as an alternative to large data centers. This services make it possible to use surplus resources in a general way, instead of limiting its use to the execution batch tasks.

Contributory computing offers some of the most important characteristics of cloud computing, like the appearance of infinite resources available on demand and the elimination of up-front investment, while not implying any payment scheme. It is a suitable model both for small and medium enterprises and for user communities, which may not be interested in making an economic investment for deploying their services, or may be concerned about dependency from resource providers. In the former case, an enterprise can aggregate resources that are already owned, i.e., the employee's PCs, to deploy the enterprise's services. All data is stored inside the enterprise, and in case of an increase of resource demand, commodity resources can be easily added. In the latter case, members of a community provide their resources, forming a contributory community. No additional cost must be afforded by the community members than having their computers connected for a reasonable amount of time, and the amount of available resources automatically scales as the number of members of the community grows.

This thesis develops the concept of contributory computing by presenting a middleware for building contributory communities and deploying services on the contributed resources. This middleware, called CoDeS, allows resources to be put into the community and services deployed on it, and takes care of keeping the services available using the contributed resources. It is self-managed, decentralized, scalable, fault-tolerant and can deal with heterogeneous resources.

Specifically, this thesis will focus on defining an architecture for the middleware, and designing some of the main mechanisms that allow the creation of contributory communities. The designed mechanisms are a) a mechanism for service deployment, b) a resource discovery mechanism, and c) an availability-aware resource selection mechanism. These mechanisms are decentralized, scalable and failure tolerant. Specifically, to propose our resource selection mechanism we study resource availability in volunteer computing systems, and use a set of real system traces to develop an availability prediction method. This predictor is used in the design of the resource selection mechanism for service deployment, which can leverage availability information to provide service availability while minimizing the number of migrations. All these contributions serve to prove the technical feasibility of contributory computing. Moreover, we have implemented a prototype of the designed middleware and mechanisms, and have deployed it over PlanetLab[3] [112], a real distributed testbed. This forms a basis to build real contributory systems.

The remainder of this chapter is organized as follows. Section 1.1 presents some of the current distributed computing paradigms, their purposes and the limitations that motivate our research on a novel paradigm. Section 1.2 presents contributory computing, a model for distributed systems formed by non-dedicated resources contributed by individual users to be used collectively forming a community. Section 1.3 presents the objectives of this thesis, and Section 1.4 details the contributions of this research. Finally, Section 1.5 explains the outline of the remainder of this dissertation.

1.1 Background and Motivation

1.1.1 Grid Computing

Distributed computing has been around since almost the beginning of the history of computing, when computational problems exceeding the capacity of a single computer fostered the creation of computer networks. Soon appeared the idea of a "computational grid", parallel to the electric power grid, which would allow users to obtain computational power by just connecting to it and pay by usage.

The idea of the grid was encouraged by the fact that most computers were underutilized for most of the time [57]. This gave the intuition that enough computing power did exist to satisfy the needs of scientists and researchers, if it could be organized into a system like the conceptual "computational grid". Over time, different systems appeared to try and get the unused cycles of computers, like Condor [139].

When the concept of the grid finally materialized as a named field for research and innovation [66], it was in the form of an effort to achieve interoperability of distributed systems. What was seen as the main problem towards building a grid was the existence of a plethora of separated different distributed systems with their independent protocols. This made it impossible for a user or organization to access and use resources that belonged to another organization and were managed using different software.

The main efforts of the grid community turned towards developing a set of protocols and tools, like the *de facto* standard Globus Toolkit[4], that would allow easy interaction between computer systems in different administrative domains. This involved having open standards and protocols for managing different processes, both functional (file transfer, remote execution [15, 63], resource discovery) and administrative (access and use permissions, auditing and accounting).

[3]http://www.planet-lab.eu/

[4]`http://www.globus.org/toolkit/`

Resources provided by many organizations, in different administrative domains and with different and independent internal policies, can be federated in a Virtual Organization (VO). The grid standard protocols are used by members of the organizations to access other resources inside the VO. More details on how grids are implemented can be found in Section 2.2.1.

While grid efforts fructified in the construction of a number of scientific grids, *grid* had also become a buzzword that has lately decreased in popularity. It did not reach its supposedly ultimate goal of offering computational resources on demand as easily and simply as plugging a cable.

1.1.2 Cloud Computing

Cloud computing has appeared in recent years following the initial concept behind grid computing, that would allow users to easily access unlimited computational power. The main advantages of cloud computing [16, 17] are the appearance of infinite computing resources available for the cloud user, and the ability of paying for used resources, measured in a fine granularity, which substitutes the large investments required to have equivalent amounts of resources for exclusive use. These put it closer to the idea of a "computational power grid", similar to the electric power grid.

While grid computing had focused on creating Virtual Organizations where resources belonging to different organizations were federated and accessible, following specific policies, to members of all involved organizations, cloud computing turns to a pure provider-consumer model. A cloud provider owns a large amount of resources, distributed in large data centers, which are used by many different consumers at some price. Some use models are Infrastructure-as-a-Service (IaaS), where the provider offers direct access to virtualized resources, like Amazon's Elastic Cloud Computing (EC2), and Platform-as-a-Service (PaaS), where the provider offers a software platform to deploy the client's applications, like Google AppEngine. Some details of the different offerings and implementations are presented in Section 2.2.4.

The main advantage for clients is that it frees them of the economical burdens of purchasing and maintaining a large amount of computational resources. No up-front investment is required. Moreover, they can pay for the amount of resources that they actually use, instead of caring for provisioning for peak load.

However, there are also a few limitations in the cloud model that prevent its total and widespread adoption, especially for businesses [77]. While these limitations are strictly technical, some concerns that are derived from them could be considered more philosophical issues.

A simple summary of the limits of the cloud could be that the model, pursued first by grid computing and now by cloud computing, of simply gaining access to unlimited compute power is excessively simplistic. Computation and especially data are not like electricity. There are many additional complexities that need to be considered. We will discuss some of them in the next paragraphs.

Interoperability is a big issue here, because a client must adapt to the protocols and data formats used by its provider, and can find itself in a lock-in position where the cost of moving to a different provider is too high. While efforts in grid computing pushed towards the creation of open standards to solve this problem, the situation of competence between providers in the cloud provider-consumer model, in contrast to the cooperation that motivated the grid, has caused that different providers develop different products, for example in the data storage field.

Actually data storage is an unsolved problem for cloud computing, because there is no current solution for scalable storage which provides similar flexibility than other systems like relational databases. However, this is not the only problem for applications that must make intense use of large volumes of data. Because the computing is performed far from where the data has been originated, this data must be transmitted to the cloud, with its corresponding costs, both temporal and economical.

Moreover, storing data in the cloud also brings security concerns. While for some small companies it may be better to rely on the cloud provider's security than on their own, it is probably not so for enterprises which depend on the privacy of their data. While the cloud provider can offer good security against outside threats, data privacy may be violated from the inside (by the provider itself or by some of its employees). This is not the only aspect where cloud consumers may find themselves lacking control of their systems. The lack of well-defined and meaningful service-level agreements (SLA) and performance instability may further make cloud providers look unreliable to potential consumers.

In the end, there are still limitation in what consumers can use the cloud to do (for instance, intensive analysis of large volumes of data or guaranteed service availability) and in what consumers would want the cloud to do (for example, storing all their sensitive and private data or effectively controlling their systems by locking them in to their vendor-specific protocols).

1.1.3 Desktop Grids and Volunteer Computing

Many studies have confirmed the fact that most computers spend a large deal of time unused or underutilized, with data hinting that idle time may be at least 60% in most computers[5] and in some cases as high as 97% [57]. This led to the creation of systems that are able to manage the use of these surplus resources. This kind of systems have been called by different names, from CPU scavenging [139] to others like desktop grids [45, 83] or ad-hoc grids [1], influenced by grid's high popularity in the time

[5]http://www.hpcwire.com/industry/academia/European-Grid-Infrastructure-Project-Launched-102966344.html - Retrieved on April 4, 2011.

when they were coined.

The first incarnation of desktop grids were designed to manage resources inside the network of an organization. Systems like Condor [139] or Entropia [45] use a central server to assign jobs submitted by users to resources that are currently unused, and return the results to the submitter. These jobs are usually batch tasks that can be executed independently, although there are some extensions [49] to support tasks with dependencies and interactions.

A second incarnation consists of what has been called *volunteer computing*. In these systems, popularized by SETI@home's search for extraterrestrial intelligence[6], a set of individual computer owners contribute their resources to an organization for executing computational tasks for a specific project. The volunteers contribute their resources to the project because of an interest in its success. Some examples are research projects of public interest because they are expected to produce important medical results (e.g. Folding@home[7], which studies protein folding and associated diseases like Alzheimer's, Parkinson's and many Cancers, among others, and World Community Grid[8], with many medical projects) or other scientific advances (e.g. climateprediction.net[9], which tests climate models, Einstein@home[10], which analyzes data from the LIGO gravitational wave detector, and LHC@home[11], which simulates parts of CERN's Large Hadron Collider).

The main difference between enterprise desktop grids and volunteer computing is philosophical: while the former can fit in the definition of grid as a way to share resources inside and between organizations, the latter is based on the fact that individuals contribute their own resources to a project.

Technologically, both kinds of systems are similar, with a central server that manages the use of the resources and assigns tasks to the available computers. This is also adequate for volunteer computing because there is the figure of the project organizers, who can provide a server to manage the resources that are voluntarily given to them. However, in volunteer computing hosts are distributed over the Internet, and the amount of active computers can be very high (in the order of hundreds of thousands, in popular projects like SETI@home[12], and Folding@home[13]. These facts impose some limitations on the systems, and currently systems like BOINC[14] [5], a middleware used by many volunteer computing

[6]http://setiathome.berkeley.edu/
[7]http://fah-web.stanford.edu/
[8]http://www.worldcommunitygrid.org/
[9]http://climateprediction.net/
[10]http://einstein.phys.uwm.edu/
[11]http://lhcathome.cern.ch/
[12]238,556 on February 11th, 2011. http://boincstats.com/stats/project_graph.php?pr=sah
[13]460,157 on February 11th, 2011. http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats
[14]`http://boinc.berkeley.edu/`

projects, only support independent compute-intensive tasks. No support for sets of interrelated tasks with data dependencies or data-intensive tasks is provided. More information on the implementation of desktop grids and volunteer computing systems is provided in Section 2.2.2

1.1.4 Peer-to-Peer Systems

Peer-to-peer (P2P) networks are a kind of organizational structure where the different components (in the case of computer networks, computers) are connected to each other as *peers*. This is opposed to the common client-server model, where one computer (the server) provides a service (access to data or a specific computational functionality), and the client contacts it to use the service, and therefore the connection between both is clearly defined and unidirectional. In a peer-to-peer system, any node in the system can establish a peer-like connection with any other. Through this bidirectional connection, both nodes can engage in an interaction acting as clients, as servers or as peers, depending on the functionality of the system (e.g. both nodes may share information, or one node may request information from the other).

While many aspects and applications of the earlier Internet can be seen as having a peer-to-peer structure, the Internet, with the growing importance of the World Wide Web and the prevalence of the client-server model, became more centralized over time. It was in 1999 when the appearance of Napster started a movement of P2P file-sharing systems.

P2P file-sharing became widely popular among the public, despite legal controversies, and a first generation of centralized systems (like Napster itself, were a central server stored an index of the shared files, and file transmission was direct between users) was followed by completely decentralized systems, like Gnutella [122], and hybrid approaches like FastTrack [91], BitTorrent [113] and others.

In the research community, the arrival of peer-to-peer systems led to the creation of structured overlay networks, also referred as distributed hash tables (DHT), like Chord [135], Pastry [123], CAN [120] or Tapestry [146], followed by many others [81, 97, 98, 102, 131]. The underlying connectivity paradigm of these systems, called key-based routing (KBR), was put to use in research works for a variety of applications, from file systems [32, 53, 107] to more complex communication primitives like multicast and publish/subsribe [37]. More technical information on these systems is provided in Section 2.2.3.

Structured overlay networks have been used out of the research community, in file-sharing systems (Kademlia [102] has been incorporated into eMule, BitTorrent and others), in telephony (e.g. Skype) and others. There have also been some research works to integrate P2P technologies into grids and desktop grids [2, 14, 31, 64, 83, 129, 142, 143]. However, no general framework for computation in peer-to-peer environments has appeared.

1.2 Contributory Computing

1.2.1 General Definition

This thesis presents the concept of *contributory computing* as a novel paradigm for aggregating computational resources, belonging to one or more owners (whether individuals or organizations), to be used collectively. A *contributory community* is an aggregation of contributed resources which can be used as a platform to deploy computational services (from now on called simply *services*). We also use the term to refer to the aggregation of resources as well as their owners, and the motivations and objectives that keep them together. We refer to the computers that are contributed to the community as hosts, nodes or resources. Contributory computing is intended to overcome some of the limitations of the systems and paradigms presented in Section 1.1, and allow the creation of new applications and use scenarios that are not possible with existing models.

We take a very simple definition of service. Any application that can be executed on a computer or a set of computers, receive messages (queries) from other computers and send messages in response to those queries, is a service that can be potentially deployed on a contributory community. We consider that this is a powerful abstraction to build many different applications, as demonstrated by the appearance of service-oriented models and architectures in recent years [24, 75]. Therefore, the higher-level practical objectives of the community should be implemented in the form of services. In consequence, offering services is, at a computational level, the main purpose of the community. Simple services could be a web server or a file converter. We present a technical analysis of the classes of services that will be supported in Section 2.3.3.

The community members (owners or local users of the resources that form the community) contribute their surplus resources, like in volunteer computing systems. Different models of surplus resource usage and quantification are allowed. For instance, a contributor can decide to let the community use her computer whenever she is not using it, and stop any assigned task when mouse or keyboard is detected; or she can decide to contribute a fixed portion of her resources (e.g. a given percentage of CPU, RAM, disk space and bandwidth) continuously while the computer is switched on. The members are free to decide the policies that will guide their contribution, as well as to change them at any time or to withdraw their resources from the community.

Just as the model of resource contribution is the same than in volunteer computing, the need for a motivation for people to contribute is also the same. Therefore, communities are expected to be formed by individuals or organizations who share a common goal that can be achieved or supported through offering a set of services. Community members voluntarily contribute their resources to help their community achieve its objectives. We do not consider incentive mechanisms that determine the amount of resources that a member can use in function of the amount of resources that he or she contributes [11], since members are not considered to use the resources of the community individually for their own personal benefit. However, a credit system similar to BOINC's [5] to measure and incentive contribution in a competitive way, for instance in the form of leaderboards, could be of use for a contributory community.

A final consideration about contributory computing systems is that they do not require dedicated resources. A contributory community is self-sufficient, i.e., it only depends the resources contributed by its members. Since all resources are (potentially) non-dedicated and dynamic, the system must be able to tolerate sudden connections and disconnections of nodes. Moreover, it cannot rely on any centralized component, since all resources are susceptible of being withdrawn by their owners at any moment, and must therefore work in a completely decentralized way. However, because a community only uses contributed resources, it cannot exist if the members do not contribute enough resources for its intended objectives, i.e., the amount and type of services that have to be deployed.

1.2.2 Example Scenarios

This section will present a few scenarios where a contributory community could be built by aggregating resources. These are instances of the general concept described in Section 1.2.1, and present an overview of how the contributory computing model could be materialized in various situations. More technical discussions on how to solve the problems that arise in each scenario can be found in Chapter 2.

We classify them according to two aspects: the membership policy, and the software that can be deployed. We consider that both can be either open or closed. In the case of membership, a community can be open if any individual can become a member of the community. It is closed if, on the contrary, the set of members is strictly defined and there are restrictions to entering the community. Of course, some scenarios could be in the middle ground, having some membership control policies but not strictly restrictive to a fixed set of members.

The difference in software refers to whether a community allows any service to be deployed on the community, or if only some specific software from a predefined set, defined in strict policies that are known beforehand upon entrance in the community or decided by some administrator, may be deployed. We consider the former an *open software* community, while the latter would be a *closed software* community. Note that, despite similarity in name, this distinction is not related to open source software, nor to the distinction between free or proprietary software.

Scenario 1. Enterprise: closed software, closed community

An organization could aggregate the unused capacity of their commodity computational resources (e.g., employees' desktop PCs) and use them to deploy services that support the organization's business processes. This could be considered as an extension of enterprise desktop grids, changing task execution for service deployment. This allows more applications to be transferred to the non-dedicated resources, and alleviate the investment in dedicated servers and resources.

In this case, all resources belong to the same owner: the organization. However, they may be under the direct, even if only partial, control of a local user. This user is free to withdraw resources from the community, whether intentionally or implicitly as a consequence of her work and her local use of the host. Therefore, the assumptions about resource dynamism in a contributory community still hold in an enterprise environment.

Some practical advantages stem from the existence of a common owner and administration policy for all the resources in the community. Firstly, the organization can have a greater control on aspects that may cause conflicts and problems in an open community, like authentication of software deployed on the community, user behavior, access control, etc. The organization can also provide a set of dedicated or semi-dedicated resources to support the community even in moments of low contribution, or to implement specific components of the system that may be complex to implement otherwise. However, it will still be able to utilize the unused resources in the community to execute its services and therefore reduce the necessary investment in hardware.

Scenario 2. Social network: closed software, open community

Social networks are a popular phenomenon bringing together millions of users[15]. They provide access to user-generated content as well as allow inter-user communication, together with a variety of additional services. In current implementations, application providers host their services in large data centers to serve their huge community. These data centers store user-generated content and user-related data.

An alternative implementation could involve the creation of a contributory community among the members of a social network. Each user would contribute her resources to execute the services that conform the functionality of the social network. These services would help members keep their information available when they are offline. Moreover, the total amount of resources in the community would be proportional to the number of users of the social network, and the amount of available resources at any instant would be proportional to the amount of active users. However, maybe some membership

[15]More than 500 million active users in Facebook, according to their public statistics on February 11th 2011. http://www.facebook.com/press/info.php?statistics

policies requiring members to contribute a minimum amount or resources would be necessary to ensure that the community has enough resources.

While any individual could join the community, only some specific trusted software (the one that implements the social network functionality which all members agree to use) is deployed. Security issues arise, however, regarding the privacy of the information stored in the community. End-to-end encryption mechanisms, where all information is stored encrypted and is only decrypted in the host of the final consumer, could be used to ensure that only trusted users can access one member's information. Other measures may be needed to ensure that the copy of the service that a client contacts is legitimate, and not a modified version created by a malicious user.

Scenario 3. Trusted group: open software, closed community

Another possible community is one where membership is tightly controlled, but members can freely deploy services. Some examples may be an association or a non-profit organization which needs to deploy some services to support their activities, a group of friends who want to put their resources in common, or a software development community where members can freely deploy the applications they develop.

This kind of community needs mechanisms to tightly control the access to the community. The membership may be decided externally through a different channel of communication that allows direct contact among the potential members. This is adequate for this kind of community because members are expected to know and trust each other. The system needs to enforce the membership rights and block access to non-members.

All members are free to deploy their software thanks to the trust relationship among them. However, even in a community with tightly controlled access, it is possible that someone makes malicious use of the community, for instance, by deploying services that spy on other members or that make excessive use of resources. In case that some actions need to be taken, there are other communication channels among the members of the community, and the membership enforcement mechanisms could be used to expel any member who is no longer trusted.

Scenario 4. Open software, open community

The most open contributory community would be one where any individual could join and deploy different services. However, there are a number of issues that could make it difficult for such a community to exist without additional administration and security mechanisms. First of all, if the community is completely open, there is no way to ensure that the amount of resources will be enough for offering all the services deployed in the community, since members could join but contribute only a tiny amount of resources. If there is no way to expel a member from the community and prevent him from joining again, the community may not be able to work because of members that deploy services which consume too many resources, even without considering other security risks. Therefore, some kind of membership control would be advisable.

The presence of an index to find services deployed in the community would be essential in this kind of community, even though it may be useful in all communities. However, while locating services can be solved in other ways in a closed community (e.g., having external communication channels among members to publish service information) and with closed software (e.g., having a static well-known list of services available), in the *open software, open community* model an index that can be modified dynamically is required. Members can deploy new services at any time, and they will need mechanisms to find the existing services that match their interests among all the ones that are present in the community.

Notwithstanding the presence of some level of membership control, the fact that the community may be formed by individuals who do not share a bond of trust raises some security concerns. A member can deploy services that are malicious in different ways. While a service that is malicious to its users could be simply not accessed once its risks are known, the danger of a service which is malicious towards its host may be more complex to avoid if the system deploys the services autonomously. Another risk is that the owner of a computer that has been tasked with executing a specific service may execute instead a modified version of it, which has a malicious behavior like, for instance, returning incorrect results or obtaining private data from the client.

1.2.3 Features

Contributory computing has a series of features that define it as a novel and promising new paradigm for computation. While some of them are actually shared with some previous paradigms discussed in Section 1.1, the specific combination of features makes contributory computing different. Moreover, it has the potential to overcome some of the current limitations of previous paradigms.

- *Use of underutilized resources.* A contributory community makes use of the resources of its members that were previously unused or underutilized. The efficiency of the system is improved by not letting these resources be wasted

- *No need for investment in dedicated resources.* The fact that a community works only with the resources contributed by its members, and requires no additional dedicated resources, makes it unnecessary to make any investment in resources at the community level, since members already possess resources. The aggregation of these resources provides enough computing power for the activities of the community.

- *Amount of resources adaptable to load.* The load of a community will be in most cases directly related to the amount of members that form it, since it must deploy services that serve these members. Therefore, since members also contribute resources, there must be an equilibrium between the amount of resources that a member contributes, in average, and the load that is generated per member. In case there is some fixed load, for instance caused by services that are offered by the community to the public, the community can compensate this by attracting more members or by encouraging existing members to provide more resources.

- *Service-oriented computation model.* This model for building applications offers more flexibility than the execution of parallel tasks. Almost any application can be modeled as a service [24, 75], and services can interact with each other and even create composite services.

- *Collective use of resources.* Users contribute their resources voluntarily for the community, and these resources are used collectively to deploy the services of the community. This model is therefore substantially different from models where users can obtain access to resources in function of their contribution or through payment. While it may not be adequate in some scenarios, like interaction between businesses wanting to trade resources, it gives members the freedom to collaborate by deploying services for the use of others and by using services deployed by others. An accounting scheme for such an scenario would need payment schemes for many interactions (deploying a service, hosting a service, using a service) which would require some level of user interaction and therefore hinder self-organization and ease of use, besides of imposing selfish motivational mechanisms in the community.

1.2.4 Requirements and Challenges

The main defining characteristic of a contributory community is that it is only formed by the resources contributed by its members. This imposes a series of requirements on the implementation of a contributory computing system. We deal with how to overcome the challenges presented by these requirements in Chapter 2.

- *Self-organization and self-management*: no manual configuration of the community as a whole must be required. The function of each node must be decided autonomously by the contributory software. Participating nodes must only need the usual individual management, and some simple initial configuration of the contributory software. Nodes must be able to join and leave the community without requiring any management at the community level.

- *Decentralization*: no node can be central to the system. All tasks must be susceptible of being performed by different nodes, which can be dynamically and autonomously selected by the system. Therefore, the community must not depend on any particular resource, and there must be no single point of failure in the community.

- *Failure tolerance*: a community that can work with only the resources that are contributed by its members must be prepared to deal with non-dedicated resources. These are only partially devoted to the community, and therefore may have intermittent connectivity and dedication, being occupied in other tasks or simply turned off for a significant amount of time. Moreover, these interruptions in the availability of resources may not only occur at short notice leaving little time for reaction (for example, a member wanting to use his computer for individual work immediately), but they may also happen without prior notice at all (for example, caused by a power cut). The community must therefore be able to deal with sudden disconnections of hosts.

- *Scalability*: a community can be formed by tens of users, or grow to hundreds of thousands of users. This cannot be known in advance except for the most closed communities. Therefore, the mechanisms involved in the management of the community must be able to scale up to hundreds of thousands of nodes and offer a good quality of service for its members.

- *Service availability*: since the main function of the community is to offer a set of services to its members, these services must be always available. A user must be able to access a service in a reasonable time, in any situation as long as the amount of resources in the community is enough to offer all its services. Because of the intermittent availability of resources, the mechanisms may either use replication and reactive repair actions to compensate for apparently unpredictable host behavior, or try to predict host availability. Prediction may be used for guaranteeing service availability while making an efficient use of resources, or, in other words, obtaining the maximum possible availability with the minimum resources.

- *Ability to deal with Internet-distributed resources*: a community may be formed by members scattered around the world who have connections to the Internet of different qualities. The system must be able to deal with long latencies and reduced bandwidths.

- *Ability to dealing with heterogeneous resources*: the resources contributed by members may be different in many aspects, from their raw capacity (CPU, memory, disk space) to their behavior (hosts with different levels of availability). The system must be able to select the fitting resources for each task among all those that form the community.

- *Security and privacy*: because of the decentralized nature of contributory communities, all data may be stored in resources belonging to any particular member, probably one who is not the original and legitimate owner of the data. Similarly, user activity and interactions between members and services or among members may be monitored by a third party. This may cause security concerns, some of which have been presented in Section 1.2.2. Therefore, a contributory system should incorporate mechanisms to guarantee the security and privacy of its users.

1.2.5 Related Proposals

Some other authors have proposed similar ideas with the goal of merging technological advances on different fields that have appeared thanks to paradigms like cloud computing, grid computing and peer-to-peer networks. The proposed models try to inherit the advantages and features of previous ones while overcoming their limitations.

Nebulas

Chandra and Weissman [40] define the Nebula concept as an aggregation of voluntary resources, donated by end-user hosts, to handle cloud services. It would constitute a more dispersed, less managed cloud. They present some use scenarios for this model, as a platform to test experimental cloud services, as a way to deploy dispersed-data-intensive applications close to the place where the data is stored, and as a way to deploy services to be shared with others as a "public service". The latter is close to the scenario that we envision with contributory computing.

Their work defines three requirements for the platform. The first one is defined as providing service-centric performance differentiation. They propose to address this challenge by using large-scale resource discovery techniques to select resources that can provide the needed performance for each service, and subdividing tasks to adapt to the capacity of the resources.

The second requirement is dealing with the coupling of data and computation, which they propose to address by using P2P data location techniques and mechanisms to estimate the network distance between a potential node to execute the service and the location of the data.

The last requirement they mention is the need to provide robustness to small-scale failures. They propose to address this with replication techniques guided by information about individual node reliability, together with aggressive checkpointing techniques. However no real platform is presented in order to validate their statements.

Community Cloud Computing

Marinos and Briscoe [100] introduced the Community Cloud Concept to address some of the perceived limitations of cloud computing. They identify three specific limitations. The first one is the failure of what they call *monocultures*, i.e., the failure of a cloud provider implies the failure of all the services hosted by them, which might visibly impact the performance of the Internet. The second limitation is the lack of control for the consumer, since it is transferred to the provider instead. The third limitation mentioned is the environmental impact of the tremendous power consumption of the large data centers necessary needed for cloud computing.

Their community-centered approach is similar to our definition of contributory computing. However, they focus on building the model for a community that can replace proprietary clouds on a global level. Their proposal tries to address social, economic and environmental aspects of a new paradigm. Because of this global scale, some kind of currency is required as a method to control the use and contribution of resources. Contributory computing focuses instead on communities of a limited, although potentially large, size, where collective use of the resources is possible and no currency is needed.

They define a general architecture for building a Community Cloud, and discuss the requirements of several aspects of building such a system. Moreover, they briefly discuss how existing systems like Wikipedia and YouTube could be implemented on a Community Cloud. However, they present no actual implementation of their model, although they express their intention of working on different mechanisms that can support the Community Cloud, as well as evaluate its environmental impact compared to vendor Clouds.

Cloud@Home

Cunsolo et al. [51] propose to merge the cloud and volunteer computing paradigms into Cloud@Home, a way to generalize volunteer computing environment using cloud techniques and principles. They envision the use of resources contributed by users to form a cloud, that can interoperate with current commercial clouds and hybrid clouds which can sell their resources or give them for free. However, they propose a centralized architecture inside each cloud, contrary to our decentralized approach that only depends on contributed resources. No implementation of their architecture has been presented yet.

They envision two main application scenarios for Cloud@Home. The first one is based on scientific research centers, where the model of current volunteer computing projects could be extended to make a better use of resources contributed by the public. The second one is inside enterprises, where they propose to use the existing resources inside the organization to create a cloud.

Their second use case is similar to our enterprise scenario for contributory computing. However, they focus on the interoperability between their Cloud@Home and commercial clouds to create a relationship between the enterprise and a cloud provider, so that the organization can buy resources when the ones at its internal cloud are not enough, or sell them when they have a surplus. They also mention the need to define Quality-of-Service (QoS) policies and Service Level Agreements (SLA) to mediate the interaction between clouds. We consider instead that resources can be easily added into a contributory community, and therefore an enterprise could integrate resources from a cloud provider into its community when needed and release them when no longer required. While additional mechanisms could be designed to allow interaction between different communities, we do not focus on this aspect of our model.

1.3 Objectives and Approach

The goal of this thesis is to prove the feasibility of the novel contributory computing model and to build the foundation of its implementation, paving the way for its final development and deployment on real communities. We intend to achieve this objective by designing and implementing the core software and mechanisms that are needed for contributory computing systems, and prove that they provide the features, satisfy the requirements and overcome the main challenges of the contributory model.

Our approach is to develop a middleware that can be installed by every member of the community and that can autonomously aggregate and manage the contributed resources. This middleware should allow members to control all the aspects of their contribution, including how many resources they contribute and when to do it. It should also offer a service development platform, with defined interfaces which allow services to be deployed transparently using the non-dedicated resources of the community without having to deal with all the complexities of the environment. Community members would only need to install the middleware and join a community in order to support all the services deployed on the community, instead of having to install specific software for every service that the client wants to support.

It would be also possible, although challenging, to build a service that can be deployed on a number of non-dedicated computers and make them behave like a contributory community, using the resources contributed by each member. However, a community may desire to deploy any number of services. It would be unnecessarily complex and inefficient to incorporate mechanisms to deal with all the requirements in every service designed to be deployed in a contributory community. Therefore, we consider that the creation of a contributory middleware is the right approach for the construction of contributory communities.

The specific objective of this thesis is to create a middleware to support contributory communities. This implies, first, designing an architecture that can satisfy all the requirements of the contributory computing model, and second, designing the basic mechanisms that can provide the necessary functionality of a contributory community. We focus on the service deployment mechanism, and identify three main parts that compose it: service management, resource discovery and resource selection.

Service management is the decentralized mechanism that dictates how resources are used to keep services available and how clients can find services. Because of the requirements of contributory communities, this mechanism must be decentralized and self-managed. While there have been some proposals of mechanisms for decentralized task deployment in P2P environments, to the best of our knowledge no proposal of decentralized service deployment mechanisms exists that satisfies the requirements of contributory computing. we call it service deployment in this thesis

In order to deploy services, at some point of the service deployment mechanism there is a necessity of finding resources that satisfy a specific set of requirements which qualifies them to execute the service. This need can be fulfilled by a separate mechanism of resource discovery. Since this is a critical part of service deployment, it is necessary to integrate a resource discovery mechanism into CoDeS which could satisfy all the requirements. Our approach is to first define the specific requirements for resource discovery mechanisms, and then analyze and compare existing proposals to evaluate their adequacy to contributory computing. Afterward we propose a resource discovery mechanism which leverages the features of the middleware while satisfying all the requirements of contributory communities.

The resource selection step is where we choose which of the found resources will be used for executing the service. An important factor that will determine the feasibility of contributory computing is user behavior. No only do community members need to contribute enough resources to deploy the services they want, but they also need to provide the system with some stability that allows efficient use of these resources. For this reason, we consider the use of host availability prediction in the resource selection step for the service deployment mechanism. Since no contributory community, as we define it, exists currently, we will base our user behavior analysis and evaluation on real traces from a volunteer computing system, which we consider to be similar in user behavior to what is expected from contributory communities.

1.4 Contributions

The main contributions of this thesis can be summarized in the following categories.

Design of a middleware for contributory computing systems

We have defined the system model of contributory communities and have designed a middleware to support them. This middleware, called CoDeS, has a modular and extensible architecture, which allow it to use a variety of mechanisms for different parts of its functionality. For instance, it can use different implementations of peer-to-peer connectivity layers to allow communication between nodes, or different distributed storage systems to persistently store files in the community, without requiring changes in the other parts of the middleware. This makes CoDeS able of leveraging technological advances in specific areas to improve the performance of some of its components.

In order to validate the design, we have implemented CoDeS using a mix of the novel mechanisms presented in this thesis, existing implementations of mechanisms like overlay networks, and simple implementations of the parts that are out of the scope of this thesis to serve as a proof of concept. This prototype can work both in simulation mode and in real networks. We have used the former to evaluate by simulation some of the mechanisms presented in this thesis, while latter has allowed us to have our middleware deployed on PlanetLab.

A mechanism for decentralized autonomous service deployment

We have designed a mechanism for service deployment in contributory communities, which is one of the core components of CoDeS. This mechanism is completely decentralized and self-managed, and can work depending only on contributed resources. It uses a combination of peer-to-peer mechanisms like key-based routing (KBR) and symmetric replication with optimistic algorithms, while maintaining a high degree of modularity and extensibility. For instance, it only uses generic KBR operations, and is independent of the formats used to describe resources or the mechanisms used to discover them.

We have implemented the proposed mechanism into the CoDeS prototype and validated it by simulation. We have proved that services are kept available under high levels of churn and accessed efficiently from a user perspective. Moreover, we have proved that the mechanism is scalable in *a)* community size, with a logarithmic growth in response time and no increase in communication overhead per node, and in *b)* number of services deployed, with no increase in response time and a linear increase in communication overhead.

Analysis and comparison of the architecture and behavior of existing resource discovery mechanisms

Since CoDeS requires a resource discovery mechanism as a fundamental step of the service deployment process, we have studied in depth the problem of distributed resource discovery. With this purpose, we have made a general characterization of different design decisions that define a resource discovery mechanism and how they relate to the system's requirements. We have then reviewed and classified twenty-six existing resource discovery mechanisms, and have compared them according to the requirements they fulfill and the approach taken in each of the presented design decisions. This has been accompanied by a comparison of the performance reported for some of the systems by their respective authors. These comparisons are used to find out which design approaches are suitable for fulfilling each of the requirements.

A mechanism for decentralized resource discovery in contributory communities

We have analyzed the requirements of resource discovery for contributory communities with the same criteria used for classifying existing resource discovery mechanisms. After careful study of these requirements and how they are fulfilled by some of the reviewed systems, we have designed a novel resource discovery mechanism for contributory communities. This mechanism is built on top of the service discovery mechanism, and offers a high flexibility, allowing multi-attribute range queries among others. It is independent of the resource specification format used.

We have implemented the proposed resource discovery mechanism into the prototype of the middleware, and evaluated it via simulation. We have proved its scalability and failure tolerance by simulating different community sizes and different levels of churn, and have concluded that it offers a performance similar to that of a centralized resource discovery system where resource descriptions are stored in a single server. Moreover, the overhead of our proposal is only slightly higher than that of the centralized approach. Queries are solved with a constant number of messages in average, in contrast to most of the reviewed resource discovery systems which have a logarithmic cost with respect to community size.

An availability-aware resource selection method for service deployment

We have studied how to improve the performance of service deployment by using availability prediction in the resource selection process. Particularly, although random selection with replacement proved to keep a service available under high churn, it does so at the cost of a constant rate of migrations, which are costly in terms of time and, even more important, of bandwidth used for transferring files. Since bandwidth in home environments is limited, this could be an important bottleneck for a community, depending on the amount of data required for each service and the migration rate. Therefore, we decided to leverage availability prediction to minimize the number of migrations while providing good service availability. Our approach consisted on predicting long-term host behavior, which would allow us to select a stable set of hosts for deploying the service which can collectively provide service availability.

We used traces from a real volunteer computing project, SETI@home, to analyze user behavior in environments where users contribute their own resources, as would be the case of contributory communities. Specifically, we have used these traces to study long-term availability patterns, identifying mainly three classes of hosts: those who are available for a large fraction of time, those who are only sporadically available for short periods of time, and those who follow periodical patterns. To automatically detect host availability patterns and predict future availability, we have proposed a novel prediction tool, called *bit vector*, which summarizes host availability over time and predicts future availability at different moments of the week. We have assessed its quality as a predictor using the SETI@home traces.

Finally, we have designed a resource selection mechanism that uses the bit vector to predict host availability and select hosts which can collectively provide a good service availability. This method needs no complete knowledge of the community. On the contrary, it is compatible with the other mechanisms of CoDeS and only needs information about a random set of hosts, which are then used or discarded depending on the availability they contribute to the service. We have validated this mechanism with the SETI@home traces and proved that it is able to find a stable set of hosts that can keep the service available for weeks, requiring a much lower number of migrations than random selection with replacement and short-term prediction-based resource selection mechanisms.

1.5 Outline

This dissertation is structured in 6 chapters, which approximately fit the main contributions of this thesis.

Chapter 2 is focused on detailing the contributory computing system model and presenting the architecture of CoDeS, our middleware for contributory communities. This chapter first presents a technical overview of the computing paradigms introduced in 1.1, and then proceeds to explain a variety of aspects of the contributory computing system model. It then presents the architecture of CoDeS, detailing the functionality of each of its components, and finally presents the prototype we have implemented, detailing the implementation of aspects beside the main service deployment mechanisms, which are presented in the other chapters of this dissertation, and discussing alternative approaches for some of its components.

Chapter 3 presents our service deployment mechanism. It provides a detailed explanation of the behavior of our mechanism, which we have implemented into CoDeS, and then presents an evaluation of its performance via simulation. It shows that our mechanism can provide service availability from a user perspective, measured in client access time, in large communities and under heavy churn.

Chapter 4 is focused on the topic of resource discovery. It first presents a description of requirements for resource discovery mechanisms and how they apply to CoDeS, and also a general description of design characteristics that appear in them. It then proceeds to provide an extensive evaluation of existing resource discovery mechanisms, classifying them according to their features and design characteristics and comparing their performance. This chapter then details the resource discovery mechanism that we have designed, which leverages CoDeS' service deployment functionality to offer decentralized, scalable and fault-tolerant resource discovery. We have implemented this mechanism into our middleware prototype and tested it by simulation, seeing that its average cost per query, in latency and in number of messages, is constant with respect to community size, and that it also tolerates high levels of churn with a low communication overhead.

Chapter 5 is focused on availability prediction and its use in resource selection. In this chapter we first study a large set of traces from SETI@home, identifying the presence of different classes of hosts according to their long term behavior, and proving the presence of hosts with periodical availability patterns. Then we present a prediction tool which summarizes host availability over time, and evaluate its performance using the same set of real traces. This prediction tool is then applied into a mechanism for resource selection. We evaluate our mechanism by trace-driven simulation and prove that it provides high levels of service availability, highly reducing the number of migrations with respect to availability-agnostic resource selection and short-term prediction-based selection.

Chapter 6 presents the main conclusions of this thesis and the results that have been produced. It also indicates the main lines for future work that could follow this thesis.

2 CoDeS: A Middleware for Service Deployment in Contributory Communities

This chapter details our proposal of a middleware for contributory communities called CoDeS. After a technical description of related distributed computing paradigms, it defines the system model, and follows with a description of the architecture of the middleware. It ends with a description of the prototype we have implemented.

2.1 Overview

The goal of this thesis is to prove the feasibility of the contributory computing model and to create a foundation to build contributory communities. Our approach to achieve this objective is to design a middleware for the creation of contributory communities. This middleware should provide functionality to services and users. The former would use the middleware to transparently use non-dedicated resources, while the latter would use it to contribute resources to the community and access services. This chapter introduces the system model we propose for contributory computing and the architecture of our middleware for contributory communities.

Different system models like grid, peer-to-peer, volunteer and cloud computing have all dealt with partially intersecting sets of requirements, and have addressed many of them in different manners. Contributory computing also shares some of these issues, and may consequently share some of the solutions. For this reason, this chapter starts with a review of the models and implementations of previous distributed computing paradigms to which contributory computing is related. However, the specific combination of requirements presented in Section 1.2.4, together with specific challenges to contributory computing like autonomously providing service availability, has not been addressed by previous paradigms.

In this chapter we present CoDeS, a middleware for service deployment in contributory communities, which implements the contributory computing model presented in Section 1.2. The contributions of this chapter are as follows:

- **System model of a contributory community using CoDeS**. This includes specifying aspects like the types of services that can be deployed and how resources are contributed to the community.
- **Design of a modular and extensible architecture for CoDeS.** This architecture provides a good foundation for satisfying the requirements of contributory computing and, thanks to its modularity, can leverage technological advances made in some specific areas to improve the performance of some of its components.
- **A functional prototype implementation of CoDeS.** In order to validate the design, we

have implemented CoDeS using a mix of the novel mechanisms presented in this thesis, existing implementations of mechanisms like overlay networks, and simple implementations of the parts that are out of the scope of this thesis to serve as a proof of concept. Specifically, in this chapter we analyze the requirements of the different components of this architecture, present the decisions we have taken when implementing them in our prototype and propose possible alternative approaches for their implementation.

The rest of this chapter is organized as follows. Section 2.2 presents the architectural characteristics of implementations of the previous paradigms presented in Section 1.1, namely grid computing, desktop grids and volunteer computing, peer-to-peer networks and cloud computing. Section 2.3 details the system model, and Section 2.4 presents the architecture of CoDeS. Section 2.5 explains some aspects of the prototype we have built to validate the mechanisms and architecture of CoDeS. Finally, Section 2.6 concludes and presents some future work concerning expansions of the functionality of CoDeS and specifically aspects that could be improved in the current implementation of our prototype.

2.2 Related Work

As seen in Chapter 1, there are a number of previous paradigms that are essentially related to contributory computing. Many of the approaches used in such systems may be of use for implementing parts of the contributory computing paradigm. Therefore, we present an overview of the architectures used for building those systems.

2.2.1 Grid Computing

Grid's purpose was to permit the general aggregation of resources in different administrative domains. This requires interoperability, which can be achieved by defining common protocols to be used by all participants. For this purpose, they proposed an architecture divided in five layers [67], that organize protocols addressing different concerns. The architecture is based on an *hourglass model*. The lower layers must be able to deal with heterogeneous hardware and software components. Therefore, there is a wide range of protocols to control these components. The central layers have only a few core abstractions and protocols that can be implemented on a variety of substrates. This constitutes the narrow neck of the hourglass. At the upper layers, a variety of protocols can be implemented over the central ones to provide a wide range of application-specific functionality.

Figure 2.1 [67] shows the layered architecture of the grid. Layers are not strictly enforced, i.e., a layer can contact other layers than the one that is directly below it. Specifically, the Application layer may use the protocols of different layers to build the desired specific functionality.

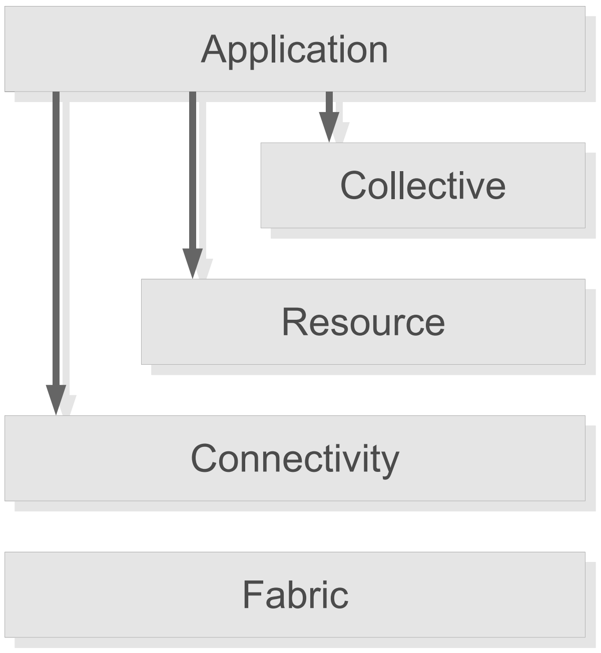


Figure 2.1: Layered architecture of the grid. Each layer contains protocols for specific functions. Protocols in the upper layers may use any of the protocols in the layers below.

Fabric

This layer provides access to the resources controlled by grid protocols. This includes computational resources, storage systems, network resources and sensors. A resource may be a logical entity that acts as an individual resource. For instance, it could be a network file system or a pool of computers. Its internal control protocols are not important for the grid, which only cares about the protocols that are used to access the resources externally.

Each resource may offer a different set of capabilities, depending on its type and on the capacities it implements internally. However, all resources should implement a minimum set of functionality in order to be shared in the grid. This should include *enquiry* mechanisms for discovering the state and capabilities of a resource, and *resource management* mechanisms to control the resource. For instance, enquiry mechanisms for a computational resource would include functions for determining hardware and software characteristics. Resource management mechanisms would include functions to start, stop and monitor programs.

Connectivity

This layer provides core communication and authentication protocols. It is part of the neck of the hourglass model, and therefore only a limited number of protocols are supported. The communication protocols must provide functions like transport, routing and naming. The standard protocols for these purposes are drawn from the TCP/IP stack. The security part, however, builds on existing Internet protocols to develop new standards that satisfy its requirements.

The specific requirements are that: a) the system should require a single sign-on per user, which allow them to access any resource; b) a user should be able to run programs in her behalf that inherit her permissions; c) security must be integrated with local security solutions; d) user permissions grant access to different resources without requiring the administrators of the resources to interact. These requirements are addressed by the Grid Security Infrastructure (GSI) protocols [33] implemented in the Globus Toolkit[1].

Resource

This layer allows remote access and control of individual resources. For instance, its protocols define how to start execution in a computer, involving all the steps of negotiation, initiation, monitoring, control, accounting and payment. They use the protocols on the Connectivity layer to authenticate and communicate with other resources, and the Fabric layer protocols to access and control local resources. This layer is, together with the Connectivity layer, the neck of the hourglass model, so the protocols are limited to a small and focused set.

The Grid Resource Information Protocol (GRIP) [52] is used to define resource information. The Grid Resource Access and Management (GRAM) [63] protocol is used for allocating computational resources and for controlling and monitoring execution in these resources. The Grid File Transfer Protocol (GridFTP) [4] is used for data access. The Globus Toolkit[2] implements APIs and SDKs in C and Java for each of these protocols.

Collective

This layer contains protocols and mechanisms that are not related to any single resource but rather are global in nature. Examples include finding resources or sending jobs to a pool of resources. This layer is in the upper part of the hourglass, on top of the narrow neck formed by the Resource and Connectivity layers. Therefore, it can contain a wide variety of protocols that implement different sharing behaviors,

[1]http://www.globus.org/toolkit/

[2]http://www.globus.org/toolkit/

including specific mechanisms tailored to the requirements of specific VOs.

Application

This layer contains the user applications that operate inside the VO and use the resources through organization boundaries. These applications call upon the services built at other layers, without having to care about the internal protocols that implement them.

2.2.2 Desktop Grids and Volunteer Computing

The main function of desktop grids is to use pools of non-dedicated resources to deploy computational tasks. They also must retrieve and store the results for the user that submitted the job. They can be, and some actually have been, integrated into the Collective layer of the grid architecture, as a way to control a pool of resources. However, they are implemented as complete systems that can perform all the steps of the distributed computation. Here we will review some of the most important platforms for desktop grids, namely Condor and Entropia, as well as the leading platform for volunteer computing, BOINC.

Condor

The Condor project [92, 139] started in 1988 with the purpose of creating a new system for distributed batch computing. Specifically, it allowed users to contribute resources to form a resource pool inside their community, and to submit jobs to be executed in that same pool. It focused on protecting individual interests, which led to its motto: *Leave the owner in control, regardless of the cost.* This was seen as a requirement for machine owners to remain in the community and allow other users to use their computer. Users can decide to execute Condor tasks under specific conditions, like when their computer has had no keyboard activity for some time of its load is below a given threshold.

The main components of the Condor architecture are three: agents, resources and matchmakers. Agents are computers that submit jobs to be executed in the community. Resources are computers contributed by users that can be used by other users to execute jobs. Matchmakers store the descriptions of the available resources and the jobs submitted by agents, and inform the interested parties when a match is made. Typically there is a single matchmaker per community.

Condor uses the ClassAd language [117] for matching resource requests with resource offers. Both agents and resources send their descriptions to the matchmaker using the ClassAd language. Each description contains a series of named attributes, with a given value. These values can be defined as integers or strings, expressions comprised of arithmetic and logical operators, or even complex data structures such as records, lists and sets. This semi-structured data model provides a lot of flexibility for defining resources and job requirements. The matchmaker simply compares the attributes defined by agents and resources using a three-valued logic which allows for undefined values. Therefore, it can deal with ClassAds defining different attributes. This matchmaking mechanism is also provided as a standalone open source software package in C++ and Java

Once a match has been made, the matchmaker informs both the implied agent and resource. After this, it no longer intervenes in the process. The agent and resource can independently verify the match and negotiate the claiming of the resources. The agent is in control of the execution, and must therefore remain available for the duration of the job.

In order to give absolute control to users, both agents and resources can specify which users they trust in their ClassAds. Specifically, agents can specify which computers they want to use for their job, or give preference to some resources. They may select resources that belong to a specific user or domain, or may define other kinds of groups, for instance, to select hosts that are close together and improve performance. Similarly, resources can specify who they want to serve, or give preference to some users.

Besides the supposed trust relationship between users, Condor also enhances user autonomy and security by providing isolation between jobs and resources. A technical reason for this is the difficulty of preparing the exact environment that the jobs expects for its execution. For this purpose it uses sandboxing. This is done by linking the job dynamically to a modified C library which redirects system calls to the agent. Therefore, the job actually interacts with the agent's system environment and file system. This is used only for applications programmed in C, but some extensions have been made to Condor to support applications in other languages, like Java.

Some additional components have been developed by the Condor project to allow users submit more complex series of jobs. Two examples are Master-Worker (MW) [76], which can be used to submit a large amount of jobs controlled by a master that can determine execution order and parameters in function of the results of completed tasks (useful for tasks like parameter searches), and the Directed Acyclic Graph Manager (DAGMan) [49] which can deploy multiple jobs with dependencies. Some extensions for data-intensive computing have also been developed [44, 85].

Condor has also been extended to allow interaction between different communities. The first method developed, Gateway Flocking [114], consisted of setting a gateway in each community which would communicate with other communities to send unsatisfied Class Ads. After the matching has been made by the matchmaker of one community, the agent and the resource interact directly like if they were in the same community. A different approach is Direct Flocking, where a user can individually send

ClassAds to multiple matchmakers, therefore participating in multiple communities. Other mechanisms were developed later to allow more complex interaction with grid environments, like Condor-G [68].

Entropia

Entropia [45] has a layered architecture, with three layers: physical node management layer, resource scheduling layer and job management layer. The physical node management layer controls all the aspects of remote execution, which are controlled by the central Entropia server, unlike Condor, which lets users directly control their executions. The resource scheduling layer assigns jobs to nodes, and the job management layer, where a job manager divides large tasks into subtasks to be deployed by the other layers. Compared to the grid architecture, the job management and resource scheduling layers would be located in the collective layer, while the physical node management layer would correspond to all layers below, including Fabric, Connectivity and Resource layers.

The resource scheduling layer contains mainly a subjob scheduler, which stores a list of available nodes, together with their characteristics, and a queue of subjobs to run, including their execution requirements. The subjob scheduler matches the resource descriptions with the subjobs requirements to assign executions to nodes. If a node with an assigned execution is unavailable for too long, the job is rescheduled in a different machine. After a number of retries, failed jobs are returned to the job manager, which decides whether to retry the job or return an error to the user.

Entropia can execute any Windows binary using sandboxing techniques [35] to isolate the application from the host. The binaries do not require any modification or recompiling, besides the binary modification done by Entropia that intercepts Windows API calls. The sandboxing environment controls use of resources and pauses or terminates applications that attempt to use too many resources. It also controls that the local user does not interfere with the application using encryption and consistency checks.

BOINC

BOINC[3] [5] is a platform for volunteer computing that is currently used in many successful projects (see Section 1.1.3). Its functionality differs from Condor in one fundamental aspect, which is a consequence of their different motivations. While Condor allows user to submit jobs to be executed in other users' computers while maintaining all users' control, BOINC simply develops a set of jobs in the volunteer computers. Since these jobs are created by the project managers, they are all managed from a centralized server, which controls the jobs, assigns them to the volunteers and retrieves and stores the results.

[3]`http://boinc.berkeley.edu/`

Volunteers earn credit from correct executions, which are published in world leader-boards to promote sporting competition.

The central server is composed of three different modules for different tasks. These are the web interface, which gives volunteers information about their performance and earned credits, the task server, which deploys tasks to volunteers, and the data server, which controls the file storage. They all access information stored in a set of databases, including application information and a list of pending jobs. Clients periodically contact the task server to fetch new jobs, using exponential back-off in case of failure to avoid overloading the server.

The task server [7] has to deal with the requests for jobs of hundreds of thousands of volunteers. For this purpose, it is composed of a set of independent processes that separately access the database and can be replicated for increasing performance. The main ones are the feeder, which reads jobs from the list in the database and puts them on a shared memory, and the scheduler, which answers client requests with a list of jobs that the client can execute, extracted from the shared memory updated by the feeder. The jobs are selected according to the characteristics reported by the client, thus allowing jobs with different requirements to be deployed in the project [9]. Jobs are replicated to be safe against erroneous results, caused by processing errors or cheating participants.

Applications deployed in BOINC must make use of a provided API to communicate with the BOINC client. They must implement functionality like checkpointing and displaying graphics, which may serve as an incentive for the user. The runtime environment [6] monitors application performance and stops applications that use more resources than the limit specified in the user preferences. The runtime environment also estimates the amount of work done by the machine, which is used to compute the credits earned by the user. Since different computers may provide different number of credits for the same job because of the way the work is estimated (multiplying benchmark scores by the application's CPU time), all users executing a specific job are granted the minimum credit amount computed by all the users that executed the job. The client also controls which application to execute and when to contact what server, since a user can be part of different projects [8].

2.2.3 P2P Networks

We already presented the P2P paradigm on Chapter 1. Now we will focus on the different architectures that have supported P2P applications over time.

Centralized architectures

This kind of applications use a central server to manage in some way the interaction among the participants. The server mainly allows users locate other users, which can then be directly contacted. Therefore, the true peer-to-peer interaction in centralized systems only starts after a query to the server has allowed a user to find another user. Further interaction does not require contacting the server and can therefore be considered as peer-to-peer interaction. However, the central point of the network may be attacked and prevent new connections to start, although it would not affect ongoing connections between peers.

In file-sharing applications like Napster and BitTorrent, the server is used to locate peers who own a copy of the file that the user wants to download. Napster had clients upload a list of their files to the server. Users could submit text-based queries to find files, and the server would return a list of peers who could serve the selected file. BitTorrent, on the other hand, does not have a central point for all BitTorrent users, having a centralized server (called *tracker*) for each file instead. The tracker lists users who have the file or are currently downloading it. Users are expected to find trackers through other means, like web searching. Extensions have been made to the BitTorrent protocol to use other techniques, discussed in Section 2.2.3, instead of centralized trackers.

Unstructured Overlay Networks

Motivated by technological and legal problems of centralized systems, completely decentralized networks were introduced in P2P file-sharing systems. This adoption started with unstructured overlay networks like Gnutella [122]. Unstructured networks are characterized by establishing connections between peers that do not follow any fixed deterministic structure. Random connections are typically used, although different heuristics can also be used to select nodes with desirable properties as neighbors.

