An Efficient Mechanism for Adaptive Resource Discovery in Grids

James Salter and Nick Antonopoulos

Department of Computing, University of Surrey, Guildford, United Kingdom
e-mail: j.salter@surrey.ac.uk, n.antonopoulos@surrey.ac.uk

Abstract

Computational Grids are designed to bring together collections of resources distributed among diverse physical locations, allowing an individual to exploit a huge amount of computing power, specialist instruments and vast databases. It is essential that an effective method of resource discovery is available for users and software agents to find the resources they require. We present an initial model for resource discovery in Grid environments, designed to remove the need for broadcast of updates and queries across the network. We compare our system with several others in terms of the number of messages needed to query for resources and the ability to guarantee to find matching resources if they exist anywhere in the network.

Keywords

Computational Grids, resource discovery, software agents

1. Introduction

Grid-based Computing has recently been touted in the press as the next Internet, with several major organisations announcing next-generation products and services designed to maximise the potential of Computational Grids. Efficient discovery of resources is an ongoing problem. Solutions must be scalable, fault-tolerant and able to deliver high levels of performance.

In this paper, we present an outline of an initial model for resource discovery in which we aim to remove the need for message flooding (broadcasting) whilst providing a scalable environment without a centralised structure that could lead to a single point of failure. By utilising a simple learning mechanism on top of a distributed inverted index structure we are able to provide guarantees that matching resources will be found wherever they exist in the network.

The rest of this paper is structured as follows: In Section 2 we review some resource discovery systems related to our proposed solution, with our motivations outlined in Section 3. Section 4 describes the main architecture of our system, with updating and searching algorithms discussed in two sub-sections. We present a comparison of our system against Gnutella, Freenet, Routing Indices and Chord in Sections 5 and 6. Section 7 gives an indication of our future work, before we present conclusions in Section 8.

2. Related Work

Resource discovery in Grid environments shares many elements in common with resource discovery in Peer-to-Peer (P2P) Networks. Although there are some differences between the

two approaches, several researchers believe there will be an eventual convergence (Foster and Iamnitchi, 2003), so it is valid to compare our work with other models from both approaches. We envision Grids will eventually become specialised forms of P2P Networks.

Centralised resource discovery schemes such as Napster (Saroiu, 2002) and Globus' original MDS implementation (Fitzgerald et al, 1997) are based around a cluster of central servers hosting a directory of resources. These introduce issues such as scalability, since the entire index must be stored on a single cluster. Additionally, there must be an organisation willing to donate computing resources for hosting the index. Centralised schemes can be vulnerable to attack, such as Denial-of-Service attacks, because there is a single point of failure.

Distributed solutions such as Gnutella (Ripeanu, 2001) discover resources by broadcasting queries to all connected nodes. When a node receives a query, its list of local resources is checked for matches, and any results are back propagated to the requester. Regardless of whether a match has been found a Time-to-Live (TTL) counter is decremented and, if greater than zero, the query is forwarded to neighbouring nodes.

Freenet (Clarke et al, 2000) is a distributed information storage system designed to ensure anonymity and make it infeasible to determine the origin or final destination of a resource passing through the network by utilising hash functions and public key encryption. Queries are forwarded across the network in a serial fashion, based on similarity of the query to keys stored in routing tables. Successful matches cause the resource to be cached in the local file store of each node along the path between the original requestor and the node answering the query. Routing tables are also updated to point to the node that provided the resource. The primary motivation behind Freenet is anonymity across a decentralised data store, rather than performance or long-term storage capability.

Routing Indices (Crespo and Garcia-Molina, 2002) provides a framework for nodes to forward queries to neighbouring nodes that are most likely to provide answers or be able to forward the query to a node with matching resources. Each node maintains a routing index pointing to each neighbouring node, together with a count of the total number of resources available and the number of resources available on each topic (group of keywords) by following the path from each neighbour.

