

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

Raoul Felix, Paulo Ferreira, and Luís Veiga

INESC ID /IST
Rua Alves Redol 9
Portugal

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

1 Introduction

There are millions of computers connected to the Internet¹ with more and more going online each day due to laptops, netbooks, PDAs, and smartphones. With so many devices connected to the same network, distributed computing on such a large scale cannot be ignored. As such, resource sharing has become immensely popular and has led to the development of Grid and Peer-to-Peer (P2P) infrastructures dedicated to that purpose. These infrastructures ease the sharing of various types of resources, that range from simple files, to software offering different services, and even hardware like CPUs and Printers.

The most popular form of resource sharing across the Internet is File Sharing via Peer-to-Peer applications, occupying roughly 50%-90% of all Internet traffic.² A lot of work has been done in this area to create robust and scalable systems, capable of efficiently supporting a large number of users in a decentralized manner. P2P Infrastructures can be divided between those that do not perform any node organization (Unstructured systems), such as Gnutella [1] and Freenet [2]; and those that structure their nodes to improve message routing (Structured systems), such as Chord [3], CAN [4], and Pastry [5].

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds. Raoul Felix and this work were supported by FCT research project GINGER - PTDC/EIA/73240/2006

¹ <http://www.internetworldstats.com>

² <http://torrentfreak.com/bittorrent-dominates-internet-traffic-070901>

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

Even though Peer-to-Peer and Grid systems are different, the literature [6–9] says that they will eventually converge. In this fashion, GINGER³ [10], 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. Tasks require discoverable resources that satisfy certain requirements that 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). This is where the work described in this paper comes in, where the objective is to create an effective, efficient, and scalable discovery protocol of resources, applications, and services for inclusion in the GINGER project.

The rest of this work is structured as follows. In Section 2 we discuss similar systems that also provide service or resource discovery. Section 3 describes the architecture of SERD⁴, while in Section 5 we show some relevant performance results. Section 6 concludes this paper offering final remarks.

2 Related Work

This section can be divided into three main areas: i) efficient data representation where reducing the size of data storage and transmission is the objective, ii) resource discovery which only deals with the discovery of physical (e.g. CPU, RAM, etc.) or virtual (e.g. files) resources, and iii) service discovery where the main concern is discovering the services (e.g. facial recognition, high-resolution rendering, applications, etc.) provided by computers in a network.

2.1 Efficient Data Representation

Efficient Data Representation is important in this work because nodes have to store and transmit resource information about themselves and neighbors. **Compression** reduces the size of highly redundant information via a dictionary based (LZW [11]) or statistic based (Huffman coding [12]) encoding process. RSync [13] and the Low-Bandwidth File System [14] use **Chunks and Hashing** to divide data into chunks, calculating the hash of each chunk, and only transmitting those that have changed between versions of the same file. **Erasure codes** take another approach, and encode a message into a few symbols which can then be used to reconstruct a partially received message. Reperasure [15] uses this technique

³ Grid Infrastructure for Non-Grid EnviRonments

⁴ Scalable and Efficient Resource Discovery

to provide data replication without storing full-replicas. The three techniques, although important, are not directly applicable in this work. The reduced message size cancels the need to compress messages or divide them into chunks. We also do not need to perform any forward error correction nor replicate data.

The final and most useful technique is a space-efficient probabilistic data structure called **Bloom Filters**, which efficiently test whether an element is a member in a set with the possibility of a false-positive occurring. A set $S = \{x_1, x_2, \dots, x_n\}$ 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 maps 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 computed 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 has 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.

Bloom Filter variations exist to either extend their functionality or address some limitation. **Counting Bloom Filters** [16] allow both insertion and removal of elements by using an array of counters, instead of bits. In [17], Mitzenmacher shows that **Compressed Bloom Filters** can either occupy the same space but have a lower false-positive rate, or reduce their size and maintain their false-positive rate. Almeida et al. [18] created a **Scalable Bloom Filter** that dynamically grows in order to support the desired false-positive rate.

Finally, **Attenuated Bloom filters** were proposed in [19] to optimize search performance w.r.t. locality of objects. 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, 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 less so the further away. The major advantage of this technique is that it permits us to efficiently locate objects up to d hops away, using as little storage space as possible (due to the Bloom filters) at the cost of a certain false positive rate. The disadvantage is that it *only* lets us search information about nodes up to d hops away.

