ROME: Optimising DHT-based Peer-to-Peer Networks

James Salter and Nick Antonopoulos

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

Abstract

Distributed Hash Tables (DHTs) have been used in Peer-to-Peer networks to provide key lookups in typically O(log *n*) hops whilst requiring maintenance of only small amounts of routing state. We propose ROME, a layer to be run on top of one such DHT to provide control over nodes joining the network. We show this can reduce further the hop counts in networks where available node capacity far exceeds workload, without the need to modify any processes of the underlying DHT protocol.

Keywords

Peer-to-peer networks, distributed hash tables, Chord protocol

1. Introduction

An important area of research into structured Peer-to-Peer (P2P) networks has been the use of Distributed Hash Tables (DHTs), which provide guaranteed lookups and typically O(log *n*) hop counts whilst maintaining only small amounts of routing state (Kelaskar et al, 2002). One well-known protocol is Chord (Stoica et al, 2003). Given a key, the Chord protocol allows for the lookup of the value associated with the key. It organises network nodes into a one-dimensional key space in the form of a ring. Nodes are given identifiers and become responsible for a key if their ID most closely follows the numerical hash of the key. Nodes maintain a finger table containing O(log *n*) pointers to other nodes in the ring. Key lookups are forwarded from node to node, effectively halving the distance at each hop between the current node and the node holding the desired key, allowing for key lookup in O(log *n*) hops.

We present ROME (Reactive Overlay Monitoring and Expansion), an additional layer running above the standard Chord protocol allowing control over the construction of the network via the selection and placement of nodes within the Chord ring. ROME provides a monitoring process to detect overloaded or failed nodes and actively seek either direct replacements or additional nodes to service the workload.

As we show in section 4, the size of a Chord ring has an impact on the number of hops it takes to resolve a lookup, with smaller rings having better performance. Our primary motivation is to provide a mechanism for controlling the size of the Chord ring to ensure it is the minimum size possible, therefore optimising the number of messages consumed. In addition to minimising lookup traffic, ROME also provides other benefits over standard Chord:

1. Node IDs are uniformly distributed around standard Chord rings, assuming uniform

workload per key. However, although the random hash function should ensure the distribution of *keys* is uniform, the *workload* per key can vary substantially since some keys may be more popular. Using ROME, nodes are given IDs to place them in areas of the ring with the greatest workload in order to provide better load balancing.
2. The Chord protocol provides no mechanism for controlling the quality of nodes joining the ring. Since nodes with short lifetimes/high probability of failure can increase the rate of lookup failures, ideally only the best machines should become members of the ring. ROME creates a pool of available machines from which the best nodes can be selected.
3. Chord provides no mechanisms for dealing with overloaded nodes/hotspots in network. Since we only add nodes to our ring when they are needed, ROME adds a process to monitor the workload placed on each node. This allows nodes to be added or replaced in the ring to deal with overload and minimise hotspots.

2. Related Work

Many extensions and modifications to the standard Chord protocol have been proposed, several of which share our aim of reducing the number of hops required to process a lookup. However, to our knowledge none of these do this by controlling the expansion of the Chord ring itself. In fact, ROME is complementary to these extensions. Using them together may provide even greater message savings.

Kaashoek and Karger (2003) combine Chord with de Bruijn graphs to produce Koorde, allowing for varying amounts of routing information to be stored and thus achieving hop counts between $O(\log n)$ and $O((\log n)/\log \log n)$, where n is the number of nodes in the network. It is clear that a reduction in n would also reduce Koorde hop counts. The authors also suggest that finding a system that is load balanced and degree optimal (optimising the amount of routing information stored) is an open question. With our control over node joins, small ring size and placement of nodes, we believe ROME can go at least some way to achieving this.

