An Evaluation of EpiChord in OverSim

Jamie Furness, Farida Chowdhury, Mario Kolberg

Computing Science and Mathematics,
Universty of Stirling, Stirling, Scotland
{jrf,fch,mko}@cs.stir.ac.uk

Abstract. EpiChord is a Distributed Hash Table (DHT) algorithm which supports data storage/retrieval in large scale distributed systems. It removes the typical *O(logn)*-state-per-node restriction imposed by the majority of other DHT topologies by employing a reactive routing state maintenance strategy that amortizes network maintenance costs into lookup queries. Under ideal condition, EpiChord's lookup performance can approach O(1) hops - with maintenance costs comparable to traditional multi-hop DHTs. This paper presents an implementation of EpiChord in OverSim, and validates the performance of our model against the performance reported in the original EpiChord paper. We also present some adjustments to the algorithm to remove a discrepancy and then compare our modified results with the original ones. Finally, we present additional results showing the EpiChord algorithm is stable over time and performs well for larger networks.

1 Introduction

Distributed Hash Tables (DHTs) [2] supported Peer-to-Peer (P2P) applications are an ideal substrate for building large scale distributed systems because they are self-organizing, adaptable and scalable and offer efficient routing between nodes within a bounded number of hops. EpiChord [1] is a DHT lookup algorithm which demonstrates that node state restrictions can be relaxed which were imposed by the majority of other DHT algorithms by using a reactive routing state maintenance strategy. Nodes piggyback additional network information on lookup queries to keep their routing state up-to-date. This makes EpiChord ideally suited to large scale environments. This paper discusses an implementation [24] of EpiChord within the OverSim Simulator [3]. The model is validated against the original EpiChord paper. Specifically, the contributions of the paper are as follows:

- An independent evaluation of EpiChord, by comparing results from our simulation model to the results presented in the original EpiChord paper.
- Performance evaluation in multiple scenarios, defined in the original paper, which test both the routing and maintenance algorithms of the model.
- Amendments to the original model together with a comparison of the results obtained from our model against the corrected results from the original model.
- Performance evaluation of EpiChord in larger networks and for longer simulations.
- A freely available EpiChord model in OverSim.
- A review of available simulators.

The original implementation of EpiChord was a model for the SSFNet simulation framework [4] which is not publicly available. The authors are not aware of other EpiChord models which are publicly available. In other work [5] we have validated the models for both Chord and Pastry in OverSim.

The remainder of the paper is structured as follows: Section 2 discusses related work, Section 3 provides an overview of the EpiChord DHT algorithm, Section 4 compares network simulators, Section 5 discusses implementation details of the EpiChord model in OverSim, Section 6 presents an evaluation of results after changes to the original EpiChord model. Section 7 presents validating results from our EpiChord model as well as results demonstrating EpiChord's scalability. Section 8 concludes this paper.

2 Related Work

A large number of multi-hop structured Peer-to-Peer (P2P) algorithms have been proposed [2]. These algorithms are characterized by O(log N) hop count. Because each overlay hop translates to potentially many hops in the underlying network, multi-hop overlays have a relatively poor latency characteristic for connecting large numbers of peers. Consequently, systems have been developed to trade-off latency for larger routing tables. However these designs lead to increased network traffic for managing the larger routing tables. Thus efficient overlay maintenance in O(1)-hop (one-hop) overlays is an important research question. Two techniques have emerged [2] for maintaining routing tables in overlays: active stabilization where peers have fixed communication to maintain a target routing table accuracy, and opportunistic updating where routing table maintenance depends on lookup load and available bandwidth.

An example active stabilization algorithm is EDRA (Event Detection and Reporting Algorithm) used in the D1HT one-hop overlay [8]. EDRA has been proposed to give reasonable message rate for high levels of routing table accuracy. For example, D1HT has up to an order magnitude lower maintenance bandwidth usage compared to the OneHop [10], another active stabilization one-hop overlay. EDRA* [11] offers some improvements over EDRA. Examples of opportunistic overlay maintenance include EpiChord [1] (used in this paper) and Accordion [9].

