Efficient resource discovery in grids and P2P networks

Nick Antonopoulos and James Salter

The authors

Nick Antonopoulos is University Lecturer and **James Salter** is a PhD Student, both in the Department of Computing, University of Surrey, Guildford, UK.

Keywords

Computer networks, Resources, Computer software

Abstract

Presents a new model for resource discovery in grids and peer-to-peer networks designed to utilise efficiently small numbers of messages for query processing and building of the network. Outlines and evaluates the model through a theoretical comparison with other resource discovery systems and a mathematical analysis of the number of messages utilised in contrast with Chord, a distributed hash table. Shows that through careful setting of parameter values the model is able to provide responses to queries and node addition in fewer messages than Chord. The model is shown to have significant benefits over other peer-to-peer networks reviewed. Uses a case study to show the applicability of the model as a methodology for building resource discovery systems in peer-to-peer networks using different underlying structures. Shows a promising new method of creating a resource discovery system by building a timeline structure on demand, which will be of interest to both researchers and system implementers in the fields of grid computing, peer-to-peer networks and distributed resource discovery in general.

Electronic access

The Emerald Research Register for this journal is available at
www.emeraldinsight.com/researchregister

The current issue and full text archive of this journal is available at
www.emeraldinsight.com/1066-2243.htm

Internet Research
Volume 14 · Number 5 · 2004 · pp. 339-346
© Emerald Group Publishing Limited · ISSN 1066-2243
DOI 10.1108/10662240410566926

Introduction

Grid computing and peer-to-peer (P2P) networks are emerging technologies enabling widespread sharing of distributed resources. However, efficient discovery of resources is an ongoing problem in both fields. Solutions must be scalable, fault-tolerant and able to deliver high levels of performance.

In this paper, we present an initial model for resource discovery in which we aim to reduce the number of messages required to resolve queries while 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, within a small number of messages.

Related work

Resource discovery in grid environments shares many elements in common with resource discovery in 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.

Centralised resource discovery schemes such as Napster (Saroiu *et al.*, 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. 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

Internet Research
Volume 14 · Number 5 · 2004 · 339-346

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 (RI) (Crespo and Garcia-Molina, 2002) provide 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.

Distributed hash tables (DHTs) (Kelaskar *et al.*, 2002) such as Chord (Stoica *et al.*, 2003) have become the dominant methodology for resource discovery in structured P2P networks, typically providing query routing in $O(\log N)$ messages and updates following a node join in $O(\log^2 N)$ messages. Each node in the network hosts part of the index, and queries are hashed to create a key that is mapped to the node with the matching identifier. In Chord, nodes are organised in a ring. Each node maintains a small finger table that is used to forward queries around the ring until the correct node is located.

Motivations

Our approach to resource discovery describes a distributed architecture that efficiently processes queries and updates across the network using a small number of messages. Several resource discovery mechanisms, such as Gnutella, which use broadcasting techniques for querying employ 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.

System model

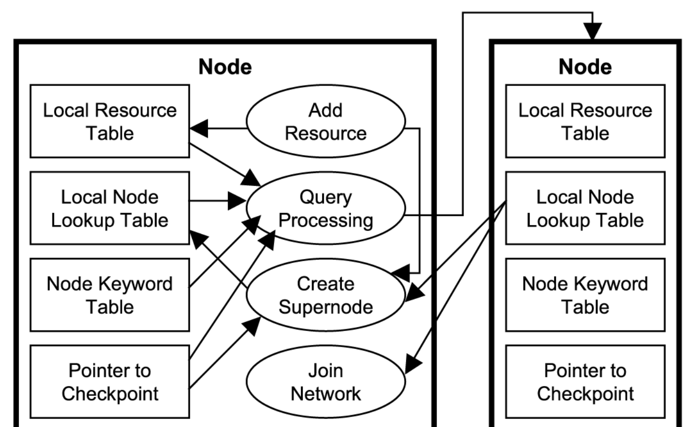
Centralised architectures arguably provide the best performance in terms of number of messages required to give an answer to a query, but suffer weaknesses such as scalability and single point-of-failure as described above. The benefits of guaranteed answers to queries and elimination of query broadcasting 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.

Our resource discovery mechanism comprises a series of nodes connected together in an arbitrary manner, each running a software agent. Figure 1 shows the model of a typical node. Each node represents a group of one or more machines connected to the network that may have resources, such as software, databases or CPU cycles available for use. These resources are registered against a set of one or more 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

Figure 1 Node model showing data structures and processes of a typical node



supernode and following its pointer to the newest supernode, which will hold the latest list of keywords.

