# Scalable and Efficient Discovery of Resources, Applications, and Services in P2P Grids

Raoul Gabriel Martins Felix
rf@rfelix.com
Student Number 57693

Instituto Superior Técnico

**Abstract.** With millions of computers connected to the Internet, and more coming online each day, how can we ignore such a massive pool of resources? Through distributed computing, namely resource sharing and discovery, we are able to harness all those resources and computing power. Therefore, the aim of this report is to construct 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.) installed on computers connected to the same Peer-to-Peer Grid network. This will be done in a novel way that consists of combining all resource information into attenuated Bloom filters by following naming rules and using namespaces. The research performed here shows that various systems tackled this problem by focusing on each resource type in isolation. This process consisted of analyzing P2P, Grid, and Cycle Sharing systems that either provide (physical) resource discovery or service discovery. Efficient forms to represent data were also researched in order to minimize storage and transmission costs which impact system efficiency and scalability. This report also presents the performance metrics that will be used to evaluate the system w.r.t. our objectives. Such metrics consist of testing the system by simulating typical usage scenarios, in various network configurations, whilst analyzing the number of messages and hops (among others), along with the effect of fine tuning Bloom filter parameters. Only once the system is implemented and evaluated will we be able to determine if our objectives were met, even though the architecture proposed here was designed with efficiency and scalability in mind.

## 1 Introduction

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]. Bittorrent represents around 50-75% of P2P traffic, which is roughly 25-65% of Internet traffic. 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.

Grid computing has emerged in scientific and commercial communities to perform large-scale computations in a parallel manner. Infrastructures have been built in order to harness the power of many interconnected computers with the objective of performing extremely expensive computations. Normally, these clusters of computers and the network are either centrally or hierarchically managed by the institutions that run them, and can be dispersed around the globe.

Rather than buy clusters of computers, as it is normally done in institutional grids, why not take advantage of the fact that there are already many computers and game consoles connected to the Internet while

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

idle. This is precisely what Cycle Sharing or Cycle Stealing systems do. Large computation intensive tasks are divided into smaller ones and are distributed to computers during idle periods. Results are then collected at a central server for analysis. Users volunteer their computers during idle times to a cause they believe in, such as SETI@Home [7].

In the literature [8–11] 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 [12] (**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.).

The common theme between these different 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 sharing their resources and, in order to do that, the resources need to be discoverable. Not only do they need to be discoverable, but they also need to be matched against the requirements of user tasks. This is where Resource and Service Discovery protocols come in, for without them, these systems would be rendered almost useless for computation. Having a good resource discovery mechanism can make or break a system. Therefore, this paper presents the ongoing work towards an efficient and scalable discovery protocol of resources, applications, and services for possible inclusion in the GINGER project.

The rest of the paper is structured as follows. In Section 2 we present the objectives of this work. Section 3 contains the analysis of the state of the art, describing related works. We then present the proposed architecture to be implemented in Section 4, along with the proposed methodology to be used to evaluate the system in Section 5. Section 6 concludes this paper offering final remarks.

## 2 Objectives

The overall aim of this work is to enhance the resource discovery mechanism present in the GINGER project, making it more complete and decentralized. In other words, this work constitutes the development of a discovery method that not only searches for basic resources such as CPU, memory, and bandwidth, but also for applications, services, and libraries that are installed in each of the nodes that form the P2P Grid. This system should also be scalable and adapt 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 to:

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 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.

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

Objectives 3 and 4 are the bulk of this work. A new discovery mechanism will be implemented in order to: enhance resource discovery in GINGER, making it 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.

Finally, with Objective 5, the system will be evaluated in a simulator and will be compared to other discovery mechanisms. Results will then be analyzed with regards to the satisfaction of aforementioned objectives.

## 3 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 3.1 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 3.2 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 3.3 deals with the many ways data can be stored and transmitted, emphasizing efficiency. Finally, Section 3.4 will conclude the related work analysis and also present an overview of the analyzed works.

### 3.1 Resource Discovery

Resource discovery [13] 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 3.1.1); traditional Grid systems that are not based on Peer-to-Peer models used by scientific and commercial communities (Section 3.1.2); and Cycle Sharing systems used for academic and scientific projects (Section 3.1.3).

#### 3.1.1 Peer-to-Peer

Peer-to-peer systems [8, 13–15] 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.

##### 3.1.1.1 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, 16] 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 [17], iterative deepening [1, 18], probabilistic forwarding [18, 19], learning-based [17, 18, 20], and heuristic-based [2]. Even though the lack of structure in these systems may lead to inefficient searching, it has the advantage of being able to adapt to a very transient node population, in which nodes join and leave at a high rate.

###### 3.1.1.1.1 Napster

Napster [21] 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-node's address that contains the requested song. The client that initiated the search then connects directly to the node with the desired file and starts the transfer.

The central directory server was only used to map users to files and to facilitate the searching of audio files, while the actual transfer was done between nodes (thus considered as being a Peer-to-Peer system). There are a few major problems to using a centralized architecture. The first has to do with scalability issues. As the number of users increase, the directory server becomes a bottleneck. Although, Napster allowed the addition of various directory servers to help alleviate the server, making it more scalable. Another problem

is that the central index server is a single point of failure. By bringing it down, either by a Denial-of-Service attack or by legal measures (as was the case with Napster), the whole system is disrupted and becomes unusable.