Chord (Stoica et al, 2001) implements a Distributed Hash Table. Nodes are organised in a ring, with each node having a unique identifier. Queries are hashed to create a key that is mapped to the node with the matching identifier. Each node maintains a small finger table that is used to forward queries around the ring until the correct node is located, which can host a list of addresses of machines holding matching resources.

3. Motivations

Our approach to resource discovery describes a distributed architecture that eliminates the need for broadcasting (or selective broadcasting) of both queries and update messages across the network. The bandwidth consumed during broadcasting can be better utilised for the transfer of useful data between users or agents.

Several resource discovery mechanisms, such as Gnutella, which use broadcasting techniques for querying employ Time-to-Live (TTL) counters to limit the number of messages generated.

TTL counters are decremented each time a query is sent to a new node. When the TTL counter reaches zero, the query expires and is not forwarded any further through the network. Since queries in such systems do not reach every node, resource discovery cannot be guaranteed. If a resource exists but is outside the query "horizon", it will not be discovered. One of the criteria of our system was that it should guarantee to provide an answer to a query if one exists, without the need for a query to be sent to every node in the network.

4. System Model

Centralised architectures arguably provide the best performance in terms of number of messages required to give an answer to a query, but suffer weaknesses as described above. The benefits of guaranteed answers to queries and elimination of query broadcasting provided by the inverted index nature of centralised architectures are utilised by our system. We gain improvements over this by distributing the inverted index over many network nodes, replicating information and utilising preference lists (see below).

Our resource discovery mechanism comprises a series of nodes connected together in an arbitrary manner, each running a software agent. Each node represents a group of machines (a LAN, for example) connected to the Grid that may have resources, such as software, databases or CPU cycles available for use. These resources are registered against a set of keywords describing them in a Local Resource Table on their local node.

A node may become a supernode responsible for one or more keywords. A supernode maintains a Node Keyword Table listing each node providing resources matching the keyword(s) for which it is responsible.

Every node hosts a Local Node Lookup Table that contains a list of which supernode is responsible for which keyword. If each node had to store information for each supernode and keyword, a message broadcast would be needed every time a new supernode or keyword was added. Instead, we introduce a pointer in each supernode pointing to the newest supernode created. Only the Local Node Lookup Table of the newest supernode must be updated when new keywords are added. Keyword information not available in a node's Local Node Lookup Table can be found by contacting a supernode and following its pointer to the newest supernode, which will hold the latest list of keywords.

We designate certain supernodes along this "timeline" as checkpoints (Figure 1). This prevents the need to update the pointer to the newest supernode on every supernode in the Grid whenever a new supernode is formed. Instead, the update only needs to propagate backward to the latest checkpoint. If a supernode further back needs to find the newest supernode it will follow its pointer, which will point to the next checkpoint. By following the sequence of checkpoint pointers, the newest supernode will be found within a few hops.

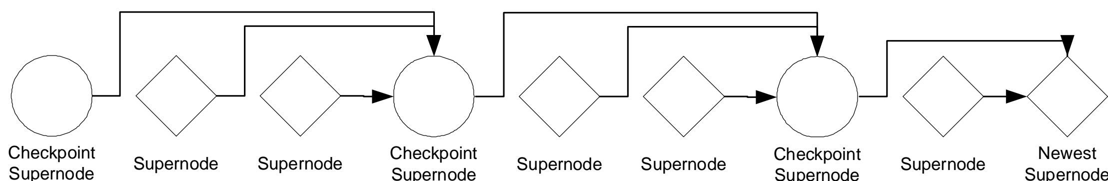

Figure 1. Timeline of supernodes connected in order of joining

4.1 Searching For Resources

When a query is made using our resource discovery mechanism, firstly the local node's software agent checks the Local Resource Table for matching resources. This allows for matches to be provided within the node's local environment, without the need to send any messages across the Grid.

