INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Scalable and Efficient Discovery of Resource, Applications, and Services in P2PGrids

Raoul Gabriel Martins Felix

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

Júri

Presidente:	Professor António Rito Silva
Orientador:	Professor Luís Veiga
Co-orientador:	Professor Paulo Ferreira
Vogais:	Professor Sérgio Duarte

Outubro de 2010

Abstract

Distributed computing enables us to harness all the resources and computing power of the millions of computers connected to the Internet. Therefore, this work describes the construction of an efficient and scalable resource discovery mechanism, capable of searching not only for physical resources (e.g. CPU, Memory, etc.), but also services (e.g. facial recognition, high-resolution rendering, etc.) and applications (e.g. ffmpeg video encoder, programming language compilers, etc.) from computers connected to the same Peer-to-Peer Grid network. This is done in a novel way by combining all resource information into Attenuated Bloom Filters, which also allows us to efficiently route messages in a completely decentralized unstructured P2P network (no super-peers). The research shows that previous P2P, Grid, and Cycle Sharing systems tackled this problem by focusing on each resource type in isolation, such as (physical) resource discovery and service discovery. Methods to minimize storage and transmission costs were also researched. The discovery mechanism was evaluated with a number of different test scenarios that varied resource distribution, resource values, topologies, etc. For comparison, we also evaluated the Random Walk discovery method which served as a baseline. The results were favorable over Random Walk, having higher query success rates with less hops while requiring increase in message size and storage space at each node (for routing information), thus attaining our objectives of effectiveness, efficiency, and scalability.

Keywords

Resource Discovery, Service Discovery, Application Discovery, Bloom Filter, P2P, Grid

Resumo

A computação distribuída permite-nos tirar proveito de todos os recursos e poder computacional de milhões de computadores ligados à Internet. Neste sentido, este trabalho descreve a construção de um mecanismo de descoberta de recursos escalável e eficiente, capaz de procurar não só por recursos físicos (e.g. CPU, Memória, etc.), mas também por serviços (e.g. reconhecimento facial, renderização de alta definição, etc.) e aplicações (e.g. ffmpeg codificador de vídeo, compiladores de linguagens de programação, etc.) de computadores ligados à mesma rede Peer-to-Peer Grid. Isto foi feito de forma inovadora, combinando toda a informação sobre os vários recursos em Attenuated Bloom Filters, o que também nos permite encaminhar mensagens de forma eficiente numa rede descentralizada P2P não estruturada (sem super-peers). O estado da arte demonstra que os sistemas anteriores de P2P, Grid, e Cycle Sharing resolveram este problema focando cada tipo de recurso de forma isolada, nomeadamente a descoberta de recursos (físicos) e a descoberta de serviços. Técnicas para minimizar o custo de armazenamento e de transmissão também foram pesquisadas. Este mecanismo de descoberta foi avaliado num de conjunto cenários de teste que variam a distribuição de recursos, os valores dos recursos, as topologias, etc. Para efeitos de comparação, avaliamos também o mecanismo de descoberta Random Walk que serve como *baseline*. Os resultados dos testes favoreceram o sistema desenvolvido relativamente ao Random Walk, tendo maior taxa de satisfação de pedidos consumindo menos *hops*, mas à custa de mensagens maiores e maior armazenamento em cada nó (para informação de encaminhamento). Este resultados permitiram concluir que os objectivos propostos de efectividade, eficiência, e escalabilidade foram atingidos.

Palavras Chave

Descoberta de Recursos, Descoberta de Serviços, Descoberta de Aplicações, Bloom Filter, Sistema Entre Pares, P2P, Grid

Acknowledgements

First and foremost I would like to offer my sincerest gratitude to my advisor, Prof. Dr. Luís Veiga, who throughout my thesis has supported me with his knowledge, guidance, and his patience, whilst giving me room to work in my own way. Without his help this thesis would not have been completed or even written. To put it simply, one could not wish for a better or friendlier supervisor.

Secondly, I would like to thank my co-advisor, Prof. Dr Paulo Ferreira, and Prof. Dr. Sérgio Duarte for their valuable feedback from the initial version of the dissertation which helped guide me the rest of the way.

I am ever so grateful for the support my mother, Amelia Felix, and father, José Felix, gave me during during the writing of this thesis, and for the sacrifices they made for me. Without the love and support you have given me my entire life I would not be where I am today and been able to complete this work.

I am especially grateful for the love, caring, and understanding my girlfriend, Tatiana Guerra, gave me during this period of my life, for once again, without her, the completion of this thesis would not have been possible.

To my classmates and friends André Veiga, Antonio Higgs, João Edmundo, Pedro Cruz Sousa, Pedro Sousa, Rodrigo Dias, and Tiago Garrochinho, who made all those afternoons working at Tagus fun, for their motivational support, and for the *HoN* games which helped alleviate the stress at the end of the day.

I would also like to thank my long time friends, Marcelo Mendes and Pedro Sebastião, who have been with me since moving to Portugal and have played a crucial part in my life. Thank you for your support and for our outings which helped me relax during such a stressful time.

Finally, I would like to thank my family both in Portugal and South Africa for their support and good wishes. As a small mention, even though she's not able to understand this, I would like to say thank you to my dog Safira for her company during all those afternoon naps when I had hard problems on my mind to solve.

Contents

List of Figures

List of Tables

Abbreviations

P2P	**P**eer-**to**-**P**eer
GINGER	**G**rid **I**nfrastructure for **N**on-**G**rid **E**nvi**R**onments
GiGi	**G**rid **I**nfrastructure for Non-**G**rid Env**I**ronments
SERD	**S**calable and **E**fficient **R**esource **D**iscovery
RW	**R**andom **W**alk
PDA	**P**ersonal **D**igital **A**ssistant
CPU	**C**entral **P**rocessing **U**nit
RAM	**R**andom **A**access **M**emory
DHT	**D**istributed **H**ash **T**able
ID	**ID**entifier
NASL	**N**ode **A**ctivity **S**pecification **L**anguage
NRDL	**N**ode **R**esource **D**escription **L**anguage
XML	e**X**tended **M**arkup **L**anguage
GML	**G**raph **M**odelling **L**anguage

Dedicated to my family.

Chapter 1

Introduction

1.1 Background

There are millions of computers connected to the Internet in today's age.[1] The number of people interconnected around the globe by this medium is growing every day as mobile devices such as laptops, netbooks, PDAs, and smartphones go online. Not only are there more and more computers online, but their computing capacity is also increasing rapidly.

Distributed computing on such a large scale cannot be ignored. With so many computers all connected to the same network, with increasingly larger capacities, the most logical thing to do is to find a way to harness these resources. As such, resource sharing has become immensely popular and has led to the development of Grid and Peer-to-Peer (P2P) infrastructures.

The most popular form of resource sharing across the Internet is File Sharing via Peer-to-Peer applications. Peer-to-Peer traffic is responsible for roughly 50%-90% of all Internet traffic.[2] A lot of work has been done in this area to create robust and scalable systems, capable of tolerating a large number of users. P2P infrastastructures can be divided into two major categories: unstructured and structured. Unstructured systems like Gnutella[1] and Freenet[2] do not perform any organization of nodes, as opposed to structured systems, such as Chord[3], Pastry[4], CAN[5], and Kademlia[6], which maintain nodes in an organized structure to speed up message routing. Nevertheless, systems in both categories have something in common: they operate in a decentralized manner with volunteered computers that belong to, and are administered by, different owners, unlike Grid infrastructures where administration is federated.

[1] http://www.internetworldstats.com
[2] http://torrentfreak.com/bittorrent-dominates-internet-traffic-070901

Grid and Cycle Sharing systems are similar in nature, as their objective is to perform large-scale parallel computations in scientific and corporate communities. While Grid systems harness the power of many interconnected networks of computers, which are usually centrally or hierarchically managed by the institutions that run them; Cycle Sharing systems take advantage of the many idle computers and game consoles already connected to the Internet, volunteered by home users.

In the literature[7–10] it is said that Grid and Peer-to-Peer systems will eventually converge. As Grids increase in size, they will tend towards P2P systems, and as P2P systems become more complex, they will tend towards Grids.

In this fashion, GINGER[11] (**G**rid **I**nfrastructure for **N**on-**G**rid **E**nvi**R**onments), or simply GiGi, is a P2P Grid infrastructure that fuses three approaches (grid infrastructures, distributed cycle sharing, and decentralized P2P architectures) into one. GiGi's objective is to bring a Grid processing infrastructure to home-users, i.e. a "grid-for-the-masses" (e.g. achieve faster video compression, face recognition in pictures/movies, high-res rendering, molecular modeling, chemical reaction simulation, etc.).

1.2 Research Focus

The common theme between the different aforementioned systems is that users have a task that they want to accomplish: share files in P2P file sharing systems; perform scientific calculations in Grids; or perform CPU intensive tasks over a massive amount of idle home user computers in Cycle Sharing systems. The requirements to perform each of these tasks can range from almost no requirements (file sharing), to simple requirements (idle CPU), to complex requirements (free CPU with X much RAM, with at least Y much storage space, and with application Z installed).

Tasks can be run over a large number of distributed computers. But in order to do that, computers need to be able to find resources that satisfy task requirements from other nodes in the network. For this to be possible, nodes in a network need to be able to notify others about what resources they possess and their availability (e.g. a CPU being used 100% cannot be used by another task); and nodes need to be able to locate other peers that contain the necessary resources to perform a task.

This is where Resource and Service Discovery protocols come in, for without them, P2P, Grid, and Cycle Sharing systems would be rendered almost useless for computation. Having a good resource discovery mechanism can make or break a system. Therefore, this dissertation presents a discovery protocol of (physical) resources, applications, and services for inclusion in the GINGER project.

1.3 Objectives and Contributions

The overall aim of this work is to enhance the discovery mechanism present in the GINGER project, making it more complete and decentralized (no superpeers). Unlike the current discovery protocol implemented in GINGER which only searches for physical resources (e.g. CPU, Memory, etc.), this work needs to be able to locate not only basic resources such as CPU, memory, and bandwidth, but also applications, services, and libraries that are installed in each of the nodes that form the P2P Grid. This system should also be scalable by adapting to a large number of nodes, and be as efficient as possible in terms of space occupied in each node and size of transmitted messages over the network.

Specifically, the objectives of this work are the following (with the related contributions described in the next paragraphs):

1. Analyze previous resource and service discovery methods in Peer-to-Peer, Grid, and Cycle Sharing systems.
2. Assess various methods used to represent information in an efficient manner.
3. Develop a resource, service, and application discovery mechanism to improve the current discovery mechanism used in GINGER.
4. Construct a system that is effective (in terms of replying successfully to queries), scalable (adapt to a highly dynamic node population) and efficient (in terms of storage, number of network messages, and message size).
5. Evaluate the proposed discovery mechanism in a simulated environment against another discovery mechanism.

The contributions to Objectives 1 and 2 form the core of the Related Work review and involve the analysis of discovery methods used in different types of systems along with their performance, as well as the assessment of various techniques to help reduce the storage and data transmission costs of resource, service, and application information.

The contributions to Objectives 3 and 4 comprise the core of this work. A new discovery mechanism to enhance resource discovery in GINGER is described, by being more complete (find resources, services, applications, and libraries); decentralized (without resorting to super-peers); efficient (in terms of occupied space and network message length); and scalable.

In the last contribution, targeting Objective 5, the system is evaluated in a simulator and compared to another discovery mechanism. Results are then analyzed with regards to the satisfaction of aforementioned objectives.

1.4 Organization of the Dissertation

This dissertation is divided into five chapters. Chapter 1 gives an introduction to the research area, and presents the objectives and contributions of the work. Chapter 2 contains the Related Work, where similar systems that provide resource and service discovery are analyzed, along with various techniques to store information in an efficient manner. Chapter 3 describes the SERD discovery mechanism's architecture and Chapter 4 describes its implementation. Chapter 5 presents the evaluation of the system, benchmarking its performance by extensive simulation with various parameters, and comparing it to another discovery mechanism in order to determine whether the efficiency and scalability objectives have been met. Chapter 6 concludes this dissertation by summarizing the work, offering final remarks, and providing suggestions on how to further continue the development of the system.

1.5 Scientific Publications

A scientific paper describing a preliminary version of this work was published and presented at the conference *INForum 2010* under the title "Scalable and Efficient Discovery of Resources, Applications, and Services in P2P Grids."

Chapter 2

Related Work

The main objective of this work is to create a resource discovery mechanism capable of searching for physical resources (e.g. CPU, memory, storage, etc.) and services (e.g. facial recognition, high-resolution rendering, etc.) offered by volunteered computers in a network, as well as installed applications (e.g. ffmpeg video encoder) or libraries (e.g. Boost C++ library). Previous work has focused on each of the aforementioned types of resources in isolation. Therefore, Section 2.1 will discuss the various systems that make use of resource discovery mechanisms that search either for physical computer resources or for computer files, and Section 2.2 will present the various systems that enable the location of various types of services offered by computers in a network. But, for all of these systems to work, we need to store information about the various resources each computer has and transfer it over the network. Thus, Section 2.3 deals with the many ways data can be stored and transmitted, emphasizing efficiency. Finally, Section 2.4 will conclude the related work analysis and also present an overview of the analyzed works.

2.1 Resource Discovery

Resource discovery[12] consists of performing a search for resources, either hardware or software, offered by many computers connected to a network. In this section, we will consider resources to be either physical (e.g. CPU, memory, bandwidth, etc.) or virtual (e.g. computer files). There are many uses for such a discovery mechanism: applications can locate files shared by many users, powerful computers with specific requirements can be searched for in order to perform large-scale parallel computations, and idle computers with enough storage and CPU cycles to perform large computationally intensive tasks can be found.

Therefore, in this section we will consider Peer-to-Peer systems used by file sharing applications and by resource discovery mechanisms for Grid environments (Section 2.1.1); traditional Grid systems that are not based on Peer-to-Peer models used by scientific and commercial communities (Section 2.1.2); and Cycle Sharing systems used for academic and scientific projects (Section 2.1.3).

2.1.1 Peer-to-Peer

Peer-to-peer systems[7, 12–14] are characterized by the principle that every component in the system is equal. There are no servers and no clients; each component acts as both, and are normally referred to as *servents* (from the words **ser**ver and cli**ent**), peers, or nodes. P2P systems can be split into two main categories based on the way they organize connections to their neighbors, namely unstructured and structured. We can further define a third category: hybrid, which attempts to merge the best from unstructured and structured systems into one.

2.1.1.1 Unstructured

In unstructured systems, nodes are randomly connected to a fixed number of neighbors. There is no information about the location of resources (e.g. files) and, therefore, these systems need to use searching techniques that contact other peers in the network, like flooding, to perform lookups. Flooding[1, 15] is extremely inefficient and is the reason why unstructured systems do not scale well. Several methods have been proposed to address this situation such as: random walks[16], iterative deepening[1, 17], probabilistic forwarding[17, 18], learning-based[16, 17, 19], and heuristic-based[2]. Even though the lack of structure in these systems may lead to inefficient searching, they has the advantage of being able to adapt to a very transient node population, in which nodes join and leave at a high rate.

Napster[20] was the first massively popular P2P system used for file distribution, namely MP3 files.We are considering this system for historical reasons, as it is very different from the unstructured P2P systems of today. It relies on a central directory server that maintains a mapping of clients to the audio files they share. Searches are performed on behalf of the clients, resulting in the peer-nodes' addresses that contains the requested song. The client that initiated the search then connects directly to a node with the desired file and starts the transfer, thus being considered a Peer-to-Peer system. The use of a centralized server has two major drawbacks: it becomes a bottleneck as the number of users increases and is not scalable, but this was mitigated by allowing

multiple directory servers; it is a single point of failure and is vulnerable to Denial-of-Service attacks or legal measures (which was the case with Napster), disrupting the whole system and disabling it.

After Napster's demise, **Gnutella**[1] was the next major P2P network, with 1.18 million computers connected to it as of June 2005[1]. It is a totally decentralized system and a classical example of an unstructured P2P network that exchanged files. As each node was connected to a small number of neighbors, file search queries were propagated to each one, i.e. it used basic breadth-first flooding with iterative deepening (Figure 2.1). Using iterative deepening limited the flooding depth by assigning a Time-To-Live to queries that started from 1 and continued until depth D, or until a certain number of results were returned. But it also limited the scope of the search to a certain depth making it impossible to find rare files, so this technique really only works well with popular and well-replicated files.

With the message propagation strategy used by Gnutella, depending on the number of results, more and more nodes were queried. This made the network unscalable, for the amount of needed bandwidth grew exponentially as the number of searched nodes increased, leading to saturation. Low capacity nodes were the most affected and were rendered useless, causing enormous delays and making the search mechanism completely unreliable. On top of all this, frequent peer disconnects, also known as churn, never allowed the network to stabilize (40% of nodes leave the network in less than 4 hours[21]).

Even though Gnutella is unscalable, the work done in **GIA**[22] improves Gnutella's simple architecture and proved to be three to five orders of magnitude better. This was done by building on previous works: instead of the inefficient flooding propagation strategy, random walks are used (explained later in this Section); to make the random walk strategy perform better, network topology adaptation techniques are used to ensure high capacity nodes are the ones with a higher degree of neighbor connections; and, finally, to prevent overloading a node, GIA uses a token-based flow control algorithm that only allows messages to be forwarded to a node if that node explicitly notifies the sender of its willingness to receive messages.

Freenet[2] is a third generation[2] Peer-to-Peer system, with the goal of providing "uncensorable and secure global information storage"[2]. It uses a decentralized architecture and enables users to anonymously publish and retrieve files. Not relying on a central server is important for this system because it avoids having a single point of failure - even if one or more nodes are taken down, by say a government, a corporation, or others, it will still be able to survive and function. It is different from a typical file sharing

[1] http://www.slyck.com/news.php?story=814
[2] http://www.ucalgary.ca/it/help/articles/security/awareness/p2p#generation

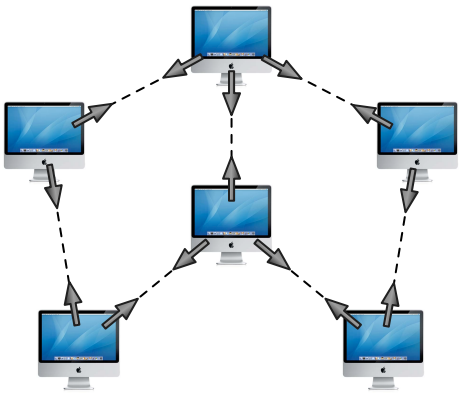

FIGURE 2.1: Flooding in Gnutella

system like Gnutella, as it only allows users to insert files (and not remove them) into the network, due to the replication of files to a number of other nodes. This provides high reliability, as the file will still be accessible even if the node that submitted the file goes offline.

Rather than using an unscalable searching technique like Gnutella's flooding, Freenet uses heuristics, i.e. the solution is not guaranteed to be optimal, but a good one is usually found in a reasonable amount of time. More specifically, Freenet uses a steepest-ascent hill-climbing search[7]: all neighboring nodes are compared and the search query is forwarded to the node that is the closest to the target. If the search path results in a dead-end or a loop it uses backtracking and tries another path of nodes.

