INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Descoberta de Recursos em Redes P2P Semi-estruturadas para CPU Cycle-Sharing

Resource Discovery In Semi-structured P2P Networks for CPU Cycle-Sharing

João Pedro Gomes Neves

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Alberto Manuel Ramos da Cunha
Orientador:	Professor Luís Manuel Antunes Veiga
Co-Orientador:	Professor Paulo Jorge Pires Ferreira
Vogais:	Professor Hugo Alexandre Tavares Miranda

Outubro 2010

Acknowledgements

I wish to thank, above all, my coordinator Professor Luís Veiga, for putting up with my inconstant availability and news. His motivation, empathy and knowledge were indispensible for the successful elaboration for this work.

I'd also like to thank my family and friends for all their support during all this time. In particular I wish to extend my most heartfelt thanks to my parents, Carlos Alberto Glória Neves and Rosa Maria Bernardes Gomes Neves, as well as my sister and brother-in-law, Inês Alexandra Gomes Neves and Pedro Miguel Soares da Motta Fraústo Basso.

As for my friends and colleagues, I wish to thank Cláudia Maia, Marta Moleiro, Ana Alves, Pedro Abrantes, Nuno Ferreira, André Santos, André Conrado, Artur Ferreira, Luís Santos, António Ferreira, António Novais, André Nogueira, António Afonso, Filipe Cabecinhas, Andreia Silva, Carlos Fonseca, Inês Carreira, Bráulio Silva, Daniela Loio, Gabriel Barata, Filipe Silva, Filipe Cristovão, Cátia Ribeiro and João Lemos, among many others, for their continued support and friendship. The strength they lent me was a vital contribution for this work.

Lisboa, November 24, 2010
João Neves

I dedicate this work to all my esteemed colleagues who have supported me through all these years, and to my family, for their love and support.

-João Neves

Resumo

Actualmente, existem vários projectos que tentam tirar partido dos ciclos de CPU excedentários existentes em máquinas ligadas à Internet. No entanto, estes projectos destinam-se principalmente à execução de código pertencente a projectos de investigação em larga escala, não dando a possibilidade dos utilizadores normais, que contribuem ciclos para esses projectos, poderem utilizar essa mesma infraestrutura para a execução de aplicações *desktop* comuns para benefício próprio.

No contexto da resolução deste problema, esta dissertação estuda e propõe mecanismos de descoberta de recursos, assim como topologias de redes Peer-to-Peer que permitam elaborar um sistema distribuído de partilha de recursos. Esta solução encontra-se contextualizada no projecto GINGER, que almeja oferecer uma rede de partilha de recursos capaz de interagir com as diversas aplicações *desktop* existentes sem ser necessária a sua modificação, recorrendo a técnicas encontradas em redes *Grid* institucionais, aplicações de partilha de recursos como as mencionadas anteriormente e as populares redes Peer-to-Peer de partilha de ficheiros.

Assim, a solução integra-se no desenvolvimento de uma infra-estrutura de *middleware* Peer-to-Peer centrada no conceito de *Gridlet*, a unidade de trabalho fundamental do sistema GINGER. Esta solução baseia-se principalmente no uso de uma hierarquia de pares na rede semelhante à encontrada em sistemas Peer-to-Peer populares, como a rede Gnutella, para possibilitar a descoberta de recursos de forma eficiente num ambiente Peer-to-Peer.

Abstract

Today, there is a number of projects that attempt to take advantage of the excess CPU cycles existing in machines connected to the Internet. However, these projects are mainly focused in providing distributed code execution for large scale research projects and do not make it possible for normal users, which contribute their cycles for those projects, to use the same infrastructure to execute common desktop applications for their own benefit.

In the context of this problem, this dissertation studies and proposes resource discovery mechanisms, as well as Peer-to-Peer network topologies that allow the creation of a distributed resource sharing system. This solution is part of the GINGER project, which aims to offer a resource sharing network capable of interacting with the many existing desktop applications without the need to modify them, using techniques found in institutional Grid networks, resource sharing systems as the ones mentioned above and the popular Peer-to-Peer file sharing networks.

As such, this solution is integrated in the development of a Peer-to-Peer middleware infrastructure centered around the concept of a *Gridlet*, the fundamental work unit in the GINGER system. This solution is mainly based on the use of a peer hierarchy similar to the one found in popular Peer-to-Peer systems, such as the Gnutella network, to enable efficient resource discovery in a Peer-to-Peer environment.

Palavras Chave
Keywords

Palavras Chave

Descoberta
Recurso
P2P
Pastry
Aplicação
Super-peer
Topologia

Keywords

Discovery
Resource
P2P
Pastry
Application
Super-peer
Topology

Contents

List of Figures

List of Tables

List of Abbreviations

CPU	Central Processing Unit
DNS	Domain Name Service
P2P	Peer-To-Peer
RAM	Random Access Memory
UDDI	Universal Description Discovery and Integration
UPnP	Universal Plug and Play
VoIP	Voice over IP

Chapter 1

Introduction

1.1 Motivation

Over the last two decades and following Moore's Law, the available computing power throughout the world shifted from large mainframes and supercomputers to the ever increasing number of personal computers connected to the Internet, as these became more powerful at a very fast rate. However, most of the computing power offered by these machines is wasted on idle processes, and as personal computers grow more powerful this trend tends to increase.

To use this untapped resource, several projects like BOINC[6] and SETI@Home[7] emerged, giving research institutes the ability to perform massive computations without having to acquire or maintain an expensive grid environment. Through these projects ordinary computers could contribute their unused CPU cycles to scientific research by downloading work units from each project's servers, perform the necessary calculations and uploading the obtained results.

Although these projects approach their desired goals, they fail to meaningfully give back to their contributors, as people who contribute with CPU cycles for these projects are unable to use those same resources for their own benefit.

1.2 Current limitations

The approach followed by BOINC and SETI@Home has some disadvantages, however, including the fact it requires users (in this case, researchers) to be able to compile custom versions of the applications whose work they wish to distribute and as such is not an option for most end-users. Additionally, as previously mentioned, the computing power supplied by the contributors is still in the hands of a very small number of institutions, and users cannot tap those same resources to accomplish tasks that matter to them, as ordinary PC users.

On GINGER[60], the authors propose a peer-to-peer based infrastructure for CPU cycle sharing based on the concept of Gridlets, which allow for the use of unmodified applications on the data to be processed, and as such enables every user not only to contribute by helping other users on their computations but also allows them to submit their own work to the environment, in a mutualistic manner. Such an approach requires each node in the network to be able to locate the required resources for the work it needs to do in an efficient and scalable manner, but empowers each user with the ability to use the available cycles to perform everyday tasks which would usually require very large amounts of computing power or

time.

The resource discovery problem in institutional Grid infrastructures and distributed cycle sharing projects such as the ones mentioned above is relatively well known and several centralized solutions have been proposed. On a fully decentralized peer-to-peer architecture, however, this problem is further amplified by the lack of global knowledge of the nodes composing the network, and a scalable solution for the general case is difficult to achieve.

Beyond the issue of discovering resources like CPU and disk storage, it is also necessary to tackle the problem of discovering services, which may be provided by a software application present in a system, or physically through the machine itself (e.g. a network printer).

To solve this problem, a number of service discovery frameworks such as JXTA[26], Jini[62], SLP[27] and UPnP[47] have been proposed in the context of small-scale networks, most of the time resorting to central servers (e.g. UDDI), but some also resorting to existing network infrastructure (e.g. DNS Service Discovery[2]) or avoiding them altogether while applying the same techniques found in proven Internet technologies (e.g. Multicast DNS[3], Zeroconf[4] and Avahi[1]).

Additionally, there is a scheduling problem to be solved. Ideally, all jobs should be executed on a reasonable amount of time, and they should be executed in the most appropriate locations, according to some set of criteria.

For instance, it may neither be desirable nor fair or even resource- efficient to keep loading a single machine or group of machines just because they are available and fit or exceed all the requirements.

1.3 Contributions

This work introduces a resource discovery mechanism for P2P networks based on the Pastry overlay, more specifically in the context of the GINGER platform. It aims to evaluate and implement several resource discovery mechanisms to be used on the GINGER platform.

A resource in this system can be any of the following: CPU power and availability, memory capacity, disk capacity, network bandwidth, installed applications, services, peripherals, users and others that may apply.

The chosen mechanism should ideally be:

1. **Scalable:** The overall efficiency should not decrease significantly with a larger network;
2. **Complete:** If the resource to be discovered exists, it shall be discovered;
3. **Efficient:** The overhead caused by the discovery protocol should not disrupt the normal functioning of the network;
4. **Flexible:** We aim to support multi-attribute ranged queries, as it is necessary on such a heterogeneous environment.

1.4 Document Organization

This paper is divided into six chapters. The first and current chapter introduces the problem at hand, why it is important, and what contributions should be expected as a result of this work.

The second chapter details previously developed work related to the problem, important conclusions taken from that work and how they may be relevant for the problem at hand.

The third chapter will detail a high-level architecture chosen for this project, the design decisions that were taken as well as the rationale behind those decisions.

The fourth chapter describes the implementation of the proposed system, as well as specifying the technologies used in the elaboration of this work and the differences between the proposed architecture and the actual implementation.

A fifth chapter deals with the evaluation of the implemented work, such as the tests that were developed for this work and metrics that were obtained.

Finally, in the sixth chapter, we draw our conclusions on the results obtained in this work and propose future work to improve on our own.

1.4.1 Terminology

Throughout this paper, and although some of the cited authors interchange these terms, the following definitions will be used:

- Resource - a physical, generally constrained attribute of a given system (e.g. CPU power, available RAM, disk capacity, etc.)
- Service - a functionality provided by a given system (e.g. a printing service, provided by a printer) or, as is most commonly the case, a tool or application (e.g. a software application installed in a given system)

Chapter 2

Related Work

2.1 Introduction

Several articles and projects have been produced on the topic of resource and service discovery on institutional grids, cycle-sharing desktop grids and peer-to-peer networks. In this section, some of this work is presented and discussed.

We will focus on a few aspects of each system that are relevant for their study in general and worthy of note in the scope of this project. The most important aspects pertain to overall system architecture and resource discovery.

2.1.1 Institutional Grids

Overview

Grid computing is used to describe a technique that utilizes the resources of several computers simultaneously in order to solve complex problems which typically require very large amounts of computation power due to either the nature of the problem or the sheer volume of data that needs to be analyzed.

Institutional grid networks have been the main form of distributed computing platforms available worldwide, mainly to large research or academic centers.

Grid network environments are usually centrally controlled by a single entity, which makes these environments implicitly trusted, and as such, less vulnerable to malicious use. This allows for a simpler or non-existent result validation procedure.

Applications

Grid networks are commonly found in the domain of high performance computing and manipulation of very large datasets replacing traditional, monolithic supercomputers.

This is partially due to the fact that supercomputers, while taking advantage of faster interconnects between their components, are often custom-made and thus more expensive and of difficult maintenance. Grid networks, on the other hand, while incurring in some potential performance penalties, are often built out of commodity parts, which are easily upgradeable at an affordable price.

The usage of grid networks is not limited to the execution of computation over data and can be used for distributed data storage as well (e.g. Amazon's *S3* "cloud" storage infrastructure).

2.1.2 Cycle-sharing Desktop Grids

Overview

Cycle-sharing desktop grid networks are built using regular desktop computers, whose idle CPU cycles are often voluntarily donated by regular computer users. These cycles are often used to perform distributed computations on very large data sets for research purposes.

BOINC, with over half a million active computers providing an average of over 1400 TeraFLOPS of computing as of the time of this writing, is a clear proof that users are willing to provide their idle cycles and that there is an enormous amount of processing yet to be exploited worldwide. The emergence of the use of GPUs for massively parallel processing is expected to greatly increase the total throughput of this kind of grid.

Applications

Desktop grids include well known systems such as BOINC, SETI@Home and Distributed.net. The type of projects run in these grids can vary greatly. BOINC hosts a very large number of projects spanning several areas of interest, including protein folding simulations, cryptography and game playing.

SETI@Home specializes in the analysis of data obtained through radio telescopes for the purpose of detecting extra-terrestrial intelligence by searching the data for patterns indicating intentional emissions.

Distributed.net was one of the first desktop grid networks to appear and focuses on projects regarding cryptography, mainly attempting to brute force well known ciphers in order to assess their resilience against such attacks. These projects are usually sponsored by companies who develop the ciphers.

2.1.3 Peer-to-Peer Networks

Peer-to-peer (P2P) networks have become one of the driving forces of widespread online content production and consumption worldwide. Unlike traditional client-server networks, P2P networks enable cooperation between users without the existence of centralized, third-party authorities and effectively removing the need for costly infrastructure as well as eliminating the single point of failure present in traditional systems.

In a P2P network, as its name implies, every client can also act as a server without requiring any additional configuration or special setup. Peer coordination in a P2P network is essentially free and can be achieved in an autonomous manner, without the need for special peers in the network, although in some cases it is desirable to have a number of peers empowered with extra capabilities for the purposes of coordination, commonly known as super-peers or ultra-peers. Thus, P2P networks can be accurately described as a self-organizing set of nodes.

The decentralized and distributed nature of P2P networks makes them highly resilient to the common issues affecting traditional centralized systems. In particular, P2P nodes are highly resilient to Denial-of-Service attacks, distributed or otherwise, and typically enjoy increased efficiency with a greater number of peers present in the system at any given time.

Nodes participating in a peer-to-peer network are not necessarily actively contributing to the overall network at every point in time and as such it is possible to aggregate unused capacity existing throughout the network and apply load balancing techniques to prevent overloading only a subset of the nodes.

Some peer-to-peer networks organize their nodes according to a given topology, usually distinct from the underlying physical topology, in order to better achieve their goals in terms of robustness, performance and scalability.

Some of these Peer-to-Peer networks, such as Napster[31] and BitTorrent[16] are not fully decentralized, however, as they rely on one or multiple servers to locate the files they wish to share, and to advertise their own files. This makes those networks more vulnerable to Denial-of-Service attacks as there is a central location with which peers must communicate in order to find the required resources. Nonetheless, for all intents and purposes, these networks retain most of the characteristics of a pure P2P network and are considered one of them.

A type of peer-to-peer network designed to solve this problem can be a Distributed Hash Table or DHT, where nodes are responsible for a subset of all the keys in the network's key space and cooperate to provide scalable and efficient key location.

Large scale implementations of DHTs are rarer than other types of P2P networks, although some BitTorrent clients already implement a DHT protocol (referred to as the "Mainline" protocol) to decentralize the network in a way that no server (known as a "tracker") is required to locate other users with the same file.
Well known DHTs include Pastry[52] and Chord[57], which organize the nodes in a logical ring and route requests through other nodes in the ring.

P2P networks on the whole comprise a large part of all Internet traffic and as such have proved to be scalable, with room for future growth.

Applications

Several P2P networks already exist deployed on a wide scale for various purposes. The following are a few of the most relevant applications of P2P networks today.

File-sharing P2P networks are commonly used for the purpose of file-sharing. The most widely used protocols for this purpose include Gnutella, Kazaa[30], Napster[31] and BitTorrent[16].

Over these networks users can search and download files to their computers through other users that possess the same files (totally or partially), while sharing their own files. BitTorrent in particular is also known to be used in order to alleviate server and bandwidth costs associated with the distribution of extremely popular files, such as Linux distributions or video game updates which often weigh in the order of the hundreds of megabytes.

P2P traffic through file-sharing was, as of February 2009, the largest type of traffic seen on the Internet across all regions and BitTorrent in particular has enormously boosted the use of P2P networks and continues to be one of the top protocols in use for the purpose of file-sharing worldwide[54].

Streaming audio and video In recent years, with the rise in popularity of video streaming services such as YouTube, a number of attempts to exploit P2P networks for the purpose of real-time streaming of video and audio have come into place.

The main issue that needs to be solved is the fact that this kind of usage poses a number of problems which do not exist in the file- sharing scenario. File-sharing is inherently a problem of distribution of time insensitive content, whereas streaming is the exact opposite. Distribution of time sensitive content presents a specific set of constraints that are not usually taken into account when developing a P2P network, such as the timeliness for the arrival of a given packet at a receiving peer.

A number of approaches have been proposed to deal with these issues. Vlavianos et al propose BiToS[61], which seeks to adapt the BitTorrent protocol for use in streaming video and audio content. The authors concede the point that BitTorrent in particular is ill-suited for the purposes of live data streaming (such as VoIP calls or live video broadcast) since the pieces are not known *a priori* and the piece selection mechanism used by BitTorrent cannot be used as-is, as well as a number of other issues regarding piece advertisement.

However, for the purpose of streaming pre-existent content, the authors propose a number of small changes to the BitTorrent protocol which enable it to accommodate this kind of use. These changes include replacing BitTorrent's default "rarest first" piece selection algorithm by a "highest priority first" algorithm as well as adding mechanisms to determine whether a given piece would be downloaded in a timely manner.

Live streaming of data can still be achieved using P2P networks and a number of systems have been developed for this purpose, such as PULSE[49], CoolStreaming[65] and AnySee[37].

Multiplayer online video games Another rising use for P2P networks is the organization of players in multiplayer online video games.

Traditionally, multiplayer online gaming requires one machine to act as a host where player machines connect in a typical client-server manner. Hosts are normally split into two types: dedicated and undedicated hosts. Dedicated hosts only serve the game to other players and act as a hub to synchronize and exchange the game state between those players. On the other hand, undedicated hosts are typically a local server running on the same machine as one of the clients. Both of these approaches have key advantages and disadvantages.