#### 3.1.1.1.2 Gnutella

After Napster's demise, Gnutella [1] was the next major P2P network, with 1.18 million computers connected to it as of June 2005[3]. It is a totally decentralized system and a classical example of an unstructured P2P network, where the exchanged resources are 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 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. Thus, nodes closest to the originating node were queried first, and 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. Saturation was a big problem, especially with low capacity nodes as they 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 [22]). Another problem was that the depth $D$ chosen for termination made it impossible for a node to find resources from nodes further than $D$ hops away, so this technique really only works well with popular and well-replicated files.

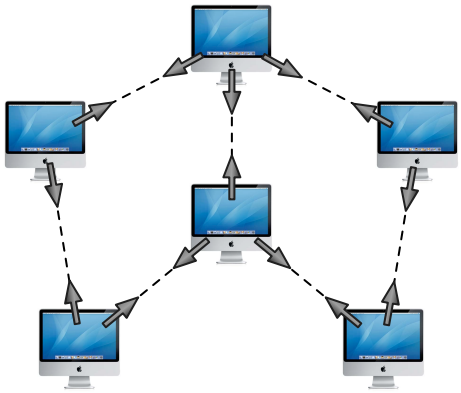

**Fig. 1.** Flooding in Gnutella

#### 3.1.1.1.3 Freenet

Freenet [2] is a third generation[4] 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 normal file sharing applications, as it only allows users to insert files (and not remove them) into the network, which in turn will be replicated to a number of other nodes. This way, the file will still be accessible even if the original node that submitted the file goes offline, thus providing high reliability. Rather than using flooding for searching like Gnutella, making the system unscalable, it is based on 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 [8]: 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.

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

#### 3.1.1.1.4 Iamnitchi et al.

Iamnitchi et al. [17] propose a fully decentralized Peer-to-Peer architecture for Grid Resource Discovery. Participants are called Virtual Organizations (VO) and can be individuals (like 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 that the formation of neighboring nodes is random, as typical in an unstructured system. Four propagation strategies are proposed, with trade-offs between the amount of storage space used in each node and the search performance: **random walk**, **learning-based**, **best-neighbor**, **learning-based + best-neighbor**.

The **random walk** strategy is the simplest: the request is forwarded to a randomly chosen node. 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. 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. 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.

Experimental results showed that the learning-based strategy was the best all-round. It is performance boost is due to the exploitation of similarities between requests by using a possibly large cache. It starts out slowly, but once the cache starts to be filled, performance improves drastically. The more expensive version of the algorithm (learning-based + best-neighbor) turned out to be unpredictable with regards to performance. The best-neighbor strategy performed best in an environment where requests were distributed evenly over participating nodes. Finally, Random walks proved to be the least efficient, but has the advantage of not needing additional storage space in each peer to record historic information.

#### 3.1.1.1.5 Filali et al.

In [16] Filali et al. propose a P2P resource discovery mechanism for Grids, with the goal of improving an existing system described in [23] by addressing the following limitations: 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 and Requester nodes. Grant nodes offer resources that can be used; Requester nodes search for resources and use them. All nodes are also relay nodes. Grant messages received by nodes are stored 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 broadcasted to all neighbors.

Experimental results show that the system is more efficient than basic flooding and random walks with regards to request success rate and network overhead. This is explained by the use of a cache that forwards messages to peers closer to the node with the requested resources, until it is eventually found.

#### 3.1.1.1.6 Liu et al.

Liu et al. [20] 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, which is basically a cache that uses a Least-Recently-Used policy. By using a knowledge index, 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 searches are split into two phases and are performed in the following way. First, the node looks at its own knowledge index for peers directly related to the search topic in question. This phase has a high probability of failing due to lack of information, especially for recently connected peers. When the first phase fails, the second phase searches for nodes that are sharing content in the same interest area (found from the Open Directory Categories[5]) of the query from the knowledge index. In case of failure, the request is forwarded randomly to a node.

[5] http://dmoz.org

The knowledge index is a vital part of the system in terms of performance. It does not require any significant processing overhead as the node just observes the successes and failures of searches, and updates the knowledge index accordingly. Plus it can also reduce network traffic as the routing algorithm tries to leverage the knowledge index as much as possible. Although, these benefits come with the cost of additional memory usage.

#### 3.1.1.2 Structured

Structured systems address the 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.

##### 3.1.1.2.1 Chord

Chord [3] is the first structured Peer-to-Peer system to be proposed by Stoica et al. It provides a routing and location infrastructure, so it is not a resource discovery system per se, but it can be used to implement one. File identifiers (a.k.a. Keys) are mapped onto node identifiers. Files and peers are mapped with the same hash function to a $m$-bit key space. Data location can be implemented on top of Chord by identifying data items as keys and storing the pair (key, data item) at the node whose identifier maps to the data item key. Nodes in Chord are ordered in a circle and packets can only be forwarded in one direction: clockwise. A key $k$ is assigned to the node whose identifier is bigger or equal to $k$ in the identifier space. For routing to work, peers are connected to (and have knowledge of) their successor and predecessor in the ring. In its most basic form, routing between Chord nodes can be performed by forwarding messages to their successor until the requested key is found, but is highly inefficient. To speed up the key lookup process, each node uses a finger table with $m$ entries, which maintains a connection, for node $n$, 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 2a. This lookup process emulates a binary search, thus requiring only $\mathcal{O}(\log N)$ messages and steps (Figure 2b). Periodic stabilization messages are used to maintain Chord's rigid structure. When a node joins the network, it removes the keys it is responsible for from its successor. When it leaves, the node's successor becomes responsible for its keys. For robustness, each node keeps $r$ successors: if a successor does not respond, it can be substituted for on in the successor list. Therefore, as long as the nodes in the successor list do not fail simultaneously, the Chord network will not be disrupted. In theory, each peer is responsible for an equal number of keys with a high probability due to the use of consistent hashing, thus achieving load balancing, with regards to the number of files stored at each node.