Moving away from file sharing systems, **Iamnitchi et al.**[16] propose a fully decentralized Peer-to-Peer architecture for Grid Resource Discovery. Participants are called Virtual Organizations (VO) and can be individuals (like in an ordinary P2P system) or institutions (like an institutional grid). Each VO publishes resource information to one or more local servers that participate in the network, called peers or nodes. Resource discovery in this system assumes that a node answers if a request matches locally, otherwise it uses a request propagation technique; those requests have a TTL and stop when it reaches zero; and the topology is unstructured.

Four propagation strategies are discussed, with trade-offs between the amount of storage space used in each node and the search performance. The **random walk** strategy is the simplest: the request is forwarded to a randomly chosen node. It is also the strategy that performed the worst, but has the advantage of not requiring additional storage space for routing information. The **learning-based** technique forwards search queries to nodes that have answered similar requests in the past by keeping track of previously answered queries by other nodes. If no previous information has been recorded, then the random walk strategy is used. This proved to be the best all-round strategy once

the cache starts to fill up. The **best-neighbor** method also records answers from other nodes, but ignores the type of answered requests. Queries are then forwarded to the node who answered the largest number of requests. This strategy performed best in an environment where requests are evenly distributed amongst nodes. Finally, the last strategy is the combination of **learning-based strategy and best-neighbor**. It is identical to the learning-based algorithm, except that when no previous information is available, the request is forwarded based on the best-neighbor strategy. Even though it is the most expensive, experimental results showed that performance was unpredictable.

Filali et al.[15] also propose a P2P resource discovery mechanism for Grids, but with the goal of improving the following limitations of an existing system described in [23]: only one resource could be managed (CPU) without a precise description, resources were booked for an unlimited amount of time, and resource discovery was based mainly on flooding. Nodes are divided into two categories: Grant nodes, that offer resources that can be used, and Requester nodes, that search for and use resources. All nodes are also relay nodes and store grant messages in a cache that is periodically cleansed for expired messages.

Two types of transport mechanisms are used: flooding when no information is available, and the cache. Request messages are compared to the local cache for matching grant messages. If found, the node acts as a relay and propagates the request to the last node that transmitted the grant message; if not, the query is broadcast to all neighbors. Experimental results show this system to be more efficient than basic flooding and random walk, as is to be expected due to the usage of a cache: messages are forwarded to peers that are closer to the requested resources, until they are eventually found.

Liu et al. in [19] take a different approach and propose a system for resource discovery that mimics human behaviour in social networks, i.e. ask acquaintances for knowledge on a desired resource or service (e.g. a good mechanic). It exploits the small world phenomenon observed by Stanley Milgram, hypothesizing that everyone in the world can be reached through a short chain of social acquaintances.

Although unstructured networks are resilient in a dynamic environment, current search methods either require too much overhead or generate too much network traffic. To combat this, each node listens to requests and records successful ones in its knowledge index (basically a Least-Recently-Used cache). Nodes learn from previous requests making future searches more focused as interest groups are formed automatically based on previous search results, without extra overhead or explicit interest declaration. Resource searching is performed by first consulting the knowledge index for peers directly related to the search topic. If no peer was found, which is most likely in the beginning, then

the system looks for nodes in the knowledge index that share content in the same interest area (based on the Open Directory Categories[3]). If that also fails, then the system resorts to the random walk strategy.

2.1.1.2 Structured

Structured systems address the routing scalability problems originally faced by unstructured systems, by employing a rigid organization of its nodes. These systems are designated as Distributed Hash Tables (DHTs) and provide a mapping between an identifier and its content. Exact-match queries are routed efficiently due to the tight control over network topology and file locations. Range queries can still be supported with these types of systems [24], but one cannot say the same about non-exact queries because of the way routing is performed in relation to network's structure. Another disadvantage to using such a rigid structure is the required overhead to maintain it in a highly dynamic node population.

Chord[3] and Content Addressable Network (CAN)[5], although not resource discovery systems, provide a routing and location infrastructure that can be uses as a basis. Nodes are organized into rigid structures where file identifiers (aka. keys) are mapped onto node identifiers using a hash function. These two systems differ in the way they organize their nodes and in the functions they use to map keys onto nodes.

Chord[3] was the first structured Peer-to-Peer system, proposed by Stoica et al., where nodes are organized in a circle and packets can only be forwarded clockwise. Keys are calculated using a m-bit key space, and are mapped to the node whose identifier is bigger or equal to a key. For routing to be possible, each node is aware of its predecessor and successor in the ring. The most basic, and unefficient, routing prcess can be performed by forwarding a message to the successor until the right node is found. A more efficient technique consists of using a finger table with m entries that, for a node n, maintains a connection to the first peer on the circle that succeeds $(n + 2^{k-1}) \bmod 2^m$, for $1 \leq k \leq m$, as exemplified in Figure 2.2a. This lookup process emulates a binary search, thus requiring only $\mathcal{O}(\log N)$ messages and steps (Figure 2.2b).

CAN[5], on the other hand, takes a different appraoch and uses a virtual d-dimensional space to store (key, value) pairs. Each node is responsible for a zone, which is a segment of the coordinate space. This space is divided equally between all participating nodes. Therefore, peers only connect to nodes responsible for neighboring zones, i.e. each node has $\mathcal{O}(d)$ neighbors. Keys are mapped deterministically onto a point in the coordinate space, and the (key,value) pair is stored at the node responsible for the zone in which the

[3] http://dmoz.org

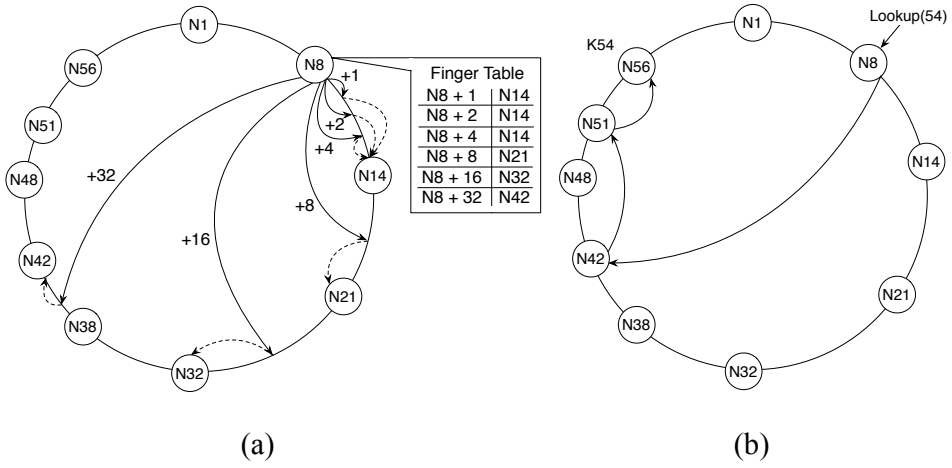

FIGURE 2.2: (a) Example finger table for node 8. (b) Routing of query for key 54 using the finger table to speedup lookup.

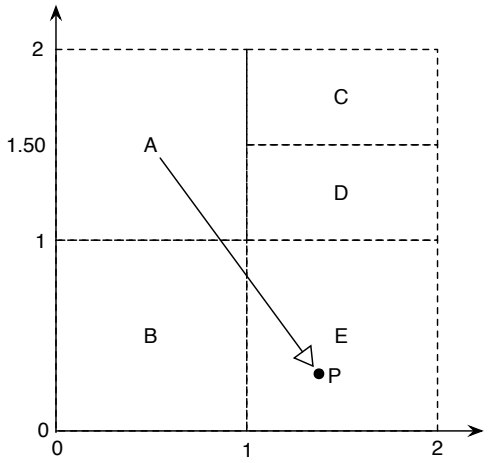

FIGURE 2.3: Example CAN coordinate space of [0,2]x[0,2]. Nodes B, C, and D are node A's neighbors. Query for key that maps to (1.3, 0.3), starting from node A, is routed first through B and then finally to E, which is responsible for that zone.

point falls under. To retrieve an entry, the same deterministic function has to be applied. If the resulting coordinate does not fall into a neighboring node's zone, the request is then routed from node-to-node until the node that is responsible for the target zone is reached. Intuitively, routing is performed by following a straight line through the Cartesian space from source to destination coordinates, as can be seen in Figure 2.3.

Because CAN only supports exact match queries, **Andrzejak and Xu**[24] propose an extension that allows range attribute queries for usage in a Grid environment (where resource queries typically include ranges). Depending on the resource attribute's type, the system will either use a standard DHT, for attribute values with a limited number of values, or the extended CAN when attributes have continuous values. Multi-attribute requests are also supported by consulting the appropriate DHTs and then integrating

the results. The extended CAN only uses a subset of Grid nodes, called Interval Keepers (IK), that are responsible for a sub-interval of the attribute's value. Each server in the Grid reports values to the IK with the corresponding interval.

The authors also propose three strategies for propagating range queries and methods to reduce communication overhead during attribute updates, which are frequent in a Grid environment. **Brute force flooding** first finds all the IKs that intersect a range query and then flood the request in a BFS-manner, which wastes effort by contacting nodes that aren't part of range query. **Controlled flooding** makes sure requests are only sent to the nodes that intersect the range query, but comes at the cost of less parallelization as nodes may receive multiple messages for the same query. **Directed controlled flooding** avoids duplication by using two propagation waves: the first only contacts IKs that have a "higher" interval than the current IK, the second then propagates queries to neighbors with a "lower" interval, which effectively reduces message duplication. These strategies were tested using simulations of synthetic and real-life workloads and results show they were effective in meeting the system's goals of scalability, availability, and communication-efficiency.

Moving away from a virtual d-dimensional coordinate space, **Schmidt et al.**[25] propose a system that supports multi-attribute queries in a single one-dimensional DHT by using a space filling curve which maps all possible dimensions onto one.

Attribute values are mapped onto nodes whose ID is generated by interleaving the binary representation of the attribute's values. For example, a resource containing three attributes with values (3,2,1) is represented in binary as (1**1**,1**0**,0**1**). The interleaving process is done by taking the first number from each attribute's binary representation (the most significant bit) and join them to construct the first part of the ID. The second part of the ID is generated by taking the least significant bit from each attribute's binary representation and also joining them. Thus, (3,2,1) will be mapped to the node whose $ID = 110\mathbf{101}$.

Range queries are constructed in the same way, except that it uses some "wild card" bits. For example, searching for a resource attribute with values (2, 1, 0-3), with binary representation (10, 01, 00 - 11), is represented as 10*01*. They are resolved like point queries, with the only difference being that when an undefined bit is found, the query is then propagated to more than one node. Requests are forwarded to nodes with an ID that has a larger common prefix with the query than the current node. Thus, an originating node's ID that starts with 0 (searching for 10*01*) will first propagate the query to any node in the form 1*****. Then, that node sends the query to any peer with ID in the form 10****. That node, in turn, forwards the query to two nodes: one with the ID 100*** and the other with ID 101*** and so on. But doing this means that

the more wild cards are present in a query, the more nodes are contacted, effectively reducing the performance of the system.

An interesting fact about this system is that there is no bottleneck at the lookup root node, which is common in tree-like structures, because any node whose ID's first bit is equal to the query's first bit can be used as a root node, i.e. there is no single root node.

Ratnasamy et al.[26] also support range queries over DHTs, but use a distributed data structure called a Prefix Hash Tree (PHT). The PHT is agnostic to the routing algorithm as it is an additional overlay on top of a DHT. This overlay is used because locality between ranges is not maintained with classic DHTs.

Data items are stored at the PHT node with the longest matching prefix between node label and the item being inserted. Each node has a maximum limit of data items it can store; once exceeded, it "splits" into two child vertices and the data items are partitioned between its children depending on their prefixes. Therefore, the system only starts with one root node. As data items are inserted, it starts growing as node recursively "split." Resources are stored in their own PHT for every attribute they contain, which means that all attributes are actually stored in the common DHT. The PHT structure is distributed across the DHT by hashing the labels of PHT vertices. This is done by using a uniform hash function with the attribute name, lower attribute value range, and higher attribute value range as parameters. For example, the PHT node responsible for attribute A from x to y is mapped to the DHT node whose $ID = hash(A, x, y)$.

Lookups are performed by recursively dividing the attribute value range in half, until the smallest range that contains the whole query range is found. Then, a normal DHT lookup is used to find the node responsible for that range. Once located, that node then broadcasts a message to all children in its subtree to retrieve the desired items. Notice that the root node is not a bottleneck as access to individual nodes does not need to traverse the root node. Multi-attribute queries are simply resolved in parallel, consulting different PHTs depending on the attribute in question, but results in as many messages as there are attributes.

Marzolla et al. in [27] describe another system that organizes nodes into a tree-structured overlay, but without an underlying DHT like Ratnasamy et al. Instead, routing indexes are used to locate resources in this Grid discovery system. Each node manages its own resource information and also keeps a compact representation of resources from children nodes in bitmap indexes.

Each resource attribute is stored in its own bitmap and they are used to route queries to a node that might be able to satisfy the request. The attribute value space is divided into k sub intervals and are stored in a k-sized bitmap. All entries are set to 0, except for

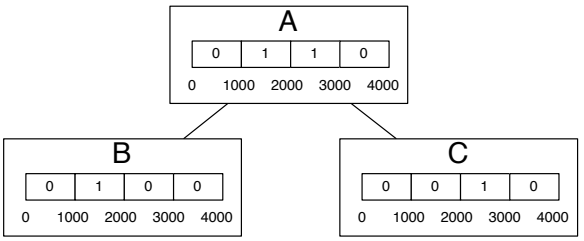

FIGURE 2.4: Example bitmap indexes, as used in [27], to represent CPU Speed (in MHz) in nodes B and C, along with aggregated information (bitwise `OR`) in parent node A.

the one corresponding to the sub interval that contains the actual value of the attribute in question (nodes B and C in Figure 2.4). To obtain a compact representation of resources for a subtree, the bitwise `OR` operator is applied on all bitmaps belonging to the same attribute, local to each child node (node A in Figure 2.4). Multi-attribute queries are split into sub-queries for each attribute and are then forwarded to the nodes whose bitmap indexes satisfy all sub-queries.

To handle the dynamic nature of a Grid environment, bitmap indexes are recalculated periodically. Neighbors are only notified if the changes to resource consumption modify the index. Simulation results show that this system scales well because it avoids flooding by routing messages to a small number of nodes, and the updating method involves a constant number of peers, regardless of the network size.

2.1.1.3 Hybrid

Finally, hybrid systems try to address the disadvantages of both structured and unstructured systems, while still trying to retain their benefits. Systems like Pastry[4] and Kademlia[6] will be considered hybrid systems, even though they tend more towards structured systems than unstructured, because their similar structure is less "rigid" compared to that of Chord and CAN (from Section 2.1.1.2). In Chord and CAN, all neighboring connections are strictly defined and only one node contains the value for a key; as opposed to Pastry and Kademlia where any peer belonging to a defined subspace can act as a contact for the values in that subspace. We shall also consider P2P systems that employ super-peers or clustering as being hybrid, for the nodes that are chosen as the leader of a group form another overlay between themselves to increase routing performance. In contrast to Pastry and Kademlia, these systems tend more towards unstructured P2P systems than structured.

Pastry[4] is a scalable, distributed object location and routing infrastructure, allowing the creation of various types of peer-to-peer Internet applications. Each peer has a

unique 128-bit identifier (nodeId), indicating its position in the circular ID space. It is randomly assigned when a peer joins the network, thus adjacent nodes, with high probability, are diverse in terms of geography, ownership, jurisdiction, etc. The objective of this system is not only to efficiently route messages to nodes, but also to take into account network locality by using a proximity metric (e.g. IP routing hops or geographic distance).

Messages can be routed in a tree-like fashion or, if that fails, using a ring approach (similar to Chord). To support these routing procedures, each node needs to maintain some state: a routing table and a leaf set. It also keeps a neighbor set which is used to maintain locality properties. The routing table is used by the tree-routing method. Each level n in the routing table refers to a node that shares a n digit prefix with the local node, but where the $n + 1th$ digit is different. The leaf set is used to perform ring like searching. One half of the set contains the nodes whose IDs are smaller and numerically closest to the current nodeId, while the other half contains the bigger and numerically closer node IDs. As long as no more than half of the nodes in the leaf set fail simultaneously, Pastry will continue to function. The neighbor set contains the IP addresses of the nodes that are closest, with regards to the proximity metric (e.g. round-trip time), to the local node.

Routing is performed by first checking if the key falls in range of the leaf set. If not, then the routing table is used to find the nodeId that shares a common prefix with the key by at least one more digit than the local node. If that fails, either the entry is empty or the node died, then the message is forwarded using a ring approach and is sent to a node (from all tables) whose prefix with the key is just as long as the current node, but is numerically closer to the key. This routing procedure always converges because with each step the message is forwarded to a node that is numerically closer to the key than the local node. To increase robustness and minimize the distance a message travels, replicas can be stored at a set of k-nodes that are numerically closest to the key.

Kademlia[6] is another routing and lookup infrastructure, like Pastry, but differs in the way the distance between keys are calculated: using the XOR metric ($distance = key \oplus nodeId$). Most of Kademlia's benefits are from using this metric due to its symmetry. This means that nodes can use information from lookups to update the routing tables, unlike Chord nodes which cannot learn useful routing information from the queries they receive. The asymmetry of the metric used by Chord also makes routing tables rigid, needing a precise node in an interval within the ID space. Kademlia on the other hand can send a query to any node in the interval, allowing different routes to be selected depending on, for example, latency.

Keys are stored at the k closest nodes to that key, which are assigned in a 160-bit space. Routing is performed using special lists called k-buckets are used, where k is a system wide number (e.g. 20). Buckets store information about nodes situated at a particular range from itself: from 2^i to 2^{i+1} for $0 < i < 160$. When a node receives any type of message, it updates the appropriate bucket. This process is optimized for keeping the longest living nodes in the routing table. k-buckets also provide some resistance to certain DoS attacks: the network cannot be flooded by new nodes.

To locate a node, a single routing algorithm is used from start to finish. Routing uses the same XOR metric to determine the n closest nodes to the desired key. This lookup process is recursive, as it consists of picking α nodes closest to the desired key and asking them (in parallel) to return the n closest nodes they know about. Once results are obtained, the process starts again and selects another α nodes that are even closer to the desired key than in the first step. This process continues until the n best nodes have been found. α is a system wide concurrency parameter, such as 3. If $\alpha = 1$, then message cost and latency of failure detection resemble that of Chord. This parameter can be configured and lets users trade bandwidth for better latency and fault recovery. The lookup process stops immediately when a value is found.

The (key,value) pair can additionally be cached at the nodes closest to the key that were queried but did not contain the pair. This caching method exploits the unidirectionality of the XOR metric, as all lookups for the same key converge along the same path regardless of the originating node. Therefore, future queries will likely hit the caches entries before querying the closest node.

XenoSearch[28] uses the Pastry location and routing infrastructure as its basis, and adds support for multi-attribute and range queries while being only a factor of 3-5 slower. A Pastry ring is constructed separately for each attribute. Range queries are made possible by exploiting the fact that the information is conceptually stored in a tree, where the leaves are XenoServers and the interior nodes are aggregation points (APs). APs summarize the range of values of the nodes below them in the tree and are identified by a key which is stored in the same key space as the attributes.