Kelips [7] is a O(1)-hop overlay which uses an epidemic multicast protocol for exchanging overlay membership and other soft state between nodes. Such a protocol consists of two sub-protocols: a multicast data dissemination protocol and a gossip protocol to exchange message history for reliability purposes.

Accordion [9] is a variable hop overlay, in which a peer limits its routing table update message level based on its available bandwidth. During periods of low bandwidth, routing table accuracy can approach that of multi-hop overlays while for higher bandwidth, routing table accuracy reaches one-hop. Accordion uses recursive parallel lookups so as to maintain fresh routing table entries in its neighborhood of the overlay and reduce the probability of timeout. Note that recursive parallel lookups create more load on the target peer compared to iterative parallel lookups.

3 EpiChord Background

EpiChord [1] is a DHT algorithm which can achieve one-hop lookup performance under lookup intensive workloads, and at worst case $O(\log_2(N))$ hop, as offered in many multi-hop networks. As the name suggests, EpiChord is based on the Chord DHT [6]. Like Chord, EpiChord is organized in a one-dimensional circular address space where each node is assigned a unique node identifier. The node responsible for a key is the node whose identifier most closely follows the key. In addition to maintaining a list of k succeeding nodes, EpiChord also maintains a list of the k preceding nodes. Instead of maintaining a finger table, as in Chord, EpiChord maintains a cache of nodes. Nodes update their cache by observing lookup traffic, and add an entry anytime they learn of a node not already in the cache. Nodes in the cache each have a timeout, resulting in stale nodes being removed.

In general terms EpiChord can be thought of as Chord with a cache of extra node addresses. As such the routing algorithm is similar to that in Chord. With a well populated cache this results in lookup performance of one hop. Under high churn the performance drops to that of Chord, $O(log_2(N))$ hops in the worst case.

3.1 Lookup Algorithm

EpiChord uses an iterative lookup algorithm, as it avoids sending redundant queries when using parallel requests. It also allows the querying node to receive all information related to the query path, and hence updates its cache with new entries. To lookup a data item with the key *id*, a node will initiate *p* queries in parallel - to the node immediate succeeding *id* and to the *p-1* nodes preceding *id*. When queried, a node will respond as follows (*l* and *p* are both system parameters):

- If it owns *id*, it will return the value associated with *id*, and information on its predecessor and successor.
- If it is a predecessor of *id* relative to the querying node, it will provide information about its successor and the *l* best next hops towards the destination.
- If it is a successor of *id* relative to the querying node, it will provide information on its predecessor and the *l* best next hops towards the destination.

When a reply is received, further queries are dispatched in parallel if the querying node learns about any node closer to the target *id* than the best successor and predecessor nodes that have already responded.

3.2 Cache Invariant

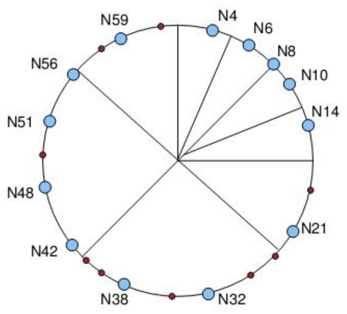

Fig. 1. Example of slicing of address space for N8.

To guarantee worst case lookup performance of $O(log_2(N))$ each node divides the address space into two sets of exponentially smaller slices, as seen in Fig. 1. Each node maintains their cache such that every slice contains at least $\frac{j}{1-\gamma}$ cache entries at all times, where j is a network parameter and γ is a local estimate of the probability that a cache entry is out-of-date. Nodes periodically check their cache slices to ensure that there are sufficient unexpired cache entries. To calculate γ, each node keeps track of n_p, the number of messages sent, and n_t, the number of messages which timed out. γ is calculated using n_t / n_p. In addition, n_p and n_t are periodically (when the cache is flushed) multiplied by a network parameter δ to obtain exponentially weighted moving averages.