Certain supernodes along this timeline are designated as checkpoints (Figure 2). This prevents the need to update the pointer to the newest supernode on every node in the timeline whenever a new one is formed. Instead, only the pointer on the latest checkpoint is updated. If a supernode further back needs to find the newest supernode it will contact the checkpoint preceding itself, which will in turn point to the next checkpoint in the timeline. By following the sequence of checkpoint pointers, the newest supernode will be found within a few hops.

To overcome the continuous increase in size of the local node lookup table on the newest supernode, the timeline is split into segments. When the size of the table grows over a set limit, the current segment is closed and a new segment created. The local node lookup table is cleared at the start of each segment, meaning it will only list information on supernodes within the current segment. A mesh of segments is created to enable queries to be multicast to other segments if they are not resolvable in the current segment.

The timeline can support name-value pair style keywords in addition to standard single-word keywords. These are useful for defining the semantics of a keyword. For example, a query for "cpu_speed=1,600" is split into name (cpu_speed) and value (1,600) components, giving more meaning than a single-word keyword "1,600". The name component is treated as a standard keyword. However, rather than listing all resource providers with resources matching that keyword, the supernode responsible for the keyword acts as the root of a second timeline. This value timeline is constructed of a series of keyword value supernodes, which act in a similar way to supernodes described above, but index the value component of the name-value pair for the single keyword.

Searching for resources

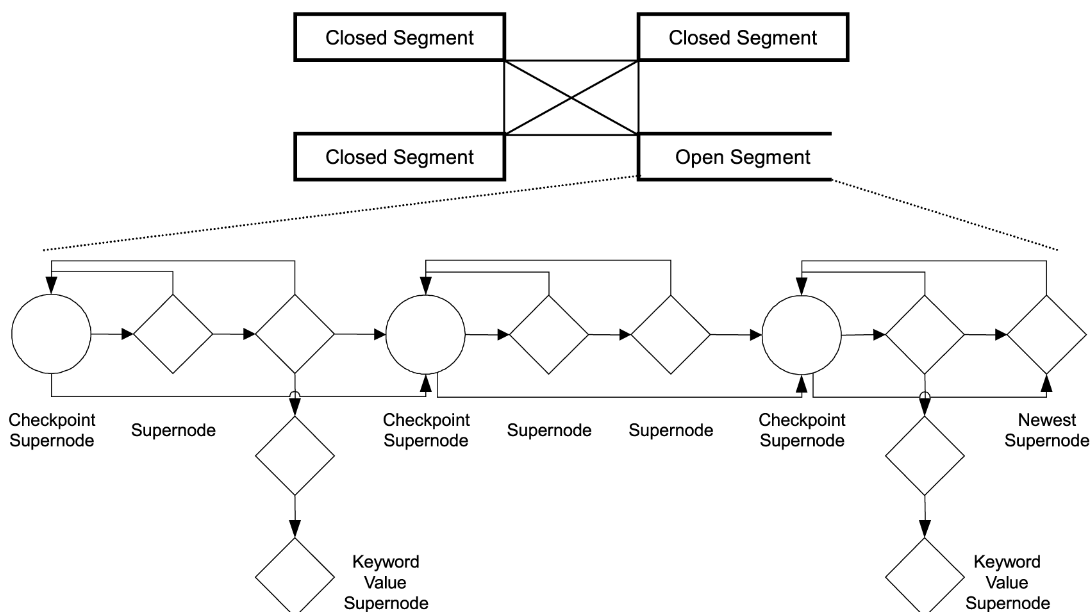
When a query is made using our resource discovery mechanism, firstly the local node's software agent checks the local resource table for matching resources. This allows matches to be provided within the node's local environment, without the need to send any messages across the network.

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 (Table I) is searched to see if keywords in the query are listed. If not, the most recent supernode listed in the table is contacted. If searching this supernode's local node lookup table does not list the required

Table I Sample local node lookup table

Keyword	Supermode	Preference list
Temperature	45	3, 7, 45, 863, 23
Rs232	757	757
Turing	235	235, 343, 1, 55, 34, 76
...	...	...

Figure 2 Timeline of supernodes connected in order of joining



Note: Value timelines can be seen extending vertically

supernode, the software agent running the query will be sent through the timeline. Each supernode has a pointer to the previous checkpoint in the timeline. In closed segments, the checkpoint hosts a pointer directly to the newest checkpoint (the closing supernode) in the segment. In the open segment, checkpoints forward queries to the next checkpoint in the timeline. The query is sent across checkpoint supernodes (or directly to the newest segment checkpoint), checking each one's local node lookup table as it goes, until either the keyword is found or the newest supernode is reached. If the keyword is not found, the query is multicast to the closing node of each other segment and their local node lookup tables searched.