Identifier generation is performed by creating keys that are prefixes of child node keys. For instance, the AP directly above 10233102 is 10233101, then 10233110, then 10233100, and so on. This way, just by knowing the key of an AP we can determine the range of values of leaf-nodes, which gives us the range of values of the leaf-node XenoServer attributes. The XenoServer node closest to the AP in the key space is responsible for managing the information related to that AP.

Multi-attribute queries are resolved by dividing the query into sub-queries, one per attribute, and performing a range search in the corresponding Pastry ring. Results are then intersected and the Client is given a set of possible XenoServers that might be able to satisfy the query's requirements. The Client still needs to directly contact the XenoServer with said resources to confirm they are still available. This step is necessary because information in the system is only periodically updated, therefore the results obtained from a query may not be up-to-date.

Mastroianni et al.[29] propose a resource discovery system in a Grid environment that takes a less structured approach than the previously mentioned systems and uses a super-peer model. This model tries to strike a balance between the inefficiency and scalability problems of centralized search, and the load balancing, autonomy, and fault tolerance features of distributed search.

Super-peer nodes act as a server for regular peers. The former tend to be nodes with higher capacity, while the latter are usually regular or low capacity nodes. Super-peers are interconnected and form a P2P overlay. This model exploits the natural tendency of large-scale grids forming into interconnected clusters of computers, each under their own administrative domain - called Virtual Organizations (VOs). Each VO has one or more nodes that act as super-peers for the other nodes in the organization, and are responsible for maintaining metadata about the resources of connected clients, as well as communicating with other VOs.

Regular nodes searching for resources send a query to a local super-peer, which, in turn, scans its local metadata for a match. If found, a queryHit is generated and sent directly back to the requesting node; if not, the query is forwarded to a limited number of neighbors. The neighbor selection process uses the best-neighbor technique, i.e. nodes that have answered the most queries are preferred. Whenever a matching resource is found, a queryHit is forwarded along the same path back to the requesting node. Additionally, a notification message is sent by the remote super-peer to the node that has the requested resources. The authors also propose a number of techniques to decrease network load, reduce response time, and increase the probability of success. Such technique are, but not limited to: limiting the Time-to-Live of queries, using an additional field in a query to record the path traveled, and the caching of queries so as to not process duplicate requests.

2.1.2 Grids

Grid computing is defined by the combination of computer resources in order to perform a specific task. These resources are usually distributed geographically and fall under

different administrative domains. The tasks that are usually performed either require lots of CPU processing power, or need to process large amounts of data, which is common with scientific, technical, or business problems. The divide-and-conquer strategy is used where large tasks are divided into smaller ones and distributed across many computers, potentially thousands. Grid computing can be done in a small LAN for, say, a university, or it can function under a larger network comprised of several smaller interconnected networks that belong to a different institution, corporation, or university. The computers that provide the resources, either a normal PC or even a super-computer, are sometimes referred to as metacomputers, while a cluster of these metacomputers are usually referred to as Virtual Organizations (VOs).

Condor[30] is a specialized workload management system for compute-intensive jobs that can be used to build Grid-style computing environments that cross administrative boundaries. Resource discovery is performed using the ClassAd mechanism[31], which is responsible for matching resource requests (jobs) with resource offers (machines). Agents and resources advertise their characteristics and requirements in classified advertisements (ClassAds), which declare job or machine requirements and preferences. These ClassAds are semi-structured data models that consist of uniquely named expressions called attributes. Each attribute has a name and a corresponding value. Attribute values range from simple types (e.g. integers, floats, strings, etc.) to richer types (e.g. records, sets, etc.) and conditional operators. As requirements and preferences can be described in powerful expressions, Condor is able to adapt to nearly any desired policy.

Job and machine advertisements are sent to a dedicated matchmaker server, making resource discovery in this system centralized. It is responsible for scanning known ClassAds and creating pairs between jobs and machines that satisfy each others constraints. When new pairs are discovered, the matchmaker server informs both parties of the match, thus leaving it up to the agent to directly contact and claim the desired resource. The separation of the matching and claiming phases brings greater flexibility to the system, allowing the resource, for example, to independently authenticate and authorize the match, or to verify that match constraints are still satisfied with respect to current conditions.

Legion[32] is an object-oriented metacomputing environment, intended to connect many thousands, potentially millions, of hosts ranging from PCs to massively parallel super computers. Machine attributes are represented in Host Objects, acting as an arbiter for the machine's capabilities, while jobs are represented as Objects with a set of requirements. Legion is similar to Condor in the way it uses a centralized component to perform resource discovery. The only difference is that Legion's "matchmaker" equivalent in Condor is split up into three sub-components: the Collection, Scheduler, and Enactor.

The Collection is populated with information describing the resources, therefore acting as a repository for information about the state of the resources comprising the system. This population of information can be done in two ways: using the pull-model, where the Collection component queries hosts to determine their current state, and the push-model, where the Host Objects periodically deposit their information into its known Collection(s). The Scheduler then queries the Collection and, based on the results, computes a mapping of objects to resources. This mapping is passed along to the Enactor, which attempts to reserve the resources named in the mapping on Host Objects. Once reserved, the Enactor consults with the Scheduler to either confirm or cancel the schedule, and in case of an affirmative response, tries to instantiate the resource objects.

Another infrastructure that allows the construction of Grid systems is **Globus**[33] and can use MDS-2[34] (Meta Directory Service) as a resource discovery mechanism. It makes use of two fundamental components: highly distributed information providers and specialized aggregate directory services. Information providers allow access to information about available resources and is neutral to Virtual Organizations (VOs). Aggregate directories provide specialized view of resources within a VO. The information provider speaks two basic protocols: GRid Information Protocol (GRIP) to access information about entities, and GRid Registration Protocol (GRRP) to notify aggregate directories of resource availability.

The two aforementioned protocols are the building blocks on which this architecture is built on. The aggregate directory also uses the GRIP and GRRP protocols to obtain information from a set of information providers and to respond to queries about those entities. Each VO has its own aggregate directory, which is vital to the scalability of the system. This way, queries for resources from a specific VO can be directed to the corresponding aggregate directory service. Thus, the scope within which search operations take place is limited, without resorting to searches that do not scale well to large numbers of distributed information providers.

Aggregate directories organization can be quite flexible, but the most convenient structure is a hierarchical one, as it mirrors a typical decomposition of VO administration with multiple site administrators coordinating with the VO service administrator. This organization implies that each aggregate directory acts as an information provider for all the resources available beneath it using GRIP, while using GRRP to register with higher-level directories to construct the hierarchy.

2.1.3 Cycle Sharing

Volunteer computing or public-resource computing consists of computer owners from around the globe donating their computing resources, such as CPU cycles and storage, to one or more projects they believe in. Most cycle sharing systems have the same basic structure: a client program that runs on the volunteer's computer which periodically contacts the project's servers to request jobs or report back results. The project servers normally give credit to users when a job is completed successfully, which is then used to measure how much work the user's computer has contributed to the project. There are a number of problems that arise from using volunteered computers, such as their heterogeneity, sporadic availability, as well as not interfering with their performance during regular use. That is why the client software normally only contacts the project's servers when the computer has been idle for some time.

Another problem that these systems must resolve has to do with result correctness, as there is no volunteer accountability because they are essentially anonymous. Other factors that can affect the correctness of results are computer malfunctions and the forging of results in order to gain more credit or sabotage the project. To deal with this, the servers need to send the same job to more than one client and compare all the results. Only if they sufficiently agree, is credit given to the users that performed the work.

Berkely Open Infrastructure for Network Computing[35] (BOINC) is a software system that allows scientists to easily create public-resource computing projects. It supports diverse applications, including ones that have large storage or communication requirements. The main objective of BOINC can be summarized as giving scientists access to the enormous processing power of personal computers around the world.

A simplified overview of how the system functions is as follows. A user that wishes to volunteer their PC for a cause, such as Folding@Home for example, will go to the project's website and download the BOINC Client. The user is then able to configure the resource consumption, so as to not disturb during working hours. When the BOINC Client runs in the designated times, it will contact the project's central server, which is responsible for the coordination of various clients by sending them jobs and collecting the results.

Saying that BOINC has a resource discovery mechanism is a bit of a stretch. What it does provide is a flexible framework that allows the distribution of application executables over a number of platforms. The project administrator can specify which applications are needed in order to do useful work for the project. Typically, the BOINC Client just downloads the pre-compiled binaries from the central server and executes

them along with the associated work unit. But there are some users that do not want to run these pre-compiled binaries: security reasons, because there are no pre-compiled binaries for the user's platform, or others. For this case, BOINC provides an anonymous platform mechanism which allows the user to compile the required applications himself, and specify them in a configuration file. Then, when the BOINC client communicates with the project server, it indicates its platform as anonymous and supplies a list of available application versions. The server, in turn, just sends the work units to be performed, without any pre-compiled binaries.

In **Cluster Computing on the Fly**[36] (CCOF), the authors take a different approach to the centralized system BOINC. Instead, they performed a comprehensive study of resource sharing methods in a highly dynamic P2P environment for locating idle cycles to be consumed by workpile[4] applications. Workpile applications consume huge amounts of processing power and are embarrassingly parallel, i.e. nodes performing computations do not need to communicate with each other to accomplish the task. Another difference w.r.t. BOINC is that CCOF is more general as users can be donors, or consumers of idle cycles, or even both. Idle cycle resource information is described using a profile-based model which is generated automatically by monitoring CPU usage patterns of the user's PC.

The authors evaluated four different search methods. One of them is **Random Walk** which we already discussed in Section 2.1.1.1. In the **Expanding Ring** search clients send a query for cycles to their direct neighbors. The neighbor compares the request against its profile and turns the request down if it cannot be satisfied. If the client determines there is not enough peers to perform the computation, it resends the query to nodes that are two hops away. This process continues until the computation can proceed or until the search depth limit is exceeded. The third method is **Advertisement-based** and has nodes send their profile to a limited number of neighbors to be cached when they join the network. Lists of available candidates are selected based on cached profiles, but a client still needs to contact the host directly to determine if the cycles are still available. If not, it just tries another peer in the list. Simulation results show that this method incurred a high message passing overhead. Finally, the last method, **Rendezvous Point** search groups dynamically select peers as Rendezvous Points to enable efficient query and information gathering. When a node joins the system, they advertise their profiles to nearby Rendezvous Points. Searching is performed by sending queries to the nearest Rendezvous Point(s). It is important that these special nodes are selected so that the system is balanced and, therefore, a sufficient number of Rendezvous Points exist within a short distance to every peer, which is another problem unto its own. This technique

[4] Also known as bag-of-tasks

performed better than the rest under both light and heavy workloads, with a consistently low message passing overhead.

2.2 Service Discovery Protocols

Service discovery[12] aims to provide a mechanism that enables, without any configuration, the automatic detection of services provided by devices present in a computer network. This computer network can be a small, home Local Area Network, or a large, enterprise-scale network at a corporation or university. The types of services offered by devices in those networks can range from simple tasks, such as printing or the usage of a projector to display a presentation, to more complex tasks, like facial recognition or video encoding.

One of the first well known service discovery systems is the **Service Location Protocol** (SLP)[37]. SLP can operate in two different modes. The first is not centralized and uses multicast for locating services, but is unusable in large networks due to flooding. The second mode is centralized and uses directory agents to handle a large number of queries. Nodes can assume three different roles: service agents (SAs) which advertise and provide services, directory agents (DAs) that collect and index service advertisements, and user agents (UAs) which query DAs for services and utilizes them through SAs. Before UAs and SAs use the directory agents, they need to locate it first. This can be done passively by detecting multicast advertisements, or actively by sending SLP requests.When a DA is present, UAs communicate with them via unicast, otherwise multicast is used to query SAs for services.

Jini, by Sun Microsystems, is also a centralized service discovery system that is based on the Java Virtual Machine (JVM) that is platform and protocol independent. The system is built on top of the Java Remote Method Invocation (Java-RMI) system to handle interactions between nodes, which enables the system to adapt to network changes and not require any configuration. Jini's architecture is similar to that of SLP by using a lookup server that functions like a directory agent, collecting advertisements and searching on the behalf of clients. Service discovery is performed by first detecting the lookup server in the network. After that, the Jini agents can then send queries to search for, or publish, service information. Unlike SLP, the lookup service component is not optional for the system to function and can be located using multicast discovery messages.

Communication between service providers and service users is done via special Java objects, called proxy objects, that are stored in the lookup server's directory. Jini is

capable of working in any type of network, requiring only the presence of a JVM, which can be considered a limitation. Another limitation is that lookup servers are single points of failures due to Jini's exclusive use of a centralized architecture.

Goering et al.[38] go a different way and propose a decentralized service discovery protocol for local ad-hoc networks based on the use of Attenuated Bloom Filters. Bloom Filters, discussed in more detail in Section 2.3.4, is a hash coding technique that provides an efficient way to test the membership of a text string in a given set of strings, while using as little storage space as possible. The only drawback is that there is a small chance a false positive may occur, i.e. the system claims the string is probably in the set when it really is not. This is not a problem if the chance of it occurring is small enough. In a worst case scenario, an application will try to contact a resource that does not exist, but that is not a problem because the application will find out and will just try to contact another peer.

The authors use Attenuated Bloom Filters which provide a method to locate objects, giving preference to objects located nearby. It is simply an array of Bloom Filters of depth d, where each row represents objects at different distances which, in this case, is in term of hops. Each node has an Attenuated Bloom Filter for each of its neighbors. When a node receives a query, it will consult them to find a neighbor that is likely in the direction the requested service can be found. The first level of the Attenuated Bloom Filter corresponds to the services that are one hop away, the second to services two hops away, and so forth. Therefore, the larger the distance from the node, the more services will be contained in the corresponding Attenuated Bloom Filter which will increase chance of false positives. In this case, one can think of Bloom Filters as a way to summarize the information of available services, where more accurate information will be available closer to the destination. That is why queries are forwarded to a neighbor where the resource can most likely be found.

Query forwarding can be performed three different ways. It can be done in parallel, where the query is sent in each direction a match is found, although it consumes a lot of bandwidth. It can be done in a sequential manner, where the query is propagated only to the direction with the best/first match and traces back in case of failure, but tends to be slow. Finally, a hybrid approach can be used which combines the best of both worlds: the query is forwarded in parallel to a limited number of best matches and allow them to trace back when no match is found in order to try another set of best matches. This system does have a big limitation: only the services located up to d-hops away can be discovered by using an Attenuated Bloom Filter of depth d, and no further.

In [39], **Lv and Cao** present a service discovery protocol that addresses the limitations of the previously described system by Goering et al. This system also uses Attenuated

Bloom Filters to find local services efficiently, but when none can be found within d-hops, another service discovery method is used (originally proposed by Sailhan and Issarny in [40] under Global Service Discovery). This method creates a bridge between nodes that are further than d-hops away to act as gateways for finding services further away.

Service discovery is performed in the following way. Each node receives the Attenuated Bloom Filters from its neighbors and caches them, so there are as many Attenuated Bloom Filters as there are neighbors. When a query is received, the node first checks its Attenuated Bloom Filter to see if the service exists. If there is a match, it sends a response to the originating node; if not, the node will check the cached Bloom Filters of its neighbors. If the node has several neighbors, the node that is checked first is the one with the smallest network branch. If the first does not contain the desired service, it will traceback and query the second smallest branch. If there is still no match, then the query is sent to a node d-hops away, where the discovery process is repeated at that node. This way, the system is still able to discover services that may be located more than d-hops away. The only problem is that the authors do not mention how to handle false-positives sometimes given by the Bloom Filters, although one can assume that the system will simply continue the discovery process when no match was found.

Czerwinski et al. in the Secure Service Discovery Service[41] system also use Bloom Filters, but in a hierarchically organized set of servers. This system is intended to be used by a large number of clients (wide-area network) that are able to compose complex queries for locating services in a secure manner. To address scalability issues, the servers that collect service information are organized in a hierarchy where a node will create a "child" server and assign a portion of the network if it is overloaded.

Servers in the upper tiers of the hierarchy are not overloaded with update or query traffic as updates are localized and service descriptions are only stored at the servers where they are being periodically refreshed.

Service descriptions (and client queries) are defined by tags in an XML file. A subset of these tags are then inserted into the Bloom Filter to provide a summary of the various services a server contains. If a node has any children, then associated with the children's are their corresponding Bloom Filters. The node's and the children's Bloom Filters are then aggregated (by `OR`ing them together) and passed up to the parent.

Routing is performed as follows. If a query is going up the hierarchy, then each node checks to see if there is a local hit or a hit in any one of its children's Bloom Filters; if not, then it passes the query upward. When a query is going down, the process is the same, except for the direction the query travels.

By using Bloom Filters, the tens or hundreds of hashes describing services do not consume large amounts, of space. The only problem is updating the Bloom Filter when services die. The authors propose either periodically rebuilding the Bloom Filters, or using per-bit counters (Counting Bloom Filters in Section 2.3.4).

2.3 Efficient Data Representation

This section deals with methods and data structures that help reduce data storage and network transmission costs. This is important in a Peer-to-Peer system because nodes not only have to store information about other neighboring nodes, but also have to transmit data between themselves. Therefore, it is vital that storage and transmission overhead is reduced as much as possible in order to increase the scalability of the system, for if message size is reduced the network will not become saturated as easily.

2.3.1 Compression

Compression techniques[42] can be used to reduce storage and transmission costs by reducing the size of largely repetitive data. They can be divided into two categories. The first is dictionary-based, such as **LZW**[43]. It starts by initializing a dictionary to all the symbols of the alphabet, which will then be used to encode sequences of 8-bit symbols as fixed-length 12-bit codes. Thus, the entries from 0 to 255 represent 1-character sequences consisting of the alphabet. Entries 256 through 4095 are then created in the dictionary for sequences encountered in the data as it is being encoded. At each step of the encoding process, input symbols are gathered into sequences until it finds a combination not yet present in the dictionary if the next character were to be read. It then outputs the code for the previously known sequence present in the dictionary, without that character, and then adds the new sequence, this time with the newly read character, to the dictionary. The decoding process proceeds in the same manner, but instead of operating on normal text symbols, it works on the codes that were emitted by the encoder. This is possible due to the fact that the manner the codes are added to the dictionary is determined by the actual data.

Huffman coding[44] is part of the statistical-based methods, which form the second category. It uses a specific method for choosing the representation for each symbol in the text to be compressed, which results in a prefix code. This prefix code is a string of bits that represent some symbol, and whose prefix is never the same as any other bit string that represents another symbol. The prefix code emitted is shorter for the most common characters and larger for the less common symbols. This is done by building a

binary tree where each node has an associated weight, and the sum of a node's sibling's weights results in the parent node's weight. The prefix code is obtained by traversing the tree until the desired symbol is reached, resulting in a binary prefix. The most common symbols will have a bigger weight than the less common ones, thus the prefix codes for common characters will be short as the depth the algorithm needs to traverse into the tree is shorter compared to the less popular symbols.

2.3.2 Chunks and Hashing

Transferring large files over a network can consume a lot of time and bandwidth. Instead of resorting to compression, some systems exploit the similarity between different versions of the same file, as it is not common that a file changes completely between versions, or even between different files in order to reduce the amount of data to be transmitted over the network. In general, this is done by dividing a file into fragments and sending only those fragments that have been modified since the last version stored at the destination node.