3.3 Routing Table Updates

Each node periodically probes their immediate neighbours to ensure that they are still alive. The delay between these stabilization attempts is calculated based on the observed lifetime of nodes in the finger cache. For this reason the finger cache also contains a map of dead nodes, and the observed lifetime is calculated by taking the time between first learning of the node (s_{start}) until learning of its death (s_{end}). The observed lifetime for each dead node is averaged, and the obtained estimate is then multiplied by the lifetime estimate multiplier, ω, to calculate when the next stabilization attempt should be scheduled.

$$s = \frac{\sum s_{end} - s_{start}}{n} \cdot \omega \qquad (1)$$

In case where the sample size, n, is less than 5, the stabilization interval is simply set to the network parameter s.

With active propagation, nodes will inform their neighbours of any detected changes in the successor or predecessor lists as soon as they happen, rather than waiting for the next stabilization attempt. This increases the maintenance bandwidth when under high churn, however also results in more accurate successor and predecessor lists, and hence fewer false-negatives.

If a node has an outdated view of the local key space that they are responsible for, they may fail to respond correctly to all queries. By including their believed predecessor and successor in the query response, the querying node can either make a step towards the destination or, if the believed predecessor does not respond, determine that the responsible node is dead. This false-negative detection allows the querying node to resolve the lookup correctly. If a false-negative is detected, the querying node will immediately inform the new responsible node that their predecessor has failed and now they should be responsible for the requested key.

4 Review of Simulators

Before deciding on OverSim, a detailed review of other available and active P2P network simulators was carried out. A summary of these tools is provided in Table 1.

PeerSim [6] is written in Java. Its main focus is to provide high scalability and can handle a network of up to 10^6 nodes. However, this scalability comes at the cost of not including a model of the behavior of the underlying communication network, e.g. TCP/IP stack and latencies. P2PSim [13] is a discrete event simulator for P2P overlays written in C++. It supports Chord, Accordion, Koorde, Kelips, Tapestry, and Kademlia. However, these implementations are specific to P2PSim and do not model all features of the protocols. P2PSim has been simulated with up to 3,000 nodes using the Chord implementation. This simulator is largely undocumented and therefore hard to extend.

Overlay Weaver [14] is a toolkit for P2P Overlays written in Java. It has been tested with tens of thousands of nodes (their website quotes 300,000). Chord, Kademlia, Pastry, Tapestry and Koorde are available. The simulations have to be run in real-time environments and there is no statistical output which makes its use very limited. PlanetSim [14] is a discrete event simulation framework for both structured and unstructured overlays, written in Java. It has a modular, well-structured architecture and services can be re-used for other overlays. Chord and Symphony models exist and can consist of up to 100,000 nodes. However, it provides rather limited support to collect statistics. It also has a very simplified underlying network layer without any consideration of bandwidth and latency costs.

Table 1. A comparison of available active P2P simulators.

Simulator	P2P Protocols	Network size	Language
PeerSim	Collection of internally developed P2P models	$>10^6$	Java
P2PSim	Chord, Accordian, Koorde, Kelips, Tapestry, Kademlia	3000	C++
Overlay Weaver	Chord, Kademlia, Koorde, Pastry, Tapestry and FRT-Chord	Tens of thousand	Java
PlanetSim	Chord, Symphony	100,000	Java
NS2	Gnutella	N/A	C++/OTcl
SSFNet	Chord, EpiChord	33,000	Java/C++/DML
OverSim	Chord, Kademlia, Pastry, Bamboo, Broose, Gia	100,000	C++
PeerfactSim.Kom	CAN, Chord, Kademlia, Gia, C-DHT, Gnutella 0.4/0.6, Pastry	50,000	Java
D-P2P-Sim+	Chord	400,000	Java

NS2 [16] is a discrete-event simulator that provides substantial support for simulation of lower layer protocols. Only one P2P protocol, Gnutella, is available in NS2. Simulations in NS2 are constructed using C++ and OTcl. It is mostly used for small networks and is generally unsuitable for large scale P2P overlay networks.

SSFNet [4] is a discrete-event simulation framework written in Java and C++. This framework is built on the Scalable Simulation Framework (SSF) and uses the Domain Modelling Language (DML) to configure networks. Chord and EpiChord have been implemented in SSFNet. There is a claim that SSFNet manages to run models with 33,000 nodes, however, the authors of the original EpiChord paper [1] and ourselves could not simulate networks with more than 10k nodes.