2.2 Resource Discovery

Resource Discovery systems do a subset of what we want to accomplish with this work: locating physical or virtual resources to perform jobs. They can be split into three categories: Peer-to-Peer, Grid, and Cycle Sharing.

Peer-to-Peer systems do not distinguish between clients and servers; all nodes are equal and have no central coordination, making them decentralized. This leads to the various types of node topology organization: unstructured, structured, and hybrid. **Unstructured** system nodes are randomly connected to a fixed number of neighbors; there is no information about where resources

are located so message routing has to be performed by flooding. Searching can be uninformed or informed. Uninformed searches use no additional information to route queries, they are either flooded to all neighbors (Gnutella [1]), or are forwarded to a randomly selected neighbor (Iamnitchi et al. [20]). Informed searches are more intelligent and route messages based on collected information, but require more memory. Lie et al. [21] and the learning-based technique in Iamnitchi et al. forward queries to nodes that have replied to similar requests. Another strategy called best-neighbor in Iamnitchi et al. [20] just forwards queries to nodes with the highest success rate. **Structured** systems, such as Chord [3] and CAN [4] organize nodes into a rigid structure, called a Distributed Hash Table (DHT), which enables efficient exact-match query routing. Each node is assigned an identifier (key) which makes him responsible for all content (values) whose hash resolves to that key. Finally, **Hybrid** systems try to combine the best of both worlds without their disadvantages. Some systems in this category, like Pastry [5] and Kademlia [22], tend more towards structured systems, albeit with a less “rigid” structure, where any node belonging to a defined key subspace can act as a contact for those values. Others follow a more unstructured approach and use super-peers [23] that communicate between themselves on the behalf of less capable nodes (in terms of bandwidth or CPU performance), thus increasing routing performance.

Grid and Cycle Sharing systems share the same objective: to combine many geographically dispersed computer resources in order to perform tasks that require lots of CPU processing power, or that need to process huge amounts of data. Tasks like these are common when dealing with scientific, technical, or business problems. Grid systems can run in LAN environments such as that of a university, or in a much larger network comprised of interconnected networks that belong to different institutions, corporations, or universities. Condor [24] and Legion [25] are typical examples of such systems, where information about all resources are stored in a central component, known as the Matchmaker in Condor, and in Legion is divided into 3 subcomponents: the Collection, Scheduler, and Enactor. This central component receives job requests, tries to match their requirements to available resources, and reserve those resources while notifying the requester. Cycle Sharing systems rather operate over the Internet, which can be highly unreliable with variable connection quality. Another important difference is that anyone with a computer can join a cycle sharing project of interest (e.g. SETI@Home [26] or Folding@Home) and volunteer their resources during idle times. This brings the additional problem of unreliable peer connections and possibly forged results from untrusted peers.

2.3 Service Discovery

Service Discovery systems, like Resource Discovery, do the missing subset of this work: enabling the automatic detection of services provided by computers in small LAN environments, like home networks, or in large-scale enterprise networks, like a corporation or university. SLP [27] and Jini [28] use a client/server architecture, where servers collect service information and perform lookups for clients. SLP can function without directory servers using multicast to find services, but only in small LAN environments.

The systems presented by Goering et al. [29] and Lv and Cao [30] use a Peer-to-Peer architecture instead, with the objective of being able to function in ad-hoc networks. Goering et al. propose a service discovery protocol based on the use of 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, which is consulted when a query is received in order to send it in a direction it will have a higher chance of success. 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 the chance of false positives. Relying solely on Attenuated Bloom Filters gives this system a big limitation: only the services located up to d -hops away can be easily found. Lv and Cao resolve this drawback by having nodes more than $d + 1$ hops away cooperate among themselves. Thus, when a query is received, it follows the same process of checking the Attenuated Bloom Filters of its neighbors like Goering et al, but if no services are found, then the query is forwarded to a node $d + 1$ hops away where the search begins again.

3 Architecture

The objective of this work is to enhance the resource discovery mechanism in GINGER [10], 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 (e.g. video encoders, simulators, etc.) and services (e.g. face recognition, high-res rendering, etc.). Because GiGi can be used in many different ways ("grid-for-the-masses"), it has to be flexible enough to run different types of jobs normally performed by home-users.