In order to enter a decentralized network, a user needs to contact some node that is already part of the network. The entering node can then obtain a list of other participating nodes, and select a neighbor set from them. Nodes in an unstructured networks typically refresh this neighbor set periodically by obtaining a list of nodes from their neighbors. This makes the network highly resilient to failures, since hosts can easily correct their neighbor set by adding other nodes when some fail.

Although they can be considered scalable according to routing table size, they are regarded as not scalable when considering time and number of messages required to perform searches. These can work using both push and pull methods.

In the pull approach, a node that wants to find an item (a file stored in some nodes, or machines with specific characteristics) issues a query, which is propagated to its neighbors. These neighbors may either answer the query if they store the required file or have the specified characteristics, or forward the query to their neighbors. In a large network with unlimited propagation, this rapidly generates a huge number of messages. A TTL can be set to determine the maximum number of times that a message will be forwarded, but this limits the search to the nodes close to the host. Because neighbors are selected randomly, they may be distributed all over the network, but this does not provide any guarantee for finding rare or infrequent items.

In the push approach, nodes periodically submit their relevant information (list of items stored or resource characteristic of the machine, for instance) to their neighbors. This information may be propagated with a specific TTL, or by using more complex mechanisms to avoid or deal with outdated information. While this only implies a constant number of messages per node (which depends on the number of neighbors per node), it may not be scalable in terms of stored information if it is propagated without any limit.

Structured Overlay Networks

Structured overlay networks are completely decentralized and therefore share many of the advantages and some of the characteristics of unstructured networks. Like those, they have no single point of failure (technical or legal), but also need an entry point into the network. However, in structured networks, connections follow a specific scheme based on independent parameters, like the identifier assigned to each node by the overlay network. This causes that the overlay network topology adheres to a fixed structure which is independent of the underlying physical topology. This fact provides some desirable properties that are useful in many applications.

This kind of network presents a key-based routing (KBR) layer [54]. They create a limited identifier space, usually a uni-dimensional range limited by a number of bits, for instance, 128 bits in Pastry [123]. Hosts have a unique identifier, and the identifier space is divided among the existing hosts. Connections are distributed so that efficient paths can be found to any key. The KBR interface allows nodes to send messages to the node responsible for a specific key, usually with logarithmic cost in number of messages with respect to the number of nodes in the network (exact details depend on the specific protocol used). The size of the routing table stored in each node is also small, usually logarithmic with respect to network size.

Structured overlay networks are usually called distributed hash tables (DHT) because their structure can be compared to a hash table. Thanks to their KBR capabilities, they are adequate for storing key-value pairs in a distributed environment. However, they can also be used for building other applications such as multicast [37], or as as substrate for distributed file systems [32,53,107]. Some other applications of structured overlay networks have been briefly discussed in Section 1.1.4. More information on their use in distributed computing systems will be discussed in Section 3.2.2.

There are different structured topologies proposed in the literature. Most of them provide a KBR functionality. Next we present a brief overview of the most common ones.

Ring

Overlay networks with a ring topology assign a numeric identifier to each node, and identifiers are ordered in a circle. The number of nodes in the network is expected to be much lower than the number of possible identifiers, and therefore the population of the ring is expected to be sparse. Within this circle, each node, depending on the identifier that has been assigned to it and on the identifiers closer to his that are assigned to other nodes, is held responsible for a range of keys. Messages are routed using KBR, and in the case of DHTs, the node responsible for a range stores all the objects associated to a key in that range.

In order to route messages to the required key, each node stores a routing table containing the addresses of a set of other nodes. The requirements that the entries in this table must follow depend on the specific overlay network, but the usual condition is that they allow messages to be sent to different parts of the ring, in a way that can guarantee that messages will be routed to the node responsible for a key in a low, bounded number of hops.

The identifiers are usually generated randomly or by hashing some data related to the entity that requires the key. For example, identifiers for a node can be generated by hashing its IP address, while identifiers for objects might be generated by hashing their name or their content.

Examples of ring-based overlay networks are Chord [135] and Pastry [123].

Multidimensional space

Overlay networks like CAN (Content Addressable Network) [120] place nodes in a D-dimensional space. The identifier assigned to each node is composed of D values, each inside a range specified for the corresponding dimension, of the D defined. This identifier corresponds to a point on the D-dimensional space. Each node has a zone of the identifier space assigned to it. In a DHT, objects also have the same kind of identifiers assigned, and are stored by the node responsible of the zone to which they belong.

The identifiers to the nodes and objects in a D-dimensional space network might be assigned by different methods. They can be generated randomly, but the multidimensionality of identifiers can also be leveraged to address entities according to some criteria like a set of properties. For example, a set of D attributes of nodes can be defined, and the identifier of a node generated by its values on each attribute, e.g. CPU speed, RAM, disk space, etc.

Tree

Some overlay networks have the structure of a tree, where each node has a link to its parents as well as one to each of its children. There are a large number of different kinds of trees with different characteristics and properties in the field of data structures, and many can be directly translated into an overlay network. Overlay networks that use this structure leverage the properties of a specific type of tree to achieve the desired behavior of the system.

However, there are a few problems inherent to trees when translating them to distributed systems. The most obvious is the fact that the root of the tree becomes a point of centralization. Therefore, it can become a bottleneck or a single point of failure, hampering the performance of the network. This requires designers to develop workarounds to avoid starting searches on the root and overloading it. However, some of the common functionalities that a tree can perform, like information spread and aggregation, are highly desirable in some systems. Because of these, sometimes trees are used together with a different overlay network to leverage the advantages of both.

One example of an overlay network with a tree structure is Willow [141], which provides DHT functionality as well as publish/subscribe information dissemination and data aggregation.

2.2.4 Cloud Computing

Cloud computing is provided in three different levels [65]: *Infrastructure as a Service* (IaaS), where access to computer infrastructure, often virtualized, is offered to clients in a pay-per-use model; *Platform as a Service* (PaaS), where a platform for deploying distributed applications on external infrastructure is offered to clients; and *Software as a Service* (SaaS), where users can transparently access software over a network without regard of its implementation and infrastructure deployment. These three levels can be used as layers to build a system: applications deployed on a PaaS service can be offered as SaaS, and the platform itself can be built on top of resources provided by an IaaS vendor.

There are a number of cloud providers who offer different solutions for cloud computing. Moreover, the details about their internal implementations known to the public are scarce. Therefore, we will offer a brief description of the functionality offered by some of the most important cloud vendors, namely Amazon, Google and Microsoft. We will also describe some proposals for the creation of so-called private clouds (as opposed to public clouds offered by cloud providers) which try to leverage some of the advantages of cloud computing while maintaining data and computation inside the resources of an organization.

Amazon Web Services

Amazon Web Services (AWS)[4] offer a number of services that constitute its cloud offerings and can be used jointly or separately. Most of Amazon's services are relatively low-level, meaning that they offer access to resources (virtual machines) for deploying applications, and specific tools (queues, storage, etc) that can be used for building applications.

Elastic Cloud Computing (EC2) offers direct access to virtual machines. Users can configure their virtual machine image, using different operating systems, or take one of the images predefined by Amazon. New server instances can be obtained and booted in minutes, and they can also be shut down at any time. Amazon also offers an Auto Scaling functionality that can be used to automatically create or stop instances under certain conditions. Users pay for the amount of resources they consume, measured in CPUs per hour. Different classes of machines with specific resources are available, to adapt to specific user needs. These include instances with different amounts of memory or CPU speeds and GPUs.

EC2 Spot Instances offers access to unused EC2 instances by means of an auction mechanism. Users set the highest price they are willing to pay for using the resources, and the available resources are assigned to the users who bid the highest prices in an hourly auction. The price is set as the lowest of them. This makes resources unreliable, since at every auction (every hour) the user can lose the resource. However, users can get cheaper computational resources for applications that do not require high availability and reliability and can instead make use of sporadically available resources.

Amazon also offers storage services that can be used independently or in combination with EC2. These are the Simple Storage Service (S3), which stores key-value pairs, and the Elastic Block Store (EBS), which provides unformatted storage volumes that can be mounted in EC2 instances. S3 offers reliable and highly available storage (99.99% availability) for objects of up to 5 GB, charging for storage used an data transmission. It can be accessed via HTTP and the BitTorrent protocol. EBS, on the other hand, can create volumes of up to 1TB that can only be attached to one EC2 instance at a time, but offer reliable storage for the data stored in them.

They also offer other tools for building applications like the Simple Queue Service (SQS), which stores messages of up to 64KB to be shared among computers, and the Simple Notification Service (SNS), which offers a topic-based publish/subscribe service with HTTP/HTTPS and e-mail notification. Other services are applications deployed over Amazon's infrastructure which can be used also as tools. These include the Relational Database Service (RDS) and SimpleDB (a database with a simplified interface), MapReduce and CloudFront, which manages content distribution to clients over HTTP or

[4]`http://aws.amazon.com/`

streaming.

Google App Engine

Google's offering[5] differs from Amazon Web Sservices in that it is more oriented towards offering a platform for deploying web applications. Google App Engine provides an API which developers can use to create their applications. These applications are then deployed on Google's infrastructure. A certain amount of resources is freely provided to users by Google, and additional resources beyond that free quota can be purchased. App Engine is fundamentally oriented towards applications that require using only a limited amount of resources at a low price.

Users can only deploy applications programmed in Java (and other languages supported by the Java Virtual Machine) or in Python, but the system automatically manages runtime aspects like load balance and scaling. The API also offers a *datastore*, based on Google's BigTable [41], which applications can use to store all their data reliably (access to local storage is not permitted). Two datastore implementations are available: Master/Slave and High Replication. The Master/Slave Datastore sets a data center as the master and stores slave copies in other data centers, and therefore provides strong consistency but may have downtimes caused by data center issues. The High Replication Datastore uses Paxos-based replication to provide high availability, but only guarantees eventual consistency. However, Google has also announced that a new App Engine for Business[6] will include hosting of SQL databases

Windows Azure

Microsoft's cloud computing offer[7] includes two different brands. Windows Azure provides a platform for deploying applications, while SQL Azure provides hosted SQL databases. They are based on Windows Server and SQL Server technology, and offer both consumption and subscription payment models. Their platform approach is similar to Google App Engine.

Their computation platform offers different types of *roles*, resources that can be used for different purposes: the Web role, for deploying web applications, the Worker role, for generalized development, and the Virtual Machine role, which allows users to create and load a VM image. The latter is intended for existing Windows Server applications which would be difficult to install in the new platform, where the OS is not under the direct control of the developer. Web and Worker roles are intended to be combined for building services, where clients interact with a Web role and background processing is done by a Worker role.

[5] `http://code.google.com/intl/en/appengine/`

[6] http://code.google.com/intl/en/appengine/business/ - Retrieved on February 22nd, 2011

[7] `http://www.microsoft.com/windowsazure/`

Windows Azure also provides some additional storage and communication solutions [144] for use together with the computation platform and help building services. These include a queue service for communicating roles, a binary data storage called *blob*, which can be used for storing and streaming content such as audio and video or to provide a file system to the roles, and a schema-less table storage.

Private and Hybrid Clouds

The resource usage model in cloud computing is not only useful for selling resources to external entities. It can also allow flexible and efficient resource usage inside an organization. This is done mainly through virtualization, that permits users to easily deploy applications on physical servers [133]. Since virtual machines can be easily created, destroyed and migrated, the amount of resources dedicated to each application can vary dynamically. For example, an application with small load can deploy a virtual machine with limited resources on a server, leaving the rest of the machine unaltered and with its unused resources being usable by other applications. If the load of the application grows, the virtual machine can be resized or other virtual machines executing the application can deployed.

The main challenges of these resource usage model are providing a uniform view of virtualized resources that allows to create and control virtual machines with different platforms (such as Xen[8] [21], VMWare[9], VirtualBox[10] or Kernel-based Virtual Machine[11]), and dynamically allocating resources while following organization-specific policies and adapting to changes in resource needs. A number of frameworks and toolkits have appeared for the creation of such private clouds, which use and manage virtualization on top of an organization's resources. They also allow the creation of hybrid clouds, by interacting with public clouds and purchasing additional resources when needed, allowing users to transparently manage both internal and external resources using the same interfaces. These toolkits include both commercial solutions (VMware vSphere[12]) and open source developments (OpenNebula[13] [133], Nimbus[14], Eucalyptus[15] [109]).

[8]http://www.xen.org/
[9]http://www.vmware.com/
[10]http://www.virtualbox.org/
[11]http://www.linux-kvm.org/page/Main_Page
[12]http://www.vmware.com/products/vsphere/
[13]http://opennebula.org/
[14]http://www.nimbusproject.org/
[15]http://open.eucalyptus.com/

2.3 Contributory Computing System Model

Section 1.2.4 presented the requirements of a contributory computing community and the software supporting it. In this section we will show the approaches taken in various aspects, and how these elections are geared towards fulfilling the requirements. This includes concerns about the architecture of the system, how members can contribute resources, and the kind of services that can be deployed on the system.

2.3.1 Architectural Concerns

After reviewing some existing distributed models and architectures, it is obvious that some of the issues and challenges of creating a contributory community have been previously solved or are part of currently active research areas. However, none of them completely fulfilled the specific requirements that characterize contributory computing. The conclusions extracted from the works reviewed will guide the design of our middleware.

Platform

The purpose of a contributory community, as stated in Section 1.2.1, to deploy services using the non-dedicated resources contributed by members. To make it possible for users to create their own communities, it is necessary to have a platform that can automatically aggregate resources and use them to deploy services. This platform approach is similar to those provided by existing PaaS vendors such as Google or Microsoft.

In order to provide this platform, we consider it necessary to design and implement a middleware that can be used to aggregate resources, automatically and autonomously deploy services on them, and also provide tools for building applications to be deployed in a community. This middleware should be easy to install and use, to allow communities to have a working platform with little effort.

Modularity

Grid computing proposed the creation of a layered architecture, where each layer contained a set of standard and open protocols for performing a well-defined set of tasks. This allows interoperability by having different systems implement a common protocol at the required part, notwithstanding implementation differences in the other parts of each system [67]. Moreover, it also allows higher efficiency of the overall system in two ways. On the one hand, the systems can continuously improve because existing components can interact with new enhanced implementations of the protocols as long as they maintain the defined interface. Therefore, research results that improve the existing mechanisms in some way can be implemented and easily incorporated into a working system. On the other hand, users can choose among a variety of implementations to select the one that better suits their infrastructure and software requirements.

Other systems, like P2P applications and desktop grids, may also present layered or modular architectures, but they are usually more tightly coupled. This is understandable because their purpose is to provide a functional application. This purpose is different of large standardization frameworks, like grid computing or the recent efforts to create interoperable cloud platforms [133] commented in Section 2.2.4, that need to make large existing systems interoperable with the minimum internal changes.

Contributory computing does not aim at joining existing infrastructures, but instead tries to aggregate existing resources and use them in novel ways. Therefore, interoperability and protocols are not a major focus of our middleware. However, the other advantage of modularity is that it facilitates extendability, which is critical for contributory computing for two reasons. The first one is that contributory computing is a novel paradigm and therefore lacks actual usage data. Real deployment of a contributory community will allow to take real performance measures and user behavior. The system will mature as emergent user requirements are found and new mechanisms are developed to satisfy them. Therefore, the system must be modular and extensible to be able to accommodate user requirements that are discovered in a real production deployment. The second reason for modularity and extendability is that many of the subproblems of building a contributory community are also studied in the context of other paradigms. Being active research fields, it is expected that alternative mechanisms or improvements with respect to existing ones will appear over time. It must be possible to incorporate these new advances into the middleware to enhance its performance and adaptability.

Decentralization

As stated in Section 1.2.4, a contributory community must work without any fixed infrastructure. This is different to the approach taken in most of the systems discussed in Section 2.2, like grid and cloud computing, desktop grids and some P2P applications. In these systems, some specialized components were present in defined locations of a network and managed different parts of the system, for instance, how tasks are assigned to resources.

A contributory community should be able to operate depending only on the resources contributed by members,. Therefore, no specific assumption should be made about the capacity or availability of any single resource. Specifically, it is not possible to expect any one node to be able to continuously and efficiently perform a critical function of the system. Therefore, it is necessary that the community self-organizes and autonomously assigns specific roles to each node. For this purpose, all the members must have a common middleware that allows them to take any function in the system. Users must be able to access every functionality of the platform from any node that is member of the community.

2.3.2 Resource Contribution

Members of a contributory community voluntarily donate a portion of their resources to be used by the community. However, as considered in desktop grids and volunteer computing systems discussed in Section 2.2.2, users should always remain in control of their resources. For this purpose, they should be able to decide what resources to contribute, when will they be contributed and for what purposes will they be used (for instance, only for community management mechanisms or only for executing some specific services).

The user needs tools to administer the contribution of resources in an explicit way, addressing the three main questions (what, when and for what). However, it would be desirable to have also ways to define how the resources will be used in an implicit way. For this purpose we define two main ways to control the contribution of resources: contributing on idle, and contributing always. It is vital to clarify how resources can be contributed to the community in order to design mechanisms that can aggregate and use these resources collectively.

Contributing on idle

Contributing on idle means that a user's resources will only be used by the community when the user is not using them. This approach is the same taken, for instance, in Condor [139] and BOINC [5], where computation is only started in a host after no keyboard or mouse activity has been detected for some time, and interrupted whenever there is user activity.

An alternative approach is to take the decision of whether sharing the resources based on the load of the computer. This requires monitoring the usage of different resources, like CPU or memory, of the applications controlled by the user. Resource contribution would be started when the load of the host goes below a threshold specified by the user.

The two methods are slightly different. In the keyboard/mouse approach, the resources are shared when the user is not interacting with the computer. Load-based contribution, instead, shares resources when the user is not making full use of the computer, independently of direct interaction. To see the difference between both, consider the case of a user who is reading a document on the screen, which may require keyboard or mouse interaction but may only consume a small portion of the resources, and the case where a user starts a compute-intensive task in the machine which requires no interaction. A combination of both methods could be used to allow the user complete control by specifying whether to contribute resources based on both user activity and load.

Contributing always

A different way to control resource contribution is to decide a fixed amount of resources that can be shared, and contribute this constantly. For instance, if a computer has a fast processor and a lot of memory, it could decide to contribute a specific proportion of these resources, and still have enough for the local user to work fine. This is supported by reports of resource underutilization of both desktop computers and servers [57].

This can be achieved through virtualization. A user may define the amount of resources that will be contributed, and instantiate a virtual machine of that specific capacity on top of his physical resources. By setting the amount of each type of resource contributed to an appropriate value, a user could transparently and continuously contribute resources to the community. Of course, the user would always retain the freedom to explicitly withdrawn the contributed resources.

Hybrid approaches

Finally, hybrid methods could be used to allow users to contribute an amount of resources that evolves dynamically. For example, the user could define a minimum amount of resources that would be virtualized and contributed continuously. The user would also define a larger threshold for the amount of resources that can be used under certain conditions, as defined in the *idle* policy (i.e., a combination of system load and user activity). When the conditions are met, a larger amount of resources could be contributed, whether by dynamically incrementing the capacity of the virtual machine that is being contributed, or by creating a new virtual machine with the additional resources that are contributed. While the latter may be technologically easier to implement, it will have more overhead, because of the need to maintain two virtual machines, and will provide less flexibility as the resources contributed by the user could not be used together, but as two different sets of resources.

In any case, community members are considered to contribute a specific amount of resources which can be defined in a resource description document, whether this amount of resources constitutes the whole physical machine or just a part of it. The amount of resources available for the community may vary, because of dynamic changes in the contribution as just mentioned, or because a service has been deployed in the host using only a part of the machine's resources. The fact that users contribute continuously or on idle will only change the rate of entrances and departures in the system and of changes in the amount of contributed resources. The specific mechanisms to control this on the individual machine level are out of the scope of this thesis.

2.3.3 Services

The characteristics of non-dedicated resources impose a number of restrictions to the systems that can be deployed on them. Applications that are designed to be executed in a dedicated and well-maintained service may not work correctly when developed in such an environment. It is therefore necessary to define the kind of services which can be deployed on a contributory community.

Stateless services

The simplest type of service that can be deployed is stateless services. The behavior of these services, i.e., the answer to client queries, does not depend on previous queries. Therefore, these services can be easily migrated or replicated, since each instance of the service can correctly answer queries independently. They can be located anywhere where the clients can access them, since they have no dependence on the location of other instances or some specific data besides that which constitutes the service itself.

One example of this type of service could be a web server that serves static content. It does not track connections nor allows web pages to be dynamically modified by clients. Another example would be a file converter, which converts a file sent by a client in a specific format into an equivalent file in another format (for example, converting audio or video files to different encoding schemes). The result returned by such a service (the file in the target format) only depends on the content of the query (the original file in the source format).

Three-tier services

The next type of service that can be deployed on a contributory community is three-tier services. The three-tier architecture is often used for web applications. In these systems, there are three tiers, namely the presentation tier, the application tier and the data tier, as shown in Figure 2.2. The presentation tier is the one that presents information to the user. Therefore, it is usually located in the client machine. The application tier is the one that manipulates the data following the application logic. The data is stored in a data tier, from where it can be retrieved by the application layer. Each of these layers can be deployed in different hosts or replicated.

The application layer acts like a stateless service that obtains information from both the presentation tier (in the form of queries) and from the data tier (the data that is required to answer the queries). It also may send information to the presentation tier (as answers to the queries) or to the data tier (modifications to the data model caused by the client queries). The application tier may be replicated, as long as all the replicas have a consistent view of the data tier. No other communication or coordination is needed among different instances of the application tier of a service.

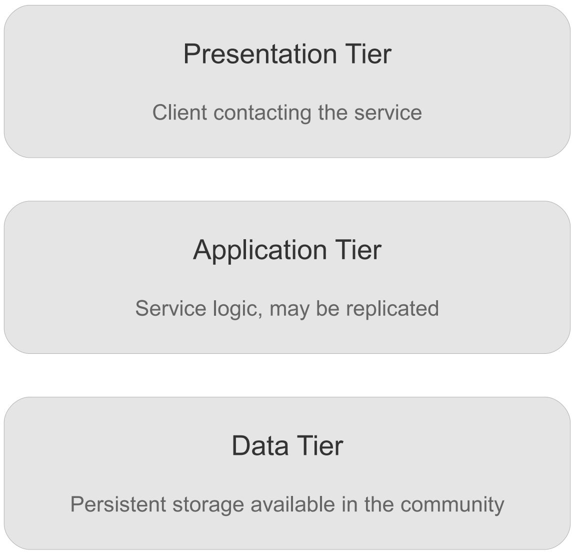

Presentation Tier
Client contacting the service
Application Tier
Service logic, may be replicated
Data Tier
Persistent storage available in the community

Figure 2.2: Architecture of a three-tier service.

Therefore, the application tier of a service following a three-tier architecture can be deployed on a contributory community. The presentation tier may be located on the client host, and the data tier in a separate storage service that is available to the community. Therefore, deploying services (or the application tier of a service) and offering a consistent data storage which can act as the data tier for these services are separate concerns.

Distributed services

The last type of service that can be deployed includes those services that are distributed in nature, or that can have a distributed implementation. Such implementation consists of a set of processes, running at different physical locations, that communicate with each other to carry out their intended purpose. These services must be able to deal with dynamic membership, as instances of the service are destroyed and new instances are created. They must implement their own coordination mechanisms so that client queries received at any one instance of the service can be correctly answered.

Implementing such a service does inherit many of the complexities of a contributory community, specifically those that stem from a dynamic and decentralized environment. However, even such a service benefits from being deployed on top of a contributory community instead of just being executed in a set of collaborating hosts with no middleware support.

The first advantage is that the service constitutes an overlay network deployed on top of the community (which can be considered itself an overlay network, as seen in Figure 2.3). This means that the amount of nodes that form the service can be adapted to the requirements of the service, considering, for example, the load of the service. Not all hosts in the community need to install and run the specific software of the service, and still all members can access the service. Moreover, updating the software to a new version can be done easily by updating the version that is available in the community (however this version is stored and used to deploy instances of the service, and including updating active instances), instead of requiring all members to update their version.

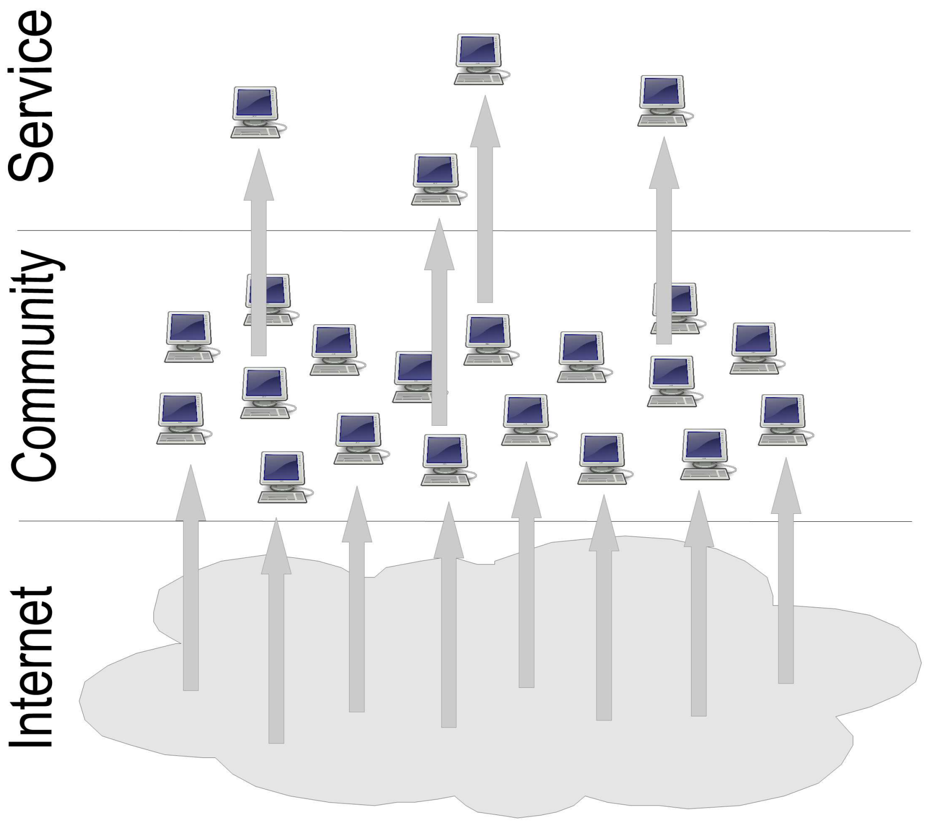


Figure 2.3: A distributed service deployed in a community. The community is formed by a subset of the hosts in the Internet, and the service is formed by a subset of the hosts in the community.

The second advantage is that the service can use the tools provided by the middleware. For instance, a necessary step in any decentralized system is discovering an entry point, i.e., a node that is already part of the system, which can be contacted by an entering node in order to join. The fact that the middleware of the community provides the functionality of finding active instances of a service (which is necessary for users to access the service) trivializes this step, as a new instance can use it to find the rest of them and join the distributed system that constitutes the service. Other tools and functionality that may be useful for implementing a distributed service will be detailed later.

2.3.4 Open Issues

In the previous parts of this section we have presented what we consider the core aspects of contributory computing, which make it a novel and interesting approach. Next we will discuss some other aspects that could further enrich the model of contributory computing and the middleware that implements it. However, they are not necessary to validate our proposal, and therefore they will not be integrated into the core mechanisms of our middleware.

Security

We mentioned the main security issues that a contributory community may find in Section 1.2.2. While this is an aspect of utmost importance for the real deployment of a community, security is not the focus of this thesis, and we will only briefly discuss the topic at a general level. The security risks in a contributory community include, on the one hand, privacy risks caused by users having access to sensitive information belonging to other members, and, on the other hand, the risk of malicious users taking partial control of the community. On a first approximation, we group the latter in three main categories: malicious users deploying services with a malicious behavior, malicious computers executing legitimate services, and malicious users interfering with the management of the community (which may lead to risks of the two first categories).

There are a number of ways of dealing with malicious services. One would be restricting the rights for deploying services to a subset of trusted users. While this may be fine for some communities, it may not be so for others which have more open policies. Reputation mechanisms or even other simple approaches like blacklisting can be used to identify malicious services and avoid using or executing them. However, this would allow a service to damage or abuse the hosting machine until the service has been identified as malicious. Moreover, a service can cause failures because of other causes than malicious behavior, like programming bugs. Therefore it is necessary to use other alternatives to protect hosts from the service they execute. This protection can be achieved by enclosing the service in a virtual machine [21, 35], which has other advantages like control of the amount of resources used.

The case of malicious computers executing legitimate services can be more tricky. Even using virtualization, a user who has complete access to the physical machine (as may be the case of the individual owner of the computer) may find a way to access and modify the data of a service [35]. For services that work with sensitive data, the easiest solution is to keep all data encrypted when in untrusted nodes, and only decrypt it in the host of the end client. To the best of our knowledge, this is the only possible solution to the problem.

Whatever methods are used to keep the data of a service unavailable to the owner of the machine that is hosting it, the malicious user may decide to execute a modified version of the service (or an altogether different service with a superficially similar behavior) instead of the legitimate one. In this case, the member can return wrong results to the clients of the service that try to contact that particular instance. To solve these problem, some services can leverage replication to implement simple quorum-based solutions. For example, a user contacting a service may be required to contact a given number of replicas of the service before assuming the returned result as correct. More sophisticated solutions [38,39] have been developed to attain Byzantine fault tolerance in asynchronous environments with good response times.

The third category of security risks refers to the fact that, since management is distributed among the members of the community, a malicious user could try to control certain aspects like routing of messages, service deployment or data storage. Moreover, a single malicious entity could enter the community simulating a number of different identities [58], and therefore obtain a great control over the activities of the community. One method to avoid this is using strict policies to control group membership, which allow only very trusted individuals to become part of a community. However, enforcing such policies may be hard or even inconvenient for many communities

Therefore, management mechanisms should take into account the possible existence of malicious nodes in the community. We will consider two approaches that can help deal with this problem. One is verifiability, i.e., management functions are assigned to nodes in a way that is verifiable by any other member, for example in a deterministic way. This prevents malicious nodes from arbitrarily claiming control over community management functions. The second approach is to allow some degree of local decision at different points of the management or in user actions. For example, several replicas of a service may be presented to a member, from which she might then choose one that she trusts. This allows mechanisms to be combined with complementary techniques like trust or reputation management mechanisms.

A combination of these two approaches can reduce the damage that malicious users can do to the community. We will consider the use of a combination of both when designing the mechanisms that manage the community. However, we must note again that security is not the main area of this thesis, and therefore the security of the mechanisms will not be the main focus on their design. Moreover, we will not define nor select any specific mechanism for membership control on the community.

Composite services

The types of services that may be deployed on a contributory community are those that fit in the definitions provided in Section 2.3.3. In summary, the services must be composed of a number of identical instances which can be dynamically created, destroyed or migrated. However, other kinds of services could be thought of, which present a more complex distributed architecture. Such services would be composed of a number of individual components that could be deployed on different machines. Deployment of such services could be handled by an upper layer, where individual components would be treated as independent services by the middleware. However, this could still require additional considerations related, for instance, to the location of each component, since they may possibly require inter-component communication. This thesis does not address the design and development of such mechanisms.

Incentives

Since a contributory community depends on the resources contributed by its members, encouraging contribution is important to achieve the objectives of the community. Therefore, incorporating mechanisms to provide incentives to members could be of great use for a community. It could be useful, for instance, to have a credit system similar to that used in volunteer computing projects, specifically in those built on top of BOINC [5]. This could promote contribution while being true to the collaborative and contributory nature of the community.

Such an accounting mechanism could also be used to enforce some contribution policies, like requiring a minimum contribution to be part of the community or giving privileges to members who contribute more. However, it is not trivial to build such a system in this kind of environment. This is a consequence of the decentralized and dynamic nature of the system. Moreover, resource usage is decided autonomously by the middleware, i.e., services are deployed in a node without direct decision of the involved users. Therefore, it is difficult to measure contribution separating how many resources have been offered to the community by the member and how many have been actually used. Since we do not consider that it is an essential part of a contributory community, we will not incorporate it into the middleware.

A different approach is to restrict resource usage, allowing a member to use only as many resources as she contributes. This is applied in some distributed applications, for instance in P2P applications like file-sharing [113] or in some desktop grids [11], to prevent users from using a large amount of the resources of the network while not contributing, at least, proportionally, which is known as *free-riding*. However, in a contributory community, resources are used collectively to deploy services. It could be possible to devise an scheme where users could be charged by using resources to deploy their services, as well as for the resources used to satisfy their queries to other deployed services. Nevertheless, we believe this is not an adequate way of controlling resource usage and contribution in a contributory community.

Resource scarcity

The services that a community can deploy depend directly on the resources that its members contribute. Moreover, the amount of resources that can be assigned to each service (for example, to create more replicas of a service with a high load) also depends on the amount of resources available to the community. However, in the design of our middleware we will consider that a community has enough resources for the services that it wants to deploy, and for the amount of resources that it wants to dedicate to each service.

Additional mechanisms could be defined to deal with situations of resource scarcity, where the resources are not enough to deploy all the services and the system needs to decide which services to prioritize and how, possibly following a community-defined policy. However, we will not develop these, and our middleware will provide no guarantees about the state of a community and its services when there are not enough resources to deploy them all.

2.4 CoDeS Architecture

In this section we present our proposal of an architecture of a middleware for decentralized service deployment in contributory communities. We have named this middleware CoDeS. This architecture is modular and extensible. All the components are installed in every node of the community. Each module implements a part of the functionality of CoDeS, and modules interact with each other in the local host to have access to their respective functionality.

For applications, CoDeS can be seen as a Service Directory (SD), as shown in Figure 2.4. Users can register a service in the directory, providing a unique identifier for the service and a specification. The latter must include the requirements of the service, the number of replicas that are needed and a list of files required for executing the service together with other information required to prepare the execution environment. When a user sets a service's state as *active*, CoDeS makes sure that the required number of replicas are available.

An application can query the SD using a service's id to get a list of locations where the service is currently hosted. Knowing these locations, the application can directly access an instance of the service. However, since service locations might change at any time, applications must implement some failure tolerance mechanisms, whether proactive (e.g., periodically repeating the query to have up-to-date location information) or reactive (e.g., detecting failure of a service instance through a time-out counter, and repeat the query to find a new location).

CoDeS implements this Service Directory in a distributed and decentralized way. Services in the directory are maintained autonomously in presence of churn, and they can be deployed over a network of heterogeneous nodes. A single interface for applications mediates access to the functionality provided by each module of the middleware.

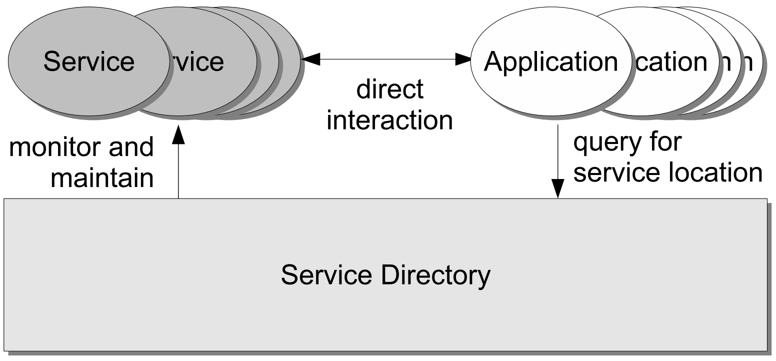


Figure 2.4: Applications can query the Service Directory to discover the locations of a service.

The modules that form the middleware are presented in Figure 2.5. Some modules execute a functionality that is distributed in nature, and they use distributed mechanisms to do so. In these cases, the module instance in a node interacts with instances of the same module in other nodes. This interaction between different nodes is mediated by the connectivity layer. All communication between different modules happens locally, i.e., a module can interact with the other modules in the local node, but only with instances of the same module in remote nodes. This clarifies separation of concerns inside the architecture, keeping functions, even distributed ones, contained in a single module.

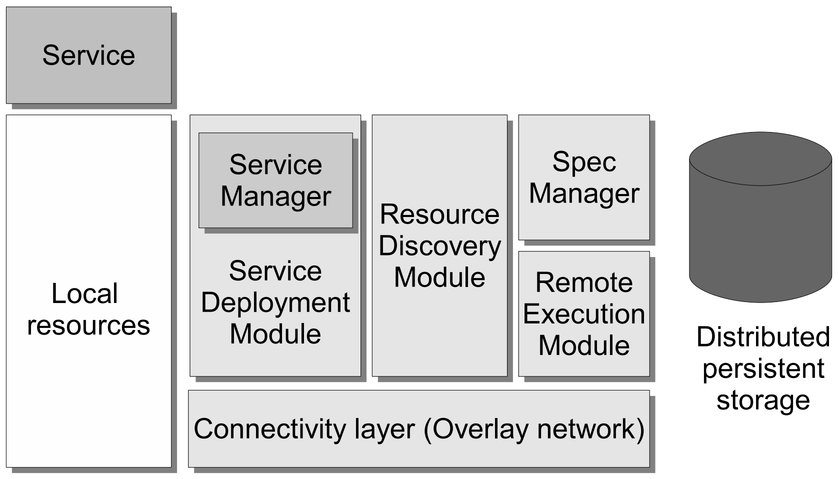

Service
Local resources
Service Manager
Service Deployment Module
Resource Discovery Module
Spec Manager
Remote Execution Module
Distributed persistent storage
Connectivity layer (Overlay network)

Figure 2.5: Architecture of the middleware. All the modules are present in every node executing CoDeS. Some of them provide their functionality by cooperating with the modules in other nodes using distributed mechanisms.

2.4.1 Local Resources

This module controls the use of the local resources of the node. This includes enforcing which is the amount of resources that can be used by the community, and when it can be used. As detailed in Section 2.3.2, the contribution may be controlled in different ways, including virtualization. The local resources module provides the other modules with a description of the available resources at any given time. This is used by the Resource Discovery Module to periodically update the resource information that is published to the rest of the community.

It also offers an interface to start, stop and monitor services. In order to start a service, the local resources module needs a service specification document, which contains all the information required to execute it. This includes the information about the executable files, how to retrieve them and how to start execution. This module also has an interface to the tools provided by CoDeS for the service. This allows the service instance locate and contact other instances, access the distributed persistent storage or use other middleware functionality when required.

A local resources module may also aggregate the resources of a number of computers. The access to these resources from the rest of the community is then mediated and managed by this module. For instance, it could use grid or desktop grid software to manage a pool of resources and provide an abstraction to the community. A hierarchical community could be created by this method, with only a number of hosts participating directly in the community and acting as mediators to larger pools of resources. However, this should be transparent for the rest of the middleware, which will treat each resource as an individual physical machine.

2.4.2 Service Deployment Module

The Service Deployment Module (SDM) controls the collective use of the resources to deploy services. It does so using a distributed mechanisms, with the collaboration of the SDMs in all the nodes in the community. It uses the Resource Discovery Module (RDM) to find resources that satisfy the requirements of each service. The details of its mechanisms are presented in Chapter 3.

2.4.3 Resource Discovery Module

The Resource Discovery Module (RDM) is in charge of finding computers that satisfy specific requirements. Each hosts uses it publish a specification of its capabilities and characteristics, including the amount of resources that are contributed. The RDM can work with any resource specification format, as long as the documents used for resource specification and for the definition of service requirements can be matched by the Specification Manager. The details of the resource discovery mechanisms are presented in Chapter 4.

2.4.4 Specification Manager

This module translates and compares different formats of resource specification. These formats are used to define both the resources that each member contributes and the requirements for the execution of each service, and the Specification Manager is in charge of deciding if a pair of specifications (one for a resource, and one for the requirements of a service) match.

By implementing different Specification Managers, different specification formats can be used in a community. Moreover, an implementation of the Specification Manager may be able to interpret and compare documents presented in a variety of resource specification formats. All members in the community must use a format that can be interpreted by the Specification Manager implemented in the community, for both the definition of the resources they contribute and the requirements of the services they deploy.

2.4.5 Remote Execution Module

The Remote Execution Module (REM) controls access to remotes resources. When a CoDeS component in a node intends to start an execution using the resources of another node, it contacts the local REM to do so. The REM then contacts the remote REM through the connectivity layer. The remote REM asks the local resources module of its node to start the execution. Communication between the two REMs is periodically established to monitor the state of the execution, and if the execution fails or there is any other event that has to be reported, the local REM informs the module that ordered to start the execution. The behavior of this module is explained in more detail in Chapter 3.

2.4.6 Connectivity Layer

This layer abstracts and controls communication among the different nodes. It must allow two kinds of communication: direct node-to-node communication, and key-based routing (KBR). Many of the core requirements of contributory computing are addressed in this layer, including scalability, failure tolerance and decentralization.

The connectivity layer must allow node-to-node communication independently of the physical network and the specific protocols used at the lower layers. For example, a connectivity layer may implement communication over direct links in a TCP/IP network, using NAT traversal techniques, on an overlay network or even on a simulated network, useful for testing purposes. This must be transparent to the other modules of the middleware. For this purpose, the connectivity layer offers an abstraction to identify and contact nodes, the node handler (NH). A node handler can be used by any module to send messages to the node represented by it. Typically, in an IP network, it will contain an IP address and a port number. This data may be extracted from the specific node handler data object by the user applications and services, if they require so for their communication processes out of the connectivity layer.

The KBR functionality of the connectivity layer must adhere to the common KBR API [54] implemented by most structured overlay networks. This is basically composed of functions for sending a message to a host responsible for a specific key, checking the range for which the local node is responsible, obtaining a list of neighbors. The KBR abstraction is useful for building decentralized and self-managed networks, as discussed in Section 2.2.3, and specially for building distributed indexes. It provides a number of desirable features for many distributed systems, including contributory communities, like scalability or failure tolerance. For this reason, research on structured overlay networks has been an active field of study.

The different modules of our middleware will only rely on the functionality provided by the common KBR API. This allows different implementations to be used at the connectivity layer. Therefore, a community can choose any specific KBR implementation which is found to better fit the requirements of the community, comparing the characteristics and features of each of the existing products. Moreover, CoDeS can benefit from improvements of KBR networks achieved in areas like efficiency or security, among others, once they have been implemented into a complete KBR product.

2.4.7 Distributed Persistent Storage

This module provides access to the persistent storage where the data of the community is stored. It is used by the mechanisms that CoDeS uses to manage the community, and it is also a tool to be used by services and user applications. The details of how the different mechanisms use this module are presented in Chapters 3 and 4. These mechanisms, however, only require a simple interface that allows storage of files and data objects associated to a key, through *put* and *get* methods. This fits the key-value abstraction of distributed hash tables (DHT), which are implemented as an application on top of a KBR layer. Therefore, a DHT may be used on top of the connectivity layer to implement this module.

Services and applications, however, may benefit from having a distributed persistent storage with additional features, like a distributed file system, or a storage with some specific guarantees in aspects like consistency and availability. Therefore, implementations of the distributed persistent storage may offer additional interfaces besides the *put/get* interface used by the CoDeS mechanisms.

2.5 CoDeS Prototype

We implemented a prototype of CoDeS in order to prove the feasibility of the concepts and mechanisms presented in this thesis. When implementing it, we tried to keep a focus on the properties specified in the high-level design, namely modularity and extendability. This has allowed us to implement different mechanisms in our prototype and easily incorporate changes and improvements on existing ones. We hope that it will also allow future extensions and improvements of the middleware.

We decided to implement our prototype in Java for a number of reasons. First, a community will be formed by heterogeneous nodes. Therefore, CoDeS should be compatible with a wide variety of system configurations, including computer architectures and operating systems. The Java language and the Java Virtual Machine (JVM) where it is executed provide that. Java is very popular and a JVM is available in most computers. Moreover, its popularity has caused developers to implement a great number of libraries that provide a large body of functionality. For instance, there are implementations of KBR overlay networks, as well as implementations of many grid protocols and formats for resource specification management. This simplifies building CoDeS because some of the modules can use these external libraries, which fits the philosophy followed in the design.

Another reason to choose Java is that it is an object-oriented language, which naturally leans toward modularity. Interfaces can be used to abstract methods with independence of their implementation, and classes can be loaded dynamically into a program and only specified at runtime. This allows us to implement each of the modules with independence of the rest. In many points, the middleware allows users to select the specific implementation of a component that they want to load. This is done by specifying the class name in the appropriate field of a configuration file, and the class is later dynamically loaded by the middleware when the associated functionality is required. This way users can choose between a variety of implementations provided with the middleware to select the one that better fits the specific requirements and characteristics of their community, or they can develop their own implementation. The only requirement is that they adhere to the interface specified for the component they implement, and that the user-specified class can be found in the classpath (the list of directories where the JVM searches for classes).

The next subsections will explain how some of the modules that form CoDeS have been implemented in our prototype. However, we will not detail here some of the modules, namely the Service Deployment Module, the Remote Execution Module, the Resource Discovery Module and the Specification Manager. These are part of the service deployment and resource discovery mechanisms, which are by themselves contributions of this thesis and will be presented in all detail in Chapters 3 and 4.

2.5.1 Local Resources

The local resources module may be implemented in many different ways, using a variety of sandboxing and virtualization techniques to control the use of the resources and provide isolation between the service and the host. However, in our prototype we have only developed a simple implementation to allow the execution of Java classes as services. This is intended to work only as a proof-of-concept implementation, and does not provide most of the features that would be desired by a community for real service deployment. We have successfully used it to deploy a number of simple services. Moreover, Java classes deployed as services may also start other executable programs natively in the operating system. However, techniques for providing security, isolation and control of the amount of resources used should be implemented in this module.

One example of service that has been deployed using CoDeS has been a modified version of Jetty, a lightweight web server programmed in Java and distributed . When the modified web server receives a query from a client, it retrieves the content that has to be served from the distributed persistent storage of the community, and then serves it to the client. The content is stored in a local cache for serving future queries.

2.5.2 Connectivity Layer

The connectivity layer in the prototype is an interface which provides all the methods on the common KBR API as well as methods for direct communication between nodes. Other interfaces and abstract classes are also part of the connectivity layer, including node handles. All these classes are instantiated at runtime on the specific implementation that has to be used in the community. Any implementation of these interfaces can be developed and loaded without requiring any changes in the rest of the middleware. This is a part of CoDeS where extendability is critically important because of the large number of structured overlay network protocols that have been proposed over time [81, 97, 98, 102, 120, 123, 131, 135, 146], the applications that have been designed on top of them [32, 37, 53, 107], and the continuous improvements on aspects like security [140] developed by the research community.

The connectivity layer implementation that we have developed and used is built on top of FreePastry[16], an open-source implementation of the Pastry overlay network protocol [123]. It is programmed in Java, which makes it easy to incorporate into our CoDeS prototype. FreePastry can be deployed over a TCP/IP network, and includes optional capabilities like NAT traversal and secure communication through SSL. It can also be extended to implement, for instance, different transport-level protocols and to add object-specific methods of data serialization.

[16]`http://www.freepastry.org/FreePastry/`

FreePastry also provides a simulation mode where a network formed by a large number of nodes can be simulated in a single machine. Using it requires no modification on the application code, which can work transparently with a real or simulated network. It simulates transport delays using euclidean or spherical topologies or through a user-supplied latency matrix. It does not simulate some of the factors that other low-level simulators include, like message loss, bandwidth or varying latency. However, its functionality allows testing applications in a wide enough range of conditions, and has been used for our tests presented in Chapters 3 and 4.

2.5.3 Distributed Persistent Storage

We have incorporated a few different persistent storage implementations on our prototype. These have the purpose of serving as a proof-of-concept and supporting the basic functionality of the middleware to allow validation in a distributed environment. Therefore, a more thorough study would be required to select a specific set of mechanisms to implement a distributed persistent storage that can fulfill the requirements of both the community management mechanisms and a variety of services that may be deployed in the community.

The current prototype has two distributed storage implementations. One is based on PAST [60], Pastry's DHT which is included in the FreePastry distribution. It provides the typical DHT interface that allows to put and get values associated to a key. It can be used to store data objects persistently, and provides availability using replication. However, it can only be used to store Java objects of a limited size, providing no specific support for storing files. Therefore, this implementation does not fit all the requirements of CoDeS, since it cannot be directly used to store, for instance, the executable files associated to a service.

The other implementation of the storage module uses a centralized server to store files, which can be accessed from any node in the community. This server is statically set and defined in the configuration of the storage module, which controls transfer of files to and from the server. While this does not fit the requirements of open communities which need to work without depending on dedicated resources, it is a simple and fast implementation that can be used to easily validate the behavior of the rest of the mechanisms in a real distributed environment.

Finally, a third implementation is provided specifically for the simulation environment. Since a simulated network has all nodes interacting inside the same JVM and accessing the same disk space, this implementation manages files stored in the local disk and data objects stored in memory. Nodes can easily access the data without any overhead. Therefore, in our simulations storage access has no cost. This cost should be quantified according to the specific storage implementation that is deployed in a community.

2.6 Conclusions

In this chapter we have presented CoDeS, a middleware for building contributory communities. It can be used as a platform to deploy services, similar to platform-as-a-service (PaaS) offerings from cloud vendors such as Google App Engine. Users can deploy a service and use the middleware as a directory, which they can query to find a list of locations of the service. The service is autonomously deployed and maintained by the middleware.

The architecture of CoDeS has been designed with modularity and extendability in mind. This allows the use of different existing mechanisms and protocols, leveraging advances made in other related fields such as grid computing, cloud computing or P2P networks. Moreover, we have built a prototype of CoDeS which can be easily extended thanks to its modularity, and can also load different components dynamically to offer a different combination of mechanisms and policies that adapt to each community.

The presented architecture, together with the mechanisms that will be presented in the next chapters of this thesis, satisfies the requirements of contributory computing presented in Section 1.2.4. One of its key characteristics is the use of structured overlay networks, which provide decentralization, scalability, failure tolerance, ability to deal with Internet-distributed resources and self-organization. Of course, the mechanisms built on the upper layers must preserve these features, but having them at the connectivity layer is a strong foundation for designing the community management mechanisms. The existence of heterogeneity is addressed by using a portable language like Java, although it must also be addressed in the management mechanisms. The last remaining requirement, service availability, is provided by the mechanisms we have developed, presented in the remainder of this thesis, while preserving the features provided by the lower layers.

2.6.1 Future Work

However, there are still a number of areas where CoDeS, and specifically our prototype, could be improved. For instance, CoDeS needs a persistent storage, which is a critical part of the system, since service specifications and executable files are stored in it. The requirements of this subsystem are not very demanding, since the stored files can be considered immutable: the data related to a service is only stored when the service is registered in the community, and it is not modified after that. A service may be modified, for example to upload a new version of the service, by a deletion and insertion of the new version. However, as has been pointed in a study by Blake and Rodrigues [29], even the bandwidth required for keeping data available through replication using non-dedicated hosts may be unreasonable.

Note, in any case, that the profile of a contributory community is not expected to closely match that of P2P networks such as Gnutella [122], which is the one used as a base in that study.

Services, however, may require a distributed persistent storage with more features than those needed by CoDeS for community management. These may include the storage of mutable data with multiple readers and writers, with high levels of consistency and availability. Providing this is not trivial, so the design of services should carefully consider the consequences of the requirements that are imposed over the storage system. In any case, our prototype could be improved by incorporating one or more implementations of a distributed persistent storage with more features to provide a wider range of options to choose when creating a contributory community and designing services to deploy in it.

Security is another aspect not addressed in our prototype of CoDeS. Some of the specific issues have been presented in Section 2.3.4. However, it would be advisable to perform a complete security analysis of CoDeS, both at the design level and at the implementation level. Security problems should be solved in the implementation of the middleware, by incorporating additional mechanisms and tools like membership control, or at least clearly identified and defined to allow users to choose whether their community model can deal with the existing security risk.

Security is also related to execution isolation and the control of the execution environment of services. However, as we have discussed earlier, this part of the system should also provide features beyond security, including a greater control of how and how many resources are contributed. This is not currently implemented in our prototype, since the simple JVM environment does not limit the access to resources available to the service. For instance, it does not control the amount of CPU or memory that the service can consume, nor forbid access to private data stored by the user. It is necessary to integrate a virtualization-based execution environment into the middleware. This could be done using some existing virtualization tools such as Xen [21] or KVM[17].

It is our intention to deal with the presented issues of the prototype in the near future. However, as it is, the prototype fulfills its purpose of serving as a proof-of-concept and allowing autonomous service deployment on a relatively controlled environment such as PlanetLab [112]. Other limitations and open issues of our current middleware design and implementation, presented in Section 2.3.4, would require further research to develop decentralized mechanisms that implement the required features.

[17] `http://www.linux-kvm.org/page/Main_Page`

3 Service Deployment

This chapter describes the mechanisms used for service deployment and management in CoDeS. These novel mechanisms use a mixture of peer-to-peer, agents and optimistic techniques to keep a service available in a self-managed and decentralized manner.

3.1 Overview

Service deployment and maintenance is the main functionality of CoDeS. Therefore, the mechanisms that allow autonomous service deployment and management are the core of the middleware. This feature is the main difference of CoDeS with previous distributed computing models and implementations, which have not used resources in a decentralized and autonomous way for service deployment. In CoDeS, these mechanisms are implemented mainly through three components: the Service Deployment Module (SDM), the Remote Execution Module (REM) and the Resource Discovery Module (RDM).

The Service Deployment Module is the component that contains the main mechanisms of service deployment. It uses the functionality provided by the RDM, to find resources in the community, and the REM, to use these resources. However, the internal implementation of these modules is transparent to the SDM, as long as they adhere to a fixed implementation. This chapter will present the internal mechanisms of the SDM and the REM. The mechanisms for resource discovery are the focus of Chapter 4, and mechanisms for availability-aware resource selection are presented in Chapter 5.

The main contribution presented in this chapter is **a mechanism for decentralized autonomous service deployment.** Following the CoDeS philosophy of modularity and extendability, it is compatible with different resource discovery mechanisms and key-based routing (KBR) substrates. Specifically, it only uses generic KBR operations, and is independent of the formats used for describing resources (both offered and required) and the mechanisms used to find resources that match a service requirements. We validate it by simulation and prove that services are kept available and accessed efficiently from a user perspective. Moreover, we prove its scalability and failure tolerance by simulating different community sizes and different levels of churn.

The rest of this chapter is organized as follows. Section 3.2 presents some systems which provide similar functionality, including the few that perform a service deployment similar to the one in CoDeS, and some of the many that perform task deployment in P2P networks. Section 3.3 presents the mechanisms that manage service deployment in CoDeS, and Section 3.4 present our evaluation of these mechanisms. Finally, Section 3.5 concludes and presents some future work related to service deployment mechanisms.

3.2 Related Work

3.2.1 Service Deployment

There are few systems which have tried to provide autonomous and decentralized service deployment. The one that is closer to our approach is Snap [71], which deploys web applications and services on a P2P network. All peers are required to have a lightweight web server, Jetty, which is used to host the web applications. Application code is stored in a DHT, and whenever a user wants to access a service or application and cannot find an available replica of it, the application code is obtained from the DHT and a new replica is started locally. Replicas are therefore created on demand and close to their users. The P2P overlay network is also used to create a multicast group among the existing replicas of a service, which is also used by clients to discover existing replicas.

Snap can deploy services which use a relational database, which is managed at a lower layer. Snap's persistent storage replicates databases and propagates state updates to all replicas, with a distributed interceptor which provides lightweight event ordering. This allows the deployment of the same three classes of services than in CoDeS, namely stateless, three-tier and distributed services. While the consistency of their storage implementation is limited, it could be integrated into CoDeS as a persistent distributed storage module. However, we do not focus on these components.