OverSim [1] is an open-source P2P simulation framework for the OMNeT++ simulation environment. It provides a generic lookup mechanism and an RPC interface to facilitate additional protocol implementations. It allows large-scale simulations of simplified networks as well as complex heterogeneous underlay networks. Several P2P algorithms such as Chord, Kademlia, Bamboo, Broose, Koorde, NICE, NTree, Pastry, and GIA have been implemented in OverSim. Models can scale to over 100,000 nodes. More comprehensive surveys of P2P network simulators can be found in [19,20].

PeerfactSim.Kom [17] is a discrete event based P2P simulator environment. Its focus is on being extendable and on large scale network models. This simulator offers the potential to model different types of peer-to-peer systems including distributed CDNs, streaming applications and overlay systems. It comes with a built-in churn generator. The simulator includes models of lower layers but does not yet include TCP.

D-P2P-Sim+[18] is a distributed simulation environment which employs multi-threading, asynchronous message passing and distributed environment with graphical user interface. There is little information on this simulator besides a short paper and poster. These report simulated network sizes of up to 400,000 nodes. It seems the only implemented overlay algorithm is Chord. However, the system is extendible and other algorithms could be implemented. Multiple computers running the simulator may be interconnected to achieve larger simulated network sizes.

Based on this study OverSim was selected for our experimentation due to its flexibility with respect to underlay characteristics and possible high scalability.

5 Oversim Implementation

OverSim [3] is designed as a modular simulation framework, with many common overlay features implemented as part of a generic base overlay class. OverSim provides message passing using Remote Procedure Calls (RPC), and supports both iterative and recursive routing. Applications within OverSim are split into multiple tiers, allowing an application to sit on-top of another application. These applications are implemented as modules and interface with overlays through the Key-Based Routing (KBR) API [21], which represents basic capabilities common to all structured overlays. As mentioned above, OverSim provides a number of different network models, for both structured and unstructured overlays. The OverSim architecture is illustrated in Fig. 2.

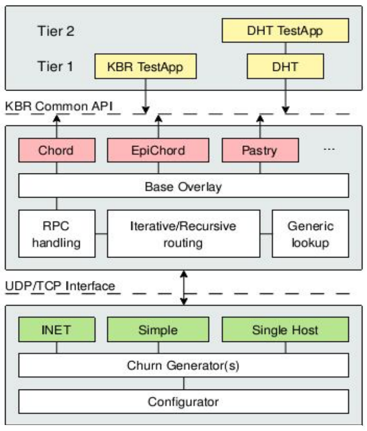

Fig. 2. Oversim architecture

At the lower layer OverSim provides multiple underlay models to allow for inclusion of specific underlay characteristics in the simulation (at a cost of scalability), or underlay abstraction for increased scalability. Using the simple model, data packets are sent directly from one node to another by using a global routing table. The INET underlay model includes simulation models for all network layers. The single host underlay allows for simulation of a single node, connected to other OverSim instances over a real network.

Below we discuss some alterations which we made to the original EpiChord protocol when implementing it as an OverSim module.

5.1 Node Join Protocol

In the original EpiChord algorithm, upon receipt of a join request a node will instantly update their predecessor list and finger cache to include the joining node. In our implementation we found this was occasionally causing messages to be routed to nodes who are still in the process of joining, and not yet ready to correctly handle requests. To solve this issue we implemented a three-way handshake. In our implementation the joining node will send a final acknowledgment when they are ready to handle requests, indicating they can now be safely added as a predecessor.

5.2 Lookup Algorithm

The OverSim framework provides modules for iterative and recursive routing, as can be seen in Fig. 2, with support for parallelism. While this makes implementation of many overlays easier and reduces duplicated code, only certain parts of the module can be easily overridden. This was a problem for EpiChord, primarily due to the non-linear order in which nodes are to be queried, and EpiChord's ability to check for false negative responses. To implement these features we had to make changes to the iterative routing module, allowing us to override additional parts of the module with code specific to EpiChord.

