TOWARDS AN INTELLIGENT AGENT MODEL FOR EFFICIENT RESOURCE DISCOVERY IN GRID ENVIRONMENTS

Nick Antonopoulos
Department of Computing, University of Surrey,
Guildford, Surrey. GU2 7XH
n.antonopoulos@surrey.ac.uk

James Salter
Department of Computing, University of Surrey,
Guildford, Surrey. GU2 7XH
j.salter@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 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 whilst providing a fault-tolerant environment minimising the impact failure of a centralised index node would have on the ability to discover surviving resources.

KEYWORDS

Computational GRIDs, resource discovery, Intelligent 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. However, there is still a large amount of work to be completed before serious GRID implementations will be usable outside the scientific or large-scale business communities.

Efficient discovery of resources is an ongoing problem. Solutions must be scalable, fault-tolerant, capable of adapting in the quickly changing GRID environment and able to deliver high levels of performance.

In this short 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.

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.

Distributed solutions such as Gnutella (Ripeanu, 2001) discover resources by flooding the network and forwarding queries to neighbouring peers. Significant overheads are introduced as the network grows

because of the flooding techniques employed. Further, it is not guaranteed that all relevant query matches will be returned, because queries have limited Time To Live as they are forwarded through the network.

At the other extreme are centralised schemes such as Napster and Globus' original MDS implementation (Fitzgerald et al, 1997). Such schemes are based around a cluster of central servers hosting a directory of resources, which introduces problems to be addressed such as speed of updates and fault tolerance.

Other approaches such as CAN (Ratnasamy et al, 2001) and Chord (Stoica et al, 2001) implement distributed hash tables. Tapestry (Zhao et al, 2001) provides location-independent routing to the closest copy of a resource using point-to-point links, without any form of central control. All three create a form of logical mapping on top of the underlying physical structure of the network.

3. SYSTEM MODEL

Our resource discovery mechanism comprises a series of nodes and supernodes. Each node represents a machine or cluster of machines connected to the GRID and 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 index table on the node on which they are hosted. It is expected that each resource will be associated with a small set of keywords and the system has not been conceived to provide full-text searching capabilities.

Nodes may be designated as supernodes for one or more keywords related to the resources they hold. A supernode maintains a list of each node providing resources matching the keyword(s) for which it is responsible.

The address of each supernode and the keywords it is responsible for are listed in tables maintained at each supernode. However, if each supernode 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. Then, only the newest supernode must be updated when new keywords are added, as the pointers in the other supernodes can be followed to find the latest list of keywords if necessary.

We designate certain supernodes along this "timeline" as checkpoints (Figure 1). This prevents the need to update the pointer to the newest supernode in 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, providing the checkpoints are located at sensible distances apart. For example, if we assume that there are n checkpoint supernodes in the timeline, the worst-case number of hops will be $n+1$, the best-case being 1 hop and the average being $(n/2)+1$.

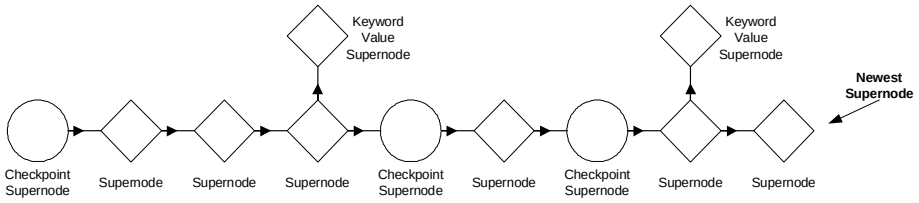

Figure 1. Supernodes connected in order of joining, with keyword value supernodes linked vertically above

Unlike in P2P file sharing systems where queries are always constructed of a list of one or more keywords, queries within GRID environments may contain a combination of keywords and name-value pairs ("CPUSpeed=1.8Ghz, RAM=512Mb" for example). We treat the name part of these pairs as the keyword, and introduce the concept of keyword value supernodes to distribute the supernode load for different values over multiple nodes.

When a new node with new resources wishes to join the GRID, it first contacts an existing node, which will point to a supernode that will in turn point to the newest supernode. The newest supernode's lookup table is consulted to see whether keywords registered against each resource in the new node's local index table 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(s).

If one or more new keywords are to be introduced, the new node will be designated as the supernode for those keywords. The 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 will be updated to reflect the presence of the new supernode, as described above. Again, a broadcast of information relating to the new node is unnecessary.

Our system uses two layers of processing running in parallel. Intelligent agents are used to manage the background processes associated with system management, such as adding nodes and designating supernodes. Concurrently, other agents actively seek to answer queries using the methods discussed in section 3.1 below.

Using agents to represent queries allows for a reduction in messages compared to a system where searches are controlled from the originating node, because null results do not need to be sent back to the node to decide what action must be taken next (which node should be searched next for example). Agents will also allow queries and searching strategies to be modified in response to intermediate results during query execution.

3.1 Searching for Resources

In nearly all P2P systems, a query is assumed to be searching for remote resources. For example, typically users are searching for music, videos, documents and the like, so they would know if their local machine already held that file and would not use the P2P system to search for it. The same cannot be said in GRID environments, where queries are often for resources such as CPU cycles or storage space, which may be available within the local environment. Additionally, queries may be made by agents as well as users. Agents cannot be expected to know whether a resource (a specialist scientific instrument for example) is available locally, in the same way that a human who had physically installed the equipment or software would.