Single hop lookups are achieved by storing a global routing table on every node by Leong and Li (2004), but only in circumstances where network churn is low, something rarely achievable in realistic P2P implementations. Other researchers have explored creating hierarchies of multiple Chord rings. Garcés-Erice et al (2003) demonstrate reduction in hop counts through their assumption that high quality nodes are always available to add to the top layer of their hierarchy. Mislove and Druschel (2004) build sub-rings within administrative domains, exploiting node locality to reduce latency of lookups. Locality could also be exploited using our protocol in the single layer rings, becoming an extra node selection criteria (see section 3.2). Ratnasamy et al (2002) review several DHT-based routing algorithms and direct researchers toward several open questions. They suggest that node heterogeneity is extreme, which is a property we use to our advantage. By allowing only the best nodes to join the ring, we build a ring with less probability of failure than one including all available nodes.

3. ROME

ROME has been designed to provide extensions to the basic Chord protocol, whereby all Chord processes will operate as originally defined and are oblivious to the existence of our layer. This allows implementations built on Chord to utilise ROME without many changes.

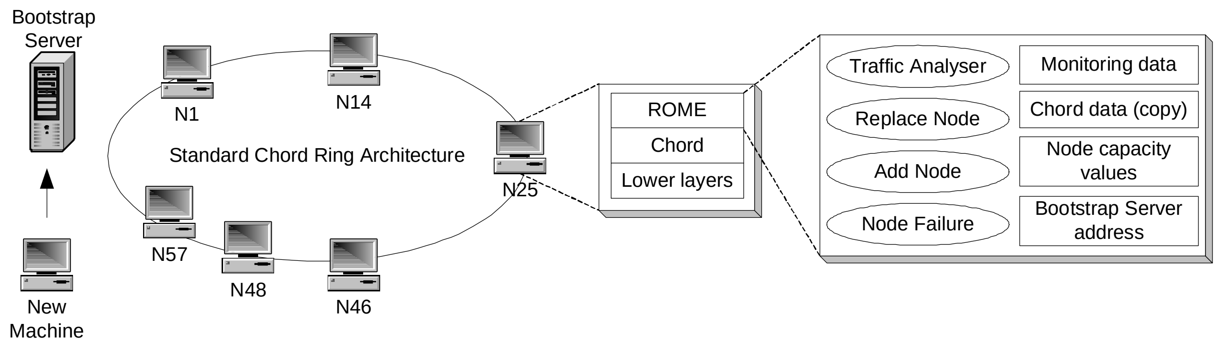

Figure 1: Architecture and Processes

Figure 1 illustrates an example network utilising Chord together with our extensions. ROME runs on each machine in the network and consists of a traffic analyser (allowing us to monitor Chord-related network traffic without needing to modify the underlying Chord protocol), processes to add or replace nodes and deal with detected node failures, plus data structures to store monitoring data, a copy of the node's Chord data (captured using the traffic analyser) and ROME specific data such as the address of the bootstrap server and the node's capacity. Additionally, we show a bootstrap server because in P2P networks a new machine must know the address of an existing node that it can use to establish a connection with the network. New machines contact a bootstrap server, a well-known machine that lists the addresses of the nodes currently connected to the network. Using ROME, the addresses of machines that have requested to join the network are held on the bootstrap server. Instances of ROME running on nodes already in the network query the bootstrap server for the addresses of machines to add to the network as they are required.

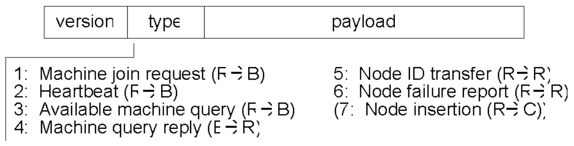
version
type
payload
1: Machine join request (R → B)
2: Heartbeat (R → B)
3: Available machine query (R → B)
4: Machine query reply (B → R)
5: Node ID transfer (R → R)
6: Node failure report (R → R)
(7: Node insertion (R → C))

Figure 2: Structure of a ROME message

The structure of ROME's text-based messages is shown in Figure 2. Along with the version number and message-specific payload, a type field indicates what category of message it is. Messages can be one of six types, with a seventh being used to communicate with Chord (this type follows the structure of a Chord message rather than a ROME message). Messages can be sent between **R**OME, the **B**ootstrap server and **C**hord, as indicated on the diagram.