Dedicated hosts usually offer better performance due to being run in dedicated machines connected to a high bandwidth connection, but often incur costs to either the game developer, which now has to maintain these servers, or to the players who opt to rent such a server. Additionally, this kind of server can be considered to be always available, so clients only disconnect when they so desire.

Undedicated hosts are subject to lower performance due to the nature of traditional desktop computing environments which have to perform a number of different tasks on a traditionally lower performance machine, usually connected to a typical broadband home connection. Since hosts are created by other players, there is also no guarantee the gaming experience will be smooth, since it is subject to the capabilities of the hosting machine. This causes client players to experience greater amounts of network delay and to be subject to the game host's will to maintain the game running. When the host player leaves the

game, the hosting server is usually shut down as well (we'll refer to this issue throughout this section as *orphaning*). The main advantage of this method is the basically zero cost of hosting the game.

P2P networks used in the context of online gaming can solve most of these problems at once, while incurring in only insignificant penalties. By connecting users in a P2P environment, game developers are not required to host the game servers themselves and through a smart selection algorithm it is possible to select the best host among the peers currently connected to a given gaming session.

This technique has been used in a number of high profile gaming titles such as Stardock's "Demigod" and more recently Activision's "Call of Duty: Modern Warfare 2".

In both cases, a P2P connection is established among all players in a given session, and one of the peers is chosen as a host, with which all other peers exchange game state information. Whenever the currently hosting peer leaves the game, any other peer can be selected as the next host, as all peers would know the same game state. The new host is selected through the use of metrics such as CPU power, latency and possibly others (the actual metrics used in commercial games are not disclosed in detail by the developers) in order to minimize latency for all players in the current session. The actual game needs only to be interrupted for a few moments as the next best peer is selected.

In this manner, the gaming session can remain active as long as at least a single peer remains connected, eliminating the orphaning problem completely. Through the use of a selection algorithm as described above, the issue of low performing hosts can be greatly alleviated, although not completely eliminated.

All these types of infrastructure have their own merits and demerits and have been subject to extensive study in order to provide clues on how they can be combined to take advantage of the best features each system has to offer[58].

In the next sections, we will detail all three platform types mentioned above.

2.2 Institutional Grids

Several systems have been designed for use in institutional grid infrastructures to serve various purposes. One of the earliest was Condor[38], which provided the ability to harvest idle cycles on a workstation network to perform various tasks. Several other systems later appeared with the same purpose, such as the Globus Toolkit[25] and OurGrid[8], enabling the construction of complex grid environments.

These projects focused mostly on resource discovery, scheduling, interoperability and transparency, however, and several other systems were developed to enable service discovery[46] or knowledge discovery through the joint use of data-mining algorithms and very large data sets[17].

2.2.1 Architecture

Condor[38] was one of the first systems designed to harvest idle workstations and introduces the concept of local schedulers and coordinators.

Workstations possess a local scheduler, responsible for ordering jobs according to their relative priority, while a central coordinator is responsible for actively seeking workstations with available computing

capacity and informing the other workstations whenever a workstation is "offering" its idle cycles. Condor has characteristics strongly reminiscent of desktop grids (the entire system runs on workstations where users can locally connect to perform their own work) and of peer-to-peer networks (any workstation can be both a client and a server, although there is only one server at a time), but since Condor depends on the server to push new work loads to the clients (see Section 2.2.2) and because of the context in which it was originally introduced in, it can be classified as an institutional grid system.

The typical grid network, as provided by the Globus Toolkit[25] and others, have a centralized network topology where a small number of machines act as directory services and resource schedulers.

On Globus, work requests are submitted to a resource scheduling service (Global Resource Allocation and Management) by client workstations, which then allocates the work units to the required machines. These are located through a directory service called the Metacomputing Directory Service, which aggregates and manages information on all available resources throughout the grid.

OurGrid[8], on the other hand, tries to alleviate the need for a user to setup his own grid environment, which is a requirement for systems like the ones provided by Globus, while maintaining the main characteristics of a regular grid infrastructure.

Users run clients which schedule applications and connect to "OurGrid peers", which provide access to other resources made available in the grid (possibly a Globus infrastructure, for instance). OurGrid peers are comprised of two modules: a *consumer* module, which accepts requests from clients and is responsible for locating the necessary resources on the grid and later execute the requested jobs; and a *producer* module, which manages and makes the resources available for the consumers.

2.2.2 Resource discovery in Institutional Grids

Resource discovery in grid environments is usually achieved by the existence of centralized Directory Services, which contain and manage information on all the available resources on the grid. Clients who wish to locate workstations in the grid refer to this directory service.

Some of the most relevant projects in this area follow:

Condor

Condor, as previously mentioned, uses a coordinator to schedule tasks to idling workstations automatically, so users don't have to actively search for workstations that meet a search of requirements. Since the workstations are assumed to be fairly similar among themselves, resource discovery in Condor is based mostly on determining CPU power and availability.

Every workstation possesses a local job scheduler and a background job queue, and one of the workstations also maintains a coordinator service, which is responsible for allocating capacity from idle workstations to local schedulers on workstations that have waiting background jobs.

In the original Condor implementation, the central coordinator actively polls the existing workstations to see if they are available to serve remote cycles and which stations have background jobs waiting. Between each poll, each local job scheduler also monitors its station to see if it is available as a source of remote cycles.

This approach is semi-centralized, as it requires a special node to act as a coordinator, but such is only required for locating workstations available to perform work. In the case the coordinator is taken offline, all existing work units are handled normally by the local job schedulers, adding to the overall system robustness.

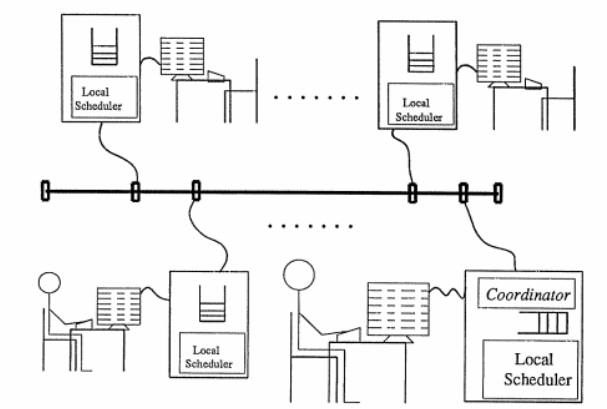

Figure 2.1: Condor architecture

Globus

In the case of the Globus Toolkit, resource discovery is provided by the previously mentioned Metacomputing Directory Service, or MDS. The MDS can store various types of information such as configuration details (CPU speed, network interfaces) and instantaneous performance information such as network latency, among others. This information can be stored as key- value pairs for each known node in the system, and is usually maintained within conventional LDAP (Lightweight Directory Access Protocol) servers.

The Globus Toolkit allows for the entire resource discovery and selection process to be exposed to, and guided by, higher-level tools and applications. The toolkit exposes three mechanisms for such purposes:

- *Rule-based selection* - Through this mechanism, Globus modules can specify a set of rules that allows them to choose a particular resource from many alternatives. Additionally, a rule replacement mechanism allows for a module developer to specify alternative selection strategies for the case where the default rules do not provide the required resources.
- *Resource property inquiry* - This mechanism allows a user to provide rules that perform resource selection while taking into account specific variables. For instance, a user can provide a rule in the form of "use ATM interface is load is low, otherwise Internet" in order to guide resource selection by taking into account network load.
- *Notification* - The notification mechanism allows for the specification of constraints on the quality of service provided by a given Globus service and a callback function that should be invoked if such constraints are violated.

The use of these mechanisms ultimately allows computations to be configured efficiently for the available resources and to adapt those computations or other behaviors dynamically when the availability or quality of such resources change, providing greater flexibility.

OurGrid

On OurGrid, clients (applications that manage access to the grid's resources) contact OurGrid peers to locate resources on the grid.
Upon receiving a client request, the consumer module of a peer will proceed to locate the required resources by broadcasting a message containing the desired resources through the other peers to their producer modules, including its own peer. All producers with the available resources that fit these requirements reply to the sending consumer, effectively defining the available grid environment for that job.
The consumer then passes this information to the client application which is responsible for scheduling tasks onto the providers using an arbitrarily chosen algorithm and communicates with the consumer that originally requested the resources to send the tasks.

This allows the consumer to account all the resources it consumes on behalf of other clients and permits for the construction of a favor-based system.

2.2.3 Service discovery in Institutional Grids

Service discovery is usually done in the same way as resource discovery, through the use of a centralized directory server which provides information about the location and properties of a given service.

Services publish themselves on the directory service periodically for as long as they are available

Several different systems have been developed for the purpose of service discovery, independently of its application on grid networks, as is the case with JXTA, Jini, UPnP and UDDI, but can be used in this context.

This kind of systems, however, is generally not scalable to an very large number of nodes, and their application is restricted to institutional grids and smaller networks.
In [46], the authors introduce UDDI-M$^{\mathrm{T}}$, an extension to the UDDI infrastructure that allows the association of metadata to regular UDDI records in order to describe service-specific attributes such as quality of service, reliability of that service, among others.
UDDI-M$^{\mathrm{T}}$ works transparently with existing UDDI infrastructures, acting as a gateway that filters incoming messages to store relevant information on its own back-end while sending all the necessary information to the UDDI back-end. This system enables the extension of client queries in order to find services that match the stored metadata, and further extends the UDDI infrastructure by adding leases to force services to republish themselves regularly.

2.3 Desktop Grids

Desktop grid systems enable the use of ordinary desktop computers to perform distributed computing.

One of the most popular desktop grid systems is SETI@Home[7], which later evolved into the BOINC[6] desktop grid system, which allows multiple, heterogeneous projects to take advantage of the same desktop grid environment comprised of thousands of ordinary desktop computers around the globe.

The SZTAKI Desktop Grid[34, 10, 40] builds on BOINC to enable the scheduling of BOINC tasks on cluster infrastructures by manipulating incoming tasks and converting them to regular cluster jobs.

Cluster Computing On the Fly(CCOF)[66] builds an overlay network for the purpose of CPU cycle sharing in non institutional environments. Although it is a desktop grid system, it shares many properties with Peer-to-Peer systems, namely the fact that nodes communicate directly with each other to share resources, only using the overlay for resource location and discovery.

Lastly, the commercial Entropia[15] system distinguishes itself by leveraging existing, unmodified applications to perform work in a desktop grid environment. This last system will not be discussed in depth due to much of its functionality uses proprietary techniques and is not described in detail.

2.3.1 Architecture

Desktop grids, as described above, are usually built with regular, off-the-shelf PCs. These computers regularly connect to a centralized server to get new jobs to work on, freeing that server from the task of periodically querying clients for availability.

Since every client is outside of the grid's administration control in terms of availability, it's often preferable to use this type of *pull* approach as opposed to the *push* approach used by traditional grid systems such as Condor.
The architecture for some of the most prominent desktop grid systems is described in the next sections.

BOINC

BOINC uses a client-server architecture, where the server side possesses a relational database in which are stored diverse pieces of information regarding each project hosted by the platform, such as registered users, available work-units, as well as sent and received work-units.

Each project has a number of different components. A back-end, which is responsible for distributing work-units and process the results sent by the clients; a data server, responsible for distributing each work-unit's data files and gather their respective result files; the scheduler, which controls the work-unit delivery to each client according to their productivity, and finally a Web server which allows for each project to serve its own pages for participant interaction.

On the client side, there is only a core BOINC component and a project specific "science application", which processes incoming work-units.

A schematic of this architecture can be seen in Figure 2.2. In BOINC, new jobs are requested by the clients to the scheduling server, which determines the best work-unit for a given client, taking into account which projects the user has selected to contribute for. Additionally, each client makes its own local scheduling to try to meet performance metrics and deadlines for each individual project it is registered on.

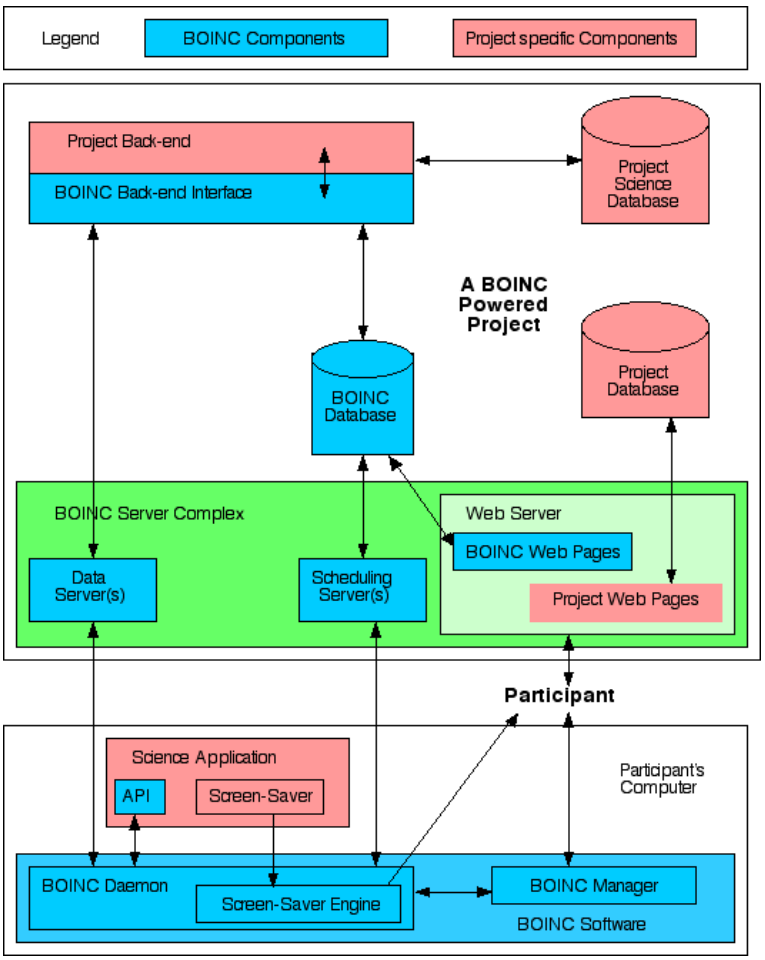

Figure 2.2: BOINC's client-server architecture

Cluster Computing On the Fly

In CCOF, nodes organize themselves in communities through the use of overlay networks. Nodes associate themselves to each of these overlay networks depending on how they'd like their idle cycles to be used.

The architecture in this system, as shown in Figure 2.3, is comprised of the following components:

- Community manager
- Application scheduler
- Local scheduler

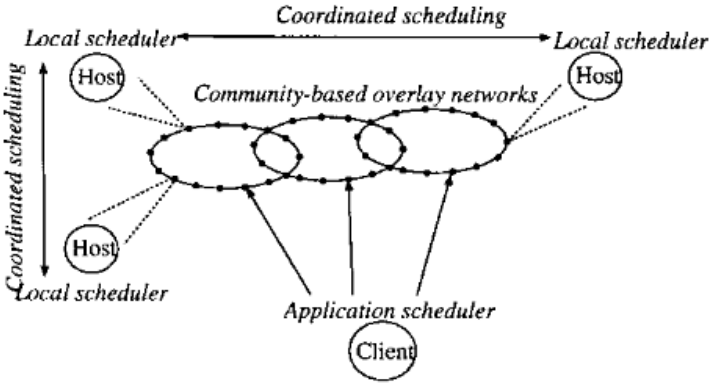

Figure 2.3: Cluster Computing On the Fly

SZTAKI Desktop Grid

The SZTAKI grid, being derived from BOINC, employs the same base topological arrangement as that system, but since its goal is to leverage existing cluster infrastructures for desktop grid computing, a few changes are required on the client side.

In a desktop grid, as previously discussed, client machines connect to the server and request work units that they may process. Typically, each client only requests a limited number of work units at a time, as it usually cannot process many at any given time. In SZTAKI, a cluster may request a much larger number of work units as if it were a single powerful machine, from the server's point of view, and then redistribute those same work units as cluster jobs among all the machines using a local Job Manager.

This architecture is displayed in Figure 2.4.

2.3.2 Resource discovery in Desktop Grids

Different desktop grid systems typically employ different strategies for resource discovery. Let us explore the strategies employed in two of the most well known systems, BOINC and CCOF:

BOINC

BOINC uses the concept of *work-unit*, which is defined on a per project basis and consists on a set of actions to be performed on the data as well as some requirements regarding CPU capacity or network bandwidth, for instance.

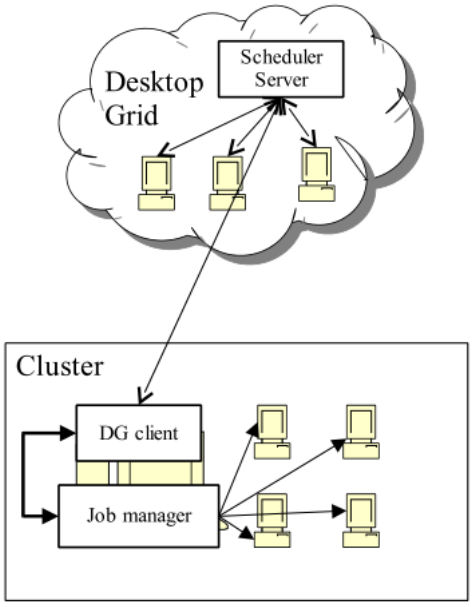

Figure 2.4: The SZTAKI Desktop Grid

Work-units are managed by the server's scheduling component, which the client machines query for pending work on a given project. If there are work-units available for that project, the scheduling server will send a number of those work-units to the client and await for the results until they are sent by the client or a specified deadline is reached.