There are a few differences between Snap and CoDeS. First, Snap is limited to deployment of web applications. CoDeS, on the other hand, is designed to support more types of execution environments, not limited to web servers. Second, Snap considers that all web applications can be hosted in all nodes, therefore treating them as homogeneous. Contributory communities, on the other hand, may be formed by heterogeneous hosts. Therefore, CoDeS should deploy each service only in nodes that can host it. Finally, Snap is focused in on-demand replica creation, while CoDeS has a user-defined target number of replicas which have to be deployed. On-demand replica creation may be supported by CoDeS adding an additional mechanism that dynamically changes the number of replicas that has to be deployed.

Xenoservers[1] [121] also has the objective of deploying services in a group of computers that can scale up to millions of servers, but there are many aspects that set it apart from CoDeS and contributory communities. Resources are dedicated and provide some availability guarantees. They are virtualized using Xen [21], which was developed as a part of this project, and users can buy usage time of a given resource to deploy their services. The system helps them to find the adequate resources [134], but the final choice is taken by the users themselves, who must also consider the economical aspects of the deployment. This is a fundamental difference with our approach, as in Xenoservers users individually

[1]List of publications related to the Xenoservers project.
http://www.cl.cam.ac.uk/research/srg/netos/xeno/publications.html

decide to purchase resources to execute their services. The target of CoDeS, instead, is to use the contributed resources collectively, and deploy services autonomously.

Another interesting work by Godfrey et al. [74] studied different node selection mechanisms to be used in different applications. While they did not propose a system for service deployment, their study can apply to this case, among others. They basically studied fixed versus replacement strategies, where the former are those where a service is never migrated after initial deployment, and the latter are those where every time a node fails, the service hosted by it is migrated. They conclude that replacement strategies seem to work better, specifically random replacement. The explanation provided is that more stable hosts are selected over time when the initially selected, less stable hosts fail.

3.2.2 Task Deployment

Another related field is task deployment over non-dedicated resources. This is done in systems like Condor, where clients use a centralized component (the matchmaker) to find suitable resources and then submit their tasks to other machines. The submitter must remain connected in order to receive the results from the tasks. The same approach is taken in many so-called P2P desktop grids, although the matchmaker is usually a replicated component which can be deployed on any peer.

For instance, JNGI [142] uses the JXTA protocols and the peer groups they define to separate nodes in different working groups and different classes of components. It is similar to a hierarchical model, where monitor groups control joining nodes and assign them to groups. Task dispatchers control tasks assigned to nodes, and job submitters and workers contact task dispatchers to submit tasks, get tasks to execute, and send or receive task results.

P2PComp [129], also built on top of JXTA, adds support for parallel tasks with message passing, whereas other systems only support bag-of-tasks applications which require no communication between processes. Matchmaking, however, is done using the basic JXTA protocols where advertisements are broadcasted, which provides worse scalability since it requires direct communication among all nodes in the network.

Another P2P desktop grid [64] presents two mechanisms for job distribution: epidemic, where a node sends half of its local jobs to a random neighbor, and chord-style, where the jobs are routed to a node on the opposite side of the ring. They are also combined into a third hybrid mechanism. Nodes first insert the jobs they want to submit into their local queue, and they remain in charge of distributing their jobs and getting the results for each of them.

Another system [61], built on the Microsoft .Net technology, presents mechanisms for deploying jobs with two modes: pull, where each worker sends a task request to every client, and push, where each client asks for resource information to every worker. Clients perform the matchmaking locally and send their tasks to appropriate workers. This is similar to P2PComp in that it also is based on broadcast, which limits its scalability.

OurGrid [10] has a matchmaker per site, used by clients to find workers. When no workers are found on the local site, the matchmaker sends the query to matchmakers in other sites, which are connected through a central index. Clients and workers count the amount of resources that they have been donated by each other node, and donate an equivalent amount of resources to them. However, a failure of the submitting machine causes failure of all tasks submitted by that user.

ShareGrid [14] is based on OurGrid, but adds a few components to improve its functionality. A VPNserver allows access to sites protected by firewalls. A storage server is used to store the results of tasks when the submitter is not present. A web portal is used as an interface to submit tasks to the system. It also uses multi-core processors, not supported by OurGrid, by either using multiple user agents, or multiple virtual machines.

All of these systems have at least one of the following two issues that make them unsuitable for service deployment on a contributory community. One is the use of statically defined centralized components like a matchmaker. This is not possible in a contributory community where dedicated servers are not generally available. The second issue is the fact that tasks are monitored by the node that submitted them, and fail if this node disappears. This is not suitable for a contributory community because services must be offered for the whole community, independently of the node that initiated the deployment. It is unreasonable to expect a single node to remain connected for the whole life of a service, since this lifetime may be arbitrarily long, only limited by the community's own lifetime.

3.3 CoDeS Service Deployment Mechanism

3.3.1 Overview

The service deployment mechanism must accomplish the same requirements than the middleware itself. This includes decentralization, scalability and fault tolerance. However, achieving these three features simultaneously in a system is difficult: the combination of decentralization and scalability implies that information may be distributed among a large amount of nodes, and adding fault tolerance means that any of these nodes may fail.

Traditional algorithms which try to keep a consistent view of the system, e.g. through using locks to coordinate all state modifications, are too costly for this kind of environment, where changes may be very frequent because of entrance and departure of nodes. Moreover, they require that the system is in a consistent state in order to advance. This may prevent the processing of user queries and activities since a consistent state can only be achieved after a long enough period with no failures, which is unlikely in a large system. This is a consequence of Brewer's Theorem, which states the impossibility of offering consistency, availability and failure tolerance at the same time [73]

For this reason, we choose to provide weaker forms of consistency, as done by optimistic algorithms [124]. These mechanisms do not use locks, which reduces their cost. Instead, they allow independent modifications in the information stored by a node, without requiring consensus with other nodes storing the same information. Therefore, they are adequate for systems that can tolerate some temporal inconsistencies in trade of a lower communication cost.

The basic inconsistency that can happen in service deployment, as seen from the user side, is that a user is given a location of the service which is not operative, either because the node has left the community, or because it is no longer executing the service. Most applications can tolerate the delay caused by trying to contact a failed replica. Using a time-out, the client can detect that the service instance is not working, and repeat the query to discover a new instance. To reduce this delay, the mechanism can return a list of locations of the service, instead of just one.

The general approach of our mechanism is to assign control over a specific piece of data (for instance, the location of a set of active replicas of a service) to a single node. This piece data is replicated in other nodes, so that a recent version can be recovered in case the primary node fails. By replicating information we bound the degree of inconsistencies that can appear in the system, but by making this replication weakly consistent (in this case, allowing modification of the primary copy without guaranteeing that the modification is done in the rest of the copies) we limit the communication cost of the system. Moreover, by dividing the information into small independent pieces (for instance, information about different services and about different replicas of a service) that are assigned to different nodes, we also bound the effects that a single failure may have in the consistency of the system.

Consistency is expected, however, from the components and mechanisms that are used as building blocks for the service deployment mechanism. These building blocks are the key-based routing (KBR) overlay and the persistent storage. The former may present some temporal inconsistencies that would be tolerated by CoDeS' mechanisms, but it is required that messages are correctly and consistently routed towards the node responsible for a key with a high probability for the system to work. The persistent storage, on the other hand, is used to store information like the specification and executable files of services. The service deployment mechanism does rely on recovering correct and consistent versions of these files from the persistent storage.

Table 3.1: Summary of the main operations that form the service deployment mechanism, organized by module.

Module	Operation	Description
SDM	startService	Send StartServiceMsg through the KBR layer to the host responsible for the service id.
	onStartServiceMsg	Create SM locally and invoke deployService.
	getLocations	Send GetLocationsMsg through the KBR layer to the SM.
SM	deployService	Retrieve service specification from the persistent storage and invoke deployNReplicas to deploy the needed number of replicas.
	deployNReplicas	Get list of suitable hosts from the RDM and invoke REM's startRemoteExecution method for each host in the list, up to N.
	onReplicaStopped	Invoke deployNReplicas to deploy $required - current$ replicas.
	onGetLocationsMsg	Send list of locations to the sender of the message.
REM	startRemoteExecution	Send StartRemoteExecutionMsg to the host passed as a parameter. Schedule time-out to detect failed executions.
	onStartRemoteExecutionMsg	Tell local resources module to start execution and schedule periodical messages to the sender.
	onRETimeout	Invoke onReplicaStopped on SM.

The deployment mechanism presented in the remainder of this section uses replication with replacement [74]. Assignment of a service to a node is temporary, meaning that once a node leaves the system, it will no longer host the services it was running unless they are assigned to it again later. Alternative approaches are considered in Chapter 5.

Table 3.1 shows a summary of the operations related to service deployment and management for each of the modules of the middleware involved in the mechanism. The details of the mechanism and the operations of each module are presented in the remainder of this section.

3.3.2 Service Specification

The specification that is required by CoDeS in order to deploy a service contains three separate documents.

The first one is the deployment specification. It contains information to be used by CoDeS during the deployment, specifically during the resource selection phase. In this chapter, we will consider that it contains the number of replicas of the service that have to be created. Other resource selection policies may require different information, as discussed in Section 5.7.

The second specification document contains the service requirements. It is used in the resource discovery phase to locate hosts that can execute the service. This document should be in a format compatible with the Specification Manager deployed in the community, as discussed in Section 2.4.4.

Finally, the third specification document contains information required to start the execution of the service in a host. This should include a list of the files required for executions and instructions on how to retrieve them and how to start execution. This document is used by the local resources module, as discussed in Section 2.4.1. Different implementations of the local resources module may accept different formats for this document.

3.3.3 Service Management

A Service Manager (SM) is an autonomous agent that can be hosted by any node of the community. It is a part of the Service Deployment Module, and is in charge of managing a specific service by creating and maintaining the required number of replicas for the service. It is also in charge of answering queries for the service's location. Its computational requirements are negligible, and its communication costs are limited to sending and receiving control messages.

A SM locally stores the service's requirements specification, and uses the Resource Discovery Module (RDM) to find nodes that can execute the service, using the mechanisms explained in Chapter 4. Once found, it uses the Remote Execution Module (REM) to start a replica of the service in those nodes. Service specifications define the number of replicas required, and they could also provide additional policies to dynamically vary this number depending on factors like the load experienced by the service. However, in the design of the mechanism we will consider that the required number of replicas if static. If the SM is informed by the REM that one of the replicas has failed, it creates a new one.

The SM is hosted in the node responsible for the service id according to the KBR layer. It periodically checks that the service id is in the range covered by the node it is in. If it is not (i.e. a node with a closer id has entered the community), the SM migrates to the adequate node according to the KBR.

Figure 3.1 shows the first step of service deployment, where the initiating node sends a message to the node that must become the SM. The SM uses the RDM to find suitable nodes, and uses them to execute the service, as seen in Figure 3.2. The state of the community with the service deployed is shown in Figure 3.3, where only a SM and a replica of the service are shown for clarity.

Fault tolerance

In order to provide fault tolerance, each service has multiple SMs associated to it. Each of them independently controls a set of replicas. The number of active SMs is fixed, so the required number of

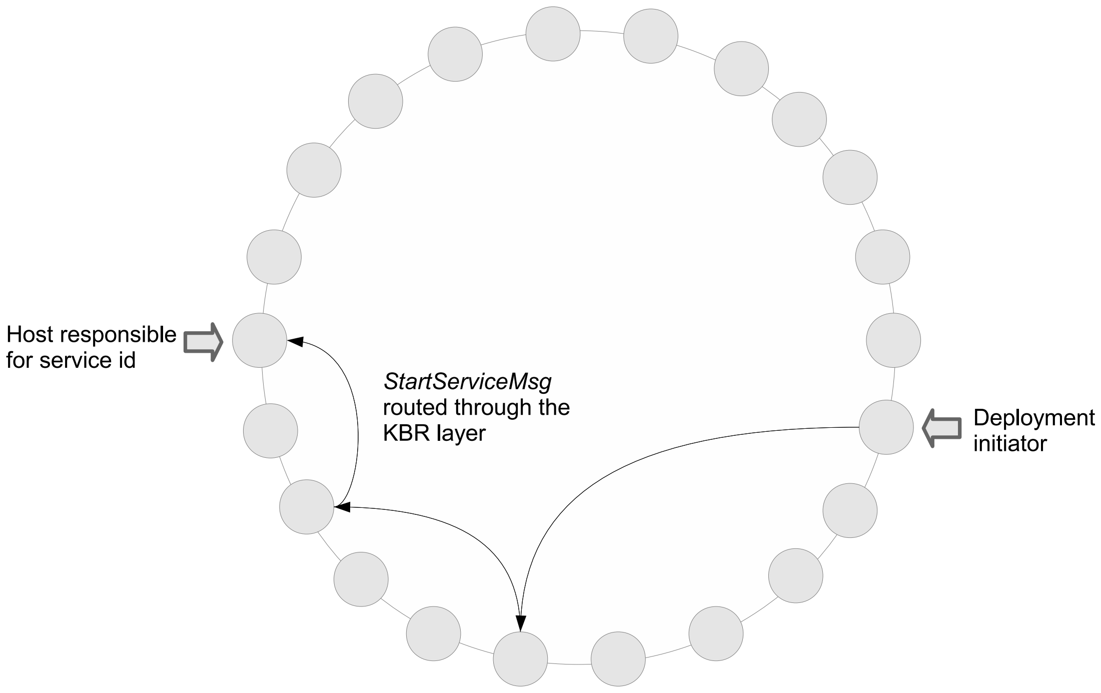

Host responsible for service id
StartServiceMsg routed through the KBR layer
Deployment initiator

Figure 3.1: First step of service deployment. The deployment initiator routes a message through the KBR layer using the service id. The node that receives it becomes the Service Manager.

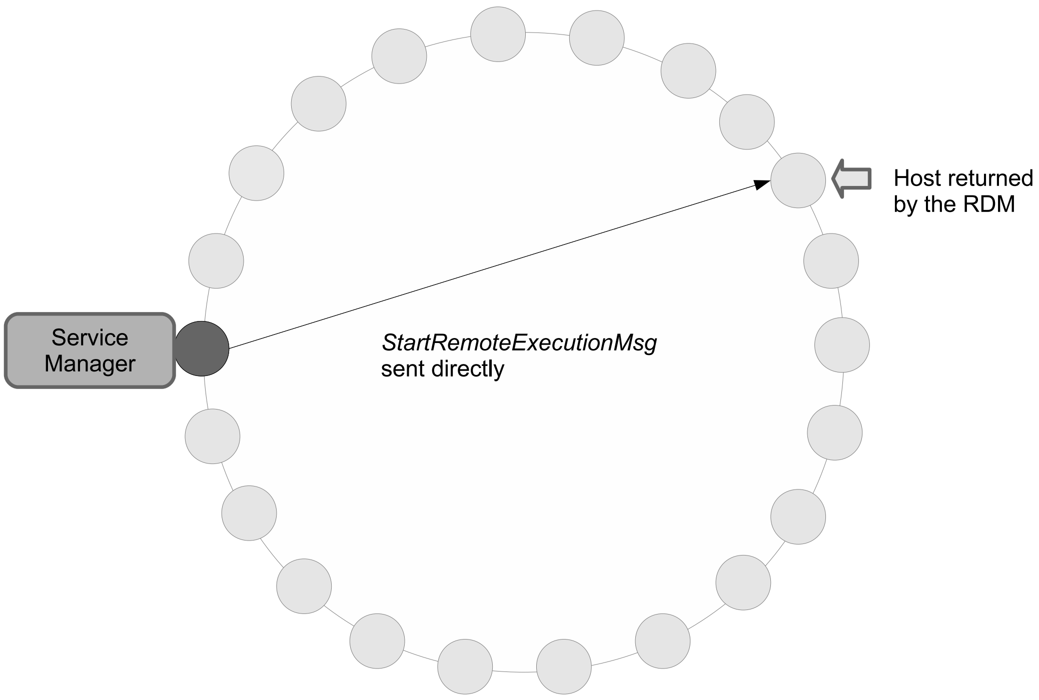


Figure 3.2: Second step of service deployment. After using the RDM to find a node that can execute the service, the Service Manager initiates a remote execution in it.

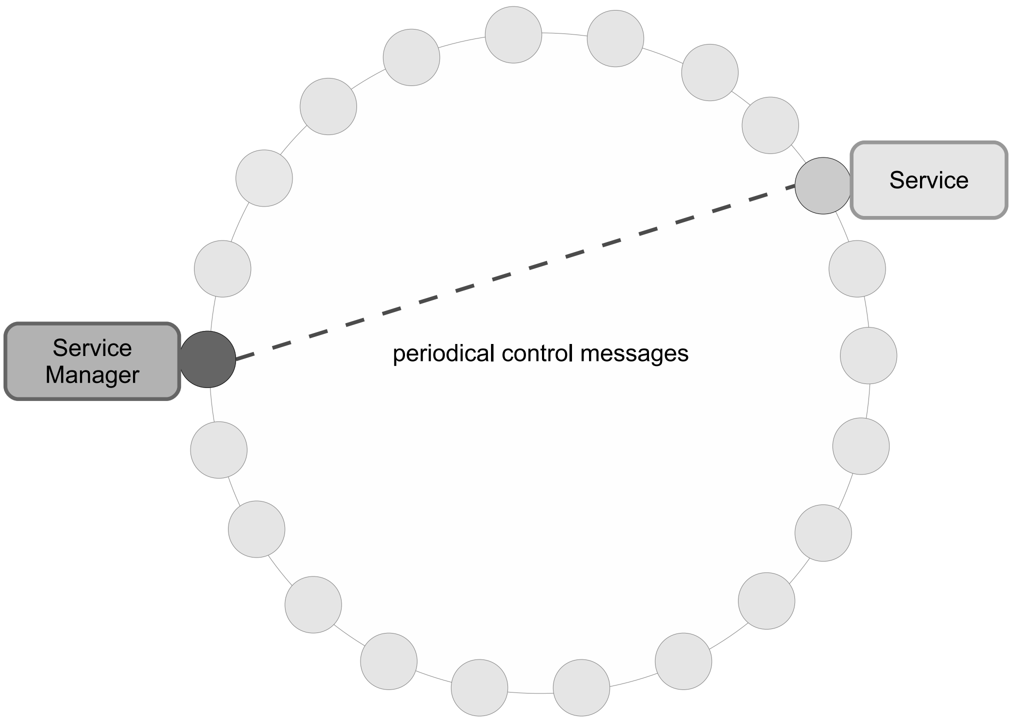


Figure 3.3: Simplified view of the state of a community with a service deployed with one replica.

replicas can be easily divided among them. A variable number of backup inactive SMs can be present in the system to enable fast recovery when a SM fails. Each SM monitors another SM by sending it a periodical message. In case one of them fails, the SM that monitored it activates one of the backup SMs or creates a new one.

In order to have several independent SM for each service, we use symmetrical replication [72]. This replication scheme bidirectionally associates each key with a set of other keys (e.g. symmetrically placed in the id space), allowing for the replicas to be accessed independently.

Backup SMs, on the other hand, are placed using neighbor replication. The SM uses the KBR layer to obtain a list of nodes which would become roots for the service id when the current root fails. This may be implemented differently by each specific KBR implementation, but it is part of the common API that all should offer [54]. The SM sends the list of current locations of the service to the nodes in the list returned by the KBR layer, turning them into backup SMs. This way, when a SM fails, the node determined by the KBR layer to host the new SM will be one of the backups, which already has enough information (the service specification, the replica locations) for the SM to instantly start working. In case the new SM does not have this information (because multiple failures have made that the node that hosts it is not one of the previous backup SM), it recovers the service specification and creates a new set of replicas. It might also be informed by the REM about some existing replicas of the service and decide if it incorporates them to its current list of replicas. Figure 3.4 shows an example of

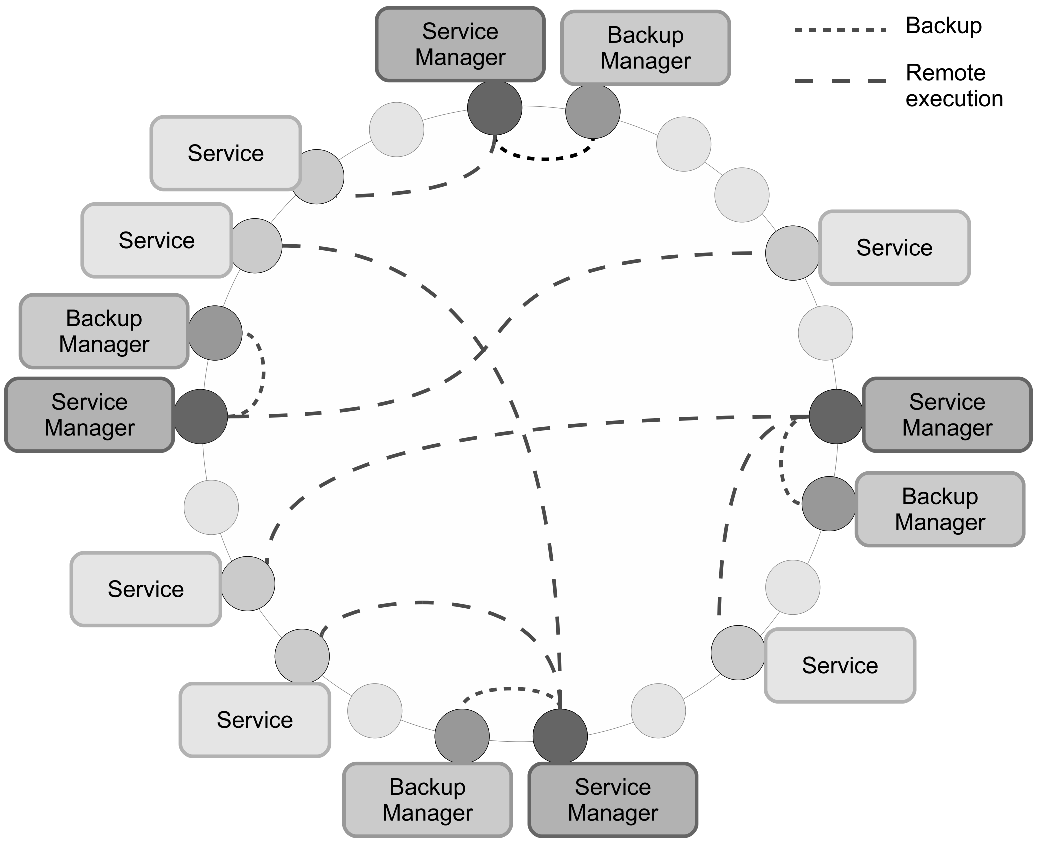


Figure 3.4: A view of a community with one service deployed, with six replicas in total, four Service Managers and one backup SM for each SM. The lines show the communication among hosts. Specifically, they show the periodical messages sent between a SM and its backup SM, and between a SM and the services it controls.

how the Service Managers and backup managers are distributed in a community.

Service location epidemic dissemination

The SMs also include in the periodical messages that they send to each other a list of all the replicas known to them. This includes the replicas managed by the SM originating the message, as well as the replicas managed by the rest of the SMs except the recipient of the message. This way, the list of replicas is propagated epidemically, providing eventual consistency: after a sufficiently long period of time when no service replicas are created or destroyed, all the SMs will have complete and correct knowledge of the total set of replicas of the service. Specifically, if SMs send this message with a periodicity T and there are M Service Managers per service, the maximum delay in the propagation of the information is $(M - 1) \times T$ This allows service users to know the complete set of replicas of a service contacting only a single Service Manager.

In case the number of service replicas is too large, it may be inconvenient to store the whole list in all SMs and send it in every periodical message. In this case, a maximum number of replicas may be fixed to limit communication and storage costs. SMs will select a random subset of their known replicas and include only this in the periodical messages. This way, each SM can know alternative replicas not managed by itself, while avoiding the communication and storage cost of knowing all of them.

3.3.4 Remote Execution

The Remote Execution Module (REM) allows other components (i.e., the Service Manager and user applications) to start and control a program execution in a remote mode. The REM of the node that starts the execution, called master REM in the rest of this explanation, takes an execution specification and a node handle, which can be used to contact the node through the connectivity layer. The execution specification contains the list of all the files that are needed to execute the program, and the instructions to start the execution. The master REM then sends the execution specification to the remote node's REM, called worker REM in the rest of this explanation. The worker REM then invokes the local execution component in the fabric layer. This recovers the needed files from the Persistence Module and starts the execution.

In order to keep the execution active and detect possible failures, there is continuous communication between the master REM and the worker REM. Specifically, the worker REM periodically sends a keep-alive message to the master REM so that it knows that the execution is still working. If the master REM stops receiving these messages, it will assume that the worker REM has failed and is no longer executing the program, and will inform the component that ordered the execution (in the case of service deployment, the SM).

In order to reduce the number of replicas created in case of failure of a SM by the new one, a worker REM periodically sends some of its keep-alive messages through the KBR layer when executing a service, so that it will arrive to whatever node that hosts the SM. When a REM receives a keep-alive of an execution it wasn't controlling, it becomes the master REM for that execution. It checks whether the associated SM is hosted locally, and if it is, the master REM informs it of the execution. If the SM is not hosted but should be, according to the service id and the KBR layer, it is created.

If the SM accepts the execution, the master REM informs the worker REM that it is the new master, so that it will send the direct keep-alive messages correctly. If the SM does not accept the execution, or it is not hosted on the node, the master REM will tell the worker REM to stop the execution. This way, when a SM fails, eventually all service replicas controlled by it will be either managed by the new SM that substitutes it, or stopped. The security concerns of accepting orphan replicas are discussed in Section 3.3.6.

3.3.5 Service Deployment and Access

In order to deploy a service, an application uses the API provided by CoDeS to store all the files required for execution and to pass the specification of the service. This specification is stored using the distributed persistent storage using the service id, so that SMs can recover it. This specification also stores if the service is active or inactive. When a service is activated, the Service Deployment Module (SDM) that initiates this activation, after setting the specification as active in the persistent storage, sends a message, using the KBR, to the nodes that must host the Service Managers. There, each SM is created and starts managing the service, deploying the needed replicas. When a service is deactivated, its specification is set accordingly in the persistent storage, and a message is sent to the SMs. They stop all the replicas of the service and then destroy themselves.

The operation of the CoDeS API for finding a service makes the SDM send a query to one of the SMs using the id provided by the user. If a SM is found, it sends back a list of locations, which the SDM returns to the calling application. If the node contacted does not host a SM, it retrieves the service's specification from the distributed persistent storage. If it is set as active, a SM is created. Otherwise, an error message is returned to the application. This way, even if all managers of an active service and all its replicas fail, it will be restored whenever a user application tries to reach it.

Two operations are provided by the CoDeS API for service location, one which allows finding a subset of the service replicas, and one that returns the complete list of replicas. The former only requests that a SM sends the list of replicas that it manages locally. This list is composed of replicas that are directly monitored by the SM and have a high probability of being working and available at the moment. The complete list is obtained through the epidemic dissemination mechanism presented in Section 3.3.3. The contacted SM returns both the list of locally managed replicas, and the list obtained from the neighbor SM. The user must note that the reliability of the information obtained this way is lower, since the list may not be updated and some of the replicas may have failed.

3.3.6 Security

As stated in Section 2.3.4, our approach towards security in service deployment will be based on two features: verifiability and local decision. Verifiability is provided by the connectivity layer. Since the placement of Service Managers is decided by the KBR layer, it can be used to verify whether a node that claims to host a SM is legitimately assigned to control the service. The condition is that the KBR implementation can guarantee consistency in presence of malicious nodes, which may try to influence the routing mechanisms.

There is a point, however, in the presented mechanism where verifiability is not enforced. This is the case where a SM fails and another node must take its place. The new SM may receive messages from malicious nodes claiming to host replicas of the service. The only possible way of verifying the legitimacy of these orphan replicas is, in the case where the new SM was previously a backup SM, to only accept replicas that were in the replica list of the previous SM. If the new SM was not a backup SM, it cannot do this verification. Moreover, it is possible that some legitimate replicas are not in the list of the new SM because the previous one failed before transmitting the updated list to the backup SMs. However, this provides more security to the system by not allowing users to autonomously create replicas and pass them as legitimate. The drawback is the cost of potentially having to create more replicas to replace the possibly legitimate ones that cannot be verified.

The other feature is based on allowing local decision at different steps of the process. This is compatible with reputation mechanisms, where a user can decide whether to trust a specific node. Local decisions can be informed by reputation mechanisms to allow users access services deployed on hosts they trust. Alternatively, they can rely on randomness to increase the probability of finding legitimate nodes, instead of being forced to use a single node that may be malicious.

A legitimate SM will get a list of potential nodes from the RDM. While the security of this step will depend on the specific resource discovery mechanism, presented in Chapter 4, the SM can ask for a longer list of resources. If the resource discovery mechanism is not controlled by malicious nodes, and therefore does not return a list formed exclusively by malicious nodes, the SM has the opportunity to select trusted hosts, if a reputation mechanism is present in the community, or to rely on randomness.

Malicious nodes hosting a SM may create all the replicas of a service under their control on malicious nodes. However, since each SM acts independently, and SM location is verifiable by the KBR, it would be difficult for a malicious host to enforce the deployment of a service exclusively on malicious nodes unless it gains control over the KBR layer. When a client asks for locations of a service, it can decide if the SM is trustworthy, and also evaluate the reputation of each of the replicas of the service. Therefore, the user can exert control over which nodes are trusted.

3.4 Evaluation

3.4.1 Test Environment and Evaluation Criteria

In order to test the service deployment mechanism, we have used the CoDeS prototype presented in Section 2.5. We have used FreePastry's simulation mode to create a whole community in a single JVM and deploy services in it. We have implemented the mechanisms presented in this chapter inside the SDM and REM, and have tested their behavior in a variety of conditions.

Because of the variety of choices presented in Section 2.5, we have tried to obtain measures of the performance of the service deployment mechanism independently of the other mechanisms and modules that exist in the system. For this reason, we have not measured the cost related to storage, neither for keeping data available nor for accessing the data. All nodes in our tests are able to access all data locally. This is trivial to implement in the simulation because all nodes run in the same JVM.

For resource discovery, we have simulated a centralized resource discovery mechanism. Hosts periodically send an ad to a well-known matchmaker, and SMs contact that same matchmaker to get a list of suitable hosts. The matchmaker, therefore, knows all the hosts in the community, and locating it requires no additional communication. The only cost of resource discovery is one periodical message per node, and two messages (query and answer) for finding resources. The actual resource discovery mechanism we have developed for CoDeS is presented in Chapter 4, together with a comparison with this centralized scheme.

The main aspects we have considered for our tests are community size, churn and load. The latter has been modeled as the number of services deployed in a community, in proportion to the number of nodes that form it. The services deployed are dummy services which require no file transfer, state maintenance, coordination or initialization time. Therefore, our communication measurements do not include file transfers, as mentioned earlier, nor application-level messages between service replicas.

For simplicity, in these tests we have considered that resources have an unlimited capacity, i.e., each node can execute any number of services. However, we have given services a resource selectivity of 0.1. This means that each service can be executed, in average, in a 10% of the nodes of the community. This value is high enough to guarantee that there are enough hosts which can execute each service. Moreover, note that these tests are not affected by possible effects of selectivity in resource discovery, as they use a centralized resource discovery mechanism. In summary, our tests model a stable system, with light stateless services, where the amount of resources provided is enough for the services that are deployed.

We have modeled churn synthetically using a triangular distribution to determine the rate of entrance and departure of nodes. We have given them both the same values, to keep the total number of nodes stable. We use a triangular distribution in order to test different levels of churn, as it allows us to easily modify the range of values used for testing purposes. The default mode inter-arrival/departure time is 7 seconds, with the limits of the triangular distribution being always 100 times more and less the mode. We also evaluate our approach using models based on actual trace data in Section 3.4.3.

We have used three different configurations of the system, which differ in the number of messages they send. This affects mainly the periodicity of messages sent among nodes. We have named the

Table 3.2: System parameter values for the three configurations of CoDeS used in the tests.

Parameters	**Eager**	**Medium**	**Lazy**
Number of SMs per service		4	
Number of backup SMs per SM		1	
Periodicity of Service Manager local checkings (check if it is in the correct host and send info to backup), in seconds		1	
Periodicity of Service Manager remote checkings (check if other SM is active), in seconds	20	120	600
Periodicity of remote execution checks, in seconds	20	60	300
Remote execution TTL, in minutes	1	3	15
Periodicity of remote execution checks sent through the key-based routing (instead of direct message to the known Service Manager), in minutes	5	15	30
Ad periodicity, in seconds	20	60	180
Time before retrying RD query, in seconds		1.5	
Service query timeout, in seconds		50	

configurations eager, medium and lazy. Table 3.2 presents the values assigned to different system parameters in each of these configurations.

We want to evaluate the perception of service availability for the user and the load of the system. To do this, we have run a simulation for each configuration where we start a network, and after all the hosts (as many as determined by the initial network size of the configuration) have been booted and the services (as many as determined by the load level of the configuration) have been deployed, we let the simulation run for 8 hours of simulation time. During this time, an application issues a query to locate a service (randomly selected from the ones deployed in the community) every 10 seconds, for a total of 2880 queries per simulation. Each query is sent from a randomly selected node. Once the application receives a response, it checks the returned locations to find if the hosted replicas are alive. If they are not, or no response is received after a fixed timeout, the query is repeated until an active replica of the service is found.

The user perception of service availability is measured as the time it takes to find a replica of a service, i.e., time that a client has to wait until it can access the service. This is the overhead that our system incurs over having the service in a well-known dedicated server. We measure this in number of messages per query and in milliseconds. Because the tests are done in a simulation environment, the millisecond values do not correspond to actual times in a real system. This is why we chose to also show number of messages, which is a more objective and independent measure. Since messages sent are sequential, they can give an idea of the time service discovery would take in a real environment. For each of these metrics, we show the average and the standard deviation of the values obtained in all the queries in each simulation, i.e., we average the results of the 2880 queries issued for each configuration.

Delays below 1 or 2 seconds are often considered tolerable for complex actions [36, 104, 108], while it is commonly regarded that actions that take longer than that require some kind of visual indication, e.g. a progress bar. Therefore, we consider times for locating a service replica below 2 seconds to be a good performance. Other associated delays may come from trying to contact a replica that is currently being initialized. In this case, the application could show a visual indication to inform the user that the service is booting, in case no working replicas are found. We must also consider that a user will only require to find a service the first time it is accessed or if the known instance fails.

The communication load of the system is measured as messages per node per minute, because of the limitations of the simulation model. FreePastry's simulation mode uses Java objects to transfer information between simulated nodes, which makes it difficult to obtain a measure of metrics like used bandwidth. We obtain this information by obtaining the number of messages received by each living node during each minute of the simulation (including those that are merely forwarded as part of the the key-based routing), and we show the average and standard deviation of the collected data. Therefore, high deviations may mean differences in the number of messages sent at different moments of the simulation and differences in the number of messages sent by each node.

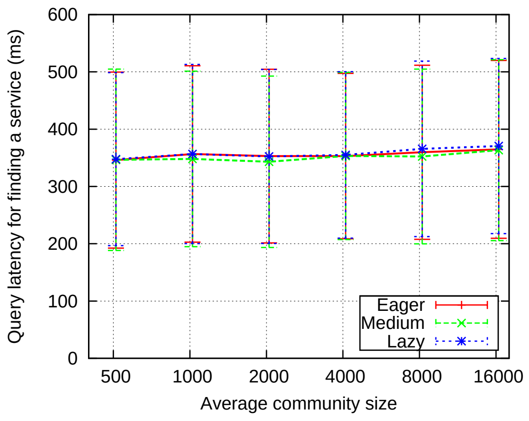
3.4.2 Scalability

CoDeS is designed to support large communities of hundreds of thousands of nodes. In order to verify the scalability of the mechanisms, we have tested from small communities of 512 hosts to large ones of more than 16,000. We do not present tests with larger communities because of the computational cost of simulating a community of tens of thousands of nodes or more. However, the results presented by preliminary tests with larger communities of more than 32000 were consistent with the ones shown here.

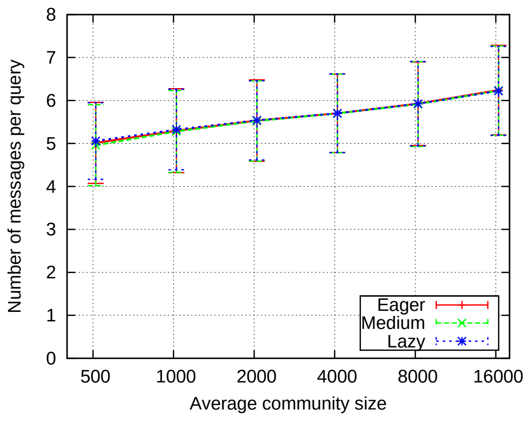
Another factor in these simulations is the load of the community in amount of services deployed. We have measured load as the ratio between the number of services deployed and the initial number of nodes in the community. For these scalability tests, we have fixed the load at 0.1. However, in these tests we have used a low value to minimize the cost of simulating large communities.

Figure 3.5(a) shows the latency of these queries, i.e. the time until the location of an active replica is received, without counting the time needed to check it. Figure 3.5(b) shows the number of messages involved in a query. Both graphs show the mean values and the standard deviation.

We see a logarithmic increase tendency in the cost per query, both in time and in number of



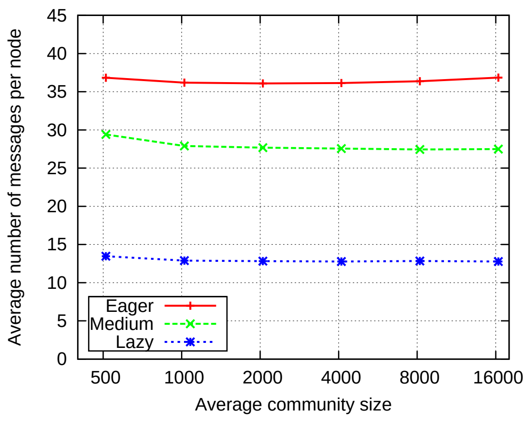
(a) Latency of service queries, for different sizes.



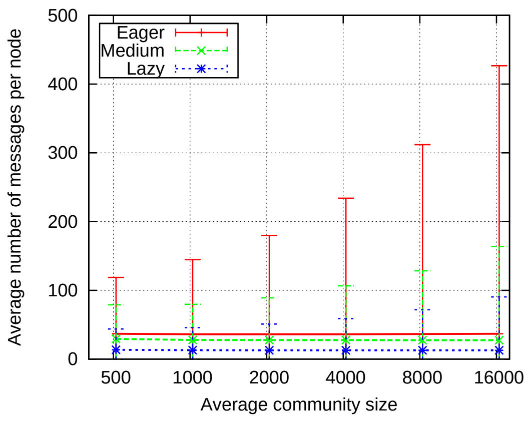
(b) Number of messages involved in service queries, for different sizes.

Figure 3.5: Cost per query for different community sizes.

messages, when the size of the community grows. This is the expected behavior as the SM lookup is done using FreePastry's KBR, which has a logarithmic cost. On the other hand, since each service is managed independently, the increase in the total number of deployed services does not cause any variation on this cost.



(a) Number of messages sent per node per minute, for different sizes. Shows only average values.

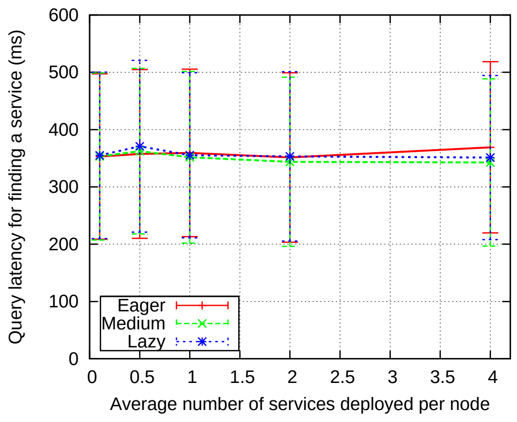


(b) Number of messages sent per node per minute, for different sizes.

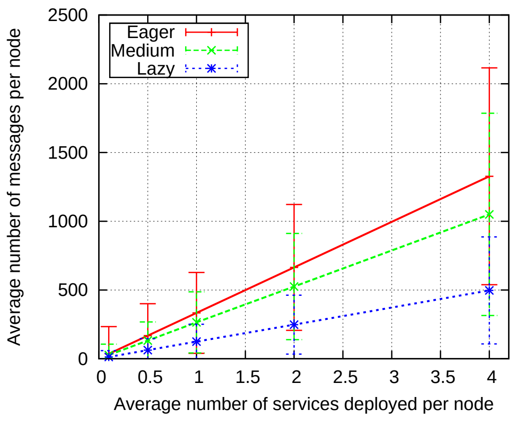
Figure 3.6: Communication overhead for different community sizes.

Figure 3.6 shows the overhead of the system, in the form of messages per node per minute. Increasing community sizes seem to cause a small decrease in the overhead, as seen in Figure 3.6(a), especially for the medium and lazy configurations, which could be explained by the fact that messages that are routed through the KBR overlay, like query messages, go through a lower fraction of nodes. However, the variation is not significant. On the other hand, Figure 3.6(b) shows that the standard deviation grows significantly with size, especially for the eager configuration. This load imbalance may be caused by non balanced distribution of Service Managers throughout the community. As the community and the number of services grow, so does the probability of a large number of SMs being deployed on the same node.

Figure 3.7(a) shows that query latency remains constant with increasing load levels. Number of messages per query, not shown here, does not vary either. However, Figure 3.7(b) shows that the number of deployed services, as a function of the number of nodes, does have a linear relation with overhead, as expected. This proves that our system scales in size, while maintaining a constant load on the system. This is consequent with the description of our target environment, where a bigger number of users brings both a greater load and a higher number of resources available. CoDeS can scale well in this conditions, maintaining a constant overhead when the size of the community and the load increase at the same rate.



(a) Latency of service queries, for different load levels.



(b) Number of messages sent per node per minute, for different load levels.

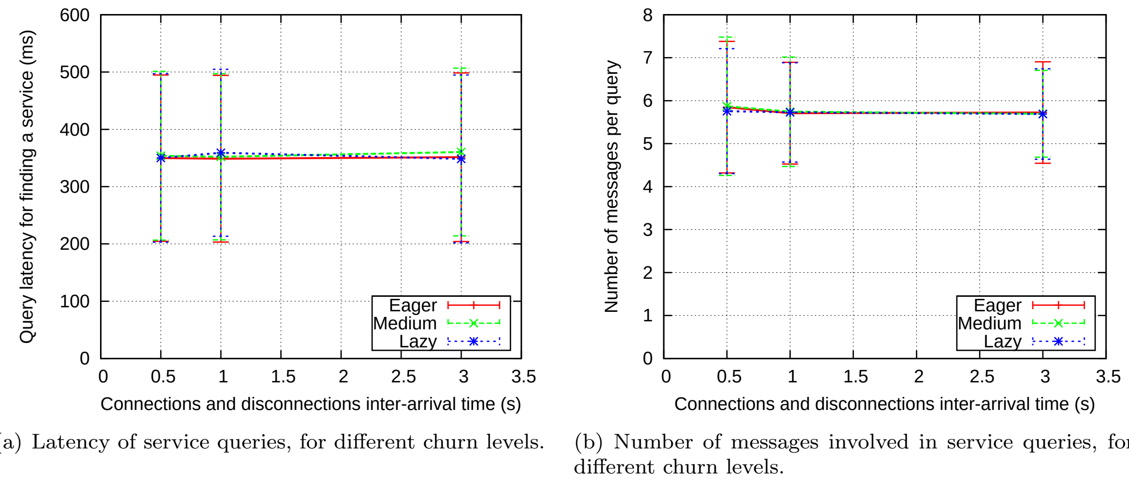
Figure 3.7: Performance of service deployment for different load levels.

3.4.3 Fault Tolerance

In order to test CoDeS' fault tolerance, we have ran simulations with different levels of synthetically generated churn. We tested a community with an initial population of 4096 nodes (a large enough size to be representative, once analyzed the results of the scalability tests), with continuous arrival and departure of nodes following a triangular distribution, varying the mode of the distribution from the higher value of 30 seconds to the lowest of 0.5 seconds. The upper and lower limits are changed accordingly to values 100 times higher and lower, respectively, as explained in Section 3.4.1. For these tests we used a value of 0.5 for the load level, i.e., ratio of services per node.

We hypothesize that service initialization time may be an important factor for fault tolerance. This includes both the time required for transferring the necessary files to the node that will execute the service and the time required to boot the service in that node. This time may vary widely among different services, and for this reason we have chosen to evaluate CoDeS' performance with a range of values for service initialization time. After a service is assigned to a node by the SM, it waits for a given time, specified in the test configuration, until it becomes active. Until that moment, although it is recognized as a working replica by the SM, it does not answer user queries, therefore appearing as a non-working service replica for the application that issued the query.

Figure 3.8 shows the query cost, in latency and number of messages per query in each configuration. In these tests, we set a service initialization delay of 10 minutes, which we consider a high enough value to see the effects that a reasonable initialization time may have on service initialization. The figures show that CoDeS can provide access to services in a short time even when services take a significant time for booting and churn level is high. We also tested shorter initialization times, but since the results are the same than when using 10 minutes, we only show the latter here. Similarly, we only show the results of higher churn levels, with modal inter-arrival values equal or lower than 3 seconds. As would be expected, lower churn levels give the same results.

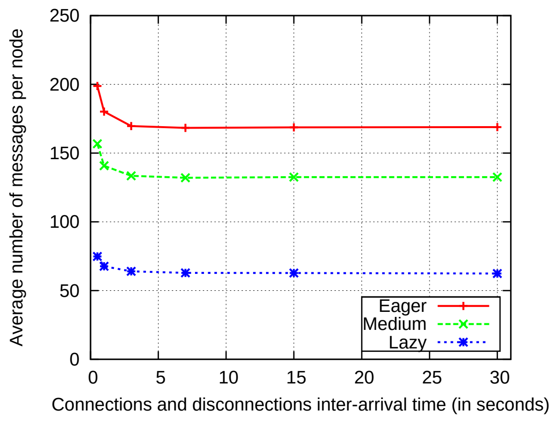


(a) Latency of service queries, for different churn levels.

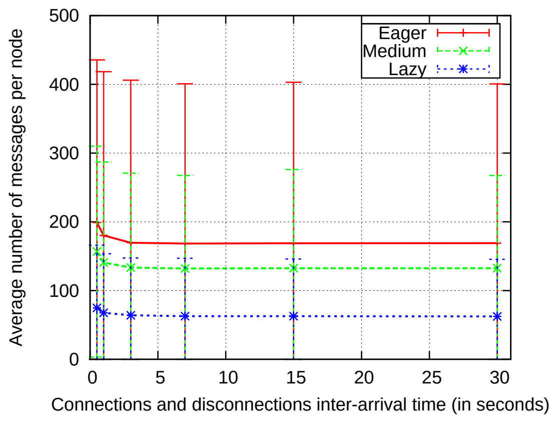
(b) Number of messages involved in service queries, for different churn levels.

Figure 3.8: Cost per query for different churn levels, expressed in the modal value of inter-arrival and inter-departure time. Service initialization time is 10 minutes, and the number of replicas per service is 4.

The only variation we find is in the overhead, as seen in Figure 3.9. The number of messages per host per minute seems to grow when churn is very high. Since these tests used synthetically generated churn to see the performance of CoDeS under a wide range of circumstances, we now need to know where in this range would an actual community find itself. Therefore, we took some distributions based on analysis of real systems and repeated the simulations.



(a) Number of messages sent per node per minute, for different churn levels. Shows only average values.



(b) Number of messages sent per node per minute, for different churn levels.

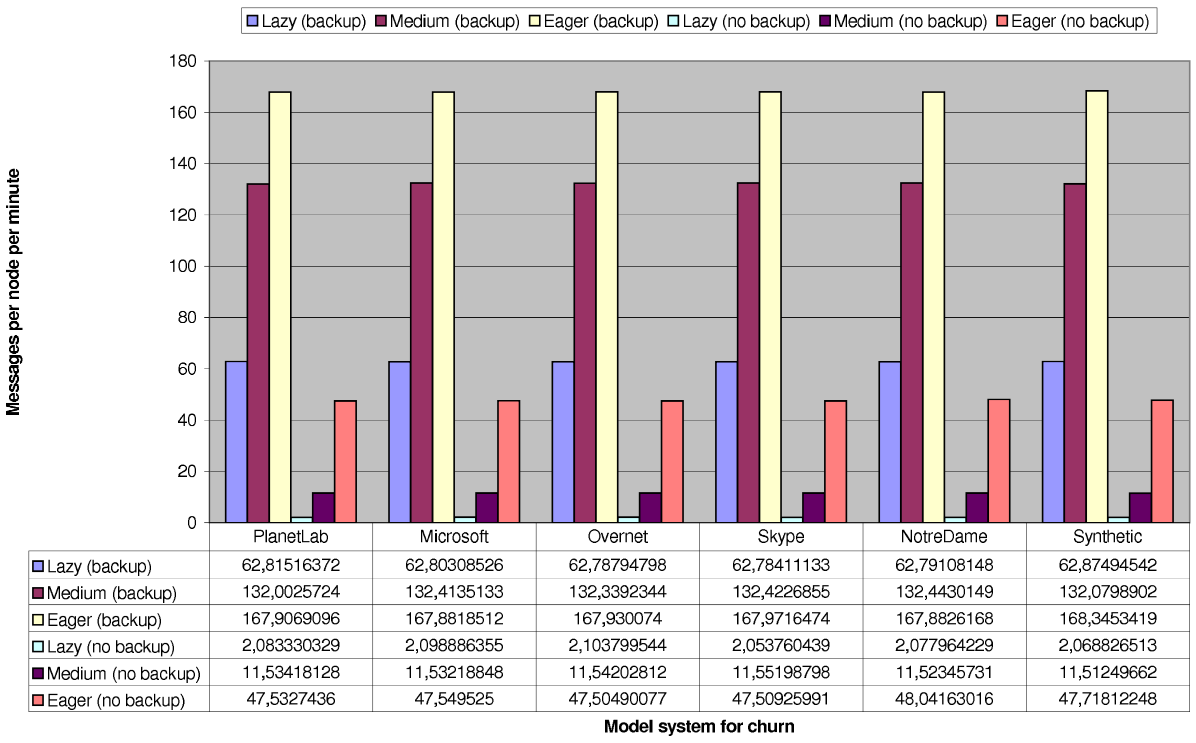
Figure 3.9: Communication overhead for different churn levels, expressed in the modal value of inter-arrival and inter-departure time.

These distributions are based on five data sets publicly available in a repository of availability traces called Failure Trace Archive[2] [88]. We simulate five different systems. The traces we used include: log files of 51,663 desktops PCs at Microsoft Corporation with a ping every hour, for 35 days (Microsoft); trace data of all PlanetLab nodes using pings every 15 minutes (PlanetLab), with a total of 692 over a period of one year and a half; traces of the desktop systems at the University of Notre Dame, with 700 hosts over 6 months (ND); pings to 2000 nodes in the Skype superpeer network every 30 minutes for 1 month (Skype); and a probe-based masurment of 3000 Overnet nodes for 2 weeks (Overnet).

We simulated these systems using the models for availability and unavailability intervals length described by Kondo et al. [88]. The most accurate approximation of the analyzed traces, according to Kondo et al., is with a Log-Normal distribution. For each test we created a network formed by a total 4096 nodes, which entered and left the network following the corresponding model. The overhead of the system, as shown in Figure 3.10, is equivalent to the one on the simulations with synthetic churn, with the medium churn values. Specifically, the figure shows the results with inter-arrival times of 7 seconds, although the results are similar with all modal values of 3 seconds or more. The latency and message cost of queries, not shown here, is also the same than with the synthetic churn.

Once we have proved CoDeS fault tolerance, the next question is whether we can reduce the communication cost of the system while maintaining the same availability level. Since the lazy configuration, the one with the lower costs of the three we have tested, provides the same availability than the more costly ones, we believe that there is still space for cost reduction in our mechanism. For this reason, we

[2]`http://fta.inria.fr`



	PlanetLab	Microsoft	Overnet	Skype	NotreDame	Synthetic
Lazy (backup)	62,81516372	62,80308526	62,78794798	62,78411133	62,79108148	62,87494542
Medium (backup)	132,0025724	132,4135133	132,3392344	132,4226855	132,4430149	132,0798902
Eager (backup)	167,9069096	167,8818512	167,930074	167,9716474	167,8826168	168,3453419
Lazy (no backup)	2,083330329	2,098886355	2,103799544	2,053760439	2,077964229	2,068826513
Medium (no backup)	11,53418128	11,53218848	11,54202812	11,55198798	11,52345731	11,51249662
Eager (no backup)	47,5327436	47,549525	47,50490077	47,50925991	48,04163016	47,71812248

Figure 3.10: Number of messages per node per minute, for different churn levels modeled after five real systems. Columns show the results for each of the three configurations (eager, medium and lazy) with and without using a backup SM per active SM.

have also tested CoDeS without using the backup SMs explained in Section 3.3.3. This has provided great improvements in the communication cost of the system, which can be seen also in Figure 3.10, with no variation whatsoever in the level of availability achieved.

We measured availability, as before, in latency and number of messages per query, as a user-centric measure of availability, obtaining the same results shown in Figure 3.8. This proves that the use of backup SMs does not have any significant impact on service availability, while it does increase the communication cost of the mechanism. The reason may be that while the use of backup SMs causes an improvement in recovery time when a SM fails, this improvement is not significant considering the query rate in these tests. However, the use of backup SMs may also prevent the system from creating unnecessary replicas in some cases. Specifically, when a SM fails leaving orphan replicas, this are unknown to the new SM at first if it was not a backup SM previously, and then it may create new replicas, causing the community to have more replicas of the service than required. The actual costs of this case should be evaluated in a real system where deployment costs can be precisely measured. This could reveal some of the advantages of using backup SMs, although it is clear that the information to backup SMs should be sent with a much lower frequency.

3.5 Conclusions

In this chapter we have presented a novel service deployment mechanism. One of its key features is self-organization: unlike existing distributed task deployment mechanisms for non-dedicated resources, it does not require neither a centralized dedicated server to manage the deployments, nor individual management from the deployment initiator. It is completely decentralized, needing no specific node to work. It is also compatible with resource discovery and selection mechanisms, which allows users to place restrictions over which resources are suitable to execute each service.

Our simulation tests have proved that CoDeS maintains services available from a user perspective. This means that the user can access the service in a short time (less than one second in average in our simulations). We have proved that this is the case for large communities and even under continuous churn. However, our simulation environment has some limitations that set it apart from a real deployment environment. For instance, we do not have actual measures of bandwidth consumed by our mechanisms (we can only estimate using our measurements of messages per node per minute). Therefore, it is our intention to deploy CoDeS in a real testbed in the short future. Specifically, we intend to deploy it in PlanetLab [112], a testbed for computer networking and distributed systems research composed of more than 1000 nodes in more than 500 sites worldwide.

One of the limitations of these tests is that we have used a centralized resource discovery mechanism. However, in Chapter 4 we present a resource discovery mechanism for CoDeS, and compare its performance with the centralized approach used here. We also evaluate how the use of one mechanism or another affects the performance of service deployment. While the complete results are shown in Chapter 4, we can say that the results presented here extend without noticeable changes to a community using the distributed resource discovery mechanism.