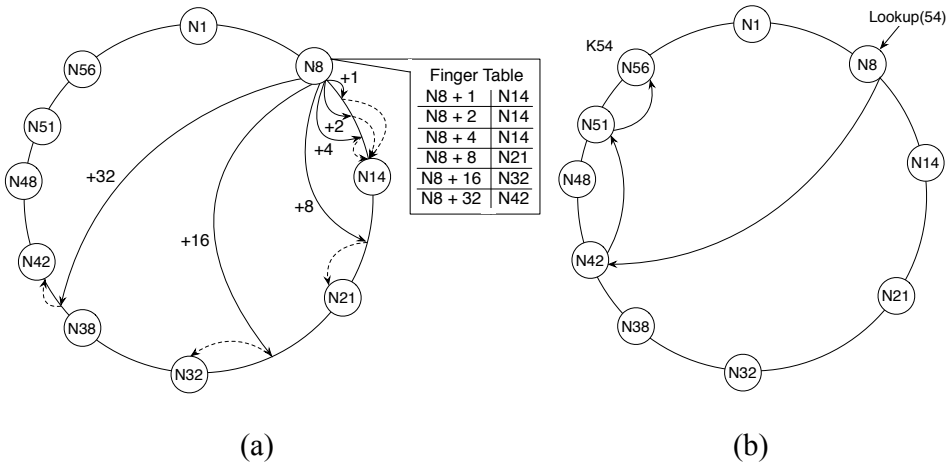


**Fig. 2.** (a) Example finger table for node 8. (b) Routing of query for key 54 using the finger table to speedup lookup.

#### 3.1.1.2.2 CAN

Content Addressable Network (CAN) [5] is another routing and location infrastructure. It uses a virtual $d$-dimensional coordinate 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 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 3. When a node joins the network, it randomly chooses a coordinate space and joins the node covering it. That zone is then split in half and the surrounding nodes are notified of the new node. On departure, the zone and entries are explicitly handed over to one of the neighbors. If any neighboring node fails, CAN initiates a controlled take over mechanism. If many nodes of a failed node also fail, then an expanding ring search mechanism is used to identify any functioning nodes outside the failure region. Thus, peer arrivals and departures have a localized effect as only $\mathcal{O}(d)$ other peers are affected. As is, CAN does not support replication, but it can easily be added by using more than one hash function, which will reduce the lookup cost and provide fault tolerance in case of ungraceful departures.

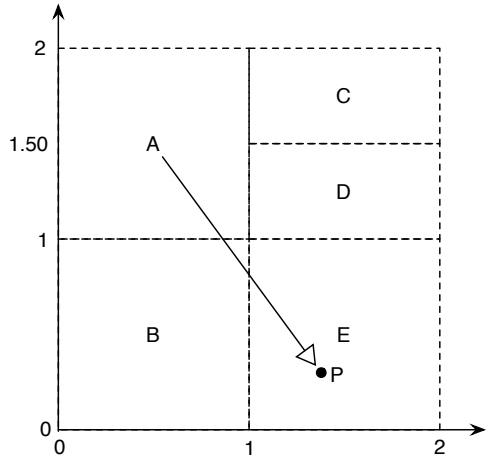


**Fig. 3.** Example CAN coordinate space of [0,2]x[0,2]. Nodes B, C, and D are node A's neighbors. Query for key that maps to (1.3, 0.3), starting from node A, is routed first through B and then finally to E, which is responsible for that zone.

#### 3.1.1.2.3 Andrzejak and Xu

Andrzejak and Xu [24] propose an extension of CAN that allows it to support range attribute queries. The system is intended for usage in a Grid environment where resources tend to be highly dynamic and queries usually need to specify ranges. Each resource has a set of attributes and depending on the attribute's type, the system can either use a standard DHT or the CAN extension. The former is used when attribute has a limited number of values, while the latter is used when attributes have continuous values. Location of multi-attribute requests is performed by consulting the appropriate DHTs and then concatenating the results at the end. Only a subset of nodes in the Grid system participate in the extended CAN network. Each of those nodes are called an Interval Keeper (IK) and are responsible for a sub-interval of the attribute's value. This sub-interval of the attribute's value is the interval of the IK. Each server in the Grid reports values to the IK with the corresponding interval. An interesting property of this extension is that if a range is split into two sub-ranges, then those zones together form the primary range. This also means that nearby ranges translate to nearby zones. 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. These strategies were tested using simulations of synthetic and real-life workloads. The results show they were effective in meeting the system's goals: scalability, availability, and communication-efficiency.

#### 3.1.1.2.4 Schmidt et al.

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. Notice that ranges can only start from powers of two; same goes with range query sizes. Requests are forwarded to nodes with an ID that has a larger common prefix with the query than the current node. Thus, for example, the 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. Another 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.