In order to cope with a dynamic peer population and high churn rate, this system uses an unstructured peer-to-peer approach to resource discovery, even though message routing may not have optimum efficiency. If a structured system were to be used, the messages needed to keep the structure intact with an unstable population, such as home-users, could possibly result in a high overhead. Attenuated Bloom Filters are used to enhance message routing and speed up resource location. Note that this solution is different to the systems mentioned in the Related Work because it combines all types of different resources into one discovery mechanism. It is especially different to the works [29,30] that also make use of Attenuated Bloom Filters due to usage of one aggregated Attenuated Bloom Filter (explained next), and the fact that all the different types of basic resources, services, and applications are encoded in the Bloom Filter.

Each node in the network stores a cached version of the Attenuated Bloom Filters of their neighbors. This information is then merged into one single Attenuated Bloom Filter by inserting the union (OR operation) of all neighbor Bloom Filters at a certain depth k into depth $k + 1$ (Figure 1). The consequence of using an Attenuated Bloom Filter of, for example, depth $d = 2$ is that a node will only know about the resources of nodes up to 2 hops away. A solution for this problem is discussed further in Section 3.

Discovery Mechanism: The discovery of resources, applications, and services (illustrated as a flowchart in Figure 2) 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 $d+1$

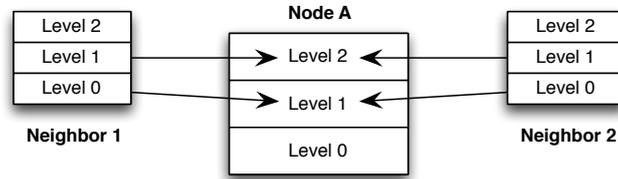


Fig. 1. Example of a node A creating a single Attenuated Bloom Filter by merging each Level i of its neighbors' Attenuated Bloom Filters into Level $i + 1$.

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 for it to 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. As each message is forwarded to a node, the sender adds his own ID to the resource query's Bloom Filter which keeps track of where the message has been sent. This Bloom Filter is cleared when a query jumps to a node $d + 1$ hops away. If any node received a query message and its ID is in the Bloom Filter, then there must have been a false positive and therefore the query should fail.

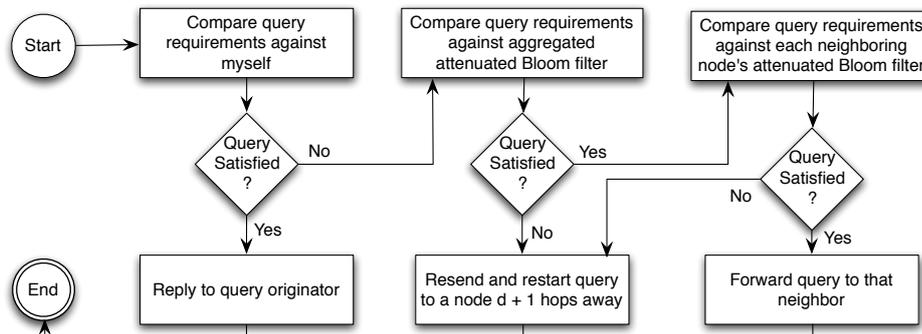


Fig. 2. Flowchart of resource, service, and application discovery from Section 3

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

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

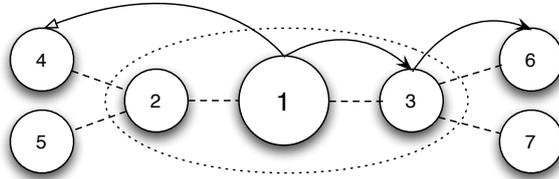


Fig. 3. Example showing how resource queries are forwarded with an Attenuated Bloom Filter of $d = 1$. When a neighbor has information about the desired resource, such as Node 3, then query is forwarded to that peer, who in turn forwards the query to Node 6 which contains the resource. In another case, when there is no information about the desired resource in Node 1's area (consisting of Nodes 1, 2 and 3), then the query is forwarded to an Outer Limit Node 4, where the search is then restarted.

instead. The difference now is when a dynamic resource changes its value and passes a certain threshold, the direct neighbors of that node are notified. Thus, the information closest to the node with the resource is kept up to date. The updating of nodes further away occurs at a later stage, when there are enough resource value alterations that can be sent in a batch, in order to save messages.