3.1 Joining the Network

As would be the case for a standard Chord implementation using a server-based bootstrapping mechanism, a machine wishing to join the network contacts the bootstrap server. Our bootstrap server adds the address of the new machine in its node database, rather than providing the address of an existing network node. By preventing machines immediately joining the ring, we automatically filter out unreliable nodes that disconnect quickly. The traffic analyser running on the new machine will detect the join request message, triggering ROME to send the bootstrap server a type 1 message containing the capacity the machine is prepared to set aside for the network. In real terms, this capacity is a measure of memory (to store key and routing information) and processor time/cycles (for running the protocol processes). Depending on the implementation, these capacity values are either determined by the higher-level application or explicitly defined by the user of the machine. Each machine has three capacity values: target, the ideal workload placed on the node; threshold, above which a node is considered overloaded; and limit, the maximum workload a node can handle. On machines that have registered, ROME sends periodic type 2 heartbeat messages to the bootstrap server, which allow the server to evaluate and record the machine's reliability (many missed heartbeat messages indicate an unreliable machine with low uptime).

3.2 Expanding the Ring

The instance of ROME running on each node analyses the Chord-related traffic flowing through the node by using the traffic analyser. This allows it to monitor the workload on the node in relation to its capacity. When the current workload exceeds the node's capacity threshold (Figure 3), action needs to be taken as this node has become a hotspot. In real terms, this state can be reached in three ways: The node does not have capacity to store a new key, the node does not have capacity to cope with its query processing workload, or the node cannot cope with routing workload.

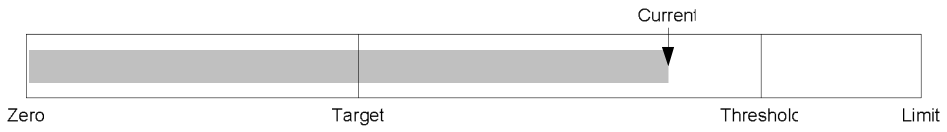

Figure 3: Current Workload vs Node Capacity

At first glance, the obvious action to take is to expand the Chord ring by adding an additional node from the pool of machines available listed on the bootstrap server. However, it may be that there is an available machine with higher capacity that could cope with the current node's full workload. Since our primary goal is to keep the ring as small as possible, a direct node replacement is preferable, with the addition of another node to share the current node's workload as a second option. The ring expansion process is summarised in Figure 4.

```
TargetCapacity = thisnode.CurrentWorkload * q
Replacement = BServer.FindMachine(TargetCapacity)
If (Replacement NOT null) {
   ReplaceNode(Replacement, thisnode.NodeID)
} else {
   TargetCapacity = (thisnode.CurrentWorkload - thisnode.TargetCapacity) * q
   Replacement = BServer.FindMachine(TargetCapacity)
   If (Replacement NOT null) {
      NewNodeID = SelectNodeID(thisnode.PredecessorID, thisnode.NodeID)
      AddNewNode(Replacement, NewNodeID)
   } else {
      // pause, then retry from start
      Wait()
}  }
```

Figure 4: Ring Expansion Pseudocode

When ROME has identified that the workload exceeds the capacity threshold, it sends a type 3 message to the bootstrap server requesting the address of a replacement machine with a target capacity greater than *current_workload* * *q* where *q*>1 (*q* is a parameter to allow for a small amount of fluctuation in demand). Providing there is a match, the bootstrap server replies with a type 4 message containing the address of a matching machine and removes the machine from its database. The node ID of the old node is transferred to the new machine via a type 5 message. Type 7 messages are then sent to the Chord protocol layer running on the old node's successor and predecessor nodes to alert them to update their references to the node's new IP address. Other nodes' finger table information will be updated as the standard Chord stabilisation routine is run. The migration of keys is handled by the Chord protocol.