6 Results - Changes to the Original Model

6.1 Application layer Lookups

In the original EpiChord model all lookup types (JOIN, MAINTENANCE, and APPLICATION) are included when calculating results. The KBRTestApp in OverSim only includes lookups it has initiated (APPLICATION) in the results. We feel this is actually a more useful metric for anyone wishing to build on-top of EpiChord, so we instead recalculated the results from the original model using only APPLICATION lookups. A comparison of the average path lengths can be seen in Fig. 3; the other metrics remained unchanged.

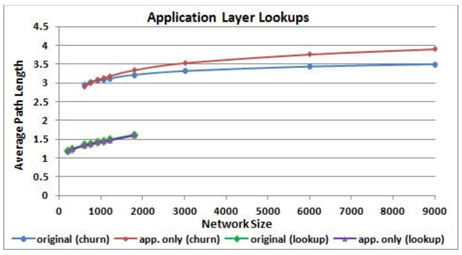

Fig. 3. Comparison of average path length with APPLICATION lookups only vs. all lookup.

In [22], authors proposed two generic classes of workloads: *lookup intensive* and *churn intensive*. These metrics were adopted by the EpiChord authors for experimentation. For the purposes of validating our model, we also adopt these two metrics. In the *lookup intensive* workload, node lifetimes are exponentially distributed with a mean of 10 minutes, and each node performs lookups on average every 0.5 seconds. In this scenario the background maintenance traffic is negligible compared to the active lookup rate. In the *churn intensive* workload, node lifetimes are again exponentially distributed with a mean of 10 minutes, however this time each node only performs lookups on average every 100 seconds. In this scenario the lookup rate is so low, most of the lookups captured are lookups arising from node joins and cache maintenance.

Fig 3 shows the average path length remains unchanged for the *lookup intensive* workload. This is to be expected, as the lookup intensive workload is dominated by APPLICATION lookups. In the churn intensive workload we see a rise in average hop count as the network size increases; this is because the result was originally dominated by JOIN and MAINTENANCE lookups, which tend to be for closer keys.

6.2 Fixing p

In the source of the original model we encountered a minor mistake[1], which, in many cases, resulted in *p+1* parallel requests being generated - rather than the supposed maximum of *p*. Results comparing the average path lengths and success rates when *p=1* can be seen in Fig. 4.

From these results we observe a rise in average path length, and a small decline in lookup success rate, for both workloads. We also observe a drop in the size of nodes cache tables, which increases with the network size. This is to be expected, as fewer queries are dispatched and hence fewer new nodes are discovered.

[1]When receiving a timeout or negative response, further queries are dispatched while *pending* <= p_{max}, resulting in $p_{max}+1$ pending queries.

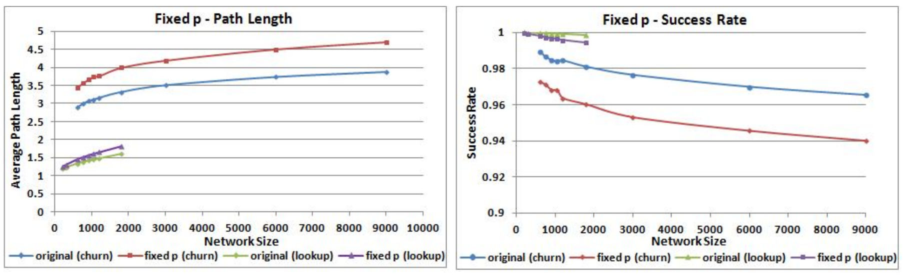

Fig. 4. Average path length and success rate with fixed p.

7 OverSim Results

To match the original scenarios, lookups were performed throughout the entire simulation, with measurements taken from the very beginning. OverSim, by default, only starts performing lookups and recording measurements once the network has reached the desired size, however this is configurable in the settings.

An overview of the simulation parameters can be found in Table 2. When we refer to results from the original model, we refer to the results generated after taking the changes in Section 6 into account. All results are averages of 5 simulation runs.

Table 2. OverSim simulation parameters.