Outer Limit Peer Discovery: Using an Attenuated Bloom Filter of a certain depth d limits the amount of information a node has about its surrounding neighbors. If a query is received and cannot be satisfied using the information the node knows about its peers in the same area, then it forwards the query to another node that is $d + 1$ hops away (which, conceptually, is part of another area).

To find those Outer Limit Peers, the system uses a simple Random Walk strategy to forward a discovery message until it reaches a node $d + 1$ hops away. Once that node is found, it replies to the message originator. The Peer Discovery protocol has two parameters which can be fine tuned, such as: width (w) and length (l). Width represents the number of nodes the discovery query should be sent to in parallel, and the length is the number of hops that the outer limit node should have. On the off chance that a node does not know about any outer limit peers, either due to particular topology configurations or node failures, the system just forwards the query to a random neighbor.

3.1 Resource Representation

Information about resources, applications, and services that each node offers are represented inside a Bloom Filter. But, because a Bloom Filter is only capable of performing membership tests given a key (in this case a string), we need to add information about the actual resource (like type, value, etc.) to that key on insertion for it to be useful in discovering resources. Therefore, keys use namespaces to differentiate between resources and their values, which also helps with performing membership tests for resources. The naming convention uses a 3-level namespace, each separated using the colon (":") as a delimiter, and follows the following rules:

- *Level 1:* Name of the Resource, Service, or Application (e.g. CPU or ffmpeg)
- *Level 2:* Type of the Resource, Service, or Application (e.g. MHz or version)
- *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 reads a configuration file that specifies the name of the resource among other things. This configuration file needs to be the same for all nodes in the network.

Insertion: 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]. Or, 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”; and so on.

Querying: 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 searching for a resource with a key one *quantum* value higher than required will ensure query satisfaction.

4 Implementation Details

This work is implemented using the PeerSim [31] simulator with its Event Driven capabilities, approximating the simulation more to real-life as opposed to a Cycle Driven simulation. Because PeerSim is implemented in Java, the SERD discovery mechanism is also implemented in Java, which also allowed us to use an open source Bloom Filter implementation from the well known Hadoop project, providing us a certain amount of confidence w.r.t. its quality.

In order to be able to evaluate this work, we had to build an infra-structure around PeerSim to allow things such as topology creation, resource distribution, and node activity specification. The **topology** of a network can either be loaded using a file that describes the connections between nodes, or can be generated randomly using parameters that ensure minimum and maximum number of neighbors. **Resource distribution** among nodes can be performed in a static way using a simple file that specifies which node should have what resource; or, it can be specified in a more random fashion by specifying criteria to select a certain number of nodes. Distribution criteria can be the number of hops between nodes, the density/frequency of nodes that have the resource, or even the homogeneity of resource distribution. **Node activity specification** also uses a text file where nodes can be selected using various types of specifiers (e.g. randomly, exact match, nodes with a certain resource, etc.) along with the actions that they should perform (e.g. search for some resource) and when that action should be executed (in terms of simulation cycles or periodicity).

Another implementation issue we had was the initial construction of the Attenuated Bloom Filters. PeerSim starts with an already defined topology, so we simulated a joining phase on top of PeerSim for nodes to exchange resource information when a new peer enters the network.

5 Evaluation

PeerSim was used to evaluate SERD in a virtual network environment with six different test scenarios. These include varying the number of nodes that have the desired resource, which ranges from *very abundant* (50% of the nodes have the resource), *abundant* (25% have the resource), and *scarce* (only 5% have the resource). For each of those cases, we also vary the values of the resources, separating them into two groups: *uniform*, which is common with an application like GCC, either a computer has it or it does not; and *non-uniform* where values vary quite a bit, similar to a Hard Drive where the range we used was 0GB to 1000GB.