When the address of the supernode responsible for the keyword is found, the supernode is contacted. If the query is for a single-word keyword, the supernode's node keyword table gives a list of all nodes providing resources matching the keyword. If the keyword is a name-value pair, the value timeline relating to the name is traversed to find the keyword value supernode responsible for the value component of the pair, which will have a node keyword table listing nodes providing matching resources.

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.

When a preference list for a keyword exists on a node's local node 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. 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. Another approach is to weight each preference list entry and randomly select a node, to enable alternative nodes to be tested periodically and avoid potentially overloading the most popular resource providers.

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. Periodically full searches are conducted for keywords which have associated preference lists, to enable the preference list to be refreshed and new resource providers discovered.

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 contact directly 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, while 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 minimise the message count further.

Adding new nodes and resources

When a new node wishes to join the network, 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 network.

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 in the network. 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 local node lookup tables of each segment are 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 on the currently open segment and then updated with entries showing the new supernode as the supernode for the new keywords. The pointer on 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.

If the size of the local node lookup table has exceeded the maximum limit (configured prior to the timeline being built), the current segment will be closed and the next supernode added will begin a new one. The supernode will not inherit the current local node lookup table, but will begin a blank one. On closure of a segment, a message is sent to each checkpoint in the segment to update their pointers to point to the final node of the segment, allowing the complete segment local node lookup table to be quickly accessed.

Multiple keyword queries

Queries may be submitted comprising multiple keywords ("distribution = binomial AND factor = 34 AND fractal" for example). To resolve these queries efficiently using a minimal number of messages, for keywords with preference list entries the preference lists can be intersected to find common resources.

Where preference lists for each pair do not exist, a query is executed for a single keyword that is selected as the one most likely to be resolved in the least messages (if one of the keywords had been previously queried for then this pair should be selected as the supernode for the keyword would already be listed in the local node lookup table). If the set of results returned is large, a query is run for another keyword and the set of results intersected. If the set of results is small, each resource provider is individually queried to check whether the remaining keywords match any of their resources.

Comparison with other systems

By describing the main architecture and algorithms of our system we have shown that we have eliminated message broadcasting while 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.

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. RI 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

Internet Research
Volume 14 · Number 5 · 2004 · 339-346

introduced or removed, routing indices on every node in a path pointing to the node hosting the resource must be updated.

Comparison with Chord

Chord and similar single-layer DHTs do not directly provide any mechanism for lookup of name-value pair keywords where the name and value are split into separate components. Therefore, to provide a fair comparison with Chord we only discuss a timeline indexing standard single-word keywords. Chord provides a mechanism for resource discovery with $\log(n)$ messages required for query processing and $\log^2(n)$ messages for updating routing information following a node joining the network in the worst-case.

The number of messages to process a query in our system is dependant on the number of checkpoints in the timeline and the number of segments. For a query originating in a closed segment, the worst-case number of messages required is $s + 2$, where s is the number of segments. The constant 2 messages are required to locate a checkpoint and then hop to the newest (closing) checkpoint in the segment. For queries originating in the open segment, the worst-case number of messages required is $c + s + 2$ where c is the number of checkpoints in each segment, since the query must be sent across each checkpoint in the open segment, then multicast to all other segments. In both cases we assume the query is not resolvable in the originating segment. To show improvement over Chord in the worst-case we must have:

$$c + s + 2 < \log_2(n) \quad (1)$$

where n is the total number of supernodes in the timeline.

We define latency as a measure of the time taken to resolve a query, which is influenced by the number of steps (hops) involved in the lookup process. Chord's routing algorithm requires routing to take place sequentially. However, in our approach since each timeline segment is connected within a mesh, a query can be multicast to each segment simultaneously. Reduced latency over Chord is possible providing the number of hops required to be completed sequentially (hopping across the timeline) is less than the total number of hops required to resolve the query in an equivalent Chord ring. This can be achieved for queries originating in closed segments in the worst case where $\log_2(n) > 3$ and $\log_2(n) > c + 3$ (by substituting 1 for s in the above formulae to measure the number of sequential steps) for queries originating in the open segment.