As the work-units in a given project all have the same cost, or weight, there is no particular criteria that the scheduling server applies before handing work-units to a client. It is up to the client's local scheduling service to determine which projects it will fetch work-units from and to schedule which work-units should be executed next in order to achieve a number of goals:

- Maximize resource usage;
- Meet result delivery deadlines;
- Maintain a certain degree of variety among projects;
- Respect the resource share allocation determined by the participant.

Cluster Computing On the Fly

Cluster Computing On the Fly uses a decentralized approach to resource discovery by using profiles to generate resource information on the machines. A profile is predetermined according to some assumptions (like for instance the fact that most users tend to use their machines in the morning and having it idle during most of the remainder of the day) and is applied equally on every node. Nodes in the network are then searched for the required resources and jobs are given to some or all of these nodes after being picked by the job submitter.

CCOF also served as a simulation platform to test four different search methods for resource location: *expanding ring search*, *advertisement based search*, *random walk* and *rendezvous point*.
Ultimately, the rendezvous point method was determined as the one with the best overall performance

both in low and high computational load situations. This method uses a dynamically created group of nodes, called rendezvous points, to which regular nodes announce their profiles to. Client nodes contact these rendezvous points to obtain nodes with available resources according to the required profile.

When a node is selected as a rendezvous point it announces its new function to all nodes it is connected to within a given number of hops.

2.3.3 Service discovery in Desktop Grids

Services in desktop grids can be seen as the applications a client runs on his desktop. For instance, in BOINC, a client offers his cycles to specific projects, providing a service to those project in detriment of others.

From this perspective, service discovery in BOINC is done in a centralized manner, by scheduling work-units from a given project to client who have registered their machines to that particular project. CCOF offers no distinction between resources and services, which means available applications can be simply as another attribute included in the description of the system.

2.3.4 Security in Desktop Grids

Desktop grids, due to the fact that they run on regular, out of the box desktop computers usually owned by regular computer users and as such interact with a far more complex and unpredictable software and hardware ecosystem than on a traditional grid system, must employ additional measures to protect both the host system and the desktop grid infrastructure as a whole.

In order for a desktop grid to work, the system must be resilient to tampering by malicious desktop providers to prevent result falsification and cheating, as well as prevent malicious code from running in the desktop computers through the desktop grid applications. Failure to do accomplish either of these aspects would break the existing level of trust between all participants and render such a system unusable.

Let us analyze how the previous systems deal with some of these issues.

BOINC

In BOINC, result falsification is a great issue, since as the authors discovered during their work on SETI@Home, participants can be highly motivated by the use of "credit", a scoring unit defined by a combination of computation, storage and network transfer. There is a number of leaderboards that display ranking among participants by taking into account their amount of credits, and as such there is enough incentive for participants to attempt to cheat by awarding themselves extra credits by forging results.

In order to deal with result falsification, BOINC employs a redundant computing mechanism that identifies and rejects erroneous results. Each BOINC project can specify that a number of results should be created for each BOINC work-unit. Once a number of these results have been obtained, they are compared and a canonical result is obtained through consensus. If no consensus is found, BOINC would create new results for the work-unit and repeat this process until the result or time limit is reached.

In this manner, malicious participants can still game the system by obtaining large numbers of results

and attempt to form a consensus using a number of them, but since each user can only send at most one result of a given work-unit, gaming the results this way is exceedingly difficult.

As for securing the client machines against malicious code, the only safeguard presented by BOINC is the fact that all code is made specifically for each project by the respective research teams and cannot run outside the BOINC environment, and as such offers no real security from malicious code, whether it is intentionally malicious or not.

Entropia

Entropia offers an interesting approach to security, by applying sandboxing, encryption and binary modification to achieve the aforementioned security goals.

In order to have some degree of control over the interaction between the grid application and the desktop computer, Entropia employs binary modification in order to intercept Windows API calls. The application is run on a sandboxed environment provided by virtualization techniques, thus disallowing the grid application from directly modifying the contents of the desktop computer's disk, Windows Registry and other system resources. An application that attempts to write to a physical disk location will in fact to a sandbox directory controlled by Entropia, and calls to Windows API functions that should not be necessary for the normal functioning of the sandbox application (e.g. API calls that allow for shutting down the desktop computer) are not allowed.

In order to protect the grid environment from result falsification, the Entropia sandbox environment applies encryption to all managed applications, as well as their data and result files. This prevents third-party applications from inspecting or tampering with the application directly.

The sandbox also automatically checks data integrity for the application and its files to detect tampering and will reschedule the tampered work unit on another client, to prevent further tampering attempts.

2.4 Peer-to-peer networks

In a peer-to-peer network, all connected entities, referred to as "nodes", act both as a client and server, distinctly from the traditional client-server architecture present in many networks, where there are many clients for a single server.
A number of systems have been developed using this kind of networking for several purposes, like file-sharing (e.g. Napster, Kazaa, BitTorrent) and resource discovery (e.g. INS/Twine[11], XenoSearch[56]).

In the following sections, several aspects of peer-to-peer networks will be detailed.

2.4.1 Topology

In many of these peer-to-peer networks, all nodes are considered equal, having equal shares of responsibility throughout the system, although in some cases it may be desirable to have a relatively small number of promoted nodes, commonly known as *super-peers* or *ultra-peers*.

These super-peers can act as brokers or gate keepers for the remaining nodes, and generally possess a more accurate knowledge of the system. In networks such as Gnutella2, for instance, super-peers act as indexing servers which other nodes can consult to locate files.

Super-peers can in this way provide some of the efficiency of traditional centralized networks while maintaining the robustness and flexibility inherent to peer-to-peer networks nearly intact. In [63], the authors describe several techniques that can be used in the design of super-peer networks in order to maximize the efficiency of the network.

P2P networks can be generally split into two categories: **unstructured** and **structured** networks. Structure in a P2P network can be seen as a logical arrangement of the participating nodes and their connection to each other, which is often completely unrelated to the underlying physical network and geographical distribution of nodes. This logical arrangement is often referred to as the **overlay** network. Unstructured networks, like the name implies, have no predefined structure and links between nodes are

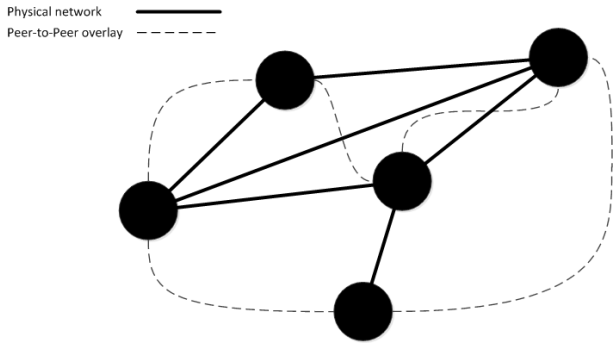

Figure 2.5: Sample overlay network, and its underlying physical network

generally formed in an arbitrary manner.
There are cases, however, where structure naturally emerges as a consequence of high popularity among certain nodes which contain frequently accessed resources (in the case of file-sharing networks, nodes containing very popular files obtain this status, for instance).
Examples of well studied unstructured P2P networks include Gnutella[35], Napster[31] and most recently, BitTorrent[16].

Structured P2P networks, on the other hand, tend to have a given structured, predetermined by design constraints. The typical use of a structured P2P network is to construct a distributed hash table, or DHT.
DHTs use key-value pairs to map resources to nodes. Keys can be generated in many ways, and usually consist of an m-bit identifier generated by a hash function applied to files or node IP addresses.

There are several possible topologies in this kind of networks.
In Chord[57], nodes are organized in a logical ring and possess a m-bit unique identifier. Each node has a *finger table* which contains routing information of up to m other nodes in order to increase the lookup efficiency. Each node is responsible for every item which key is between that node's key and its predecessors.
Lookups are performed by calculating the key of the desired resource and then routing the message as if trying to locate a node with that key. The result of that lookup will be the node which currently holds the resource with that key.
An example of this topology is shown in Figure 2.6.

CAN[50], short for Content Addressable Network, organizes its nodes in a k-dimensional Cartesian space. Nodes are responsible for a region of the space (in a 2-D plane, nodes are responsible for areas of that

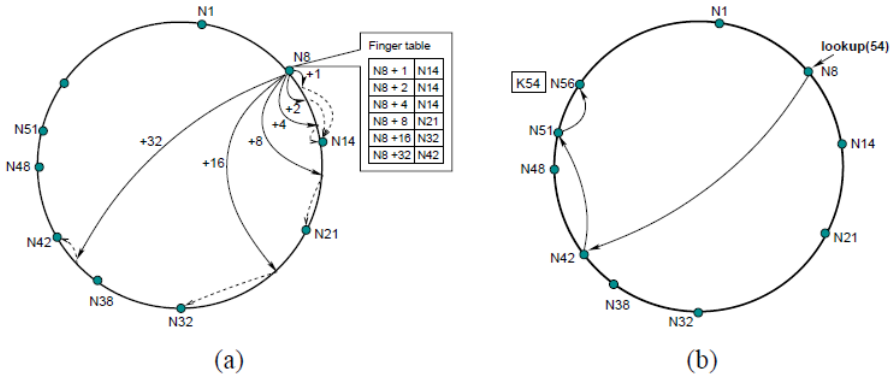

Figure 2.6: a) Chord network with finger table for node 8. b) Lookup path for key 54.

plane, for instance, as depicted in Figure 2.7) , and as nodes join and leave the network, existing nodes split their areas at any given moment.
Upon joining, nodes randomly pick a point in space and send a request for that node to a previously known node. This request is then routed to the node owning the zone in which that point lies. That zone is then split equally between that node and the new one.

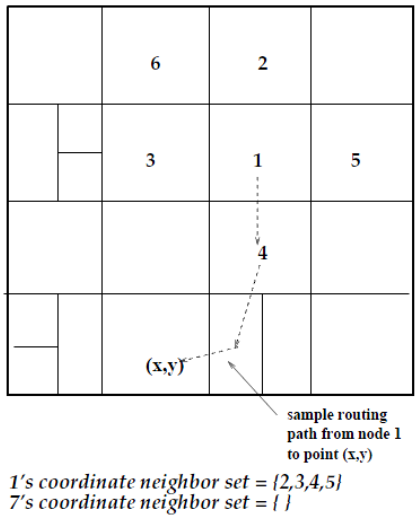

Figure 2.7: An example 2-dimensional CAN overlay.

Routing in CAN is performed by a simple greedy algorithm that routes messages to the current node's neighbor which is closer to the destination point. Two nodes are considered neighbors if they overlap along $d-1$ dimensions and are adjacent along only one dimension. In Figure 2.7 , nodes 1 and 5 are neighbors, but 1 and 6 are not. There are several possible paths along the overlay, as a node may have several neighbors, and this property is useful if the best possible neighbor is found to be dead, as the message can be routed through another neighbor.

Kademlia[44], on the other hand, uses a tree-shaped configuration by arranging nodes into i *k-buckets*, which consist of lists of tuples containing the IP address, UDP port and ID for nodes with a distance between 2^i and 2^{i+1} from itself, where i is an integer between 0 and m, with m being the number of key bits.

These *k-buckets* are kept sorted by the last time each node was "seen" by the current node. The *k-buckets* will generally be empty for small values of i (as no nodes with the appropriate ID will be present), but for larger values they can grow up to size k, which is a value chosen such that any given k nodes are unlikely to fail within an hour of each other.[44]
The *k-buckets* are updated whenever a node receives a message from another node, thus updating the bucket corresponding to the sender's ID, and a node is only removed from its *k-bucket* if it fails to respond. If a *k-bucket* is full and the least recently seen node is still alive when a new sender's message is received, then the new node's information is discarded.

Node distance in Kademlia is based on the XOR metric, which defines the distance d between two nodes with IDs x and y as $d(x, y) = x \oplus y$. Node IDs are similar to the ones used in Chord and can be calculated using the same methods.

Lookups are performed by selecting a small number of the closest known nodes to a given key and recursively looking for the k closest nodes to that key. When looking up for a key-value pair, the procedure stops as soon as a value is returned (as nodes can return the value if they possess it, rather than returning a triple as mentioned above).
To provide redundancy, values are stored in up to k nodes, and values are periodically replicated from nodes that already store those values by locating the k closest nodes to that value's key.

It is also possible to mix topologies, as in Pastry[52], where a tree-like topology is used for general routing, whereas a ring-like topology is assumed to route when we are close to the destination or the tree-like approach fails to find an appropriate node.[45]

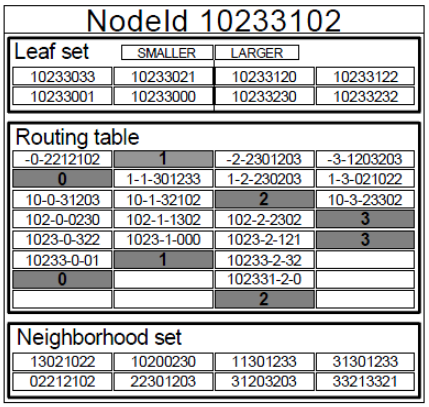

Figure 2.8: Possible state of a Pastry node.

Pastry possesses a rather complex set of routing entries (seen on Figure 2.8) . Its state contains a *leaf set*, a *routing table* and a *neighborhood set*.
The leaf set contains the set of nodes with the $|L|/2$ numerically closest larger and smaller nodes to the current node ID, where $|L|$ typically is 2^b or 2^{b+1} and b is a configurable number chosen such that it provides a good trade-off between the size of the populated portion of the routing table and the maximum number of hops required to route between any two nodes. It is used during message routing when the current node is close to the message's destination.

The routing table has $\lceil \log_{2^b} n \rceil$ rows with $2^b - 1$ entries each. Entries at row n of the routing table refer to a node whose ID shares at most the first n digits with the current node. Multiple nodes may

share this common prefix, and to maintain good locality properties, the closest one according to the proximity metric is the one chosen for the entry.

The neighborhood set points to a set of nodes that are the closest to the current node according to the proximity metric. Unlike the routing table and the leaf set, this set is not used in routing messages most of the time, but it is useful in maintaining locality properties.

Routing in Pastry is done by first trying to route through the neighborhood set, and if the destination node is not encompassed by it, then route through the routing table. In the rare case the message cannot be routed through the routing table, it will be routed through any known node such that its ID shares a prefix with the key at least as long as the current node and it is numerically closer to the key than the current node's ID.

Pastry is also extremely robust and resilient to concurrent node failures, as eventual delivery is guaranteed unless $\lfloor |L/2| \rfloor$ with adjacent IDs fail simultaneously. This makes it very hard to significantly disturb the network through node failure. Also, at worst, the required routing steps will be at most N, while the nodes are updating their state, where N is the number of nodes currently present and alive on the overlay.

A comparison of these DHT topologies is located on Table 2.1.[45, 59] The main advantage of structured

	Chord	CAN	Kademlia	Pastry
Topology structure	Ring	Cartesian spaces	XOR-tree	Hybrid
Lookup cost	$O(\log n)$	$O(d \cdot n^{1/d})$	$O(\log n)$	$O(\log n)$
Join/leave cost	$\log n$	$2d$	$2^b \cdot \log_{2^b} n + 2^b$	$2^{b+1} \cdot \log_{2^b} n$
Number of entries/node	$\log n^2$	$2d$	$\log_{2^b} n$	$\log_{2^b} N + c$

Table 2.1: Comparison of different DHT topologies. Legend: n - number of nodes, d - number of dimensions, b - number of key bits

networks over unstructured ones is the ability to perform simple and efficient lookups in the network, since the structural constraints allow for deterministic placement of nodes. The usage of keys to identify both nodes and content allow to efficiently determine which node contains a given resource (or service). From the previous table, we can see that in most cases, querying takes only $O(\log n)$ messages for structured networks.

To perform a lookup on an unstructured network, the typical approach consists of a constrained flood, where the query is spread to all known nodes with a given Time-To-Live to prevent saturating link bandwidth, or through the use of random walks, where neighbor nodes are picked at random as the next node to query. These approaches do not guarantee a resource will be found, even if it exists (it may be beyond the TTL boundary, for instance). The use of flooding techniques greatly increase the number of messages on the network, as querying requires $n \times avg.degree$ messages, where the average degree is the average number of outbound links per node.

This also limits the efficiency of range queries, as the lack of a structure denies the inherent efficiency of determining nodes belonging to a given range through their relation with other nodes in the network.

Structured networks, however, are much less stable than unstructured networks under churn (the effect caused by nodes constantly entering or exiting the system), since the structural requirements of the network add to the complexity of the network and additional steps must be taken to ensure this structure

is maintained to preserve the desirable properties of structured networks.

The following table summarizes these points:

	Scalability	No. of messages	Robustness	Range queries
Unstructured	$O(\log n)$	$n \times avg.degree$	High	No
Structured	$O(\log n)$	$O(\log n)$	Low	Yes

Table 2.2: Comparison of structured and unstructured networks

2.4.2 Resource discovery in P2P networks

As said above, unstructured P2P networks such as Gnutella generally use flooding to perform lookups.

This introduces an enormous overhead on the network, and is unscalable since the bandwidth cost of a query increases exponentially with the number of searched nodes, as each node propagates the query to all its neighbors[45]. Additionally, the fact that many clients are connected only for very short amounts of time causes many searches to deliver unsatisfactory results since only a small amount of the total users are effectively searched due to query drops.

To solve these issues, the Gnutella protocol has been heavily improved through the addition of search algorithms based on random walks, super-peer clustering (known as ultra-peers in Gnutella terminology) and other improvements. This gave birth to the Gnutella2 protocol, which is far more scalable, at the cost of additional vulnerability to DoS attacks and network maintenance protocols (due to the existence of super-peers).