If, on the other hand, the bootstrap server is unable to give the address of a machine matching the supplied capacity requirements, the second option of adding a new node to the ring to share the current node's workload must be utilised. ROME sends a similar type 3 message to the bootstrap server as before, this time for a machine with a target capacity of (*current_workload* - *currentnode_target_capacity*) * *q*. Once it has received a type 4 message containing the address of a machine to be added to the ring, it must organise for the keys of the current node to be shared between the current node and the new node. This is achieved by inserting the new node into the ring between the current node and its predecessor, giving the new node a node ID such that *Predecessor_ID* < *New_Node_ID* < *Current_Node_ID*. The actual node ID is chosen based on the cumulative workload generated by each key on the current node and how much workload must be moved to the new node. Figure 5 shows how 300 units of workload could be moved to a new node from the current node by (in the example) transferring keys 164-186 to the new node. Similar to above, a type 5 message informs the new node of its node ID. The new node is inserted into the ring by sending type 7 insert messages to the standard Chord protocol running on itself, its successor and its old predecessor. Again, key migration is handled by the Chord protocol.

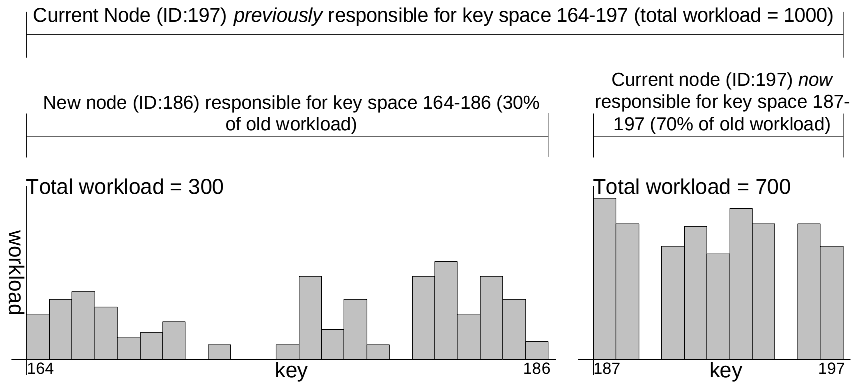

Figure 5: Example key transfer from current node to new node during node addition

3.3 Discovering and Replacing Failed Nodes

Identification of node failure by ROME is a relatively trivial process – a simple case of analysing relevant Chord traffic using the traffic analyser. However, since in standard Chord implementations $\log_2(n)$ nodes will have finger table entries pointing towards a particular node (where *n* is the number of nodes in the ring), there is the potential for the failure of a node to be identified simultaneously by instances of ROME running on several nodes. If any node was allowed to take corrective action on discovering a failure, situations could arise where multiple nodes all attempt to replace the failed one. Instead, we place control of the replacement process with ROME running on the failed node's immediate successor in the ring. Since ROME holds a copy of the Chord routing tables on each node, a type 6 node failure report message can be routed towards the failed node's successor by utilising Chord routing, albeit on the ROME layer rather than on the Chord layer. The ID of the successor can be found by searching for the failed node's ID + 1. ROME running on the successor node then performs the procedure described above to replace the failed node with an available machine from the bootstrap server. We assume that, as suggested by Stoica et al (2003), Chord will store a node's keys on *r* nodes following the node actually responsible for that key space, so that no information is lost if the node fails. Following a type 5 message to ROME on the replacement node informing it of its node ID (the same as the failed node's ID) and type 7 messages to the predecessor and successor, the relevant keys will be migrated to it by the standard Chord protocol.

3.4 Initialisation/Failure of the Bootstrap Server

Prior to the initialisation of an instance of the network, a bootstrap server must be defined. (We assume here the bootstrap server is external to the network itself but in implementations a machine could of course run the relevant protocols and also be eligible to become a network node). The bootstrap server starts with an empty node database. A machine wishing to join the network then contacts the bootstrap server. The ring is immediately created with this node, as per the standard Chord process. Since no keys will have been stored at this point, the node ID cannot be determined using the methods outlined in section 3.2. Therefore, the initial node is given node ID 0. With the network initialised, the standard processes then continue as defined previously.

The failure of the bootstrap server has no impact above the failure of a single server bootstrap mechanism in a standard Chord implementation. In both of these scenarios, the Chord ring remains functional as long as workload does not exceed capacity on each node, but additional nodes can no longer join or be replaced. In effect, in the absence of a bootstrap server a network running ROME simply degrades to the performance of a standard Chord implementation (also with a failed bootstrap mechanism).