#### 3.1.1.2.5 Ratnasamy et al.

Ratnasamy et al. [26] describe a distributed data structure that supports range queries over DHTs, called a Prefix Hash Tree (PHT). As locality between ranges are not maintained, another overlay is used on top of a Distributed Hash Table to allow efficient range query resolution. This system is agnostic to the underlying DHT routing algorithm. 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 as 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. Results are then merged at the node where the query originated from.

#### 3.1.1.2.6 Marzolla et al.

Marzolla et al. [27] describe a system based on routing indexes for Grid resource discovery. Nodes are organized into a tree-structured overlay. 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 the one corresponding to the sub interval that contains the actual value of the attribute in question, as can be seen in Figure 4 with nodes B and C. To obtain a compact representation of resources for a subtree, one need only apply the bitwise `OR` operator on all bitmaps belonging to the same attribute, local to each child node (node A in Figure 4).

Multi-attribute queries are handled by dividing them into separate sub-queries: one for each attribute. They are then first matched against the local indexes, and then against the routing indexes that contains the information about other nodes. The query is then forwarded to the neighbor whose bitmap indexes satisfy all sub-queries.

As the objective of this system is to perform resource discovery in a Grid environment, it has to address the issue of attribute values changing over time due to resource utilization. Thus, the bitmap indexes need to be updated overtime, although care must be taken as to not inundate the network with update messages. Therefore, bitmap indexes are calculated periodically as resources consumption changes. Only if the index changes from the previous one are the neighboring nodes notified of the alteration.

Simulation results show that the system scales well. This is due to two reasons: query propagation only routes messages to a small number of neighboring nodes, effectively avoiding flooding; and the update method involves a constant number of peers, regardless of network size.

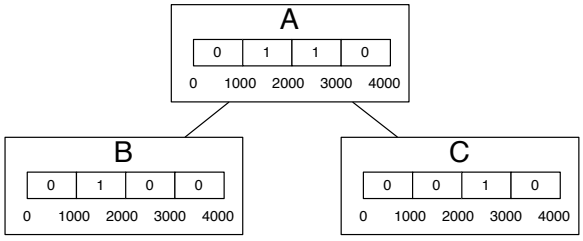


**Fig. 4.** Example bitmap indexes, as used in [27], to represent CPU Speed (in MHz) in nodes B and C, along with aggregated information (bitwise `OR`) in parent node A.

#### 3.1.1.3 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 (Section 3.1.1.3.1) and Kademlia (Section 3.1.1.3.2) 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 (Section 3.1.1.2.1) and CAN (Section 3.1.1.2.2). 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.

##### 3.1.1.3.1 Pastry

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 described in Section 3.1.1.2.1). 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. Message 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 the node that is numerically closer to the key than the

local node. For robustness, a set of $k$ nodes with nodeIds numerically closest to a key contain a replica of the (key,value) pair. Notice that this means replicas will have sequential IDs; no *salt* was used to disperse the replicas among the ID space. Not only does this increase reliability, but also minimizes the distance the message travels, as the replica that is closest to the local node is chosen first.

#### 3.1.1.3.2 Kademlia

Kademlia [6] is another storage and lookup infrastructure. Its topology is organized using a XOR metric. Key and nodeIds are assigned in the same 160-bit space. (Key,value) pairs are stored in $k$ nodes whose IDs are closest to the key, where distance is measured using the XOR metric ($distance = key \oplus nodeId$). Most of Kademlia's benefits are from using this metric, as it is symmetric. 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. To support the routing process, 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 $\alpha$ 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 $\alpha$ 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. $\alpha$ 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.

#### 3.1.1.3.3 Mastroianni et al.

Mastroianni et al. [28] propose a resource discovery system in a Grid environment that is based on the 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, forming 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.

#### 3.1.1.3.4 XenoSearch

XenoSearch [29] is based on Pastry's index and routing system and offers support for multi-attribute queries and range queries, while only being a factor of 3-5 slow than plain Pastry routing. 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. They 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.

#### 3.1.2 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 supercomputer, are sometimes referred to as metacomputers, while a cluster of these metacomputers are usually referred to as Virtual Organizations (VOs).

##### 3.1.2.1 Condor

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.

##### 3.1.2.2 Globus MDS-2

Globus [32] is a toolkit that provides an infrastructure to create Grid systems that exploit diverse geographically distributed resources in order to form *networked virtual supercomputers* or *metacomputers*. MDS-2 [33] (Meta Directory Service) is a resource discovery mechanism that can be used in Globus. 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. These two basic 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.

#### 3.1.2.3 Legion

Legion [34] 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. They also allow the future reservation of services for scheduling purposes. Jobs are represented as Objects with a set of requirements. The resource discovery involves a number of components, such as the Host Objects, the actual resource Objects, the Collection, the Scheduler, and the Enactor; and proceeds as follows. 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.

### 3.1.3 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.

#### 3.1.3.1 BOINC

Berkely Open Infrastructure for Network Computing [35] 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.

#### 3.1.3.2 CCOF

In Cluster Computing on the Fly [36], the authors conducted a comprehensive study of resource searching methods in a highly dynamic P2P environment for locating idle cycles to be consumed by workpile 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. The problem in cycle sharing systems is that the number of users can be large and consist of a highly dynamic population. Not only do peers join and leave the network unpredictably, but the amount of variable CPU cycles change at an extremely rapid rate. CCOF is different to BOINC in that it 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 scalable search methods: expanding ring, random walks, advertisement-based, and rendezvous point.