Other protocols, such as BitTorrent, rely on the existence of a central resource information repository, known as a tracker, to discover which nodes possess different files or chunks of files. BitTorrent is, in its essence, an unstructured network, but the use of a tracker overcomes the usual problem associated with this type of networks, at the cost of fragmenting the nodes using this protocol into several smaller networks where not every file is available.

In structured networks resource discovery usually amount to performing a hash over the required resource description and performing a lookup on the network. This, however, has a few shortcomings.

Lookups in DHTs are often exact queries as the use of hash functions makes a slight change in a resource description (for instance, a file or its metadata) yield a completely different key and a the same time makes it difficult to perform "range queries" which provide multiple results that fit a specified range.

This kind of lookup also makes it difficult to handle highly dynamic resources (for instance, in a grid, the available RAM or CPU power in a given machine), as the frequent changes in values would cause nodes to frequently "move" in the overlay. A few proposals have been made to overcome this problem, such using specialized encoding functions instead of hash functions to encode static and dynamic attributes of a node into a key[14].

To solve these issues and allow for range queries a number of systems have been developed as well.

Marzolla et al propose, in [41], a system that organizes nodes in a tree-structured overlay network based on Routing Indexes[19], where nodes maintain information about the resources they manage as well as a

description of the resources present in the sub-trees whose roots are that node's neighbors. The range (or *domain*) of each attribute is split into k sub-intervals, where k may differ between different attribute types.

For each resource, the index of a given attribute of that resource is then represented as a k-bit vector with all its entries set to 0, with the exception of the one corresponding to the sub-interval where the actual value of that attribute is contained. The index that represents that attribute for all the local resources of a given peer is then obtained by preforming a logical OR operation on all the attribute bit vectors of the local items.

Multi-attribute range queries are split into multiple sub-queries, one per attribute, which are then mapped into binary vectors of length k (with k naturally being the value associated with each attribute type) whose entries are set to 1 if they correspond to sub-intervals contained in the range specified in each sub-query.

Sub-queries are initially matched against local indexes to determine if local resources satisfy the query, and then against routing indexes to determine which neighbors contain indexes that satisfy all sub-queries, to where they will be routed. Updates are dealt with by only sending messages if the new bitmap representation of the resources differs from the old one. This avoids unnecessary communication and adds to the scalability of the system.

The authors later expand on this work, where they propose using a two- level overlay network[42, 43] by splitting the nodes into a set of smaller trees to solve maintenance issues in the above architecture (since the tree is required to be balanced and that is hard to obtain on a peer-to-peer environment due to churn) and also to further increase the system's scalability by preventing overloading in a single root node.

In another work, Crainiceanu et al.[18] propose the use of P-trees, which form a distributed B+ tree, where each node maintains a semi-independent sub-tree and views the key space as a ring where its own root value is the smallest. The semi-independence stems from the fact that each node only maintains its own root to leaf path, while maintaining pointers to the nodes necessary to complete the tree.
Sub-trees can have overlapping ranges to allow for nodes to independently grow or shrink their own trees, eliminating any coordination steps that would otherwise be necessary. Thus, local inconsistency is allowed but periodically corrected to guarantee global consistency.
Range queries are performed as queries in a B+ tree, using the stored information and contacting other nodes as necessary.

Andrzejak et Xu[9], on the other hand, propose extending an existing CAN-based DHT to deal with range queries through the use of specific servers in the network known as *interval keepers*.
These servers are responsible for a sub-interval of a given attribute's possible value range, and other nodes report their attribute value to the responsible keeper. Servers are mapped to regions in the k-dimensional space through Space Filling Curves, and maintain locality properties such that close values of a given attribute are also "close" in the Cartesian space.
Multiple attributes in a resource's description are mapped individually to the corresponding DHT.
Range queries in the system are routed to the interval keeper that owns the middle point in the range and that node will then recursively propagate (through flooding) the request to all its neighbors until all the keepers that maintain points inside the specified range are visited.

XenoSearch[56] is the resource discovery system employed in the XenoServer Open Platform[28]. It

uses the Pastry DHT to perform indexing and querying. Every attribute on a given resource is mapped into a different Pastry ring to allow for multiple attribute queries. Queries are simply decomposed into sub-queries, one per attribute, and then a range query is made for each of these sub-queries.

Range queries are enabled due to the fact the information is stored in a tree-like structure where leaves correspond to independent peers in the network and the tree's internal nodes (or Aggregation Points) are parents of all nodes that fit a given range for a requirement. These points are mapped into Pastry keys and the closest peer to a given Aggregation Point key according to the ring's distance metric is responsible for maintaining all the information corresponding to that Aggregation Point.

SWORD[48, 5] provides a mechanism for the location of nodes based on both static and dynamic characteristics, as well as inter-node characteristics. It provides multi-attribute ranged queries and several other notable features such as allowing to filter out nodes when there are more nodes available than the user actually requires. There are two types of nodes in this system: *reporting nodes*, which periodically measure and report available resources, and *server nodes*, which receive the reports and handle user queries.

SWORD uses the Bamboo[51] DHT, similar to Pastry, to organize server nodes. For each of the existing attributes, reporting nodes map the attribute to a key and send a tuple containing all attributes to the server node in the DHT corresponding to that key, where that tuple is stored for each reporting node. Range queries are supported by mapping the range of possible values to a contiguous region of the DHT space using a given function.

In Mercury[13], the authors propose a system that allows for scalable, multi-attribute queries in which nodes are split into *attribute hubs* and queries are made to the hubs responsible for the necessary attributes. This system makes a "small world" assumption that guarantees inter-hub routing is efficient, as it is assumed any given node is reachable in a small number of hops.

Routing and indexing in Mercury is provided by the Symphony[39] DHT, a one-dimensional DHT designed with the aforementioned small-world assumptions in mind. Symphony allows for range queries by locating the node corresponding to the smallest value in the range and then proceeding through the next values until the largest value is reached. The query then returns a list of all resources in the range whose other attributes also match the corresponding ranges specified in the query.

INS/Twine[11] presents a scalable resource discovery architecture for peer-to-peer networks through the use of DHTs. It splits a resource's description into a subsequence of attributes and values, called a *strand*. These strands are then hashed, and these hashes form the numeric keys to use in the DHT.

From the resource description, the system also builds an *AVTree* (attribute-value tree), which describes the resource. This allows for the existence of partial matching of queries, i.e. queries whose corresponding AVTree only partially matches the published resources' AVTrees. Queries must then have an hierarchical structure similar to the resource descriptions that allow them to be transformed into the same data structures for comparison.

In INS/Twine, queries are resolved by a set of resolvers organized in an overlay network which route resource descriptions among themselves for storage and collaborate to resolve client queries.
Clients submit queries to a resolver that transforms that query's AVTree into strands and computes the

key for each strand. Then the message is routed to the correct resolver for each strand through the DHT using the previously calculated keys. Resolvers receiving these messages respond to the original resolver which in turn sends the query results to the client application.

Finally, Camargo et al. present OppStore[21, 20], based on the Pastry DHT, which like the work presented by Cheema et al. attempts to address the problem of dynamic attributes in a node. This is done by calculating the node's *capacity*, based on characteristics such as the node's general availability, free disk space, processing power and others. This capacity value is then used to assign to each node a *virtual ID* using the same range of valid Pastry IDs as used in the regular Pastry keys, through the *virtual space partition protocol*.

This protocol is executed whenever a node joins or leaves the network, or when a node's capacity changes. After joining the network through the usual Pastry joining protocol, new nodes obtain the virtual ID and capacity of each node in their Pastry leafset and partitions the virtual ID range covered by these nodes, assigning to each node a virtual ID range proportional to their capacity so that queries will be assigned to nodes with more capacity automatically. Nodes are only required to store an additional identifier, as well as *virtual leafset* and as such incur only a small penalty on the complexity of the entire system.

Routing is also mostly unchanged, despite the addition of virtual IDs. The only change occurs in the last hop of routing a message, where the virtual leafset is used instead of the regular Pastry leafset in order to determine the node to which the message will be delivered. In fact, the Pastry leafset is still used, but it only provides the real ID that maps to the virtual ID in the virtual leafset. As such, routing using OppStore virtual IDs takes the same number of hops as through the regular Pastry overlay.

By using this protocol, and because virtual IDs increase the system's flexibility while adding little overhead to the base Pastry protocol, the system can adapt easily to highly dynamic environments with changing node capacities and churn.

Note that other systems exist, such as SkipNet[29], which support range queries but do not support multiple attribute queries. This is caused by the DHT structure which sacrifices this ability for the ability to perform range queries in logarithmic time and guarantee path locality (by arranging nodes in a ring where nodes are sorted by lexicographical order), a characteristic that most DHTs disregard.

The main characteristics for the systems presented above are summarized in the following table:

	Architecture	Multi-attribute queries	Range queries
Marzolla et al.	Tree-based	Yes	Yes
Crainiceanu et al.	Distributed B+ tree	No	Yes
Andrzejak et al.	CAN	Yes	Yes
XenoSearch	Pastry	Yes	Yes
SWORD	Bamboo	Yes	Yes
Mercury	Symphony	Yes	Yes
INS/Twine	Any	Yes	Yes
SkipNet	Skip List	Yes	No
OppStore	Pastry	Yes	Yes

Table 2.3: Summary of resource discovery systems

Service discovery in P2P networks

Service discovery in P2P networks is a fairly unexplored subject, as most of the use given to this type of networks revolves around file-sharing and data replication, which amounts to resource discovery rather than service discovery.

Banerjee et al.[12] propose building a distributed UDDI registry over a DHT, which acts as a rendezvous network between several separate UDDI registries. The use of a DHT allows to overcome the lack of scalability of the UDDI registry system, thus avoiding relying on public UDDI systems.

In this system, the various UDDI registries are kept as separate entities, which coordinate through the DHT to determine which registry holds a given record. When a new record is to be inserted, the client contacts a Proxy UDDI server, which in turn uses the DHT to map it to a key that will be used to determine the UDDI registry where that record will be placed. When the registry is located, the service is published there.
If the service has multiple words, each word is mapped to a key separately, which can result in multiple UDDI registries being used to store the service record. Querying the system to locate a service follows the same steps to locate the necessary registries, and queries are sent to all UDDI registries that match the queried word.

2.4.3 Representing resources and services

Representing resources and services in a way that allows for easy indexing and querying is a problem tackled by many of the systems mentioned in the previous sections.

Most solutions, such as the ones proposed by Balazinska[11], Bharambe[13], Oppenheimer[48] and Albrecht[5], rely on the use of XML due to its simplicity and structure, which allows to establish hierarchical relations between attributes, as well as specify characteristics particular to a given attribute type.

This XML representation is later mapped into an internal representation for compression and efficient replication. Queries can be introduced in XML form (or translated to this form from an easier to write form, such as flat text) to provide an equivalent representation for both the desired resources and those that are known in the system.

2.5 Analysis

Recently, the two major development areas in distributed computing systems have been Desktop Grid and Peer-to-Peer systems, due to the increasing pervasiveness of computers in our everyday lives. Increasing amounts of available bandwidth, coupled with a very strong need for content consumption have turned Peer-to-Peer systems almost ubiquitous and part of mainstream computing lingo. On the other hand, there has been growing awareness about Desktop Grid systems and their use to advance scientific research in areas people feel empathic about due to various factors. One example is the Folding@Home project[36], which has had enormous success, counting with over 350,000 CPUs contributing for the project as of 2010 for the purpose of advancing medical and pharmaceutical sciences.

Both technologies have similar goals, as they attempt to exploit existing resources and infrastructures in a distributed manner although for very different purposes and with differing priorities.

Nevertheless, traditional Grid infrastructures continue to have great importance in academia and scientific projects due to their reliability and are one of the enabling technologies for very large scale research projects such as the Large Hadron Collider.

Let us offer a comparison of the most important points for these technologies, with focus on Peer-to-Peer networks due to their particular relevance for this work.

2.5.1 P2P vs Institutional Grids

Traditionally, institutional grid systems have been developed with the intention of providing computational power for very specific and sophisticated applications, on a tightly controlled environment. As the applications' complexity increases, so does the required computational power and the difficulty to install and maintain such systems in a scalable manner.

Peer-to-Peer networks, on the other hand, have focused mostly on providing services with little sophistication, such as file-sharing but are now starting to appear in more complex scenarios, such as video streaming, online gaming and cycle-sharing applications due to their scalability and robustness. As research in P2P systems advances, better, more sophisticated ways to manage the various peers participating in a P2P environment emerge.

Although both systems approach each other in some areas, they diverge in a few major points, as follows.

Scalability Scalability in distributed systems can be measured in two ways: the total amount of participants the system supports, or the overall throughput of the system, where "throughput" is specific to the type of distributed application at hand. For instance, in a file-sharing distributed system, throughput can be defined as the amount of data that can be shared in a period of time, while in a cycle-sharing application it can be seen as the amount of work that was performed in a given period.

Institutional Grids traditionally have a modest number of participants, as each institution only has a relatively small number of nodes available for computation, in the thousands. Despite this, a grid's throughput is generally very high, with very specialized nodes running very sophisticated, optimized algorithms. One limiting factor of traditional grid systems' scalability is their reliance on centralized data repositories to feed the various nodes available. This type of approach does not scale well with a growing number of nodes. This is not obvious from the designs presented in the Globus toolkit, for instance, but actual implementations usually suffer from this type of flaw.

Peer-to-Peer systems, in contrast, usually scale very well in terms of the number of participating nodes, up to the hundreds of thousands of nodes at any given time and although the overall throughput is not as high as in a grid environment due to a number of factors including the general heterogeneity of the nodes involved, it is generally significant. For instance, it has been demonstrated that BitTorrent can saturate a peer's bandwidth during file transfer with the right conditions[64] and that this can be used to perform Distributed Denial of Service attacks by simply overwhelming the target's TCP stack[22].

Resources Grid networks are typically homogeneous and as such benefit from having standardized, well defined characteristics both in terms of hardware and software, facilitating resource and service discovery. Additionally, since these systems are typically deployed in the context of large scientific projects they are comprised of more powerful, better integrated components. They possess better centralized ad-

ministration and are easier to manage, with fine-grained control as well as having predictable availability, faults not withstanding.

Peer-to-peer systems possess little to no centralized management mechanisms due to their very nature, which makes them extremely resilient against denial of service attacks, but at the same time this precludes almost any type of control over the system. This lack of control also brings along a serious downside, because nodes can leave and join the network as they please, so the overall availability of system is not predictable in every situation.

Resources in a peer-to-peer environment are very heterogeneous, with machines running different combinations of hardware and software, with varying degrees of availability even among machines sharing the same characteristics. Since many of these systems run on machines that are also manipulated for a local user, availability can vary significantly outside of the context of the distributed computing applications.

These problems thus require more sophisticated resource and service discovery mechanisms than those found on grid systems.

2.5.2 P2P vs Desktop Grids

A desktop grid environment shares many characteristics with both traditional Grid systems and Peer-to-Peer systems and form a somewhat intermediate step between those two systems.

We shall compare a number of key points where these systems approach and differ.

Scalability Like a Grid, desktop grids are usually centrally managed, with a single point responsible for the global scheduling of the required work units and as such incur in some of the scalability penalties associated with traditional grids. However, since nodes are voluntarily donated without centralized intervention and work units tend to be similar regardless of the amount of computing power offered by each node (as seen in BOINC, for instance), managing a desktop grid in order to scale with growing numbers of nodes can be a much easier task, as it only involves making sure the central services also scale accordingly.

In practice, scaling those central resources is not an easy or inexpensive task and desktop grids can suffer from bottlenecks in the client-server communication. Peer-to-peer systems can perform much better in this respect by decentralizing the global scheduling services, distributing that task among a number of nodes who are themselves clients. A single point of failure cannot exist and even if a number of nodes fails, other nodes can take their place and maintain the entire system running at a comparatively low performance penalty.

Availability Unlike a grid and much like a peer-to-peer network, desktop grids are volatile, as their participating nodes can join and leave the network at will, often without completing their respective tasks in the system. In both cases, it is often required to use incentive mechanisms to keep nodes connected and participating. We've already seen that BOINC implements a "credit" system to provide incentives for participants (as seen in Section 2.3.4), and a number of peer-to-peer systems employ other mechanisms to ensure user participation. An example is the "tit-for-tat" mechanism employed by BitTorrent[16] which allows peers to choose to upload only to peers from which they downloaded as well, in order to avoid freeloaders. Desktop grids are also deployed in some contexts (academia, for instance) where such incentive mechanisms are not strictly required.

Both desktop grids and peer-to-peer systems have very heterogeneous environments and possess very different nodes present at any given time in the system. This contrasts with traditional grids which typically have very well defined and controlled combinations of hardware and software, dedicated to the task at hand. This brings issues in terms of availability management, since not all nodes are as capable as the others and mechanisms for finding the "right" participant for a given job must exist.

Resource and service location in desktop grids can be much simpler however, as the information regarding participant nodes is centrally located. In order to provide similar performance regarding resource location, peer-to-peer networks must often adopt more structured overlays, usually in the form of structured topologies and the use of more specialized peers specifically for the purpose of resource location.

Security Desktop grids and peer-to-peer networks present similar security challenges. In both systems, since they cannot be controlled in any significant manner, a number of attacks are possible to perform in ways which attempt to exploit the existing incentive systems or to simply deliver erroneous final results.

In both systems, result verification mechanisms must exist such that, with a given certainty, a "correct" result can be found even in the presence of a number of incorrect results.