The `diff`[45] Unix utility uses delta encoding techniques to calculate the difference between two files. The program's output can then be transfered, say via e-mail, and later on be used by the `patch` utility to transform one file into another. This technique is also used by the version control system **CVS**[46] to bring a user's working directory up-to-date.

Rsync[47] is yet another system that makes use of delta encoding to synchronize files and directories from one location to another over the network while minimizing data transfer by exploiting commonality between files. An example transfer using a simplified version of rsync could proceed as follows. First, the recipient breaks a previous version of a file into non-overlapping, contiguous, fixed-sized blocks. It then calculates and transmits the hashes for those blocks. Once the sender receives those hashes, it computes the hashes of all overlapping blocks of the file with the same name. If any of those hashes match the ones sent by the recipient, then those sections are not sent over the network, instead, the recipient is notified of the location to the data in the previous version of the local file.

Rather than only exploiting the similarities between different version of the same file, the **Low-Bandwidth Network File System** (LBFS)[48] saves even more bandwidth by also exploiting similarities between different files (e.g. auto-save files, sometimes used by text editors, that have different names but whose content is very much the same). It avoids sending duplicated data when the same data can be found in the client's cache. To exploit similarities between different versions and files, the LBFS server divides the files

into chunks and indexes them by calculating their hash value. The client also maintains a database of chunks to help identify duplicated data. LBFS detects which chunks are already present on client-side, thus avoiding the transmission of redundant data. The system relies of the extremely low probability of collision of the SHA-1 hash function and assumes chunks with the same hash value are indeed the same chunk.

Apart from also considering similarities between files, LBFS differs yet again from Rsync with regards to file division: LBFS divides the files into chunks based on their contents rather than on position within a file, by using a technique called Rabin fingerprints. This technique creates a type of insulation around chunks, as any modifications to the content in a block will only affect that chunk and not the boundaries of the remaining ones. Therefore, as chunk boundary positions generally stay the same, except for the places where the content has changes, the system is more intelligent as to which chunks really have differed and need to be sent to the client, and which the client already has. The same cannot be said for systems that rely on boundaries based on position, for any alteration in the beginning of the file may impact the boundaries in the rest of the file, resulting in many new chunks to be sent over the network, even though the actual content is mostly the same.

2.3.3 Erasure Codes

Erasure codes permit the transformation of a message of k symbols into a larger message with n symbols, such that the message can be recovered from a subset of the n symbols. Therefore, they can be used to correct data, up to a certain point, that has been corrupted during its transmission. Erasure codes can also be used to tolerate failures[49], as is common in storage Peer-to-Peer applications, data grids, and so on. In general, by taking n data devices and encoding them in m additional data devices, the system will be able to tolerate up to m failures.

The Reperasure[50] system uses erasure codes to ensure data availability and to speed up client data access for a P2P storage system. The authors only consider P2P systems that guarantee object retrieval, such as DHTs, and assume that nodes belong to a well-defined administrative domain, i.e. there are no volunteered computers from outsiders. Traditionally, replication can be performed by generating multiple full replicas and distributing them over failure-independent and geographically dispersed nodes. But in this system, the authors consider there to be, logically, one single copy. This copy is then divided into many blocks, known as data blocks, and is distributed across the nodes in the underlying DHT. The check blocks, which are the additional blocks that were encoded using an erasure code, are also stored in the DHT along with the data

blocks. The storage space needed to host all these blocks is much smaller than having to distribute and store full replicas. An additional benefit can be achieved if access to a sufficient number of blocks is done in parallel, which will increase performance and make more efficient use of the network and storage bandwidth. The novelty of this system is that we can logically consider the DHT as a super-reliable disk with very high I/O bandwidth.

2.3.4 Bloom Filters

Bloom Filters[51] are a probabilistic data structure capable of storing a list of items to conduct membership tests with very little storage space. Because of this, not only do they reduce storage overhead, but they can also be transferred over a network without incurring too much transmission overhead. This comes at the price of a small false positive rate (items not in the set have a small constant probability of being listed as in the set), but no false negatives are possible (items that were never in the set will not mistakingly be listed as such). Bloom Filters have been applied in a variety of systems[52], such as dictionaries, databases, and network applications.

A **Bloom Filter** representing a set $S = \{x_1, x_2, ..., x_n\}$ of n elements is stored in an array of m bits all initially set to 0. It must also use k different hash functions, each of which map some element to one position in the m bit array. Because Bloom Filters are implemented as bit arrays, the union of two sets can be computed by performing the `OR` operation between the two, while their approximate intersections can be computing using the `AND` operation. Insertion is performed by passing the element through each of the k different hash functions and setting the resulting position in the m bit array to one. To test whether an element is in the set or not, it has to be passed through all hash functions and if all the resulting positions in the array are set to one, then the element hash a high probability of being in the set. If any position has the value zero, then we know for definite that it is not in the set (no false negatives). The small false positive rate arises from the fact that when querying for an element that is not in the set, some hash functions may result in positions that were already used (have the value one) for a previously inserted item. Therefore, the more elements are inserted into the Bloom Filter, the higher the chance of a query resulting in a false positive. Another shortcoming is the inability to remove an element from the Bloom Filter, as simply setting the positions given by the k hash functions to zero have the side effect of removing other elements as well.

The inability of removing entries from a standard Bloom Filter can be solved by using a **counting Bloom Filter**[53]. The way it works is, instead of using a bit array

to represent the Bloom Filter, it uses a small counter. When an element is inserted, the counters at the positions given by the k hash functions are incremented; deletion is supported by decrementing the corresponding counters. In order to avoid counter overflow, a large enough counter needs to be chosen. One possible solution is to leave the counter at its maximum value when it overflows. But, one needs to take care because, later on, it may cause a false negative if the counter reaches 0 when it should be non-zero.

In [54], Mitzenmacher shows that using a larger, but sparser, bloom filter can have the same false positive rate with a smaller number of transmitted bits. Alternatively, the transmission of the same number of bits can be used to improve the false positive rate, or even another suitable tradeoff between the two. Therefore, **compressed Bloom Filters** can be used to reduce the number of bits to broadcast, the false positive rate, and/or the computation per lookup. As mentioned in [54], counting Bloom Filters can also benefit from compression.

Almeida et al. [55] proposed another variant of Bloom Filters: rather than needing to calculate the ideal size of a Bloom Filter to have a certain false positive rate which cannot increase in size as more elements are inserted, **scalable Bloom Filters** can be used to dynamically adapt to the number of stored items, while retaining a minimum false positive rate. This is achieved by using a sequence of standard Bloom Filters, each with increasing capacity and a tighter false positive rate. Therefore, one only needs to determine the desired minimum false positive probability regardless of the number of elements to be inserted. This also avoids the waste of space as one does not need to be conservative with regards to the size of the Bloom Filter because scalable Bloom Filters are automatically adjusted.

Attenuated Bloom Filters (partially discussed in Section 2.2) were proposed in [56] to optimize location performance, especially for objects that are located near the searching node. It uses an array of Bloom Filters with depth d, where each row i, for $1 \leq i \leq d$, corresponds to the information stored at nodes i hops away. As the depth increases the more information will be stored in that Bloom Filter row, making the respective filter more attenuated and resulting in a higher probability of false positives. Therefore, information closest to the node is more accurate, and becomes less so as the distance between nodes increases. The major advantage of this technique is that it permits us to efficiently locate objects, with a certain false positive rate, up to d hops away, using little storage space, as Bloom Filters themselves are space efficient. The disadvantage is that it *only* lets us search information about nodes up to d hops away.

2.4 Concluding Remarks

In this section we discussed the state of the art of Peer-to-Peer (Table 2.1), Grid, and Cycle Sharing systems (Table 2.2) that perform resource discovery. We also analyzed relevant service discovery protocols (Table 2.3) and various forms to represent data in an efficient manner (Table 2.4).

Note that many of the systems discussed in the P2P Resource Discovery section are related to resource discovery in Grid environments. This reinforces the idea presented in the Background of this work that as Grid systems grow in size they will tend toward P2P systems in order to support a larger and more transient node population. To add to this argument is the fact that Cycle Sharing systems, which can be considered a subset of Grid computing where the only resource that matters is CPU cycles, also utilize P2P technology, enabling them to harness the power of many volunteered computers connected to the Internet. As the overall objective of the GINGER (a.k.a. GiGi) project[11] is to create a "grid-for-the-masses" and bring Grid computing to home users connected to the Internet, it only makes sense for us to create a P2P resource discovery mechanism to be able to support a vast amount of users.

Because of GiGi's usage scenarios, not only does the discovery mechanism have to support the localization (discovery) of physical resources, but also of services, applications, and libraries installed in each user's computers. Each of the systems presented in this section handled these problems in isolation: systems in Section 2.1 only handled the discovery of physical computer resources and files, while systems in Section 2.2 only deal with the discovery of services. None of them attempted to aggregate all that information into one system to allow the discovery of various types of resources. This is precisely what the architecture described in Chapter 3 does.

For any discovery mechanism to work, we need to be able to store and transmit resource information. That is why we assessed various forms to efficiently represent data. **Compression** provides us with a way to reduce the size of data at the cost of CPU usage. As compression techniques yield higher compression rates with data that has a lot of repetition, we will not gain any advantage because there is very little redundant data when storing resource information. Another disadvantage would be the constant compressing and decompressing of information when receiving and sending queries, which are already small in size. The small query size is also a reason that **Chunks and Hashing** techniques are not really applicable here, as the major advantage they bring is reducing the amount of data needed to transfer large files by exploiting cross-file similarities. **Erasure codes** can be used as forward error correction codes, which permit the reconstruction of the original message using a subset of encoded symbols. This same

technique can be used to provide replication of files without creating full-replicas and thus reducing the required storage space. None of these usage cases are applicable to the discovery of resources where queries are small and are almost always different.

Finally, **Bloom Filters**, the last technique that was assessed, are highly applicable for what we want to do. They allow us to perform membership tests in an efficient manner, while requiring very little storage space. This does come at a price though: the possibility of false positives occurring. But, as long as it can be mitigated, Bloom Filters can help improve the efficiency of the system, in terms of performance, required storage space, and size of transmitted data. Because we are able to mitigate the occurrence of a false positive by requiring an additional hop, we find that Bloom Filters will help us accomplish our goals of efficiency and scalability.

System	Centralization	Type	Routing	Search Type
Napster	Centralized	Unstructured	Centralized	Knowledge Index
Gnutella	Decentralized	Unstructured	Flooding	Uninformed
Freenet	Decentralized	Unstructured	Flooding	Informed
Iamnitchi et al.	Decentralized	Unstructured	Flooding	Informed + Uninformed
Filali et al.	Decentralized	Unstructured	Flooding	Uninformed + Cache
Liu et al.	Decentralized	Unstructured	Flooding	Knowledge Index + Uninformed
Chord	Decentralized	Structured	DHT	Exact-match
CAN	Decentralized	Structured	DHT	Exact-match
Andrzejak and Xu	Decentralized	Structured	DHT	Exact-match + Range
Schmidt et al.	Decentralized	Structured	DHT	Exact-match + Multi-attribute
Ratnasamy et al.	Decentralized	Structured	DHT	Exact-match + Range + Multi-attribute
Marzolla et al.	Decentralized	Structured	Tree	Informed
Pastry	Decentralized	Hybrid	DHT	Exact-match
Kademlia	Decentralized	Hybrid	DHT	Exact-match
Mastroianni et al.	Partially Centralized	Hybrid	Flooding (Super-peer)	Informed
XenoSearch	Decentralized	Hybrid	DHT	Exact-match + Range + Multi-attribute

TABLE 2.1: Overview of P2P Systems

System	Organization	Administration	Technology	Scale	Provider	Provider Connectivity
Condor	Centralized	Federated	Grid	LAN	Institution	Stable
Globus MDS-2	Centralized	Federated	Grid	LAN	Institution	Stable
Legion	Centralized	Federated	Grid	LAN	Institution	Stable
BOINC	Distributed	Centralized	P2P + Grid	Internet	Volunteer	Unstable
CCOF	Distributed	Centralized	P2P + Grid	Internet	Volunteer	Unstable

TABLE 2.2: Overview of Grid and Cycle Sharing Systems

System	Architecture	Scale	Search Type
SLP	Client-Server	Enterprise	Directory
Jini	Client-Server	Enterprise	Directory
Goering et al.	P2P	Ad-hoc Network	Informed
Lv and Cao	P2P	Ad-hoc Network	Informed

TABLE 2.3: Overview of Service Discovery Protocols

Compression	Reduce data size via an encoding process which takes advantage of redundant information.
Chunks and Hashing	Divide files into chunks and hash them in order to determine which chunks a client already has and only send the ones that differ.
Erasure Codes	Encode a message into a few symbols which can then be used later on to reconstruct the original message when there are pieces missing from the received message.
Bloom Filters	Space-efficient probabilistic data structure that is used to efficiently test whether an element is a member in a set, with the possibility of a false-positive occurring.

TABLE 2.4: Summary of Efficient Data Representation techniques

Chapter 3

Architecture

The objective of this work is to enhance the resource discovery mechanism in GINGER[11] (**G**rid **I**nfrastructure for **N**on-**G**rid **E**nvi**R**onments), also known as GiGi, by making it completely decentralized and more complete. This completeness regards the system's ability to discover, not only basic resources (e.g. CPU, Bandwidth, Memory, etc.), but also specific installed applications and services. Because GiGi can be used in many different ways ("grid-for-the-masses"), it has to be flexible enough to run different types of jobs normally performed by home-users. Each job has a set of minimum requirements in order to be completed. Thus, the discovery of resources (e.g. CPU, memory, storage, etc.), services (e.g. face recognition, high-res rendering, etc.), and applications (e.g. video encoders, simulators, etc.) is a critical component that needs to be as efficient as possible. For, if it is not efficient, it will not be used. After all, if the main objective of GiGi is to bring more computing power via parallelization of tasks to home-users and resource discovery is slow, then it has failed. Note that, as Ginger targets home-users, this architecture assumes that each node can be both a consumer and producer of resources, so no distinction is made.

This Section, therefore, contains the architectural description of a resource, application, and service discovery mechanism (named SERD) and is divided as follows. Section 3.1 presents the context in which the architecture should be taken into. Section 3.2 provides an overview of the discovery mechanism, along with a description of how it functions. Finally, naming conventions and rules used for resources, applications, and services are discussed in Section 3.4, along with how resource insertion and querying is performed.

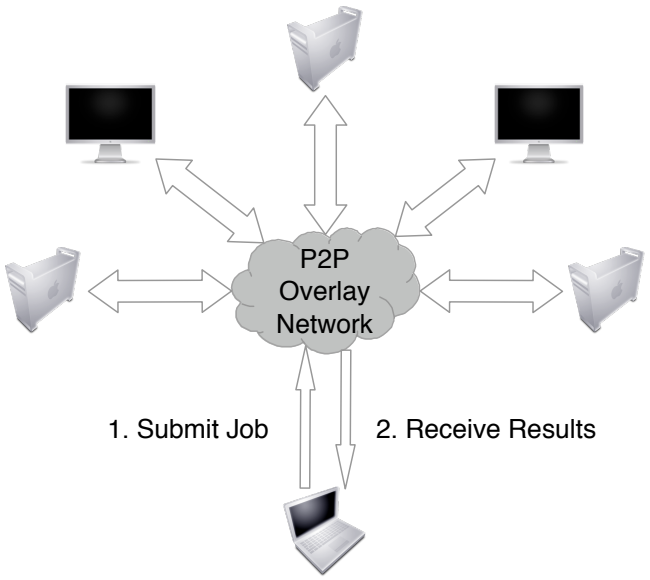

FIGURE 3.1: General overview of a system where resource discovery is necessary. This image represents the Ginger architecture in its most abstract form.

3.1 Ginger Overview

In its most abstract form, the Ginger[11] project can be thought of as a system where a user can submit a job and then later on retrieve its results, as can be seen in Figure 3.1. This job is divided into smaller tasks (called Gridlets), which are then distributed over many volunteered computers that are interconnected in a Peer-to-Peer overlay. Each of these jobs, and its enclosed tasks, have a set of requirements that need to be met in order to be executed. Such requirements may include things like: a CPU of at least 2GHz, version 2.3 of the video encoding application ffmpeg, and at least 50 GBs of free storage space. This is where the work presented in this paper comes into play. It provides a mechanism that is able to locate a computer, connected to a P2P network, that contains a specific resource which is required in order to satisfy the requirements of a task.

3.2 System Overview

In order to cope with a dynamic peer population and high churn rate, this system uses an unstructured peer-to-peer approach, even though message routing may not have optimum efficiency. The reasoning behind this is threefold: the objective of this work is to create a completely decentralized discovery mechanism as the current approach used by GINGER is hybrid (superpeers); second, if a structured system were to be used, the messages needed to keep the structure intact with an unstable population, such as home users, could possibly result in a very high overhead[22]. The third and final reason is while there may be many exact-match queries in resource discovery (e.g. GCC

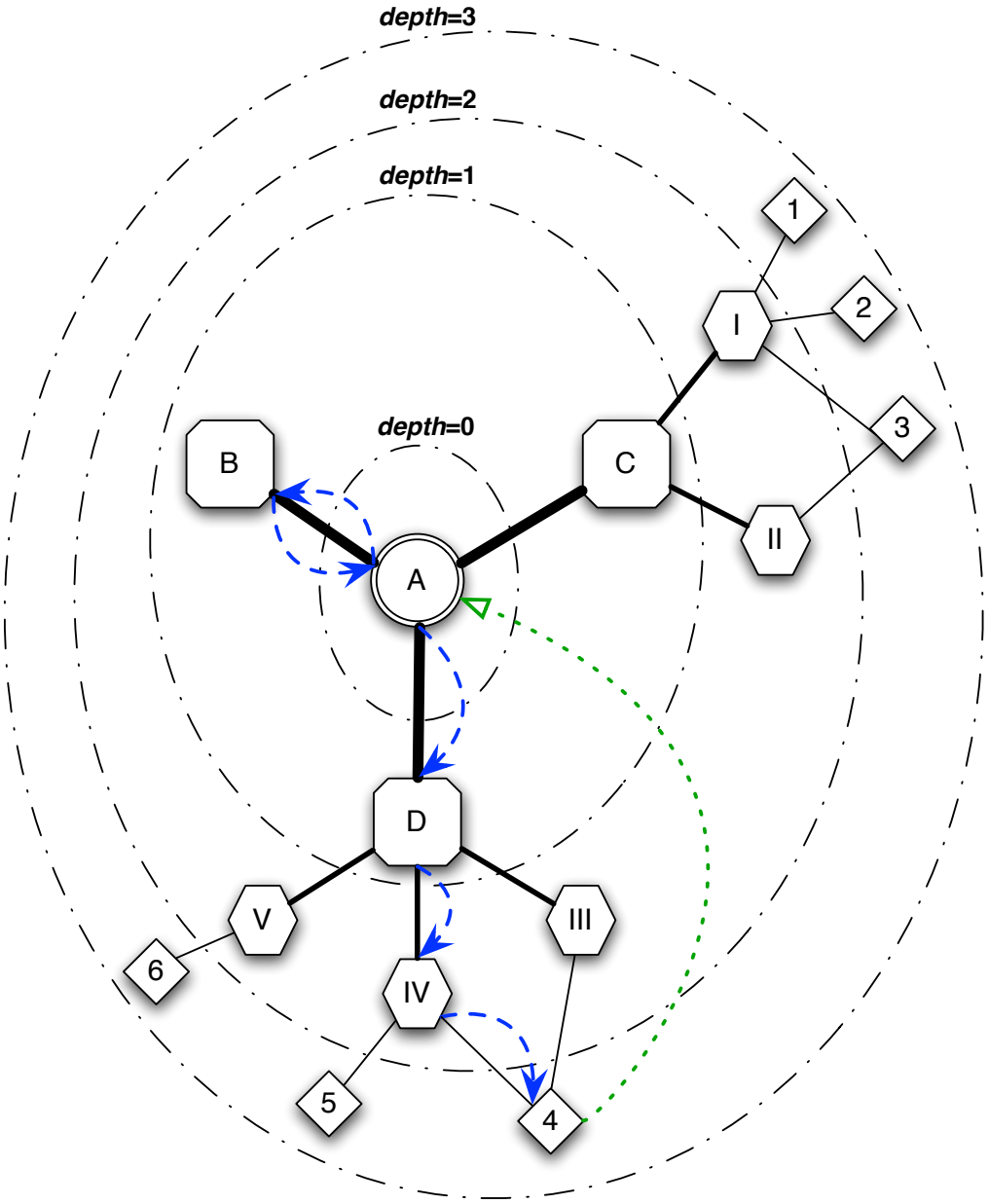

FIGURE 3.2: Example architecture overview of network from node A's perspective. Nodes are enclosed in circles with increasing *depth*, meaning they are *depth* hops away from node A. To help distinguish between nodes at different distances the following visual aids were used: connection line thickness varies from thick (closer) to thin (further away), and nodes at different hops away from node A have their own shape and naming scheme. More specifically, nodes 1 hop away have alphabetic names and are octagons; nodes 2 hops away have roman numerals as names and are hexagons; and nodes 3 hops away have arabic numbers as names and are diamonds. Blue dashed lines with arrow tips show how node A finds peers that are out of reach of the Attenuated Bloom Filter (assuming $depth = 2$), which is explained in Section 3.2.1. The green dotted line with a hollow triangle represents a peer responding to node A's peer discovery query.

application version 4.2), there are also many non-exact queries (e.g. CPU with speed greater than 2000 MHz) which DHTs do not forward as efficiently.