Another important factor of the performance of CoDeS that has not been considered in these simulations is the storage subsystem. However, because of the wide range of possible solutions that have been already proposed by the research community, it is difficult to evaluate the impact of a real distributed storage. Moreover, the associated costs will be highly dependent on the specific services that are deployed. While deployment of lightweight services may be managed by a simple storage system, services with large executable or data files may impose a considerable load on the storage system and require optimized solutions. We provide more data for evaluating this problem in Chapter 5, specifically for measuring the cost that may be imposed by service migrations. However, we expect to get a more precise idea of the requirements for the persistent distributed storage module when we deploy real services in a distributed environment.

4 Resource discovery

This chapter is focused on decentralized resource discovery mechanisms. After presenting a list of requirements and possible approaches for designing these mechanisms, it follows with a review and comparison of existing proposals. It ends with a description and validation via simulation of a novel mechanism for resource discovery in contributory communities.

4.1 Overview

As has been seen in Chapter 3, resource discovery is an essential part of the service deployment mechanism in CoDeS, as it is in many other distributed systems. It allows participants (users, agents, or other components of the system) to find resources in the network that satisfy certain requirements. In the most general case, resource discovery can include finding any type of resource, from searching files shared by members of a group to services located in specific computers, among others.

In the context of CoDeS, when we use the word *resource* we refer to a set of computational resources that can be used to execute a piece of code, whether it is a short computational job or a permanent service. In short, a resource would be whether a physical machine, or a virtual machine. Other uses for resources, like storage or provision of specific services, might also be supported. An important factor is that they are in a fixed location, so the system cannot autonomously distribute and replicate them as would be done with files and other examples of more general definitions of resources.

By resource discovery we specifically mean finding a computer, whose capacities and characteristics are described in a specification published by its owner, that satisfies a given set of requirements. Once these resource has been found, it can be used for various purposes, like executing a job or, in the specific case of CoDeS, running a service. Because of the characteristics of contributory computing, resource discovery in CoDeS must not depend on dedicated resources, and therefore centralized or hierarchical systems which depend on one or a set of fixed servers are not suitable.

There are also other kinds of distributed computing systems which require more decentralized resource discovery mechanisms, like peer-to-peer (p2p) systems, ad-hoc grids, utility computing systems, etc. In many cases, including contributory computing, the same resources that are available for execution of tasks must also manage the discovery mechanisms without the support of dedicated servers that would allow a centralized approach. In other situations, the size of the system is such that a great number of dedicated servers are needed to efficiently perform resource discovery. Therefore, in all these cases a great number of nodes must be coordinated to provide the capacity of finding resources in a large scale network.

When designing a resource discovery system, it is important to determine the kind of environment in which it is intended to work, as well as the type and quality of service that will be acceptable from it. This will determine the specific requirements that it must accomplish, both functional and non-functional. Some systems require highly scalable mechanisms. Some require low response times, while others put an emphasis on completeness and accuracy of returned results, or in minimizing network traffic overhead.

Depending on its specific requirements, there are a number of design decisions that can achieve their fulfillment and build an adequate resource discovery mechanism for the targeted environment. In this chapter we identify the main requirements that resource discovery can have, and how they apply to CoDeS. We also propose a classification of the different design decisions according to some common criteria.

In order to decide how to implement resource discovery in CoDeS, we review resource discovery mechanisms used in a number of systems aimed at similar environments, that is, decentralized dynamic systems for remote execution of jobs and/or services. We analyze the requirements fulfilled by each of these systems and classify them. We also compare the performance of some of the systems using the data provided by the authors.

After analyzing the existing systems, we design a novel mechanism for resource discovery in CoDeS which leverages the service deployment functionality of the middleware to fulfill all the requirements of the system. We evaluate its performance by simulation, and also test how it affects service deployment. The main contributions of this chapter are as follows:

- **Analysis of the requirements of resource discovery for decentralized systems.** We create a taxonomy to classify the requirements in resource description and the possible resource discovery query types. We also present some other functional and non-functional requirements for decentralized resource discovery mechanisms, discuss how all of them apply to different systems, and state which are the specific requirements of CoDeS in this regard.

- **Analysis and comparison of the architecture and behavior of existing resource discovery mechanisms.** We first present a general characterization of different design decisions and how they relate to the system's requirements. We then review a number of existing resource discovery mechanisms, analyzing the requirements they fulfill and the approach taken in each of the presented design decisions. Finally, we compare the systems to find out which design approaches are useful for fulfilling each of the requirements.

- **A mechanism for decentralized resource discovery in contributory communities.** It is intended to satisfy the specific requirements of CoDeS, and makes use of the service deployment functionality of CoDeS. We validate it by simulation and prove its scalability and failure tolerance by simulating different community sizes and different levels of churn.

The rest of this chapter is organized as follows. Section 4.2 presents a classification of the requirements that different distributed systems present for resource discovery mechanisms, and defines how each of these requirements apply to CoDeS. Section 4.3 discusses some of the most important design decisions that define existing resource discovery mechanisms. Section 4.4 compares a number of existing resource discovery mechanisms for decentralized systems, discussing their design and the requirements they fulfill. Section 4.5 presents the resource discovery mechanism that we have incorporated into CoDeS. Section 4.6 presents an evaluation of the performance of the resource discovery mechanism, and Section 4.7 concludes by summarizing the findings of this chapter and discussing some future work.

4.2 Requirements

In this section we present the main requirements that are addressed, in different degrees, by decentralized resource discovery systems, and how they apply to the specific case of contributory communities and our middleware, CoDeS. Some of these are also considered as p2p discovery properties in a previous survey by Suryanarayana and Taylor [137]. Ranjan et al. [119] also present a number of taxonomies classifying different aspects of p2p resource discovery, some of which coincide partially with our own work. However, our classification does not exactly adhere to neither of these previously proposed models, but tries instead to take the most relevant parts of them augmenting them where needed.

4.2.1 Search Flexibility

In a resource discovery service, users need to express what resources they need in a way that is convenient for them and for the activities they want to perform with the available resources. For example, in a distributed storage system, the requirements for resources could include only physical storage capacity and available network connection. However, the system could also require proximity between the user and the storage node. In a job execution system, users may need to specify other characteristics like CPU capacity. Moreover, more complex tasks composed of several distributed, interconnected processes might require a set of nodes with specific inter-node conditions, like proximity.

By search flexibility, we refer to the ability of a resource discovery system to search resources satisfying different kinds of criteria. Some examples are searching for specific values on multiple attributes, values into a given range for one or more attributes or checking inter-node relations. However, search flexibility requires both being able to handle complex queries and simple ones.

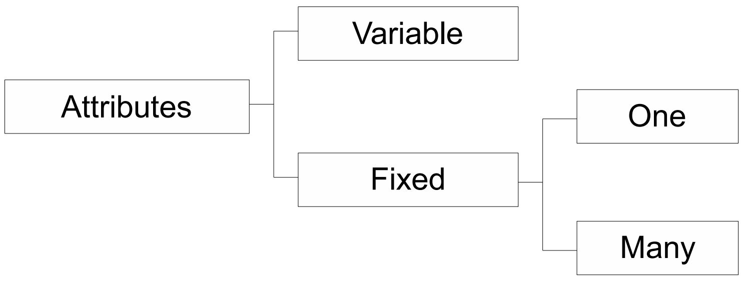


Figure 4.1: Resource description taxonomy.

The flexibility is defined by two factors: how the resources are described, and what kind of searches can be performed over the resources. We present a taxonomy for each of these factors. A different taxonomy summarizing both factors was previously presented by Ranjan et al. [119], but it does not include some aspects that, in our opinion, have a great importance,.

Resources are described by a series of attributes and their values. These attributes can be of different types: numerical, boolean, textual, etc. Each system allows different degrees of flexibility for describing resources. We present a taxonomy for flexibility in resource description in in Figure 4.1. The most flexible systems allow for a variable amount of attributes to define each resource, while others only can deal with a fixed schema. The former can allow resources to freely publish the presence of attributes other resources may not have, like rare devices, as well as allow them to provide only a partial specification, omitting the values for some attributes. This kind of resource description is not considered in the taxonomy presented by Ranjan et al. [119]. The latter approach, on the other hand, requires a fixed set of attributes, all of which must have values specified by resources. This is usually necessary to index the information for its better processing. Some systems are only able to deal with one attribute, while others can process any number of attributes, provided that it is fixed statically.

This is the most relevant division according to the resource description, since different types of attributes can be converted to other types, usually numerical, to be treated by the resource discovery system. However, it must be noted that different types might be more adequate to certain search mechanisms, or their presence might skew value distributions.

In Figure 4.2 we present a taxonomy of different types of queries supported by resource discovery systems. The exact match queries are the most limited, since they can only return resources that match exactly the value introduced by the user. This is usually of little use in resource discovery for computational purposes, since applications often have a set of minimum requirements that can be exceeded with no problems. However, this approach is far easier to implement than the others. Inequality operators can be useful for the aforementioned case where an application specifies a minimum

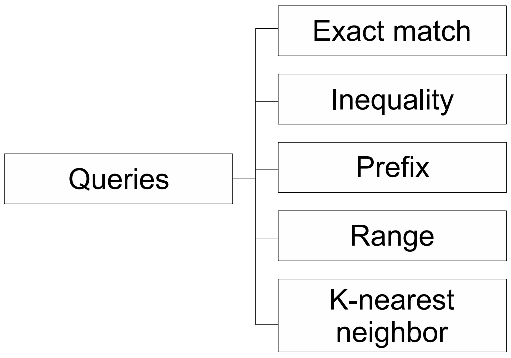


Figure 4.2: Taxonomy of types of resource discovery queries.

set of requirements, with no explicit maximum limit, and can be easier to implement than range queries. Prefix queries allow to look for values, expressed in a string (usually of bits) form, that have a given prefix. This is a case where the presence of different attribute types can make this approach more or less adequate than range queries, where resources with attribute values inside a given range are returned. Other types of queries can be processed by the system, like k-nearest neighbor (finding the k nodes whose value is nearer to a given one).

Finally, our taxonomies do not include the possibility of inter-node requirements. These are complex to implement, and only one of the systems reviewed in Section 4.4 includes them. Even in this case, they are evaluated after receiving the data from the distributed index by processing the possible combinations among the returned resources. Therefore, inter-node requirements do not appear to have an influence on the design and implementation of distributed indexes for resource discovery systems.

CoDeS must offer the maximum degree of flexibility for resource specification, since resources are managed by individual users who should not be forced to create a complex and complete specification of their resources. This means that it must at least tolerate missing values for some attributes, but ideally should also support different or exclusive attributes for nodes with rare characteristics. There is a considerable amount of existing resource specification languages, some of which already support this, like Condor's ClassAds [117]. Therefore, CoDeS should be flexible to the point of allowing different languages to be used for service specification without requiring modifications to the index.

CoDeS must also be flexible in the type of queries it supports. Equality, inequality and range queries should all be supported, as possibly other more complex queries involving logical expressions. However, other types of queries like k-nearest neighbor are not required. The reason is that the system needs to find resources that can execute a service. It is not necessary to find, for instance, the node with more resources, but just one with enough resources. Moreover, some preliminary tests with the use of simple policies, like selecting the hosts with more resources, or the ones with less resources which can satisfy the requirements, did not produce clear advantages in the efficiency of resource usage. We hypothesize that an effective deployment that can provide the most efficient resource usage requires a good knowledge of the characteristics of resource and service population. Since this knowledge is not currently available, we will not focus on resource usage efficiency.

4.2.2 Scalability

Scalable mechanisms and architectures adapt to groups of large sizes, with a high number of resources and/or clients, while still keeping a good quality of service, in whatever terms it is defined. This affects aspects like response time or overhead regarding storage, network or computational load.

The required degree of scalability depends on the intended environment of the system and the conditions under which it is expected to work. A very high scalability can allow a system to be deployed in a wide variety of systems, but some mechanisms can trade scalability for better performance in small or medium size systems, so the target environment is decisive.

Very related to scalability is the term, frequently used in overlay networks, of lookup complexity. It denotes the quantity of messages, with respect to the number of nodes in the system or other size metrics, necessary to perform a search.

As stated in Chapter 1, the requirements of CoDeS include a high level of scalability. This extends to the resource discovery mechanism incorporated into the middleware, which must be able to efficiently deal with thousands of nodes.

4.2.3 Churn and Fault Tolerance

In contributory communities as well as peer-to-peer systems, ad-hoc grids and similar systems, it is usual to have a relatively high amount of churn, i.e., connections and disconnections of nodes in the system. This can be due to users turning off their machines when they are not working, taking them out of the system for personal use or failures that cause the machine to not be available to the system. Therefore, it is desirable for a resource discovery system aimed at this kind of environments to be able to deal with churn and failures. This means that the system will recover a correct state, where it is able to discover resources in the ordinary conditions of completeness and performance, within a short time and without a great overhead.

As we have done in previous chapters, we only consider fail-stop failures. Other types of failures, like byzantine failures, are not considered inside the category of fault tolerance, but they rather fit into that of security.

4.2.4 Completeness

This refers to the ability of a resource discovery mechanism to find all resources available in its system that satisfy a certain set of requirements. However, real applications might not need to discover all resources that meet their requirements, but only a set of machines that can allow them to carry out their activities. On the other hand, complete results can be used to select the best assignment of resources for each task in order to achieve an optimal global usage of the system.

A mechanism that is not complete can fail to discover a resource that is present in the system. Therefore, it might be unable to answer a query for resources issued by a user, even in the case where the user does not require complete results. Nevertheless, in many environments the requirements expressed in most queries may be satisfied by a large amount of resources. In these cases, an incomplete mechanism can often offer a high probability of finding resources while achieving better performance than a complete one.

A contributory community can be expected to be formed mostly by “general” resources, desktop PCs with average capabilities. Moreover, it is probable that most services will not require rare resources. Therefore, we consider the contributory scenario as one with *frequent* resources. Therefore, a complete answer to a query may be too large to be desirable. Since CoDeS does not require support for query types such as k-nearest neighbor either, there is no reason for it to require completeness.

4.2.5 Accuracy

Not only is it important whether a mechanism is able to find all the available resources meeting a set of requirements or not, but also whether the results returned actually meet all the specified requirements or not. This is what we call accuracy of results. If accuracy is not total, a client will need to perform additional verification of returned results in order to be sure of their adequacy to the intended use. However, when the complexity of requirements is high, it might be a more adequate solution to provide results with a lower accuracy. Processing the returned list of resources to verify whether they actually meet the requirements can yield a better performance than checking many complex conditions in a distributed mechanism.

As we have argued in Section 3.3, CoDeS can tolerate small inconsistencies during short periods of time. Therefore, a lack of accuracy in the form of slightly outdated information is acceptable. These inconsistencies could be outdated descriptions of resources that have left the community or that have changed their capacity over time, possibly because a service has been deployed on them.

4.2.6 Security

In distributed systems, and especially in those that are open, security is an extremely important concern. There are many aspects of a system where security concerns arise, including user identification and membership, access rights and resistance to Denial-of-Service attacks, among many others.

Security is not a primary concern in the resource discovery systems that are presented in Section 4.4. However, tolerance to attacks and to the presence of malicious nodes is important for discovering resources in an open environment. For example, nodes can provide false or inaccurate descriptions of their capacities, allowing them to be selected for some tasks that they will not be able to perform. Moreover, many mechanisms give some nodes control over what resources are selected for answering a query. A malicious node in that position can force a user to use malicious nodes for their tasks, giving them control over the results and data related to that task. Even if there are other mechanisms that allow the user to identify these malicious nodes, like trust or reputation systems, a malicious node might still be able to deny discovery of legitimate resources.

Our security requirements for resource discovery in CoDeS are the same stated in Section 2.3.4 for the whole middleware. While it is not the focus of this thesis, we will briefly evaluate the security of the different systems discussed in Section 4.4 and of the mechanism incorporated in CoDeS.

4.2.7 Miscellaneous Performance Requirements

Different metrics can be considered to evaluate the performance of a system, and different systems may give a different importance to each metric. For example, it is a common objective to achieve a lower cost for more frequent operations. A system where queries are constant and the characteristics of the resources do not change frequently will gladly sacrifice a higher cost in updating resource information in exchange for optimizing the performance of queries. Meanwhile, for a system where the capacity of the resources changes rapidly, it will be preferable to achieve an acceptable level of performance for both queries and updates, rather than sacrificing one for the other.

There are other common performance trade-offs in the design of resource discovery systems. Some systems may give great importance to query latency, while others may give a higher priority to the storage or communication cost of the mechanisms (i.e. information stored by nodes, periodic messages, etc). All these requirements will affect the decisions taken in different parts of the design.

A contributory community is exposed to continuous entrance and departure of nodes. This leads to resource information getting outdated, even if the set of services in a community is relatively static, i.e., new services are not deployed frequently. Therefore, CoDeS must lower the communication costs required for keeping updated resource information. However, the cost of queries should not be excessive either, because service migrations may be frequent in a community with high churn. Each of these migrations requires at least one query to the resource discovery mechanism.

4.3 Design Characteristics

We present some characteristics that must be considered when designing a resource discovery mechanism. Decisions taken about each characteristic may influence the achievement of a specific set of requirements, although the combination of all the design decisions will ultimately determine the degree of fulfillment of the requirements.

4.3.1 Overlay Topology

An overlay network [96] is a set of nodes interconnected by logical connections that work over a physical network. Each of these logical links can abstract an arbitrary number of physical links. This concept is broadly used in peer-to-peer and decentralized systems, including resource discovery systems. Overlay networks can be classified according to their architecture into various categories, which determine many of their characteristics. An explanation of these categories can be found in Section 2.2.3.

Some systems form their overlay network by using what is usually called super-nodes or super-peers. That means that a subset of the nodes are connected among them in a decentralized overlay network, while the rest only have a connection with one of the super-nodes, usually the one closer to them according to some metric (geographical proximity, semantic similarities, etc.), in a centralized server-client fashion. In these cases, when we talk about the overlay topology (specifically in our classification of existing systems in Section 4.4) we refer to the network formed by the so-called super-nodes, which form a decentralized network and, in a sense, represent the rest of the nodes.

4.3.2 Emergent Underlying Topology

Independently of whether they have a predefined structure or not, overlay networks can be formed in a way that allows characteristics of the underlying physical network, like distance among nodes, to surface and influence the formation of an emergent topology. This is usually related to trying to reduce the cost of communication, by connecting in the overlay nodes that are physically near, or nodes with similar characteristics.

There are many examples of systems which allow the underlying topology to emerge to a certain extent in its overlay network, be it structured or unstructured. Pastry [123] achieves a good locality by influencing its selection of nodes into its routing table by a proximity metric, in addition to the protocol's structural requirements. Other systems might let emerge connections related to specific application metrics, like for example connecting nodes with similar interests.

4.3.3 Local Centralization

We say that a system is locally centralized when only a subset of the nodes that compose it are part of the overlay network, serving nearby nodes and providing a local point of centralization. The use of these usually called super-nodes or super-peers is frequently related to achieving a desired scalability with respect to performance metrics such as response time. This is due to the fact that the nodes served by a super-node do not participate in the protocols of the overlay network, and therefore its size remains small and the amount of communications and look-up latency are kept low.

However, the basic problems of centralization are also present here. Super-nodes need to be sufficiently well-provisioned to perform the functions of a centralized server for their depending nodes, and although there is no single point of failure in the system, each super-node can become a single point of failure for its users. Solutions to these problems that adapt to each environment exist, however, like configuring the number and distribution of super-nodes to avoid overloading them, and have back-up super-nodes to avoid the local single point of failure.

4.3.4 Degree of Centralization

The characteristics of the specific topology of a system and the presence or absence of local centralization will determine its overall degree of centralization. Systems with the maximum degree of centralization, which would consist of a single server to provide the resource discovery to all the clients and resources, are not considered for CoDeS, as it must provide mechanisms for generalized contributory communities that only depend on non-dedicated resources. However, some of the systems presented in Section 4.4, which are considered decentralized, present some sort of centralization. Tree structures have a natural point of centralization in their root, and hierarchical networks too. The use of rendezvous points to store information also creates centralization in the system. Different degrees of centralization can be used, in a careful trade-off to achieve various levels of desired properties like scalability, fault-tolerance and load balancing.

4.3.5 Information Spread Method

Information can be spread throughout a network by a set of different methods. The most traditional classification of these methods divides them in two categories: pull and push.

Pull methods have the node that needs the information initiate the process of receiving it. It usually sends a query, that is forwarded through the network until it reaches a node that can satisfy it. Then, this node sends the required information to the initiator of the search. Therefore, to obtain information one node needs to reach it and pull it to itself.

Push methods, on the other hand, have the node that has the information send it to those who might need it. For example, a node can send information about its status periodically to the nodes that might be interested. Therefore, the node holding the information has to push it into the other nodes.

In many systems, however, both approaches are used complementing each other. It is usual in resource discovery systems to have nodes sending what is usually called advertisements, containing a specification of their capacities, which are stored by one or more nodes of the system. Nodes that look for resources of a certain type issue a query, which contain their requirements. Queries are forwarded to nodes that store advertisements for resources that can fulfill them. In this way, both pull and push methods are applied into a single information spread mechanism.

The method of information spread chosen, or the specific combination of different methods, has a great influence on aspects like the performance and overhead of the system. Many different mechanisms can be implemented, but we keep our classification at this generic level of pull and push methods.

4.4 Related Work

In order to better understand the relation between the different approaches and decisions taken in the design of a resource discovery mechanism and the requirements fulfilled, we studied more than 25 systems presented in the literature. In this section, we classify each of these systems according to the requirements they fulfill and the approaches taken in their design, compare the performance reported by each of them and discuss the relations between design and requirements that become apparent when looking at these systems. A textual description of each of the systems considered in this study can be found in Appendix A.

4.4.1 Comparison and Discussion

In this section we compare the characteristics of each of the resource discovery systems we have studied. In Table 4.1 we present the degree of achievement of each of the requirements defined in Section 4.2, for each of the presented systems. These are search flexibility, scalability, fault-tolerance and security, which are scored in a continuous scale, from 1 to 5; and completeness and accuracy, which are classified as a binary value. In general terms, the higher the number, the better achievement of the requirement by the system. Next we show an approximate description of the meaning of some values for each of the requirements, in order to help the understanding of the characteristics of each system. Note that these descriptions might not accurately represent the value assigned to some of the systems.

A qualification of 1 in search flexibility means that the system only uses a single value to classify resources, allowing for queries requesting values equal, greater or lesser than a given value. A qualification of 2 means that the system allows multiple, predefined attributes to be used to define resources, and queries can require exact values for these attributes, while 3 means that values within a given range can also be requested. A qualification of 4 means that range queries can be issued over a variable, non-predetermined number of attributes, and 5 adds the possibility to request inter-node properties, like finding nodes that are nearby.

We measure scalability from a theoretical point of view, considering the asymptotic cost of increasing scale. Experimental results regarding scalability of each system will be presented later, as shown on the respective papers. Here we consider two factors that affect scalability: the cost of information maintenance at each node, and the cost of issuing queries. We define the scalability of a system as a function of how these two costs increase with scale. A qualification of 1 means that both costs increase linearly with network size, or worse. A qualification of 2 is for systems where one of those costs grows logarithmically while the other grows linearly, and a 3 is for those where both costs grow logarithmically. A 4 is given to systems where one of the two costs grows logarithmically with network size, while the other is constant, and finally a 5 means that both costs are constant.

Regarding fault tolerance, we qualify with 1 those systems where fault-tolerance is not considered at all, no recovery mechanisms are presented, and failures or disconnections can destroy the connectivity of the overlay network with no given way of reconfiguration. In many of the systems that are given a 1, fault-tolerance is stated to be a subject of future work, although some specifically require stable servers to work. A 2 means that the system can deal with node disconnections, but not with unpredicted failures, and that the reconfiguration can have a high cost. A qualification of 3 means that the system can deal with both ordered disconnections as well as crash failures, although it requires an important reorganization that might have a high overhead. A qualification of 4 means that the system can deal with churn and crash failures with only a moderate cost, while 5 means that almost no reconfiguration is required in case of failure.

Since security analysis of each of the systems presented is not given in the literature, we only make here a basic classification based on the influence that malicious nodes could have in the working of each of the resource discovery mechanisms. Note that these classifications are based on some assumptions, previously presented when discussing each system. We classify with a 1 systems where the presence of one or a few malicious nodes could turn all the system nonfunctional. A 2 is given if a great part of the system could become nonfunctional or unavailable to the rest of the system by the action of one or a few malicious nodes. A qualification of 3 denotes a system where a small number of malicious nodes can turn a small part of the system nonfunctional, or a great part of the system unavailable for one or a set of legitimate nodes. A 4 is appropriate where malicious nodes could make a small part of the system unavailable for one or a set of nodes, while 5 means that malicious nodes cannot make any part of the system nonfunctional or unavailable to any node.

Table 4.1: Level of achievement of each requirement, scored from 1 to 5 stars for search flexibility, scalability, fault-tolerance and security, and with a yes/no for completeness and accuracy, for each system.

	Search flexibility	**Scalability**	**Fault-tolerance**	**Completeness**	**Accuracy**	**Security**
Ranjan et al. 07 [118]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	No	Yes	⋆⋆⋆⋆
Chang-yen 08 [42]	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	No	Yes	⋆⋆⋆
Xenosearch [134]	⋆⋆⋆	⋆⋆⋆	⋆	Yes	No	⋆⋆⋆
Abdullah et al. 09 [2, 3]	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	No	Yes	⋆⋆⋆⋆
SWORD (SingleQuery) [111]	⋆⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆⋆
Cheema et al. 05 [43]	⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆
Squid [126, 127]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆
DGRID [99]	⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆⋆⋆
MAAN [34]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆
Kim et al. 07 [82, 83]	⋆⋆	⋆⋆⋆	⋆⋆⋆	No	Yes	⋆⋆⋆
Costa et al. 09 [48]	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆⋆
MURK [69]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	Yes	Yes	⋆⋆⋆
NodeWiz [22]	⋆⋆⋆	⋆⋆⋆	⋆⋆	Yes	Yes	⋆⋆
NR-Tree [93]	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆
Cone [26]	⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆
Gao et al. 04 [70]	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆
Tanin et al. 07 [138]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	Yes	Yes	⋆⋆
P-tree [50]	⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆⋆
SCRAP [69]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	No	⋆⋆⋆
Mercury [28]	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆
PHT [115]	⋆	⋆⋆⋆	⋆⋆⋆⋆	Yes	Yes	⋆⋆⋆
FaSReD [132]	⋆⋆⋆	⋆⋆⋆	⋆	Yes	No	⋆⋆
Iamnitchi et al. 01 (experience-based) [78, 79]	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆⋆⋆⋆	No	Yes	⋆⋆⋆⋆⋆
Mastroianni et al. 05 [101]	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆⋆⋆	No	Yes	⋆⋆⋆⋆⋆
CCOF (Rendezvous Points) [147]	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆⋆	No	Yes	⋆⋆⋆
Moreno-Vozmediano 06 [106]	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆	No	Yes	⋆⋆⋆⋆

Table 4.2: Resource description supported and query value selectivity for each system, and theoretical lookup complexity where available.

System	Number of attributes	Query selectivity	Lookup complexity
Ranjan et al. 07	Multiple	Range	$\theta(E[T] \times (log N + f_{max} - f_{min}))$; T is the number of disjoint paths, $f_{min,max}$ are the minimum and maximum levels of division of the space
Chang-yen 08	Multiple	Range	N.A. (uses multiple Pastry networks)
XenoSearch	Multiple	Range	N.A. (uses multiple Pastry networks)
Abdullah et al. 09	Variable (auction)	Any (auction)	$O(log_{2^b} N)$; b is 4 by default
SWORD (SingleQuery)	Variable	Range	$O(m + log N)$; m is the number of peers in the selected range
Cheema et al. 05	Multiple	Exact match	$O(log\ N)$
Squid	Multiple	Prefix	$n_c \times O(log\ N)$; n_c is the total number of isolated clusters in the index space
DGRID	One	Exact match	$O(min(log\ K, log\ N))$; K is the number of resource types
MAAN	Multiple	Range	$O(log\ N + N \times s_{min})$; s_{min} is the minimum selectivity for all attributes
Kim et al. 07	Multiple	Exact match and inequality	N.A. (uses CAN)
Costa et al. 09	Multiple	Range	N.A.
MURK	Multiple	Range	N.A.
NodeWiz	Multiple	Range	N.A. (uses a binary tree)
NR-Tree	Multiple	Range, k-nearest neighbor	N.A. (uses CAN and R*-tree)
Cone	One	Inequality	$O(log\ N)$
Gao et al. 04	Variable	Range	$O(log\ R_q)$; R_q is the query's length

System	Number of attributes	Query selectivity	Lookup complexity
Tanin et al. 07	Multiple	Range	$O(log N + f_{max} - f_{min})$; $f_{min,max}$ are the minimum and maximum levels of division of the space
P-tree	One	Range	$O(m + log_d\, N)$; d is the order of the tree, m the number of results
SCRAP	Multiple	Range	N.A. ($O(log\, N)$ for a single 1-d range)
Mercury	Multiple	Range	$O(1/k \times log^2\, N)$; k is the number of long distance links
PHT	One	Prefix	$O(log\, D \times log\, N)$; D is the number of bits of ids
FaSReD	Multiple	Range	$O(log(logN) \times logN)$; close to optimal: $O(M)$, where M is the number of attributes
Iamnitchi et al. 01 (experience-based)	Variable	Range	$O(N)$ for complete results, bounded by a TTL
Mastroianni et al. 05	Variable	Range	$O(N)$ for complete results, bounded by a TTL
CCOF (Rendezvous Points)	Variable	Range	N.A.
Moreno-Vozmediano 06	Variable	Range	N.A.

Table 4.3: Summary of some of the design characteristics of each system.

System	Topology	Emergent underlying topology	Local centralization	Total centralization	Information spread method
Ranjan et al. 07	Ring DHT (Chord)	No	Yes	Medium	Pull/push
Chang-yen 08	Ring DHT (Pastry)	Yes (proximity metric)	Optional	Very low	Push, pull if it fails
XenoSearch	Ring DHT (Pastry) and aggregation trees	No	Yes	Medium	Pull/push
Abdullah et al. 09	Ring DHT (Pastry)	No	No	Medium-high	Pull/push
SWORD (Single-Query)	Ring DHT (Bamboo)	No	No	Low	Pull/push
Cheema et al. 05	Ring DHT (Pastry)	No	No	Very low	Pull/push
Squid	Ring DHT (Chord)	No	No	Very low	Pull/push
DGRID	Ring DHT (Chord)	Yes (domains)	No	Very low	Pull
MAAN	Ring DHT (Chord)	No	No	Very low	Pull/push
Kim et al. 07	Multidimensional space DHT (CAN)	No	No	Very low	Pull
Costa et al. 09	Multidimensional space	Yes (attribute values)	No	Very low	Pull
MURK	Multidimensional space	No	No	Low	Pull/push
NodeWiz	Distributed decision tree	Yes (query distribution)	No	High	Pull/push
NR-Tree	Multidimensional space and local R*-trees	Yes (semantic proximity)	Optional	Medium	Pull/push
Cone	Trie and ring DHT	Yes (attribute values)	Optional	Medium-low	Pull

System	**Topology**	**Emergent underlying topology**	**Local centralization**	**Total centralization**	**Information spread method**
Gao et al. 04	Ring DHT (Chord) and Range Search Tree (RST)	No	No	Very low	Pull/push
Tanin et al. 07	Ring DHT (Chord) and quadtree	No	No	Low	Pull/push
P-tree	Ring DHT (Chord) and partial B+ trees	Yes (attribute values)	No	Very low	Pull
SCRAP	Skip graph	No	No	Low	Pull/push
Mercury	Multiple ring networks	Yes (attribute values)	No	Very low	Pull/push
PHT	Trie over a DHT	No	No	Very low	Pull/push
FaSReD	Ring DHT (Chord) and tree (PHT)	Yes (attribute values)	No	Medium	Pull/push
Iamnitchi et al. 01 (experience-based)	Unstructured	No	Yes	Low	Pull
Mastroianni et al. 05	Unstructured	No	Yes	Medium-low	Pull
CCOF (Rendezvous Points)	Unstructured	No	No	Medium-low	Pull/push
Moreno-Vozmediano 06	Unstructured	No	No	Very low	Pull/push

Table 4.2 shows the specific behavior of each system in resource description, query value selectivity and theoretical look-up complexity, where available. The former two are classified according to the taxonomies presented earlier, while the latter is shown as asymptotic complexity. In look-up complexity, measures are given in function of the total number of nodes N. We only show them when they are presented by the authors, except in the case of Squid, where we have obtained the look-up complexity from the paper by Ranjan et al. [119].

In Table 4.3, we show a summary of the design characteristics of each of the presented systems. We specify the topology of the overlay network, mentioning the general classification as well as the specific implementation used when available. We also mention the characteristic of the underlying topology that emerges for systems where it does.

Figures 4.3 and 4.4 show the trade-offs of each system. They both place systems in the space according to their search flexibility (x-axis) and scalability (y-axis). Additionally, the size of the dot that marks the position of each system denotes its fault-tolerance, with more fault-tolerant system having a bigger representation.

Figure 4.3 shows complete systems, while Figure 4.4 presents incomplete systems. We separated them in these two categories since the understanding of their scalability must take into account the question of completeness. Incomplete systems might be highly scalable in an asymptotic measure, since the number of messages generated by a query can be limited by a constant, e.g. a TTL value assigned to the query. However, this means that the possibility of finding results is also limited by this constant. In complete systems, on the other hand, the number of messages needed to respond a query is usually dependent on the network size, and cannot be artificially limited to a constant value since these would eliminate the guarantee of obtaining complete results. Therefore, complete systems might yield a much better performance than incomplete systems that have a better asymptotic look-up complexity, since the latter might only find suitable resources when they are abundant.

Most complete systems have a logarithmic look-up complexity (figure 4.3). The most adopted mechanism is to have resources publish themselves to a certain location, and queries sent to that same location to retrieve the resources' data. DHTs' logarithmic complexity is seen as the target cost for both processes. Only four systems have a better scalability, by reducing the cost of resource publication to constant.

Complete systems exhibit a high variety of search flexibility, ranging from systems which only allow one attribute to define resources to others which allow any combination of ¡key, value¿ pairs. SWORD is the most flexible system surveyed according to its description in the literature [111]. However, the

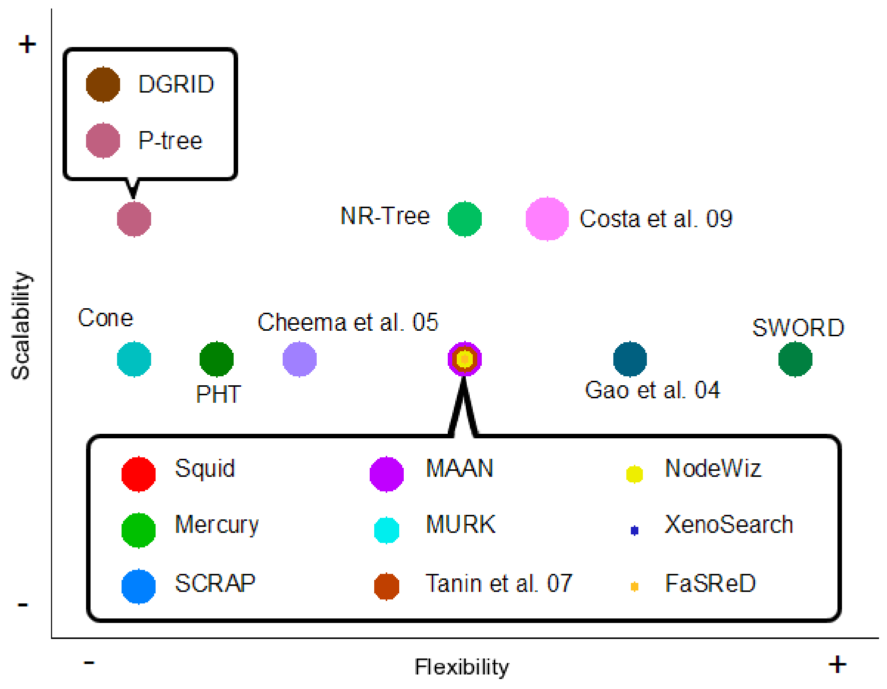


Figure 4.3: Complete systems organized for their search flexibility and scalability, with size meaning fault-tolerance. Systems with same flexibility and scalability are listed separately for clarity. Most of them have a logarithmic look-up cost.

implementation deployed on PlanetLab and available to any user[1] does not offer support for inter-node requirements, therefore reducing the search flexibility of the system in a practical environment.

In total, half of the surveyed complete systems are located at the same point. These have a logarithmic look-up and publication complexity, and support range searches for a fixed number of attributes. Moreover, most complete systems also depend on DHTs for their fault-tolerance, achieving a fairly good value, with some exceptions. The positive exception is the system by Costa et al., which uses a random gossiping protocol to attain a very high fault-tolerance.

Half of the surveyed incomplete systems have equivalent scalability and flexibility (figure 4.4). They use unstructured networks where the number of messages generated per query is limited by a TTL, providing a constant cost for searches. Their flexibility is high since the matching between queries and resource descriptions is evaluated locally. Therefore, it can include a variable number of attributes. The other systems make use of a variety of mechanisms over structured networks.

[1]Available at http://sword.cs.williams.edu/

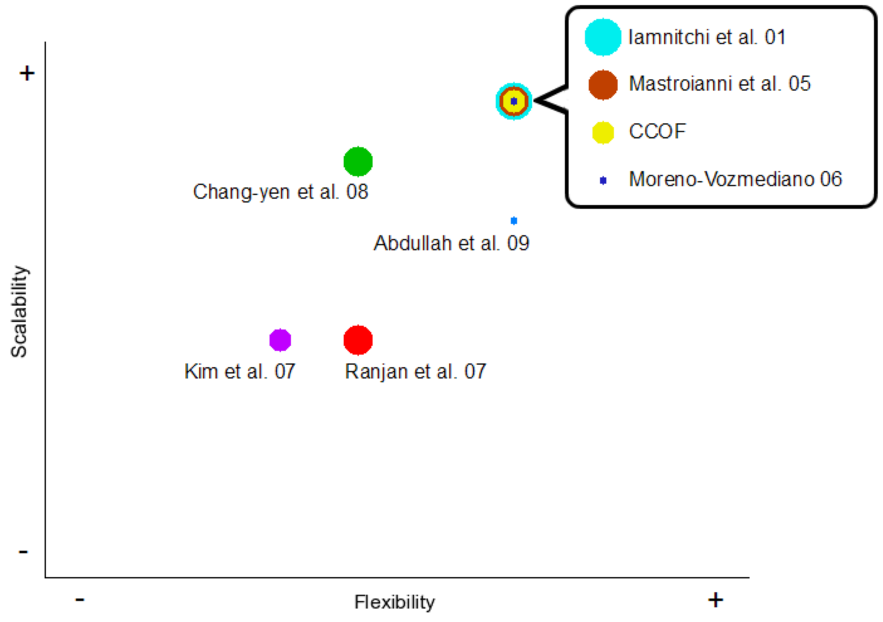


Figure 4.4: Incomplete systems organized for their search flexibility and scalability, with size meaning fault-tolerance. Systems with same flexibility and scalability are listed separately for clarity. Half of the systems have high flexibility and cost bounded by a constant.

4.4.2 Performance Comparison

Evaluation settings

Next we compare the performance of some of the systems presented throughout this section. This comparison is based on the information provided by the authors of the reviewed papers, and therefore the conditions under which results are obtained are different for each experiment, and the comparison of the values must be taken carefully.

Since each paper shows different experiments and uses different metrics for evaluation, we only took a subset of the reviewed papers, which presented similar metrics. These are divided in two groups, those that present response time measures, and those that present hop count measures. In both cases, the papers used different environments, settings and techniques for evaluation. Therefore, the direct comparison of the presented results is of limited value. Our motivation for putting them together into a single graph is merely to make it easier to spot different tendencies in the metrics of each system, rather than comparing the specific values.

Finally, we also comment the different measures of communication overhead presented in some of the surveyed papers. We do not provide a comparative graph due to the great difference in overhead measurements at each of the papers, but rather comment them and compare them where appropriate.

Note again that all the presented results do not belong to tests repeated under the same conditions

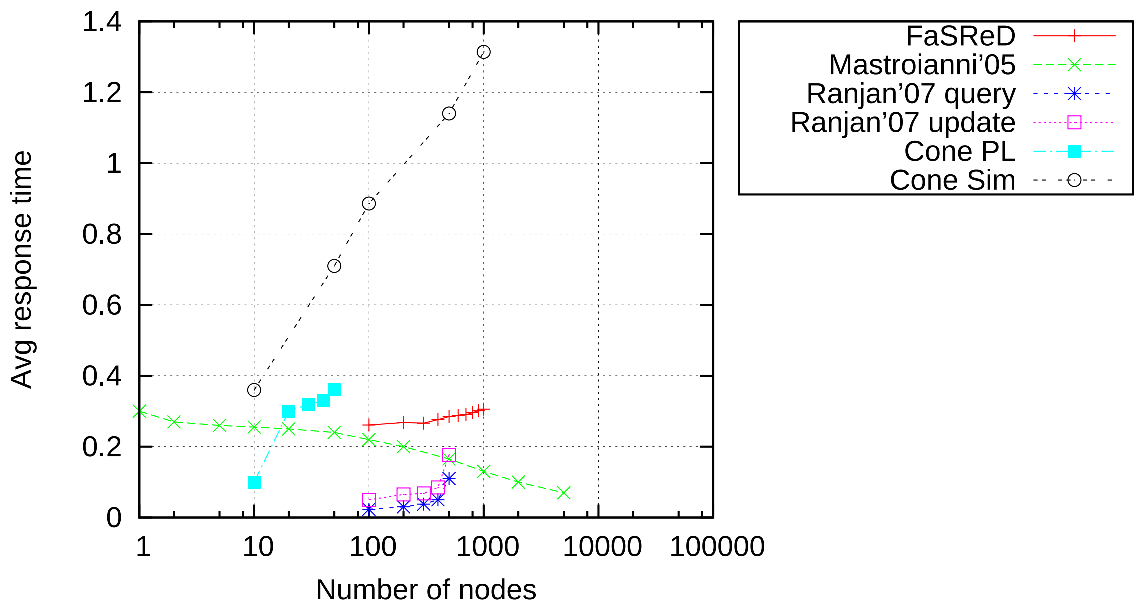


Figure 4.5: Average response time vs. number of nodes

for all the systems, but to different tests under different conditions presented in the literature. Moreover, not all the systems could be included in these comparisons.

Response time

Figure 4.5 shows the average query response time reported by various systems. The system by Ranjan et al. also reports the average response time for updates of resource descriptions. We show the results of Cone, when executed on a simulator (Cone Sim) and when executed on PlanetLab (Cone PL). We can see that the response time observed when executed over PlanetLab is significantly lower than when executed over a simulator. The authors explain that this is due to the good connections among the PlanetLab nodes selected for the experiments, which share Internet2 connections, while their simulation considers a worse connected network. In the simulation results, which are probably more realistic according to the authors, we see that the response time increases logarithmically with respect to the network size. Although this is good, the response time for groups of over 500 nodes is greater than 1 second, which more than doubles the results reported by the other systems. Moreover, even using PlanetLab and its better Internet2 connections, the results are worst than those of the other systems. In figure 4.6 we see the results without considering Cone Sim, in order to appreciate the differences among the rest in a more adequate scale. In other words, it shows the same data than figure 4.5 but with a zoom in to see more clearly the smaller values and their relative differences.

We also show the results by Ranjan et al. which comprise the response time of queries (Ranjan'07 query) as well as updates (Ranjan'07 update). The response times achieved by Ranjan et al., lower

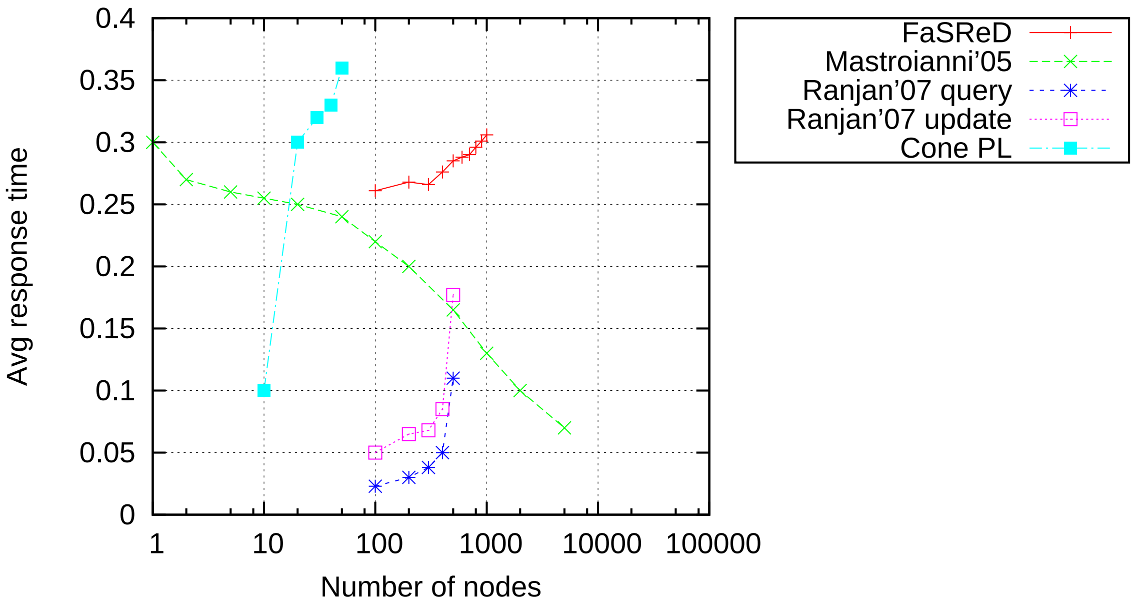


Figure 4.6: Average response time vs. number of nodes, for systems with lower response times

than 100 ms for network sizes up to 400, are better than those of the rest of the systems. However, the fact that the last value, for 500 nodes, is more than the double of the response time with 400 nodes, for both updates and queries, makes us question about the real scalability of the system. Moreover, in their paper, Ranjan et al. discuss the effects of message queues in their simulated nodes, and the harmful effects of reducing such queue, as messages are lost and results can change considerably. Therefore, the apparently good results of the system might come at the cost of increased overhead, although this will be discussed later.

FaSReD shows a very small increase in response time when tested with groups of 100 to 1000 nodes. The results are in a range between 0.26 and 0.31 seconds. However, FaSReD requires that the structure of its index tree and the distribution of queries are modeled alike, requiring precise values for some attributes while accepting any value for others. This makes it difficult to determine if the presented average values are really representative of the values that a real system would offer its users.

The system by Mastroianni et al. uses an unstructured topology with queries being forwarded with a fixed time-to-live (TTL). Therefore, the scalability of the system is guaranteed, as the maximum cost of a query is constant, limited by the TTL. Moreover, as with increase in size comes an increase in the amount of resources fulfilling a query, because the number of different types of resources is limited, the tests presented by the authors show that the response time actually decreases as network size increases. Specifically, they state that resources can belong to one of a number of different classes, and that the number of classes grows logarithmically with respect to the number of nodes in the network.

Tanin et al. also report average response times for their system, with networks of between 500 and

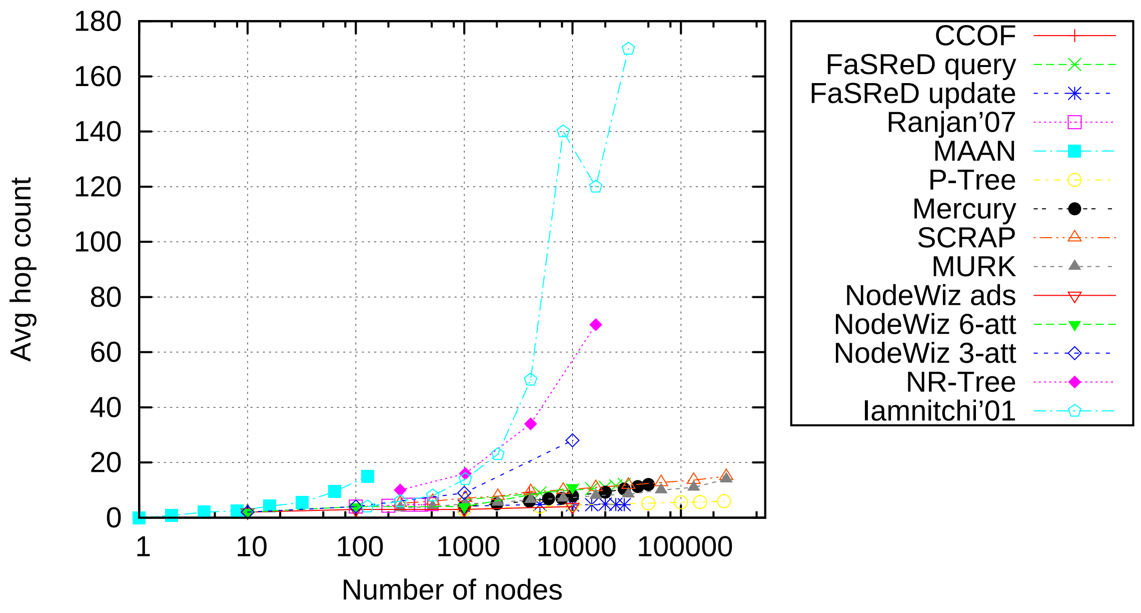


Figure 4.7: Average hop count vs. number of nodes

3000 peers. However, we have not included these data in our graph because the reported values are above 40 seconds, which is much greater than the results reported by other systems. Nevertheless, it must be noted that the presented data only shows a very minor increase in response time when the number of peers increases.

Hop count

In figure 4.7 we present the average number of hops in the overlay network per query for a series of systems, and per update or ad when specified. Figure 4.8 only considers the systems with results under 15 hops. In other words, it shows the same data than figure 4.7 but with a zoom in to see more clearly the smaller values and their relative differences. We have not tried to compare these results of average number of hops with the response time results, although both metrics are obviously related. However, trying to convert the values presented by the authors by assigning an average time value to a hop would probably disturb the results and make them less faithful to the truth. Moreover, some systems group nodes in clusters, using super-peers or proximity routing, which would further hamper this direct comparison. Therefore, these results and their relation with temporal cost measures must be taken carefully.

As a comparison referent, we can use FaSReD, which provides both response time and hop count average values. We see that updates have a constant average cost of less than 5 hops with network sizes up to 30,000 nodes. The number of hops per query, however, increases logarithmically, requiring 12 hops in a network of 30,000 nodes.

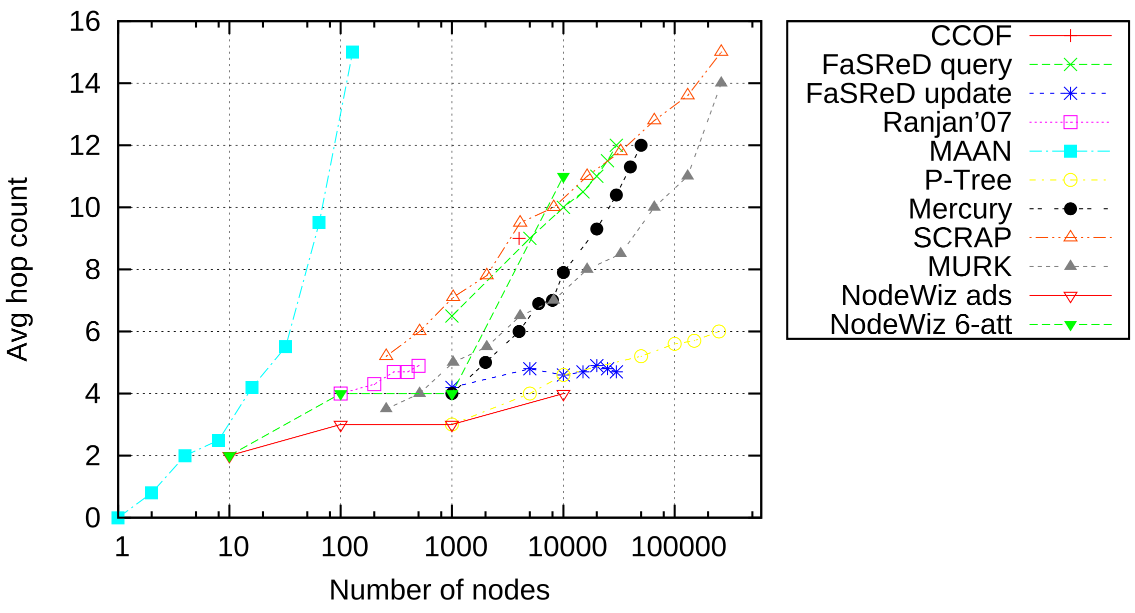


Figure 4.8: Average hop count vs. number of nodes, for systems with a lower hop count

Also evaluated by both metrics is the system by Ranjan et al., which gives indistinguishable hop count values for queries and updates, and therefore appears in figures 4.7 and 4.8 as only one data set. The number of hops is consistently under 5, although the maximum network size tested is 500 nodes. However, it is worth noting that the sudden increase in response time when network size increases from 400 to 500 nodes has no counterpart in number of hops.

The proposal of Iamnitchi et al. requires a high number of hops when the system size increases, according to the results presented in one of their papers [78], and although it remains low compared to the system sizes treated (e.g. 170 hops for 32,768 nodes), it is worse than logarithmic, and much worse than the results presented by other systems. However, in a previous paper [79], not included in the graph, they present results with up to 5,000 nodes, requiring 16 hops per query in the worst case. This is caused by the fact that, contrary to what they did in the previous one [79], in their second paper [78] they slowly increase the amount of resource types with the total amount of resources, therefore causing resource frequency to not increase as fast.

NR-Tree counts the average number of "visited peers" per query. Although its numbers are well under those presented by Iamnitchi et al., they also are above those of the rest of the systems. For NodeWiz we have three result sets, with very different performance. First we have the average number of hops for advertising resources (NodeWiz ads). It presents the best results, with only 4 hops per ad in an overlay network of 10,000 nodes. This is because the advertisement only requires a search in the tree, which has a logarithmic cost. However, the cost of the queries is substantially higher, and the results are worse as less attributes are specified in the query. When queries specify 6 attributes (NodeWiz

6-att) the average number of hops per query in a network of 10,000 nodes is 11, while for querys specifying 3 attributes (NodeWiz 3-att) the average number of hops is 28. It is a desirable property that resource advertising has a small cost, since it must be done periodically to refresh the information available to clients. However, having increased cost for queries when they have less requirements is less advantageous, since these queries are issued by users that have a greater number of resources that could potentially be used for their purpose, and should not require an extensive search.

MAAN's hop count is measured with networks of up to 128 nodes, which is small compared to the evaluation environments of other systems. Moreover, with such a small group, the required number of hops reaches 15, which is higher than the results reported by the rest of the systems. Mercury, on the other hand, presents tests with up to 50,000, reaching a maximum of 12 hops, using a direct-mapped cache with a Zipf-skewed workload. Even more impressive are the results of P-Tree, which measure a system of up to 250,000 nodes with a query cost of only 6 hops. However, this has its counterpart in the increased cost of insertions and deletions, not shown in our graphs, which can require almost 300 messages. MURK and SCRAP also present results with more than 250,000 nodes, with a routing cost of 15 hops approximately, using 2 attributes to define resources. However, SCRAP range queries have a higher cost than that presented on the graph, since a series of nodes must be sequentially visited in order to retrieve all the results that satisfy the query.