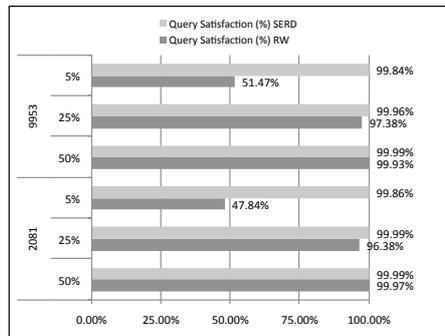


Fig. 4. Query Satisfaction for GCC (*uniform*)

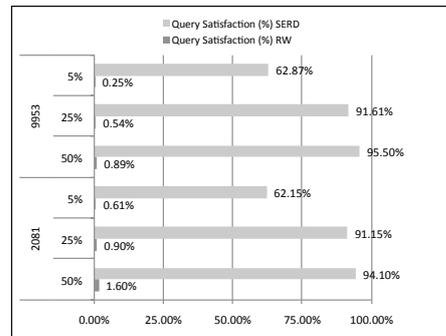


Fig. 5. Query Satisfaction for Hard Drive (*non-uniform*)

The SERD protocol was compared against the Random Walk protocol, which is used as a touchstone as it is easy to implement and functions as a baseline for performance (no protocol should perform worse). The RW implementation uses exact-match searches and just forwards queries to random neighbors.

Each scenario was tested with both RW and SERD protocols using two different network sizes: one with 2081 nodes and another with 9953 nodes, repre-

senting small and large networks respectively. Neighbors in this topology were randomly assigned, with the maximum number of neighbors being three. 10% of the nodes were randomly chosen to periodically send a query, in parallel, for a certain resource based on the scenario. Query messages were sent with a $TTL = 2 * \log_2(NETWORK_SIZE)$ to make sure resource queries eventually fail. As SERD uses an Attenuated Bloom Filter, the chosen depth for the test was $d = 3$.

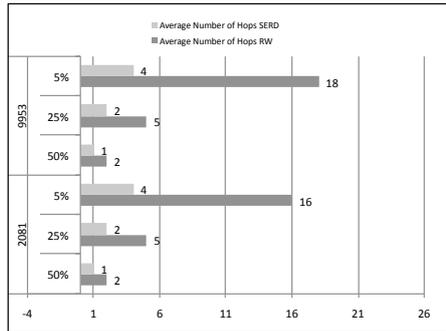


Fig. 6. Average Number of Hops for GCC (*uniform*)

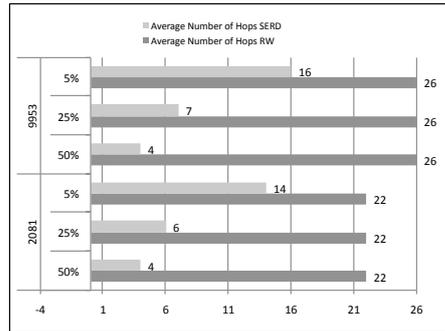


Fig. 7. Average Number of Hops for Hard Drive (*non-uniform*)

Figures 4 and 5 show the percentage of resource queries that were satisfied (out of 7072 and 33830 sent queries for network sizes of 2081 and 9953, respectively). SERD proved to be able to find the requested resources with a percentage of satisfaction consistently superior than 90%, with the exception of the *scarce* scenarios with *non-uniform* values. Still, more than half of the resource queries were satisfied even though the resources were distributed to only 5% of the nodes, further complicating the search. RW did well in the *uniform* scenario, but struggled in the *scarce* one. RW performed terribly with *non-uniform* values, barely being able to satisfy any queries. This happened because the protocol used exact-match and just randomly picked a neighbor to forward a message to, which led to dead ends.

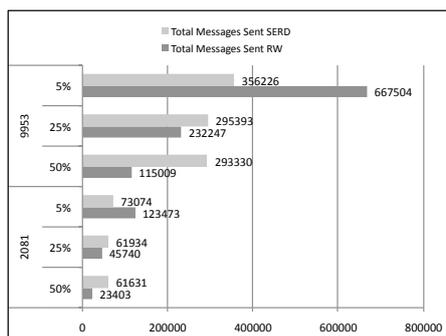


Fig. 8. Total Messages Sent for GCC (*uniform*)

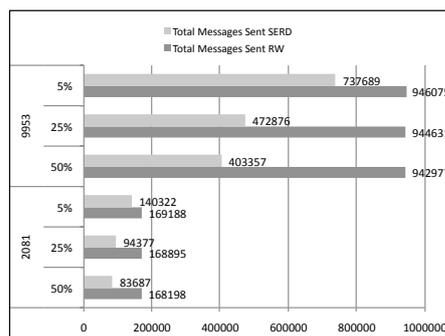


Fig. 9. Total Messages Sent for Hard Drive (*non-uniform*)

With regards to the average number of hops the resource queries traveled, seen in Figures 6 and 7, the scenarios that proved tougher had messages travel

a lot more. When searching for GCC, both RW and SERD protocols had a low number of hops, except for the *scarce* scenario where RW increased quite a bit. While looking for a Hard Drive, the average number of hops for RW were close or equal to the TTL (22 and 26 for network sizes of 2081 and 9953, resp.), which is obvious seeing as almost all queries failed.