To enhance message routing and speed up resource location in the unstructured network, we employ Attenuated Bloom Filters. Note that this solution is different to the systems mentioned in Chapter 2 because it combines all types of different resources into on discovery mechanism. It is especially different to the works [38, 39] that also make use of Attenuated Bloom Filters due to to the usage of one aggregated Attenuated Bloom Filter (explained next), and the fact that all the different types of basic resources, services, and applications are encoded in the Bloom Filter.

Each node in the network will keep a cached version of the Attenuated Bloom Filters of their neighbors. This information is then combined into one single Attenuated Bloom Filter by calculating the union of each Bloom Filter at the same depth from all neighbors. For instance, say node A receives the following Attenuated Bloom Filters from its neighbors with depth $d = 2$: (00011, 10000) and (11001, 00001). To combine the information, the `OR` operation is performed for each depth. So, for $d = 1$, the resulting information is 11011, and for $d = 2$ it is 10001.

The consequence of using an Attenuated Bloom Filter is that a node will only have access to a summary of services available up to $d = 2$ hops away. This can be seen in Figure 3.2, assuming a maximum depth of 2, where node A only has information about nodes up to 2 hops away, i.e. node A is unaware of the resources, services, and applications present in nodes 1, 2, 3, 4, 5, 6, and beyond. A solution for this problem is discussed further in Section 3.2.1.

3.2.1 Outer Limit Peer Discovery

If a query's requirements cannot be satisfied by nodes within the Attenuated Bloom Filter's depth d limit, the system will forward the query to a node that is $d + 1$ hops away and restart the search. But to do this, a node needs to know about other peers that are out of its range. To find outer limit peers, a simple random walk strategy is used, where a peer discovery query is forwarded to a random neighbor until it reaches a node l hops away, in which case a reply is sent directly to the originating node with contact information (e.g. IP address). If a node is not able to forward the discovery message to a node that has not seen the message before, then it replies to the originating node letting it know that the path it took did not lead to an outer limit node. The originator node then restarts the discovery process, this time sending it to a different neighboring node (this information is stored along with the query message).

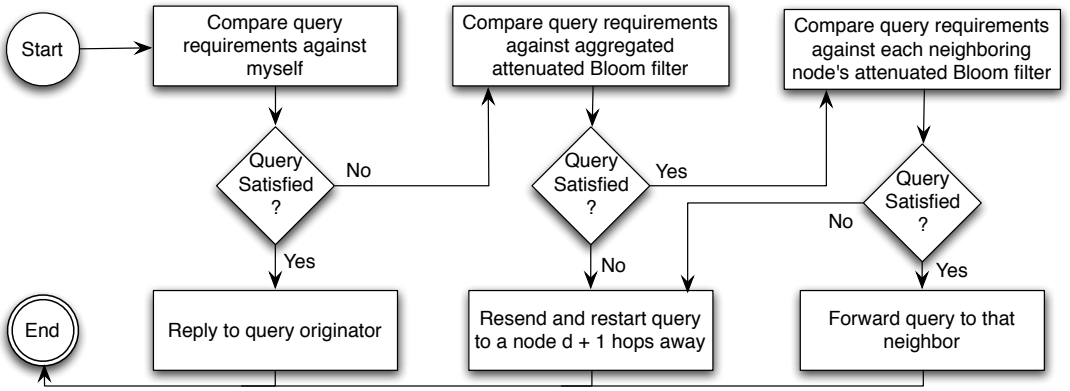

FIGURE 3.3: Flowchart illustrating resource, service, and application discovery explained in Section 3.3

For example, in Figure 3.2, suppose node A cannot satisfy a query with the resource information it has about nodes up to $d = 2$ hops away. It needs to be aware of at least one node more than $d + 1$ hops away in order to restart the query in another neighborhood. To do this, assuming $l = d + 1 = 3$, node A sends a peer discovery query (blue dashed lines with arrows) to a randomly selected node: B. When B gets the query, it sees that it cannot forward it to anyone, therefore it notifies the originator (A) about this. Node A then retries the discovery process by picking a random node (D) as long as it is not B. When D receives the query, it decrements the value of l and forwards it to a random neighbor. This continues until a node decrements l and results in 0. That is the case when the query reaches node 4, meaning it is an outer limit node, and therefor sends its contact information to node A. Now node A is able to restart the resource query at node 4 when needed. This process could be further optimized by having node B, instead of returning a failed search to A, start a new walk for an outer limit peer via a different neighbor.

3.3 Resource, Service, and Application Discovery

The discovery of resources, applications, and services (illustrated as a flowchart in Figure 3.3) is performed in the following way. When a node receives a query, it will check its own information to see if it can satisfy the requirements. If it does, a reply is sent directly to the node that originated the query. If not, it goes through its aggregated Attenuated Bloom Filter, which contains the combined information from its neighbors Attenuated Bloom Filters. This way, we can quickly determine if the query cannot be satisfied with nodes up to d hops away, in which case it will be sent directly to a node $d + 1$ hops away to restart the search. If the query can be satisfied with nodes at most d hops away, the node then needs to determine the direction to send the query in so it

can be resolved. This is done by checking all the cached Attenuated Bloom Filters of its neighbors to determine which one has the requested resources. If found, it then forwards the query to that neighbor. If not, then it is because the aggregated Attenuated Bloom Filter returned a false positive, which is mitigated by simply sending the query to a node more than $d + 1$ hops away so it can be resolved.

For example, in Figure 3.2, if node A were to receive a query it would start by looking at its own resources, services, and applications. If it cannot satisfy the requirements, then node A consults its aggregated Attenuated Bloom Filter, to see if the query can be satisfied with nodes up to 2 hops away, assuming an Attenuated Bloom Filter with depth $d = 2$. In other words, it would quickly determine if either of the nodes B, C, D, `I`, `II`, `III`, `IV`, or `V` contain the resources needed to satisfy the query. If none of them do, then node A needs to forward the query to one of the nodes more than $d + 1$ hops away: node 1, 2, 3, 4, 5, or 6. However, if the query can be satisfied within $d = 2$ hops, then node A needs to determine if it must forward the query to either node B, node C, or node D. This is done by checking the cached Attenuated Bloom Filter of those nodes. The discovery process continues until the query reaches the node that can satisfy all the requirements in the query, at which point the query originator is notified.

3.3.1 Dynamic Resources

Some resources are mostly static and do not change often, like the Operating System, CPU and Disk speed, certain application versions, etc. But there are other resources whose values can change quite often, such as amount of RAM occupied, amount of CPU in use, etc. For those cases, if we used a classic Bloom Filter then it would need to be rebuilt periodically since it does not support the removal of elements. More, this rebuilding procedure would require resending information about resources that are not expected to change, thus wasting bandwidth.

Therefore, instead of using a classic Bloom Filter to store the information about the dynamic resources, a separate Counting Bloom Filter is used. To compensate the fact that a Counting Bloom Filter occupies more storage space than a classic one, we use a smaller Counting Bloom Filter size (less precision), as the number of static resources is greater than dynamic ones. The usage of this new Bloom Filter mirrors that described in the previous sections: queries for dynamic resources use Aggregated Counting Bloom Filters instead and are checked after the static Aggregated Bloom Filter.

The difference is that when a dynamic resource changes, the resource is removed from all the Attenuated Bloom Filters. If this alteration does not affect the key that was used for the Bloom Filter (explained further in Section 3.4), then nobody needs to be updated.

But, if the resource's value changed drastically, or if it was added/removed, then the nodes in the network need to be notified. This is done by defining a periodic interval which checks for alterations to the resources in the main Attenuated Bloom Filter, which is then sent to the node's neighbors. Each neighbor also does this periodic check for alterations, and then resends its own Attenuated Bloom Filter with the changes to its neighbors. This continues until everyone is up to date. By using this periodic interval to send updates, we avoid wasted messanges and bandwith when resource values jitter.

3.3.2 Node Entry/Departure

For a node to join the network, it has to contact an already participating member. When the new node establishes a connection, the already existing member returns its Aggregated Attenuated Bloom Filter information. After the new node integrates this new information, it sends its own Aggregated Attenuated Bloom Filter to the already existing node which the updates its tables, and then sends the Aggregated Attenuated Bloom Filters with dynamic and static resource information to its direct neighbors. Those direct neighbors will eventually do the same until all proper neighbors are updated.

With regards to node departure/failure, each member of the P2P network periodically sends a Ping message in order to verify if its neighbors are still alive. If there is no response, then in the next periodic check, that neighbor's information is purged from the Aggregated Attenuated Bloom Filter and its cache is deleted. The node that detected the failure then needs to rebuild its Aggregated Attenuated Bloom Filter and resend it to its neighbors. Bloom Filters are then exchanged until all neighbors are up to date. Note that this system assumes that the TCP protocol is used for Ping messages so if there is no reply, we can consider the node has left the network.

3.4 Resource Representation

Information about resources, applications, and services that each node offer are represented inside a Bloom Filter. But, because a Bloom Filter is only capable of performing membership tests given a key, we need to store information about those resources in the actual key. For example, say a node has a CPU of 3GHz, we cannot simply store the name "CPU" in the Bloom Filter, as the only information we can extract from that is that a node has a CPU. We need to add information about the actual resource (e.g. its value: 3000MHz) to the key that is inserted in the Bloom Filter for it to be useful.

Bloom Filter keys store resource information by following a naming convention. Namespaces are used to differentiate between resources and their values, which will also help

with the searching of resources (discussed in Section 3.4.1). The naming convention uses a 3-level namespace, each separated using the colon (":") as a delimiter, with the following rules:

- Level 1: Name of the Resource, Service, or Application (e.g. CPU, ffmpeg, etc)
- Level 2: Type of the Resource, Service, or Application (e.g. MHz, version, etc.)
- Level 3: Actual value of the Resource, Service, or Application

For instance, if we wanted to store the fact that a node has a CPU of 3 GHz, the key we would insert into the Bloom Filter would be: "CPU:GHz:3". Or, if a node has the application ffmpeg version 2.3 installed, the key would look like: "ffmpeg:version:2.3". But, for different nodes to be able to communicate with each other and search for the same resources, the naming of resources, services, and applications need to be the same between all of them. An ontology could be used, but that is out of the scope of this work. For the time being, the system allows the names of these different resources to be specified in a configuration file, and we assume that all nodes that take part in the system use the same configuration files so as to use the same names.

3.4.1 Resource Insertion and Querying

However, just following a naming convention will not suffice for the discovery of resources. We also need to take into account the values used for each resource. If we do not restrict the possible values, we would need to employ a brute force strategy when querying for resources, trying each value combination and testing the Bloom Filter. For example, to find a node that at least contains a CPU of 2.6 GHz, we would need to test for values such as 2.6, 2.7, 2.8, 2.9, 3.0, etc., which is highly inefficient. To speed this up, we define a *minimum*, *maximum*, and a *quantum* for each resource value type (which are also specified in a configuration file). The *minimum* (resp. *maximum*) is the smallest (resp. largest) value that the resource will have encoded in the Bloom Filter. The *quantum* defines how the value space, from *minimum* to *maximum*, will be divided. When a resource is inserted into the Bloom Filter, it is first inserted with the key that corresponds to its range, and then with all the other keys that correspond to ranges smaller than the resource's value.

For example, if we define $minimum = 0$, $maximum = 4000$, and $quantum = 1000$ for CPU values in MHz, then the range of values is divided into the following segments:

$]0, 1000]$; $]1000, 2000]$; $]2000, 3000]$; and $]3000, 4000]$. This can be seen in Table 3.1b. If a CPU of 999MHz were to be inserted into the Bloom Filter, it would need to be inserted under the value 1000: "CPU:MHz:1000". If a CPU of 2600 MHz were to be inserted, then it would need to be inserted under the values 3000, 2000, and 1000, which results in the following keys: "CPU:MHz:3000", "CPU:MHz:2000", and "CPU:MHz:1000".

Now, when querying a Bloom Filter for a value, the range the value falls under needs to be determined for the specified resource and checked. For instance, if a query requires a CPU of at least 2600 MHz, we would only need to perform one exact match query using the range the value in the requirements belongs to, which in this case is 3000 ($2600 \subset]2000, 3000]$). Therefore, we only need to test the key "CPU:MHz:3000" against a Bloom Filter because processors with a faster CPU will also be registered under this key. This strategy avoids the brute-force approach and efficiently speeds up the querying process.

However, one needs to take care when specifying the *quantum* value due to precision problems. In this example, a CPU of at least 2600 MHz is required, but testing the Bloom Filter with key "CPU:MHz:3000" can result in CPUs that belong to the interval $]2000, 2599]$, thus not satisfying the requirements. In a real-world system, using a $quantum = 200$ would probably be more suitable, giving enough precision without requiring too much overhead. This, and using a key one *quantum* higher than the required resource value will ensure query satisfaction.

Computer	**CPU** (MHz)
P1	999
P2	1333
P3	2000
P4	2600
P5	3006

(a)

Computer	**CPU:MHz:1000**]0, 1000]	**CPU:MHz:2000**]1000, 2000]	**CPU:MHz:3000**]2000, 3000]	**CPU:MHz:4000**]3000, 4000]
P1	✓			
P2	✓	✓		
P3	✓	✓	✓	
P4	✓	✓	✓	
P5	✓	✓	✓	✓

(b)

TABLE 3.1: Example (used in Section 3.4.1) showing the keys that need to be used when inserting the CPU resource values into a Bloom filter.

Chapter 4

Implementation

SERD was implemented in the Java[1] language, version 1.6, using the Peersim[57] simulator (version 1.0.5). Peersim is ideal for simulating P2P networks where there are virtual nodes connected to others in a simulated topology. Other than that, there is no additional functionality to help with the construction and simulation of resource discovery systems. This lead to the development of additional components to facilitate the testing of this discovery mechanism, which will be discussed in this Chapter (Section 4.1 to Section 4.4). The generation and automatic execution of tests are discussed in Section 4.5; and, finally, simulation metrics gathering in Section 4.6.

4.1 Topology Manager

The topology manager has two basic functions: generating a random topology to then be used in other simulations, and loading an existing topology from a file. Topology generation takes three parameters: the number of nodes the network should have, the minimum, and the maximum number of neighbors a node should have. It then randomly assigns neighbors to each node and creates a file in the GML format with the topology. Note that this process can result in two disjoint networks. When this happens, manual intervention is needed to visually remove (using the application yEd[2]) the nodes that aren't connected to the main network. This may lead to network sizes with non exact numbers (e.g. 9982 when the network was initially created with 10000).

Topologies can be loaded from a file at the beginning of a simulation ("topology.txt"). This component can read two different types of configuration file formats. The first format is the GML filetype that can be read and manipulated using a program such as

[1] http://www.java.com
[2] http://www.yworks.com/en/products_yed_about.html

yEd (Figure 4.1b). The second is just a simple custom format (Figure 4.1a), where each line specifies the neighbors:

$$< node >: \quad < neighbor_1 > \quad < neighbor_2 > \quad \cdots \quad < neighbor_n >$$

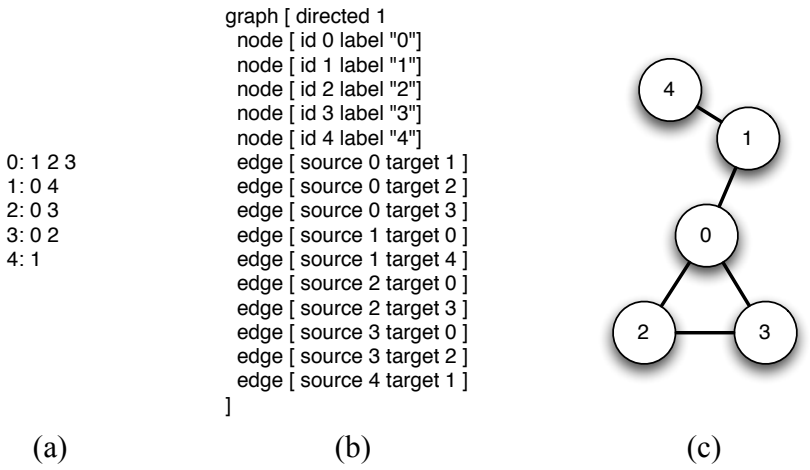

FIGURE 4.1: (a) Topology in plain format. (b) Topology in GML format. (c) Graphic representation of topology.