2.6 Conclusion

In this chapter, the state of the art regarding Institutional Grids, Desktop Grids and Peer-to-Peer networks was presented, providing the theoretical basis for the currently developed work. A number of systems were presented and analyzed for the purpose of determining the strengths and weaknesses of each so the best combination of characteristics could be taken from previous experience for the construction of this work.

From this analysis we can conclude that there are significant benefits to be drawn from bringing the traditional Grid infrastructures and the new Peer-to-Peer systems together as they share some of the same concerns, such as efficiently locating requested resources; use some of the same techniques, like building an overlay topology on top of the existing physical network and have different, complementary issues and solutions for them, like the scalability issue previously described, which has serious implications on traditional grid networks but is almost inherently solved in peer-to-peer networks.

Desktop Grid solutions provide some of the benefits of Peer-to-Peer networks and Institutional Grid systems, but also some of the biggest drawbacks. For instance, while the remote scheduling services seen in BOINC work in heterogeneous environments, they fail to optimize task distribution according to the actual performance of each peer and are a central point of failure since those services alone can distribute tasks among nodes. They also fail to exploit routing mechanisms between nodes, thus potentially overloading the remote scheduling services, aggravating the previous problem.

The greatest drawback of both Desktop Grids and Institutional Grid systems however, is the fact that they both effectively limit the generation of new distributable tasks to the small fraction of the participant universe that could possibly already afford large computational resources, while giving little to no benefits to the common cycle donor. Peer-to-Peer systems have the unique property of enabling any participant to not only contribute to perform computationally heavy tasks in lieu of others, but also to take advantage of the immense distributed computational power available for themselves, and as such are the best approach

to fulfill the ultimate goal proposed in GINGER, i.e. to deliver high performance computing to the masses.

For these reasons, GINGER should use a P2P overlay as its network substrate, as it can provide a fully decentralized, failure tolerant network. The use of *super-peers* can grant the network the necessary resource discovery mechanisms by providing a distributed index at a relatively low cost both in terms of added complexity and overhead imposed globally on the network. In particular, by using a P2P overlay we expect to be able to reach most of our goals:

- **Scalability:** Peer-to-Peer networks offer the greatest scalability potential due to their self-organizing nature;
- **Efficiency:** Peer-to-Peer networks (in particular, structured Peer-to-Peer networks) are efficient in the amount of messages and the average number of hops needed to perform their tasks. In particular, Pastry, the overlay protocol selected for this project routes messages on average in $O(\log n)$ steps;
- **Flexibility:** The use of *super-peers* allows us to go beyond what the P2P overlay routing mechanisms offer in terms of resource discovery, namely enabling us to map more complex information about each node and route accordingly.

Pastry was chosen as the P2P overlay for this work because it has shown to be efficient, presenting a relatively simple topology and also for the fact that routing can be optimized for different criteria, other than node identifier proximity, if the need arises. This last point in particular is much harder to achieve in other ring-like topologies like Chord, which although simple, has a larger dependency on identifier similarity for routing.

Chapter 3

Architecture

In this chapter, we present the architecture used in this work and the rationale for the options taken in its structuring. Both the general architecture for the GINGER[60] project and the specific architecture related to the overlay network used will be presented in the following sections.

One of the goals of GINGER as a platform is to obtain the benefits of Grid-like computing infrastructures with the flexibility of P2P networks in order to enable its use by the masses. For this, it is important to determine a good P2P overlay topology, as the resource discovery mechanisms depend on this topology, as opposed to the underlying physical networks.

The main goal should be a system that obtains the best nodes available for a given work unit based on a number of criteria such as available bandwidth, processing power and so on.

This chapter is structured into four different sections:

- **System architecture**, where the overall system design will be discussed;
- **Overlay network topology**, where the rationale and conclusion for the chosen topology is presented;
- **Resource Discovery**, in which the resource discovery protocol is described;
- **Data Structures**, where both pre-existing and new data structures are presented, as well as the rationale for their existence.

3.1 System Architecture

GINGER is a middleware platform running on top of a semi-structured P2P overlay network whose basic work unit is a *Gridlet*. A Gridlet contains information regarding the data workload as well as the transformations required to be applied on it. Each Gridlet also contains the *cost* associated with performing those transformations to the data and optionally the actual code that performs them.

The Gridlet application model is divided into three stages: Gridlet creation, Gridlet processing and Gridlet-result aggregation. In gPastry, no Gridlet processing is done at this level, as it is irrelevant for the problem at hand. Also, Gridlets are simplified and only contain information regarding the required resources (application and other requirements) for the Gridlet to be processed.

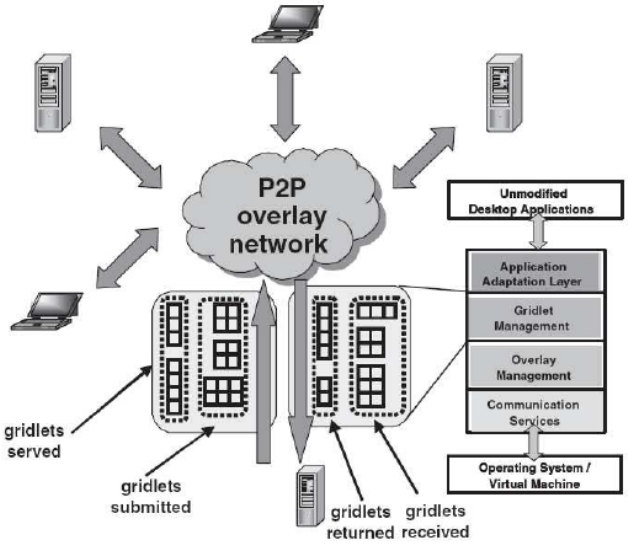

Figure 3.1: Ginger architecture

Whenever a node needs to submit work to the network, the input data is chunked into one or more Gridlets that will then be routed throughout the P2P overlay for processing. When the necessary transformations are computed, the results are packaged in special result Gridlets, which can either be sent directly to the origin or become stored throughout the overlay for later retrieval using a distributed storage system such as PAST[53] or, alternatively, a distributed caching system as the one described in Squirrel[32].

3.1.1 GINGER layers

As shown in the above figure, GINGER is composed of four layers (summarily described here from top to bottom):

- *Application Adaptation* layer - handles interaction between GINGER and the actual applications;
- *Gridlet Management* layer - handles all necessary Gridlet operations;
- *Overlay Management* layer - handles overlay maintenance, node discovery, message routing and resource management;
- *Communication Services* layer - handles communication and data transfers through the network;

This work focuses on the Overlay Management layer, as all other layers play little to no role in regards to resource and service discovery. Let us define these layers more thoroughly:

Application Adaptation layer

The Application Adaptation layer is responsible for the interaction with the unmodified desktop applications, in order to launch them and feed the data received from gridlets, as well as collecting the resulting data from the transformations performed on the original data by the application.

Gridlet Management layer

On the Gridlet Management layer, files and data are partitioned into the corresponding Gridlets and received Gridlet results are reassembled into result files as those that the application would produce if ran locally on a single machine.

Overlay Management layer

The Overlay Management layer is, as its name implies, responsible for maintaining the overlay network in order to exchange gridlets with other nodes. This layer only maintains the logical structure that forms a node's information about that overlay however, and all actual network data transfer and communication is handled by the Communication Services layer.

This layer holds all the information regarding the overlay and transparently performs resource discovery in order to locate the resources required by the upper layers. Both super-peers and regular peers maintain a common subset of information, including known super- peers and their own capacity, with super-peers maintaining extra information regarding the availability of all applications its child peers have registered with it, as well as the aggregate capacities announced by other super-peers regarding the applications which fall under the application family that super-peer is responsible for.

It is through this information that each node is capable of performing resource discovery by sending and receiving Gridlets. The information maintained by each super-peer is illustrated in Figure 3.2. Super-peers must maintain the aggregate availability for individual applications under their responsible

Child nodes

Application	Node ID	Capacity
4309ad398f	1a4908543	75
	420945da	0
	104978fd	764
	1ad54ef984	43
a765de453	1a4908543	21
	420945da	393

Known super-peers

Super-peer ID	Application	Capacity
4897d398e	4309ad398f	516
4897d398e	a765de453	43
6397fd893a	4309ad398f	298

Figure 3.2: Information maintained by a super-peer.

family for other super-peers in order to only choose super-peers which actually possess the necessary applications. Otherwise, it would be possible for a super-peer to forward a Gridlet to another super-peer which possessed large capacity (in its group) for a number of applications that fell under the same family, but no capacity for the actual required application.

Communication Services layer

As previously mentioned, the Communication Services layer's function is simply one of performing data transfer throughout the network. It performs no routing or other type of computation in order to determine an incoming or outgoing message's destination and leaves that function to the Overlay Management layer.

The major focus of this work is on the *Overlay Management* layer, as it is the layer responsible for maintaining the overlay and performing discovery, as explained above.

3.2 Overlay network topology

The overlay network used in gPastry is based upon the Pastry[52] overlay, described in Section 2.4.1, as it presents a robust P2P overlay with efficient and flexible routing, while using a number of modifications to accommodate our resource discovery protocol, which will be detailed across the next sections.

The immediate concerns when designing the architecture for this work were to both minimize the number of messages that need to be exchanged between nodes in the network, to avoid flooding, as well as attempt to provide enough coverage such that the required resources will be found when they are needed, if they are present in the network.

In order to accomplish this, nodes in the network are divided into two sets: regular nodes and super-peer nodes. This closely follows Gnutella's approach using "ultrapeers" for data file indexing, as well as CCOF's *rendezvous points* as mentioned in Section 2.3.2. Super-peer nodes perform every function a regular node does and, additionally, they maintain and share high-level information regarding other nodes in the network. As such, super-peers are not distinct from regular nodes but perform additional tasks with regards to overlay management when compared with the remaining nodes in the overlay.

The overall layout of the nodes is displayed in Figure 3.3. For reference, note that the overlay network does not need to connect nodes in any particular order for super-peers to exist. Repeated links between nodes are omitted in this figure. Let us explore super-peers in more detail in the following section.

3.2.1 Super-peers

Super-peers form a ring amongst themselves to allow faster communication and avoid hopping messages through the overlay network. They share information about the availability of applications among their child nodes and act as resource brokers, sharing their child nodes' resources with each other when such is needed (for instance, when a super-peer's children cannot perform the work required for a given gridlet, it requests work from another super-peer which whose children have enough availability to perform the task).

Regular nodes, in turn, are "clustered" around these super-peers and use them to perform service discovery and gridlet requests to other nodes in the network. Every node is assigned to a single super-peer, determined from all known super-peers through the Pastry proximity metric it is then referred to as that node's *primary super- peer*. The use of the proximity metric was chosen because it explicitly represents the degree of proximity between two given nodes in the overlay, as defined by the basic Pastry protocol.

Despite being assigned to a single super-peer, every node possesses a list of existing super-peers in order to choose a new super-peer in case of failure of the existing one.

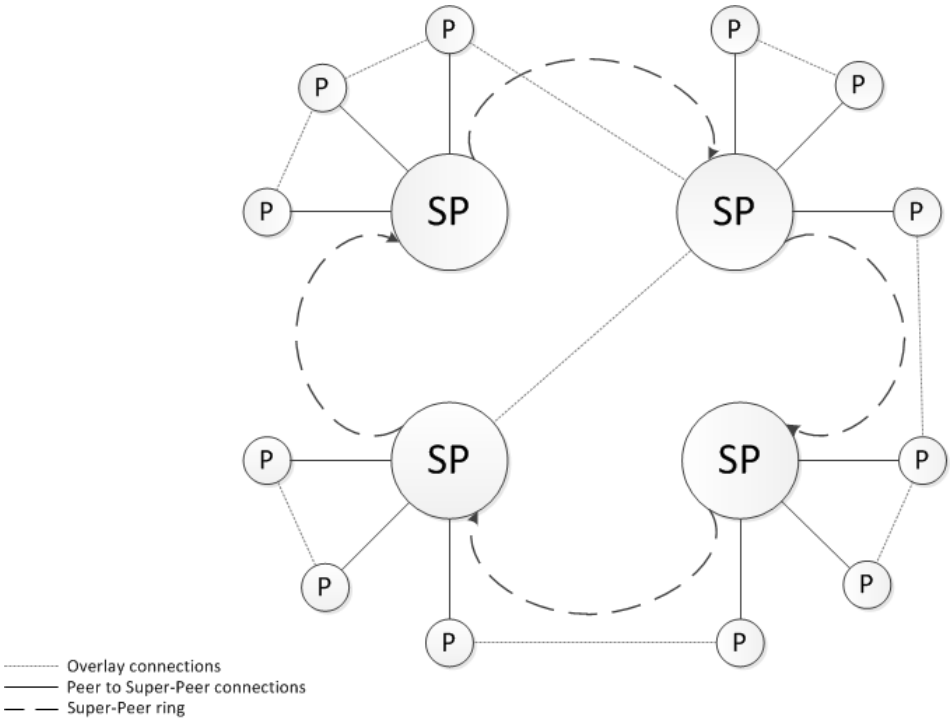

Figure 3.3: Node organization in the network. "SP" indicates a super- peer, while "P" indicates a regular peer.

Super-peers election is performed in a way that attempts to balance the ratio of super-peers to regular nodes and make sure no nodes are *orphaned*, i.e. left without knowing any super-peers. Nodes will periodically check the number of super-peers they know; if this number is below a certain threshold, those nodes can select themselves to become super-peers with a given probability. This probability has to be kept low enough so that the number of super-peers cannot grow too large at any given point in time, but such that allows a fairly rapid expansion of the number of super-peers in case of shortage.

3.3 Resource Discovery

Applications are the main resource taken into consideration in gPastry and the resource discovery mechanism focuses mostly on finding nodes which have the given application installed.

In order to structure information regarding applications throughout the overlay, using the topology described in the previous section, two pieces of information are kept about each application: the *family hash* and the *application hash*.

The *family hash* is used to describe the family or class of applications any given application belongs to. In the context of gPastry, the family hash is a hash of the *canonical name* of an application, which in this case is, for instance, the URL of the application's main web page online (e.g. for the Python application, the canonical name would be "`www.python.org`"). This *family hash* is used to aggregate information regarding all nodes with an application belonging to that family in the same super- peer, in a way that will be described further in this section.

The *application hash*, on the other hand, describes a specific combination of application name and version

and is used to determine, in a given super-peer containing information about a given application family, which nodes possess a specific application installed in the system. For instance, considering the Python application family example, there could be a number of distinct application versions described uniquely by this hash such as `Python 2.6` and `Python 3.0`.

This distinction using the application's version is important for dealing with cases where different versions of a given application are incompatible with each other and thus cannot be used to perform work on the same set of data, or the user requires functionality present only in a specific (or more recent) version. Note that although only the version number is used in the given examples, different "versions" can be easily created for applications running in different operating systems as well.

As described previously, there are a number of super-peer nodes whose purpose is to aggregate information regarding the installed applications. These super-peers are "responsible" for one or more application families, which means they aggregate the high-level information on all current super-peers regarding those application families.

In particular, each of these super-peers maintains a table containing other super-peers' aggregate availability for a given application inside the family they are responsible for as well as the availability of every child node (including themselves) for each application those nodes register when joining the network, even if that application falls outside the family it owns. However, regarding the capacity available under other super- peers for those families, each super-peer only maintains the aggregate available capacity, with no information about what specific nodes under that super-peer possess the application or their individual capacity. Furthermore, super-peers do not store such information about applications whose family is not of their responsibility when this information does not belong to one of its direct child nodes.

Responsibility is determined according to the application's *family hash*, which is mapped into the super-peers' set of keys. If a super-peer's node ID is larger (according to the Pastry proximity metric) than the key given by the family hash, which belongs to the same key space, then the super-peer is responsible for that family. This is done in order to evenly distribute the application family's key space among all super-peers and avoid overloading any given one. Whenever a new super-peer joins the ring, every super-peer adds it to its super-peer lists and if the new super-peer should become the one responsible for an already existing application family, the super-peer containing that information sends it to the joining super-peer. Super-peers can be responsible for multiple families, as long as the family hash is lower than their own node ID but not lower than any other super-peers' node ID.

When submitting a Gridlet, nodes send it to their super-peer who, according to the information aggregated from his child nodes and other super-peers, distributes the workload of that given Gridlet in the following manner:

- The super-peer forwards the Gridlet to all its child nodes which have the required application and capacity;
- If more nodes are needed, the super-peer contacts the super-peer deemed responsible for that application family by forwarding the Gridlet. Otherwise the Gridlet is dispatched and the super-peer does nothing else;
- Upon receiving the request from another super-peer, the super- peer responsible for that application family checks the capacity table for another super-peer possessing enough child nodes to fulfill the request and forwards the Gridlet to it.

- If no other super-peers exist that can fulfill the request, the responsible super-peer simply replies immediately with a message.

A sample of the described discovery mechanism is illustrated in Figure 3.4 and Figure 3.5. Please note that in regards to information about the capacity of other super-peers, the representation presented in Figure 3.5 was simplified as to represent only the capacity for the application requested in the current Gridlet.

Node capacity is determined by a number of different factors, such as CPU power, total memory and bandwidth. When a node's capacity changes (due to work in progress or external factors, such as a local user interacting with the system), its new capacity is sent to its corresponding super-peer.