In Figures 8 and 9 we can see the total number of sent message. With the exception of the RW protocol's search for Hard Drives, the SERD protocol uses more messages than the RW protocol. This is to be expected because SERD needs to exchange neighbor resource information (the Attenuated Bloom Filters) and look for Outer Limit Peers, unlike RW. As this is a work in progress, this is one area that we will try to optimize in order to reduce the number of messages.

Finally, the average message size and average routing information storage size occupied by each node can be seen in Figure 10. SERD messages are almost double the size of RW messages, which is expected because SERD messages include a Bloom Filter to keep track of nodes the message has passed through. With regards to the information stored at each node, SERD uses much more space than RW because RW nodes only keep information about their own resources, whereas SERD nodes store the Attenuated Bloom Filters of its neighbors and needs space for its own Aggregated Attenuated Bloom Filter.

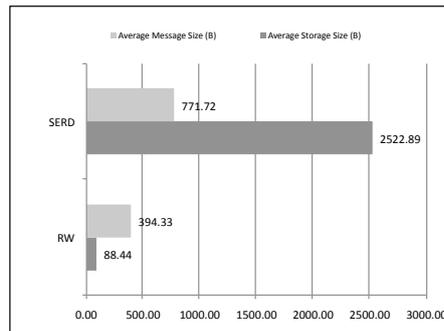


Fig. 10. Average Message Size and Average Storage Size at each Node

6 Conclusion

GiGi [10] allows home users to take advantage of Grid computing which was previously only available to scientific and corporate communities. Tasks that would usually take a 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, to distribute the tasks GiGi needs to locate the resources that satisfy task prerequisites. This is precisely what the architecture described in this paper does: discovering physical resources, services, and applications of computers connected to the same P2P Grid. The main objectives are to create a decentralized discovery mechanism that is efficient and scalable for the GiGi project. Even though this work is for the GiGi project, it is completely independent and can be used in other types of networks, such as cycle-sharing networks.

The current implementation focuses mainly on static resources (work in progress) and was evaluated alongside another, albeit simpler, discovery mechanism called Random Walk (RW). Results show that SERD proved to be better than RW,

with higher query success rates using less hops at the expense of increased message size and storage space. There is still work to be done to increase the efficiency and scalability of the system.

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. 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.
5. 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.
6. S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, Jan 2004.
7. 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.
8. D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7:96–96, 2003.
9. A. Iamnitchi and D. Talia. P2p computing and interaction with grids. *Future Generation Computer Systems*, 21(3):331–332, 2005.
10. 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.
11. M Nelson. Lzw data compression. *Dr. Dobb's Journal*, Jan 1989.
12. D Huffman. A method for the construction of minimum-redundancy codes. *Resonance*, Jan 2006.
13. A. Tridgell. Efficient algorithms for sorting and synchronization. *Doktorarbeit, Australian National University*, 1999.
14. 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.
15. 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.
16. 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.
17. Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
18. PS Almeida, C Baquero, N Pregoça, and D Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.
19. Sean C Rhea and John Kubiatowicz. Probabilistic location and routing. 2002.
20. 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.
21. 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.
22. P Maymounkov and D Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Proceedings of IPTPS02*, Jan 2002.
23. 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.
24. D Thain, T Tannenbaum, and M Livny. Condor and the grid. *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335, Jan 2003.
25. S Chapin, D Katramatos, and J Karpovich. Resource management in legion. *Future Generation Computer Systems*, Jan 1999.
26. 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.
27. E Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, Jan 1999.
28. J Waldo. The jini architecture for network-centric computing. *Communications of the ACM*, Jan 1999.
29. P Goering and G Heijenck. Service discovery using bloom filters. *Proc. Twelfth Annual Conference of the Advanced School for Computing and Imaging, Belgium*, Jan 2006.
30. Qingcong Lv and Qiyang 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.
31. PeerSim. Last checked: 2009-12-27. <http://peersim.sourceforge.net/>.