The system by Costa et al. also measures the number of hops required to answer a query, although it is presented as routing overhead, the number of nodes that forward the query and are not able to fulfill its requirements. However, this is not exactly the same as the number of hops per query, since the mechanism needs the query to individually be forwarded to each of the nodes that can answer it, before returning the list of matches to the user. Therefore, although the overhead is small (below 3 hops for up to 100,000 nodes), complete or large queries can take a much larger number of hops. Moreover, the results are presented versus the selectivity of queries, with results of as much as almost 300 hops in the worst case. Routing overhead versus network size is presented, but no additional information about the simulation settings is given in this case, so we do not put this data in the comparison.

Finally, CCOF presents only results for overlay networks formed by 4,000 nodes. Therefore, we present it here as a single point, with an average of 9 hops. It is very similar to the results obtained by FaSReD queries and NodeWiz queries with 6 attributes specified. In fact, in general we see that resource advertising takes less hops than queries. This may be due to the fact that advertisement have fixed values for each attribute, while queries usually must check a range of values. The exception is the system by Ranjan et al., which reports equal or lesser cost for queries than for ads.

Overhead

Although many of the papers describing the presented systems offer some measurements related to this topic, there is no common measure for the communications overhead of each of them, so no precise comparison can be made.

For some systems, there are measurements of the load supported by each node during a test. Mastroianni et al. present the number of messages per second at super-peers for different network sizes, from 10 to 10,000 nodes, and show that the average value stabilizes at about 9 messages per second for networks composed of more than 2,000 nodes, with a cluster size fixed at 10 nodes. With greater cluster sizes, the number of messages is shown to increase greatly, reaching a maximum of approximately 300 messages per second with clusters of 200 nodes. CCOF shows instead the total number of messages sent per node during a simulation, which has a duration of 24 hours. In a network of 4,000 nodes, the size used in their tests, the authors report an average of approximately 200 messages per node using the Rendez-vous points approach. Other approaches send a higher number of messages, reaching a maximum of over 1,600 messages per node with the ads-based and expanding ring approaches. This is well under the values presented by Mastroianni et al., which would require more than 700,000 messages per node in 24 hours. However, although Mastroianni et al. state that nodes issue queries at random with a mean inter-query time of 300 ms, the literature of CCOF does not provide any indication on the number of queries issued over the 24-hour period. Therefore, conclusions cannot be extracted from the comparison of these data. NodeWiz shows instead the average number of messages that traverse each used link of the overlay during a simulation. The length of the simulation depends on the number of events (queries and updates), which is fixed to be 100 times the number of nodes. The number of messages is shown to decrease as the network size increases, with an (approximate) average value of near 48 messages for 10 nodes, near 14 for 100 nodes, about 7 for 1,000 nodes and below 6 for 10,000 nodes. Since in Mastroianni et al.'s simulations all nodes issue queries, the duration of the simulation, in order to be comparable to that of NodeWiz, should be that which allowed for 100 queries to be issued by each node. As the mean time for query generation is fixed at 300 ms, 30 seconds would be required in order to send the number of queries, and in this time nodes would process an average of 270 messages. In any case, the comparison among NodeWiz's reported overhead and that of other systems is difficult since the authors only report the number of messages sent over used links, whose number is not given. Moreover, only the query and advertisement messages are counted.

Other systems report the total overhead on the network, like Kim et al., who report an approximate 500 MB of data sent every 5 minute interval by their system. Ranjan et al. count the total number of messages corresponding to queries and to updates for different sizes of their system during the simulation. However, the duration of the simulation is not specified, and therefore their data cannot be compared to that provided by other systems. The results shown by Chang-yen et al. report a total of approximately 10,000 packets sent in the system during their simulations. However, the duration of each test is the time needed to assign a minimum of 100 jobs to each node. Therefore, precise comparison with metrics used by other systems cannot be made.

Moreno-Vozmediano reports the number of messages required by his system to find resources in a network of unspecified size as 124 with his parameter R, the distance to which queries and ads are forwarded, fixed to 1, and 195 with R fixed to 2. However, in this network the size is not related to the number of messages per query, but rather the connectivity degree and the topology in general. No data about these factors is given either. FaSReD also presents results on the number of messages per query, in this case in function of the number of nodes in the PHT. The results reported for the best performing configuration range from 25 messages with a PHT of 100 nodes to 42 for a PHT of 1,000 nodes.

Finally, SWORD reports the average bandwidth consumed in one hour with 1000 nodes in the network to be approximately 11.1 Mb/s in their SingleQuery approach. This is similar to the data reported by Kim et al., which would be equivalent to approximately 13 Mb/s.

4.4.3 Conclusions

The relation among the different requirements faced by resource discovery systems and the design approaches taken by different proposals can be seen through the comparison of the systems discussed in this section.

The first thing that we see is that having a query language with enough flexibility to specify range values for a set of attributes is required for any system that aims at deployment in a real environment. However, systems that use DHTs and similar methods to organize information in a distributed index usually require beforehand knowledge of the defined attributes. Moreover, they might also require knowledge of the distribution of values that will be present and queried, whether in order to optimize the organization of the index for better performance, or in order to work at all. Query-forwarding techniques for unstructured topologies, on the other hand, require each node that processes the query to locally compare it with the data stored by it about available resources, which allows for a greater flexibility.

Only one of the discussed systems, SWORD, presents support for inter-node properties. These properties are evaluated by a centralized optimizer, executed locally by the node issuing a query over the returned results. Therefore, many other systems, including CoDeS, could make use of such an optimizer to further refine the results of their queries, incorporating inter-node relations or other types of properties. The information required to perform this evaluation is obtained through a mechanism of representatives, which are a subset of the nodes of the networks. The latency among the representatives is measured and stored, and the latency between any two nodes is considered to be that of their assigned representatives. This mechanism is independent from the one used to obtain resources that satisfy individual requirements. However, the implementation of SWORD deployed in PlanetLab does not provide support for these type of queries. We believe that more research on methods to efficiently provide such evaluation of inter-node requirements is needed to enable their use in real dynamic systems. Since we do not address the support of composite services, as stated in Section 2.3.4, we consider inter-node properties to be out of the scope of this thesis.

Unstructured topologies offer a better scalability in general, since resources can be found with a constant cost, derived from fixed values like the TTL of queries. However, this is only true when required resources are frequent, i.e. there is a great amount of resources available to satisfy any given query. In order to find rare resources, completeness is required. This is provided by organized indexes based on structured topologies, but at a cost of greater overhead. This overhead comes in the form of topology maintenance and reconfiguration messages, periodical advertisement messages in most systems, which use a combination of pull and push techniques, or longer paths toward resources, among others. Moreover, some systems that use structured topologies provide complete answers to range queries at a great cost, where in most cases completeness is not required. However, this overhead can be easily surpassed by the cost of finding rare resources in unstructured systems, which might require flooding the whole network with a query. These are clear trade-offs that must guide the selection of a specific resource discovery mechanism for a system depending on the environment where it will be deployed and the requirements that it has to meet.

One aspect that many systems do no develop enough is fault-tolerance. Many systems rely on the fault-tolerance claimed by the overlay network, without considering the specific aspects of the resource discovery mechanisms that might hamper performance in presence of failures, or even render the system unusable. Similarly, security is seldom considered, and while most systems cannot be victims of denial-of-service attacks in their totality, they are vulnerable to attacks rendering the system partially nonfunctional or unavailable. Moreover, malicious nodes can often control what resources are returned by a significant proportion of queries, requiring additional security guarantees for the remote execution mechanisms in order to avoid attackers obtaining

4.5 Resource Discovery Mechanism for Contributory Systems

After carefully considering all the conclusions from the study of existing resource discovery mechanisms, we have decided to develop a novel mechanism for CoDeS. Our mechanism leverages the service deployment feature present in the middleware to attain the desired trade-offs between the different requirements.

Resource discovery functionality is provided by a set of matchmakers scattered throughout the community. They receive and store ads from resource providers. Users send queries to them and retrieve the ads of resources that match the query. The use of matchmakers resembles many centralized computing environments, retaining part of their simplicity and flexibility. Specifically, it allows flexibility thanks to comparing requirements and resource descriptions locally, which allows the use of any description language and different matchmaking mechanisms and policies.

Scalability is provided by using a number of matchmakers proportional to the size of the community, so that they are not overloaded and do not become a bottleneck for the system. The trade-off between completeness and performance is also solved by having each matchmaker store the descriptions for only a part of the nodes in the community. Clients only need to contact a single matchmaker to have a good probability of finding frequent resources. For rare resources, on the other hand, clients can contact more matchmakers, even all of them if complete results are required.

The use of matchmakers in decentralized systems has been proposed before, for instance in the system by Abdullah et al. [2, 3], but the creation and location of matchmakers were considered to be performed out of band. In CoDeS, matchmakers are deployed as services on top of the community. They are, therefore, managed autonomously and use the same resources that users contribute for program execution.

There are three main actors in the process of discovering resources using the matchmakers mechanism, namely the workers, the clients and the matchmakers. Workers offer resources, clients consume resources and therefore need to discover such resources, and matchmakers put both clients and workers in contact. Any node can play the role of worker, client and matchmaker, even simultaneously.

Our mechanism combines the push and pull approaches. Workers send ads to the matchmakers, which store them. When a client needs resources, it asks a matchmaker, and this checks the ads it stores in order to find possible matches. If some ads satisfy the query, the matchmaker sends the contact data of the workers to the client. The client then contacts the workers in order to start a remote execution.

4.5.1 Matchmakers

Matchmaking is a default service deployed in the contributory community using a predefined identifier. As seen in Figure 4.9, matchmakers are deployed along with user-defined services. Clients and workers contact the Service Manager to obtain a list of matchmaker locations. After this, the node sends its ads or queries directly to the matchmaker.

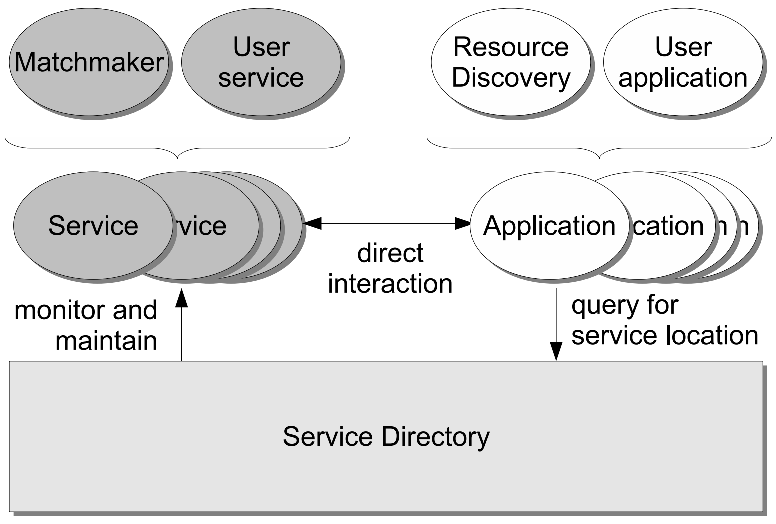


Figure 4.9: Matchmakers are deployed as a service in the community. Once the location of a matchmaker is known, the Resource Discovery Module can directly send queries or ads to it.

The number of matchmakers is part of the description of the service, and should be set accordingly to the expected size of the community. In case the size changes, the number of matchmakers may be modified in the service description, either by users or by an automatic process. However, such a process is out of the focus of this thesis.

Matchmakers use a soft-state approach to manage ads. Ads are stored for a limited period of time, marked by a maximum Time-To-Live (TTL) assigned to all ads. When this TTL expires, the ad is deleted. However, if another ad for the same resource is received while a previous one still exists, the new one substitutes the old with a full TTL. This is useful for resources that suffer some change in their capacity (for instance, because they start executing a service which consumes a part of it), or simply to renew the publication of the resource.

Matchmakers try to match received queries with stored ads. The contact information of the resources found is sent to the client, or a subset of them if more than the required number has been found. The format of ads and queries as well as the process by which they are matched is independent of the rest of the resource discovery mechanism, controlled by the Specification Manager. It can be adapted to standard schemes of resource specification.

Another responsibility of matchmakers is to keep a partial list of the matchmakers in the community. This is received periodically from the Service Manager that controls the matchmaker. Depending on the security policies of the community, this list may exclusively contain matchmakers that are controlled by the same SM, or it may include matchmakers from other SMs.

4.5.2 Workers

Workers offer resources to the community. However, their information is not stored in all the matchmakers. Ads are periodically sent to only a random subset of matchmakers. This way, matchmakers do not have to store the information of all the nodes, which makes the mechanism scalable. On the other hand, having the ad replicated in a few matchmakers increases the probability that a query will find the worker even in case of failure or disconnection of some of the contacted matchmakers.

When the information of the resource changes, whether because its owner decides to modify the amount of resources provided, or because some service starts or ends being executed in it, a new ad is sent to those matchmakers that received the previous one. This, together with the limited TTL of ads, helps prevent the presence of stale data in matchmakers.

Workers keep a list of matchmakers, so they can send their ads directly with a constant cost, instead of using the KBR. This list must be periodically updated in order to eliminate stale data containing failed matchmakers as well as balance their load by allowing nodes send messages to different matchmakers, including those recently created. The list is first obtained by contacting any of the Service Managers responsible for the matchmakers. However, it is periodically refreshed by setting a flag in the ads. In response to this, the matchmaker sends a list of other matchmakers in the community, which the worker uses to update its list. Therefore, this update does not require a periodical look-up operation on the overlay network, but only a single additional message containing the list. This communication cost is important because typically all nodes in a community will act as workers, and the aggregated cost of the update operations is not negligible.

4.5.3 Clients

Clients look for resources that satisfy a certain set of requirements. This role is frequently played by Service Managers, which need to find resources where their managed services can be executed. However, user applications may also use the resource discovery mechanism as clients in order to find resources to use for their purposes.

Clients looking for resources send a query to one randomly selected matchmaker or a set of them, containing the requirements and the required number of results.

Clients maintain a list of matchmakers to send both ads and queries, since they normally contribute resources as workers. The maintenance of the list is only started when the node sends its first query or ad, with the cost of a KBR look-up operation. Otherwise the query has a constant cost.

If a query does not receive the required number of resources, the client sends the query to a different set of matchmakers, until the required resources are found or all the matchmakers in the list have been contacted. After this, the list of retrieved resources is returned to the user that requested it. If no resources have still been found, the client obtains the whole list of matchmakers from a SM and sends the query to all those that have not been contacted yet.

4.5.4 Bootstrapping

Service Managers deploy the matchmakers and keep them available in the community. However, they need to use the resource discovery mechanism to find resources that can execute the service. Therefore, we need a specific mechanism to deploy the first matchmakers of the community.

When a community is created, the matchmakers service does not initially exist. However, it is implicitly created the first time an ad or query is issued, respectively, by a worker or a client. When a worker tries to send an ad, a request for matchmakers is sent to one of the nodes responsible for the identifier of the matchmaker service, as determined by the KBR layer. If a Service Manager is not present, it is created there, and the service specification is stored in the distributed storage with the parameters (for instance, number of replicas) detailed in the CoDeS configuration files. The newly created SM takes the contact information of the worker that asked for matchmakers and starts the execution of a matchmaker in that resource. The same is done if the SM existed previously but its list of matchmakers is empty. This requires that the worker has enough capacity to host a matchmaker, but the requirements of this service are not very high and ultimately depend on the configuration of the community, which determines the number of ads that have to be stored and the number of messages that are received by each matchmaker. In Section 4.6 we evaluate the influence of some of these configuration parameters in the performance of the system.

Once the first matchmaker has been created, Service Managers can use it to find more resources and start the deployment of more replicas of the service. On the other hand, if the first request for the resource discovery mechanism was a query instead of an ad, the client will be contacted by the SM to start executing the matchmaker. If the client does not contribute (enough) resources as a worker, the execution will be rejected. The client will not be able to find resources, because no resources have been provided to the community yet. Eventually, if some member of the community contributes resources, the former case will occur and the matchmakers service will be started, leading to the resource discovery mechanism working as explained earlier.

4.6 Evaluation

4.6.1 Test Environment and Evaluation Criteria

In order to test the resource discovery mechanism, we have used the CoDeS prototype presented in Section 2.5, just like we did in Section 3.4. We have used FreePastry's simulation mode again to create a whole community in a single JVM. We have implemented the mechanisms presented in this chapter inside the RDM and used them to issue queries asking for resources in a variety of conditions.

To maintain modularity, the functions that are assumed by a SM for the matchmaker service are not implemented into the SM code, but as a separate class that is managed by the RDM. This class is only instantiated in the nodes that host a Service Manager for the resource discovery service. This way, resource discovery is implemented exclusively in the RDM and its subcomponents, which makes it easily replaceable by any other resource discovery mechanism that could fit the requirements of some specific contributory community.

The main aspects we have considered for our tests are community size and churn. Unlike in Section 3.4, we have not created load by deploying services on the community. The only activity that is present in these tests is that related with resource discovery: matchmakers deployed on the community, workers (all nodes) sending ads periodically, and clients (randomly selected nodes) issuing queries for resources. Additional aspects that we have considered are number of matchmakers in a community and query selectivity.

Again, we have given queries a default resource selectivity of 0.1. This means that each query can be satisfied, in average, by a 10% of the nodes of the community. Resource selection is implemented in a simplified way to avoid using any specific resource description language. We assign each node a number between 0 and 1. Queries select a random range of length 0.1 between 0 and 1, which means that they can be satisfied by hosts with a value inside that range. This way we can easily control the selectivity of the queries while not relating our tests to any specific resource sample. This will allow us to test different selectivity levels.

Like in Section 3.4, we have modeled churn synthetically using a triangular distribution to determine the rate of entrance and departure of nodes. To keep the total number of nodes stable, we have used the same distribution parameters for both inter-arrival and inter-departure time. The default modal value is 7 seconds, with the limits of the triangular distribution being always 100 times more and less the mode.

Table 4.4: System parameter values for the three configurations of resource discovery used in the tests.

System parameters	Eager	Medium	Lazy
Number of matchmakers		4	
Ad periodicity (seconds)	20	60	180
Number of ads sent	4	4	2
Ad TTL (minutes)	1	3	10
Matchmaker list refresh periodicity for workers (minutes)	1	1	15
Matchmaker list refresh periodicity for matchmakers (minutes)	1	3	9
Matchmakers list TTL (seconds)	100	300	300
Length of matchmakers list sent to workers		4	
Length of matchmakers list sent to matchmakers		4	

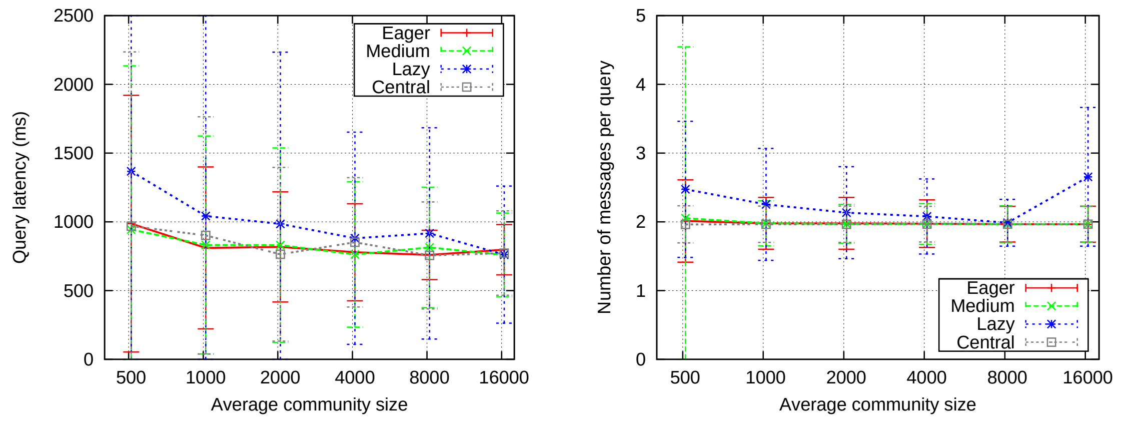
We have used again three configurations of the system, which differ in the number of messages they send. This configurations are named eager, medium and lazy. Table 4.4 presents the values assigned to different system parameters in each of these configurations. Parameters not related specifically to resource discovery are shown in Table 3.2, in Section 3.4.1.

We evaluate the cost of queries for finding resources. We measure it both in number of messages per query and in time required to get the results. We also measure the overhead of the mechanism in messages per node per minute. We compare the performance of our mechanisms with the centralized mechanism used in Section 3.4. The centralized matchmaker uses the same code than the matchmakers that are deployed as a service, but there is only one matchmaker for the whole community. The central matchmaker is migrated immediately if the node that hosts it fails, and its location is known to all the nodes in the community without requiring any communication.

Our simulations last for 8 hours (in simulation time), and during all this time queries are issued periodically. The time between queries is fixed at 10 seconds, which gives a total of 2880 queries issued per simulation. Each query is issued by a randomly selected host among the ones that are alive at that moment. Our plots show, for each simulation, the average and the standard deviation of the latency and number of messages for each query. The plots that show number of messages per minute show the average during all the simulation for all nodes.

4.6.2 Scalability

We test the effect of community size on the performance of the resource discovery mechanism. For this purpose, we simulate communities with sizes ranging from 128 to 16384 members. We compare our three configurations of distributed resource discovery, namely eager, medium and lazy, with a centralized resource discovery which follows the parameters of the medium configuration with respect to ad periodicity.



(a) Query latency for different sizes.

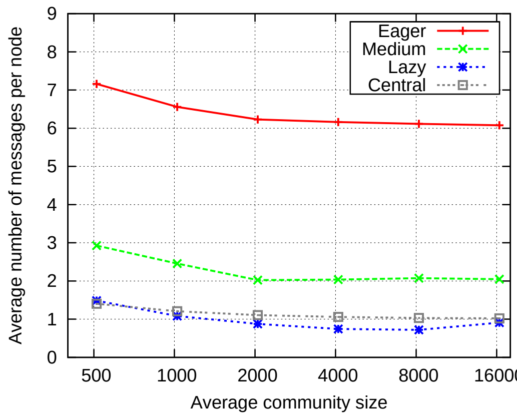
(b) Number of messages involved in resource discovery queries, for different sizes.

Figure 4.10: Cost per query for different community sizes.

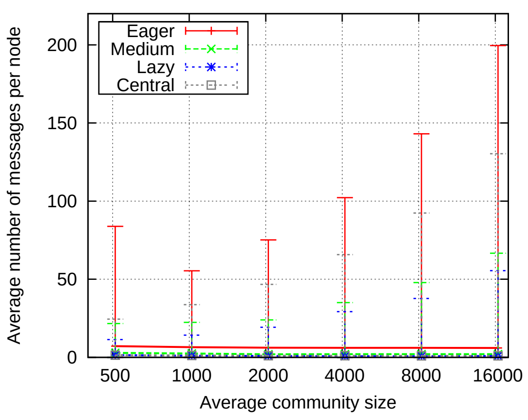
As seen in Figure 4.10, query cost, both in latency and in number of messages, is almost constant for the medium and eager configurations, because queries are solved with a single message. Lazy gives worse results, especially for the smaller communities. The main reason is that this configuration has a higher probability of returning outdated information about resources. If a query returns results that are no longer valid (in these tests, because the referred node has left the community) the client that issued it sends the query to a different matchmaker. This is more likely to happen for the lazy configuration, causing a higher average query latency and message count. Moreover, we identify two possible causes that increase this effect in smaller communities.

One is the lower probability of finding resources with a selectivity of 10% in such small groups. While in a large community, a 10% of the members is still a large amount, the number of hosts that satisfy a query in a small community is proportionally small. Since we have used the same number of matchmakers in these tests with different community sizes, the number of ads stored per matchmaker decreases in small communities. Therefore, the probability of finding resources that fulfill a query in a single matchmaker decreases with the community size.

The second reason is simply that the effects of churn are more visible when a community is small. This comes from the fact that we quantify churn as time between consecutive events (both entrances and departures). This rate is the same in all the scalability tests, and therefore affects differently communities of different sizes, because the mean lifetime of a node will be different. Since resource discovery uses optimistic techniques for keeping the list of matchmakers, churn can make this list temporarily incorrect and cause that hosts issuing queries cannot find available matchmakers. The mechanism could therefore be improved to accelerate detection of failed matchmakers, and contact the SM when the matchmakers list does not contain enough working matchmakers. In the case of the lazy configuration, the higher churn makes the presence of outdated ads even more likely, which together with the smaller amount of ads per matchmaker, explains the worse performance of this configuration.



(a) Number of messages sent per node per minute, for different sizes. Shows only average values.



(b) Number of messages sent per node per minute, for different sizes.

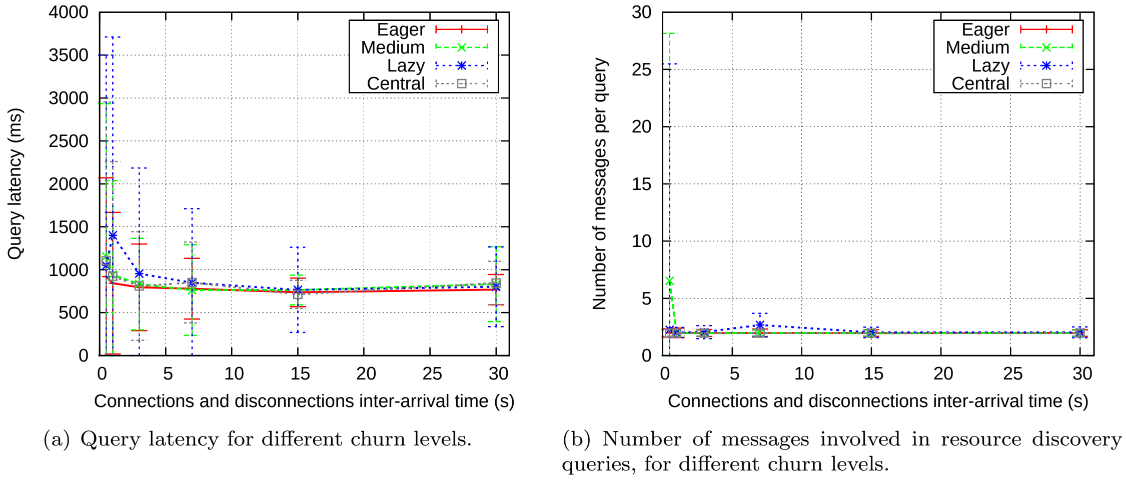
Figure 4.11: Communication overhead for different community sizes.

Figure 4.11 shows that number of messages per node per minute does not change much with size, although the standard deviation is increased because the load is not balanced equally among all nodes. Matchmakers are points of centralization, and increasing community size without proportionally increasing the number of matchmakers results in an increase of their load. Average number of messages per node does vary noticeably for each system configuration, as is expected. The central configuration has a lower cost than the medium and eager configurations. The latter is more costly because ads are sent more frequently, but the difference between central and medium is caused exclusively by the costs of our mechanism, i.e., the cost of keeping matchmakers available and informing workers and clients of their location. However, this overhead is small, and the plots show that variation of some parameters can counter this increased overhead, since the lazy configuration has a lower cost than the central.

The results have shown that the lazy configuration has potential for providing good performance for resource discovery in scenarios where probability of finding resources is relatively high, but it does behave worse than the other configurations with the churn level considered in these tests. However, the eager configuration does not provide any improvement over the performance of the medium configuration. Therefore, it is not justified to incur the increased cost of the eager configuration in a community that suffers a churn level similar to the one considered in these tests.

4.6.3 Fault Tolerance

We evaluate the results of churn on the resource discovery mechanism. We test different values for the mode of the triangular distribution that models time between node arrivals and between node departures. Figure 4.12shows that average cost per query, in number of messages, does not increase noticeably, but it does increase, albeit slightly, in latency. However, standard deviation does increase for lower values of time between churn events. This means that some queries do have a worse cost, but these are not enough to produce a large modification in the average values.



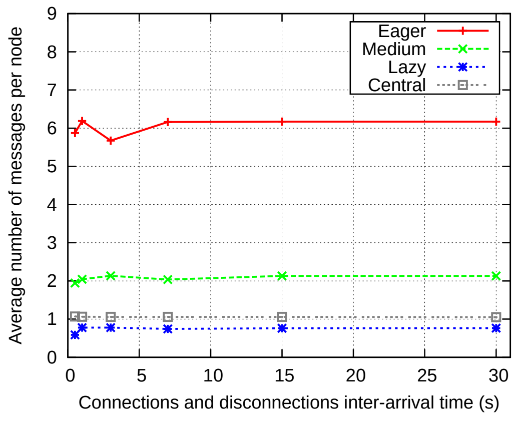
(a) Query latency for different churn levels.

(b) Number of messages involved in resource discovery queries, for different churn levels.

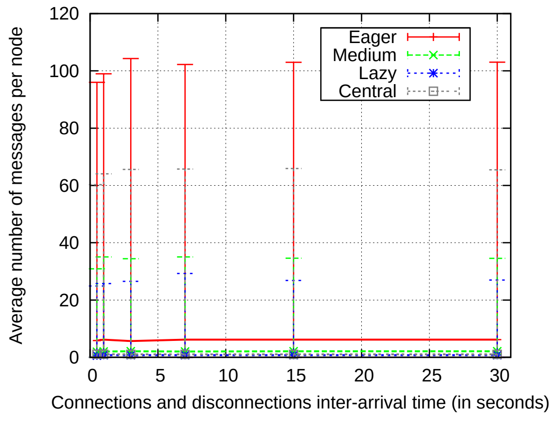
Figure 4.12: Cost per query for different churn levels, expressed in the modal value of inter-arrival and inter-departure time.

Communication overhead remains does not vary with different levels of churn, as shown in Figure 4.13. This seems logical, because ads and queries are sent periodically regardless of disconnections. A high level of churn could cause ads to be sent to failed matchmakers and then sent again until a working matchmaker is found, but since most ads do not expect an acknowledgement (only those which also request a list of matchmakers), this is not the case.

The results show that the proposed resource discovery mechanism offers good failure tolerance. It can deal with high levels of churn without large perturbations in cost per query nor in total communication overhead. Moreover, as shown in Section 3.4.3, the higher levels of churn tested here are higher than those found in actual systems. Therefore, we can expect the proposed mechanism to work correctly in a real environment.



(a) Number of messages sent per node per minute, for different churn levels. Shows only average values.



(b) Number of messages sent per node per minute, for different churn levels.

Figure 4.13: Communication overhead for different churn levels, expressed in the modal value of inter-arrival and inter-departure time.

4.6.4 Effects on Service Deployment

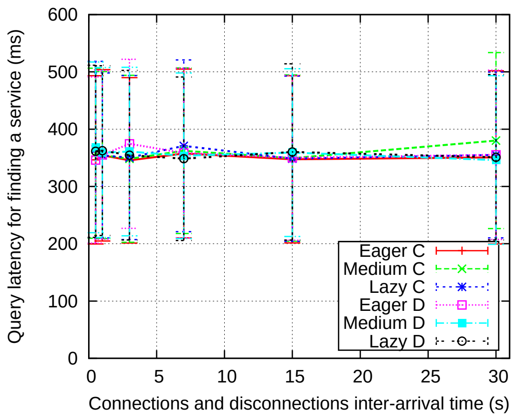
We have repeated the tests in Section 3.4 integrating the decentralized resource discovery mechanisms. We present here only a subset of the results, which are enough to show that the difference in the performance of service discovery when using one or the other is minimal. Specifically, Figure 4.14(a) shows that latency of queries for finding service locations is identical when using the centralized or distributed resource discovery mechanisms. Therefore, the use of the distributed mechanism has no detrimental effect on service availability, which is the main purpose of CoDeS.

Figure 4.14(b) shows the cost in messages per node per minute of each of the configurations. As expected, the cost of the configurations using the distributed mechanism is higher, because of the overhead of keeping an additional service. However, this cost is marginal with respect to the total cost of the system. This verifies the results in Figures 4.11 and 4.13 that show the small overhead of the resource discovery mechanism.

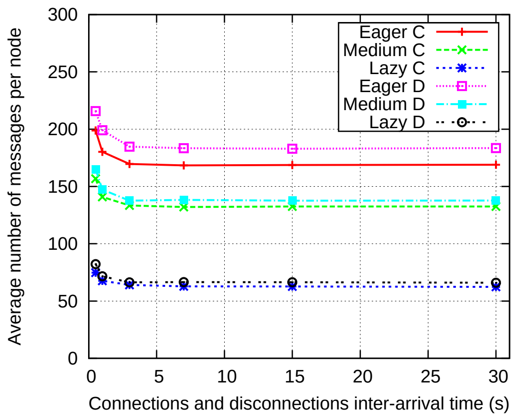
4.6.5 Comparison with Other Systems

In order to see how our resource discovery mechanism compares to other existing mechanisms, we have added CoReD to the plots in Section 4.4. Figure 4.15 presents the number of messages associated to each query. In this metric, our resource discovery mechanism performs better than the rest, requiring only two messages per query and remaining constant in communities of increasing size.

Figure 4.16 compares, instead, the latencies provided by our system with those reported by other systems. We see that our latencies are higher than most, but this result must be carefully considered,



(a) Query latency for different churn levels



(b) Number of messages received per node per minute, for different churn levels

Figure 4.14: Effects of the resource discovery mechanism in service deployment. The results ended with *C* show the configurations which use centralized resource discovery, and those ended with *D* show the configurations using distributed resource discovery.

since it depends heavily on the configuration of each test. For instance, the Cone mechanism presents two different results for its deployment over PlanetLab, in a low-latency network, and for its simulation tests. Moreover, the latter presents latency values higher than those obtained by our mechanism. In other words, each test, most of them simulations, may use different values for average message latency, which will affect the latency results.

4.7 Conclusions

In this chapter we have presented a resource discovery mechanism for contributory communities integrated in CoDeS. After careful evaluation of previously existing resource discovery mechanisms for a variety of distributed systems, we concluded that none of them satisfied completely the requirements of our middleware. For this reason, we decided to create a resource discovery mechanism that would leverage the features of CoDeS. We identified the use of matchmakers as a simple, effective and flexible way of implementing resource discovery. Therefore, it was a natural evolution to deploy matchmakers as a service on top of CoDeS.

We have evaluated the proposed resource discovery mechanism and seen that it can be used to find resources efficiently. The average cost of queries, in time and in number of messages, is constant with respect to community size. Moreover, it also tolerates large levels of churn. It does so thanks to its use of an optimistic approach, which offers a weaker consistency in trade for an improved performance.

The presented resource discovery mechanism has been designed as a part of CoDeS, and not as an independent mechanism for general use by external applications. However, a subset of CoDeS could be

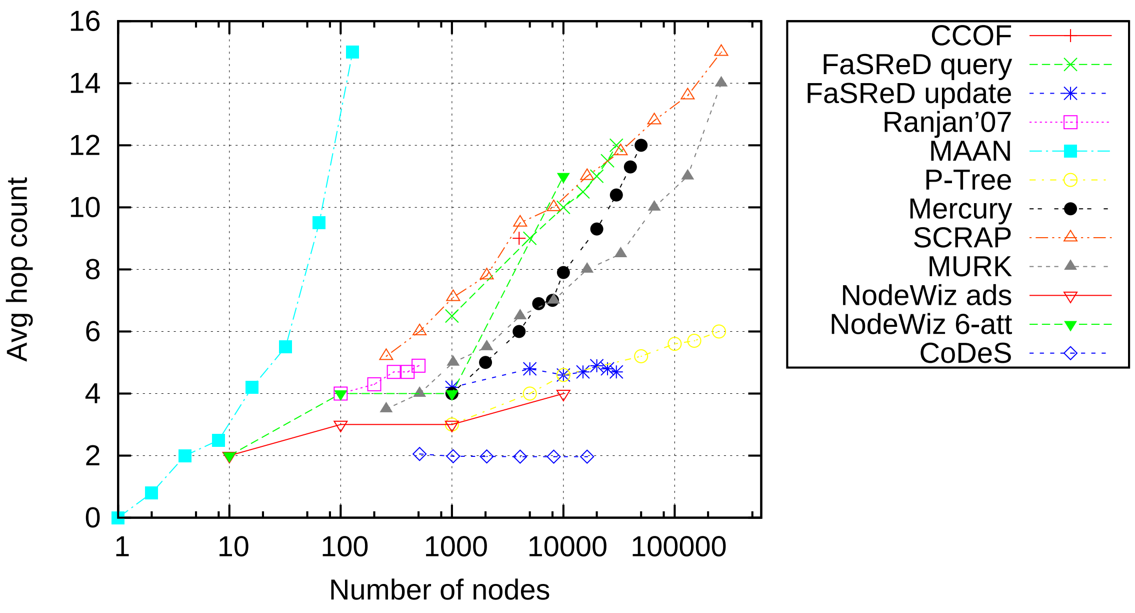


Figure 4.15: Comparison of CoDeS resource discovery mechanism's number of messages involved in resource discovery queries, for different community sizes,with other systems and mechanisms.

packaged as a resource discovery mechanism for grid or peer-to-peer applications. This subset would include the service deployment and resource discovery mechanisms and related modules, but would not require other complex parts like persistent storage or resource virtualization which would add overhead. Our results show that the communication cost of the proposed resource discovery mechanism is reasonable, and inferior to that of some other approaches, while offering a good performance and flexibility. Therefore, this mechanism could also be used as a stand-alone resource discovery mechanism to support general distributed applications.

4.7.1 Future Work

Some lines of future work for the resource discovery mechanisms presented are related with future developments of the whole CoDeS middleware. For instance, improving security in the middleware must also take into account the mechanisms used for resource discovery, and how to incorporate it into them.

An improvement that could be done to the resource discovery mechanism is the addition of auto-scaling. This refers to the action of adapting the number of matchmakers available in the system to the size of the community. There are obvious relations between the number of matchmakers and aspects like performance and load balance. For instance, the storage space required for executing a matchmaker is bigger when there are few matchmakers, while the maintenance overhead grows with the amount of

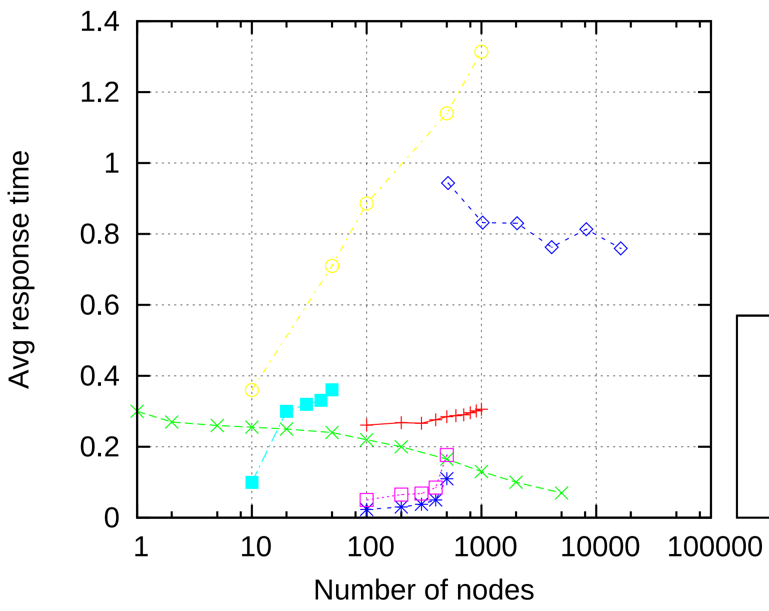


Figure 4.16: Comparison of CoDeS resource discovery mechanism's query latency for different community sizes with other systems and mechanisms.

matchmakers that have to be maintained. Some of our tests have hinted this relation, but we have not incorporated any automated mechanism to dynamically adapt the number of matchmakers. Therefore, it is an area of potential improvement, which could also lead to the creation of general mechanisms for service auto-scaling. These should incorporate a wider support for user-defined policies. However, we believe that such mechanisms should be designed with generalization in mind.

5 Availability Aware Resource Selection

This chapter deals with availability prediction and its use in resource selection for service deployment. To this end, the chapter includes some analysis of SETI@home traces, and the proposal and validation of an availability prediction method and a resource selection mechanism. Finally, it presents how different resource selection mechanisms can be integrated into CoDeS' architecture.

5.1 Overview

In the previous chapters we have presented the basic mechanisms of CoDeS, which provide service availability through a mechanism of replication with random replacement. This has been proven to provide good availability in our simulations. However, these simulations did not consider some factors like the transmission of files required for service execution, and the related communication costs.

The volatility of resources causes a continuous rate of service migrations. Services that are built to be deployed in a contributory community must be able to deal with churn in their replica set. However, the cost of file transmissions triggered by these migrations may not be negligible, and may hinder the performance of the whole community.

In a dynamic environment like a contributory community, there are two types of failures: transient and permanent. The former happen when nodes become unavailable over the course of time because of the activities of their owner (using the machine for her own tasks, or turning it off). The latter happen when either a machine is permanently broken or replaced, or when a user leaves the community.

Permanent failures must necessarily cause a replacement action for the services affected (those that where being executed by the failed node). Transient failures, on the other hand, can be dealt with in a variety of ways. A returning node could resume execution of the services that had been assigned to it. Therefore, if the failure of a node does not cause the number of available replicas of a service to fall below a given threshold (the number of replicas that the user considered necessary for the correct performance of the service), no replacement is required.

In a model with permanent assignments of services to hosts, it would be useful to try to predict the availability of hosts in order to select a pool of hosts which can provide the required collective availability over time. This means that out of M selected hosts, at least N of them will be available at any moment, without the need for migrating the service or data, thus minimizing communication overhead. Since we want to have a long-lasting deployment, we will require long-term availability prediction. However, long-term prediction may have lower accuracy than short-term prediction, where the system uses the most recent information to predict the immediate future.

One way to perform long-term prediction is to try and detect cyclical patterns in the behavior of hosts. Some works [59] have reported the presence of such kind of hosts in environments composed of non-dedicated resources, such as enterprise desktop grids, volunteer computing systems and peer-to-peer networks. This chapter studies such hosts using trace data of SETI@Home [136], and presents possible ways of using them to deploy services with high availability and little migration.

The contributions of this chapter are the following:

- **Analysis and identification of resource availability patterns in a real large-scale system.** We use a large set of trace data extracted from a real volunteer computing system (SETI@Home) to identify the presence of different types of hosts (always-on, always-off, cyclic, random) and quantify their presence and contribution to the overall availability of the system. We base this analysis on a volunteer computing system because: *a)* it is a large-scale system with real users, and *b)* the type of user that we consider for contributory systems is similar to the type of user found in volunteer computing systems.

- **Proposal of an availability prediction method based on the presence of availability patterns.** The method uses a bit vector, which summarizes the availability information for different intervals of time in a week, to predict the future availability of a host. We assess its quality as a predictor for real hosts using the SETI@home trace.

- **A resource selection method for service deployment based on the bit vector-based availability prediction which provides a high level of collective availability.** It can be used to deploy a service, be it a computational service or a set of data objects (e.g. erasure-coded), and achieve a good level of collective availability. We validate this method using real data, and compare it to random selection and short-term prediction-based methods.

- **Proposal of a design to integrate availability-aware resource selection mechanisms into CoDeS.** This design follows CoDeS' philosophy of extendability and modularity. Specifically, we design some extensions to CoDeS' architecture that make it capable of using different resource selection policies for each service, and present two example implementations of such policies.

The rest of this chapter is organized as follows. Section 5.2 discusses some related work. On the one hand, our review includes trace collection and analysis efforts for distributed systems based on non-dedicated resources. We focus on desktop grid because of their focus on CPU availability, which refers to the possibility of using the node for computation instead of merely contacting it through the network. Section 5.3 presents an analysis of a large set of SETI@home traces, and identifies the presence of periodic patterns in host availability. Section 5.4 presents a method to predict host availability, and Section 5.5 puts this method in practice to deploy services and obtain high collective availability. We then validate the presented mechanism using the SETI@home traces. Section 5.7 presents the extensions made to CoDeS to include availability-aware resource selection mechanisms. To clarify how new mechanisms and policies can be implemented, we present two examples, including the resource selection method presented in Section 5.5 and a previously existing prediction-based method. Finally, Section 5.8 concludes and presents some future work in the area of availability-aware service deployment.

5.2 Related work

5.2.1 Trace analysis

Predicting the future availability of a resource or a set of resources is only possible through the knowledge and understanding of their behavior. This can only be achieved analyzing data of real systems. However, obtaining data of distributed systems has some complexities that are only aggravated when considering Internet-distributed systems like volunteer computing or peer-to-peer networks. Considerable efforts have been devoted to obtaining real traces of a variety of systems, as well as to their analysis.

First we must make a distinction in the possible types of availability to be measured and analyzed. The most basic one is *host availability*. This refers to the fact that a host has a working connection to the network, and is therefore reachable by other hosts (we could say it is *pingable*). This notion has been used in most availability studies for Internet hosts, particularly inside P2P networks and other large-scale distributed systems [25, 30, 46, 59, 95, 125].

A different notion is *CPU availability*. This refers to the fact that a host's CPU is available to perform computations in behalf of the distributed systems. This distinction makes sense in cases like contributory or volunteer computing systems, where machines are used locally by their owner, and only surplus resources are contributed to the community or to a third party. CPU availability supersedes host availability, as network connectivity is a required but not sufficient condition.

Depending on the system and user, the definition of the conditions that classify as CPU availability may vary slightly. For instance, in some systems a machine may be only considered to be available when it enters idle state, after its local user has not touched the keyboard or mouse for a period of time. In other cases, the machine may be considered available when the percentage of CPU consumed by its local user does not reach a given threshold.

Some studies have analyzed CPU availability by obtaining information about host load [18, 30, 56, 57, 145]. However, this may not include different causes for CPU unavailability for the desktop, like user interactive activity (mouse or keyboard activity) or OS idiosyncrasies [87] like internal scheduling.

A different way to obtain data about CPU availability in a desktop grid is to submit measurement data that is seen by the workers as a real task. This way, the time assigned to the measurement task, and therefore counted as CPU availability, is the same that it would be to a real task. This method of measurement has been used for some works on different systems. However, the first availability studies of desktop grids only obtained information from a limited number (hundreds) of hosts located in enterprises or universities during a relatively short time period [87].

More recently, some bigger data sets have been published, like the SETI@home trace published in the Failure Trace Archive [88]. It contains about 230,000 hosts over the Internet, mainly at home locations, which is an important difference with older data sets. The trace period goes from April 1, 2007 to January 1, 2009, more than one and a half years. They are recorded at the BOINC server for SETI@home, thanks to an instrumented BOINC client. In total, the traces capture about 57,800 years of CPU time and 102,416,434 continuous intervals of CPU availability. This data set has been analyzed for characterizing the behavior of purely random hosts [80].

All these traces have been analyzed to find relevant properties in the behavior of hosts, as well as statistical models that can describe them. Some interesting general conclusions have been found in several of these works.

Douceur [59] studied the distribution of host availability in a few sets of Internet-distributed and enterprise hosts, considering the percentage of time each host is available. He identified this as the graduated mix of two uniform distributions. This could be caused by the existence of two types of hosts, identified by Douceur as one group of hosts with cyclical behavior, and another of hosts which are left on at all times.

Other studies aimed specifically at desktop grids, namely SETI@home, confirmed the existence of several types of hosts [86], broadly fitting the separation made by Douceur between cyclical and always-on hosts, together with always-off hosts, machines that are off almost all of the time and have very little time of availability (such machines could probably be counted into the group of cyclical hosts by Douceur's approach). It was also proved that no correlation exists between CPU availability and host speed [86]. Other works analyzed the temporal structure of availability by modeling the length of availability and unavailability intervals [80, 88], finding that some hosts exhibit a purely random behavior which can be accurately modeled by statistical distributions.

5.2.2 Availability prediction and guarantees

The purpose of CoDeS is to keep a service available. This is achieved by having it deployed on a set of hosts with at least a minimum subset of replicas of the service available at any given moment.

This notion has been called *collective availability* in previous works [12, 13]. A host may need to be network-available or CPU-available to be considered as being hosting an available replica of the service, depending on the policies applied in the community.

Andrzejak et al. [13] focused on achieving collective availability by predicting short-term availability. They used a prediction model, created with a set of training data, to forecast the availability of each host during a following interval of time of a specified length (called prediction interval length or pil). They proposed selecting hosts which are predicted to be available, getting a total amount of N*R, being R a replication factor needed to compensate prediction errors. This set would be able to keep the service available, with at least N available hosts, during the next prediction interval. They also presented metrics to evaluate predictability of a host, and showed that using highly predictable hosts, a replication factor of R=1.3 is enough to get availability values higher than 0.9.

One way to adapt this short-term prediction for achieving long-term availability is dividing the service lifetime (potentially very long) into shorter prediction intervals. The method presented above can be used to predict whether a host will be available during this interval, and hosts are selected to deploy the service based on these predictions. After the prediction interval has finished, the most recent data can be used to predict host availability during the next interval and modify the deployment as required.

This technique can give good results for relatively short-lived services. However, most hosts in peer-to-peer and volunteer environments suffer transient disconnections. Therefore, this adaptation will require migrations at a constant rate in the long run, even when no host permanently leaves the network. This will have a communication cost that may not be negligible. Moreover, since each host disconnection implies a migration, the only way to reduce the expected number of migrations is choosing the hosts with longer availability periods. This would cause a competition for a small subset of the available resources between the services that need to be deployed, while under-utilizing the resources with shorter availability intervals.

A different approach to predict host availability would be capturing long-term availability patterns. This could be useful for hosts that fit in the previous classification of always-on/cyclical/always-off. However, random hosts, whose presence in SETI@home has been proved by previous studies [80], may be difficult to model in this way. This approach has been previously used by Ramachandran et al. [116]. They proposed a summary of the availability of hosts during a week. Our approach, detailed in Sections 5.4 and 5.5, is similar to it. However, while Ramachandran et al. used the availability summary to select hosts with high expected availability during a period of time to execute a task, we will apply long-term availability prediction for service deployment. In other words, their purpose is to select hosts that are available over a short period of time, whereas our purpose is to select a set of hosts that provide collective availability over a long period of time. Another important difference is that they only tried their mechanisms with a small number of hosts (75) in three labs, while we have done a more exhaustive validation of our approach using the previously presented SETI@home trace, which contains hundreds of thousands of hosts, most of them in home environments.

A different case that can, however, be regarded as a matter of collective availability is that of data storage, where a data object is stored over distributed hosts and needs to be available for clients to retrieve it at any time. In order to improve the availability of the object, it may be replicated, i.e., stored in M hosts out of which at least one needs to be available at a given moment so that the object can be retrieved. Another possibility is to use erasure codes [55], which divide a data object in M fragments. A client needs to obtain N fragments ($N \leq M$) to restore the whole file. This case is similar to the service deployment case, as both require at least N available hosts. The main difference is that this only requires host availability (i.e., the host is reachable), while services require CPU availability (i.e., the host's CPU is free to execute the service).

A possible approach to providing availability guarantees in storage is to compute the required number of replicas (or fragments) of a data object, according to the average availability of hosts in the system [27, 84]. It is possible to compute the probability of having at least N out of M available replicas if the average host availability is known. In the case of storage, failures are often considered to be transient, meaning that when a host fails and recovers, the data it stored does not disappear. Therefore, whenever N of the hosts are available, it can be considered that there are N available replicas (or fragments) of the data object.

One problem of this approach is that it requires knowledge of the average host availability, which may require expensive procedures to be estimated. However, even more important is the fact that these works only consider average availability, as if all hosts' behaviors are identical and independent. According to Bakkaloglu et al. [20], using average availability without considering correlation, i.e., assuming host independence, may highly overestimate the actual availability obtained by a deployment. They also present methods to model the correlation level in a set of hosts.

As a final consideration, it must be noted that, in order to use predictive techniques in real time inside a distributed system, it is necessary to obtain updated data about the behavior and availability of the hosts that compose it. This requires monitoring the usage of these resources, which, as pointed by Morales and Gupta [105], is far from trivial.

A first basic solution would be having each host self-reporting its own availability. However, this is not reliable because a host can lie about its own availability. A second solution is using a centralized server to check each host's availability. This approach has scalability and failure tolerance problems, usually related to centralization. A third solution is assigning a set of observers to each node, which can report its availability. This is not trivial because of the entrance and departure of hosts in the system, which may make the monitoring system unavailable or unreliable. The latter may happen because it may be difficult to verify that a node is actually responsible for monitoring the hosts that it claims to monitor. A malicious node could use this to introduce false availability data. Morales and Gupta's approach [105] belongs to the third class. They solve the mentioned issues by assigning random observers to each node in a verifiable way, and computing the average among the availability reported by all of the observers of each node.

Another problem of the centralized and distributed approaches is that they can only be used to detect host availability, which can be monitored externally. If the system needs data about the load of the system, and the related notion of CPU availability, it can only be obtained from the host itself. In this case, it would be necessary to have methods to evaluate the reliability of the obtained information.

5.3 Trace analysis

5.3.1 Motivation

In order to detect periodical availability patterns, we have used a set of real CPU availability traces from SETI@home [88], which are part of the Failure Trace Archive, publicly available at http://fta.inria.fr. We selected this set of traces for two reasons.

The first reason is that the kind of environment represented in these traces closely resembles the environments which we target with the concept of contributory computing: a large community of individual users who voluntarily contribute their resources. While both our motivational reasoning (sharing resources in a community instead of providing resources to some entity) and resource usage model (general services versus computationally intensive parallel tasks) is different, there is a fundamental similarity. In both volunteer computing and contributory communities, users are willing to provide their resources. In other environments that use non-dedicated resources, like peer-to-peer systems, users are assumed to behave selfishly, which causes an extremely high dynamism and other issues like free-riding. Therefore, we believe that the user behavior of a large volunteer community like SETI@home is fundamentally similar to what could be expected of a contributory community.

The second reason for choosing these SETI@home traces is their scope. The traces were collected using the BOINC middleware for volunteer computing. BOINC serves as the basis for projects such as SETI@home, EINSTEIN@home, and climateprediction.net. These traces contain information taken from more than 200,000 hosts during a period of 1 year and 9 months. This turns them into the largest and most representative trace set of a desktop grid composed of hosts mainly in home environments, and one of the largest trace sets, if not the largest, of an Internet-distributed system formed by non-dedicated resources. Additionally, the traces do not show host availability, but CPU availability. The traces record the exact times when the CPU is available for computation as defined by the BOINC user preferences.

Because the target of our analysis is very specific (detecting patterns in host availability that can be exploited for long-term availability prediction), we refrain from presenting here more general characterizations of availability. Such analysis of the SETI@home traces can be found in other works [80, 86].

One exception, in the form of a general characteristic we will show here, is related to the question of whether hosts with low or medium availability contribute a significant portion of the compute power of a community. This question is important because, if the great majority of the capacity of the system comes from dedicated resources, then striving to make use of hosts with moderate availability, like those with cyclical patterns, would not make sense.

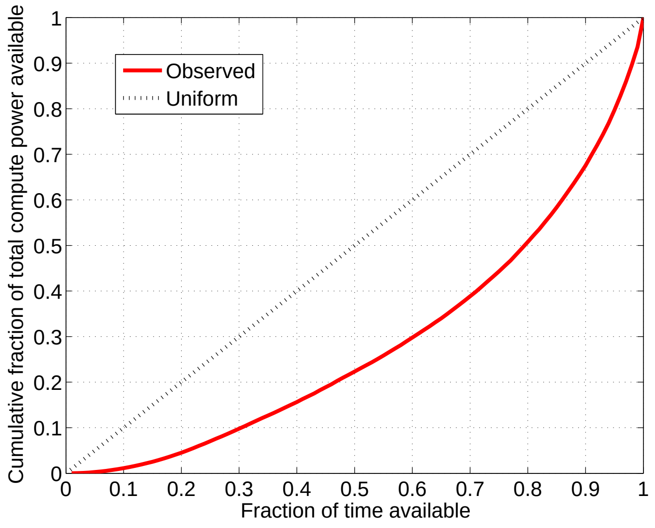


Figure 5.1: Cumulative CPU time multiplied by host speed over all hosts.