If no matching resources are found, or those that match are unavailable to the user, the search will proceed to find resources on remote nodes. The local node's Local Node Lookup Table (Figure 2) is searched to see if keywords in the query are listed. If not, any supernode listed in the table can be chosen arbitrarily and contacted. If searching this supernode's Local Node Lookup Table still does not find the required supernode, the software agent running the query will traverse the timeline to contact the newest supernode and use its Local Node Lookup Table to find the keyword and a pointer to the supernode responsible for it. If the keyword is not listed there, we can be certain the keyword does not exist in the system at all and return a null result.

keyword	supernode	preference list
temperature	45	3, 7, 45, 863, 23
rs232	757	757
turing	235	235, 343, 1, 55, 34, 76
...	...	...

Figure 2: Sample Local Node Lookup Table

When the address of the supernode responsible for the keyword is found, the supernode is contacted. The supernode's Node Keyword Table gives a list of all nodes providing resources matching the keyword. The list of nodes is attached to the query, which is then forwarded to each node's agent in turn until an available matching resource is found through searching of each node's Local Resource Table. In addition to the address of the machine with the matching resource, the list of nodes from the supernode is sent back to the original querying node. The node can then use this list to build its preference list.

Preference lists are maintained in Local Node Lookup Tables at each node, independently of supernode/keyword lists stored at other nodes. The lists are sorted in order of the node most likely to be able to match the query, based on successful matches in the past. They act as a learning mechanism to enhance the performance of future similar queries. Using preference lists in addition to indexes stored on supernodes helps eliminate the need to broadcast queries across the Grid to find relevant resources.

When a preference list for a keyword exists on a node's Local Lookup Table, it can be used to begin the search directly without the need to contact the keyword's supernode, which helps to spread the query load across multiple servers for popular queries:

We attach the preference list to the query, so if the first node to be sent the query cannot provide matching resources the query will be forwarded directly to the next node in the list. Using this scheme, the node most likely to answer the query is always contacted first, then other nodes are contacted in serial until the query is answered.

If the supernode is not listed in the preference list, it is attached to the end of the preference list prior to sending it out from the local node. When the query reaches the supernode for the

keyword, any other nodes listed on the supernode but not currently in the preference list can be added to the end of the list. This allows for a complete search, without the need to broadcast the query to nodes that do not provide any matching resources.

Preference lists are important as they form the first stage in amalgamating simple keyword matching with more complex matching of users to resources in the context of availability and access policies (Antonopoulos and Shafarenko, 2001). For example, the supernode for a keyword may offer resources matching a user's query, but its security policy may prevent that user from exploiting the resources because of the relationship with the owner of the node on which they are connected. Therefore, it would be more beneficial for a user to directly contact a different node with matching resources and a compatible security policy, rather than first contacting the supernode.

By introducing the concept of preference lists into our system we are able to minimise the number of messages required to provide an answer to a query, whilst still providing guarantees that if a matching resource exists we will be able to find it. Searching for a keyword's supernode as described above is the worst-case scenario in terms of number of messages required to process a query. This activity, which uses only a small number of additional messages compared to a standard query, is a one-time occurrence. The keyword's supernode will be listed in the node's Local Node Lookup Table for subsequent queries and preference lists will be deployed to further minimise the message count.

4.2 Adding New Nodes and Resources

When a new node wishes to join the Grid, it contacts the software agent running on an existing node in order to inherit the node's Local Node Lookup Table. A blank Local Resource Table is also created on the node to list any local resources that will be introduced. If the node does not offer any resources, this is all it needs to do to begin participation within the Grid.

If a resource is to be introduced into the system, it is first listed against each keyword for which it is associated in the Local Resource Table on its local node. If keywords not previously associated with the local node are being introduced, the keyword information must also be published on the relevant supernodes to allow the new resource to be found by any node on the Grid.

The address of a supernode is chosen from the local node's Local Node Lookup Table, which will in turn provide a pointer to the newest supernode (possibly via hops across checkpoints along the timeline).

The newest supernode's lookup table is consulted to see whether keywords associated with the new resource already exist. If they do, the address of the new node can simply be registered with the relevant supernodes responsible for each keyword as providing resources matching that keyword. A broadcast message is not necessary, as information about the new resources can be found by contacting the relevant supernode.