In 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.

Random Walk search consists of sending the query to $k$-random neighbors, which in turn forward the query to another $k$-random neighbors. The candidate list is then returned to the client.

Advertisement-based search 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 caches 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.

Finally, in the Rendezvous Point search, groups of peers are dynamically selected as Rendezvous Points in the system 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.

Simulation results showed that the Rendezvous Point method, compared to the other methods, performed better under both light and heavy workload conditions. In light workloads, the advertisement-based approach incurred a high message passing overhead, while in heavy workloads all algorithms showed a significant increase in message overhead, except Rendezvous Point search which was consistently low.

### 3.2 Service Discovery Protocols

Service discovery [13] 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.

#### 3.2.1 SLP

The Service Location Protocol (SLP) [37] is one of the first well known service discovery systems. It is also a classical example of a centralized system, capable of functioning from LANs to large enterprise-scale networks. SLP can operate in two different modes: one that can only be used in small networks and another that can handle a large number of nodes. The first approach is not centralized as messages between nodes are exchanged via multicast, which is why it cannot be used in a large network due to message flooding. The second approach is centralized and uses directory agents to handle a large number of queries. Nodes can assume three different roles: user agents (UAs), service agents (SAs), and directory agents (DAs). Multiple roles can be combined into a single node if need be. The SA provides services in the network and advertises them; the DA collects service advertisements and indexes them; and the UA consumes the services provided by the SAs and can query the DA for new services. Directory agent discovery can be performed either passively, by detecting multicast advertisements, or actively, by sending SLP requests. If a DA node is present, then UAs contact them directly via unicast; if not, UAs use multicast to query SAs for services.

### 3.2.2 Jini

Sun Microsystem's Jini framework [38] provides a platform and protocol independent service discovery system that relies on the Java Virtual Machine (JVM), leveraging Java's uniformity across platforms. It 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. The Jini discovery architecture is similar to that of SLP (discussed in Section 3.2.1) and also uses a centralized approach. The directory agent equivalent from SLP is the lookup server, which collects service advertisements and searches for specific services 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.

### 3.2.3 Goering et al.

Goering et al. [39] propose a service discovery protocol for local ad-hoc networks based on the use of attenuated Bloom filters. Bloom filters, discussed in more detail in Section 3.3.4, 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.

### 3.2.4 Lv and Cao

Lv and Cao [40] propose another service discovery protocol based on Bloom Filters, but address the drawback of the system proposed by Goering et al., where services cannot be located further than $d$-hops away. The system first tries to use the service discovery method based on Attentuated Bloom Filters, discussed in Section 3.2.3, as it has the advantage of being able to find local services efficiently. When no service can be found $d$-hops away, the system uses another service discovery method proposed by Sailhan and Issarny [41] (under Global Service Discovery) where nodes that are $d$-hops away need to cooperate amongst themselves, possibly relying on nodes acting as gateways to bridge with nodes more than $d$-hops away. Therefore, service discovery in this system proceeds as follows. 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.

### 3.3 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.

#### 3.3.1 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: dictionary based methods and statistical based methods.

The LZW [43] method is an example of a dictionary based approach to compression, whose main feature is eliminating the second field of a token. 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 another data compression algorithm, but uses a statistical based approach rather than a dictionary. 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.

#### 3.3.2 Chunks and Hashing

Transferring large files over a network can consume a lot of time and bandwidth. In such cases, there are systems that can 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.

Rsync [45] synchronizes files and directories from one location to another over the network while minimizing data transfer by exploiting commonality between files. This technique is usually referred to as delta encoding and it consists of storing and transmitting data in the form of differences between data. 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 block 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.

The `diff` [46] Unix utility also operates on differences between files. It takes two versions of the same files and calculates the difference between them. The program output can then be used by the `patch` utility

to transform one file into another. The CVS [47] version control system is another system that uses this delta encoding technique to bring a users working directory tree up to date.

LBFS [48] is a Low-Bandwidth Network File System that saves bandwidth by exploiting not only the similarities between different versions of the same file, but also by 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.

### 3.3.3 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.

[50] describes a replication protocol, called Reperasure, for a peer-to-peer storage system with the primary objective of ensuring data availability and, secondarily, to speed up the access of data from many clients. Although, the authors are only interested in P2P systems that will guarantee the retrieval of an existing object, such as Chord (Section 3.1.1.2.1), CAN (Section 3.1.1.2.2), Pastry (Section 3.1.1.3.1), or any other DHT. They also assume nodes belong to a well-defined administration domain, unlike other systems such as Gnutella (Section 3.1.1.1.2) and Freenet (Section 3.1.1.1.3), where nodes are volunteered from owners. Therefore, system dynamism assumed is not as dramatic, but they still consider the fact that nodes can fail unexpectedly. 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.

### 3.3.4 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. Or, 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 that are able 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** (already discussed in Section 3.2.3) 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.

### 3.4 Concluding Remarks

In this section we discussed the state of the art of Peer-to-Peer (Table 1), Grid, and Cycle Sharing systems (Table 2) that perform resource discovery. We also analyzed a few service discovery protocols (Table 3) and various forms to represent data in an efficient manner (Table 4).

Notice that many of the systems discussed in the P2P Resource Discovery subsection are related to resource discovery in Grid environments. This reinforces the idea presented in the Introduction 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 [12] 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 location of physical resources, but also the 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 3.1 only handled the discovery of physical computer resources and files, while systems in Section 3.2 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 proposed in this report will do.