To investigate this issue, we follow the analysis found in a previous work by Kondo et al. [86] updated with the new data included in the latest version of the SETI@home trace. Figure 5.1 shows the fraction of compute power (CPU time multiplied by FPOPS peavailt) contributed by hosts according to their availability levels. The data point (x, y) means that the hosts with CPU availability of x or less contributed y of the total compute power. The plot corresponding to *Uniform* is used as a reference.

There is a significant skew in the amount of compute power contributed by hosts. 30% of total CPU time is contributed by hosts that are available 90% or more of the time. However, this does not mean that the contribution of hosts with lower availability is negligible. For instance, about 45% of the CPU time is contributed by hosts with moderate availability, between 55% and 90%. Moreover, hosts with 100% availability contribute less than 5% of the compute power. This is enough to reject the hypothesis that dedicated hosts are the only source of computing power in a volunteer or contributory community, and motivates us to develop mechanisms to make the most efficient use of non-dedicated resources.

5.3.2 Availability Patterns

As stated earlier, the purpose of CoDeS is to offer collective availability for a service over a long period of time. This means that we need to have a number of hosts M out of which at any moment at least N are available. One way to do this is by computing the average availability of hosts and deciding the number of replicas [27, 84]. This would give us a probabilistic guarantee of availability, considering the case where hosts have independent and random behavior.

However, real hosts do not (always) have random behavior. Previous works [59] have identified at least two types of hosts: those that are always on, and those that have cyclic availability patterns (e.g. available during work hours). This information can be leveraged to provide higher availability guarantees with a lower degree of redundancy.

Our first step to detect patterns in host behavior is to inspect the traces using a visualization software called Pajé [128]. This showed us that there were hosts with clearly periodic patterns (e.g. connecting every day during office hours), but at the same time there were others with long availability or unavailability stretches, or with no distinguishable patterns. Figure 5.2 shows a small random subsample of the trace visualized with Pajé.

Representing Patterns with Bit Vectors

Following the method in [86], we represent the availability of each host as a vector of 168 bits (24×7). Each bit represents an hour of the week, and its value is 1 if the host is usually available during that hour, and 0 otherwise. In more detail, we compute the percentage of time the host has been available during each hour in its lifetime (from the beginning of its first availability period in the trace to the end of the last one), and find the average for each hour of the week. Then we convert this value to 1 if it is equal or greater than a certain threshold (binarization threshold), or to 0 if it is lower. For the clustering, we set the threshold to 0.75. By setting it high, we focus only on patterns that are repetitive over a relatively long term.

A measure of the difference between the bit vector and the actual trace is the percentage of false

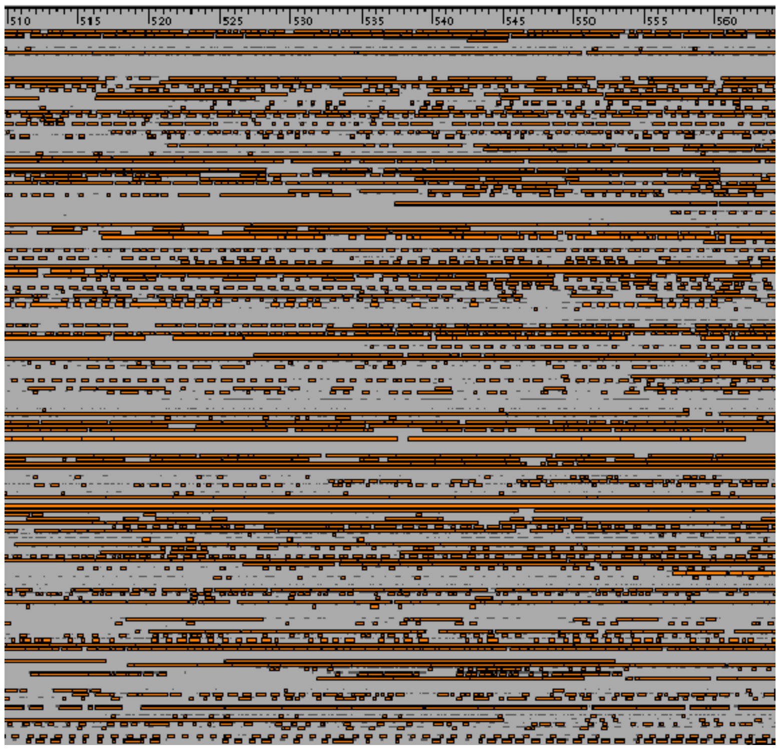


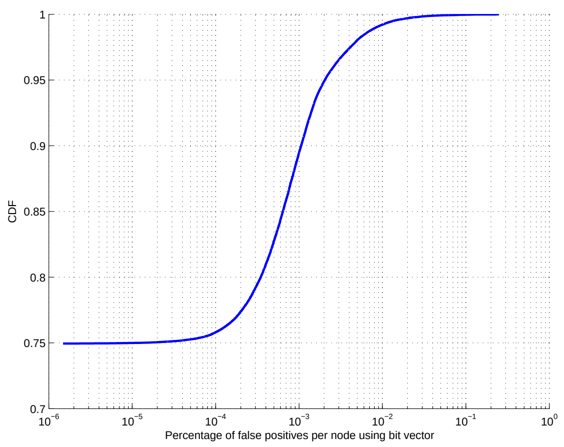
Figure 5.2: Trace visualized with Pajé. Each line represents a host, and the numbers show the days in the trace.

positives. We define it as the amount of time when the host is actually unavailable, but is shown to be available by the bit vector at that time of the week. Since our binarization threshold is 0.75, the maximum percentage of false positives that our bit vectors can show is 25%. This is because the worst case would be a host which is available at most 75% of the time during some hours of the week. Such a host would be considered to be available during those hours, but it would not be actually available in 25% of these cases. An hour for which the host has been available less than 75% of the time would have a 0 on the bit vector. Figure 5.3(a) shows that the amount of false positives is much lower than that for most cases, being lower than 1% for about 99% of the hosts.

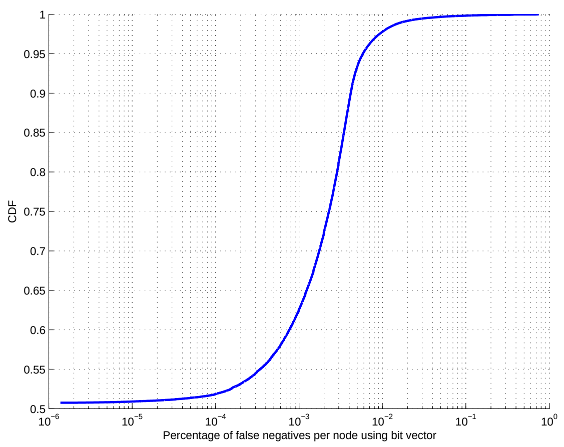
Similarly, we define the amount of false negatives as the percentage of time when the host was available but the bit vector showed it as unavailable. It is lower than 1% for about 97% of the hosts (see Figure 5.3(b)), much lower than the theoretical maximum of 75%.

Clustering

Our next step is to follow the method in [86], and use clustering to separate hosts with different patterns. The clustering is performed using the k-means algorithm [62]. It works by selecting k random points to use as cluster centroids, and assigning each element to the cluster with the nearest centroid. It does



(a) CDF of the fraction of false positives for all hosts, in log scale



(b) CDF of the percentage of false negatives for all hosts, in log scale

Figure 5.3: Difference between the bit vector summary and the actual trace

so iteratively, calculating the cluster centroid from the elements in the cluster and recomputing the distances between the elements and the centroids, until no changes occur. It ends when each element is in the cluster with the closest centroid, as calculated from the elements of the cluster.

To determine the similarity between hosts and centroids, we used a Hamming distance metric. Given two binary vectors, this metric measures the fraction of unequal values in each dimension.

The number of clusters k is an input parameter, and the results may vary in each execution because of the random selection of initial centroids. We tried different values of k, from 3 to 30. We then carefully inspected the quality of each cluster visually by plotting the centroids, and quantitatively by measuring the inter- and intra- vector distances from the centroid. We found that the clustering revealed some dominant clusters that were unmodified by changing k. We chose to show the results with k=6 as, after visual inspection, in these clusters the patterns were the most pronounced.

Our trace contains more hosts and a longer period of time than the one used in [86], but it's of the same system (SETI@home) and the results are highly consistent. The largest clusters are the one with 100% availability (*always-on*) and the one with 0% availability (*always-off*), as reflected in the cluster centroid. Of course this does not mean hosts actually have 100 or 0 availability, nor that their bit vectors say so. It only means that they are close to this behavior. The rest of the clusters comprise in total about 10% of the hosts and show more periodical patterns. Chosing higher values of k further subdivided these cyclical clusters, while leaving the always-on and always-off clusters unchanged. Figure 5.4 shows, for each cluster, with k=6, the sum of the bit vectors of all the hosts in the cluster, together with the centroid of the cluster multiplied by the total number of hosts in the cluster.

We also take a look at the actual behavior of the hosts in each cluster using Pajé. Figure 5.5(a) shows a small period of time of a random subsample of the hosts in the *always-off* cluster. Each horizontal line represents the availability of a host, and the numbers in the x axis show the days of the trace. Hosts do not have 0 availability, but they have sporadic availability intervals that do not follow identifiable patterns. For instance, some hosts in this portion of the trace show long availability intervals, but these are preceded and followed by unavailability intervals in the rest of the trace, making their average availability low. Figure 5.6 shows hosts from the *always-on* cluster, which have long availability intervals separated by short periods of unavailability. Note that our random subsample includes a few nodes that are not alive (they have not joined the system yet, or they have departed permanently) during the period shown in the figure, and therefore appear as unavailable even if they remained available for most of their lifetime. Figure 5.5(b) shows the behavior of a subsample of cluster 3, one of the clusters with cyclical availability patterns. Although not all hosts exhibit a clear periodical pattern, such behaviors are clearly present in the cluster.

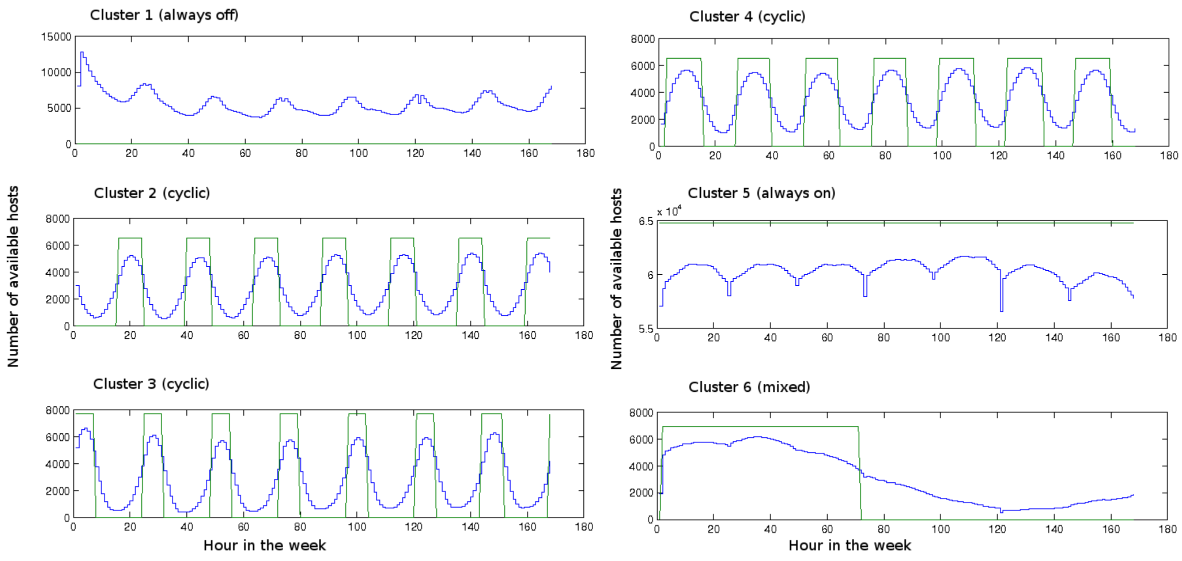
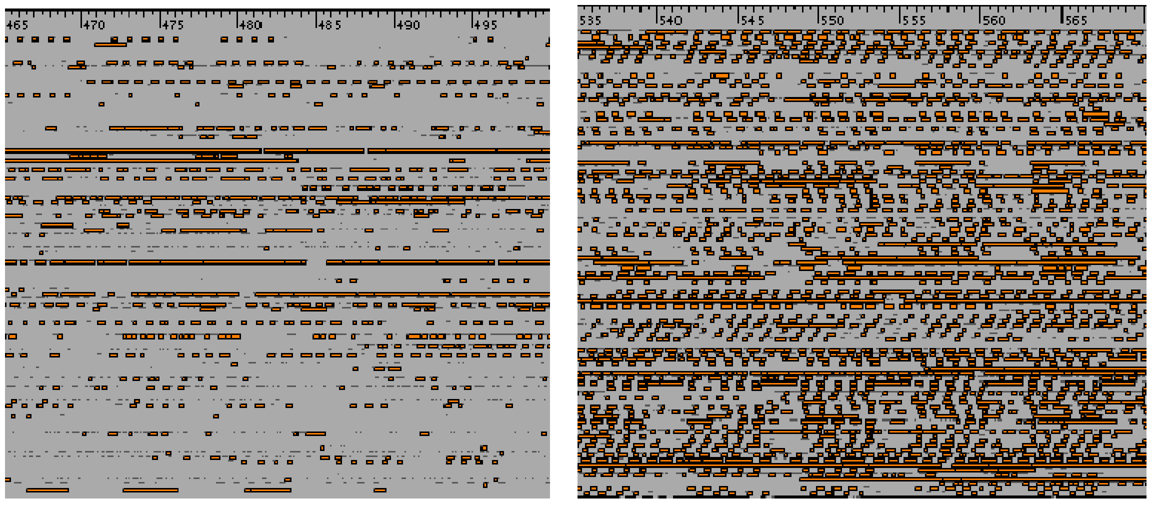


Figure 5.4: Sum of the bit vectors of the hosts of each cluster (blue line), compared to the cluster centroid multiplied by the number of hosts in the cluster (green line).

5.4 Online Availability Prediction

In the previous section, we showed that the bit vectors were an effective representation of the availability of individual resources. These bit vectors were formed using trace data from the entire trace period. In this section, we apply bit vectors for online availability prediction, where the bit vectors are formed online only from historical training data.

Specifically, trace analysis has shown that the bit vector is a good and accurate summary of the availability of hosts over time for a large fraction of hosts, detecting existing daily and weekly patterns.



(a) Trace of a random subsample of the hosts in cluster 1 (always off).

(b) Trace of a random subsample of the hosts in cluster 3 (cyclic)

Figure 5.5: Traces visualized with Pajé.

These patterns are usually repeated over the weeks, with only small punctual variations, or changing over longer periods with seasonal effects. For this reason, we can expect that the patterns detected by the bit vector will also be repeated in the near future. Therefore, we will consider the usability of the bit vector as a predictor.

To predict the behavior of a host, its first weeks of life are used as training data to generate a bit vector. This is done by dividing the week in intervals (e.g. one hour) and measuring the fraction of time the host is available at each interval during all the training period. The values assigned to each interval are then converted to bits, by putting a 1 when the value is over a given threshold (e.g. 0.75) and a 0 otherwise. This vector can then be used to predict the behavior of the host during the next week: the host is expected to be available during the intervals where the bit vector has a 1, and unavailable during the rest of the week, when the bit vector has a 0.

Metrics for prediction quality

We use two metrics to assess the quality of the predictions made by the bit vector. The first one is the $r$ate of false positives: amount of time when the host is unavailable and predicted to be available, divided by the total time the host is predicted to be available. This gives a measure of the reliability of the prediction, i.e., the confidence that a host will be available when the bit vector predicts it to be available. The second metric used is the $r$ate of false negatives, which is similarly defined as the amount of time when the host is available and predicted to be unavailable, divided by the total time the host is predicted to be unavailable. This could be considered a measure of the efficiency permitted by the predictor. A system relying completely in the availability prediction would not try to use hosts

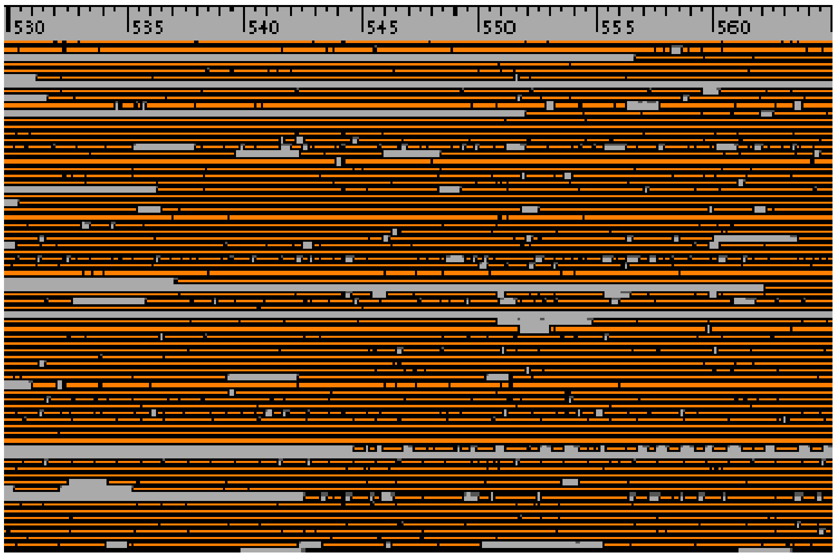


Figure 5.6: Trace of a random subsample of the hosts in cluster 5 (always on) visualized with Pajé.

when they are not expected to be available, and therefore their unpredicted time of availability would be wasted.

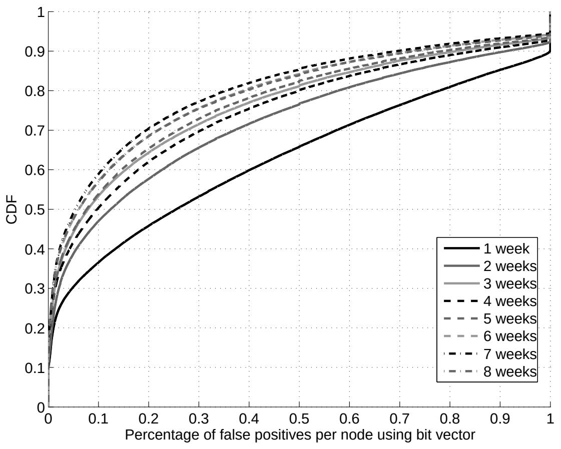
5.4.1 Evaluating Different Parameters for Prediction.

Prediction parameter 1: length of training period

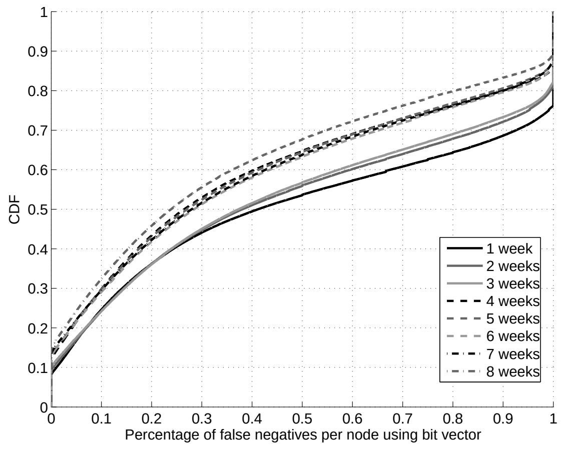
One fact that may determine the quality of the predictions made by the bit vector is the length of the training period. Short periods may not be enough to capture the presence of cyclic patterns. However, requiring very large amounts of data to make the predictions may be a problem in a real system, e.g. because of the space required to store it or because of the time required to obtain it. Figures 5.7(a) and 5.7(b) show the amount of false positives and false negatives obtained when creating the bit vector with different amounts of information, from 1 week to 8 weeks. The threshold used is 0.75, and the interval length used is one hour. These plots do not show the hosts with 0 availability for the false positives. Although these have naturally no false positives, the prediction is useless, because a prediction mechanism would consider them unavailable and therefore unusable. Figure 5.8 shows the amount of hosts with availability predicted as 0 by the bit vector for different number of hosts. Similarly, we have excluded hosts with 100% availability predicted by the bit vector from the false negatives plot.

Between 10 and 20% of hosts have no false positives at all, depending on the training period length (see Figure 5.7(a)). 50% of the hosts, or more, have less than 10% false positives when using a training period longer than 2 weeks. However, not all hosts are this predictable, and there is even an amount as large as 10% in the worst case with 100% error (false positives). There is a large difference between using one week to generate the bit vector and using more, but when using more than 2 the improvements obtained by using one more week are moderate.

Regarding false negatives (see Figure 5.7(b)), however, the biggest difference is between using 3 and 4 weeks as training period. Using 4 or more weeks, 50% of the hosts have less than 30% of false negatives, but even in the best case, there is more than 10% of the hosts with 100% false negatives, and the error is higher than when considering false positives. This is consequent with the fact that our binarization threshold is relatively high (0.75), considerably higher than 0.5, therefore prioritizing reliability over efficiency. Another consequence of the conservativeness of our method is the fact that a high amount of hosts (more than 20% for many configurations) have no predicted availability (see Figure 5.8).



(a) CDF of the false positives.



(b) CDF of the false negatives.

Figure 5.7: Prediction error obtained when using a bit vector (generated with different number of weeks of information and threshold 0.75) to predict the behavior of the next week.

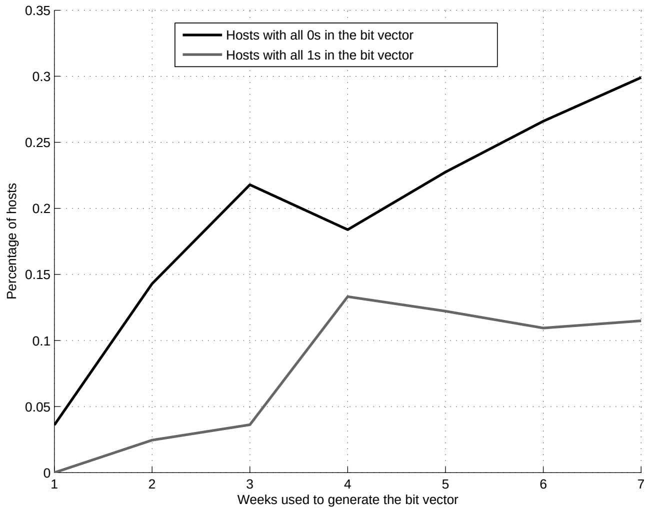
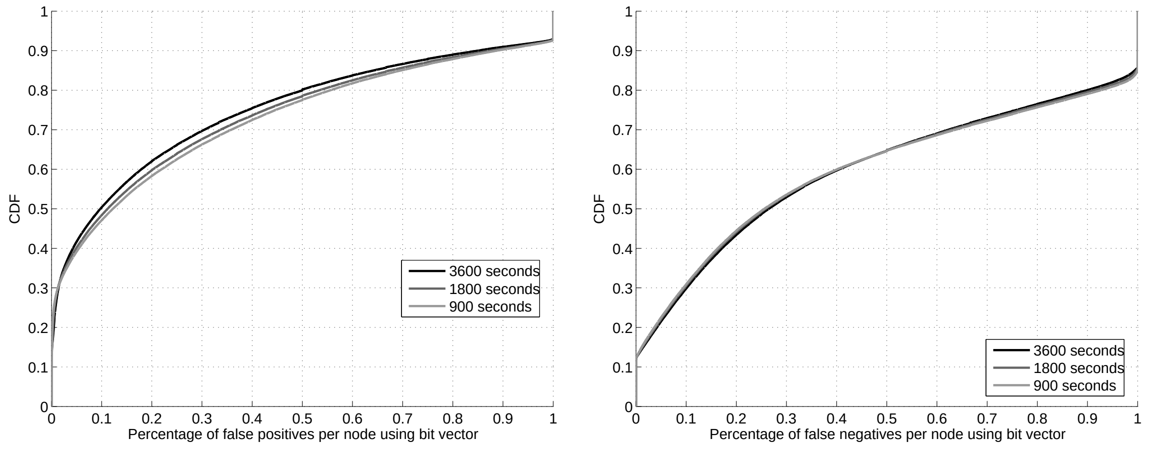


Figure 5.8: Amount of hosts with availability predicted as 0 and 100% with different number of weeks.

Prediction parameter 2: bit vector granularity

Another important factor may be the granularity of the bit vector, i.e., the length of the intervals in which the week is divided. A fine granularity may give more accurate predictions, but it also requires more data in the bit vector. This may be an issue if the data storage space is limited, or if the bit vectors are to be sent through a network. We have considered that one hour is a coarse enough granularity, so we have not tried intervals longer than that. On the other hand, we have considered that intervals as short as one minute would generate a huge amount of data that would make them almost intractable. The intervals lengths we have tested are 60, 30 and 15 minutes. Figs. 5.9(a), 5.9(b) and 5.10 show, however, that there is almost no difference between these granularities.



(a) CDF of the false positives.

(b) CDF of the false negatives.

Figure 5.9: Prediction error obtained when using a bit vector (generated with 4 weeks of information and threshold 0.75) to predict the behavior of the next week, using different binarization interval length (in seconds).

Prediction parameter 3: threshold for binarization

Another parameter we consider is the binarization threshold. Our purpose is to detect patterns in the behavior of hosts, by finding the intervals when the host is almost always available, and those when it is almost never available. In this ideal case, all values would be near 0 or near 1 before binarization, so the threshold would not be important. However, because in the real trace there are many values in the middle ground, we need to set the threshold carefully so that we do not predict availability for intervals when a host is only sporadically available, but we do predict availability when it is due. As expected, the higher thresholds cause a lower rate of false positives, but a higher rate of false negatives (Figs. 5.11 and 5.12). However, there is a larger difference between the thresholds 0.75 and 0.85. This may mean that there are a large number of hosts with an average availability of near 80% during some

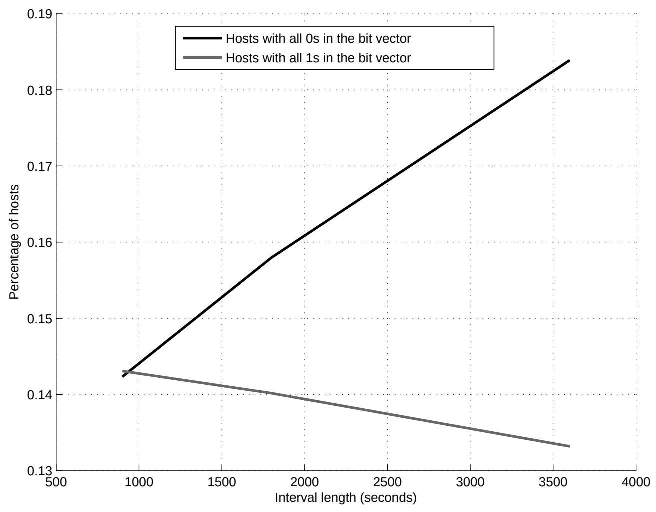
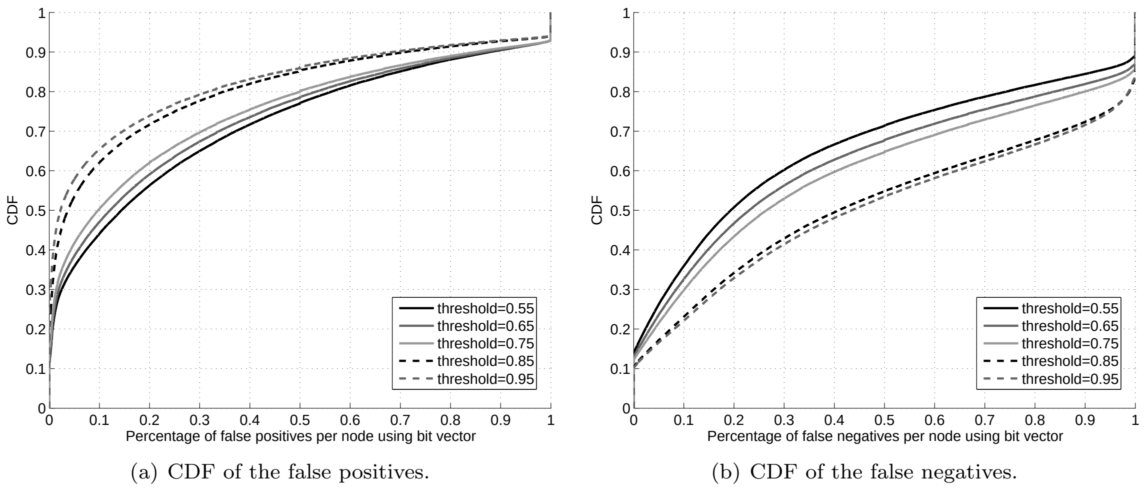


Figure 5.10: Amount of hosts with availability predicted as 0 and 100% with different binarization interval length (in seconds).

hours. These intervals are considered as available when using thresholds equal or lower than 0.75, but unavailable when using a threshold of 0.85. Predicting such intervals as available or unavailable makes a great difference, and could be considered to mark the limit between an "optimistic" and a "pessimistic" prediction.



(a) CDF of the false positives.

(b) CDF of the false negatives.

Figure 5.11: Prediction error obtained when using a bit vector (generated with 4 weeks of information) to predict the behavior of the next week, using different binarization thresholds.

Prediction parameter 4: length of prediction interval into the future

Until now, we have only tried to predict the behavior of hosts during one week using the availability data of immediately previous weeks. In a real system, the bit vector could be generated every week with the latest data and then used to predict the next week. However, if the results of predicting the

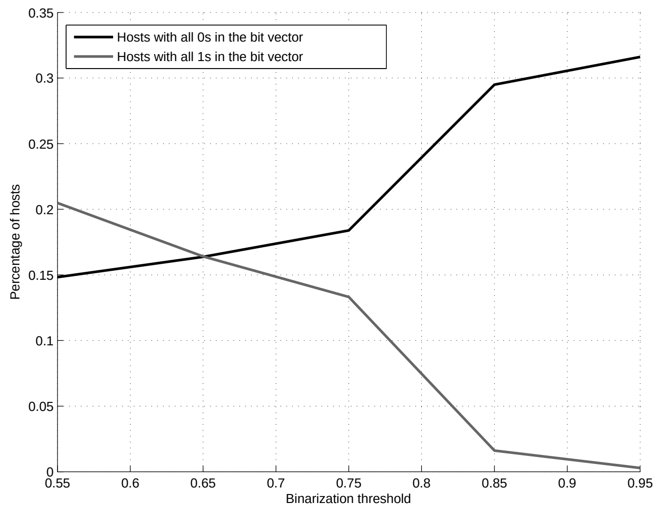
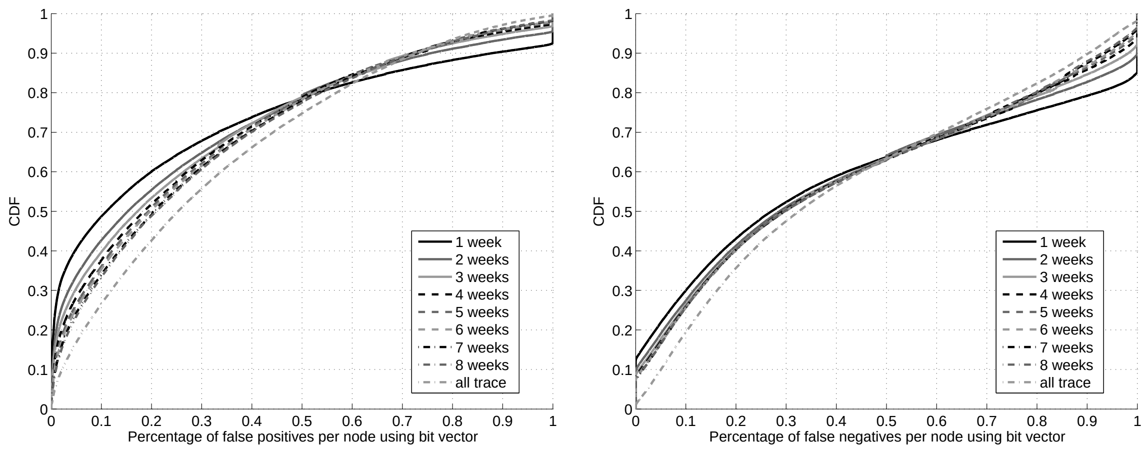


Figure 5.12: Amount of hosts with availability predicted as 0 and 100% with different binarization thresholds.

next N weeks are not much worse than predicting the next N-1, we could say that the prediction does not degrade excessively over time. If this is the case, a system may not need to generate and apply a new bit vector every week.

Fig. 5.13 shows how the quality of predictions degrades when using the bit vector generated with the first 4 weeks of life of each host to predict its behavior during the next 1 to 8 weeks, and during the rest of the trace. The biggest difference in false positives is between predicting 1 and 2 weeks, while the difference when predicting more weeks is smaller. Therefore, it would probably be a good practice to update the bit vector weekly with the latest availability information to preventing the predictions from degrading.

It must be noted that, while the amount of hosts with low prediction error decreases over time, the amount of hosts with high prediction error also decreases. This is probably because these are hosts with a highly random behavior, so when taking more time (more samples) the proportion of extreme values is lower. The degradation may be explained by the effect of seasonal variations, e.g. holidays. This may cause the big difference seen when trying to predict the whole trace, since in such a long period seasonal effects may be more present.



(a) CDF of the false positives. (b) CDF of the false negatives.

Figure 5.13: Prediction error obtained when using a bit vector (generated with 4 weeks of information and threshold 0.75) to predict the behavior of the following weeks.

5.5 Availability-Aware Resource Selection for Service Deployment

The target of this research is to use the prediction of periodical availability patterns to provide collective availability to a service. We will do this through permanently deploying the service in a set of hosts with (negatively) correlated availability patterns. In other words, each service will be deployed in M hosts which will be executing the service whenever they are available (the definition of availability is when the CPU is available for computation, as defined by the user preferences). The service is considered to be available at a given moment if at least N of the total M hosts are available. An alternative application would be the storage of files in the form of erasure codes [55], which requires retrieving N out of M existing fragments to restore the file. However, in this case the definition of availability would be different (host availability would be probably enough, instead of CPU availability).

Host selection for collective availability

A process, called the service deployer (part of the Service Manager in CoDeS architecture), uses the bit vector to decide which hosts to use for the deployment of the service, as it captures the periodical availability patterns of each host. This information is obtained through an independent availability monitor. Each host is expected to be available during the intervals for which it has a 1 in its bit vector. Therefore, the sum of the bit vectors of a set of hosts shows how many of these hosts are expected to be available at each interval.

The service deployer picks hosts randomly, and keeps only those that bring the set closer to the target, i.e., a sum vector with a value equal or bigger than N in each position. It picks hosts until the sum vector of the selected hosts has reached the target. Then it starts a second round where it checks if each of the hosts in the selection is contributing to reach the target. It removes hosts that can be taken out of the selection with the sum vector of the remaining set still reaching the target.

This is a very simple approach, but it has the advantage that its simplicity allows it to be used even in decentralized environments where the deployer has no complete knowledge of the hosts in the system. It clearly gives preference to availability over efficiency, since we will tolerate the assignment of more resources than needed (having values greater than N in some positions). A more complex approach could involve trying to find the hosts that optimize the sum vector, to have at each position a value as close to N as possible.

Predictability

The previous analysis has shown that the bit vector is a good predictor for some of the hosts, but it is not so good for some others. Therefore, host pre-selection may be important for the overall performance of the resource selection mechanism. This means restricting the hosts we consider for deployment, picked before looking at their bit vectors. For example, it would be useful to prioritize the use of more predictable hosts. The most obvious metric for predictability would be the accuracy of a prediction done using the same model. In our case, if we have a trace of W weeks to generate the bit vector, we could generate a bit vector using the first W - 1 weeks of the trace and compute the false positives obtained when comparing it to the last week of the trace. This could give us a measure of how predictable the behavior of the host is. Other metrics we can use to prioritize host selection are average availability, or the clustering done in Section 5.3.

Redundancy

Another tool that can be used to counter the possible lack of accuracy in the prediction for some of the hosts is redundancy. Instead of selecting hosts until reaching a target of N in all the positions of the bit vector, the deployer can artificially add more redundancy by using a redundancy factor R greater or equal than 1, and setting the target as R*N. This way the deployment can tolerate the unexpected availability of some of the hosts in the set. Specifically, it can tolerate at least $(R-1) \times N$ false positives, and one more for each false negative that happens simultaneously.

Detection of host's system departure

Finally, the proposed resource selection mechanism should be complemented with a method to detect permanent departures of hosts. When this happens, one or more new hosts can be selected by the deployer using the same mechanism. This is necessary because our availability prediction method does not predict permanent departures, so only relying on it cannot guarantee durability. Some of the selected hosts may leave the system permanently over time and the service may become unavailable because of this. A simple method to provide durability is to re-evaluate the deployment every week using the most recent availability data to generate a bit vector for each host. Hosts that have permanently left will end up having a bit vector of all zeros, and therefore they will be removed from the deployment. This method would also prevent performance degradation caused by seasonal effects, i.e., variations in the behavior of hosts. Alternatively, hosts that are offline for more than a given period (e.g. one week) could be removed instantly from any deployment.

5.6 Evaluation

5.6.1 Test Environment

Simulation method

We validate the presented resource management mechanism by simulation using the SETI@home traces. Given a certain point in time to be used as the beginning of the experiment, we calculate the bit vector of each host using the previous 4 weeks of traces, as section 5.4 showed it to be a long enough training period. The deployer uses these bit vectors to select a set of hosts to deploy the service as previously explained. Then we look at the behavior of the hosts after the starting point and see if the service achieves the required collective availability during the next week.

We repeat the simulation with three different starting dates, one corresponding approximately to one fourth of the trace, another near half of the trace, and a last one near three fourths of the trace's length. The three starting dates are at Monday at 00:00 hours, so we can make predictions for a whole week. Theoretically, there should be no difference when deploying a service in any other moment of the week. Since there is a random component in our resource selection mechanism, we repeated the simulation 1000 times with each starting point. For each configuration tested, we take the results obtained with the three different starting points (3000 simulations in total) and average the results.

An important parameter of the simulations is the desired number of hosts N. We have set 50 as the default value because, after some preliminary tests, this value was shown to not mask the effects of other parameters in the performance of the mechanism. However, we have also tried different values to see the effects of this parameter on the performance of the resource selection method in Section 5.6.2 .

Reliability metrics

The main metric we use to evaluate the reliability of the deployment is the *achieved availability*, which is the percentage of time in which the number of available hosts is equal or higher than required. An additional metric we measure is the number of required hours of host availability that have not been provided, or *missing host-hours*. It is equivalent to the length of an interval where there are less than N available hosts, multiplied by the difference between the number of required hosts and the available hosts. This is put in perspective by dividing it by the number of required host-hours of availability. We also measure the *success rate*, which is the fraction of simulations where the service is available (i.e., there are at least N available hosts) during the whole week.

Efficiency metrics

We also evaluate the efficiency of our approach by measuring the number of hours of host availability that are above the required. Dividing this by the number of required host-hours of availability gives a measure of the *redundancy* provided by the system. Note that this only measures those periods when the service has more available hosts than needed. The missing host-hours, on the other hand, measure only the periods where there are less available hosts than needed. Therefore, these two metrics are independent and do not compensate each other, as it would happen if we only computed the total number of host-hours attained for the service and compare it to the required number of host-hours ($N \times h$, being N the required number of replicas and h the length in hours of the considered period).

Another more straightforward metric of the performance of the resource selection method is the *number of hosts* that it selects for deployment. Although this is not necessarily related to the provided redundancy (e.g. selecting a number of hosts with low availability can provide lower redundancy than selecting fewer hosts with higher availability), each host selected has a cost in the form of transmitting data to that host. Therefore, selecting a low number of hosts is also a sign of efficiency.

Different predictive methods compared

We have selected two different threshold values to generate the bit vectors used in the simulations: 0.75 and 0.85. The reason is that the results for these thresholds show the biggest difference, as seen in Section 5.4. A third option we have tried is to use the availability vectors without the binarization step. This gives a vector of 168 positions, showing the average availability of the host for each hour of the week. We sum these vectors directly, and stop the deployment when the sum vector is equal or higher than N for all positions. Finally, we have also tried two more methods to compare against our prediction-based resource selection. These are *last value*, which just selects hosts that are available in the moment of deployment, and *random*, which consists of simply picking a number of hosts randomly.

5.6.2 Evaluation Results

Evaluation of Redundancy factor

The first step to compare the performance of these five resource selection methods is to determine the right redundancy factor required for each of them. In the methods based on bit vectors, it is the R that we use to compute the required number $R \times N$ in each position of the sum vector. For last-value and random, $R \times N$ is directly the number of hosts that we select. Therefore, these replication factors are not comparable and must be set separately for each method.

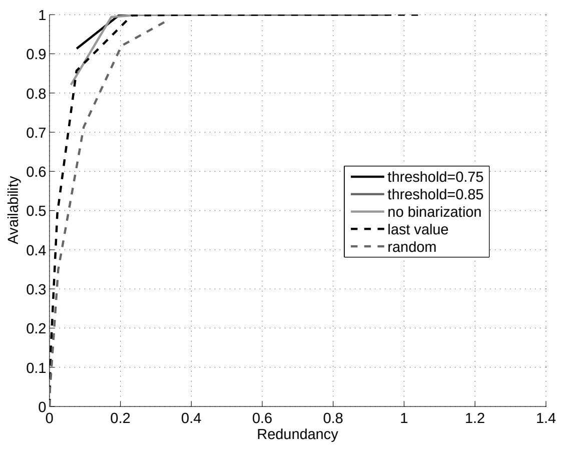
Figures 5.14 and 5.15 show the results obtained by each method with equivalent redundancy levels, in percentage of surplus host-hours and in number of selected hosts. Note that the redundancy shown in the x axis of the plot is a result metric and not a configuration parameter. For these reasons, the different methods do not necessarily have results for the same values on each axis.

Bit vector methods perform consistently better than the rest. Using threshold=0.85 gives 100% availability and success rate in all cases, although the redundancy is higher than required to obtain the same results with the threshold=0.75 (because the redundancy is higher, the line for threshold=0.85 starts more to the right of the figure than the rest and is hardly visible). Using the vectors without binarization gives relatively good results in redundancy, but requires picking a larger number of hosts. *Last value* performs remarkably well, probably because of the starting dates selected: 00.00 at Mondays may be a time when mostly highly available hosts tend to be connected. This is confirmed by the results presented later, specifically in Figure 5.16(d). Finally, *random* performs poorly, requiring the highest number of hosts.

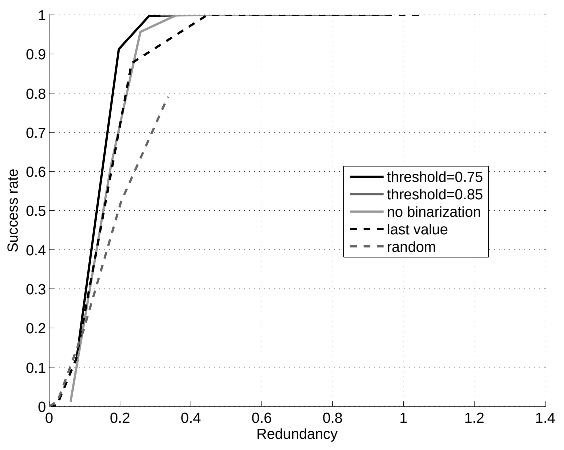
Looking at these results, we select a default value for R for each method. This value should give good results for the method, but it should not be so good (nor so bad) that it masks the effects of changing other parameters. With this in mind, we select R=1.1 for the methods based on the bit vectors, R=1.5 for *last value* and R=2.5 for *random.*

Evaluation using different host subsets

As said before, host pre-selection may have a great impact on the performance of service deployment. Moreover, it would also be interesting to check if our prediction methods could leverage hosts with mid/low availability. Specifically, hosts with cyclic patterns should be useful for our bit vector-based

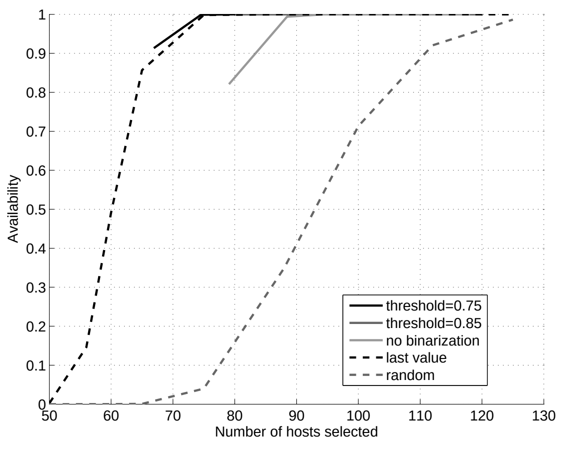


(a) Availability versus redundancy.

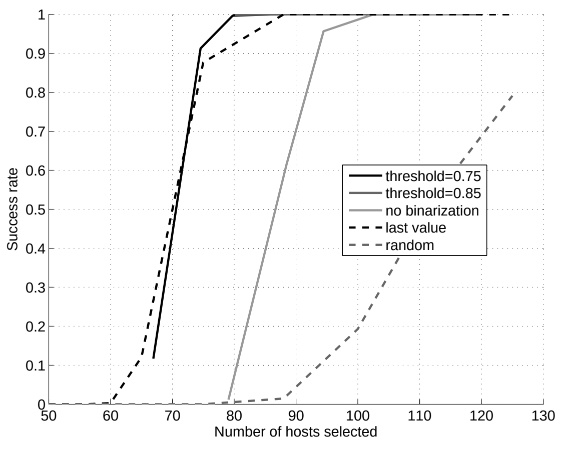


(b) Success rate versus redundancy.

Figure 5.14: Reliability obtained for different levels of redundancy.



(a) Availability versus number of hosts selected.



(b) Success rate versus number of hosts selected.

Figure 5.15: Reliability obtained for different number of hosts selected.

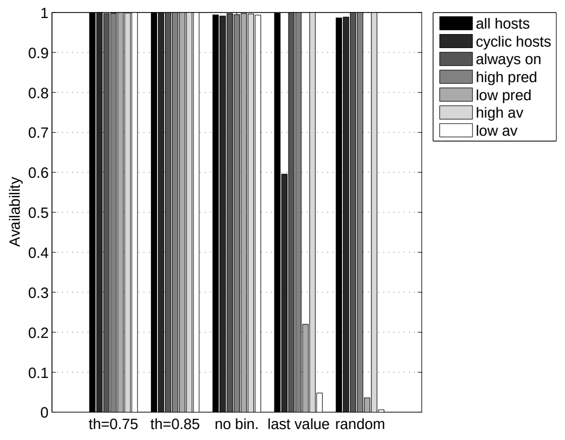
methods. Therefore, we tested the five previously explained resource selection methods with different subsets of the hosts that appear in the SETI@home trace. One division is using the predictability, as explained earlier, to divide the hosts in two subsets: those with high predictability and those with low predictability. Similarly, we have divided hosts in subsets of high and low availability, using the availability reflected in their bit vectors. Finally, we have used the results of the clustering in section 5.3 to separate hosts in always-on and cyclic hosts.

Figure 5.16(a) shows that the bit vector-based methods perform well with any subset of hosts, while last value and random are very sensitive to host pre-selection and perform very poorly in some cases. Surprisingly, however, with threshold=0.75 success rate is lower for the *'always on'* cluster and for hosts with high predictability and high availability (see Figure 5.16(b)). This is nevertheless consistent with the fact that these are the configurations where a lower redundancy is obtained (see Figure 5.16(c)) and where a lower number of hosts is selected (see Figure 5.17). That is because all these pre-selections favor hosts with higher availability (as seen in the bars of Figure 5.16(d) that belong to the random configuration, which only takes a random subset of hosts and is therefore representative of each host set). This allows the bit vector-based methods to make a deployment with low redundancy, as many hosts have 1 in all or most of the positions of their bit vectors, and therefore a number of hosts close to $R \times N$ causes the sum vector to have $R \times N$ or more in all positions. On the contrary, using cyclic hosts or hosts with low availability requires more complex combinations of hosts, and causes a higher redundancy. This shows specially when using threshold=0.85, which obtains an extremely high availability at the expense of high redundancy.

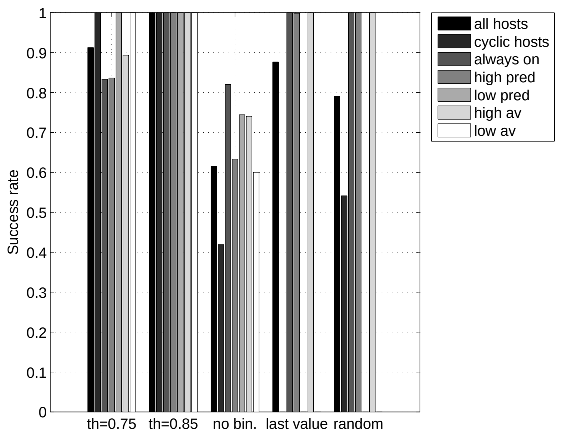
Evaluation for different sizes of host group

Figure 5.18(a) shows availability obtained for different numbers of required available hosts (N). The availability is high (over 0.9) in every case, but it is better with higher values of N (over 0.999 for all methods with N=1000). However, the bit vector-based methods have very good results even with N=10 (availability higher than 0.999 with threshold 0.85, and higher than 0.98 with threshold 0.75).

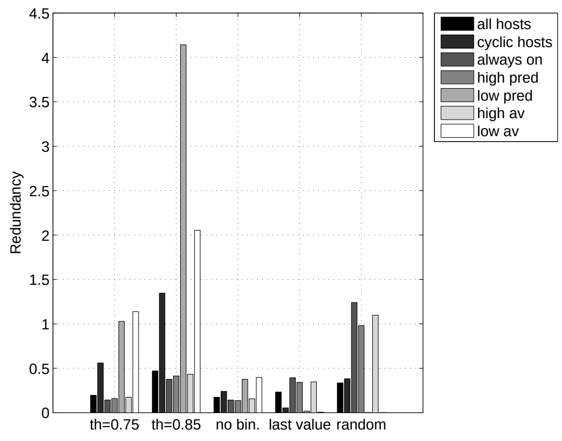
Success rate, however, is much lower with smaller values of N (see Figure 5.18(b)), and in general all methods perform better with higher values of N, having less missing capacity (see Figure 5.18(c)), and also less redundancy (see Figure 5.18(d)) in the case of vector-based methods. This may be caused because of random perturbations in the behavior of hosts, that affect more the smaller deployments as stated by the Law of Large Numbers. To counter this effect, a higher redundancy factor should be set for small deployments in a real system.



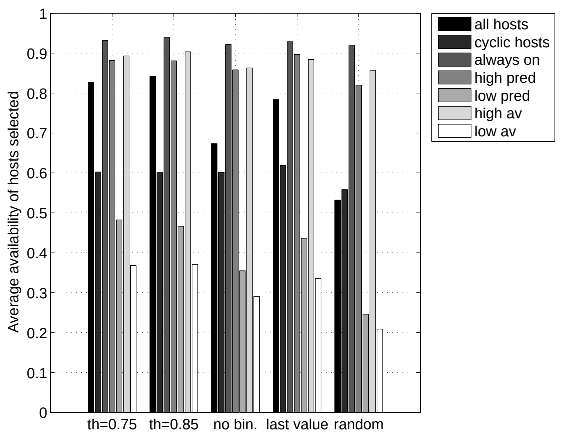
(a) Availability versus host pre-selection.



(b) Success rate versus host pre-selection.



(c) Redundancy versus host pre-selection.



(d) Average availability of selected hosts versus hosts pre-selected.

Figure 5.16: Reliability and efficiency metrics for different host pre-selection policies.

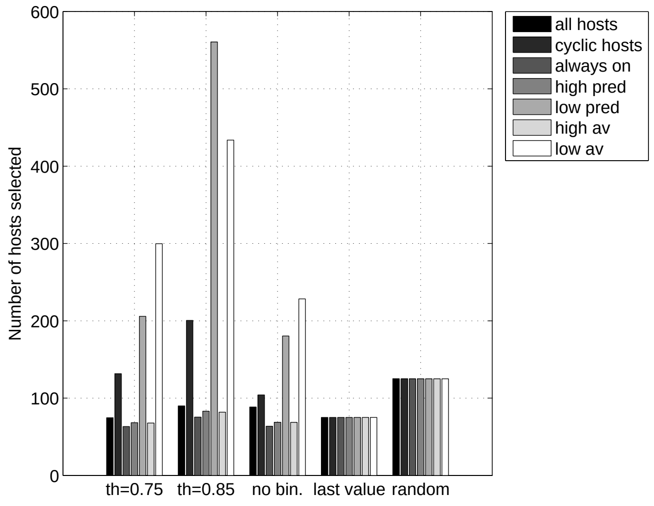


Figure 5.17: Number of hosts selected versus host pre-selection.

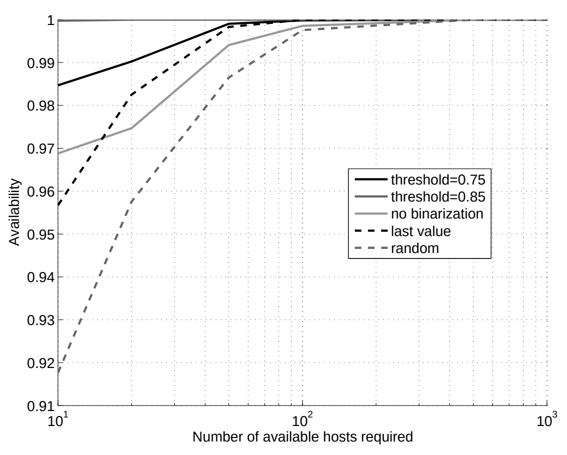
Evaluation for different lengths of predicted intervals

The predictions made by the bit vector degrade over time, as seen in section 5.5. It would be therefore advisable to re-evaluate a deployment periodically using the most recent availability data to generate a new bit vector for each host. To see how frequent such a process should be, we have tested deployments over longer periods, up to 4 weeks.

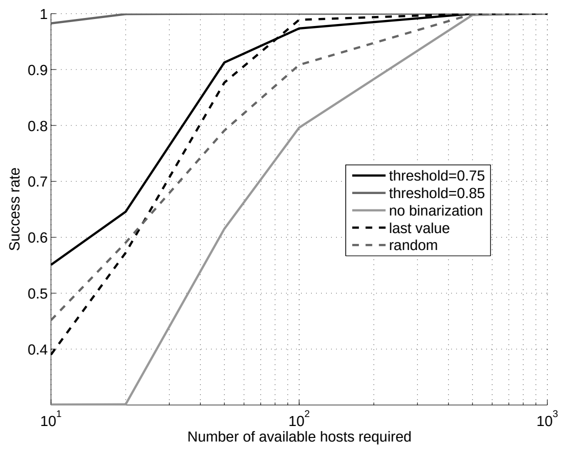
Average availability (see Figure 5.19(a)) decreases very slightly over the weeks with the vector-based methods (except for threshold 0.85, which maintains a total availability), while it does degrade more for the *last value* method. Figure 5.19(c) confirms that only a small amount of host-hours is missing over the course of the simulation, and redundancy is decreased accordingly (see Figure 5.19(d)). Success rate, however, suffers a deeper degradation (see Figure 5.19(b)). Considering the whole picture, we must note that success rate is the number of times when the service is available during the whole simulation (of a total of 3000 executions for each configuration) and therefore a single second of failure during the simulation can drastically affect its value. Low success rates may therefore be simply caused by the higher failure probability over a higher period of time, more than the degradation of predictions. Surprisingly, though, *random* suffers the smallest degradation, with only a slight decrease in success rate, and actually increasing its availability, although very slightly. The cause may be the high redundancy of *random* deployment, only second to threshold 0.85 (see Figure 5.19(d)).