If one or more keywords associated with the resource did not previously exist within the network, the node hosting the resources will be designated as the supernode for those keywords. The Local Node Lookup Table for this new supernode will be inherited from the

previous newest supernode and then updated with entries showing the new supernode as the supernode for the new keywords. Pointers in other supernodes that are newer than the current checkpoint will be updated to reflect the presence of the new supernode. Again, a broadcast of information relating to the new node is unnecessary.

5. Comparison with Chord

Chord provides a mechanism for resource discovery with O(log N) messages required for query processing and O($\log^2$ N) messages for updating routing information following a node joining the network. The number of messages to process a query in our system is dependant on the number of checkpoints in the timeline, so to show improvement in the worst-case over Chord there should be less than $\log_2(N)$ checkpoints. Similarly, the number of update messages required each time a supernode is added to the timeline is dependant on the number of supernodes between checkpoints (since updates must propagate back to the most recent checkpoint). To show improvement over Chord, the number of supernodes between checkpoints should be less than the messages taken by an update in Chord.

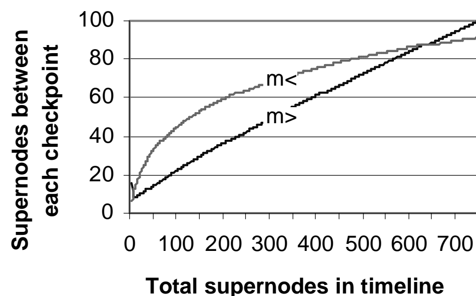

Figure 3: Maximum and Minimum Supernodes between Checkpoints

Figure 3 plots the maximum and minimum values for m, where m is the number of supernodes between checkpoints. It shows that by controlling the number of supernodes between checkpoints, improvement in both query processing and update messages is possible in worst-case scenarios for timelines with 8-627 supernodes. We have assumed updates and queries occur with similar frequency, although in real systems the update:query ratio is likely to be much smaller, meaning the timeline can scale to much larger numbers of nodes and still show improvement. For example, if the ratio is 1:10, we can find improvement with up to 27998 supernodes. Hopping across each checkpoint in the timeline to process a query only occurs for nodes that do not submit frequent queries, as nodes learn the address of the newest supernode after submitting a query, minimising timeline traversals. In combination with preference lists, this feature can be used to further reduce query messages and thus scalability.

6. Comparison with Other Systems

By describing the main architecture and algorithms of our system (see section 4) we have shown that we have eliminated message broadcasting whilst still providing guarantees that an answer to a query will be found if one exists in the network, without the need for a centralised architecture.

The original Gnutella employed both message broadcasting and TTL counters, meaning it was not able to provide guarantees that resources requested could be found in the network, even though it used many more messages than our system. To search every node in the system

would require approximately one message to be generated for every node, resulting in scalability concerns in large networks (Ritter, 2001). Since Gnutella provides no facility for improving query performance based on past experience, every time a popular query is submitted a similar number of messages will be generated.

Freenet's architecture employs a steepest-ascent hill-climbing search with backtracking, meaning queries are forwarded to nodes in a serial fashion, eliminating message broadcast. However, unlike our system, it cannot guarantee to find matching resources as it utilises TTL counters (called hops-to-live limits). Query performance improves over time, as successful queries cause matching resources to be cached and routing tables updated. However, Freenet consumes a lot of bandwidth during this process as resources are copied between locations.

Routing Indices (RI) presents an improvement over Freenet by always returning a result if one exists in the network. Rather than using a TTL counter, RI uses a Stop Condition to specify the number of matching resources that must be found before a query will be terminated. Summarisation techniques are employed to provide keyword groupings, which may mean that resources related to rare keywords may not be found because RI uses a frequency threshold that discards topics with very few documents. Indices are not modified depending on the success or failure of a query, so there is no scope for improvement in query performance through learning. Routing Indices relies on a network architecture where every node can be reached (possibly via a path consisting of several intermediate nodes) by every other node. If this level of connectivity is not maintained, resources may not be found because they are unreachable from a point in the network. There is also the potential for a high number of update messages to be generated, because when resources are introduced or removed, routing indices on every node in a path pointing to the node hosting the resource must be updated.