4.2 Node Resource Description Language

The Node Resource Description Language, or more easily known as NRDL, allows us to map resources to nodes in a simulation based on a configuration file. This can be done either by specifying exactly which resources each node has, or by using criteria to select nodes for a specific resource. The first case reads a configuration file ("resources.txt") where each line specifies the resource a node hold should using the following syntax:

$$< node > \quad < resource\_name > \quad < resource\_type > \quad < resource\_value >$$

Resources can be attributed to nodes in a more general manner with a different configuration file ("resource_distribution.txt"). Each line of this file allows us to specify a resource and the criteria to be used in order to select the nodes for that resource:

$$< r\_name > \quad < r\_type > \quad < r\_value > \quad < quantity[\%] > \quad < criteria_1 > \ [\&\& \ < criteria_n >]$$

For example, the line `CPU MHz 3000 10 random && avg_distance 2` means that the CPU of 3000 MHz resource will be distributed to 10 random nodes that have an average distance of 2 hops between them. Instead of absolute values, percentages can be used

to specify the number of node that will get the resource, for example, `HD GB 300 10% random && no_resource CPU:MHz:3000` where 10% of the node population that do not have the CPU resource of 3000 MHz are given the 10 GB Hard Drive resource.

Other node criteria can be easily added to the NRDL component, although one must take care when specifying the selection criteria as sometimes not all conditions can be met; in that case the component fails and prints an output error notifying the user. This component also has the capability of writing to a file the exact nodes that were selected based on the chosen criteria, which permits us to replicate the same resource distribution in another simulation.

Criteria	**Parameters**	**Description**
random	–	Select a random node.
avg_distance	H - Hops	Select a node that has least H hops distance with another node with the same resource.
no_resource	R - Resource	Select a node that does not have the resource R

TABLE 4.1: Implemented Criteria for NRDL

4.3 Node Activity Specification Language

NASL, or Node Activity Specification Language, is a component that allows us to script the actions of the virtual nodes in our experiments. It permits us to define which node will execute a specific action at a certain point during a simulation. The component gathers the activity information from a simple line based configuration file ("node_activity.txt").

Each line in the configuration file contains information about how to select the nodes that will be active, the time when they will be active, and the action they will perform at that time. The format of each line is:

$$< node\_specifier > [(< args >)] \ < [@]time > \ < node\_activity > [(< args >)]$$

For example, `node(4) @4 resource_query(CPU:MHz:3000)` says that node 4 (the specifier function) will perform a query (the activity or action) for the CPU resource when the simulation reaches its fourth cycle. If the @ symbol is omitted when defining the time a node will execute an activity, then that activity will be performed in a periodic manner. Another example could be `any_without_resource(10%,CPU:MHz:3000) 5 resource_query(CPU:MHz:3000)` which means that every 5 cycles, 10% of the node population that do not have the CPU resource of 3000 MHz will initiate a query for that resource. As in NRDL, node specifiers and activities can easily be added to NASL.

Node Specifier	Parameters	Description
any	N - Number (Opt)	Select N random nodes.
any_with_resource	N - Number (Opt) R - Resource	Select N nodes that has the resource R.
any_without_resource	N - Number (Opt) R - Resource	Select N nodes that does not have the resource R

TABLE 4.2: Implemented Node Specifiers for NASL

Node Specifier	Parameters	Description
add_resource	R - Resource	Add the resource R to the node(s).
remove_resource	R - Resource	Remove the resource R to the node(s).
update_resource	$R1$,$R2$ - Resource	Update the old resource value in $R1$ to the value in $R2$ to the node(s).
query_resource	R - Resource	Make node initiate a query for the resource R.

TABLE 4.3: Implemented Activities for NASL

4.4 Scenario Manager

The Scenario Manager is what brings the aforementioned components together in order to facilitate the running of experiments. This component also allows us save the output of the NRDL and NASL components in their own scenarios in order to reproduce experiment results. Scenario packages are basically folders that contain all the configuration files necessary to run simulations. These files include things such as: the peersim simulation configuration file, resource distribution files, node activity files, the resource description file (that defines the minimum, maximum, and quantum values for resources), and lastly, the topology file.

This component is implemented in a `Rakefile` that is used by the `rake`[3] build program (similar to `make`[4]) and is programmed in the Ruby[5] programming language. There are a various set of tasks that allow the running of experiments, saving current scenarios, and loading previously saved scenarios. This component can be used manually via the command line, but was also intended to be used in other shell scripts to run batch experiments in an automatic manner.

[3] http://rake.rubyforge.org
[4] http://www.gnu.org/software/make/
[5] http://ruby-lang.org

4.5 Test Generation and Automation

Test generation and execution are all automated using `rake` to specify tasks that generate test scenarios using a base scenario, and automatically execute them. The base scenario consists of all the configuration files in a scenario, but use undefined values that are only attributed during the execution of the rake task. This is done using the `ERB` module of Ruby which permits writing Ruby code inside files, which after processed become a normal file (akin to the way PHP code is embedded in HTML files).

Using Ruby code inside the actual configuration files eases somewhat the creation of complex test scenarios, which is especially useful to do things such as update resources at nodes with certain conditions, at certain times of the simulation (using NRDL and NASL). It can be seen as an embedded programmatic extension to declarative specifications in NRDL and NASL.

The test generation process iterates through a number of values of simulation parameters and processes the configuration files in order to assign those values to the files. Once all files have been generated, the task runs the scenario through the simulator using a discovery protocol that does nothing. At the end of that simulation, the resulting output of the NRDL and NASL components are then merged with the initial scenario, resulting in a test for both SERD and RW where the same nodes have the same resources and perform the same activities at the same time.

The test execution task just consists of going through a directory with all the scenarios, loading and executing them using the tasks defined by the Scenario Manager component, and storing their output in a specified directory. This task also ensures that the Redis server is running in order to collect simulation metrics (explained in the next section).

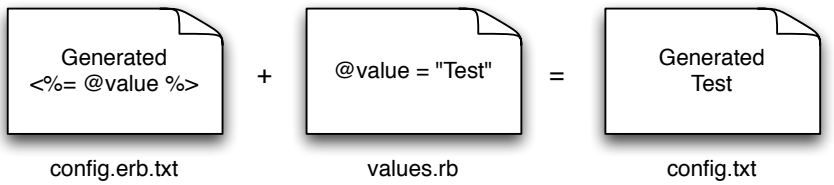

FIGURE 4.2: Simplified overview of the processing of embedded Ruby code in files using ERB

4.6 Simulation Metrics Gathering

During experiments, various metrics are gathered and stored to be analyzed later on. This is accomplished by using Peersim `Control`s that are running during the simulations and at the end output the metrics. The measurements obtained are stored in a key-value store called Redis[6] so that data can be accessed in a structured manner independent of programming language. Metrics are also output to a log file for easy visual inspection.

Metrics collection is split into two components: one for message metrics (number of messages, number of hops, etc.), and one for storage metrics (size of data at each node, message sizes, etc.). These components gather metrics data by hooking themselves into the discovery mechanism, such that when certain methods are called, these components store relevant information. The hooks (or callback) are called explicitly in the discovery mechanism. For example, when a node initiates a query for a resource, before sending the message to a node, it will call the corresponding hook. The component responsible for the message metrics will be invoked and store (in the Redis backend and logfile) that one more resource query was initiated. Other metrics are collected and will be discussed further in Chapter 5.

[6] http://code.google.com/p/redis/

Chapter 5

Evaluation

The SERD discovery mechanism has the objectives of being effective (in terms of satisfying resource queries); efficient in storage space, message number and length; and scalable. To evaluate the system to see if these objectives have been met, a set of simulation scenarios were generated in order to test various aspects of the discovery mechanism. SERD was evaluated using the (event-based) Peersim simulator (as mentioned in Chapter 4) and was compared against another, albeit simpler, discovery mechanism called Random Walk (RW for short). Various simulation scenarios were generated in order to evaluate the effects of various parameters on the discovery process. The RW protocol is used as a baseline and our expectation is that SERD should outperform RW in all scenarios.

The rest of this Chapter is organized as follows. Section 5.1 will discuss the various simulation scenarios that were generated to test SERD and RW. In Section 5.2 the results of those scenarios are presented and analyzed.

5.1 Test Scenarios

Each generated test is used with the SERD and the RW protocol. The tests were generated using the components described in Chapter 4 in order to create scenarios that are exactly the same for both protocols, i.e. the same nodes have a certain resource, the same nodes send a query for a certain resource, even the random seed is the same for the scenarios used to test SERD and RW. Note that all queries that were initiated by nodes could be fulfilled by at least one node in the network, and that every 5 cycles 10% of the node population initiated a resource query. As SERD uses Attenuated Bloom Filter, we decided to vary the depth of the Filter to see the effect is has on resource discovery. Therefore, each test that was executed used four protocols: RW, and three versions of SERD that correspond to Attenuated Bloom Filter depths of 1, 2, and 3.

We tested the protocols with two different network sizes: 5000 nodes and 10000 nodes. For each network size we generated two topologies: one where nodes have a maximum of 3 neighboring nodes, and another where the maximum neighbors is 6. More statistics about the topology graph and neighbor degrees are present in Figure 5.1.

Network Size	Degree Statistics						Graph Statistics	
	Max	Min	Avg.	Variance	Number of minimal items	Number of maximal items	Avg of clustering coefficients	Avg. Of distances to all other nodes
5000	3	3	3	0	5000	5000	2.00E-04	10.37341428
5000	6	5	5.9996	4.00E-04	2	4998	0.001	5.185918544
10000	3	3	3	0	10000	10000	2.00E-04	11.37325297
10000	6	5	5.9996	2.00E-04	2	9998	3.20E-04	5.60869937

FIGURE 5.1: Topology Statistics

Each topology (a total of 4) was used with 3 categories of varying resource abundance: *very abundant* where 50% of the nodes have the resource, *abundant* where 25% have the resource, and *scarce* where only 5% have the resource. For each of these categories, two types of resources were distributed accordingly: one with *uniform* values, such as a specific version of an application where it is either installed or not; and another with *non-uniform* values that can vary quite a bit. We split the tests for static resources and dynamic resources in order to better analyze the performance of each system. For the static scenarios, we defined the GCC v4.2 application as the *uniform* resource, and CPU speed (in MHz) as the *non-uniform* resource with the minimum, maximum, and quantum of 1000, 3000, and 1000 respectively. The dynamic scenarios define the availability of a node to be used exclusively as a *uniform* (response is either "yes" or "no"), and available Hard Drive space (in GB) as *non-uniform* with minimum, maximum, and quantum of 0, 1000, and 50, respectively.

5.1.1 SERD Parameters

SERD Variants	Join Cycles Halt	Aggregated ABF Rebuild	Dynamic Update Period
serd1	1	2	2
serd2	1	3	2
serd3	1	2	3
serd4	1	3	3
serd5	1	2	4
serd6	1	3	4
serd7	2	2	2
serd8	2	3	2
serd9	2	2	3
serd10	2	3	3
serd11	2	2	4
serd12	2	3	4

FIGURE 5.2: Parameters of SERD variations used for preliminary tests with one topology of 10381 and a maximum of 3 neighbors per node; and another topology of 10000 with 6 maximum neighbors per node

10 381.3	Query Satisfaction					Hops	Sent Messages			
Variations	Overall	GCC	CPU	Lock	HD		Overall	Resource	Updates	Peer Discovery
serd1	95.44	100	99.58	99.37	81.31	3	601,158	75,598	196,153	108,977
serd2	94.57	100	99.75	96.15	81.22	4	589,264	79,135	175,547	109,254
serd3	95.68	99.98	99.77	99.76	81.76	3	595,934	74,428	187,053	109,125
serd4	95.29	99.98	99.66	98.89	81.3	3	600,137	76,036	189,630	109,143
serd5	*95.41*	*100*	*99.66*	*98.03*	*83.96*	*4*	*601,382*	*80,273*	*186,417*	*109,364*
serd6	93.09	100	99.63	91.84	79.89	4	602,104	81,551	185,947	109,278
serd7	95.50	100	99.63	99.21	81.86	3	601,778	75,847	191,492	109,111
serd8	94.69	99.98	99.58	95.93	82.09	4	587,105	78,726	173,501	109,550
serd9	95.43	99.93	99.6	99.47	81.26	3	593,108	75,273	183,204	109,303
serd10	95.25	100	99.66	98.92	80.96	3	598,342	76,305	187,603	109,106
serd11	93.72	99.98	99.63	91.59	83.04	4	598,012	80,438	182,858	109,388
serd12	93.85	99.93	99.86	92.22	82.61	4	598,091	80,663	182,922	109,178

FIGURE 5.3: Preliminary test results for the topology of 10381 nodes and 3 maximum neighbors

The discovery mechanism in this work has many configurable parameters. As it would be impossible to test the effect of all parameters, we executed some preliminary tests in order to figure out reasonable parameters to use in the tests against the RW protocol. These tests were executed with two topologies of 10381, and 10000 nodes with 3 and 6 maximum number of neighbors respectively (both topologies were intended to have 10000 nodes, but due to the generation process described in Section 4.1 one topology resulted in 10381) . Every 5 cycles 10% of the nodes send queries for resources that can be satisfied by at least one in the population, for each resource. The resource included both the static and dynamic resource categories (*uniform* and *non-uniform*) where the distribution was the worst possible: 5 percent (*scarce*).

10 000.6	Query Satisfaction					Hops	Sent Messages			
Variations	Overall	GCC	CPU	Lock	HD		Overall	Resource	Updates	Peer Discovery
serd1	96.55	100	100	99.95	85.17	3	1,013,668	63,060	449,425	124,511
serd2	95.79	100	100	97.45	84.72	3	918,475	66,841	350,389	124,573
serd3	96.58	100	100	####	84.98	3	986,379	62,207	422,889	124,610
serd4	96.60	100	100	99.88	85.49	3	974,196	63,863	409,078	124,583
serd5	*95.94*	*100*	*100*	*98.75*	*85.02*	*3*	*974,187*	*68,815*	*404,265*	*124,435*
serd6	94.07	100	100	91.85	83.65	3	975,557	68,822	405,453	124,610
serd7	96.43	100	100	99.93	84.6	3	1,007,364	63,492	442,528	124,672
serd8	95.91	100	100	96.88	85.92	3	915,601	66,084	348,296	124,549
serd9	96.44	100	100	####	84.47	3	966,190	62,540	402,469	124,509
serd10	96.46	100	100	99.95	84.81	3	973,933	64,074	408,493	124,694
serd11	94.27	100	100	92.03	84.28	3	970,573	68,750	400,574	124,577
serd12	94.87	100	100	93.35	85.56	3	970,514	68,310	401,056	124,476

FIGURE 5.4: Preliminary test results for the topology of 10000 nodes and 6 maximum neighbors. The row in italic corresponds to the parameters that were chosen for the other scenarios.

The parameters used in these preliminary testes can be seen in Figure 5.2 with the results in Figure 5.3 and Figure 5.4. The parameters chosen for the experiments against SERD are in Table 5.1. Note that the Bloom Filter and Counting Bloom Filters were not experimentally determined due to the fact that the nodes in the generated tests do not contain very many resources, as would be typical in common usage scenarios. This is indented as we want to test the ability of the system to discovery resources,

Parameter	**Value**	**Description**
Join Protocol Halt	1	Number of cycles of inactivity to stop the join protocol
Outer Limit Jumps	$log_2(NW\_SIZE)$	Maximum number of outer limit jumps
Attenuated Bloom Filter Rebuild	2	Period that defines when filter should be rebuilt after receiving an update
Dynamic Update Period	3	Period that defines when to send resource updates
Bloom Filter - N	100	Number of items to store
Bloom Filter - P	$1.0e^{-9}$	False positive probability
Couting Bloom Filter - N	50	Number of items to store
Couting Bloom Filter - P	$1.0e^{-9}$	False positive probability

TABLE 5.1: The parameters used for the SERD protocol during the tests with RW.

not see how many resources each node can hold. Even though nodes do not have many resources, we used values for the Bloom Filters and Counting Bloom Filters as if we were in a typical case in order to obtain more realistic results in terms of storage and message size. With regards to the Counting Bloom Filter size being smaller than the classic Bloom Filter, this is intentional as in typical usage scenarios there are more static resources than dynamic ones, plus this offsets the higher storage requirements of Counting versus classic Bloom Filters. A final note regarding the experiments, because no scenario includes the entry or exiting of nodes, the node entry/departure protocol from Subsection 3.3.2 has been disabled (ping messages were being sent unnecessarily).

5.1.2 Dynamic Resource Updating

The scenarios that include the dynamic resources (availability to be used exclusively and Hard Drive) not only need to be distributed among the node population, but also need to change value over time. As there is no notion of time in Peersim, only one based on cycles, we defined typical times of resource consumption: 5 cycles (short task), 10 cycles (typical task), and 20 cycles (long running task).

In order for tests to be exactly the same for all protocols, the values 5, 10, and 20 were randomly picked until the total was at least 5 cycles less than the maximum number of cycles defined for the Peersim simulation (100 cycles in our case). Then at each point from that list, one third of each resource went down 20%, the other one third of the resources maintained their value, and the rest of resources were increased by 20% of their value.

5.2 Result Analysis

During the execution of the various test scenarios various types of metrics were collected. Raw metrics data can be seen in Figure 5.5 and Figure 5.6, but in this section we will focus on the graphic representation of the following metrics:

1. Resource Query Satisfaction
2. Average Number of Hops Resource Queries
3. Total Number of Sent Messages
4. Average Size of Storage at each Node and Message Size

As the scenarios were split between static resources and dynamic resources, we shall analyze the results the same way in Section 5.2.1 and Section 5.2.2, respectively.

5.2.1 Static Scenario Results

With regards to the satisfaction of resource queries (Figure 5.7), SERD1 and SERD2 consistently got a percentage rate above 90% except for the *scarce* scenarios with a maximum of 3 neighbors. This can be explained by the fact that the depth of the Attenuated Bloom Filters did not allow the forwarding of queries with much hindsight, especially in a scenario where very little nodes actual contain the resource and where each node only has a maximum of 3 neighbors, thus further limiting a node's knowledge about the network. SERD3 in almost all scenarios had a satisfaction rate of 100%, and in others 99%. As the algorithm had a greater depth, it was able to direct queries in the right direction for them to be satisfied. The RW algorithm's lack of intelligence in the forwarding of queries is a great contrast, with almost all satisfaction rates below or around 80%. While it performs better in scenarios where the resources are abundant, it suffers in the *scarce* ones.