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 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 a false positive 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 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.** Overview of Grid and Cycle Sharing Systems

## 4 Proposed Architecture

The objective of this work is to enhance the resource discovery mechanism in GINGER [12] (**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

| 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 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 4.** Summary of Efficient Data Representation techniques

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.

This Section, therefore, contains my proposal for a resource, application, and service discovery mechanism and is divided as follows. Section 4.1 presents the context in which the architecture I propose here should be taken into. Section 4.2 will present an overview of the proposed discovery mechanism, along with a description of how it will function. Finally, naming conventions and rules used for resources, applications, and services are discussed in Section 4.4, along with how resource insertion and querying will be performed.

### 4.1 General Overview

In its most abstract form, the Ginger [12] project can be thought of as a system where a user can submit a job and then later on retrieve those results, as can be seen in Figure 5. 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 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 will provide a mechanism to search for computers, connected in a P2P network, that satisfy the requirements needed to perform a specified task.

### 4.2 Architecture Overview

The discovery mechanism I propose in this work takes the form of an unstructured peer-to-peer network, i.e. nodes are randomly connected to a fixed number of other nodes. Using an unstructured P2P model permits us to handle a very dynamic peer population with high churn rates, but it also means that messages will not be routed as efficiently as in, say, a structured system. To combat this limitation, I also propose the usage of attenuated Bloom filters in order to speed up resource location. Bloom filters were chosen because they allow us to efficiently perform membership tests in a space efficient manner, plus, the effect of false positives can be mitigated by requiring an extra step for confirmation.

The proposed resource discovery mechanism will work as follows. 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

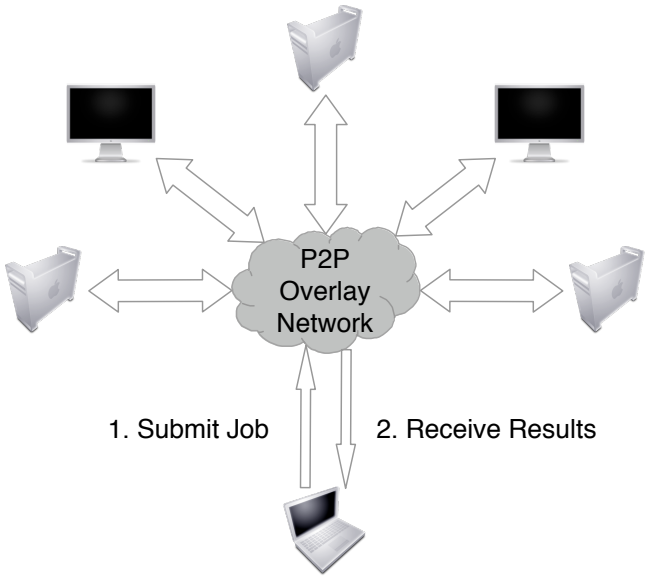


**Fig. 5.** General overview of a system where resource discovery is necessary. This image represents the Ginger architecture in its most abstract form.

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 6, 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 4.2.1.

### 4.2.1 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. Therefore, I propose a random walk strategy where a peer discovery query is sent randomly to $w$ nodes, in search for nodes that are $l$ hops away. When the query reaches nodes $l$ hops away, they reply directly to the originating node providing enough information to be contacted (e.g. IP address). The pseudo-code for this procedure can be seen in Algorithm 1.

**Algorithm 1** Pseudo-code to be executed when a Peer Discovery Query is received

```
this_query.l ← this_query.l − 1
if this_query.l ≠ 0 then
    random_nodes ← select this_query.w random nodes from neighbor list
    for each random_nodes as node do
        node.send(this_query)
    end for
else
    this_query.originator.send(IP)
end if
```

For example, in Figure 6, 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 some nodes more than $d + 1$ hops away in order to restart the query at one of those nodes. To do this, assuming $w = 2$ and $l = d + 1 = 3$, node A sends a peer discovery query (blue dashed lines with arrows) to $w = 2$ randomly selected nodes: C and D. When they get the query, they decrement $l$ by one and check if it is 0. This is not the case, so they randomly select another $w$ neighbors and resend the query. In node C's case, the nodes that were chosen are nodes `I` and `II` (which are 2 hops away from A). This continues until nodes receive a peer discovery query with $l = 1$, which in Figure 6 are nodes 1, 2, 3, 4, and 6. After decrementing $l$ and verifying $l = 0$, according to Algorithm 1, the nodes will send a reply with contact information directly to the node that originated the query (green dotted lines with hollow triangle tip): node A. Now that node A knows that nodes 1, 3, 4,