Description	Lookup Intensive	Churn Intensive
Lookup Interval	0.5s	100s
Network Size	{600,…..,2000}	{600,…..,2000}
Lifetime Mean	600s	600s
Stabilize Delay	60s	60s
Cache TTL	120s	120s
Cache Flush Delay	20s	20s
Cache Check Multiplier	3	3
Measurement Time	3000s	3000s
Neighbour list size	4	4
Redundant nodes, *l*	3	3
Parallelism, *p*	1,3,5	1,3,5
Required nodes/slice, *j*	2	2
Lifetime multiplier, ω	0.5	0.5
Slice multiplier, δ	0.5	0.5

7.1 Finger Cache State

During simulation we measure the average finger cache size for each node, as well as the average accuracy of each node's finger cache. The accuracy is a measure of how many nodes in the finger cache are actually still active within the network.

We observe an average finger cache accuracy of 87% across all network sizes and both scenarios - almost identical to that of 87.5% reported in the original paper.

As expected the finger cache size observed in the *lookup intensive* workload is much larger than that in the *churn intensive* workload, due to the extra node information received within lookup messages. The observed finger cache size for varying network sizes under a lookup intensive workload and churn intensive workload can be seen in Fig. 5.

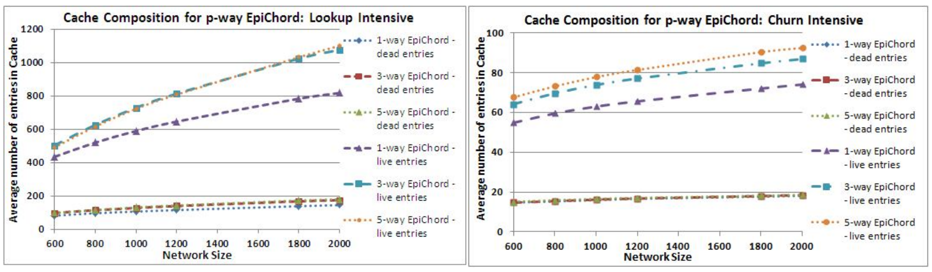

Fig. 5. Cache composition for p-way EpiChord under lookup intensive and churn intensive workload.

7.2 Lookup Success Rate

Every lookup performed can be classified into one of four categories:

- Success: The node responsible for the requested key responds positively.
- Failure: No positive response received and no more viable candidates, or reached the maximum hop/time limit
- False-positive: A node has responded positively but is not responsible for the requested key.
- False-negative: A node did not respond positively but should be responsible for the requested key.

By using false-negative detection, described in Section 3.3, nodes can detect and handle false-negatives; ultimately they are treated as successful lookups.

The observed success rate for both lookup intensive and churn intensive workloads is shown in Fig. 6. Here we use column diagram to show the success rate for *p-way* parallel queries (*p=1,3,5*) for different network sizes up to 2000 nodes.

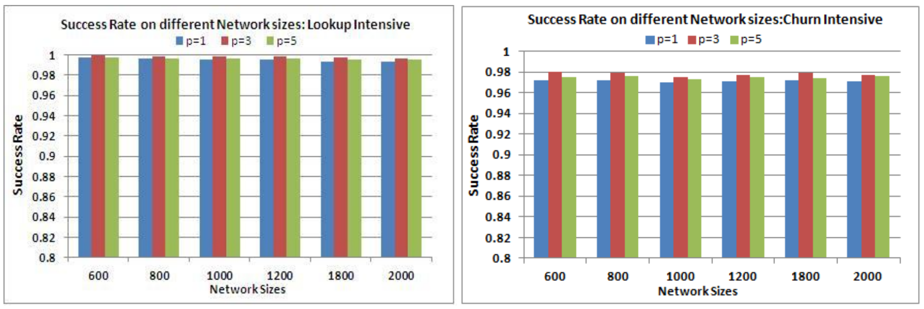

Fig. 6. Comparison of Success Rate for p-way EpiChord for varying network sizes under lookup intensive and churn intensive workload.

As shown in Fig. 6, the lookup success rate is marginally higher for lookup intensive workload than for the churn intensive workload. This is expected as under the lookup intensive workload, the larger number of lookups helps to keep the routing state up-to-date whereas for the churn intensive workload, the information propagation rate is lower. Increasing parallelism has only a very slight effect on the success rate. It appears that the lookup improvement is not worth the extra cost of the parallel lookups. The success rates for *p=5* is marginally lower than for *p=3*. This rather counter intuitive behavior has also been observed in the original paper and is due to the *5-way* network generating fewer cache-refreshing lookups than a *3-way* EpiChord network.