4. Measures of Improvement

As shown in Stoica et al (2003), the number of hops needed to lookup a key in a static Chord ring where all finger table entries are correct is $\log_2(n)$ in the worst case and $\frac{1}{2}\log_2(n)$ on average, where *n* is the number of nodes in the ring. Therefore, it is obvious that lookups will take fewer hops in the average and worst cases in a smaller ring. We have used p2psim, a freely available P2P network simulator to demonstrate that a similar property holds for dynamic conditions, where nodes are joining, leaving and failing in the Chord ring. We simulated lookups in different sized Chord rings with nodes of varying mean lifetime. Each simulation was run for 8000 seconds with a mean of 10 seconds between lookups and the stabilisation process run every 200 seconds. The results are shown in Figure 6.

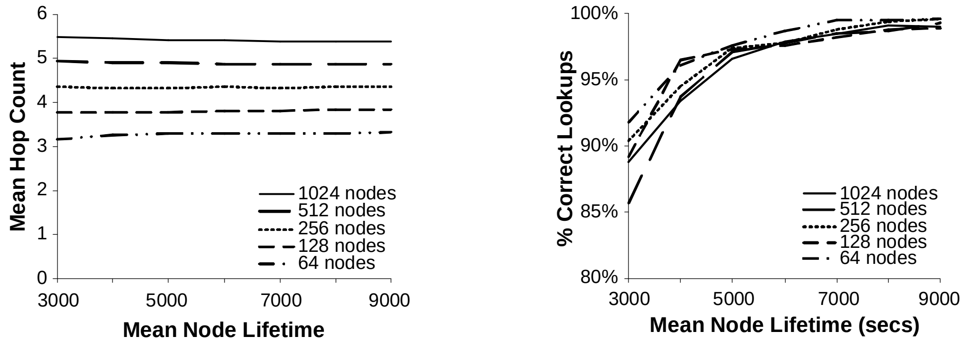

Figure 6: Effect of dynamic conditions on lookup failure and mean hop count

It is clear that smaller rings still generate fewer messages than their larger counterparts in dynamic conditions. In our simulations, the mean hop count for successful lookups remains very similar to the ½$\log_2(n)$ calculation for the average case for static rings. We also see that small rings have little impact on lookup failure. If anything, there is a general trend showing smaller rings giving higher percentages of correct lookups. More obvious is the impact node lifetime has on lookup success, showing the potential benefits of our selection process allowing only the highest quality nodes to join the Chord ring.

Next, we show a simplified example of where we can achieve a reduction in ring size. Assume we have a pool of 1million available machines each offering a maximum of 100 units of capacity, with the threshold capacity set at 95%. (These numbers have been chosen simply to demonstrate ROME's effects on a P2P network of reasonable scale). A standard Chord ring would automatically contain all of these nodes, thus its total capacity would be 100million units. In a ring running ROME, the available machines would be held at the bootstrap server until they were required. We increase network-wide workload, assuming that additional nodes will be added to the ring running ROME as the workload exceeds the existing ring capacity. Figure 7 plots the percentage ring capacity utilisation, the ring size (number of nodes) and mean hops per query (using the ½$\log_2(n)$ calculation from above for a static network) for both the standard Chord ring and the ring running our protocol. Total ring capacity is more effectively utilised when using ROME, principally because the ring size is controlled, only growing when necessary. Therefore, the capacity utilisation remains roughly constant around 95% (the individual node's threshold capacity). In turn, the ring size has an effect on the mean hops around the ring to resolve lookups. By keeping the ring small, we reduce the number of hops thereby reducing the messages transmitted and bandwidth consumed throughout the network. The results show that in general, where the total capacity provided by available machines vastly exceeds the network-wide workload, running ROME can create smaller sized rings consuming less messages than using an uncontrolled Chord ring.