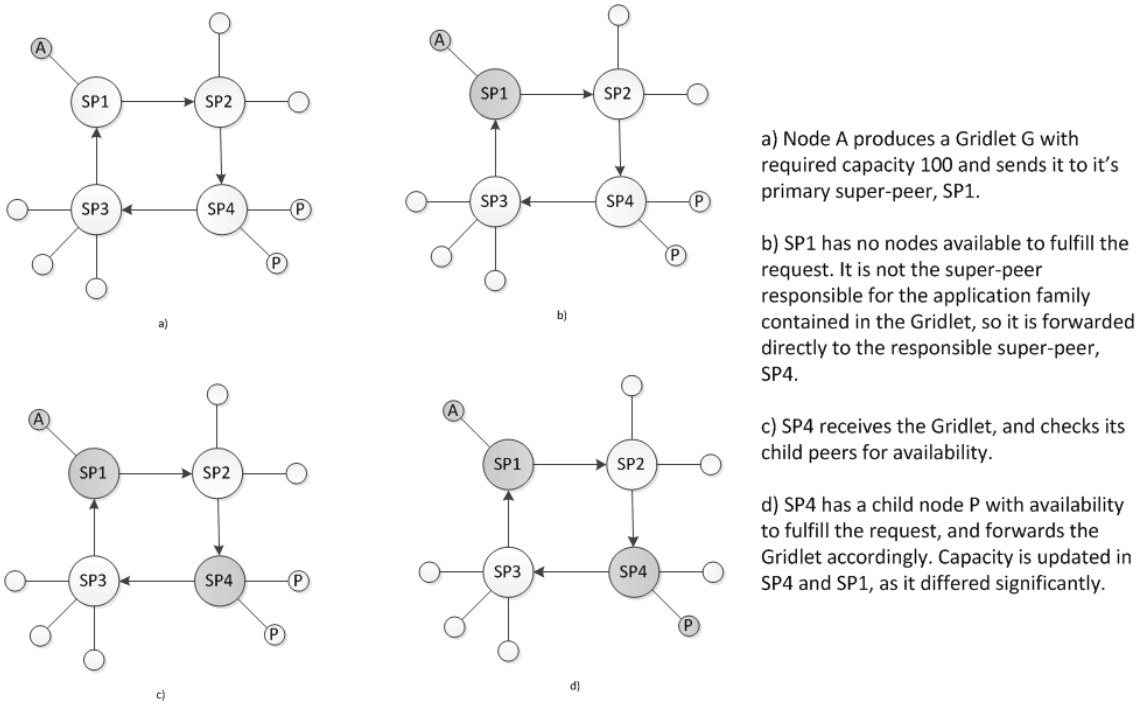

Figure 3.4: Resource discovery protocol

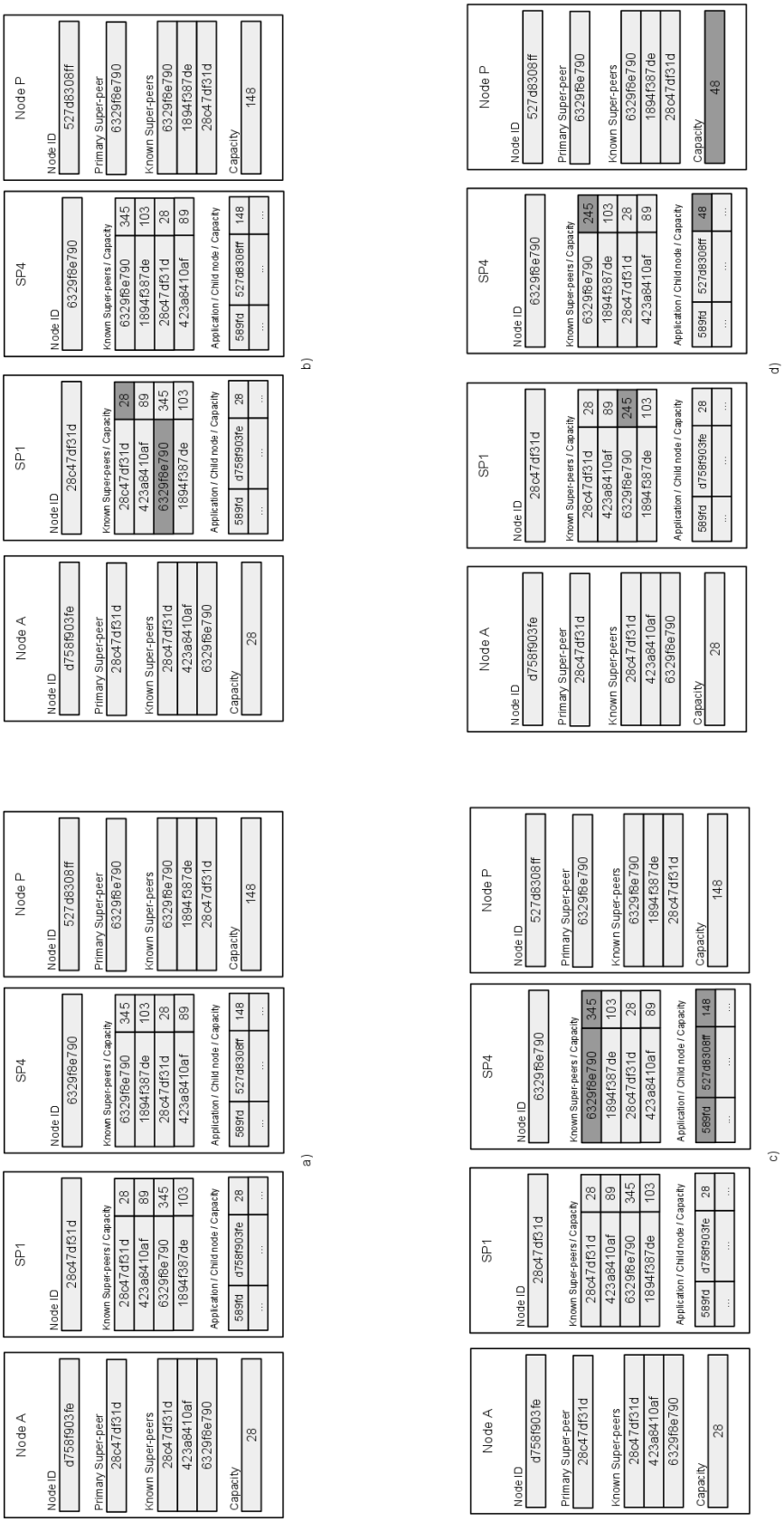

Figure 3.5: Table progression on each node during discovery

Upon receiving this information, the super-peer updates its internal aggregated capacity for the applications which this particular node provides and, if the new aggregate capacity is different enough (configurable for increments of, for instance, 10, 100, etc.) from the previous value it had transmitted to the super-peer responsible for that application, it sends an update message to that super-peer in order to update the value that super-peer is keeping.

The use of this type of asynchronous, on-demand communication instead of either having synchronous updates or sending updates every time a node updates its availability, is done in order to prevent flooding the network with messages regarding relatively insignificant changes to the overall availability of a given application.

Additionally, upon joining the network a node sends all the information regarding its installed applications to its newly determined super-peer, which then updates its aggregate capacity for each application registered by the new node and if the change in capacity for a given application is significant enough it then updates the super-peer responsible for that application, as in the previous situation.

3.4 Data Structures

gPastry builds on top of Pastry overlay and as such, a number of data structures used in Pastry are found unmodified in this work. These data structures include the basic node state information in the form of three separate structures: **routing table**, **leaf set** and **neighborhood set**. Additionally, a number of new data structures were developed to support super-peer information as well as available resource information.

We shall begin by recalling the already existing Pastry data structures, in order to support this protocol's routing mechanisms, which are used throughout this project, and then we introduce the new data structures developed specifically for this work.

3.4.1 Pastry specific data structures

Routing Table

The Pastry routing table is a $\lceil \log_{2^b} n \rceil \times 2^b - 1$ table, containing Pastry node identifiers organized by similarity with the current Pastry node such that entries at row n of the routing table refer to a node whose identifier shares at most the first n digits with the current node. In a given row r, the entry $r[i]$ contains an identifier whose n^{th} digit matches the index i. For every digit d of the current node's identifier, the table entry $r[d]$ contains the current node's identifier and this value should not be used for routing, since it would always route to the current node.

There may be many nodes sharing this common prefix so the identifier that is closest to the current node's identifier, according to the proximity metric is the one chosen to fill the position in the routing table. The routing table is primarily used to route messages to nodes that are distant from the current node from the overlay's point of view. Note that this distance is independent of both actual geographic distance and the distance in the underlying network topology.

The routing table organizes the known nodes in such a way that routing is performed in a tree-like manner, as a message navigates through the nodes approaching its destination by gradually approaching nodes whose identifier is closest to (i.e. shares a larger common prefix with) the destination node. This

kind of routing allows the message to traverse the overlay close to its destination in a relatively small number of hops, needing only $O(\log n)$ hops.

Entries in the routing table are added as new nodes enter the network and are replaced if a node is found such that its identifier is closer to the current node's identifier according to the proximity metric.

Leaf Set

The Pastry leaf set contains a small number of known nodes whose identifiers are numerically closest to the current node's identifier. Half of this set contains such identifiers which are smaller than the current nodes' and the other half contains identifiers which are larger than the current one.

This leaf set is then used to route messages whose destination is already considered close to the current node or in the case the tree- like routing strategy provided by the routing table fails to determine a node to which to route the message. In this situation, the message is routed in a ring-like manner, as the nodes selected to route the message tend to be successors or predecessors to the destination node due to their proximity to it. Taking this into account, the leaf set is usually the structure where lookups are performed first so the number of hops required for routing is as small as possible.

Neighborhood set

The neighborhood set is a simple, unordered list containing a small number of nodes whose identifiers are the closest to the current node's identifiers according to the proximity metric. This set is typically not used for routing, but in rare cases where neither the leaf set or routing table produce a suitable node for routing the neighborhood set can be used. In those cases, the nodes contained in the neighborhood set, leaf set and routing table are gathered, and a node is picked such that its identifier shares at least as many digits with the current node but that is also numerically closer to the destination.

The neighborhood set is more useful to maintain locality properties in Pastry however. It can be initialized by obtaining the neighborhood set from the node to which the current node bootstraps while joining, assuming that node is close to the current node according to the proximity metric.

3.4.2 New data structures

To complement and enhance Pastry in order to support the desirable new features for gPastry, a number of data structures were developed to hold and organize new information.

The most relevant data structures pertain to storing information about existing super-peers, as well as resources and their availability. These structures are described in the following sections.

Super-peer information

Nodes must maintain some information about existing super-peers in order to advertise their resources and become available for remote computation. To do so, nodes store a sorted circular list of known, live super-peers. This list is centered on the closest super-peer according to the Pastry proximity metric and is sorted by proximity to the current node. This particular super-peer is designed as the *primary* super-peer for that node and is used as the main point of communication between the node and the remaining overlay for purposes of resource discovery.

Whenever a new super-peer is discovered, or one joins or leaves the network, this list is updated accordingly. The closest super-peer is designated as the *primary* super-peer and is used as the primary choice for resource advertisement and lookup functionality. This allows to minimize the number of hops necessary to communicate with a super-peer to obtain information, thus preserving Pastry's performance characteristics. A super-peer's primary super-peer is always itself per the proximity metric. This introduces no inconsistencies, and does not preclude super-peers from submitting Gridlets themselves; they will follow the exact same protocol as other peers.

Resource availability information

Resource availability information is maintained only by super-peers and functions as a distributed index where super-peers can lookup resources both inside and outside their node group with different levels of detail. Super-peers maintain resource availability for each node under their respective groups, namely which applications are available among the many nodes, the actual nodes that possess them and their current availability. This allows the super-peers to pick the most available nodes for a given Gridlet.

In order to locate resources outside of their group, however, super- peers maintain only the aggregate availability of other super-peers' groups for the applications they advertise amongst themselves. This allows not only to locate more resources for a known application outside a super-peer's group in case of saturation, but also allows to discover nodes which can provide different resources which are do not exist in the current group. In order to minimize the amount of information each super-peer has to maintain, super-peers only exchange this information with the super-peers responsible for the relevant application families.

For instance, if a super-peer A gains or loses capacity related to an application X in a given family F, only the super-peer responsible for family F is informed of those changes and records the appropriate information. Other super- peers are not informed as the information is not relevant for the application families they are responsible for.

For this work, node capacity is represented by a value that corresponds to a weighted average of three metrics: available CPU, memory and bandwidth. This offers the benefit of having a simple, balanced way of comparing different nodes, but has the disadvantage of making it impossible to choose nodes according to any given characteristic.

The way this information is structured is presented in Figure 3.2.

Chapter 4

Implementation

For the implementation of gPastry, the chosen tools were Java 6 SE and the PeerSim[33] P2P network simulation platform version 1.0.3. Since PeerSim only provides a protocol-independent simulation platform, the entire code base that implements the base Pastry protocol was written from the ground up for the purposes of this project.

Using this platform as opposed to use others such as FreePastry[23], which already implements Pastry, allows greater flexibility if the need to change the underlying protocol would arise, at the cost of extra development effort. PeerSim is developed for the Java platform, and all development for it is done in Java as well.

4.1 The PeerSim platform

PeerSim has two basic types of simulator: an event-driven simulator, where progress is done by acting on events triggered by reception of messages sent from one node to the next, and a cycle-driven simulator, which allows to act on the simulation at intervals.

Both of these simulators can be integrated seamlessly to take advantage of the benefits inherent to each type of simulator. For the purpose of this work, this was the followed approach, using the event-driven simulator for most of the simulation as well as attempting to increase the realism of the simulation, and using the cycle-driven simulator to allow for easier execution of periodic functions on the nodes.

Here are a few definitions as used throughout the PeerSim documentation:

- Node - The basic building block in PeerSim, a low-level, protocol-agnostic representation of a node in a P2P overlay;
- Network - A set of interconnected nodes;
- Protocol - The protocol implemented by the nodes (in our case, gPastry).

In order to implement a given protocol, we must create a class implementing the interfaces related to the type of simulator we wish to use. There are two available interfaces: `EDProtocol`, for an event-driven protocol and `CDProtocol`, for a cycle-driven protocol.

In our case, we implement both interfaces, to perform actions on a regular basis, as well as when certain events are triggered. The protocol is implemented in the `ginger.overlay.pastry.PastryProtocol`

class.

PeerSim also introduces the concept of *controls*, which comprise the main form of direct interaction with the simulation. Controls can be used to gather information on the network, inject messages into the network and simulate churn conditions, among other uses. For this work, a few controls have been developed. They are present in the `ginger.overlay.pastry.control` package and will be detailed in Section 4.6.

PeerSim allows for the control of communications on the simulated overlay at the transport level through classes in the `peersim.transport` package. For the purpose of this work, the `UniformRandomTransport` was the one selected, which implements a transport layer which reliably delivers messages with a random delay, drawn from the interval defined in the configuration file using a uniform distribution. This simulates an environment where packet losses do not occur and/or can be corrected by retransmitting packets with delay (typical in real world applications where TCP is usually favored in detriment of UDP as the underlying transport protocol).

Additionally, we can select a *wiring* structure for the overlay. This defines a low-level topology which emulates the way nodes would be physically connected to each other in a real-world situation. PeerSim allows for the selection of several wiring methods (implemented through PeerSim controls that are executed upon initialization). The Internet is, at its core, a semi-structured network where most nodes are interconnected indirectly through other nodes, with a variable number of hops between them. As such, the control selected to wire the nodes in the network was the `peersim.dynamics.WireKOut` control, which randomly connects nodes among each other. The out degree k of the nodes can be passed as a parameter to this control through the PeerSim configuration file.

4.2 Project structure

The code is split into several Java packages. The top-level package is `ginger`, and it contains all the other packages that further organize the code:

- `ginger.overlay` and its sub packages contain all the code relevant to the implementation of the lower level protocol. It is in this package where the resource discovery protocol is implemented;
- `ginger.message` contains all the messages used throughout the protocol;
- `ginger.past` contains all classes implementing a basic distributed replication protocol;
- `ginger.layer` and its sub packages contain all code implementing the several GINGER layers.

Additionally, the `ginger.overlay.superpeer` sub package is of particular relevance as it is where all the code regarding super-peer specific features are implemented.

4.3 GINGER implementation

The GINGER platform was partially implemented as part of gPastry, with some of the layers, like the Application Adaptation layer, being implemented only as stubs, for they were not relevant for the purpose of this work. The implemented layers will be described in a bottom-up approach.

4.3.1 Communication Service

The Communication Service layer, as mentioned in Section 3.1.1, abstracts the actual communication with other nodes in the overlay. Most of the implementation details are further abstracted into the `PastryProtocol` class, for ease of implementation within the PeerSim platform.

This class provides the following methods: `route`, which routes the message through the overlay; `routeMyMsgDirect`, which sends a message directly to a given node, bypassing the routing mechanism; `deliver`, which processes an incoming message at a higher level than the one provided by the `PastryProtocol` class.

The routing methods wrap the corresponding methods present in the `PastryProtocol` class, which provide the required functionality. This is because of implementation details imposed by the PeerSim platform which are not relevant for the work at hand.

4.3.2 Overlay Management

The Overlay Management layer is implemented in the `OverlayManager` class and performs the essential functions for maintaining overlay information, as well as performing resource discovery through the protocols developed for gPastry.

This layer contains, per node, the relevant information required to route messages to other nodes, as well as information regarding existing super-peers. Message routing is performed using the Pastry routing mechanism and as such this layer must maintain the relevant data structures, namely the routing table, leaf set and neighborhood set inherited from Pastry and described in Section 3.4.1.

In order to locate resources according to the protocol we developed, the Overlay Management layer must maintain additional data structures, defined in Section 3.4.2.