When a query is made using our resource discovery mechanism, firstly the software agent running the query checks the local node's local index table for matching resources. If no matching resources are found, or those that match are unavailable to the user, the agent will proceed to find resources on remote nodes:

The local node's lookup table showing which supernodes handle which keywords (Figure 2) is consulted to see whether the required keyword is listed. If not, any supernode can be contacted and its lookup table searched for the keyword. If still no matches are found, the software agent will traverse the timeline to contact the newest supernode and use its lookup table to find the keyword and a pointer to the supernode responsible for it.

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 keyword lookup table

On the other hand, if the keyword is listed in the local lookup table we can utilise the table's preference list. Preference lists are constructed 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.

Using preference lists in addition to indexes stored on supernodes eliminates the need to broadcast queries across the GRID to find relevant resources. The software agent carries the preference list in addition to the actual query, so that if the first node contacted cannot provide matching resources the agent can 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. The number of messages required is kept to a minimum.

If no preference list exists, or the list of nodes in the preference list has been exhausted, the supernode for the keyword is contacted in an attempt to find more nodes with matching resources. Again, nodes are contacted in a serial manner until a match is found.

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.

Additionally, preference lists increase the performance and scalability of our model, since a supernode will not be targeted with every query involving a keyword for which it is responsible. We therefore believe that over time we should see a load-balancing effect on queries, as supernodes are targeted less often due to increased utilisation of preference lists and their provision of direct pointers to nodes hosting resources. This will help minimise the risk of bottlenecks forming on supernodes.

Query response time is directly proportional to the number of messages required to provide an answer to the query, with less messages meaning a faster response. Preference lists reduce the need to contact supernodes, traverse the timeline and contact nodes that are unable to provide the resource to the user, meaning the cost of finding a resource will over time be reduced to a single message.

3.2 A Simple Example

In this example, we assume that a new node has just joined the system and registered the resources it holds, but has not yet made any queries.

The first query is then made, for keyword A. The local node's keyword lookup table is checked, but A is not listed. Therefore, the query's software agent is sent to an arbitrarily chosen supernode (found from the keyword lookup table). This supernode does not list the supernode responsible for keyword A, so the software agent is sent across the timeline to arrive at the newest supernode, via several checkpoint supernodes. The agent is then sent to the supernode responsible for keyword A, the address of which has been found on the newest supernode. A list of nodes providing resources matching keyword A is provided to the agent by the supernode. The supernode is queried, but cannot provide an answer to the query at the current time, so the agent is forwarded to a node picked from the list. This node may not provide an answer either, but eventually the agent is forwarded to a node willing to provide a resource matching keyword A. Information regarding this resource, together with the list of nodes from the supernode, is returned to the original node that issued the query. The supernode discovered is listed against keyword A in the node's keyword lookup table. The list of nodes provided by the supernode is stored as a preference list with the node that eventually answered the query being placed at the front of the list.

This initial query involved several steps and generated multiple messages before a matching resource was found. However, it was a one-time activity, as the node can now utilise the extra information available to it via the preference list to answer subsequent requests for keyword A.

Later, following several other queries, another query is made on the same node for keyword A. This time, the supernode for keyword A is listed in the node's keyword lookup table. There is no need to send the agent to the supernode, however, as it can instead utilise the preference list associated with keyword A. The agent takes the query ("A") and the preference list and visits the first node in the list. As the query has now been made several times, the preference list provides a reasonable guide of the likelihood a node will be able to provide an answer to the query. In this case, the first node visited is too busy to provide the resource, so the agent moves to the next node in the list. This can answer the query, so the agent returns to the originating node with the necessary information. The preference list for keyword A is also modified on the node to reflect the positive response from the second node and the negative one from the first.

This short example has shown how the use of preference lists has reduced the number of messages taken to provide an answer to a query. It can be seen how they will allow individual nodes to adapt the routing of their queries depending on positive or negative outcomes of previous attempts.

4. FUTURE WORK

This paper has presented an initial model for resource discovery. Much work is still to be done and a number of questions need to be answered before the model will be workable in a realistic environment.

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. The positive or negative aspects of these will be analysed in terms of gains in performance and increased message traffic needed to oversee the activities.

A scheme for "unlearning" information from certain tables will be devised to ensure a manageable amount of data is retained on each node. Again, the costs and performance benefits of this will be measured.

Modifying the granularity of the keywords for which each supernode is responsible may have an effect on performance. For example, rather than giving a supernode responsibility for a single keyword, it may be beneficial to combine several keywords into a single keyphrase.

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.

5. 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.

Work on our model will now progress to establish answers to the as yet unanswered questions we have outlined. Results and a more complete description of our model will be published in due course.

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.

Fitzgerald, S. et al, 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.

Ratnasamy, S. et al, 2001. A Scalable Content-Addressable Network. *Proceedings of ACM SIGCOMM 2001.* San Diego, CA, USA, pp. 161-172.

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

Stoica, I. et al, 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.

Zhao, B. et al, 2001. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. *Technical Report UCB/CSD-01-1141.* UC Berkeley, CA, USA.