The number of hops a query messages takes in order for it to be satisfied is another important aspect in a discovery system, which needs to be as low as possible due to network latency. As we can see in Figure 5.8, RW queries were consistently higher than any of the SERD protocols because of the lack of query success, which made the query reach the maximum number of hops (Outer Limit Jumps value) and fail. SERD1 to SERD3 all had an average below 3 hops in all tests except for the *scarce* ones with a maximum of 3 neighbors. In those cases, SERD1 performed the worst, while SERD3 the best. This can be explained by the fact that the lack of resource knowledge (defined

Static Scenarios			Query Satisfaction			
			RW	SERD1	SERD2	SERD3
5000	3	50	0.869	0.999	1.000	1.000
		25	0.716	0.977	0.991	1.000
		5	0.256	0.605	0.744	0.961
	6	50	0.909	1.000	1.000	1.000
		25	0.800	1.000	1.000	1.000
		5	0.318	0.936	0.997	1.000
10000	3	50	0.878	0.997	0.999	1.000
		25	0.735	0.976	0.993	1.000
		5	0.266	0.628	0.744	0.963
	6	50	0.915	1.000	1.000	1.000
		25	0.812	1.000	1.000	1.000
		5	0.339	0.936	0.993	1.000
			Average Hops			
			RW	SERD1	SERD2	SERD3
5000	3	50	4.588	2.002	1.942	1.939
		25	6.981	2.986	2.465	2.442
		5	11.218	8.231	6.101	4.629
	6	50	4.137	1.796	1.798	1.807
		25	6.337	2.188	2.097	2.095
		5	10.892	5.079	2.998	2.934
10000	3	50	4.673	1.997	1.938	1.930
		25	7.239	3.061	2.466	2.443
		5	12.000	8.511	6.335	4.619
	6	50	4.218	1.794	1.803	1.799
		25	6.551	2.175	2.094	2.098
		5	11.560	5.318	3.020	2.941
			Sent Messages (Bytes)			
			RW	SERD1	SERD2	SERD3
5000	3	50	106,175	117,045	135,170	154,248
		25	151,632	128,559	138,452	157,707
		5	232,145	214,304	185,236	172,243
	6	50	97,612	154,757	190,042	225,035
		25	139,408	153,252	187,285	222,025
		5	225,948	180,683	178,618	215,769
10000	3	50	215,570	233,238	270,040	308,432
		25	313,078	259,568	276,419	315,475
		5	493,996	438,991	379,282	344,748
	6	50	198,275	309,955	380,402	449,641
		25	286,919	306,545	375,115	445,333
		5	477,296	370,950	358,721	430,839
			Storage Sizes (Bytes)			
			RW	SERD1	SERD2	SERD3
Average			116.813	27,934.451	39,704.824	51,776.040
Median			116.838	27,537.702	39,258.624	51,549.600
Std. Deviation			39.207	6,284.885	10,002.725	13,731.028
			Message Sizes			
			RW	SERD1	SERD2	SERD3
Average			483.385	2,086.222	2,739.632	3,426.992
Median			483.782	2,020.617	2,763.708	3,471.658
Std. Deviation			7.756	468.235	634.138	766.299

FIGURE 5.5: Test Results for the Static scenarios

by the Attenuated Bloom Filter depth) made queries take non-optimum routes while looking for the resource, or even fail.

In Figure 5.9, we can see the total messages sent by each protocol. It is to be expected that in this case, the RW protocol typically uses a lot less messages because it does not have to trade resource information. The cases where RW uses more messages than any of the SERD protocol is because of the low query success rate, which means that there were a lot of messages that traveled until the maximum depth. It is also to be expected

Dynamic Scenarios			Query Satisfaction			
			RW	SERD1	SERD2	SERD3
5000	3	50	0.804	0.953	0.962	0.973
		25	0.580	0.859	0.910	0.933
		5	0.172	0.413	0.517	0.770
	6	50	0.879	0.992	0.996	0.997
		25	0.692	0.979	0.992	0.992
		5	0.221	0.739	0.924	0.950
10000	3	50	0.824	0.951	0.967	0.974
		25	0.607	0.878	0.901	0.942
		5	0.175	0.426	0.511	0.760
	6	50	0.894	0.995	0.996	0.996
		25	0.718	0.981	0.989	0.990
		5	0.234	0.799	0.926	0.960
			Average Hops			
			RW	SERD1	SERD2	SERD3
5000	3	50	6.506	3.294	2.820	2.688
		25	8.840	5.099	3.700	3.359
		5	11.937	10.010	8.320	6.385
	6	50	5.785	2.473	2.299	2.274
		25	8.138	3.201	2.594	2.591
		5	11.701	7.539	4.109	3.609
10000	3	50	6.639	3.354	2.811	2.701
		25	9.217	5.097	3.752	3.347
		5	12.805	10.587	8.717	6.669
	6	50	5.868	2.459	2.295	2.278
		25	8.413	3.184	2.674	2.622
		5	12.484	7.516	4.230	3.555
			Sent Messages (Bytes)			
			RW	SERD1	SERD2	SERD3
5000	3	50	142,613	411,878	490,195	507,618
		25	186,959	371,774	462,658	523,537
		5	245,811	273,239	348,055	427,267
	6	50	128,923	755,191	909,318	883,116
		25	173,617	611,513	835,902	905,556
		5	241,327	397,731	732,047	916,445
10000	3	50	290,297	809,075	950,000	985,259
		25	388,233	747,874	995,752	1,046,154
		5	524,604	581,385	734,252	895,224
	6	50	261,002	1,491,106	1,697,448	1,888,343
		25	357,694	1,306,008	1,796,070	1,812,460
		5	512,377	767,788	1,526,618	1,808,986
			Storage Sizes (Bytes)			
			RW	SERD1	SERD2	SERD3
Average			117.048	26,639.434	36,666.158	46,780.660
Median			116.953	26,359.062	35,886.988	45,414.917
Std. Deviation			39.460	5,892.140	9,250.289	12,750.259
			Message Sizes			
			RW	SERD1	SERD2	SERD3
Average			488.564	3,271.013	4,592.080	5,703.195
Median			488.967	3,431.624	4,811.776	5,861.995
Std. Deviation			4.769	845.025	758.803	595.192

FIGURE 5.6: Test Results for the Dynamic scenarios

that SERD3 uses more messages than SERD1 or SERD2, especially in the scenarios with 6 maximum neighbors, due to the greater Attenuated Bloom Filter depth. Its depth and the amount of neighbors each node has knowledge of influences greatly the joining phase of the discovery process where Attenuated Bloom Filters have to be traded among nodes until everyone is up-to-date.

Figure 5.10 confirms what we already expected: the greater the Attenuated Bloom Filter depth, the higher the storage costs at each node and the bigger the message size due to

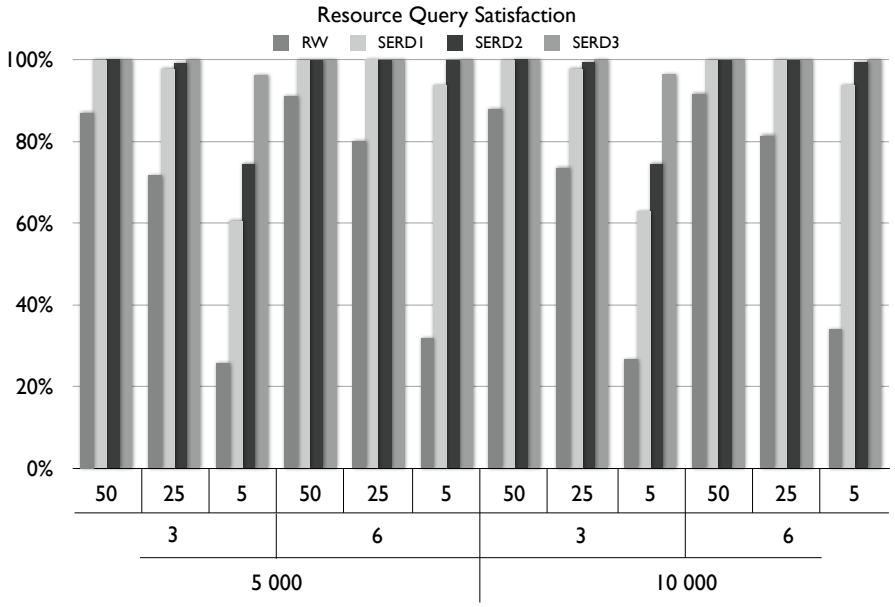

FIGURE 5.7: Query Satisfaction for Static scenarios with topologies of 5000 and 10000 nodes, where each vary between 3 and 6 neighbors per node, and resource abundance varies between 50%, 25%, and 5%.

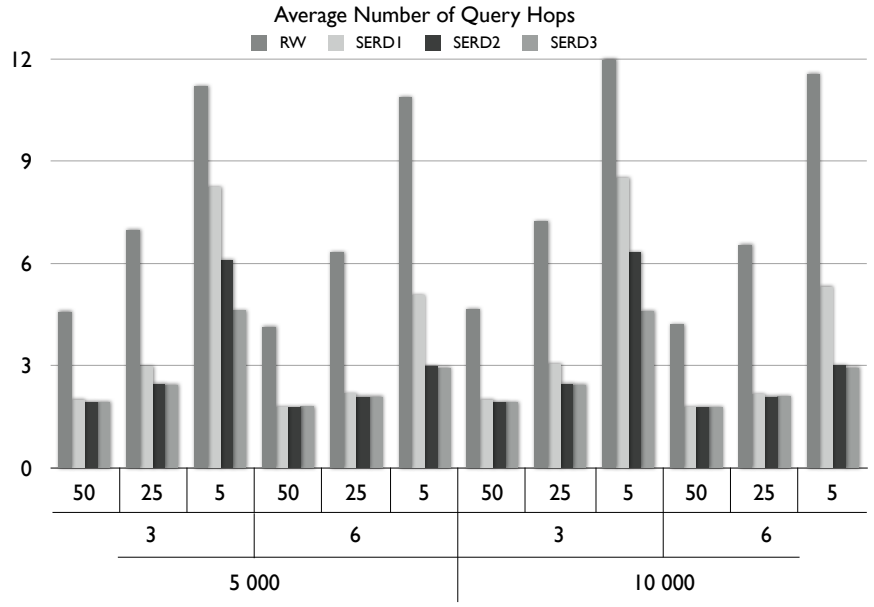

FIGURE 5.8: Average Number of Hops for Static Scenarios

the trading of resource information. The RW protocol uses so little storage space that it does not appear on the graph (average of 483.38), which is normal as it has no information about neighboring nodes and shows in terms of query satisfaction. Nonetheless, in a real scenario, RW would have to store increasingly larger information regarding local resources at each node, which unoptimized would occupy much space. SERD not only keeps information about its own resources, but also caches the Attenuated Bloom Filters

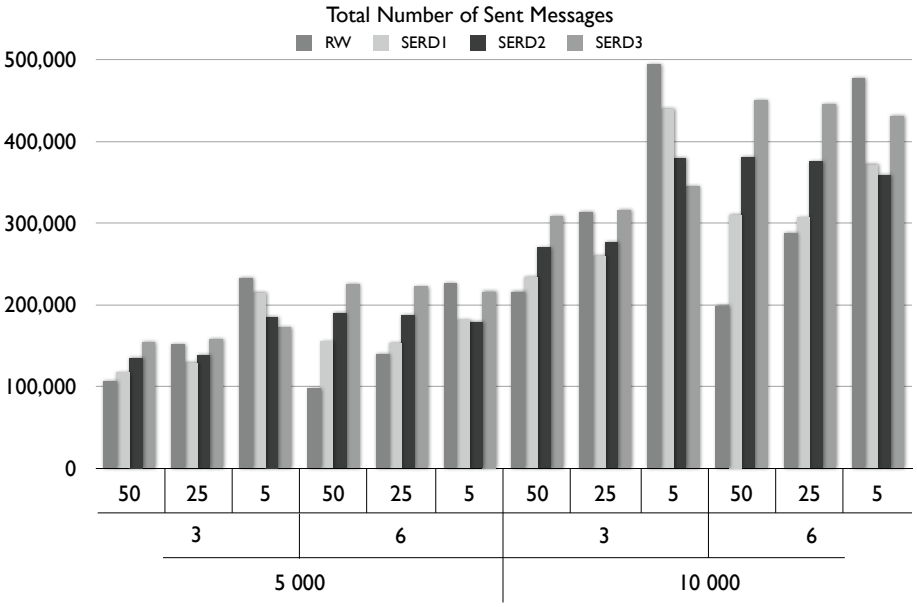

FIGURE 5.9: Total Sent Messages for Static Scenarios

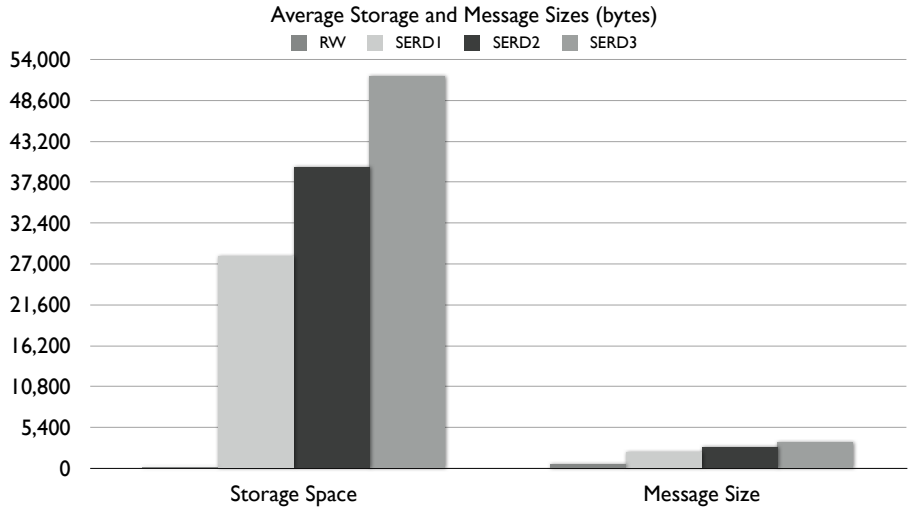

FIGURE 5.10: Average Storage and Message Size for Static Scenarios

of its neighbors. Note that these results do not depend on the number of items actually stored in the Bloom Filters as they have a fixed size (defined in Subsection 5.1.1).

5.2.2 Dynamic Scenario Results

Figure 5.11 shows us the query satisfaction for the dynamic resource scenarios, which are expected to not be as high as the static scenarios due to the varying values of the resources. Once again SERD outperformed the RW protocol, which display a success

rate of 80% and lower. In almost all tests, the SERD protocols were above 80%, except for the *scarce* scenario tests. In those, SERD1 struggled the most seeing as it hardly has information about the neighborhood. SERD2 and SERD3 only displayed a satisfaction rate lower than 80% when the *scarce* scenario was combined with a maximum of 3 neighbors, which limited the available options when forwarding query messages. RW in those cases was hardly able to reach 20% query satisfaction, making its lack of intelligence ever so apparent.

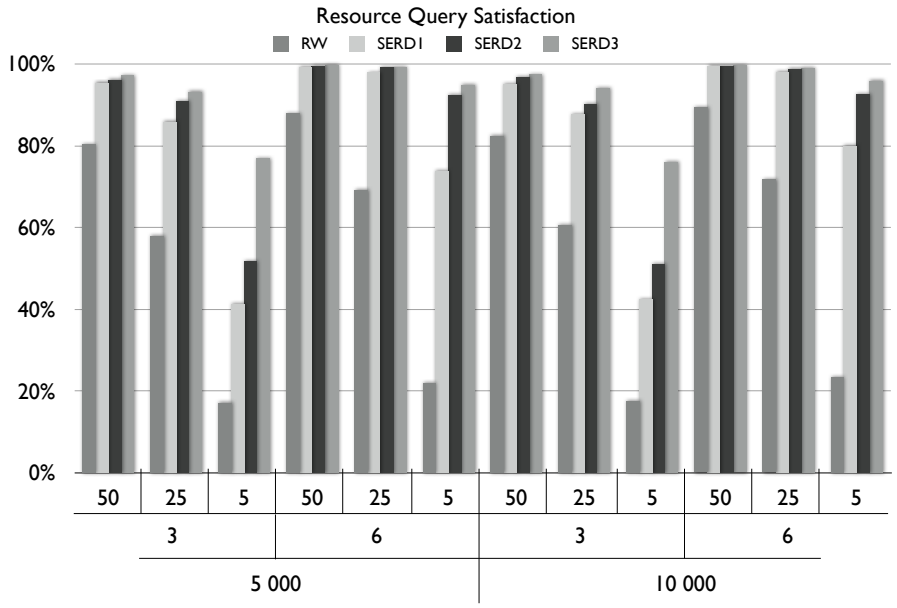

FIGURE 5.11: Query Satisfaction for Dynamic Scenarios

With not so stellar satisfaction results, the RW protocol in Figure 5.12 shows that with high hop averages, being mostly around or higher than 6 hops. The SERD protocols continued to show consistency with lower hop averages, although they had a higher increase in the scenarios where query satisfaction was lower than usual. An increase is query failures leads to a higher amount of hops as queries only fail if they reach the maximum Outer Limit Jumps.

Contrary to the static scenarios, where there were cases that the RW protocol consumed more messages than the SERD protocol, in the dynamic scenarios (Figure 5.13) SERD consistently used much more messages than RW. This is not at all surprising given that not only do nodes exchange resource information when new peers join the network, but also resource information when the dynamic resources change values during the simulation. The Figure display an interesting result: no matter the resource distribution for each topology, the number of sent messages stayed more or less the same. Another interesting result is that the topology of 5000 nodes with a maximum of 6 neighbors, and the 10000 topology with 3 maximum neighbors did not vary that much. Even though

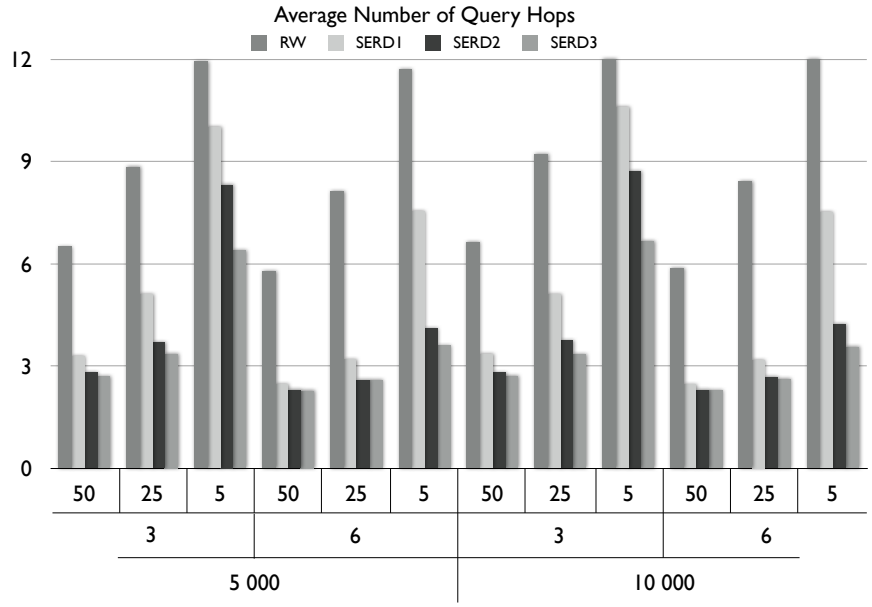

FIGURE 5.12: Average Number of Hops for Dynamic Scenarios

the former topology has less nodes, it sent more messages due to the bigger number of connections; whereas the latter has more nodes sending messages, but were doing so to a smaller number of connections.

Figure 5.14 does not present us with any new information and just confirms what happened in the static scenarios: the deeper the Attenuated Bloom Filter, the bigger the storage requirements are and the bigger the messages sent in the network are.