Each time a supernode is added, a message is sent to the most recent checkpoint to update its pointer to the newest supernode. In total mc of these will be sent in a segment, where m is the number of supernodes between segments. On creation of a checkpoint supernode, update messages are sent to the final checkpoint in each closed segment ($s - 1$ messages). Finally, when a segment is closed, each checkpoint is updated to point to the final checkpoint in the segment, using c messages. In total, the number of update messages to create a segment is $c(m + (s - 1)) + c = c(m + s)$. It follows that the average messages to create a supernode is $c(m+s)$/ nodes in segment$=c(m+s)/cm = m+s/m$. To show improvement over Chord for adding nodes in the worst case, we must have:

$$\frac{m + s}{m} < \log_2^2(n). \quad (2)$$

Utilising inequalities (1) and (2) together with (there cannot be less than one checkpoint per segment) and (the total supernodes in the network), we can derive parameter values for which the timeline can show lower message costs for queries and updates than Chord.

As an example, a timeline consisting of $n = 5{,}000$ supernodes split across $s =$ three segments with $c =$ seven checkpoints and $m = 239$ supernodes between each checkpoint would satisfy the above conditions, therefore yielding lower numbers of messages than Chord (~ 1 to insert a node and 12 worst-case query routing compared to ~ 151 worst-case for Chord to insert a node and 13 to route a query in the worst case).

Many such combinations of parameters are possible, but the expected size of the timeline (n) must be known before the network is built so they can be set correctly. In many scenarios it is impossible to know this, so the timeline must be adapted dynamically as it is built to ensure the message costs remain low. Developing a set of intelligent algorithms for determining when checkpoints should be created and the timeline segmented is a topic for future work.

The worst-case scenario of hopping across each checkpoint in a segment to process a query only occurs for nodes that believe a supernode prior to the first checkpoint in the segment is the newest supernode. When queries are sent across the timeline, the querying node learns the address of a newer supernode. Timeline traversals begin from that supernode in future queries, so nodes which submit frequent queries will experience the smallest hop counts across the timeline. Although a query from a node that has not submitted a query recently will potentially have a greater number of

Internet Research
Volume 14 · Number 5 · 2004 · 339-346

hops, by its nature the node will not frequently send out queries into the timeline and therefore not add substantial numbers of messages to the overall network traffic. Additionally, once the node has queried, it will learn the address of the newest supernode and thus its queries will be routed more quickly in future.

Case study

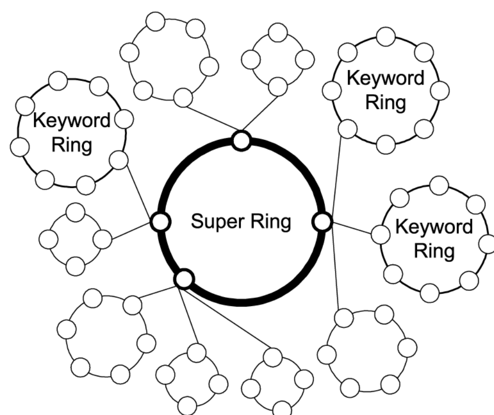
We have shown above how our model can be implemented using a timeline structure. This model can be generalised to become a methodology for building efficient resource discovery mechanisms.

The construction of our model follows a different approach from previous methods of inserting nodes into the overlay as they join the network. The topology is instead built on-demand, with nodes being added when a new keyword is registered in the system. In this case study we use a multi-tiered Chord architecture as an example of how the methodology could be deployed using other topologies, further details of which are provided separately (Salter and Antonopoulos, 2004).

Our timeline could be substituted for a topology similar in structure to other multi-tier DHTs. At the core of this overlay (Figure 3) is a central super ring that organises a Chord ring of indexed keywords, splitting the index over the supernodes in the ring. Supernodes point to one or more keyword rings. Each keyword ring is a Chord ring of indexed values associated with a single keyword. Each node in the keyword ring holds a list of IP addresses of machines hosting resources matching the values and keyword for which the node has responsibility.

Searching for resources involves a two-step process. First, the super ring is traversed to find the supernode responsible for the keyword. The relevant keyword ring is then entered and the relevant node hosting the addresses of resource providers relating to the specified value is located. Searches in both rings are conducted using the standard Chord algorithms. By introducing similar information to that contained in local node lookup tables, queries can be directed straight into keyword rings, shortcutting the super ring. Preference lists can also be used in similar ways to those described above.