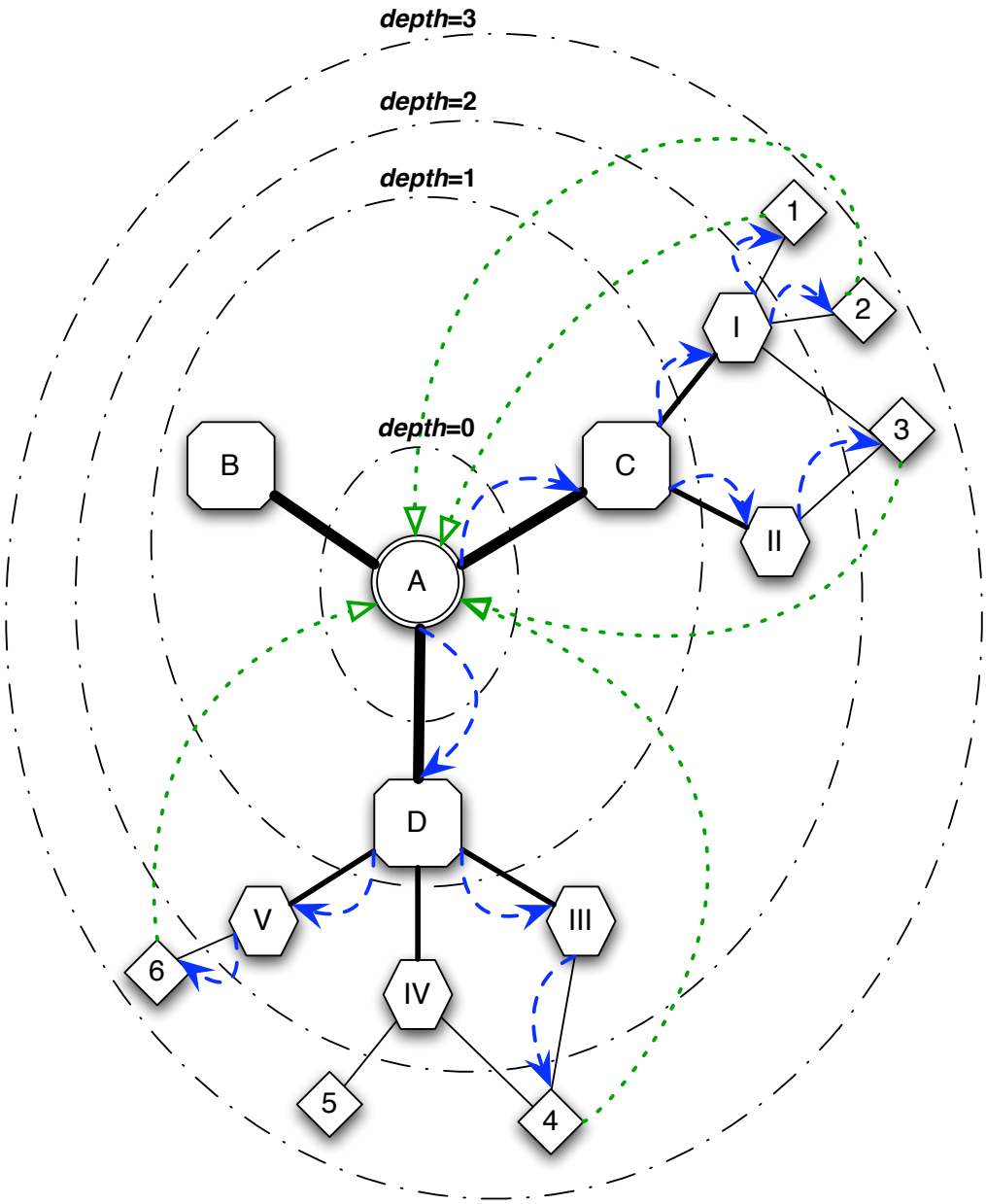


**Fig. 6.** Example architecture overview of network from node A's perspective. Nodes are enclosed in circles with increasing *depth*, meaning they are *depth* hops away from node A. To help distinguish between nodes at different distances the following visual aids were used: connection line thickness 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 4.2.1. Green dotted lines with hollow triangles represent peers responding node A's peer discovery query.

5, and 6 are more than $d = 2$ hops away, it can restart a query at any one of them when its requirements cannot be satisfied with peers up to $d$ hops.

### 4.3 Resource, Service, and Application Discovery

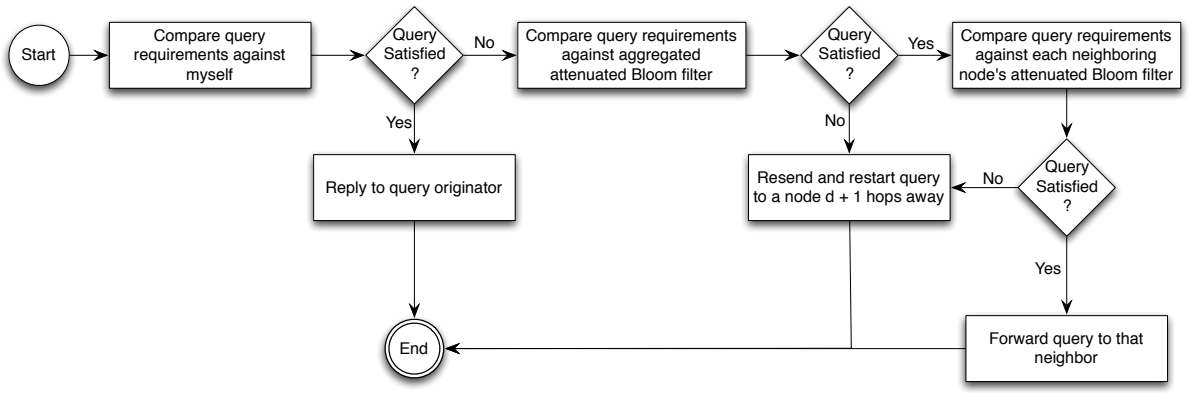


**Fig. 7.** Flowchart illustrating resource, service, and application discovery explained in Section 4.3

The discovery of resources, applications, and services (illustrated as a flowchart in Figure 7) will be 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 6, 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.