4.3.3 Gridlet Management

The Gridlet Management layer manages all operations to be performed on Gridlets. Gridlet creation happens at this level, with the data received from the Application Adaptation layer, described previously. Upon Gridlet reception on the Overlay Management layer, it is forwarded through to the Gridlet Management layer, where it is processed. If the node currently has enough capacity to process the Gridlet, it is passed on to the Application Adaptation layer. If not, it can be forwarded to a different node.

This layer is realized in the `GridletManager` class. Gridlets arriving for processing are forwarded to the Application Adaptation layer which will later return the data required for the creation of a Gridlet Result containing the result of transforming the Gridlet data on through the required application, which is cached for later forwarding to the initiating node.

4.3.4 Application Adaptation

The Application Adaptation layer was implemented only as a stub, playing no role in this work, as its purpose is to seamlessly integrate with the applications needed for Gridlet data processing.

This layer passes the Gridlet's data to the application after performing whatever transformations are needed on that data in order for the application to work correctly and receives the resulting output data. This output data should then be passed on to the Gridlet Management layer, where a new result Gridlet

is created. For the purposes of this project, we can assume the Gridlet result is a simple transposition of the input data, such as an arithmetic operation or hash code.

4.4 gPastry implementation

The gPastry protocol is implemented in the `ginger.overlay.pastry.PastryProtocol` class. In order to run in both the event-driven simulator and the cycle-driven emulator, this class implements the `EDProtocol` and `CDProtocol` interfaces, respectively.

Pastry was implemented according to the original Pastry paper[52] and as mentioned previously is realized in the `PastryProtocol` class. Messages arriving on a node are processed through the `processEvent` method, which dispatches messages to the `route` method for the message to be actually routed (or delivered in this node, according to the Pastry routing algorithm) if the node has already joined the network.

If the message is to be delivered to the current node, it is then dispatched to the `deliver` method, which will forward it to the `CommunicationService` class, implementing GINGER's Communication Service layer and perform additional processing as needed.

A node joins the network through the `join` method. An overloaded method is provided for the first node to join the network in order to simplify the process. Whenever a node sends a message to another, the `route` message dispatches that message using the Pastry routing protocol. The underlying mechanism for message transmission is provided by PeerSim through the `send` method, which in this case is wrapped in another method with the same name to simplify development.

4.4.1 Messages

Messages used in this implementation of Pastry are located in the `ginger.overlay.pastry.message` package and form the basis of all messages exchanged in the network, from gPastry's point of view.
The base class is `RawMessage`, which defines the base methods and fields all messages should share. The remaining messages have functions as follows:

- `PastryJoinRequest` - This message is sent by nodes joining the network: its purpose is to announce the new node and request Pastry state information to fill the new node's state;
- `PastryJoinReply` - This message is sent to a joining node by every other node receiving a `PastryJoinRequest` message and contains the relevant information as described by Rowstron and Druschel[52];
- `PastryState` - This message is sent as part of the join procedure, containing the state of the current node as a response to the joining node so its tables can be initialized with current information;
- `PastrySuperAnnounce` - This message is sent by nodes who turn into super-peers in order to announce their new status;
- `PastrySuperQuery` - This message is sent by joining nodes and periodically by every other node in order to keep their super-peer lists up-to-date;
- `PastrySuperReply` - Reply message to the `PastrySuperQuery` messages sent by peers. It contains the super-peers known by the replying node.

Additionally, messages belonging specifically to the GINGER platform also extend `RawMessage` and are implemented in the `ginger.message` package. They are as follow:

- `Gridlet` - The basic Gridlet as described in the GINGER paper. It also implicitly performs resource discovery functions, as super-peers will act on Gridlets to search for nodes that can perform the required transformations on these according to the information included in the message's payload;
- `GridletResponse` - This message contains the Gridlet response generated by applying the required transformations on an origin Gridlet. It is returned directly to the peer which originated the message and contains in its payload the relevant output data. The origin peer's ID is contained within the original Gridlet message for this purpose;
- `GingerRegisterApplication` - This message is used by peers to register applications they possess to their parent super- peer. It allows peers to announce their availability initially and is only sent when these peers join the network, as we assume nodes cannot change the range of applications they can use when they are connected to the network;
- `GingerUpdateAvailability` - Messages used by peers to announce their new availability to their super-peers availability when sufficient availability changes occur. It is also used by super-peers to announce their new availability to other super-peers, as relevant.

Due to PeerSim constraints, messages are sent as Java objects. Nonetheless, these objects could easily be transposed into a generic, language-independent format for real-world use. The base `RawMessage` class contains the basic fields shared by every message:

- `sender` - The original node ID from whence the message came;
- `destination` - The node ID to which the message should be routed;
- `msgtype` - Identifier field used to distinguish between different messages;
- `payload` - Generic payload containing information relevant for the indicated message type.

In this implementation, new message types are introduced by simply adding the appropriate type and filling the payload field as appropriate. For instance, the message that sends the current state of a node for the purposes of updating other nodes as per the Pastry protocol, implemented in the `PastryState` class is no different from the `RawMessage` type except for the fact that it contains a specific `msgtype` and the appropriate information in the payload field.

Upon joining the network, a node A sends a `PastryJoinRequest` message whose destination node is also A to its bootstrap node B. B then routes the message through the network using the original Pastry protocol. Every node through which the message passes through will reply to A with a `PastryJoinReply` message containing the corresponding routing table node, as per the original Pastry initialization protocol so that A can build its initial state.

Gridlet messages add a number of extra fields to accommodate for the additional information they must carry. These fields contain the application hash, the application family hash as derived from the provided canonical name, its required availability and any input data that should be processed by the application at the destination node. As with other messages, the Gridlet message is implemented as a Java object for the purposes of this specific implementation, but can easily be translated into more appropriate formats for real-world use.

The basic format for a Gridlet is shown in Figure 4.1.

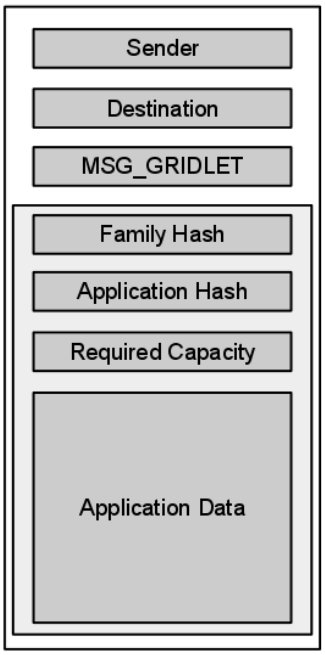

Figure 4.1: Structure of a Gridlet.

4.5 Super-peer implementation

Pastry does not contemplate different types of nodes and as such is not completely adequate for the proposed architecture. On top of the base Pastry overlay, a super-class of nodes was added to perform the work of super-peers.

Super-peers are, as previously mentioned, a subset of all nodes in the network. As such, they share a common code base in the form of the `ginger.overlay.PastryProtocol` class.

In order to implement super-peers, nodes already possess fields for handling the additional information. This approach was chosen instead of simply using the Java language inheritance mechanism due to issues found with the PeerSim platform, which, at the time this work was being implemented, did not properly support subclasses as PeerSim protocol implementations.

With this approach, what distinguishes regular nodes from super-peer nodes is the `isSuperPeer` boolean flag and the presence of an instance of the `ginger.overlay.pastry.superpeer.SuperPeerInfo` class in the `spInfo` field inside a node, which encapsulates all information the super-peer possesses.

This class contains the following mappings, from which a super-peer can obtain the information necessary to determine where sufficient nodes exist to process the currently received Gridlet:

- A mapping from application hashes to a map containing all known nodes with that application installed and their current capacity;
- A mapping of all currently known super-peers to their respective aggregate capacities.

When a Gridlet G is received by a super-peer X from one of its child-nodes and reaches the overlay management layer, it is processed through the `gridletReceived` method in the following manner: The super-peer determines which super-peer is responsible for the application family to which the Gridlet

refers to.

In the case X is the super-peer responsible for that given application family, it attempts to obtain child-nodes which can perform work on G and forwards it to at most three of those children.

Replicating work on multiple children decreases the chance for a Gridlet never to be worked upon due to the node processing it leaving the network at any time and can also yield better performance than the simpler case of assigning the Gridlet to a single node in the case the chosen node would suddenly have reduced capacity due to external intervention (e.g. a user begins locally using the node for intensive applications). This follows the work of Silva et al on *Workqueues with Replication* (WQR)[55]. Note that if a given super-peer has an application whose family hash is the one that super-peer is responsible for, the super-peer is a child node of itself.

There can be the case where there are no children nodes fit to perform work on the Gridlet X received. In that case, X looks up the other super-peers it is aware of and their respective aggregate capacity for the required application.

If X is not responsible for G, then it determines the super-peer Y by looking up in the remaining known super- peer lists and forwards the Gridlet to Y, which repeats this process until the Gridlet is forwarded to the correct child nodes and processed.

4.6 PeerSim controls

PeerSim controls, as referred above, are implemented in the `ginger.overlay.pastry.control` package. They consist of five different controls:

- `Initializer` - initializes every node on starting the simulation, so every node joins the network;
- `PastryObserver` - gathers statistics regarding the network, such as message count per message type, average hops per message, etc;
- `TrafficGenerator` - generates traffic between nodes by creating new messages at random using randomly picked parameters and applications, in a process explained below;
- `Turbulence` - generates churn on the network by disconnecting and reconnecting nodes;
- `UserControl` - allows for user interaction with the simulation.

The initializer control only runs once at the start of the simulation by randomly generating a list of applications for each node based on a configuration file and having the nodes join the network through the Pastry join protocol. The first node joins on its own, and the following nodes bootstrap on that node to join the network. This is done in order to simplify node joining as additional checks would have to be performed if a random node was chosen as the bootstrap node, such as checking if the node had already fully joined the network, adding a layer of complexity which would bring no benefit to the implementation.

The `PastryObserver` control gathers information on the network such as a count of all delivered message types and the average number of hops each message went through before reaching their destination. The hops are counted individually for each message in the `hops` field inherited from the `RawMessage` class by incrementing this field each time the message is routed through the `route` method in the `PastryProtocol`

class. Upon delivery, this count is added to the `hopStore` field in the `PastryObserver` class.

The traffic generator control randomly generates messages on each node. It can either generate Gridlets for a given application, or generate simple "ping" messages that serve no other purpose other than to generate traffic on the network. When generating Gridlets, the requirement parameters are randomly generated and the target application is chosen from the set of applications available for the simulation. From the chosen parameters the desired availability is calculated through the use of a weighted average and inserted into the Gridlet.

The message is generated and then routed to another random node like any other message. The destination node is determined by the key generated for that message and a node possessing that exact key is not required to exist, as the message is transmitted through the underlying Pastry protocol.

The `Turbulence` control generates churn by randomly disconnecting and reconnecting nodes in the network. This allows testing of the network's resilience on quasi-real life situations.

4.7 Resource representation

Existing applications are described through simple XML files. These contain information describing the application, such as its name, canonical name, version and executable location (unused in this work).

A sample XML description is shown in the following snippet:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <application>
3         <short-name>Example</short-name>
4         <version>1.0</version>
5         <canonical-name>www.example.com</canonical-name>
6         <location>/usr/bin/example</location>
7 </application>
```

Listing 4.1: Example XML application description

These descriptions are read and parsed into a specific data structure represented in the `ginger.layer.applicationlayer.` class. For the purpose of this work, the `location` field is not considered, but could be used by GINGER's *Application Adaptation* layer to launch the required application. This information is local to each node and super-peers are not required to store all this information, and instead only store the hashes obtained from this description, which are generated by the peers who wish to register this application.

The `short-name` and `version` fields are kept separate but are combined in order to determine the *application hash* for each application. Hashes are calculated using a simple MD5 hash to produce identifiers in the Pastry ID space.

The use of XML for the application description allows for a greater flexibility on the level of detail by which each application is described. If an application's minimum runtime requirements are known beforehand, these could also be included in specific elements. Alternatively, if the requirements for an application are known not to be deterministic for any given input or workload, reference values could be included as well to make sure no under-performing nodes could ever be selected to perform work on all the

Gridlets generated for that given application. This level of node filtering, however, was not implemented in the current version of gPastry and is left as future work.

An example of such a complex description is shown in the below code listing:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <application>
3          <short-name>HeavyApp</short-name>
4          <version>1.0</version>
5          <canonical-name>www.heavyapp.com</canonical-name>
6          <location>/usr/bin/heavy</location>
7          <requirements>
8                  <min-cpu>2000</min-cpu>
9                  <min-mem>1024</min-mem>
10         </requirements>
11 </application>
```

Listing 4.2: Example XML application description with requirement specification

Chapter 5

Evaluation

In order to evaluate gPastry, a number of objective metrics were defined that will allow to determine the performance of the system under different conditions.

As the nature of this work relies on gathering data on a sufficiently large peer-to-peer environment which is not possible to achieve in real-world conditions without an already existing, widespread implementation of the system, all metrics were measured through the use of the PeerSim simulator and its network observation capabilities.

5.1 Metrics

In a peer-to-peer environment, there are a number of important metrics that allow us to evaluate its overall performance and establish comparisons between different systems.

First and foremost, one must evaluate the load the protocol has on the network. This can be done by measuring the amount of messages traded between nodes as part of the protocol. This allows us to have an idea of the impact the gPastry protocol has in the overall network performance, as well as determine which parts can be improved to avoid congestion and flooding.

Secondly, it is useful to measure the amount of messages that actually reach their destination within a certain number of hops. This allows to measure the overall efficiency of the overlay. Measuring this, as well as the ratio of requests that do not find an appropriate node, is of interest in determining the resource discovery protocol's efficiency in various scenarios, such as one with resource scarcity or excess.

Additionally, we measured the average number of hops required to discover a resource through the sending of a Gridlet, as well as the average number of hops spent in super-peer redirections in order for the Gridlet to reach an appropriate node.

It would be of interest to compare the results with other algorithms and protocols such as random-walk, but this could not be done in time for this dissertation due to the need of implementing those protocols.

5.2 Procedure

The work done on gPastry was evaluated by using the PeerSim P2P network simulator. PeerSim allows for the creation of "controls", which can act as entities external to the entire network, thus providing them with global knowledge about the network and it's constituent parts.

In order to evaluate gPastry, an `Observer` control was implemented for the PeerSim platform that is able to gather statistical data generated by running the simulation. Additionally, the `TrafficGenerator` control was used to generate traffic in the network in order to obtain more realistic measurements.

The simulation was run with gPastry in increasing numbers of 1000, 5000 and 10000 peers. The average number of applications possessed by each node was also taken into account. Unfortunately, due to time constraints, it was impossible to obtain measurements that show how protocol performance changed by altering the statistical distribution of applications and application families. For the purposes of this evaluation, we assume a uniform distribution of applications amongst all nodes. In total, there were 3 application families, with one, two and three applications each, respectively.

The `TrafficGenerator` control generates Gridlets of the different applications described for the simulation through XML files according to the definitions presented in the previous chapter and assigns them a random source and destination node. Capacity metrics for those gridlets are also generated randomly in order to simulate different sets of data for each application.

All nodes join the network during an initial phase and Gridlets can be forwarded during that period through the nodes that already joined the network successfully. Nodes that have connected but have not completed the join process completely are not taken into account.

Since the tests are being processed in a simulated environment, all time units refer to the number of simulation cycles processed. For this work, we use a number of 500000 cycles. In each cycle, nodes can send or messages with low probability. However, the total number of messages will still be extremely large due to the duration of the simulation.

In effect, in each case, the key points to test were:

- Effectiveness - Whether the desired resources were located in a limited number of hops;
- Efficiency - Number of messages necessary to locate the desired resources, as well as number of hops required for a Gridlet to reach the appropriate destination;

5.3 Results

We now present the results from the evaluation of gPastry on PeerSim, as described in the previous sections. We begin by evaluating the effectiveness of the solution i.e. if the proposed resource discovery mechanism is able to locate resources when they exist. Later, we evaluate its efficiency by analyzing the number of hops Gridlets must go through in order to reach an available node.

5.3.1 Effectiveness

In order to evaluate the effectiveness of the protocol we analyze the ratio between the total number of Gridlets that were sent by the peers and the number of those Gridlets that actually reached a node where

they were processed at various points in the simulation.

In this simulation through Figure 5.1 we see that initially, when nodes are still joining the network, a large number of Gridlets do not find any appropriate node in the allotted number of hops. However, as more nodes join the network and register their availability in super-peers, the amount of Gridlets that successfully locate the required resources increases, stabilizing at different points in the various cases. Efficiency stabilizes more rapidly with larger numbers of nodes, as expected since there is a greater number of super-peers and a larger number of peers from which resources can be tapped. The number

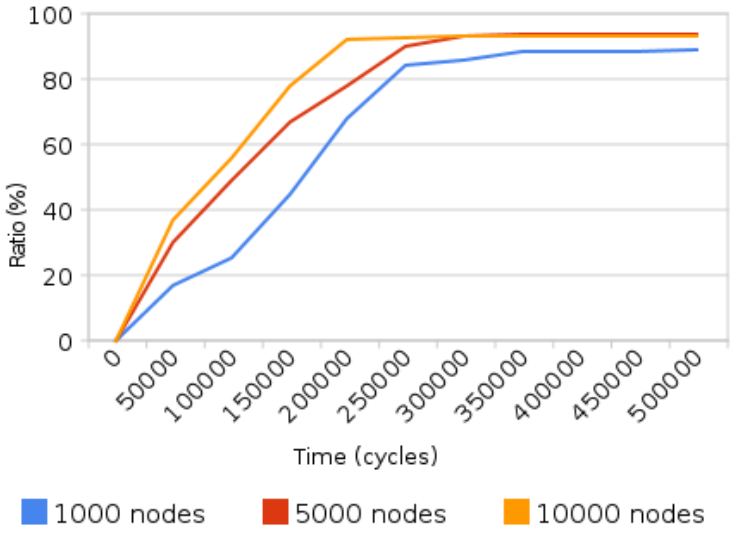

Figure 5.1: Resource discovery effectiveness

of super-peers existing in the network grow slowly, in the end accounting for less than 1% of the total number of peers. For smaller numbers of nodes, efficiency is lower during most of the simulation as it

N	Number of super-peers	Total Gridlets	Successful Gridlets
1000	27	125498	111693
5000	63	637298	598677
10000	109	1374089	1277902

Table 5.1: Effectiveness statistics

requires nodes to be both joined to the network and free to perform tasks. As nodes become busy, they become unavailable to work on new requests so the overall effectiveness of the system through time grows more slowly as at each instant there is always a number of nodes that are performing work and cannot be used for new Gridlets.