In all the above systems, the number of messages required to discover a resource is dependant on the number of nodes in the network. While Routing Indices and Chord do not require a query to visit every node in order to guarantee that all matching resources are found, the number of messages required will still increase as the network grows.

In our system, the number of messages required is dependant on the number of keywords related to resources in the network (and therefore potentially the number of hops across the supernode timeline required to find the supernode for the keyword required). Therefore, in specialist networks where the number of keywords will be bounded (such as a specialist scientific Grid), our system will provide guaranteed answers to queries in very few messages, even as the number of network nodes grows.

7. Future Work

This paper has presented an initial model for resource discovery, but it is still in its infancy. Dynamically changing properties of resources, such as CPU load, have not been discussed here but is a major issue the system will eventually need to address.

We will shortly begin investigating the environmental conditions that should lead to lookup tables on supernodes being replicated, merged, split or migrated to other nodes. For example, migrating responsibility for a keyword to a supernode responsible for a second keyword often used in combination with the first may be desirable. A scheme for "unlearning" information from certain tables will be devised to ensure a manageable amount of data is retained on each

node. Failure of checkpoint supernodes could currently break the timeline, so extra routing information must be added, with checkpoints potentially pointing to more than one newer checkpoint, giving the side-benefit of traversal across the timeline in fewer messages.

We are discussing the possibility of a contention period before new supernodes are finalised, in which other nodes could be short listed as contenders. This could allow us to optimise supernode use, by selecting the most powerful node to provide the extra supernode services.

8. Conclusions

In our work so far we have already demonstrated a method of providing a scalable resource discovery mechanism without the need for either a single centralised index or the costly method of message broadcasting associated with fully distributed resource discovery.

Keeping indexes of individual resources on the nodes hosting the resources provides a more fault tolerant and scalable solution than centralised approaches such as Napster. By imposing a certain amount of structure, in the form of supernodes connected together on a timeline-like arrangement, we are able to remove the need for flooding the network with queries, a key requirement in systems such as Gnutella. We can also guarantee to find all available resources (if necessary) as nodes always point *towards* the answer. We have presented some initial ideas addressing the concept of answering queries with resources most likely to be accessible (in terms of access and security policies) by the users requiring them.

9. References

Antonopoulos, N. and Shafarenko, A., 2001. An Active Organisation System for Customised, Secure Agent Discovery. *In The Journal of Supercomputing,* Vol. 20, No. 1, pp. 5-35.

Clarke, I., Sandberg, O., Wiley, B. and Hong, T. W., 2000. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Lecture Notes in Computer Science,* Vol. 2009, pp. 46-66.

Crespo, A. and Garcia-Molina, H., 2002. Routing Indices for Peer-to-Peer Systems. *Proceedings of the International Conference on Distributed Computing Solutions (ICDCS'02).*

Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W. and Tuecke, S., 1997. A Directory Service for Configuring High-Performance Distributed Computations. *Proceedings of the 6th IEEE Symposium on High Performance Distributed Computing.* Portland, OR, USA, pp. 365-376.

Foster, I. and Iamnitchi, A., 2003. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTP3'03).* Berkeley, CA, USA.

Ripeanu, M., 2001. Peer-to-Peer Architecture Case Study: Gnutella Network. *University of Chicago Technical Report TR-2001-26.*

Ritter, J., 2001. Why Gnutella Can't Scale. http://www.darkridge.com/~jpr5/doc/gnutella.html

Saroiu, S., Gummadi, P. K. and Gribble, S. D., 2002. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02).*

Stoica, I., Morris, R., Karger, D., Kaashoek, M., F. and Balakrishnan, H., 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the ACM SIGCOMM 2001.* San Diego, CA, USA, pp. 149-160.