### 4.4 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.

Therefore, I propose a naming convention for the keys that are inserted into a Bloom filter. It will use namespaces to differentiate between resources and their values, which will also help with the searching of resources (discussed in Section 4.4.1). The naming convention will use a 3-level namespace, each separated using the colon (":") as a delimiter, and will follow 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 will allow the names of these different resources to be specified in a configuration file, and we will assume that all nodes that take part in the system use the same configuration files so as to use the same names.

### 4.4.1 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 5b. 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 5.** Example (used in Section 4.4.1) showing the keys that need to be used when inserting the CPU resource values into a Bloom filter.

## 5 Proposed Evaluation Methodology

The system presented in this report will be evaluated in a simulator, namely PeerSim [57]. The objective of this work is to create not only an efficient resource, service, and application discovery system for a Peer-to-Peer Grid, but also a scalable one.

To evaluate the system I propose simulating the discovery of resources, services, and applications of typical application usage scenarios, namely a ray tracing execution, video transcoding, and, possibly, Monte Carlo simulations. Each simulation will be in function of input size and the number of gridlets, which will influence the number of queries with requirements for the system to resolve. For each execution the following metrics will be analysed:

- Percentage of satisfied requests;
- Total number and size of messages sent;
- Average number of hops per query;
- Time taken to resolve a query;
- Size of data each node stores in order to perform resource discovery;
- Number of failures due to peer disconnects or unavailability of resources/services.

These tests will be performed using different overlay dimensions, from a small overlay of less than a hundred nodes to a large one with thousands of nodes. The number of nodes that are able to actually satisfy requests will also vary.

Due to the fact that the architecture presented in this report makes use of Bloom filters, which will affect the efficiency of resource discovery, I also propose to analyze the influence of the false positive rate and the effect it has on Bloom filter size used for storage and transmission, the effect false positives have on resource discovery efficiency and number of hops, and how the depth of attenuated Bloom filters affects resource discovery.

## 6 Conclusion

The number of users connected to the Internet keeps on growing. All these interconnected computer resources can be combined to provide huge amounts of computing power. Nowadays, with the evolution of computers and software, average home users start requiring and wishing for better and faster computers. With GiGi [12], these home users can take advantage of Grid computing, which before was only available to scientific and corporate communities. Tasks that would usually take a lot of 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 computers. However, for GiGi to do this, it needs to be able to locate computer resources that are able to satisfy task prerequisites.

Therefore, this report presents an architecture for a scalable discovery mechanism that will not only be able to locate physical resources, but also services and applications installed in computers connected to a P2P Grid, to be included in the GiGi project. The overall aim of this work is to create a decentralized discovery mechanism, capable of discovering a variety of resources and services in an efficient and scalable manner. Various performance metrics have been proposed to evaluate the system in order to reach a conclusion with regards to efficiency and scalability. This report also presents an analysis of previous discovery methods present in P2P, Grid, and Cycle Sharing systems, along with various forms to represent information that will be stored in, and transmitted by, each node in the network.

The architecture presented in this work was created taking into account the various objectives we wish to accomplish with the system. Only once this discovery mechanism has been implemented and evaluated will we be able to offer concluding remarks with regards to the satisfaction of our objectives.

## References

1. Gnutella Protocol Specification. Last checked: 2009-12-18. 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.
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.
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.
7. D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):61, 2002.
8. S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, Jan 2004.
9. 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.
10. D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7:96–96, 2003.

11. A. Iamnitchi and D. Talia. P2p computing and interaction with grids. *Future Generation Computer Systems*, 21(3):331–332, 2005.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. B Yang and H Garcia-Molina. Efficient search in peer-to-peer networks. 2002.
19. Vana Kalogeraki, Dimitrios Gunopulos, and D Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. pages 300–307, 2002.
20. 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.
21. Napster. Last checked: 2009-11-13. http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p4.html.
22. 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.
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.
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.
26. S Ratnasamy, J Hellerstein, and S Shenker. Range queries over dhts. *IRB-TR-03-009*, Jan 2003.
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. 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.
29. D Spence and T Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. *Proceedings of HPDC*, Jan 2003.
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.
31. R Raman, M Livny, and M Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, Jan 1999.
32. I Foster and C Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115, Jan 1997.
33. 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.
34. S Chapin, D Katramatos, and J Karpovich. Resource management in legion. *Future Generation Computer Systems*, Jan 1999.
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.
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.
37. E Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, Jan 1999.
38. J Waldo. The jini architecture for network-centric computing. *Communications of the ACM*, Jan 1999.
39. 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.
40. 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.
41. 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.
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.
44. D Huffman. A method for the construction of minimum-redundancy codes. *Resonance*, Jan 2006.
45. A. Tridgell. Efficient algorithms for sorting and synchronization. *Doktorarbeit, Australian National University*, 1999.
46. J.W. Hunt and M.D. McIlroy. An algorithm for differential file comparison. *Computer Science Technical Report*, 41, 1976.
47. Concurrent Versions System. Last checked: 2009-12-27. http://ximbiot.com/cvs/.
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.
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.
51. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
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.
54. Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
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. 2002.
57. PeerSim. Last checked: 2009-12-27. http://peersim.sourceforge.net/.