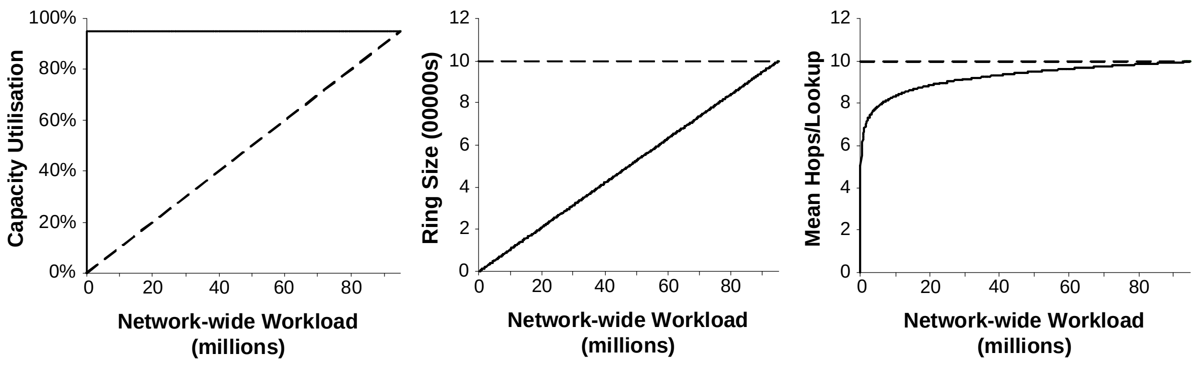

Figure 7: Comparison of a standard Chord ring (broken line) with a ring running ROME (solid line), both having 1 million available nodes

5. Conclusions and Future Work

We have shown that, by adding a layer to control node joins and processes to monitor and alleviate problems caused by overloaded and failed nodes, we are able to construct smaller than standard Chord rings with the benefit of smaller messaging costs throughout the network. Although we add more functionality to the bootstrap server, this is only utilised for node join and replacement functions. As with a standard Chord implementation, the ring is still capable of performing its core functions if the server becomes unavailable. Through simulation and calculation we have proved that construction of small rings is desirable, and demonstrated this is possible using ROME when the total capacity of machines available to become members of the ring exceeds the workload placed on the ring.

Currently we add nodes into the Chord ring when they are needed, but provide no mechanism for removing nodes from the ring due to under-utilisation. This will require a more global view of the ring than is available to an individual node, so could potentially become a function of the bootstrap server. This could then allow for the IDs of nodes to be changed, moving them backwards or forwards within the ring key space to better balance workload, rather than adding or replacing nodes. We expect patterns to emerge allowing us to establish a set of rules dictating whether a node should be moved, added, removed or replaced (for example an under-utilised node followed by an overloaded node could mean the under-utilised node should be moved forwards in the ring to take on more workload). Additionally, we envisage creating a network of bootstrap servers, potentially providing services to multiple networks and increasing scope for fault tolerant pools of available nodes. Whilst we have discussed how ROME can be utilised above Chord, we believe it could be applied to other DHTs such as CAN, Pastry or Tapestry (Kelaskar et al, 2002). Providing message costs are proportional to the number of nodes in the structure, reductions will be possible since ROME seeks to minimise the number of nodes. We will provide further details of how it could be applied to other DHTs in our future work.

7. References

Garcés-Erice, L., Biersack, E.W., Ross, K.W., Felber, P.A. and Urvoy-Keller, G. (2003), "Hierarchical peer-to-peer systems", in *Parallel Processing Letters*, Vol. 13, No. 4, pp643-657.

Kaashoek, M.F. and Karger, D.R. (2003), "Koorde: A simple degree-optimal distributed hash table", in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03),* Berkeley, CA, 20-21 February 2003.

Kelaskar, M., Matossian, V., Mehra, P., Paul, D. and Parashar, M. (2002), "A study of discovery mechanisms for peer-to-peer applications", in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems,* Berlin, Germany, May 2002, pp444-445.

Leong, B. and Li, J. (2004), "Achieving one-hop DHT lookup and strong stabilisation by passing tokens", in *Proceedings of the 12th International Conference on Networks (ICON2004),* Singapore, November 2004.

Mislove, A. and Druschel, P. (2004), "Providing administrative control and autonomy in structured peer-to-peer overlays", in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04),* San Diego, CA, 26-27 February 2004.

Ratnasamy, S., Shenker, S. and Stoica, I. (2002), "Routing algorithms for DHTs: some open questions", in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02),* Cambridge, MA, 7-8 March 2002.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., 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, pp17-32.