Figure 3 The multi-ring topology



Benefits in terms of number of messages required to resolve a query and update information can be realised by building the overlay on demand, thereby creating fewer, smaller-sized rings than in other multi-tiered DHT topologies which assume secondary-level rings are either already built or are created based on node locality (Garcés-Erice *et al.*, 2003, Mislove and Druschel, 2004). In these, representative nodes are added to the primary ring, effectively building the network upwards. The system itself has no control over how many sub-rings exist or how many nodes each contains. Following our methodology, the network is built downwards from the primary ring, with decisions on the size and number of rings being made by the system.

When a resource is registered under a new keyword in our topology, rather than immediately creating a keyword ring, the resource information is held on the supernode responsible for the keyword. A keyword ring is only built once either the number of resources registered against the keyword or the number of queries for the keyword have exceeded preset trigger levels. When one of these has been exceeded, nodes with available capacity are selected to become members of the ring from those listed in a separate ring containing all available nodes. The supernode responsible for the keyword will determine the number of nodes needed for the keyword ring and the capacity (primarily in terms of query processing load) each one will need.

Due to rings being built on demand and the size of the rings being controlled, fewer update/ maintenance and query lookup messages are required compared with either a single Chord ring or other multi-tiered Chord architectures.

Future work

Failure of certain nodes could currently break the timeline, so extra routing information must be added to give each supernode increased knowledge of its environment. This will give the side-benefit of traversal across the timeline in fewer messages. We envisage this will follow a similar scheme to Chord's successor lists.

Internet Research
Volume 14 · Number 5 · 2004 · 339-346

Pointers to other segments will be distributed throughout the timeline, reducing the reliance on the closing node of each segment. Since the message cost of inserting a supernode into the timeline is small, an increase in update traffic as a result of maintaining extra routing information should not have a substantial impact on the system.

We will create intelligent algorithms that will determine when checkpoints should be inserted and new segments created, rather than relying on static parameters defining, for example, how many supernodes should exist between checkpoints. This will enhance the adaptivity of the model to different network sizes and conditions, improving scalability, message costs and fault tolerance.

In addition to simple resource discovery, the model has potential uses as a distributed access control architecture. Segments can be viewed as autonomous administrative domains. Within each segment are a number of progressively higher security levels, with checkpoint supernodes controlling access to nodes further along the segment timeline. Methods for combining discovery and access control will be explored.

Conclusions

In our work so far we have demonstrated a method of providing a scalable resource discovery mechanism without the need for a single centralised index. We have shown that through selection of key parameters, nodes can be inserted and queries resolved in less hops than in DHT architectures such as Chord. Through a case study, we have explored how our model can be generalised into a methodology for developing efficient multi-tiered peer-to-peer networks.

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, we are able to remove the need for flooding the network with queries and the associated TTL counters, a key requirement in systems such as Gnutella. We can guarantee to find all available resources (if necessary) as nodes always point towards the answer and there is a single point-of-contact for each keyword.

We now continue our work to improve fault tolerance of the model by adding routing information and distributing pointers to multiple nodes, defining algorithms to dynamically adapt parameters involved in timeline construction, and explore the model's suitability as a distributed access control architecture.

References

Antonopoulos, N. and Shafarenko, A. (2001), "An active organisation system for customised, secure agent discovery", *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)*, Vienna, 2-5 July.

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*, pp. 365-76.

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 (IPTPS'03)*, Berkeley, CA.

Garcés-Erice, L., Biersack, E.W., Ross, K.W., Felber, P.A. and Urvoy-Keller, G. (2003), "Hierarchical peer-to-peer systems", *Parallel Processing Letters*, Vol. 13 No. 4, December, pp. 643-57.

Kelaskar, M., Matossian, V., Mehra, P., Paul, D. and Parashar, M. (2002), "A study of discovery mechanisms for peer-to-peer applications", *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems*, Berlin, May, pp. 444-5.

Mislove, A. and Druschel, P. (2004), "Providing administrative control and autonomy in structured peer-to-peer overlays", paper presented at the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04), San Diego, CA, 26-27 February.

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

Ritter, J. (2001), "Why Gnutella can't scale", available at: www.darkridge.com/~jpr5/doc/gnutella.html

Salter, J. and Antonopoulos, N. (2004), *An Efficient Fault Tolerant Approach to Resource Discovery in P2P Networks*, Computing Sciences Report CS-04-02, University of Surrey, Guildford.

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)*, San Jose, CA, 18-25 January.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M.F., Dabek, F. and Balakrishnan, H. (2003), "Chord: a scalable peer-to-peer lookup protocol for Internet applications", *IEEE/ACM Transactions on Networking*, Vol. 11 No. 1, February, pp. 17-32.