Comparison with dynamic deployment and short-term prediction methods

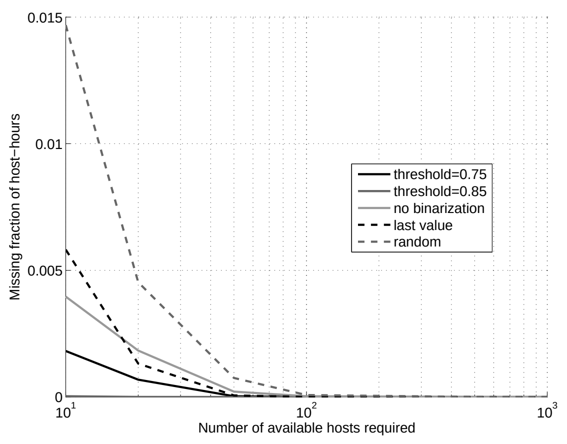
The purpose of our resource selection method is to leverage knowledge of cyclical patterns and general long-term behavior of hosts to provide high levels of collective availability while minimizing the



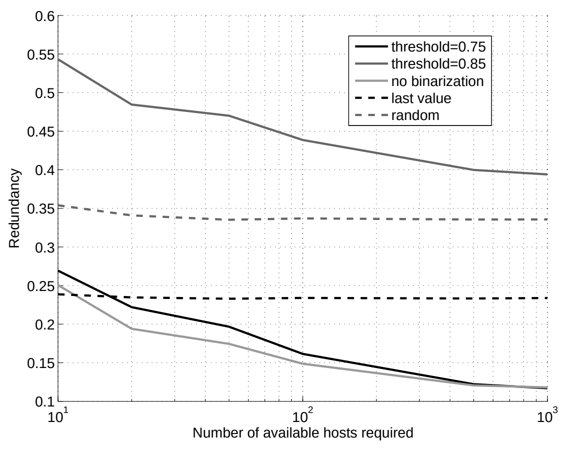
(a) Availability obtained for different numbers of required hosts.



(b) Success rate for different numbers of required hosts.



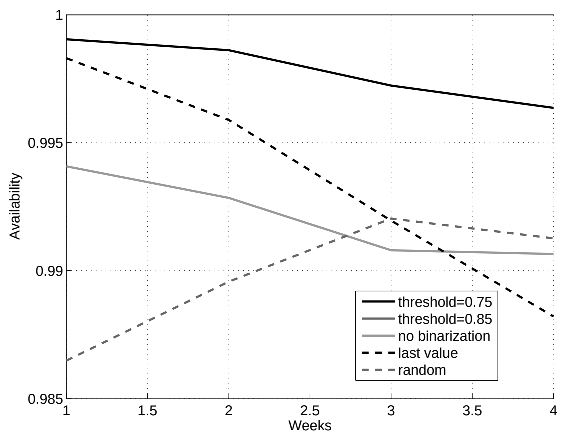
(c) Fraction of missing host-hours for different numbers of required hosts.



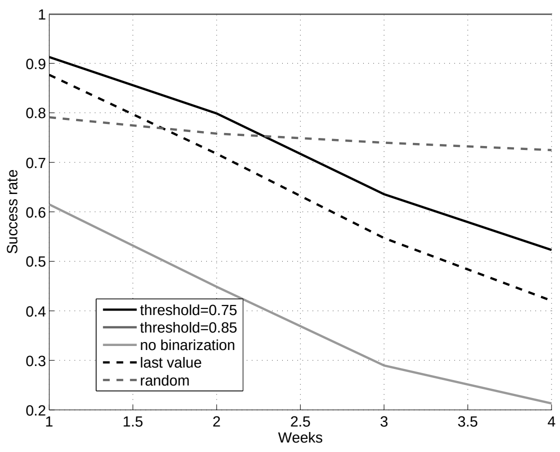
(d) Fraction of surplus host-hours for different numbers of required hosts.

Figure 5.18: Reliability and efficiency metrics for different numbers of required hosts.

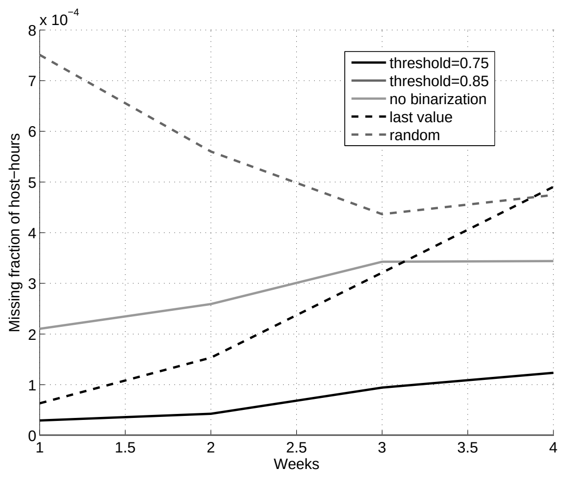
communication costs (which is partially determined by the number of migrations). Until now, we have considered that the system only performs the initial deployment, and works with the availability that it provides. However, it could combine permanent assignments with temporal assignments, used by default in CoDeS' basic service deployment mechanism, to increase availability: when the number of hosts becomes n below the required N, the deployer could select N-n hosts that are available at the moment and temporarily deploy the service in them. When the number of hosts available in the permanent deployment set is again equal or greater than N, the hosts that were used for temporary deployment stop executing the service. Note that in some cases, if the unavailability period is short, it may not be efficient, or even useful, to perform a temporary deployment. However, we will consider that the deployer always performs temporary deployments when required, and estimate the total required number of deployments (initial + temporary).



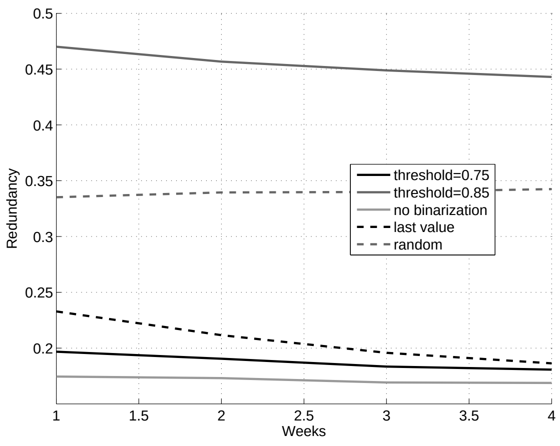
(a) Availability obtained during different number of weeks after service deployment.



(b) Success rate in different number of weeks after service deployment.



(c) Fraction of missing host-hours in different number of weeks after service deployment.



(d) Fraction of surplus host-hours in different number of weeks after service deployment.

Figure 5.19: Reliability and efficiency metrics measured in different number of weeks after service deployment.

We perform an optimistic estimation. First we count the number of violation intervals, i.e., the time between the point when a host disconnects and the point when that or another host comes back and returns the total number of available hosts to the previous level. There can be different levels of embedded violation intervals, e.g. a host disconnects starting an interval with N-1 hosts; then another one leaves and starts an *embedded* interval with N-2 hosts; one host becomes available again and ends the period of N-2 hosts; finally, another host arrives and ends the interval with N-1 hosts. We consider that each of these intervals requires looking for a host outside of the set assigned to the service and perform a temporary assignment, which will end when the interval, as originally computed, ends. This estimation is optimistic because it does not consider the possibility that the host selected for temporary deployment may disconnect before the interval ends. However, since unavailability intervals are often short in our tests, we consider that this is a reasonable assumption.

We perform this estimation with all the previously used methods, and compare it to the dynamic deployment approach with random resource selection and replacement that is the default in CoDeS. As stated in Chapter 3, it consists of randomly choosing a set of N hosts to deploy the service. It does not use permanent assignments, and therefore whenever a host of the set becomes unavailable, a migration is required. When that happens, another host is chosen randomly among those available at the moment. Its availability could be improved by adding a replication factor. However, we will consider that it only deploys the service in N hosts.

We also compared our approach to [13], where a short-term prediction is used to achieve collective availability for a service during a given prediction interval, with a typical length of a few hours. The obtained turn-over rate, i.e., the amount of hosts that need to be migrated from one prediction interval to the next one to keep guaranteeing the required availability, is about 2% of the total assignment in average, when using high predictability hosts, with a prediction interval length of 4 hours. Note that this total assignment may include a certain redundancy (from 0, i.e., no redundancy, to 0.5 of the required number of hosts) to increase the probability of providing the required availability. This means that the total number of deployments (counting both initial deployment and migrations), with zero redundancy, normalized to the required number of available hosts, would be $1 + t \times i$, being t the turn-over rate and i the number of prediction intervals. We have taken the case of zero redundancy to compare our method to the best results of short-term prediction in number of deployments.

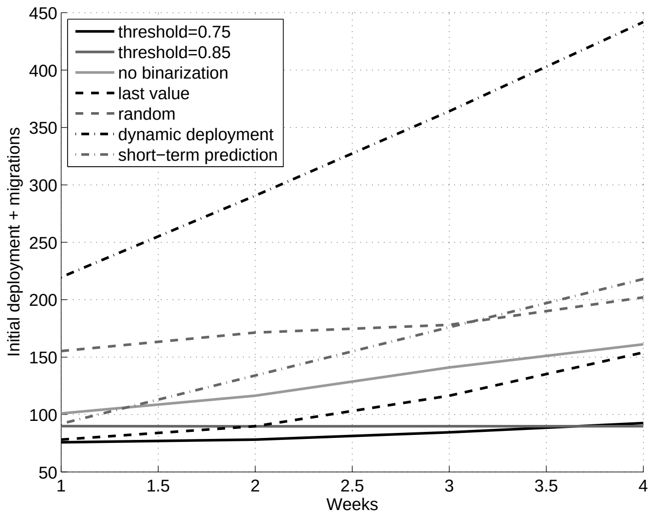


Figure 5.20: Average number of migrations in different number of weeks after initial service deployment.

Figure 5.20 shows that the number of migrations required over time increases linearly, as expected, for the methods based on temporary deployments (with and without prediction). After one week, the short-term prediction-based method has performed nearly the same number of deployments than the bit vector-based with threshold 0.85. However, the latter does not perform any temporary deployments, so its value is constant over time. After 4 weeks,the low turn-over rate of the short-term prediction-based method ends up causing more deployments than the random policy.

Note that we did not compare the actual churn (sum of all node entrance and departure events, including temporal disconnections and returning nodes) inside the set of replicas of the service. While in the random replacement and short-term prediction-based methods the number of migrations equals the number of entrances and departures, it is not so in the bit vector-based method, where hosts can leave and return without triggering a migration. A returning node may need to synchronize its state with the other replicas. If the cost of these synchronization actions is high, then the churn level caused by temporal disconnections may be to expensive for the service.

In the basic random resource selection with replacement, as well as in the short-term prediction-based method, each departure implies a migration. Therefore, the cost of a disconnection is the sum of the cost of deployment (transferring the files required for execution) and the cost of a synchronization. Only the latter applies when a node returns in the bit vector-based method. Therefore the exact trade off between the relative costs of churn in the different resource selection mechanisms compared would depend on the specific service being deployed and its migration and synchronization costs.

5.6.3 Summary of Results

After evaluating the proposed mechanism under different configurations, our results show that, with threshold=0.75, our method provides a higher availability than the other approaches, and it does so while causing less redundancy. The mechanism can be used to select hosts with only local information (i.e., the availability information of each host of a random subset of hosts) and achieve a good collective availability for a service. Moreover, the required computations are simple and the mechanism adapts to different types of hosts behavior, automatically selecting more nodes when their expected availability is low. Other mechanisms, like *random* or *last value*, require that the replication factor is manually adapted to the expected availability of hosts.

Using a threshold of 0.85 gives extremely high availability guarantees, even during long periods of time, but at the cost of a higher redundancy. This trade off may be adequate for those services that are critical to a community, which may well be worth the use of extra resources in order to guarantee their availability. Moreover, a community may fine-tune the replication factor used in their deployments to whether increase the availability provided by the mechanisms or decrease the redundant resource usage.

It would also be advisable to periodically check the deployments using the latest availability information, especially when using 0.75 as a threshold. However, comparison of our long-term prediction-based techniques with an existing short-term prediction technique has shown that our method can obtain similar availability levels with a lower number of deployments and migrations over time. These results justify the use of long-term availability prediction for service deployment in contributory communities and other similar cases, like distributed data storage, instead of short-term prediction, which may be more suited to distributed task scheduling.

5.7 Availability-Aware Resource Selection in CoDeS

5.7.1 Requirements

Our target is to extend CoDeS so that it can deploy services using knowledge about past and/or future host availability, to be able to provide some level of probabilistic service availability guarantees. For this purpose, we have developed a novel technique for availability-aware resource selection in service deployment. The next step is to integrate it into the middleware.

However, we must acknowledge that other resource selection mechanisms may be more adequate for resource selection when deploying some services, and new ones can be later developed that improve the results obtained by our proposal. For instance, the simple random resource selection policy used in the service deployment mechanism as presented in Chapter 3 may offer a lower churn in the set of nodes that host a service than the bit vector-based approach. While the predictive method can maintain an (almost) fixed set of hosts and therefore offers a lower number of migrations, the non-predictive method may cause less changes in the set of hosts that remain available, because hosts with longer availability intervals will be chosen over time [74]. For some services, the cost of migrations may be more important and therefore the bit vector mechanism is preferred. However, other services may have a high cost for dealing with churn, for instance requiring long and expensive synchronization processes, and therefore random host selection with replacement would be preferable.

In consequence, we want to modify CoDeS to allow the use of different resource selection mechanisms. The users must have the freedom to select which is the preferred way to deploy each service, according to its specific trade-offs between performance, availability and use of resources. Following CoDeS' design philosophy, the architecture must be extensible so that different resource selection algorithms and policies can be plugged into the system to guide its behavior.

A first extension to the previous model that needs to be incorporated is the addition of *permanent assignments*: when a service is assigned to a node, the node will execute it as long as it is available. If the host fails, it will restart the execution of the service after it recovers. This will co-exist with the previously defined model of *transient assignments*, where a host that recovers after a failure is expected to keep no previous state, and therefore does not resume the execution of whatever services it had assigned in its previous session. Different resource selection mechanisms will make use of the different types of assignment, or combine both.

The Service Manager needs to be able to execute some of its actions guided by definable policies. This will require that such actions are extracted from the basic behavior of the Service Manager itself. They must be guided by policies that can be dynamically loaded and selected, so that different services can use different policies.

The actions we identify that need to be modified by selected availability policies are the following:

- ***initial deployment:*** selecting the hosts that will be used to deploy the service may depend on many factors. The specific hosts selected may be chosen because of their availability, their predictability or other characteristics. The number of hosts that will be used may also depend on such factors. Therefore, the set of hosts used for the initial deployment must be guided by the specific availability-aware policies used. These will require the knowledge of the number of available replicas that are desired by the user, as well as the resource requirements of the service that must be used when selecting the hosts. Other requirements from the user (e.g., availability guarantees) may also be provided in the service specification and passed to the policy.
- ***recovery actions:*** this refers to the actions that must be taken during the life of the service to make sure that it remains available. Different policies may consider different factors to trigger their actions. However, these actions will always consist of creation, destruction or migration of service replicas. There are two types of recovery actions:
  - *proactive:* actions that are taken to prevent failures. This must be taken periodically or when certain conditions occur. Therefore, the communication between this policy and the Service Manager (SM) must be bidirectional: the SM must invoke this policy periodically with the data about the current state of the hosts, and the policy may act upon the SM triggered by other events. Each policy may implement its own sensors to trigger proactive recovery actions.
  - *reactive:* actions that are taken to go back to a correct state after a failure has happened. This is the case of replica replacement that takes place when a host disconnects. This must be invoked by the SM every time an event happens, e.g. the disconnection of a replica.

The resource selection policies will require monitoring the availability of hosts. Again, this must be a separate and completely extensible part of the proposed architecture. There are two reasons for this. The first one is that the monitoring of a decentralized system is a complex problem which has motivated its own research studies [105] and will surely keep doing so. Therefore, we must be ready to incorporate new monitoring techniques that may be proposed in the future which could further improve the capacities and performance of CoDeS.

The second reason is that different policies will require different availability information. They may require different amounts of data (e.g. global vs partial), and different types (e.g. host availability vs CPU availability). Some policies may also require specific searching mechanism (e.g. find the most available hosts), while others may only need to pick random hosts and find their availability.

For these reasons, each policy will define a set of requirements for availability monitoring, and assume that we have a mechanism that satisfies them. This leaves availability monitoring out of the focus of this thesis. However, Section 5.7.4 will further discuss the possible approaches in the implementation of this part of the system.

5.7.2 Architecture Extensions Overview

In order to implement the availability-aware resource selection policies, we will add some new components to the previously presented architecture of CoDeS. These are the Availability-Aware Assistant (A3) and the Resource Availability Monitor (RAMon). Figure 5.21 shows their interaction in the process of deploying a service. The basic process is shown in Algorithm 5.1, marking the correspondence between the methods in the algorithm and the steps in the figure.

5.7.3 Availability-Aware Assistant (A3)

This component implements the policies that guide resource selection for service deployment. Different implementations may exist, which implement different sets of policies. The description of the service must contain a field specifying the policies to be used, and the fitting A3 will be dynamically loaded by the Service Manager (SM).

The SM will invoke the methods of the A3 in the cases previously stated: when initially deploying the service, when hosts in the replica set change their state (connect or disconnect), and periodically. In all these cases, the SM will pass the current state of the service (hosts where it is deployed and their current availability state) as well as the service specification (availability guarantees required, needed number of replicas, host requirements, etc). The A3 will then decide which are the required actions.

In the case of initial deployment, the information passed by the SM will be used by the A3 to find suitable resources to deploy the service, using both the Resource Discovery (RD) module and the RAMon. Depending on the policy, this may happen in two ways: whether (1) the A3 gets from the RD

Algorithm 5.1 Behavior of service deployment using an example initial deployment policy

```
    SERVICE MANAGER

 1: procedure DEPLOYSERVICE($numReplicas, policyType, policyInfo, hostReqs$)
    //(Step 1)
 2:     $A3 \leftarrow loadAvailabilityAwareAssistant(policyType)$
 3:     $selectedHosts \leftarrow A3.initialDeployment(numReplicas, policyInfo, hostReqs)$
    //(Step 2)
 4:     $deployOnHosts(selectedHosts)$ //(Step 6)
 5: end procedure

    AVAILABILITY AWARE ASSISTANT

 6: function INITIALDEPLOYMENT($numReplicas, policyInfo, hostRequirements$)
    // This is an abstract method. We show here an example implementation,
    // using a Resource Availability Monitoring module that retrieves
    // availability information of a set of specified hosts.
 7:     $selectedHosts \leftarrow \emptyset$
 8:     return $deploy(numReplicas, policyInfo, hostRequirements, selectedHosts)$
 9: end function

10: function DEPLOY($numReplicas, policyInfo, hostRequirements, selectedHosts$)
11:     $neededHosts \leftarrow decideNeededHosts(numReplicas, policyInfo, selectedHosts)$
12:     while not $validDeployment(selectedHosts, policyInfo)$ do
13:         $potentialHosts \leftarrow ResourceDiscovery.findHosts(neededHosts, hostRequirements)$
    //(Steps 3 and 4)
14:         $availabilityInfo \leftarrow RAMon.getInfo(potentialHosts)$
15:         $validHosts \leftarrow getValidHosts(potentialHosts, availabilityInfo, policyInfo)$
16:         $selectedHosts.add(validHosts,$ "transient"$)$
17:         $neededHosts \leftarrow decideNeededHosts(numReplicas, policyInfo)$
18:     end while
19:     return $selectedHosts$ //(Step 5)
20: end function
```

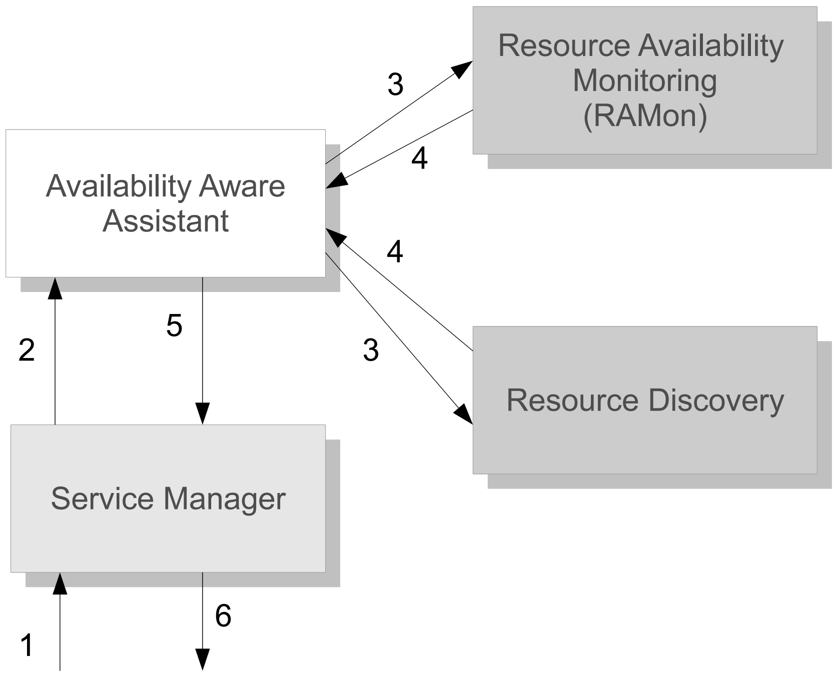

Resource Availability Monitoring (RAMon)
Availability Aware Assistant
Resource Discovery
Service Manager
1
2
3
4
4
3
5
6

Figure 5.21: A view of the modules that form the Availability-Aware CoDeS subsystem.

a list of potentially suitable hosts (those that meet the resource requirements), and gets the availability information associated to them from the RAMon to select which to use for deployment, or (2) get a list from RAMon of hosts which meet specific availability criteria. The latter requires a complex search engine to be present in the implementation of RAMon. Different possibilities for implementing RAMon are described in Section 5.7.4. Once the A3 has a list of hosts, it will decide which to use according to its policies.

If there is a previous selection of resources hosting the service, the A3 will first evaluate the current deployment to decide if changes in the membership are required. If hosts need to be added to the deployment, the A3 will find them as explained in the previous paragraph. If hosts need to be removed, they will be selected according to the specific policies implemented by the A3.

The A3 returns a list of hosts to the SM that are to be used to deploy the service. The SM compares this list with its previous list of service replicas, if it has one. Then, it proceeds to contact the hosts that need to start or stop executing the service, until the current list of service replicas matches the list returned by the A3. The list must also include the type of deployment that will be done in each host (transient or permanent). This is taken into account by the SM to know whether a failed host must be taken out of the current list of replicas (if the assignment was transient) or not (if it was permanent).

This is implemented in the SM by keeping two lists. One contains the transient replicas, which are removed from the list when the SM gets notified of their failure. The other list contains permanent replicas, which include their current state (available or unavailable). The state of each replica is modified by the SM according to the events received from the Remote Execution Module, but their membership is only modified by the A3. The current number of available replicas is the sum of the available permanent replicas plus the transient replicas. All failure events are nonetheless reported to the A3.

5.7.4 Resource Availability Monitoring (RAMon)

Different resource selection policies will have different needs of information regarding host availability. For example, the basic resource selection mechanism presented in Chapter 3 does not require any availability information. Therefore, it would not need to contact RAMon, and would only use information returned by the RD module instead.

Other availability-aware resource selection policies, however, would need some information about host availability. One example is the method presented in Section 5.5. This method requires knowing some availability of each host used in the deployment. It can work with a set of random hosts that fit the requirements, as returned by the RD module [90], and use the availability information to decide if it keeps each of the hosts or not. Therefore, the functionality that the A3 would require from RAMon to implement this policy is to just return the availability information of each host of a list (retrieved previously by the RD module).

Other resource selection policies may require different sets of availability information. For example, they may require the average host availability to compute the number of hosts that is required to provide the desired availability probability, as is done in some distributed storage systems [27]. In this case, the A3 would need RAMon to compute or estimate this average. In some other cases, the policy may require hosts with certain availability characteristics (e.g., high availability). For this purpose, RAMon would need to be able to search for hosts in the system that satisfy these requirements.

It is clear that the functionality that a particular set of policies (and therefore an A3 implementation) requires from RAMon may vary vastly. In consequence, RAMon implementations may also be wildly different to each other. Some, like the case where only availability information of a given set of hosts is required, may be implemented as a local module that contacts the hosts and asks them for their availability history. A more complex approach, where hosts do not self-report their availability, would require a set of monitors distributed in the network that store the information regarding each host. RAMon would then contact these monitors to retrieve the information of the hosts. The other policies mentioned earlier would also need distributed mechanisms to aggregate information (to get the average availability) or to search hosts with specific characteristics.

Therefore, the resource selection policies that can be used in a community will greatly depend on the RAMon implementation that is deployed in it. Because different RAMon implementations may also have different effects on the performance of the system (e.g., complex distributed implementations will have higher communication costs than simple local implementations with limited functionality), the community must choose carefully which A3 and RAMon implementations to support. Different RAMon front-ends and interfaces can be defined for compatibility with different policies and A3 implementations, each offering a subset of the functionality related to availability monitoring required by one or more policies. Therefore, different RAMon implementations may implement different subsets of such interfaces.

5.7.5 Example Implementation

To further clarify how the Availability-Aware CoDeS subsystem works, and how it can be used to implement different resource selection mechanisms and policies, we present two small examples adapting both the method presented in Section 5.5 and an existing availability-aware resource selection method for service deployment to this architecture. The latter method is one presented by Andrzejak et al. [13], where short-term availability prediction is used to guarantee collective availability for a service.

We will show how to implement these two mechanisms using the presented Availability-Aware CoDeS architecture. We will first present the short-term prediction-based method, because it is simpler and matches the simple implementation shown in Algorithm 5.1. The bit vector-based method, on the other hand, requires some changes to the basic implementation of A3 presented earlier.

Short-term prediction-based resource selection

The first resource selection method [13] we will integrate into CoDeS uses a Bayesian classifier together with a set of training data for each host to predict if the host will be available or not during the next prediction interval. Prediction interval length is fixed by default at 4 hours. The method as originally defined includes a classification of hosts into two categories, for high predictability and low predictability. In this example implementation we will omit that part for simplicity, and use all the hosts.

The method selects hosts that are predicted to be available during the prediction interval. The number of hosts selected is defined by the required number of replicas of the service, N, and a redundancy factor, R, specified separately. The total number of replicas that are deployed is $R \times N$. After the prediction interval ends, if the service is needed to be available longer, the prediction is repeated for the hosts in the deployment using updated data. If some of them are expected to be unavailable, they are removed from the deployment and more hosts are picked until the total reaches $R \times N$.

We will implement this policy in the A3. Since it only needs to know the availability information of a subset of the hosts in the community that will be used for the deployment, the RAMon implementation can be a simple module that contacts the hosts and returns their availability history, as reported by themselves. More complex implementations could address the problem of trusting the self-reported history by adding distributed monitoring.

The basic implementation of A3 is the same that appears in Algorithm 5.1. Algorithm 5.2 shows the implementation of the remaining functions of A3 not shown in the previous code. The only recovery action required by this method is to renew the deployment after every prediction interval. The implementation of this policy is also shown in Algorithm 5.2

Algorithm 5.2 Example implementation of the Availability-Aware Assistant using a short-term availability prediction method

```
function decideNeededHosts($numReplicas, policyInfo, selectedHosts$)
    $redundancy \leftarrow policyInfo.get($"redundancy"$)$
    $currentHosts \leftarrow selectedHosts.size$
    $totalHosts \leftarrow numReplicas \times redundancy$
    $neededHosts \leftarrow totalHosts - currentHosts$
    return $neededHosts$
end function

function isValidDeployment($selectedHosts, policyInfo$)
    $redundancy \leftarrow policyInfo.get($"redundancy"$)$
    $currentHosts \leftarrow selectedHosts.size$
    $totalHosts \leftarrow numReplicas \times redundancy$
    $valid \leftarrow (totalHosts \leq currentHosts)$
    return $valid$
end function

function getValidHosts($potentialHosts, availabilityInfo, policyInfo$)
    $validHosts \leftarrow \emptyset$;
    for all $h$ in $potentialHosts$ do
        $hostAv \leftarrow availabilityInfo.get(h)$ //gets the history of host $h$
        if $bayesClassifier.predict(hostAv) = 1$ then
            $validHosts.add(h)$
        end if
    end for
    return $validHosts$
end function

function renewDeployment($numReplicas, policyInfo, hostRequirements, currentHosts$)
    $availabilityInfo \leftarrow RAMon.getInfo(currentHosts)$
    $selectedHosts.set(getValidHosts(currentHosts, availabilityInfo, policyInfo)$, "transient"$)$
    return $deploy(numReplicas, policyInfo, hostRequirements, selectedHosts)$
end function
```

Bit vector-based resource selection

Implementing our bit vector-based resource selection method requires using both permanent assignments, which will be the default type in this mechanism, and transient assignments, which will be only used in case of failure, when the number of replicas of the service falls beneath the target.

The initial deployment of a service follows the mechanism explained in Section 5.5. The A3 will return a list of hosts to the SM, and the service will be permanently deployed on these nodes. This deployment can then be reevaluated periodically, by obtaining the updated availability data of each host and recomputing the sum vector. If the new sum vector is below the target, the resource selection mechanism is used again, the only difference being than in the first iteration the set of selected hosts is not empty.

When a replica failure is reported to the SM, it informs the A3, which will only react if the current number of available hosts in the selected set is lower than the required number of replicas. In this case, a random host (fulfilling the requirements) is selected to perform a transient deployment. This assignment may be later canceled if the number of available replicas gets back to acceptable levels.

Algorithm 5.3 shows the behavior of the A3 that implements the bit vector-based resource selection mechanism. The presented `deploy` function is called from both the initial deployment, as seen in Algorithm 5.1, and the periodical deployment reevaluation. Algorithm 5.4 presents the method that is called by the Service Manager when one of the hosts executing the service disconnects.

5.8 Conclusions

Host availability prediction is critically important for increasing system availability and reliability as well as efficiency. This is especially true in systems based on non-dedicated resources, like peer-to-peer systems and volunteer computing. In such cases, communication costs associated with migration can be too high given the bandwidth of end hosts. Availability prediction can help reduce these costs.

In this chapter, we have applied these principles to CoDeS. We have presented a method for long-term availability prediction, and put it in practice in an availability-aware resource selection mechanism for service deployment. This mechanism has been inspired by the analysis of host availability in real system traces, and put to test using these same real data. We have also presented extensions to the CoDeS architecture that allow this and other availability-aware resource selection mechanisms to be incorporated into the middleware.

Our simulations have shown that using the bit vectors to deploy a service by choosing a permanent set of machines to host it can provide high availability with relatively small redundancy. We have

Algorithm 5.3 Behavior of A3 for the bit vector-based resource selection method. The method is called for both initial deployment and periodical reevaluation.

```
1: function DEPLOY($numReplicas, policyInfo, hostRequirements, selectedHosts$)
2:     $bitvectors \leftarrow RAMon.getInfo(selectedHosts)$ //The returned availability
   // information is a set of bit vectors, corresponding to the passed set of hosts.
3:     $sumvector \leftarrow sum(bitvectors)$
4:     $target \leftarrow createWeekVector(numReplicas \times policyInfo.get($"redundancy"$))$
   // Creates a vector of $24 \times 7$ positions initialized to the passed value.
5:     $diff \leftarrow target - sumvector$
6:     $neededHosts \leftarrow max(diff)$ //The minimum number of hosts required
   // is the largest difference between the sum vector (number of hosts
   // expected for each hour) and the target.
7:     while $sumvector < target$ do
8:         $potentialHosts \leftarrow ResourceDiscovery.findHosts(neededHosts, hostRequirements)$
   //(Steps 3 and 4)
9:         $bitvectors \leftarrow RAMon.getInfo(potentialHosts)$ //(3, 4)
10:        for all $host = potentialHosts$ do
11:            $newsumvector \leftarrow sumvector + bitvectors(host)$
12:            $newdiff \leftarrow target - newsumvector$
13:            if $newdiff < diff$ then
14:                $selectedHosts.add(host,$"permanent"$)$
15:                $diff \leftarrow newdiff$
16:                $sumvector \leftarrow newsumvector$
17:            end if
18:        end for
19:        $neededHosts \leftarrow max(diff)$
20:    end while
21:    for all $host = selectedHosts$ do //Check all the hosts to see if some is dispensable.
22:        $newsumvector \leftarrow sumvector - bitvectors(host)$
23:        if $newsumvector \geq target$ then
24:            $selectedHosts.remove(host)$
25:            $sumvector \leftarrow newsumvector$
26:        end if
27:    end for
28:    return $selectedHosts$ //(Step 5)
29: end function
```

Algorithm 5.4 Method called by the SM in the A3 for reactive recovery in the bit vector-based resource selection method.

```
1: function ONREPLICAFAILURE($numReplicas, policyInfo, hostRequirements, selectedHosts$)
2:     $availableReplicas \leftarrow selectedHosts.numAvailableReplicas$
   // Obtains the number of hosts in the selected set that are currently available.
3:     while $availableReplicas < numReplicas$ do
4:         $potentialHosts \leftarrow ResourceDiscovery.findHosts(numReplicas -$
   $availableReplicas, hostRequirements)$
5:         $selectedHosts.add(potentialHosts,$"transient"$)$
6:     end while
7:     $availableReplicas \leftarrow selectedHosts.numAvailableReplicas$
8:     return $selectedHosts$
9: end function
```

compared it to the following: (1) a similar method without the binarization step, (2) a method of choosing hosts available at the moment of deployment and (3) a method that simply chooses a random set of hosts for permanent deployment. The bit vector methods have performed better than all the rest.

Specifically, we have shown that, using 0.75 as a binarization threshold, the redundancy is lower and the availability higher than with the other approaches. Using a threshold of 0.85 causes a higher redundancy, but gives extremely high availability guarantees. Moreover, the long-term prediction-based techniques have also shown that they can obtain similar availability levels than those achieved using a short-term prediction technique, while requiring a lower number of deployments and migrations over time. This justifies the use of long-term availability prediction for deploying long-lived services and data, as opposed to short-term prediction which is more suited to task scheduling.

5.8.1 Future Work

Work on long-term availability prediction can be further extended in some ways. For example, better mechanisms could be used to optimize the selection of hosts to have values as close to N as possible in every position of the sum vector. This could be done to minimize redundancy, and at the same time make good use of hosts with medium or low availability, contrary to the simple method presented which tends to favor hosts with high availability. Alternative ways to detect highly predictable hosts could also be used.

An unsolved problem is how to predict seasonal variations in host behavior. This requires a large amount of trace data, because seasonal effects use to appear over long periods of time (e.g. holidays). The main questions are how would a real system store and summarize this information, and how would this information be used, e.g., determining the initial deployment or rather detecting seasonal variations with short anticipation and then performing corrective actions.

Another related problem is prediction of permanent failures that can undermine durability of a deployment. This may be solved by simple methods like just removing a host from the pool whenever it has been disconnected for a long time. However, trace analysis could be used to validate this assumption or to propose other methods to guarantee durability.

We would also like to test our prediction and deployment methodology using trace data of different systems. This is needed to see if our findings can be applied to other environments, or they are only valid with volunteer computing communities like SETI@home or systems with similar characteristics. While the features of the SETI@home system, as we mentioned previously, fit the target environment of CoDeS, it would be nevertheless interesting to see how well our methodology works in other environments.

Finally, we intend to integrate different availability-aware resource selection mechanisms into the CoDeS prototype and test them in real environments to have actual measures of the reliability and efficiency of each method for a variety of services. As we mentioned in Section 5.5, the cost of churn will depend on the different costs associated to migration and synchronization in each service. Therefore, we believe that there is not a “one-size-fits-all” resource selection method for service deployment that can provide optimum results for all services. Each service may have specific needs that make some resource selection methods more appropriate than others.

6 Conclusions

The goal of this thesis was to prove the feasibility of the contributory computing model, where users form a community by contributing their resources to be used collectively in a decentralized and self-managed way. For this purpose, we have designed CoDeS, a middleware to support contributory communities. This middleware offers a platform where services can be deployed and automatically managed and replicated to provide service availability. Users can contribute resources to this platform by simply joining a community, which requires no other software or manual system management than to install CoDeS. This is a novel model which overcomes some of the limitations of existing computing paradigms like cloud or volunteer computing.

This dissertation has focused on the mechanism for service deployment, which is the core functionality of CoDeS. Specifically, we have focused on the general mechanisms for service management and on the mechanisms for resource discovery and resource selection. Our mechanisms are completely decentralized and self-managed. According to our simulations, these mechanisms can be used for achieving good service availability, giving users access to services in times lower than one or two seconds, which is generally regarded as good response times for relatively complex tasks. This has been proved via simulation for large communities and high levels of churn, therefore validating the feasibility of our proposal for real environments which can be formed by thousands of nodes and subject to continuous entrance and departure of hosts.

Our decentralized resource discovery mechanism also offers a good performance compared to other existing proposals, mainly for peer-to-peer and grid environments. Specifically, average query cost of our mechanism, in number of messages per query, is constant with respect to community size, while the best performing systems which fulfill the requirements of contributory communities have a logarithmic cost. This is thanks to the use of matchmakers, which store ads and match them to queries locally, offering a very high query flexibility. Allowing multi-attribute range queries is generally costly in systems that store ads or organize nodes following a mapping of attributes to keys over a KBR layer. However, existing mechanisms that proposed the use of matchmakers lacked a self-managed, scalable and fault-tolerant method for deploying, maintaining and discovering matchmakers. We solved it thanks to the service deployment capability of CoDeS. Therefore, the proposed mechanism could also be adapted to be used by other distributed systems needing a resource discovery mechanism with a constant average cost per query and a low overhead.

Another of our contributions is the proposal of a novel resource selection mechanism for service deployment based on long-term availability prediction. We have based this mechanism on the study of real traces of SETI@home, where we have identified the presence of hosts with periodical availability patterns which could be leveraged for predicting long-term availability. The user behavior of volunteer computing systems is very similar to what we expect from contributory community members, which makes this trace the ideal source of information about user behavior. Therefore, we consider that our results with actual user behavior prove the feasibility of service deployment over non-dedicated resources contributed by individual users, which is the base of contributory computing.

However, it is obvious that the results of our research on availability prediction and resource selection could also be applied to volunteer computing systems. Our availability prediction and resource selection mechanisms could be used for deploying services or other types of applications which can leverage the notion of collective availability in volunteer computing systems. It could also be applied to distributed storage systems, since the erasure coding model, where any set of N out of a total number M of fragments are needed to recover a data object, exactly matches the definition of collective availability which we have used.

To conclude, we must mention that the architecture of CoDeS is modular, which makes it extensible. This allows researchers and developers to design and implement different mechanisms for specific parts of the middleware, like distributed storage or key-based routing (KBR), and easily incorporate them into CoDeS. This way, the middleware can adapt to the specific requirements of each community, and also leverage advances in some of the areas which we have not been part of the scope of this thesis. Moreover, we have implemented a prototype of CoDeS, which has been used to validate some of our mechanisms by simulation. It is functional and has been deployed on PlanetLab, a testbed for distributed systems. Therefore, the middleware and mechanisms presented in this dissertation can be used as a foundation for building real contributory computing systems.

6.1 Future Work

Our main line of future work with CoDeS is the development of real services which leverage the contributory computing model. These services should be deployed on a real distributed environment and provided to real users to obtain a good evaluation of their behavior and of the performance of all the mechanisms involved. While we consider that the work presented in this dissertation has proved the feasibility of contributory computing, this real deployment would be the final step needed to encourage the adoption of this model. Some of the services we are working at include web servers with support for static and dynamic content, and software for social networks allargar la frase per dir que ens centrem en . Our plans include a first deployment on PlanetLab to have the services working in a real environment and available to clients. This should encourage users to contribute their resources and form a contributory community based mainly on non-dedicated resources.

There are two main topics that could be considered as open issues for a real large-scale deployment. As commented in Chapter 2, these are security and storage. Security, on the one hand, would be critical to allow users contribute resources without concerns about exposing sensitive data or having their computers compromised. Possible solutions could include a combination of virtualization, encryption and reputation. Distributed scalable consistent storage, on the other hand, is a problem for many systems, including cloud computing, volunteer computing and peer-to-peer systems, and it has not been solved yet. However, partial solutions with different trade-offs between the required features have been proposed in these areas. Therefore, it seems viable to find a solution which can adapt to the specific requirements of contributory communities and the services that are deployed over them.

Finally, there is also place for improvements in different areas and addition of new features to our middleware. Specific lines have been detailed throughout the chapters of this dissertation, including incentive mechanisms to encourage resource contribution and load-based service auto-scaling.

Specifically, availability prediction for non-dedicated resources is a promising line of future research. An open issue is designing more complex availability prediction mechanisms, which could include, for instance, modeling seasonal variations in host behavior, correlated availability or host lifetime. Another possible research line would include developing more efficient resource selection mechanisms, which can make a better use of the resources, for instance, considering more complex services with different interacting components, or scheduling migrations and checkpoints.

A Description of Decentralized Resource Discovery Mechanisms

In this appendix we discuss a number of resource discovery mechanisms and systems. We classify them according to their main topology, and discuss all the other aspects mentioned in Sections 4.2 and 4.3, both requirements and design characteristics.
Those systems that have a name given in the literature describing them will be identified by such name. Systems for which no name is given will be identified by the author name and year of the first publication in which they are described.

A.1 Structured

In this subsection we present the systems that are built using a structured network. Many of them use an existing DHT, whether to augment it or to build a system over it, while others present their own architecture. They are further categorized in the aforementioned different types of structured networks. However, some of them do not fit in one of these categories, whether because they use a different type of structured network or because they use more than one of the known types in their implementation. These systems are filed under the category of *Hybrids and others.*

A.1.1 Ring

Ranjan et al. 07

The system presented by Ranjan et al. [118] is modeled like a grid where each site has a dedicated server for resource discovery. This server acts as a broker who resolves queries locally in a centralized way, and also connects the site to the other organizations of the grid. This node is part of a DHT, where it publishes the computational resources of its site, and can contact the brokers of other sites. Specifically, the system is implemented over Chord.

Through a decentralized publish/subscribe system, the users who need resources subscribe to the kind of resources they need. Available resources periodically issue events to advertise their availability. The resource descriptions are mapped to a d-dimensional space, which in turn is converted to a one-dimensional value. The space is divided in cells, and each cell is converted into a value, that is hashed and stored into the DHT. Subscriptions are issued by nodes needing resources, while the events are resource ads. Both are pushed to the nodes responsible for the cells.

Queries are composed of d attributes. They can specify a single value for each attribute, or ranges of values for some or all attributes. Range queries are stored diagonally in the d-dimensional space, instead of covering the whole area, which could potentially belong to more cells.

The system is said to be scalable, and according to the authors there is no significant impact of system size on performance (query latency). The presented tests use group sizes of up to 500 nodes.

However, a high level of churn can make Chord perform badly, with a huge overhead. Their simulations showed that, with a limited message queue on nodes, there was an important degree of message loss, preventing many nodes from being found and queries from being answered since their corresponding advertisements and subscriptions were dropped. Regarding security, since there are no replication methods incorporated, any malicious node would get control over a part of the range of values, therefore being able to prevent access to nodes falling into that range.

Chang-yen et al. 08

The system presented by Chang-yen et al. [42] uses a Pastry overlay network for each considered attribute. Each node that offers a value for it above a certain threshold (e.g. a certain amount of disk, etc.) belongs to the corresponding overlay. Nodes that belong to more than one layer are called gateway nodes, and allow queries to pass among layers. Each node also stores a local cache with the definition of nearby resources in the layer or layers it belongs to. They periodically send their data to nearby nodes (a subset of the first row of Pastry routing table) and also, more infrequently, to farther nodes.

Whenever a node needs to locate resources, it first checks its local cache to know if there are any nearby nodes that can answer the query. If there are not, the query is forwarded to a layer that the node does not belong to and that can provide adequate values for a required attribute. In order to reach the target layer, the query is forwarded towards a designated gateway, located in a well-known Pastry key. Therefore, the preferred method for information dissemination is by pushing it to nearby nodes, while farther information can be pulled when required. This method does not provide completeness, and requires that great amounts of resources satisfying requirements are available for queries to complete.

Since the method for looking up resources is based on caching information about nearby nodes, and the routing towards designated gateways to reach different layers is based upon Pastry, it is scalable. However, tests only show networks sizes of 60 nodes.

Resources are defined by a fixed number of attributes, with a layer per attribute in the system. The queries only search values greater or lower than a given value. However, it is said that with a modification to create layers for intervals of values, instead of values above a threshold, range queries could be supported.

Its fault tolerance is based upon Pastry protocols and replication of the designated gateways. It is also relatively resilient to attacks, since nodes cannot prevent other nodes from sending their advertisements to neighbors. The multiplicity of gateways among layers increases the probability of reaching the desired layer for a query, but a faulty node can forward queries to layers formed entirely by malicious users.

XenoSearch

XenoSearch [134] is built to search for resources that match a description that can be complex, containing range attributes, locality properties regarding one or more nodes, etc. The system is composed of a Pastry network for each attribute/dimension. Each of them has a set of aggregation points (AP), which work together to aggregate the information of a part of the network. This aggregation structure forms a binary tree, with an AP for the whole network, one for each half of it, and recursively. The APs are located by the Pastry routing function, assigning them to a specific key obtained through the division of the identifier space. These APs contain bloom filters that indicate what nodes are in the network and can be easily aggregated.

Nodes into the overlay periodically send a message to their assigned AP to update their state. This AP forwards its aggregated information up the tree, making that each AP can answer queries regarding the range that it covers by making a summary of results using bloom filters. However, the use of bloom filters means that, while all resources fulfilling a specific set of requirements can be found, XenoSearch can return false positives. Therefore, the final responsibility on choosing a node is on the client, who has to check the returned list of nodes to identify those that actually satisfy all the requirements. Moreover, only queries that can be expressed as a range of acceptable values for a predefined set of different attributes, and OR and AND operators, are considered. The client must issue range queries in each dimension, and unite or intersect the results as needed.

The architecture of XenoSearch uses local centralization, where nodes in the overlay are only a subset of the network. These nodes offer the resource discovery service, and other nodes use it to publish their resources and to obtain them when needed. The overlay network, however, is decentralized. Each node derives its id in each overlay from its value for the corresponding attribute. However, malicious nodes can make up false identifiers to gain control over some APs, making some nodes impossible to locate.

Nothing is mentioned about the scalability and fault tolerance of XenoSearch. Since it uses Pastry, it can leverage its properties, but the presence of a number of Pastry rings can make the overhead of stabilization protocols and repairs costlier. Moreover, periodical operations for maintaining trees are required for each dimension. The tests showed in the paper only use 18 nodes.

Abdullah et al. 09

The system presented by Abdullah et al. [2, 3] is formed by three kinds of agents: producers, which execute jobs in behalf of other users; consumers, which use the resources of the network to execute jobs; and matchmakers, which match queries issued by consumers with ads of producers that can satisfy them. Nodes in the network can simultaneously perform the functions of the three kinds of agents. Moreover, nodes can be dynamically promoted to matchmakers when the existing ones are heavily loaded. Once one is promoted, it remains acting as a matchmaker until its load decreases, moment when it quits being a matchmaker.

The load of the matchmaker is measured by a function called TCost, which calculates the cost of each request counting the number of preceding requests to be processed before that one. When average TCost goes over a threshold value, a new matchmaker is promoted. Therefore, the system automatically balances the load on the matchmakers, considering that the query frequency distribution is uniform among the DHT. However, these matchmakers are a point of centralization.

The topology of the overlay network is a circular DHT, Pastry. The id space of the DHT is divided into zones. There is a total division of the maximum of N nodes admitted by the DHT (depending on the number of bits of the identifiers) on M zones. For each of these zones there is a designated matchmaker, which corresponds to the first node of the next zone. In the beginning, only one of the matchmakers is activated. The others are only activated when the contiguous matchmaker is overloaded. When the number of active matchmakers is lower than the number of zones, zones are joined and covered by one matchmaker. When there are less than M zones and matchmakers are promoted, zones with more than N/M nodes are divided. This makes the system scalable, although only resources on the same zone of the DHT can be found, and no failure tolerance is considered.

Providers send their specification to the matchmaker assigned to their zone, codified in the bidding language used by the matchmaker, in what is called an ask. Consumers requiring resources send their bids, containing their requirements about time, price and amount of resources, to the matchmaker. Matchmakers use a Continuous Double Auction to match bids with asks and answer them when they are received. The bidding language used in the auction can provide a high flexibility in asking for resources, but requires a higher intelligence into the clients to calculate pricing and bidding strategies. The auction does not provide completeness in results, even considering only the zone assigned, but does provide accuracy.

The authors present experimental results obtained with up to 650 nodes and 5 matchmakers. However, only the temporal evolution of metrics like TCost and response time while matchmakers are promoted and demoted is shown.

SWORD

SWORD [111] proposes, firstly, a language to specify the set of resources required by an application, formalized in an XML that includes minimum and preferred attribute values per node, with penalizations for values between the minimum and the preferred, and relations among nodes of a group and among groups, basically the latency. As can be seen, this language has an extreme flexibility, and allows a high variety of constrains to be specified into a query.

Secondly, the SWORD paper proposes a set of mechanisms to search for resources on a distributed community. Three of them use Bamboo, a ring DHT based on Pastry, where a pair ¡attribute-value¿ is assigned to a key. The first mechanism is called SingleQuery. It requires that each node advertises itself, including the values it has for all its attributes, on every node of the DHT assigned to each of its values of the attributes. Queries only require a message to the responsible of a key, or more precisely, to a range of keys corresponding to the range of values that fulfill the requirements. The node responsible must process the query and search for nodes whose specifications are stored locally and satisfy the requirements. The second method is MultiQuery, where each node sends only the value of an attribute to the node that stores ads for that attribute and value. Queries require contacting all nodes that store some desired value for an attribute, and the results must be compared at the querying node to find which nodes have all the required values for all attributes. The third is Fixed, which requires a set of infrastructure servers. In this mechanism, each node sends its resource description to one server chosen randomly among the predefined set, and queries are sent to one of those servers and redirected to all the others. Alternatively, ads can be sent to all servers and queries to only one. The last mechanism is Index, which combines the SingleQuery and Fixed approaches. In this mechanism, there are a set of infrastructure servers which keep a mapping of value ranges to DHT nodes that manage them. Updates are handled as with the SingleQuery approach, while queries are sent to the index server responsible for the key range of interest, that forwards the query directly to the DHT nodes responsible of that range instead of using the DHT routing.

In order to find nodes that satisfy the desired inter-node and inter-group requirements, a separate mechanism is used, where nodes are clustered and representatives are used for each cluster. These representatives measure their latency with all other representatives, and each node has as a part of its specification a link to its representative. Once nodes that satisfy individual requirements are retrieved, the representatives are contacted in order to compute the inter-node requirements. Therefore, this mechanism is compatible with any resource discovery mechanism, and can be added independently to any of the four proposed approaches, or to alternative ones. However, the mechanisms for the creation, selection and monitoring of the representative nodes are not explained. Finally, SWORD implements an optimization component which computes the possible combinations among the nodes returned by the distributed search and the information from the representatives in order to find the groups that best suit the requirements. This is executed locally, so can also be added as a local module with any resource discovery mechanism that is used.

Fault tolerance is high for the distributed approaches, since Bamboo is a DHT designed to tolerate churn. Fixed and Index mechanisms, on the other hand, depend on the number and reliability of infrastructure servers. The security of the decentralized mechanisms is high since nodes register in a number of nodes, and therefore cannot be hidden by a single malicious node.

The centralized version of SWORD is deployed and running in PlanetLab [110, 112]. Queries for resources can be submitted via web [1] using the attributes monitored by CoTop [2], a monitoring tool for PlanetLab. However, its inter-node requirements' evaluation mechanism is not implemented in it.

Cheema et al. 05

The system presented by Cheema et al. [43] uses Pastry to connect the nodes of the network. These nodes have identifiers that are generated, randomly or otherwise, by the node itself, therefore allowing for malicious nodes to occupy a specific place in the network.

The characteristics of the resources are encoded into a string, including a static part, conformed of CPU speed, RAM, disk, and OS configuration of the node, as well as a dynamic part, that contains the current CPU, RAM and disk free percentage. Therefore, resources are not assigned to a point in the space, but to an arc, from the minimum to maximum values of the dynamic part, bounded by the static part. Identifiers are assigned by hashing the static part of their description, and then substituting the lower 32 bits by those that represent the dynamic part.

The information about the resources is stored by the nearest node in the identifier space, as is the normal behavior of the DHT. Resources must issue an update when their dynamic data changes so that their description is correctly stored. However, this might lead to stale data, since the update process changes the resource id and can therefore change the node responsible for the information, leaving invalid data in the former one. To avoid this, updates can be issued periodically, allowing nodes to delete the information that has not been renewed in a specific period of time.

[1] Available at http://sword.cs.williams.edu/

[2] Available at http://codeen.cs.princeton.edu/cotop/

In order to find a certain type of resource, a node must issue a query and forward it through the DHT to the identifier that encodes the desired resources. However, queries require exact matching in static attributes in order to find and answer. To improve this, the system allows for iterative queries varying dynamic attribute values. This allows querying for a range of values for the dynamic attributes, but static attributes, which define a node's capacities, must be fixed in the query. Therefore, no real range queries are supported by the system.

The scalability of the system is that of Pastry, since both advertisement messages, including insertions and updates, and query messages must be routed through the DHT towards the node responsible for the identifier. As stated, however, performing a range query would require a large amount of iterative queries and therefore be much less efficient. The tests presented, using a network of 1000 nodes, do not show the number of messages required for queries, but only overhead measures like the total number of messages in the system. The fault tolerance is also that of Pastry, introducing only a small delay after a node fails until periodical updates bring the system's information up to date again. Finally, security is not considered in the system, and since it is explicitly allowed for a node to create several virtual nodes and insert them into the network with self-generated identifiers, a malicious node could easily take over a great part of the system, rendering it unusable or allowing only access to compromised resources.

Squid

Squid [126, 127]distributes objects, whether resource descriptions or data items, in a multidimensional space, according to the values associated to a set of attributes. As a difference with other systems that take a similar approach, Squid allows the existence of keywords for describing such items. This is done by allowing values in an axis of the multidimensional space be formed by any set of characters, and ordering them lexicographically.

A fixed number of axis, and therefore keywords or attributes, is set to describe objects. This multidimensional space is then converted by using Hilbert's space-filling curves into a unidimensional space. Nodes in a Chord DHT, whose ids are assigned randomly, store these objects, having a portion of the space (a cluster of the multidimensional space) assigned to them. Since the clustering process is recursive, so are searches, with queries sent to a node responsible for the highest-order cluster, being processed and directed to the node responsible of the fitting lower-level cluster recursively.