We were also unable to test the effectiveness of gPastry under churn conditions. With nodes continuously joining and leaving the network, the information stored by super-peers regarding their availability, as well as the availability of other super-peers could become inaccurate, as leaving nodes would reduce the effective capacity of a cluster and joining nodes could increase it.

This issue, however, would probably be somewhat mitigated by the currently implemented threshold system, that prevents super-peers from communicating changes in their cluster's availability immediately for any changes. If the changes caused by churn were not significant enough, super-peers could be able to perform work as requested, as long and the total requested capacity did not exceed the actual capacity

present in that cluster.

Under heavy load for a given family of applications, the super-peer responsible for that family could receive Gridlets that were sent by other super-peers in the assumption that they had capacity to perform work on those same Gridlets, but in case of shortage, it would be unable to cope, as the advertised capacity did not correspond to the actual values. Unfortunately this aspect could not be evaluated and the actual results are as of now unknown.

5.3.2 Efficiency

In order to evaluate gPastry with regards to efficiency, we have to look at the number of messages required to locate resources, as well as the number of hops Gridlets must take throughout the overlay in order to locate them. The results presented next show that, even with the introduction of super-peers, the number of hops necessary for a Gridlet to reach its destination is not significantly impacted.

There is a number of hops spent in super-peer redirection, but it levels rapidly as the number of super-peers stabilizes and they become known amongst themselves as to become nearly insignificant. This is due to the fact that in all cases, the number of super-peers is small enough that they tend to be known among each other fairly rapidly through the use of `PastrySuperQuery` messages. The average amount of hops for each test can be seen in Figure 5.2. The number of messages exchanged throughout the

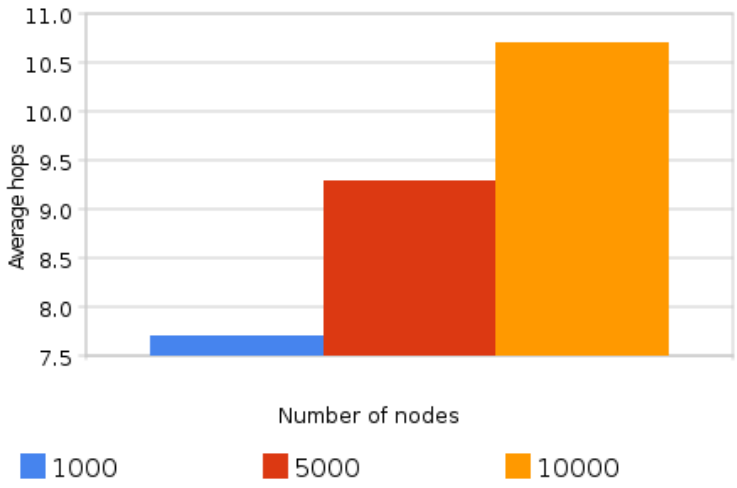

Figure 5.2: Number of hops for Gridlets

simulation is shown to be fairly high and there is much space for improvement, especially in regards to the amount of messages spent in querying for super-peers and the subsequent replies. The results are illustrated in Figure 5.3 and Table 5.2.

These messages alone account for more than 10% of all traffic in the network, which is overwhelmingly high for messages whose only function is to keep nodes informed of the current super-peers. The basic messages inherited from the Pastry overlay do not significantly impact the protocol, as they account for about 5% of the entire amount in the current tests, a number that should be lesser in a real world environment where the overlay does not come up entirely at the same time. Unfortunately, we were unable to test the protocol's performance in an environment that caused messages to fail to reach their destination, as well as introducing various degrees of churn to simulate more realistic environments. We

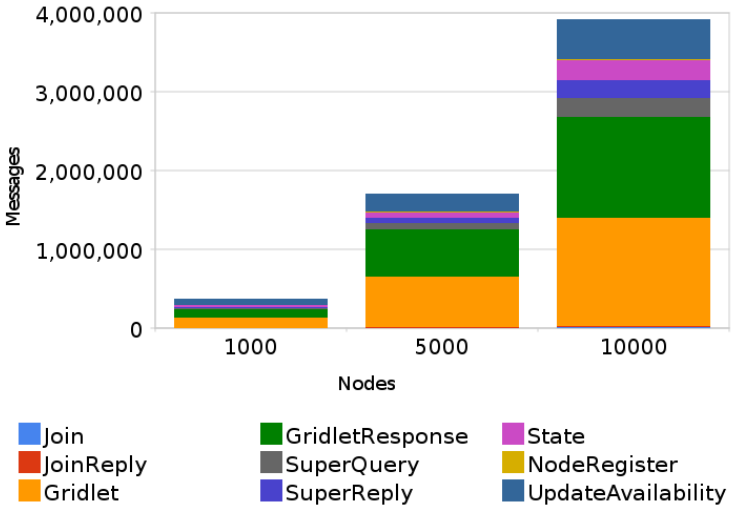

Figure 5.3: Number of messages exchanged, by type

expect such environmental characteristics to lower the overall efficiency of the system, as more messages would have to be exchanged to maintain the structured nature of gPastry.

Churn in particular would greatly increase the number of messages exchanged for node initialization and state exchange, as well as the amount of messages that query the network for super-peers and update availability. As nodes joined and left the network, the available capacity on each cluster under each super-peer would fluctuate enough so that updates would become more frequent in order for super-peers to maintain updated information over the availability of their nodes. With greater degrees of churn, more of these messages would have to be exchanged, and a great deal of traffic would be spent in overlay management as opposed to perform any useful work.

N	Join	JoinReply	Gridlet	GridletResponse	SuperQuery	SuperReply	State	NodeRegister	UpdateAvailability	Total
1000	999	1999	125489	111693	14385	13975	27611	1340	73097	370597
5000	4999	9998	637298	598677	75823	72343	70934	9478	230489	1710039
10000	9999	19998	1374089	1277902	239095	229074	249758	16807	508497	3925219

Table 5.2: Number of messages exchanged, per type

Chapter 6

Conclusion

In this work, a resource discovery model for Peer-to-Peer networks in the context of CPU cycle sharing was presented. As the available computational capacity available online grows as a consequence of Moore's Law and the expansion of broadband access, new protocols that allow users to exploit these resources in their everyday tasks grow ever more important. Through the use of systems such as GINGER, popular applications can transparently increase their perceived performance by leveraging the massively parallel computing power available throughout the world.

The obtained results show that the approach followed in the development of this system works and that resource discovery in P2P environments can be achieved efficiently through the use of a hierarchical topology with super-peers, an approach that is already well known from existing file-sharing Peer-to-Peer applications such as Gnutella. The amount of messages exchanged using the protocol in this work were clearly sub-optimal, but we believe this is more due to the current implementation, which tends to send far more messages than those required for the purposes of maintenance, than to the actual protocol.

The amount of information to be stored in each peer can become significant after large periods of time but intelligent management of such information, added to the fact that many peers only remain connected for relatively small amounts of time minimize this problem. Overall, the overhead imposed by maintaining the additional information is compensated by the improvements this protocol brings in the area of resource discovery.

Unfortunately, the lack of comparison with other protocols does not allow a comparative analysis of gPastry, and as such we cannot assert that gPastry is significantly better than any of the current solutions for this same problem.

6.1 Future Work

In the future, running this simulation in a PlanetLab environment would allow to evaluate this protocol in a more realistic environment, although at a lesser scale than the one provided by this simulation. It would also be interesting to attempt to use different topologies, as some of the current limitations may not exist using different topologies.

Exploring the possibility of adding constraints and filtering to the resource discovery protocol in order to locate better suited resources, as opposed to the current model, could greatly improve the efficiency with which resources are used in the network. The use of different representations for node capacity that

allow to distinguish between different characteristics such as CPU power, available memory and others would also be a requirement in order for this system to work as expected.

Although unimplemented in the current version of gPastry, using Gridlet replication through the use of distributed storage systems such as PAST would allow to add greater security benefits to the solution, namely achieving greater robustness in churn conditions as well as prevent the forging of results, either maliciously or due to failures in the processing peers.

It would also be of interest to test gPastry in situations of churn, where structured P2P networks tend to become more unstable and prone to failure, but this could not be accomplished for this project.

The protocol used for determining the best nodes for a given Gridlet can be improved through the use of reputation and trust mechanisms. Using a reputation metric can improve node selection by selecting more often nodes that perform their work quicker while avoiding nodes which disconnect frequently, thus compensating for high churn scenarios. Trust metrics would also allow to isolate malicious nodes attempting to corrupt the results on the network.

In a P2P environment, it is impossible to have a centralized reputation or trust manager, and as such, that information could likely be incorporated in the super-peers developed for the purposes of this work.

Additionally, other techniques would be of interest to improve the proposed protocol, such as finding a better balance between all relevant metrics regarding node performance will allow for a better overall selection of nodes, or allowing, when possible, for nodes to include information about the expected duration of the current load. Other nodes could cache this information for the specified amount of time and refer directly to it, which could be useful to lessen the amount of messages sent through the network (such as described in the work of Filali et al[24]).

In a real-world scenario, security is of great concern, and work looking towards securing Gridlets through the use of mechanisms like digital signatures as well as better ways to prevent malicious falsification of results would be of great interest, but are beyond the scope of this work.

Bibliography

[1] Avahi. `http://avahi.org`.

[2] DNS Service Discovery. `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt`.

[3] Multicast DNS. `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`.

[4] Zero Configuration Networking (Zeroconf). `http://www.zeroconf.org`.

[5] J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on planetlab with sword. In *In WORLDS*, 2004.

[6] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[7] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[8] N. Andrade, W. Cirne, F. V. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 61–86. Springer, 2003.

[9] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In R. L. Graham and N. Shahmehri, editors, *Peer-to-Peer Computing*, pages 33–40. IEEE Computer Society, 2002.

[10] Z. Balaton, G. Gombas, P. Kacsuk, A. Kornafeld, J. Kovacs, A. Marosi, G. Vida, N. Podhorszki, and T. Kiss. SZTAKI desktop grid: a modular and scalable way of building large computing grids. In *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pages 1–8, 2007.

[11] M. Balazinska, H. Balakrishnan, and D. R. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery. In F. Mattern and M. Naghshineh, editors, *Pervasive*, volume 2414 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2002.

[12] S. Banerjee, S. Basu, S. Garg, S. G. andP. Sharma. Scalable grid service discovery based on uddi. In *In Proceedings of the 3rd International Workshop on Middleware for Grid Computing (MGC'05)*, pages 1–6, 2005.

[13] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In R. Yavatkar, E. W. Zegura, and J. Rexford, editors, *SIGCOMM*, pages 353–366. ACM, 2004.

[14] A. Cheema, M. Muhammad, and I. Gupta. Peer-to-peer discovery of computational resources for grid applications. *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 7 pp.–, Nov. 2005.

[15] 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(5):597–610, 2003.

[16] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72. Citeseer, 2003.

[17] A. Congiusta, C. Mastroianni, A. Pugliese, D. Talia, and P. Trunfio. Enabling knowledge discovery services on grids. In M. D. Dikaiakos, editor, *European Across Grids Conference*, volume 3165 of *Lecture Notes in Computer Science*, pages 250–259. Springer, 2004.

[18] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. P-tree: a p2p index for resource discovery applications. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 390–391, New York, NY, USA, 2004. ACM.

[19] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, pages 23–, 2002.

[20] R. Y. de Camargo, F. C. Filho, and F. Kon. Efficient maintenance of distributed data in highly dynamic opportunistic grids. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1067–1071, New York, NY, USA, 2009. ACM.

[21] R. Y. de Camargo and F. Kon. Distributed data storage for opportunistic grids. In *MDS '06: Proceedings of the 3rd international Middleware doctoral symposium*, page 3, New York, NY, USA, 2006. ACM.

[22] K. El Defrawy, M. Gjoka, and A. Markopoulou. BotTorrent: misusing BitTorrent to launch DDoS attacks. In *Proceedings of the 3rd USENIX workshop on Steps to reducing unwanted traffic on the internet*, pages 1–6. USENIX Association, 2007.

[23] P. D. et al. FreePastry. `http://www.freepastry.org`.

[24] I. Filali, F. Huet, and C. Vergoni. A simple cache based mechanism for peer to peer resource discovery in grid environments. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:602–608, 2008.

[25] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[26] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[27] E. Guttman and S. Microsyst. Service location protocol: automatic discovery of IP networkservices. *IEEE Internet Computing*, 3(4):71–80, 1999.

[28] S. Hand, T. Harris, E. Kotsovinos, and I. Pratt. Controlling the xenoserver open platform. In *Open Architectures and Network Programming, 2003 IEEE Conference on*, pages 3–11. IEEE, 2003.

[29] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[30] N. Hemming. KaZaA. *Web Site-www. kazaa. com.*

[31] N. Inc. The napster homepage. *Online: http://www. napster. com*, 2000.

[32] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *PODC*, pages 213–222, 2002.

[33] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[34] T. Kiss, P. Kacsuk, P. Kacsuk, N. Podhorszki, and N. Podhorszki. Scalable desktop grid system. In *Institute on System Architecture, CoreGRID - Network of Excellence*, pages 1–13, 2006.

[35] T. Klingberg and R. Manfredi. Gnutella protocol specification, June 2002.

[36] S. Larson, C. Snow, M. Shirts, and V. Pande. Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology. 2009.

[37] X. Liao, H. Jin, Y. Liu, L. Ni, and D. Deng. Anysee: Peer-to-peer live streaming. In *IEEE INFOCOM*, volume 6. Citeseer, 2006.

[38] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[39] G. S. Manku, M. Bawa, P. Raghavan, and V. Inc. Symphony: Distributed hashing in a small world. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[40] A. Marosi, G. Gombás, Z. Balaton, P. Kacsuk, and T. Kiss. Sztaki desktop grid: Building a scalable, secure platform for desktop grid computing. *Making Grids Work*, pages 365–376, 2008.

[41] M. Marzolla, M. Mordacchini, and S. Orlando. Resource discovery in a dynamic grid environment. In *DEXA Workshops*, pages 356–360. IEEE Computer Society, 2005.

[42] M. Marzolla, M. Mordacchini, and S. Orlando. A p2p resource discovery system based on a forest of trees. *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Conference on*, pages 261–265, 0-0 2006.

[43] M. Marzolla, M. Mordacchini, and S. Orlando. Peer-to-peer systems for discovering resources in a dynamic grid. *Parallel Comput.*, 33(4-5):339–358, 2007.

[44] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, editors, *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[45] E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Comput. Netw.*, 52(11):2097–2128, 2008.

[46] S. Miles, J. Papay, V. Dialani, M. Luck, K. Decker, and L. Moreau. Personalised grid service discovery. In *IEE Proceedings Software: Special Issue on Performance Engineering*, pages 131–140, 2003.

[47] B. Miller, T. Nixon, C. Tai, and M. Wood. Home networking with universal plug and play. *Communications Magazine, IEEE*, 39(12):104–109, Dec 2001.

[48] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Scalable wide-area resource discovery.

[49] F. Pianese, D. Perino, J. Keller, and E. Biersack. PULSE: an adaptive, incentive-based, unstructured P2P live streaming system. *IEEE Transactions on Multimedia*, 9(8):1645–1660, 2007.

[50] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[51] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht (awarded best paper!). In *USENIX Annual Technical Conference, General Track*, pages 127–140. USENIX, 2004.

[52] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, pages 329–350, 2001.

[53] A. I. T. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, pages 188–201, 2001.

[54] H. Schulze and K. Mochalski. Internet Study 2008/2009. *IPOQUE Report*.

[55] D. P. D. Silva, W. Cirne, F. V. Brasileiro, and C. Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Applications on Computational Grids, in Proc of Euro-Par 2003*, pages 169–180, 2003.

[56] D. Spence and T. Harris. Xenosearch: distributed resource discovery in the xenoserver open platform. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing*, pages 216–225, 22–24 June 2003.

[57] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[58] D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7:96, 94–95, 2003.

[59] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, and S. Haridi. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems*, 23(7):864 – 878, 2007.

[60] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 783–788, May 2007.

[61] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for supporting streaming applications. In *IEEE Global Internet*, pages 1–6. Citeseer, 2006.

[62] J. Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.

[63] B. Yang and H. Garcia-Molina. Designing a super-peer network. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 49–. IEEE Computer Society, 2003.

[64] Y. Yue, C. Lin, and Z. Tan. Analyzing the performance and fairness of bittorrent-like networks using a general fluid model. In *GLOBECOM*. IEEE, 2006.

[65] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *INFOCOM*, pages 2102–2111. IEEE, 2005.

[66] D. Zhou and V. Lo. Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system. pages 66–73, 2004.