7.3 Lookup Path Length

For each successful lookup performed we also measure the path length - the number of hops taken to find the final destination. Fig. 7 shows the observed path length for both lookup intensive and churn intensive workloads. We observe that in the lookup intensive workload, the hop count varies from 1.1 to 1.4 in both *3-way* and *5-way* EpiChord networks, which signifies that each node has almost complete routing table information and thus allows passing messages nearly in one hop. On the other hand, the hop count varies from 2.8 to 3 under churn intensive workloads with fewer lookups which also satisfies the *O(log n)*-hop lookup performance as depicted in the original paper. Again, the results suggest that an increased level of parallelism in the lookups only marginally improves the hop count, whereas the increased number of lookups issued in the lookup intensive workload has a much more pronounced positive effect.

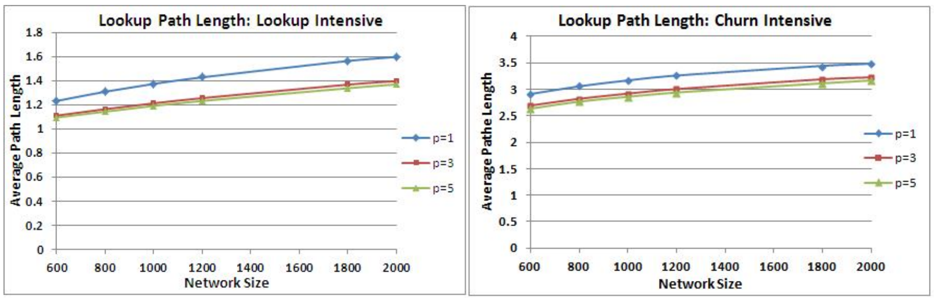

Fig. 7. Comparison of Lookup Path Length for p-way EpiChord for varying network sizes under Churn Intensive and Lookup Intensive workload.

7.4 Stability and Scalability

We measured the stability of the EpiChord model in OverSim. Fig. 8(a) shows the measurement phase vs. success ratio graph for *p=1, 3, 5.* In OverSim, during the measurement phase, the statistics are collected. Our model has been tested up to 100,000s and demonstrates that the model is stable after an initial period of between 10000s (for p=1) and 20000 (for p=3,5). EpiChord also has been tested for scalability in terms of network size for scenarios with 5,000 - 20,000 nodes. Fig. 8(b) shows the results for lookup success ratio for different network sizes. This set of results means that the network size does not affect the success rate of EpiChord. As before, p only improves the performance in a rather minor way.

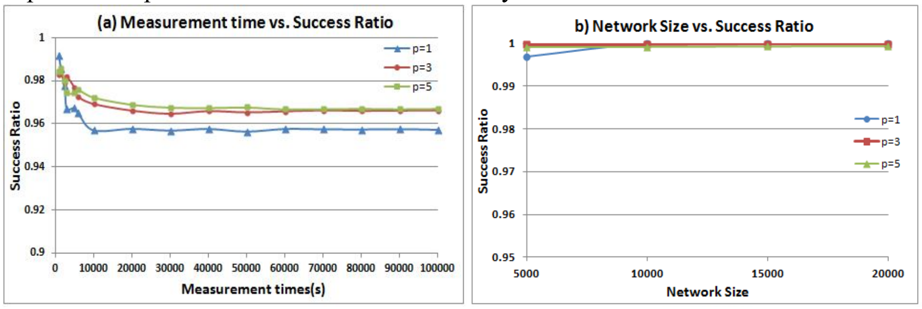

Fig. 8. a) Success Ratio of EpiChord for varying measurement times for *p=1,3,5* demonstrating the stability of the model; b) Average Success Ratio of EpiChord for networks with 5,000 to 20,000 nodes.