Queries can specify exact values or ranges, as well as wildcards for keywords, i.e. the user can specify the initial part of a word, or query for any word. They use a query optimization engine to visit the minimum number of nodes possible during the recursive process, which has a lower bound in the number of nodes that store the desired data.

Tests with up to 5400 nodes show the relation among number of matches, data nodes and processing nodes of a query. For example, as the authors state, solving a query with 160 matches can be more costly than solving one with 2600 matches.

The fault tolerance of the system is that of Chord. Security is not considered in the paper, and since node ids are generated randomly, it would be easy for a malicious node to choose an identifier for itself that would give it control over a specific cluster.

DGRID

DGRID [99] is a system for service discovery in grids composed by resources owned by several administrative domains. It categorizes resources in types, which are assigned to a specific identifier and mapped to a Chord network. However, instead of sending advertisements to the corresponding node in the network, DGRID uses a virtualization scheme to create several nodes for the same type of resource into a network, one for each domain. Each node is the resource itself being published, and by adding specific entries in the finger table of nodes that point to other nodes of the same domain, DGRID ensures that failures in a domain will not affect availability on other domains.

The overhead of maintaining the DGRID network is greater than that of an ordinary Chord, since each node must maintain a greater number of entries in its finger table. However, this allows selective proximity-based or domain-based routing, and also offers a greater fault-tolerance, since the existence of alternative paths makes it difficult for a failed node to cause a query to fail. Moreover, if security into an administrative domain is tight, its nodes can use the neighbors in their own domain for the routing without the possibility of interference by malicious nodes. The fact that resource information is only stored by the very resources also increases security and decreases overhead. However, this comes at the cost of the extremely limited flexibility for defining resources and queries.

Their tests use a network formed by 25000 administration domains, with a variable number of resource types per domain. The results show that the average hop count is between 4 and 7 approximately.

MAAN

The Multi-Attribute Addressable Network (MAAN) [34] uses Chord to create an identifier space populated by nodes. However, instead of using a common hashing algorithm like SHA-1 [130], it uses an uniform locality preserving hashing technique. This guarantees that consecutive values will be located consecutively in the DHT after applying the hash function. In order to index resources described by a fixed number of attributes, MAAN requires them to register once for each attribute, hashing the pair attribute-value and storing the resource description in the corresponding node. Queries can be sent to the node responsible for the value assigned to one of the attributes, which is considered dominant for the query, and processed there to extract the list of resources that fulfill all the attribute requirements. Determining which of the involved attributes is the dominant one is left for the issuer of the query.

Range queries leverage the use of the locality preserving hashing by starting at the lower value of the range. From there, they are sent to the next consecutive node, sequentially until the last node responsible for the range is reached. In each of these nodes, the query is processed, and the results are returned to the client. Fault tolerance and security are inherited directly from the DHT, with failures requiring a recovery process and malicious nodes being able to control parts of the identifier space.

The authors present tests using a network of up to 128 nodes where the number of routing hops required for range queries is shown. With exact match queries the number is logarithmic to network size, while with a range selectivity of 10%, it becomes linear. The query cost is also presented as a function of the range selectivity. The cost is 10 hops for a 10% selectivity, while for a 90% it increases to cover all the nodes, in a network of 64 nodes. They also show that, with a 10% range selectivity and 64 nodes, queries dominated by a single attribute require about 10 hops, while when using an iterative search the number of hops grows linearly with the number of attributes in the query.

A.1.2 Multidimensional Space

Kim et al. 07

The papers by Kim et al. [82,83] define a model where users provide their resources to execute jobs. The matchmaking is totally distributed and decentralized. It uses a CAN to organize the peers according to their offered resources.

A CAN, where the point in the d-dimensional space of each node is determined by its characteristics, i.e. the value assigned to each of the d attributes that define resources. A job description, including information on how to retrieve the files and execute the job, is sent towards the resources it requires by using the CAN routing. Once there, the node, called owner node, that received the query must *push* the job towards a node that has an empty (or lightly loaded) queue.

The system aggregates various types of information including number of nodes, length of queue, etc. The aggregation is done from nodes that have more resources, that is, a higher value in each dimension, towards the lower ones. This information must be updated periodically. The information is aggregated in a dimension, but also using nodes that are near in the other dimensions. For example, with 3 dimensions, a node (1,1,1) would receive, aggregating only in the first dimension, information from (2,1,1), (3,1,1)...(N,1,1). But they aggregate too, with a modifier based on distance, information about (2-N,0,1), (2-N,2,1), etc. This information is used to load the balance by sending the jobs to the nodes that have a shorter queue, and stop pushing them once it is determined, based on local information, that the best node has been reached.

It is assumed that a node can only execute a task at a time, and nodes have a queue of jobs to execute. They are sandboxed in a very simple way, using chroot to deny them access to the file system, although the use of virtual machines is suggested.

Queries require a minimum, or exact, value in d attributes, that are mapped into a d-dimensional space. Closed range queries are not allowed.

The fault tolerance of the system is that of CAN. The results in the paper show that the recovery process is complex, and in case of dynamic nodes (between 10% and 30% of the nodes leave) the wait time for jobs increases. The scalability of the system is also hampered by the complexity of maintenance processes, but they propose some ways to improve it. However, the presented tests do not specify the network size simulated, so there is no precise data about the scalability of the system.

Malicious nodes can control nodes for which they are owners. Furthermore, they can stop pushing jobs that arrive to them, as well as providing false aggregated information to influence the route of pushed jobs. The influence of these maneuvers in the overall system will depend on the distribution of resources' characteristics and jobs' requirements, which will determine the ability of a node to position itself in a highly influential point in the d-dimensional space.

Costa et al. 09

The paper by Costa et al. [48] presents a system where nodes are arranged in a multidimensional space according to their definition in a set of (attribute, value) pairs, assuming that the number of attributes is fixed and that their values can be uniquely mapped to natural numbers. The space is divided into smaller spaces hierarchically, called cells, with various levels of subdivisions. Each node has a set of links with nodes in the neighboring spaces in each level, and these links are used to forward queries towards the nodes that can fulfill them. In order to accommodate additional or rapidly-changing attributes, these should not be mapped as dimensions, but only be checked locally by each node while the query is forwarded, directed by the more static attributes. Queries are forwarded until the required number of nodes is found, and then the path of the query is followed back to the user by the response.

In order to maintain the overlay network in the presence of failures, the system uses a two-layered gossip-based protocol. The lower layer uses an unstructured overlay with random connections, while the upper layer is structured by attribute values and linking nodes with others in neighboring zones. The lower layer is used to maintain the upper layer connected in presence of churn by providing random links.

The evaluation of the system shows that no message has ever been received twice by the same node, and that in experiments where the system does not experience churn, completeness is achieved. The overhead caused by the overlay maintenance is not considered in the results. The routing overhead is measured in number of hops on nodes that do not match the query, which remains in average under 3 for network sizes of up to 1000,00 nodes. However, when there is a small proportion of the nodes in the system that can satisfy a query, the routing overhead can increase up to almost 300 hops in the worst case with a network of 100,000 nodes and requiring complete results.

The number of links that each node must maintain is a function of the number of dimensions and the number of levels, although the real number of neighbors might be smaller since cells that are empty do not have an associated link. However, these two factors do not depend on system size, so the number of neighbors has a constant limit, albeit potentially large, for a given system, regardless of variations in size due to churn or other factors.

Since alternative paths to a node might be possible, and the neighbors are changed through the random gossiping of the lower layer of the network, it seems like malicious nodes could not hamper the functionality of a large portion of the system. However, nothing is stated about the security of the system.

MURK

MURK [69] is a system to support multidimensional range queries in p2p environments. In this system, the multidimensional space is divided whenever a node enters the network, much like in CAN. However, unlike CAN, MURK does not split the space equally, but divides it instead trying to split the *load* associated to that space equally. The network is connected by links between neighbor nodes in the space, as well as the multidimensional equivalent to skip pointers [19], which allow nodes to contact peers at exponentially increasing distances from itself. Two methods for establishing these skip pointers are presented. The first one is random, which uses random walks on the overlay network. The second one uses space-filling curves to approximate the distance between nodes in the space.

MURK can be used as a resource discovery system based in advertisements using a pull/push approach. This approach brings potential security threats, as it gives nodes control over the access to resources in a specific range. However, its accuracy is high, as well as its efficiency in stable environments. Nevertheless, load balancing in MURK under dynamic conditions is difficult and might be highly expensive.

The authors present some tests using networks of up to 200,000 nodes with two-dimensional data, achieving a routing cost of under 15 hops. Data locality, that is, the number of nodes that store the answer to a query, is very good according to the presented data, with queries almost always hitting just one node.

A.1.3 Tree

NodeWiz

NodeWiz [22] is based on a tree structure, that is used to store advertisements spread by resources available in the network as a distributed decision tree. The descriptions of nodes are formed by pairs ¡attribute, value¿. The tree divides the resources at each non-leaf node according to the value of an attribute. That is, at each node, a value of an attribute is specified and resources with a value for that attribute lower than the one specified go in the left branch, while those with a greater value go in the right branch. When a node joins, it shares the load of the most loaded node, in order to attain a good load balancing. The division made at each node is decided through a clustering algorithm over the ads stored, in order to perform the most advantageous division.

Queries sent by users are routed from the root of the tree towards the nodes that store ads that can satisfy the queries. Range queries are allowed, but the number of nodes that the query has to visit increases as the query becomes less specific, that is, all values within a range, or even any value, are accepted.

There is a single point of entrance to the tree, the root, which might become overloaded and hamper the fault tolerance of the system. The paper presents an optimization that allows caching of subtrees on the nodes that issues queries, using information returned together with the information about the found nodes. This information can be used to accelerate frequent queries, by accessing directly to a subtree that can satisfy the query without having to contact the root.

The tests show that NodeWiz can scale up to 10,000 nodes with a slow increase in hops (about 5 in average). It provides complete and accurate results, although the effect of failures is not considered since NodeWiz is thought to be deployed on stable infrastructure nodes. Therefore, only ordered departures of nodes are managed.

Load balance is maintained throughout the lifetime of the system by having underloaded nodes leave and rejoin the system. Since the joining process has the new node dividing the load with the most loaded node, it attains load balancing dynamically. However, the balance of the tree is not considered, and non uniform distributions can result in highly unbalanced trees.

Security is not considered throughout the paper. A malicious node could produce great imbalance in the tree by giving false values regarding its load, thus making all joining nodes share their load with it, while other nodes have a higher load, and making the path to the new nodes longer. Also, if it joins soon enough to be in a high part of the tree, it can direct many queries to other malicious nodes.

A.1.4 Hybrids and Others

NR-tree

Network-R-Tree (NR-Tree) [93] proposes a mechanism for resolving complex search queries in p2p systems by adapting the spatial index R*-tree [23] to a peer-to-peer environment. Items are located in leaf nodes, which are *passive* peers that store the data indexed in a local R*-tree. Each passive node is assigned a location in a multidimensional space according to the data they store. The nodes are grouped by their multi-dimensional coordinates, using minimum bounding rectangles (MBR). This is done recursively, grouping rectangles into bigger rectangles at each level, until the root tree.

There is an overlay network formed by super-peers, having a few per cluster to provide fault-tolerance. These clusters are formed by nodes holding semantically close data, which is measured by the centroid of data in the multidimensional space. In the case of resource discovery, the space maps the attributes that define resources. Since super-peers have an identifier codified as multidimensional data, the overlay network uses a routing system similar to CAN. Therefore, for processing a query, a super-peer, which acts as a server to the peers of it cluster, first routes it to the node responsible of the adequate portion of the space. Once in the cluster, the super-peer uses its NR-tree, that stores partial information of the peers of the cluster, to forward the query to the nodes that can find the items that satisfy it.

Super-peers are elected by existing peers based on the computational capabilities and bandwidth. They are replicated and chosen dynamically, and queries are sent to backup super-peers that start processing them if no cancellation is sent in a specific period of time. This mechanisms help provide fault-tolerance and support churn. With two backup super-peers per cluster, with a 40% failure rate of super-peers, the failure rate of queries is below 5%.

Range queries are routed through rectangles intersecting with the search space, and k-nearest neighbor (kNN) queries, which require the k nodes that are nearer to a specified point, are routed in a depth-first way, selecting in the first route the k better nodes and then backtracking through the branches that offer a better distance. The number of attributes that define resources is fixed, as it matches the number of dimensions of the routing space.

The system is designed to store objects in each node, and partial information is pushed towards super-nodes. With this information, super-peers can forward queries to the adequate nodes. This makes the system scalable. The system is also capable of providing complete answers, although it can also provide approximate answers with reduced cost. With a TTL in inter-cluster propagation the cost is reduced, and responses can be adequate with a fitting distribution of resources. In the presented tests, kNN queries obtain the 100% of results with a TTL of 2, while range queries need a TTL of 3. There is also a trade-off between the number of clusters and cluster size, since they affect both query performance and fault-tolerance.

There are some aspects that are not treated in NR-Tree, nevertheless, since the system is not specifically designed for resource discovery. One of them is the publication of resource information. If each node forms part of the network and only provides its own description, then having local R*-Trees does not make much sense. If the passive peers act as super-nodes of their own local cluster, indexing a set of resources, then there is the question of how the updates of the resources are managed, since the join and departure of resources would change the centroid of the data stored by a passive peer.

Malicious peers in the network could affect the routing through the multidimensional space, as well as provide false information about the nodes in their cluster, as long as they are super-peers. The algorithm for super-peer election is not specified in the paper, although it is said that they are chosen according to their computational capacities. If the algorithm uses also a trust or reputation management system, it can increase the probability that malicious peers will be excluded from the overlay network.

Cone

Cone [26] builds a heap to order resources according to a key value. This key value corresponds to an attribute of the node, like CPU speed or RAM. Specifically, nodes form a trie, where each non-leaf node is decided based on an aggregation function over the key value. In the paper they present the mechanisms needed to use Cone with Max (or greater than) operator, where each node is the one with greater value between its two children. That means that if a node is in the higher level of the tree, it is also in all the other levels. In order to manage node joins and load balancing, it uses a DHT. Nodes have a random identifier assigned and a corresponding position in this DHT. The DHT acts as the lowest level of the trie, thus inserting nodes into a random place and helping maintain the balance of the trie. Cone is implemented using Chord, but it can be built over any DHT that allows longest-prefix routing.

Regarding its search flexibility, Cone only allows to locate resources with a key value (i.e. the value for a certain attribute) equal or greater than a given value. This is because the presented mechanisms use the Max function over a single value. In the paper they introduce methods to improve the flexibility and allow searches for multiple attributes, by combining values of different attributes in a single key by bit-interleaving, although its ordering properties are not the same, and therefore Cone does not offer the same properties for multiple attributes that it does for one. With one attribute, it provides a total accuracy and completeness when the key values do not change very frequently, but in extreme cases its accuracy can drop substantially. More specifically, the presented experiments show a lag of 2.2 seconds when values of a node change, and the expected accuracy is 82% with values changing every 6 seconds and 93% with changes every 30 seconds.

Cone uses a pull method to spread information, with nodes issuing queries that are routed towards nodes that can satisfy them. The average number of hops for finding nodes satisfying a query is log(N), being N the number of nodes in the system. Therefore, it offers a good scalability. In the simulations presented at the paper, the system has been shown to scale to network sizes of up to 10000 nodes. It also provides mechanisms to recover from failures, which require (log N)2 messages.

It is a decentralized system, although its tree structure seems to give it an inherent point of centralization. However, their tests confirm that for a variety of query distributions, the load is fairly balanced among all nodes. Although nothing is said about security, a single node reporting a false key value could prevent a large portion of the resources in the group from being found from certain positions in the DHT. However, good load balancing should help minimize the effect of such an attack.

Gao et al. 04

In the system presented by Gao et al. [70] a Range Search Tree (RTS) is built over Chord to index the stored resource descriptions and allow for range queries. Resources are defined by a variable number of attributes, and each resource must register in the DHT for the corresponding value for those attributes that define it. Queries must be issued using the attribute that is expected to be more relevant.

For load balancing and fault-tolerance, identifiers are not assigned to a single node, but to a matrix. This is done dividing the registration to the identifier randomly into a number of partitions, which can be created on demand when the load increases, and by replicating each partition into a number of nodes. The size of the matrix is stored in the head node, the one responsible for the identifier according to Chord, and registering nodes must select randomly one of the partitions and register with each of its replicas. Queries, instead, must be sent to any replica of each of the existing partitions.

Each identifier corresponds to the hash of a certain range of values for a certain attribute. Therefore, since the RTS divides the whole range of allowed values for an attribute, contained in the root, in two subranges recursively until arriving to leaf nodes, each of these nodes has an identifier associated and can therefore be found through the DHT routing.

In order to minimize the number of messages required for registering and finding resources, the RTS can adapt to load. There is a default level where queries and registrations start, to avoid overloading the root of the tree, and the rest of the tree is inactive. The nodes in this level count the number of registrations and queries they receive, and depending on the load and the range distributions, which can search ranges bigger or smaller than the one covered by this level, the active part of the tree can be expanded upwards or downwards, trying to optimize the registration and query process by minimizing the number of hops required for the predominating values.

The paper presents tests with a network of 10000 peers, using different variations of the presented algorithms, and measures the number of messages involved in query and registration. First, number of messages per query is measured with an increasing registration rate, and the results show that with a registration rate of 100 registrations per second or less, the mean number of messages is lower than 2 for the best algorithm. With higher registration rates, up to 2000 registrations per second, it increases until approximately 10 for the best approach. They also present results of the number of query messages as range length increases, showing that it grows linearly when the load is high (2000 registrations per second) but remains constant when it is low (25 registrations per second).

Security is not considered in the paper, and although matrix replication seems to avoid total control over an identifier by one node, in truth the head node can control it. However, the adaptive tree makes it difficult, although not impossible, for a malicious node to gain control over specific ranges of the attribute space.

Tanin et al. 07

The system presented by Tanin et al. [138] uses a quadtree to map resource descriptions and queries to nodes in the network. Both are considred as rectangular spaces defined by ranges of values in the attribute space. In a bidimensional space, the quadtree divides the space recursively into four square blocks. In order to process a registration or query, it is stored in a level as low as possible that doesn't require the object to be subdivided.

The quadtree blocks are hashed into the Chord identifier space and assigned to nodes in the network. Therefore, nodes can send queries to be processed to the tree by using the Chord routing mechanism. In order to avoid all queries being sent to the root of the quadtree, it is determined that they start at a specified minimum level greater than zero. In order to bound the cost of queries, a maximum level is also fixed, which determines the maximum number of DHT lookups needed. Moreover, nodes in the tree cache the addresses of their children, in order to allow direct messaging from the root to the leafs of the subtree considered in the search. The starting point can be locally calculated by the node issuing the query.

Security is low, since there is no replication and a node controlling the higher levels of the tree can hold control of a great part of the network. Fault tolerance, on the other hand, is provided by Chord, but the addresses cached by quadtree nodes can disappear or become invalid in presence of churn. Therefore, the performance of the search mechanism can be notably affected.

The authors present some tests with a network of 1000 nodes that show the behavior of the system, measured by the average time and number of messages per query, when parameters of the system change. The first one is the minimum level permitted, which causes both metrics to grow notably when increased over 5. Other factors like the maximum level and the query scale cause smaller increases in response time and message count. Finally, the system is tested with a network of between 500 and 3000 nodes showing no notable effect in response time. In these same experiments, the cache miss ratio of the tree links is also increased up to a 15% with a moderate effect on response time. However, the response time in any case is reported as above 40 seconds.

P-tree

P-tree [50] is an index structure that allows range queries in a peer-to-peer environment. It is not specifically dedicated to resource discovery. Moreover, it has little flexibility, allowing only for node id to be used for search. However, there are possibilities of using this as a base for a resource discovery system. For example, as some of the systems presented in this appendix propose, multiple overlay networks can be used to index according to multiple attributes, or an aggregation of multiple attributes can be used as identifier.

The system uses Chord to organize a set of nodes in a ring, according to their identifier. Each node locally stores the path from the root of the tree to itself, building the tree as if the node has the smallest value present in the network. This local information is periodically updated by a stabilization process. This allows queries to be routed to the adequate subtree in a logarithmic time. Specifically, the search cost in a P-tree of order d is *O(m+logdN)*, being N the number of peers in the network and m the number of peers in the selected range.

The results presented by the authors show that the increase in search cost is logarithmic, with tests simulating networks of up to 250000 nodes. However, the number of hops is high, going from almost 100 for a network of 1000 nodes to a maximum of about 150 hops. A real implementation with 6 nodes simulating 30 virtual peers shows a response time for searches of 0.043 seconds in stable environments, while in an inconsistent network it takes 2.796 seconds. The stabilization process itself takes 17.25 seconds.

P-tree offers a good fault-tolerance, leveraging the mechanisms provided by Chord and adding some eventual consistency guarantees. It also provides completeness as well as accuracy. Moreover, the use of local information and multiple visions of the tree without a single root not only enhance performance, but also security, making it difficult for a malicious node to purposefully control a range of the network.

SCRAP

Ganesan et al. [69] propose SCRAP, a system where nodes are mapped into a multidimensional space, which is converted into a unidimensional space using space-filling curves. After this conversion, the range is divided in sub-ranges, which are assigned to the nodes of the network. These nodes are linked in a skip graph [19], which consists of a circular list and a set of skip pointers, randomly set at each node to nodes at exponentially increasing distances from itself, which allow the routing to proceed faster to the target node.

Multidimensional range queries are converted, using the space-filling curves, into a number of unidimensional range queries, as a given range in the multidimensional space may not be located contiguously in the unidimensional space. This also implies that some undesired results might be returned, including nodes that do not belong into the required range.

The authors present some tests using networks of up to 200000 nodes with two-dimensional data, achieving a routing cost of under 15 hops. However, the data locality, that is, the number of nodes that store the answer to a query, grows linearly until presented values of over 30 nodes. The total cost of answering a query is, in fact, the sum of both values.

Although the system is designed for general purpose uses, it could provide resource discovery by publishing resource advertisements in the nodes responsible for the multidimensional range that includes the specific values of each resource. Therefore, it would use both pull and push methods, having queries return the advertisements with the addresses of the required resources. However, these gives nodes control over the access to resources in a specific range, with the security risks that this implies.

Mercury

Mercury [28] is a system intended for the storage of objects defined by multiple attributes and their retrieval through range queries. To attain this objectives, Mercury creates a network for each of the existing attributes, called an attribute hub. These hubs are organized similarly to ring DHTs as Chord or Pastry. However, object identifiers are not cryptographically hashed, but they use their values for the corresponding attribute. Thus, they maintain their logical order, and range queries only need to be routed to the node nearest to the beginning of the range and start a sequential search from there.

The value space for each attribute is divided into ranges. As the population of the range is not expected to be distributed uniformly, load balancing techniques must be used. Basically, Mercury subdivides highly loaded ranges, creating a denser population of nodes where there is a denser population of objects. In order to enhance routing, they have links to nodes at certain distances as well as links to successors and predecessors, much like DHTs. These links are set through a random sampling process. Nodes also have links to other attribute hubs, in order to route queries to the appropriate subnetwork. All these links are replicated for fault tolerance. The authors present results that show Mercury's scalability with up to 50000 nodes, where routing takes less than 15 hops.

In order to use Mercury for resource discovery, it has to resort to the use of advertisements, which describe resources. These are stored in a node, in each attribute hub, assigned to a range of values covering the value of the resource for such attribute. In order to issue a range query, the issuer must determine the attribute for which the query is most selective to minimize the number of nodes that must be contacted.

The unbalanced load distribution can be leveraged by attackers to gain control of the system. Since nodes can decide where to join, a malicious node could join to a desired, highly populated range and gain control of the desired range of attribute values, effectively forbidding access to existing resources in that range and with the possibility to advertise malicious resources instead. The load balancing techniques require the highly loaded node to make a request for a lightly loaded node to disconnect and reconnect in the suggested position. A malicious node could refuse all such requests issued by other nodes, and even inject requests in order to move other nodes to its convenience.

Prefix Hash Tree

Prefix Hash Trees [115] or PHTs are conceived to build an additional layer on top of generic DHT functionalities that allow to perform range searches over a single value. Each object has assigned a value, that can be viewed as a string of a determined length. These objects are organized according to prefixes in a trie. The function of PHT is to distribute such trie over a DHT.

Each prefix is hashed and assigned to the corresponding node in the DHT. The structure of the trie is created on the fly, with it being expanded and contracted as the number of objects stored in a leaf node increase or decrease past an established threshold. Since the DHT is used to locate each node of the trie, additional state is not necessary, enhancing fault tolerance. However, this implies that trie traversal requires a number of DHT lookups, each with its (usually logarithmic) own cost.

Simulations with a DHT of 1000 nodes test the sub-optimality of range queries, that is, how many leaf nodes more than the optimal are traversed in range queries, as well as the load balance of the system. The former show that the number of nodes traversed is around 1.4 times the optimal. The latter shows that about 80

PHTs can be used to build a single attribute-based resource discovery system by storing resource descriptions as objects. As in other systems, this means that the resources would need to republish their advertisements periodically to attain fault-tolerance, and that completeness of results is not guaranteed in dynamic environments, since advertisements can be lost due to node failure for the period until they are republished.

Security is not commented throughout the paper. Since trie nodes can be accessed individually using the DHT, and the trie changes depending on the load, it would be difficult for a malicious node to gain control over a large portion of the trie. It could, however, if it enters the system as a leaf node and it has the ability to prevent the expansion of the trie. However, whether it does or not is not specified in the paper.

FaSReD

FaSReD [132] is intended to be a Fast and Scalable Resource Discovery system. It uses a Prefix Hash Tree (PHT), which is built over Chord. Each resource assigns itself a key from the values of its attributes. For example, a certain value of CPU is assigned some bits (010), a value of RAM is assigned another set of bits (100) and the OS a last set of bits (011). The bits corresponding to all the attributes are concatenated to form the id of the node (010100011). Resources are inserted into the tree in the position that corresponds to its id. Queries can find resources by searching the DHT for a specific id, generated itself from the specified requirements

FaSReD supports range queries, by using the maximum common prefix among its maximum and minimum id values to identify a point to start the search. FaSReD uses the DHT to efficiently find the node which acts as root of the subtree of interest. From this point, a PHT search is performed to find those resources that meet the requirements. The main problem with this method is that, since attributes are ordered in the ids, queries with a wide range in its first attributes will start its search in a high level and will have to visit a high number of nodes. However, with an intelligent distribution of attributes, the root of the tree can be freed from most of the responsibility and load that it would have in an ordinary PHT. Nevertheless, it requires that the number of attributes and ranges is fixed beforehand, hampering its flexibility, as well as knowing the distribution of attributes values that will be queried in order to optimize their performance.

In order to have a balanced tree, FaSReD requires all values to be uniformly distributed among the range of the attribute. The mechanism is complete in a stable environment, but failures have not been considered in the maintenance of the tree. The tests presented in the paper show an acceptable scalability and performance, although the overhead that would result from failures is not considered. Security is also not considered throughout the paper, and therefore, nodes responsible for ranges can deny access to resources in their range.

A.2 Unstructured

Iamnitchi et al. 01

The papers by Iamnitchi et al. [78, 79] present some approaches for peer-to-peer resource discovery in grid environments. Their system model is composed of a number of organizations which want to share resources among them. Each of these organizations has a local, centralized or hierarchical resource discovery system which can be used to find resources satisfying a certain set of requirements. The problem that they address is how to efficiently find resources in a large scale network formed by many organizations.

They propose that each organization or site has a dedicated server acting as a super-node, which would have information about all the resources in its site. Super-nodes of different sites are connected in a peer-to-peer fashion, forming an unstructured overlay network. Queries that cannot be answered locally by the site's resource discovery system are then forwarded through the peer-to-peer network to other sites. When the required resources are found, the node that can provide the resources replies directly the node that issued the query, which in turn forwards the answer to the user.

They show four different techniques for query forwarding on the defined overlay network. The first one is *random*, where the node forwards the query to a neighbor chosen randomly among the list of neighbors stored by the node. The second one is *experience-based + random*, where a node forwards a query to a node that answered similar requests previously. This requires the collection of information as the system operates, and mechanisms to detect query similarity. In case no relevant information is available, the node just forwards the query to a random neighbor. The third approach is *best-neighbor*, where the node records the number of responses received from each neighbor. Queries are forwarded to the node that answered more requests, without considering similarity of the queries answered. Finally, *experience-based + best-neighbor* uses the experience-base approach to choose a neighbor to forward the query. When no relevant information is present, the best-neighbor policy is used instead.

In the experiments presented the experience-based + random approach happens to obtain the best results. Best-neighbor approach is good for scenarios where the resource distributions are unbalanced, and a small number of nodes offer most of the available resources. This makes plausible the assumption that nodes that returned a certain type of resources might also provide resources of a different kind. However, when the load is balanced, and all nodes provide resources in a similar quantity, best-neighbor performs worst than the random approach.

Since queries are processed locally by super-nodes which only need to manage a supposedly small number of clients and resources, the search flexibility of the system could be high. However, for the experience-based policy, similarity among queries needs to be determined, which will likely place constraints on the flexibility of queries.

Intra-organization configuration is static, so failures are repaired manually by organization managers or by fault-tolerance mechanisms established internally. However, the failures of any number of nodes will not isolate other nodes, because the overlay is dynamically maintained when responses to queries are received by previously unknown nodes.

With the experience-based forwarding policy, queries can find resources in large networks with a small number of hops, achieving a good scalability. Moreover, information about previous responses, in conjunction with grid security and authentication mechanisms, can serve as a kind of trust mechanism to avoid attacks, letting malicious nodes little capacity of action. However, the mechanism is not complete, and when resources of a certain type are scarce, the performance of queries can drop notably.

Mastroianni et al. 05

The paper by Mastroianni et al. [101] presents a grid where each site has a super-node, that is connected to other super-nodes forming a peer-to-peer network. Nodes in a site send their queries for resources to the local super-node. This checks the local information service that operates inside the site, like Globus' MDS-2 [103]. If not enough resources that satisfy the query are found, the super-node uses the overlay peer-to-peer network to forward the query to other sites.

In order to find resources in other sites, the super-node sends the query to a subset of its neighbors. This subset is empirically determined to contain the neighbors that provided a higher number of responses in previous queries. However, this approach was shown to be the worst performing of all the analyzed in the papers by Iamnitchi et al. [78, 79].

The query language is not specified, and the paper only mentions that it should be a standard language that super-nodes of all sites can understand. Moreover, this language should be translatable to that of the local information service of each site. Therefore, the flexibility of the query language will depend on the systems deployed at each grid site, although potentially it could have a great flexibility since queries are processed in centralized or hierarchical networks of small size.

The system relies on infrastructure servers at each site, so fault-tolerance should be locally provided. However, no mention is done of simple policies like replication of super-nodes. Neighbors in the overlay networks are only chosen when a super-node joins the grid, so there is the possibility that failures and disconnections end up leaving a non-failed node without active neighbors. Security is also provided by the grid mechanisms working at each site. However, even in the case of malicious nodes present in the overlay network, queries should obtain correct responses since they are forwarded along multiple paths. Grid security mechanisms should prevent the use of faulty resources pointed by a malicious node that receives a query.

The scalability of the system is high since nodes have a constant number of neighbors. In cases where there is a large number of resources that satisfy a query, there is a high probability that responses will be received in a small number of hops between sites. However, if the required type of resources is scarce, the search will require many hops until a result can be found, since the search is not directed, and the *best neighbor* algorithm does not provide the best results in this case.

Cluster Computing on the Fly

Cluster Computing on the Fly (CCOF) [94] is designed to harvest idle cycles from computers connected to the Internet. In a paper [147] the authors focus on the resource discovery component of CCOF, and define a general framework for it. In their model, nodes form an unstructured overlay network, which can be used to forward queries. Once a query arrives at a node that can satisfy it, the node sends a response to the one who issued the query. Once it has received enough answers, it proceeds to contact the nodes that will execute the required application.

The authors evaluate four search methods. The first of these methods is expanding ring search, where a node sends a request to its direct neighbors, and they send a response if they can execute the task. If not enough responses are received, then it sends the query to peers one hop farther. This process is repeated until enough responses are found or the search reaches the scope limit. The second method is random walk search, where the node forwards the query to k random neighbors. These neighbors answer to the node if they can complete the task, and forward it to k random neighbors until the scope limit is reached. The third method is ads-based search, where each node forwards its profile information, i.e. its resource specification, to its neighbors. They cache and forward it until its scope limit is reached. When a node needs to execute a job, it looks in its list of cached ads for a node that can respond the request. Finally, the Rendezvous Point search assumes that there are a set of Rendezvous Points that nodes can access to locate available hosts. In order to do this, nodes send their advertisements to the nearest Rendezvous Point. It is assumed that Rendezvous Points are placed geographically scattered and in sufficient number for the network to perform correctly by an out-of-band mechanism. Nodes discover Rendezvous Points by asking their neighbors. These four approaches are compared to a centralized resource discovery mechanism that serves as an optimum reference.

The scalability of the system is tested with 4000 nodes, and the Rendezvous Points mechanism is shown to be the best in terms of number of messages sent, while the ads-based and expanding ring require less hops per query. Fault-tolerance is high with respect to the resource discovery process for all the presented algorithms, although the job completion rate of the whole system drops notably with high peer disconnection probabilities. The security of the Rendezvous Points approach is unknown since it would depend on the absence of malicious nodes among the set of Rendezvous Points, which are assumed to be placed by an out-of-band method. If this method is not secure, malicious Rendezvous Points could make their zone of influence non functional. The other approaches seem more secure, since a single node cannot hamper query or ad routing notably.

The presented mechanisms do not achieve completeness, at least in a realistic setting. A node could contact all the Rendezvous Points in the network in the corresponding approach, but this could be non-practical depending on the network size and proportion of Rendezvous Points. The other approaches would require flooding to contact all of the nodes in the network in order to achieve completeness. Accuracy of the expanding ring and random walk mechanisms is high, since the requests are answered directly by the nodes involved. Ads-based and Rendezvous Points, however, are sensible to stale data, potentially returning nodes that are not available anymore, or that have changed some of their dynamic characteristics, like load.

Potentially, all the methods have high query flexibility, as the matching is done locally in the candidate node, so any kind of checking could be performed, or over a limited amount of ads. However, the query format explained in the paper only includes number of processors needed and time spent on each processor.

Moreno-Vozmediano 06

The paper by Moreno-Vozmediano [106] presents a resource discovery mechanism for ad-hoc grids, composed of wireless devices and forming mobile ad-hoc networks or MANETs. The model presented in the paper is an hybrid mechanism, where each node sends advertisements to nodes at distance at most R hops, and store the ads they receive. When a node is searching resources, if they are not found locally in the stored ads, a query is sent to nodes at distance R. These nodes, upon receiving the query, look for resources in their local database. If they are found, the node that issued the query is informed. If not, the query can be sent to nodes at distance 2R.

The nodes form a decentralized unstructured overlay network, where queries are processed locally over a small number of resource advertisements, and therefore there is a potentially high query flexibility. Scalability is achieved by limiting the scope of both ad and query forwarding. These also implies that the mechanism is not complete, and only nearby resources can be efficiently found, although its accuracy can be high, only having the risk of stale data if the update periodicity of ads is too long. The system's fault-tolerance, however, is not specified. Its security, on the other hand, depends on the specific topology of each network. Malicious nodes can make a part of the network unreachable for their neighbors if the network is not highly connected and alternate paths to each node are not present. However, in general, they cannot have a high impact on the overall network.

Bibliography

[1] T. Abdullah, V. Sokolov, B Pourebrahimi, and K.L.M. Bertels. Self-organizing dynamic ad hoc grids. In *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, pages 202–207, October 2008.

[2] Tariq Abdullah, Luc Onana Alima, Vassiliy Sokolov, Calom David, and Koen Bertels. Hybrid resource discovery mechanism in ad hoc grid using structured overlay. In *Proceedings of the 22nd International Conference on Architecture of Computing Systems*, ARCS '09, pages 108–119, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Tariq Abdullah, Lotfi Mhamdi, Behnaz Pourebrahimi, and Koen Bertels. Resource discovery with dynamic matchmakers in ad hoc grid. In *Proceedings of the 2009 Fourth International Conference on Systems*, pages 138–144, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, Steven Tuecke, and Ian Foster. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, MSS '01, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society.

[5] D. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[6] 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.

[7] 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*, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[8] David P. Anderson and John McLeod. Local scheduling for volunteer computing. *Parallel and Distributed Processing Symposium, International*, 0:477, 2007.

[9] D.P. Anderson and K. Reed. Celebrating diversity in volunteer computing. In *Proceedings of the 42nd Hawaii International Conference on System Sciences*, pages 1–8, Washington, DC, USA, January 2009. IEEE Computer Society.

[10] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[11] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne, and Miranda Mowbray. Automatic grid assembly by promoting collaboration in peer-to-peer grids. *Journal of Parallel and Distributed Computing*, 67(8):957 – 966, 2007.

[12] A. Andrzejak, D. Kondo, and D.P. Anderson. Exploiting non-dedicated resources for cloud computing. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 341 –348, April 2010.

[13] Artur Andrzejak, Derrick Kondo, and David P. Anderson. Ensuring collective availability in volatile resource pools via forecasting. In *DSOM*, pages 149–161, 2008.

[14] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. The sharegrid peer-to-peer desktop grid: Infrastructure, applications, and performance evaluation. *J. Grid Comput.*, 8:543–570, December 2010.

[15] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. Mcgough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, Global Grid Forum, June 2005.

[16] 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. *Commun. ACM*, 53:50–58, April 2010.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[18] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A.” Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS’95*, pages 267–278, May 1995.

[19] James Aspnes and Gauri Shah. Skip graphs. *ACM Trans. Algorithms*, 3, November 2007.

[20] Mehmet Bakkaloglu, Jay J. Wylie, Chenxi Wang, and Gregory R. Ganger. On correlated failures in survivable storage systems. Technical Report MU-CS-02-129, Carnegie Mellon University, May 2002.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[22] Sujoy Basu, Sujata Banerjee, Puneet Sharma, and Sung-Ju Lee. Nodewiz: peer-to-peer resource discovery for grids. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01*, CCGRID '05, pages 213–220, Washington, DC, USA, May 2005. IEEE Computer Society.

[23] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[24] Michael Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. John Wiley and Sons, 2008.

[25] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. In *Proceedings of IPTPS'03*, 2003.

[26] Ranjita Bhagwan, Priya Mahadevan, George Varghese, and Geoffrey M. Voelker. Cone: A distributed heap approach to resource selection. Technical report, UCSD, 2004. CS2004-0784.

[27] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *NSDI*, pages 337–350, 2004.

[28] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 353–366, New York, NY, USA, 2004. ACM.

[29] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[30] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed file System Deployed on an Existing Set of Desktop PCs. In *Proceedings of SIGMETRICS*, 2000.

[31] James C. Browne, Madulika Yalamanchi, Kevin Kane, and Karthikeyan Sankaralingam. General parallel computations on desktop grid and p2p systems. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, LCR '04, pages 1–8, New York, NY, USA, 2004. ACM.

[32] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: A highly-scalable multi-user peer-to-peer file system. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 1173–1182. Springer Berlin / Heidelberg, 2005.

[33] R. Butler, V. Welch, D. Engert, I. Foster, S. Tuecke, J. Volmer, and C. Kesselman. A national-scale authentication infrastructure. *Computer*, 33(12):60 – 66, December 2000.

[34] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 184–191, Washington, DC, USA, November 2003. IEEE Computer Society.

[35] B. Calder, A. A. Chien, J. Wang, and D. Yang. The Entropia Virtual Machine for Desktop Grids. In *Proceedings of the First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[36] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM.

[37] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489 – 1499, October 2002.

[38] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[39] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20:398–461, November 2002.

[40] 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, pages 2–2, Berkeley, CA, USA, 2009. USENIX Association.

[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[42] Ian Chang-Yen, Denvil Smith, and Nian-Feng Tzeng. Structured peer-to-peer resource discovery for computational grids. In *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*, MG '08, pages 6:1–6:8, New York, NY, USA, 2008. ACM.

[43] A. S. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer discovery of computational resources for grid applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 179–185, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Ann Chervenak, Ewa Deelman, Miron Livny, Mei-Hui Su, Rob Schuler, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Data placement for scientific applications in distributed environments. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, Austin, TX, September 2007.

[45] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63:597–610, 2003.

[46] J. Chu, K. Labonte, and B. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom: Scalability and Traffic Control in IP Networks*, July 2003.

[47] Kenneth Church, Albert Greenberg, and James Hamilton. On Delivering Embarrassingly Distributed Cloud Services. In *HotNets*, 2008.

[48] P. Costa, J. Napper, G. Pierre, and M. van Steen. Autonomous resource selection for decentralized utility computing. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 561 –570, June 2009.

[49] Peter Couvares, Tevik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow in condor. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, chapter 22, pages 357–375. Springer Press, January 2007.

[50] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, WebDB '04, pages 25–30, New York, NY, USA, 2004. ACM.

[51] 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.

[52] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 181 –194, 2001.

[53] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.*, 35:202–215, October 2001.

[54] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin / Heidelberg, 2003.

[55] A.G. Dimakis, P.B. Godfrey, Yunnan Wu, M.J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on*, 56(9):4539 –4551, September 2010.

[56] P. Dinda. Online Prediction of the Running Time of Tasks. *Cluster Computing*, 5(3):225–236, July 2002.

[57] P. Domingues, P. Marques, and L. Silva. Resource usage of windows computer laboratories. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 469 – 476, June 2005.

[58] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag.

[59] John R. Douceur. Is remote host availability governed by a universal law? *SIGMETRICS Performance Evaluation Review*, 31(3):25–29, 2003.

[60] P. Druschel and A. Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 75 – 80, May 2001.

[61] A.E. El-Desoky, H.A. Ali, and A.A. Azab. A pure peer-to-peer desktop grid framework with efficient fault tolerance. In *Computer Engineering Systems, 2007. ICCES '07. International Conference on*, pages 346 –352, November 2007.

[62] Charles Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, pages 147–153, 2003.

[63] Martin Feller, Ian Foster, and Stuart Martin. Gt4 gram: A functionality and performance study, 2007.

[64] Thomas Fischer, Stephan Fudeus, and Peter Merz. A middleware for job distribution in peer-to-peer networks. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 1147–1157, Berlin, Heidelberg, 2007. Springer-Verlag.

[65] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, November 2008.

[66] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 1999.

[67] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15:200–222, August 2001.

[68] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[69] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in p2p systems. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, WebDB '04, pages 19–24, New York, NY, USA, 2004. ACM.

[70] Jun Gao and Peter Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, pages 239–250, Washington, DC, USA, 2004. IEEE Computer Society.

[71] C.P. Gavalda, P.G. Lopez, and R.M. Andreu. Deploying wide-area applications is a snap. *Internet Computing, IEEE*, 11(2):72 –79, March-April 2007.

[72] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[73] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[74] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 147–158, New York, NY, USA, 2006. ACM.

[75] N. Gold, A. Mohan, C. Knight, and M. Munro. Understanding service-oriented software. *Software, IEEE*, 21(2):71 – 77, March–April 2004.

[76] Elisa Heymann, Miquel A. Senar, Emilio Luque, , and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000)*, Bangalore, India, December 2000.

[77] P. Hofmann and D. Woods. Cloud computing: The limits of public clouds for business applications. *Internet Computing, IEEE*, 14(6):90 –93, November–December 2010.

[78] Adriana Iamnitchi, Ian Foster, and Daniel C. Nurmi. A peer-to-peer approach to resource location in grid environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 419–, Washington, DC, USA, 2002. IEEE Computer Society.

[79] Adriana Iamnitchi and Ian T. Foster. On fully decentralized resource discovery in grid environments. In *Proceedings of the Second International Workshop on Grid Computing*, GRID '01, pages 51–62, London, UK, 2001. Springer-Verlag.

[80] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P. Anderson. Discovering statistical models of availability in large distributed systems: An empirical study of seti@home. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2011.

[81] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *IPTPS*, pages 98–107, 2003.

[82] Jik-Soo Kim, Beomseok Nam, M. Marsh, P. Keleher, B. Bhattacharjee, and A. Sussman. Integrating categorical resource types into a p2p desktop grid system. In *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 284 –291, October 2008.

[83] Jik-Soo Kim, Beomseok Nam, Michael Marsh, Peter Keleher, Bobby Bhattacharjee, Derek Richardson, Dennis Wellnitz, and Alan Sussman. Creating a robust desktop grid using peer-to-peer services. *Parallel and Distributed Processing Symposium, International*, 0:315, 2007.

[84] Predrag Knežević, Andreas Wombacher, and Thomas Risse. Dht-based self-adapting replication protocol for achieving high data availability. In Ernesto Damiani, Kokou Yetongnon, Richard Chbeir, and Albert Dipanda, editors, *Advanced Internet Based Systems and Applications*, pages 201–210. Springer-Verlag, Berlin, Heidelberg, 2009.

[85] George Kola, Tevfik Kosar, and Miron Livny. Run-time adaptation of grid data-placement jobs. In *Proceedings of the International Workshop on Adaptive Grid Middleware (AGridM2003)*, New Orleans, LA, September 2003.

[86] Derrick Kondo, Artur Andrzejak, and David P. Anderson. On correlated availability in internet distributed systems. In *IEEE/ACM International Conference on Grid Computing (Grid)*, Tsukuba,Japan, 2008.

[87] Derrick Kondo, Gilles Fedak, Franck Cappello, Andrew A. Chien, and Henri Casanova. Characterizing resource availability in enterprise desktop grids. *Future Generation Comp. Syst.*, 23(7):888–903, 2007.

[88] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 398 –407, May 2010.

[89] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *18th International Heterogeneity in Computing Workshop*, Rome, Italy, May 2009.

[90] Daniel Lázaro, Joan Manuel Marquès, and Xavier Vilajosana. Flexible resource discovery for decentralized p2p and volunteer computing systems. In *Proceedings of the Sixth International Workshop on Collaborative Peer-to-Peer Systems (COPS)*, WETICE'10, pages 235–240, June 2010.

[91] Jian Liang, Rakesh Kumar, and Keith W. Ross. The fasttrack overlay: A measurement study. *Computer Networks*, 50(6):842 – 858, 2006. Overlay Distribution Structures and their Applications.

[92] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, 1988.

[93] Bin Liu, Wang-Chien Lee, and Dik Lun Lee. Supporting complex multi-dimensional queries in p2p systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, pages 155–164, Washington, DC, USA, 2005. IEEE Computer Society.

[94] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao. Cluster computing on the fly: P2p scheduling of idle cycles in the internet. In Geoffrey M. Voelker and Scott Shenker, editors, *Peer-to-Peer Systems III*, volume 3279 of *Lecture Notes in Computer Science*, pages 227–236. Springer Berlin / Heidelberg, 2005.

[95] D. Long, A. Muir, and R. Golding. A Longitudinal Survey of Internet Host Reliability. In *14th Symposium on Reliable Distributed Systems*, pages 2–9, 1995.

[96] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys Tutorials, IEEE*, 7(2):72 – 93, 2005.

[97] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[98] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: distributed hashing in a small world. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.

[99] Verdi March, Yong Meng Teo, and Xianbing Wang. Dgrid: a dht-based resource indexing and discovery scheme for computational grids. In *Proceedings of the fifth Australasian symposium on*

ACSW frontiers - Volume 68, ACSW '07, pages 41–48, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[100] Alexandros Marinos and Gerard Briscoe. Community cloud computing. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 472–484, Berlin, Heidelberg, 2009. Springer-Verlag.

[101] Carlo Mastroianni, Domenico Talia, and Oreste Verta. A super-peer model for resource discovery services in large-scale grids. *Future Gener. Comput. Syst.*, 21:1235–1248, October 2005.

[102] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[103] The globus alliance: Information services in the globus toolkit 2 release. `http://www.globus.org/mds/mdstechnologybrief_draft4.pdf`.

[104] R. B. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277, 1968.

[105] R. Morales and I. Gupta. Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):446 –459, April 2009.

[106] Rafael Moreno-Vozmediano. Resource discovery in ad-hoc grids. In Vassil Alexandrov, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 1031–1038. Springer Berlin / Heidelberg, 2006.

[107] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36:31–44, December 2002.

[108] Brad A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '85, pages 11–17, New York, NY, USA, 1985. ACM.

[109] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[110] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[111] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.

[112] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planet-lab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[113] Johan Pouwelse, Pawel Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In Miguel Castro and Robbert van Renesse, editors, *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, pages 205–216. Springer Berlin / Heidelberg, 2005.

[114] J. Pruyne and M. Livny. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters . *Journal on Future Generations of Computer Systems*, 12, 1996.

[115] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, Intel Research Berkeley, 2004.

[116] Karthick Ramachandran, Hanan Lutfiyya, and Mark Perry. Decentralized resource availability prediction for a desktop grid. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 643 –648, May 2010.

[117] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[118] R. Ranjan, L. Chan, A. Harwood, S. Karunasekera, and R. Buyya. Decentralised resource discovery service for large scale federated grids. In *Proceedings of the Third IEEE international Conference on E-Science and Grid Computing*, pages 379–387, Washington, DC, December 2007. IEEE Computer Society.

[119] R. Ranjan, A. Harwood, and R. Buyya. Peer-to-peer-based resource discovery in global grids: a tutorial. *Communications Surveys Tutorials, IEEE*, 10(2):6 –33, 2008.

[120] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001.

[121] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accountable execution of untrusted programs. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 136 –141, 1999.

[122] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. "mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design". *IEEE Internet Computing Journal*, 6(1), Jan./Feb. 2002.

[123] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[124] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37:42–81, March 2005.

[125] S. Saroiu, P.K. Gummadi, and S.D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedinsg of MMCN*, January 2002.

[126] Cristina Schmidt and Manish Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8:19–26, May 2004.

[127] Cristina Schmidt and Manish Parashar. Squid: Enabling search in dht-based systems. *J. Parallel Distrib. Comput.*, 68:962–975, July 2008.

[128] Lucas M. Schnorr, Arnaud Legrand, and Jean-Marc Vincent. Visualization and detection of resource usage anomalies in large scale distributed systems. Technical Report INRIA-00529569, INRIA, October 2010.

[129] L.J. Senger, M.A. de Souza, and D.C. Foltran. Towards a peer-to-peer framework for parallel and distributed computing. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 127 –134, October 2010.

[130] Sha-1 standard. `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[131] Haiying Shen, Cheng-Zhong Xu, and Guihai Chen. Cycloid: a constant-degree and lookup-efficient p2p overlay network. *Perform. Eval.*, 63:195–216, March 2006.

[132] D. Smith, Nian-Feng Tzeng, and M.M. Ghantous. Fasred: Fast and scalable resource discovery in support of multiple resource range requirements for computational grids. In *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, pages 45 –51, July 2008.

[133] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, September-October 2009.

[134] D. Spence and T. Harris. Xenosearch: distributed resource discovery in the xenoserver open platform. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pages 216 – 225, June 2003.

[135] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[136] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, G. Gedye, and D. Anderson. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Proc. of the Fifth Intl. Conf. on Bioastronomy*, 1997.

[137] Girish Suryanarayana and Richard N. Taylor. A survey of trust management and resource discovery technologies in peer-to-peer applications. Technical report, UCI Institute for Software Research, July 2004. UCI-ISR-04-6.

[138] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16:165–178, April 2007.

[139] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[140] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of dht security techniques. *ACM Comput. Surv.*, 43:8:1–8:49, February 2011.

[141] Robbert van Renesse and Adrian Bozdog. Willow: Dht, aggregation, and publish/subscribe in one protocol. In Geoffrey M. Voelker and Scott Shenker, editors, *Peer-to-Peer Systems III*, volume 3279 of *Lecture Notes in Computer Science*, pages 173–183. Springer Berlin / Heidelberg, 2005.

[142] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Proceedings of the*

Third International Workshop on Grid Computing, GRID '02, pages 1–12, London, UK, 2002. Springer-Verlag.

[143] Hong Wang, Hiroyuki Takizawa, and Hiroaki Kobayashi. A dependable peer-to-peer computing platform. *Future Gener. Comput. Syst.*, 23:939–955, November 2007.

[144] Understanding data storage offerings on the windows azure platform. `http://social.technet.microsoft.com/wiki/contents/articles/understanding-data-storage-offerings-on-the-windows-azure-platform.aspx`.

[145] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, 1999.

[146] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41 – 53, January 2004.

[147] D. Zhou and V. Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 66 – 73, April 2004.

Publications

- Lázaro, D., Kondo, D., Marquès, J.M. *Using availability prediction to achieve collective availability in contributory computing communities.* Proceedings of XVIII Jornadas de Concurrencia y Sistemas Distribuidos. June 2011.
- Lázaro, D., Marquès, J.M., Vilajosana, X. *Flexible resource discovery for decentralized P2P and volunteer computing systems.* Proceedings of the Sixth International Workshop on Collaborative Peer-to-Peer Systems (COPS). June 2010, pp. 235-240. ISBN-13: 978-0-7695-4063-4
- Lázaro, D., Marquès, J.M., Vilajosana, X. *Decentralized resource discovery for contributory communities.* Proceedings of XVII Jornadas de Concurrencia y Sistemas Distribuidos. June 2010.
- Lázaro, D., Marquès, J.M., Jorba, J. *Towards an Architecture for Service Deployment in Contributory Communities.* International Journal of Grid and Utility Computing (IJGUC), Vol. 1, No. 3, August 2009, pp. 227-238. ISSN:1741-847X
- Lázaro, D., Marquès, J.M., Jorba, J. *A Best-effort Mechanism for Service Deployment in Contributory Computer Systems.* Proceedings of the third International Conference on Complex, Intelligent and Software Intensive Systems. March 2009, pp .439-444. ISBN: 978-0-7695-3575-3
- Vilajosana, X., Lázaro, D., Marquès, J.M., Juan, A. *DyMRA: A decentralized resource allocation framework for collaborative learning environments.* In Intelligent Collaborative e-Learning Systems and Applications, pp. 147-169 (Springer Series in Studies in Computational Intelligence, Vol. 246, ISSN: 1860-949X). 2009. ISBN: 978-3-642-04000-9. Springer-Verlag. Berlin, Germany
- Lázaro, D., Vilajosana, X., Marquès, J.M., Juan, A. *A Framework for Dynamic Resource Allocation in Decentralized Environments.* International Transactions on Systems Science and Applications, Vol. 4, No. 4, December 2008, pp. 356-366., ISSN 1751-1461
- Lázaro, D., Marquès J.M., Jorba, J. *An Architecture for Decentralized Service Deployment.* Proceedings of the second International Conference on Complex, Intelligent and Software Intensive Systems. March 2008, pp.327-332. ISBN: 978-0-7695-3109-0

- Vilajosana, X., Lázaro, D., Juan, A., Navarro, L. *Towards Decentralized Resource Allocation for Collaborative Peer to Peer Learning Environments.* Proceedings of the second International Conference on Complex, Intelligent and Software Intensive Systems. March 2008, pp.501-506. ISBN: 978-0-7695-3109-0

- Lázaro, D., Vilajosana, X., Marquès, J.M. *DyMRA: Dynamic Market Deployment for Decentralized Resource Allocation.* Lecture Notes in Computer Science. OTM Workshops (1), November 2007, pp. 53-63, ISSN: 0302-9743 (Print), 1611-3349 (Online). ISBN: 978-3-540-76887-6

- Lázaro, D., Marquès, J.M., Jorba, J. *Decentralized Service Deployement for Collaborative Environments.* Proceedings of the first International Conference on Complex, Intelligent and Software Intensive Systems. April 2007, pp.229-234. ISBN: 0-7695-2823-6