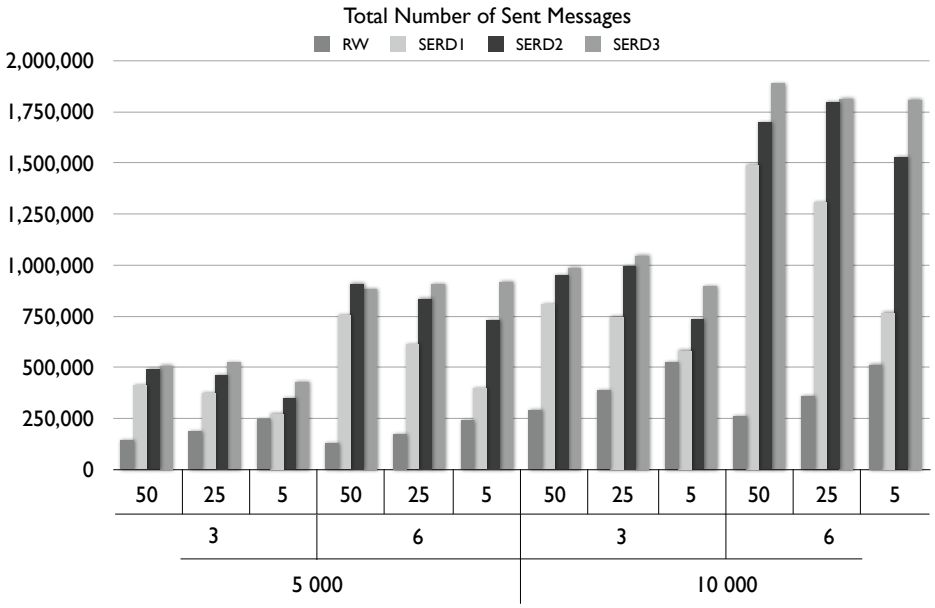

FIGURE 5.13: Total Sent Messages for Dynamic Scenarios

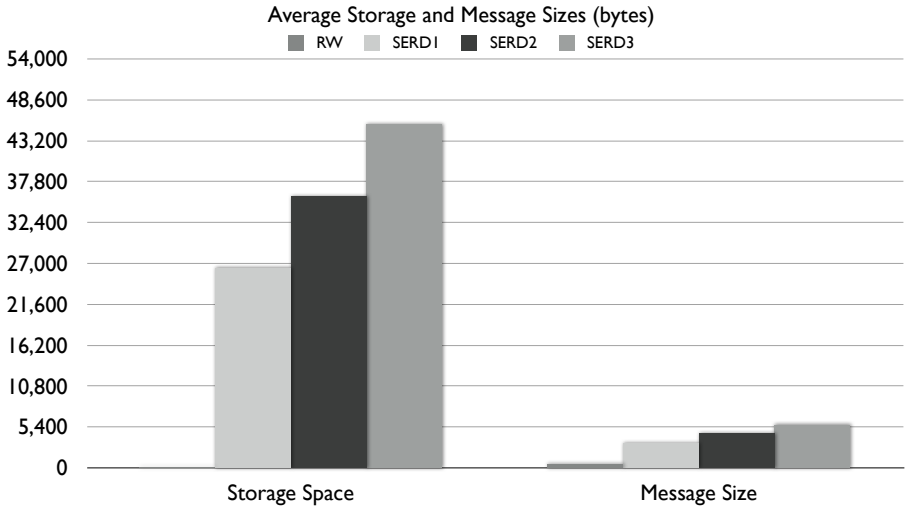

FIGURE 5.14: Average Storage and Message Size for Dynamic Scenarios

Chapter 6

Conclusion

GiGi[11] allows home users to take advantage of Grid computing which was previously only available to scientific and corporate communities. Tasks that would usually take a long time, such as audio and video compression, signal processing related to multimedia content (e.g. photo, video, and audio enhancement), intensive calculus for content generation (e.g. ray-tracing, fractal generation), among others, can now be sped up by parallelizing and distributing them over many home computers. However, to distribute the tasks GiGi needs to locate the resources that satisfy task prerequisites from a potentially large node population connected to the same network.

We analyzed various P2P, Grid, and Cycle Sharing systems that already perform (physical) resource discovery, along with protocols for service discovery. Each of those systems tackled the problem of discovery in isolation; none attempted to combine information about physical resources, applications, and services into one discovery system, which is exactly what GiGi requires. We also assessed various forms to efficiently represent data, such as compression, chunks and hashing, and erasure code techniques, along with the Bloom Filter data structures. Storing data efficiently is especially important in a system that deals with many types of different resources. This assessment led to the conclusion that Bloom Filters are ideal for discovery systems as they allow us to perform efficient membership testing, which is resource discovery at its most basic level (does a node have this resource?), while occupying very little storage space.

Therefore, the architecture presented in this work is a discovery mechanism capable of locating physical resources, services, and applications from many computers connected to the same P2P Grid. This is done in a novel way by storing all resource, application, and service information in Attenuated Bloom Filters. We created a decentralized discovery mechanism that is efficient and scalable for the GiGi project and uses an unstructured P2P network in order to accommodate a highly dynamic node population. Even though

this work addresses the GiGi project, it is completely independent and can be used in other types of networks, such as cycle-sharing networks.

The implementation of SERD consisted of using the Peersim simulator which provides us with a virtualized topology. Unfortunately, it does not provide any additional functionality to help with the construction and simulation of resource discovery mechanisms. This lead to the development of additional components such as a topology manager, to load and generate network topologies; NRDL, which allows us to distribute resources to nodes; NASL, which enables us to script the activity of the nodes in the network during a simulation; and a scenario manager which brings the previous components together in order to facilitate the executing of simulations and reproduce experiment results. Test generation and automation was done using `rake` to create tests from a base scenario. It also makes use of `ERB` that allows us to embed Ruby code into the generation of the various configuration files used by the implemented components. It can be seen as an embedded programmatic extension to the declarative specifications in NRDL and NASL, allowing us to incorporate complex logic in test generation. Finally, in order for tests to have value, various metrics are collected during simulation executions. These metrics are stored in a key-value backend called `redis`, giving us a way to access test data in a structured manner independent of the programming language.

SERD's evaluation consisted of using three variations which correspond to different values for the Attenuated Bloom Filter depth (values 1, 2, and 3). We also used another, albeit simpler, discovery protocol for comparison called Random Walk which acts as a baseline: no protocol should perform worse. Various simulation scenarios were generated in order to test the discovery mechanism's ability to find static resources and then dynamic resources. They were generated using topology sizes of 5000 and 10000 nodes, each with a maximum of 3 and 6 neighbors per node. Resource abundance was varied from *very abundant* (50%), *abundant* (25%), and *scarce* (5%). Resource values also varied and where *uniform* resources had a short variation - a node either has the resource or not (GCC in the static tests and availability for node to be used exclusively in the dynamic tests) - while *non-uniform* resources had values with bigger ranges (CPU speed in the static tests and HD storage in the dynamic tests).

In terms of test results, the query satisfaction rate in the SERD variations were mostly higher than 80% (and often above 90%) in the static and dynamic scenarios, whereas RW always performed much worse. The SERD variant with the deepest Attenuated Bloom Filter consistently performed better than other variations, which is expected as it has more information about resource localization in the network. These query satisfaction rates of SERD used a consistently low number of hops, although they came at the cost of increased message size (RW does not need to trade resource information) and storage size

(although RW does not optimize local resource storage size which could greatly increase in typical usage scenarios). With regards to the number of sent messages, SERD proved to use less messages in the static scenarios than RW due to the fact that it was able to satisfy queries with less hops. In the dynamic scenarios, SERD consistently consumed more messages as every time a resource changes value, neighbors need to exchange information and update their Attenuated Bloom Filters.

In conclusion, the SERD discovery mechanism described in this dissertation performed well in the test scenarios and outperformed RW, which was our baseline. It proved to be effective in locating various types of resources, and scalable as the number of nodes in the network did not affect the mechanism's resource query satisfaction. The results obtained are encouraging towards our objective of efficiency, taking into consideration that the more resources each node has (expected in real case scenarios), the more space RW will use and thus incur a higher storage cost than SERD (with the Bloom Filters). Although message size in SERD is larger than RW, it is also able to satisfy a lot more resource queries than RW. Taking these points into consideration, we also conclude that SERD is an efficient discovery mechanism.

6.1 Future Work

Even though the SERD discovery protocol has attained the objectives we set out, there is still much work that can be done in order to enhance it.

The major limitation of this implementation is that queries only search for one resource and return only one node that contains such a resource. Ideally, queries should be able to specify more than one resource that some node should satisfy. Finally, instead of only returning one node that satisfies the query's requirements, it could be a list of potential nodes that can be used in case a node fails or its resource was occupied in the meantime. Another limitation is that each node only knows about one Outer Limit Node. It would be interesting to see if knowing more Outer Limit nodes influences resource discovery. Even more interesting would be to determine the cost of having nodes not only know about the Outer Limit peers, but also to be aware of their Attenuated Bloom Filters in order to forward queries more intelligently through other neighborhoods.

The next natural step after seeing that this system performed well would be to implement and evaluate it in more realistic scenarios, using real computers and testbeds (e.g. PlanetLab[1]) for experiments. Another option would be to investigate node failure and entry, which although was implemented, no scenario took that into consideration.

[1] http://www.planet-lab.org/

As this work uses Bloom Filters for static resources and Counting Bloom Filters for the dynamic resources which occupy much more space, it would be interesting to investigate the usage of the Bloom Filter enhancements described in [58] and [59]. Another improvement would be to use Scalable Bloom Filters [55] so that it adapts to the number of resources stored in the Filter and not waste any unnecessary space.

Although this discovery mechanism obtained good results in terms of query satisfaction and number of hops, it consumes a lot of messages. This is due to the trading of resource information in the form of Attenuated Bloom Filters, which could probably be optimized in order to use less messages.

Finally, it would be interesting to see if the integration of the SERD protocol with topology adaption techniques (where similar resources are grouped together), such as that mentioned in [60], would result in high resource query satisfaction. Another especially interesting combination would be the integration of the work described in [61] to make requirement specification more extensive, flexible, and expressive.

Bibliography

[1] Gnutella Protocol Specification. Last checked: 2010-10-01. http://wiki.limewire.org/index.php?title=GDF.

[2] I. Clarke, S.G. Miller, T.W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. IEEE Internet Computing, 6(1):40–49, 2002.

[3] I Stoica, R Morris, D Karger, and M Kaashoek. Chord: A scalable peer-to-peer lookup service for internet applications. Proceedings of the 2001 conference on Applications, Jan 2001. URL http://portal.acm.org/citation.cfm?id=383071.

[4] A Rowstron and P Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Lecture notes in computer science, pages 329–350, Jan 2001. URL http://www.springerlink.com/index/404522p56nm85503.pdf.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, page 172. ACM, 2001.

[6] P Maymounkov and D Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. Proceedings of IPTPS02, Jan 2002. URL http://www.springerlink.com/index/2EKX2A76PTWD24QT.pdf.

[7] S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. ACM Computing Surveys, Jan 2004. URL http://portal.acm.org/citation.cfm?doid=1041680.1041681%25E5%25AF%2586.

[8] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. Lecture Notes in Computer Science, pages 118–128, 2003.

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

[10] A. Iamnitchi and D. Talia. P2p computing and interaction with grids. Future Generation Computer Systems, 21(3):331–332, 2005.

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

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

[13] J Kim, B Nam, P Keleher, and M Marsh. Resource discovery techniques in distributed desktop grid environments. Proceedings of the 7th IEEE/ACM International ..., Jan 2006. URL http://portal.acm.org/citation.cfm?id=1513991.

[14] 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.

[15] I Filali, F Huet, and C Vergoni. A simple cache based mechanism for peer to peer resource discovery in grid environments. Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, pages 602–608, Jan 2008. URL http://doi.ieeecomputersociety.org/10.1109/CCGRID.2008.110.

[16] A Iamnitchi, I Foster, and D Nurmi. A peer-to-peer approach to resource location in grid environments. INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE, pages 413–430, Jan 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.4792&rep=rep1&type=pdf.

[17] B Yang and H Garcia-Molina. Efficient search in peer-to-peer networks. 2002.

[18] Vana Kalogeraki, Dimitrios Gunopulos, and D Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. pages 300–307, 2002. doi: http://doi.acm.org/10.1145/584792.584842.

[19] L Liu, N Antonopoulos, and S Mackin. Social peer-to-peer for resource discovery. Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pages 459–466, Jan 2007. URL http://epubs.surrey.ac.uk/cgi/viewcontent.cgi?article=1011&context=publcomp3.

[20] Napster. Last checked: 2010-10-01. http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p4.html. URL http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p4.html.

[21] M. Ripeanu and I. Foster. Peer-to-peer architecture case study: Gnutella network. In Proceedings of International Conference on Peer-to-peer Computing, volume 101. Sweden: IEEE Computer Press, 2001.

[22] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, Aug 2003. URL http://portal.acm.org/citation.cfm?id=863955.864000.

[23] D. Caromel, A. Costanzo, and C. Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. Parallel Computing, 33(4-5):275–288, 2007.

[24] A Andrzejak and Z Xu. Scalable, efficient range queries for grid information services. Proc. Second IEEE Int'l Conf. on Peer to Peer Computing, Jan 2002. URL http://doi.ieeecomputersociety.org/10.110910.1109/PTP.2002.1046310.

[25] C Schmidt and M Parashar. Flexible information discovery in decentralized distributed systems. Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, page 226, Jan 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.1441&rep=rep1&type=pdf.

[26] S Ratnasamy, J Hellerstein, and S Shenker. Range queries over dhts. IRB-TR-03-009, Jan 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.243.

[27] M. Marzolla, M. Mordacchini, and S. Orlando. Resource discovery in a dynamic grid environment. In Proc. DEXA Workshop, volume 2005, pages 356–360. Citeseer, 2005.

[28] D Spence and T Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. Proceedings of HPDC, Jan 2003. URL http://doi.ieeecomputersociety.org/10.110910.1109/HPDC.2003.1210031.

[29] C Mastroianni, D Talia, and O Verta. A super-peer model for building resource discovery services in grids: Design and simulation analysis. Lecture notes in computer science, 3470:132, Jan 2005. URL http://www.springerlink.com/index/ek5n4jglrjfq8gaj.pdf.

[30] D Thain, T Tannenbaum, and M Livny. Condor and the grid. Grid Computing: Making the Global Infrastructure a Reality, pages 299–335, Jan 2003. URL http://books.google.com/books?hl=en&lr=&id=b4LWXLRBRLsC&oi=fnd&pg=PA299&dq=%2522Condor+and+the+Grid%2522&ots=GRQoCfXaSV&sig=TvJT8gxO14t-g7OZX0j0jYJnc_o.

[31] R Raman, M Livny, and M Solomon. Matchmaking: An extensible framework for distributed resource management. Cluster Computing, Jan 1999. URL http://www.springerlink.com/index/Q864Q3M056803626.pdf.

[32] S Chapin, D Katramatos, and J Karpovich. Resource management in legion. Future Generation Computer Systems, Jan 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.470&rep=rep1&type=pdf.

[33] I Foster and C Kesselman. Globus: A metacomputing infrastructure toolkit. International Journal of High Performance Computing Applications, 11(2):115, Jan 1997. URL http://hpc.sagepub.com/cgi/content/abstract/11/2/115.

[34] K Czajkowski, S Fitzgerald, and I Foster. Grid information services for distributed resource sharing. 10th IEEE International Symposium on High Performance Distributed Computing, page 184, Jan 2001. URL http://doi.ieeecomputersociety.org/10.110910.1109/HPDC.2001.945188.

[35] D Anderson. Boinc: A system for public-resource computing and storage. Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, page 10, Jan 2004. URL http://portal.acm.org/citation.cfm?id=1033223.

[36] D Zhou and V Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, pages 66–73, Jan 2004. URL http://doi.ieeecomputersociety.org/10.1109/CCGrid.2004.1336550.

[37] E Guttman. Service location protocol: Automatic discovery of ip network services. IEEE Internet Computing, Jan 1999. URL http://eprints.kfupm.edu.sa/64710.

[38] P Goering and G Heijenk. Service discovery using bloom filters. Proc. Twelfth Annual Conference of the Advanced School for Computing and Imaging, Belgium, Jan 2006. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.112.6597&rep=rep1&type=pdf.

[39] Qingcong Lv and Qiying Cao. Service discovery using hybrid bloom filters in ad-hoc networks. Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on, pages 1542–1545, 2007.

[40] F Sailhan and V Issarny. Scalable service discovery for manet. Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications, pages 235–244, Jan 2005. URL http://doi.ieeecomputersociety.org/10.1109/PERCOM.2005.36.

[41] Steven Czerwinski, Ben Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An architecture for a secure service discovery service. MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking, Aug 1999. URL http://portal.acm.org/citation.cfm?id=313451.313462.

[42] D. Salomon, G. Motta, and D. Bryant. Data compression: the complete reference. Springer-Verlag New York Inc, 2007.

[43] M Nelson. Lzw data compression. Dr. Dobb's Journal, Jan 1989. URL http://www.dsi.unive.it/~si/docs/nelson89.pdf.

[44] D Huffman. A method for the construction of minimum-redundancy codes. Resonance, Jan 2006. URL http://www.springerlink.com/index/06X3U65887922375.pdf.

[45] J.W. Hunt and M.D. McIlroy. An algorithm for differential file comparison. Computer Science Technical Report, 41, 1976.

[46] Concurrent Versions System. Last checked: 2010-10-01. http://ximbiot.com/cvs/.

[47] A. Tridgell. Efficient algorithms for sorting and synchronization. Doktorarbeit, Australian National University, 1999.

[48] A Muthitacharoen, B Chen, and D Mazieres. A low-bandwidth network file system. Proceedings of the eighteenth ACM symposium on Operating systems principles, pages 174–187, Jan 2001. URL http://portal.acm.org/citation.cfm?id=502052.

[49] J. S. Plank. Erasure codes for storage applications. Tutorial Slides, presented at FAST-2005: 4th Usenix Conference on File and Storage Technologies, http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html, 2005.

[50] Z Zhang and Q Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), pages 330–339, Jan 2002. URL http://doi.ieeecomputersociety.org/10.110910.1109/RELDIS.2002.1180205.

[51] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970. doi: http://doi.acm.org/10.1145/362686.362692.

[52] B Chazelle, J Kilian, R Rubinfeld, and A Tal. The bloomier filter: an efficient data structure for static support lookup tables. page 39, 2004.

[53] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw., 8(3):281–293, 2000. doi: http://dx.doi.org/10.1109/90.851975.

[54] Michael Mitzenmacher. Compressed bloom filters. IEEE/ACM Trans. Netw., 10 (5):604–612, 2002. doi: http://dx.doi.org/10.1109/TNET.2002.803864.

[55] PS Almeida, C Baquero, N Preguiça, and D Hutchison. Scalable bloom filters. Information Processing Letters, 101(6):255–261, 2007.

[56] Sean C Rhea and John Kubiatowicz. Probabilistic location and routing. IEEE INFOCOM, 3:1248–1257, Feb 2002. URL http://citeseer.ist.psu.edu/504898.

[57] PeerSim. Last checked: 2010-10-01. http://peersim.sourceforge.net/.

[58] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. Journal of Experimental Algorithmics (JEA), 14:4–4, 2009.

[59] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. Algorithms–ESA 2006, pages 684–695, 2006.

[60] J. Alveirinho, J. Paiva, J. Leitão, and L. Rodrigues. Flexible and efficient resource location in large-scale systems. In Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware, pages 55–60. ACM, 2010.

[61] JN Silva, P. Ferreira, and L. Veiga. Service and resource discovery in cycle-sharing environments with a utility algebra. In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–11. IEEE, 2010.