8 Conclusion

This paper presented our OverSim EpiChord model, and validated it by comparing our results against the performance of the original EpiChord model. The results for our model closely match those from the original model, supporting the claim that our model is a valid implementation of the EpiChord algorithm. We have then presented amendments to the model and investigated the effects on the performance of the model. Furthermore we have shown that EpiChord and our model in OverSim is stable over an extended period of time. We have also demonstrated that EpiChord achieves excellent results for larger networks. EpiChord's performance is strongly influenced by the number of lookups issued by the nodes as routing table information is attached to lookup return messages. Thus an increased number of lookup message improve the performance of the network, whereas an increased level of parallelism only marginally improves performance. Due to its excellent lookup performance for large scale networks, EpiChord appears well suited to support large distributed environments.

Separately, we have used this model to simulate the effect of different lookup traffic setups, and high node churn to investigate EpiChord's suitability for use in mobile networks [23].

References

1. B. Leong, B. Liskov, and E. D. Demaine. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. Computer Communications, Elsevier Science, Vol. 29, pp. 1243-1259.
2. K. Dhara, Y. Guo, M. Kolberg, X. Wu, Overview of Structured Peer-to-Peer Overlay Algorithms, Handbook of Peer-to-peer Networking, Springer, 2009.
3. I. Baumgart, B. Heep, S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. 10th IEEE Global Internet Symposium (GI '07), May 2007.
4. The SSFNet project. Accessed 01-August-2012. [Online]. Available:http://www.ssfnet.org/
5. J. Furness, M. Kolberg, Considering complex search techniques in DHTs under churn, in: 2011 IEEE Consumer Communications and Networking Conference (CCNC), IEEE, 2011.
6. I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Conf on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01), 2001. ACM.
7. I. Gupta, K. Birman, P. Linga, A. Demers, R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS '03), 2003.
8. L. Monnerat, C. Amorim. D1HT: A Distributed One Hop Hash Table. 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), April 2006.
9. J. Li, J. Stribling, R. Morris, M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. Symposium on Networked System Design and Implementation (NSDI) 2005.
10. A. Gupta, B. Liskov, R. Rodrigues. Efficient routing for peer-to-peer overlays. 1st Symposium on Networked Systems Design and Implementation (NSDI), 2004.
11. J. Buford, A. Brown, M. Kolberg. Analysis of an Active Maintenance Algorithm for an O(1)-Hop Overlay. IEEE Globecom 2007.
12. PeerSim P2P Simulator. Accessed 05-Jan-2013. http://peersim.sourceforge.net.
13. P2Psim: A Simulator for Peer-to-Peer (P2P) Protocols. http://pdos.csail.mit.edu/p2psim/
14. K. Shudo, Y. Tanaka, S. Sekiguchi. Overlay Weaver: An Overlay Construction Toolkit, Computer Communications,Vol.31, Issue2, pp. 402-412 (2007).
15. PlanetSim: An Overlay Network Simulation Framework. http://planet.urv.es/planetsim
16. The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/

17. D. Stingl, C. Groß, J. Rückert, L. Nobach, S. Kovacevic, R. Steinmetz. PeerfactSim.KOM: A Simulation Framework for Peer-to-Peer Systems, Intl. Conf. on High Performance Computing & Simulation (HPCS), 2011.
18. S. Sioutas, K. Tsichlas, G. Papaloukopoulos, Y. Manolopoulos, E. Sakkopoulos. A novel Distributed P2P Simulator Architecture: D-P2P-Sim. ACM Intl. Conf. on Information and Knowledge Management (CIKM), Hong Kong, 2009
19. A. Brown and M. Kolberg. Tools for peer-to-peer network simulation. Internet-Draft Version 00, IETF, January 2006.
20. S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A Survey of Peer-to-Peer Network Simulators. In The Seventh Annual Postgraduate Symposium, Liverpool, UK, 2006.
21. F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz. Towards a Common API for Structured Peer-to-Peer Overlays. Peer-to-Peer Systems II, 2735:33–44, 2003.
22. J. Li, J. Stribling, F. Kaashoek, R. Morris, and T. Gil. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs under Churn. In INFOCOM, 2005.
23. F.Chowdhury, M.Kolberg. An Investigation of EpiChord with high Node Churn. Submitted.
24. J. Furness, F. Chowdhury, M. Kolberg. EpiChord model for OverSim. http://www.cs.stir.ac.uk/~fch/EpiChord_Model/