0 to 10k in 20 seconds: Bootstrapping Large-scale DHT networks

Jae Woo Lee*, Henning Schulzrinne*, Wolfgang Kellerer† and Zoran Despotovic†
* *Department of Computer Science, Columbia University, New York, USA*
{jae,hgs}@cs.columbia.edu
† *DoCoMo Communications Laboratories Europe, Munich, Germany*
{kellerer,despotovic}@docomolab-euro.com

***Abstract*—A handful of proposals address the problem of bootstrapping a large DHT network from scratch, but they all forgo the standard DHT join protocols in favor of their own distributed algorithms that build routing tables directly. Motivating their algorithms, the proposals make a perfunctory claim that the standard join protocols are not designed to handle the huge number of concurrent join requests involved in such a bootstrapping scenario. Moreover, the proposals assume a pre-existing unstructured overlay as a starting point for their algorithms. We find the assumption somewhat unrealistic.**

We take a step back and reexamine the performance of the standard DHT join protocols. Starting with nothing other than a well-known bootstrap server, when faced with a large number of nodes joining nearly simultaneously, can the standard join algorithms form a stable DHT overlay? If so, how quickly? Our simulation results show that Chord and Kademlia's join protocols can actually handle the bootstrapping scenario quite well. For 10,000 nodes joining at a rate of 1,000 nodes per second, Chord and Kademlia took less than 20 and 15 seconds, respectively, to form a stable overlay. The Chord join protocol, however, requires a slight modification for fast bootstrapping. We elucidate the reason why the modification is necessary.

I. Introduction

Structured overlay networks based on distributed hash tables (DHT) have become a popular substrate for large-scale peer-to-peer (p2p) systems. DHTs provide efficient lookup of distributed data by assigning numerical keys to the peer nodes and the data items, storing each data item at the node whose key is the closest to the item's key, and locating in a small number of hops the node responsible for any given key. In order to achieve the small number of hops–usually a logarithm of the number of nodes–DHTs dictate overlay topology by imposing certain constraints on the entries in each node's routing table. In Chord [1], for example, the keys are m-bit integers arranged in a circle modulo 2^m, and each node's routing table is filled in such a way that the ith routing table entry is at a distance of at least 2^{i-1} in the key circle, aka the *Chord ring*.

Maintaining such an overlay structure when nodes are joining and leaving the network very frequently–referred to as a high *churn* rate–has been the focus of much academic research. However, we found only a handful of studies that investigate the extreme case of churn, namely, bootstrapping a DHT from scratch by having all nodes join at the same time. The massive join scenario is not contrived. A campus-wide or city-wide power outage will result in a large number of computers starting up at the same time. In fact, a large fraction of computers connected to the Internet today routinely reboot at the same time when they receive a periodic operating systems update. In August 2007, one such massive reboot triggered a hidden bug in the Skype client software, causing a world-wide failure of the Skype voice-over-IP network for two days [2]. A DHT-based overlay multicast system such as SplitStream [3] is another example. A large number of users may join the overlay for a live streaming of a popular event.

A few proposals address the problem of bootstrapping a DHT from scratch [4]–[8], but they all forgo the standard DHT join protocols in favor of their own distributed algorithms that build routing tables directly. They motivate their algorithms by claiming that the standard join protocols are not designed to handle the huge number of concurrent requests involved in bootstrapping a DHT from scratch. (Some even assume, incorrectly, that the join protocols expect nodes to be inserted sequentially into an overlay.) We find the claim unsubstantiated.

Another problem with the existing proposals is the assumption that they make about the initial state of the nodes before they start forming a DHT. Each node is assumed to hold a handful of pointers to a random selection of other nodes in the network. These pointers in effect provide an unstructured overlay network (termed a *knowledge graph* by [4]) as a starting point for the algorithms of the proposals. The algorithms thus take the approach of transforming a pre-existing unstructured overlay into a DHT. We question the validity of the assumption. One can argue that such a pre-existing overlay might be a reasonable model for the state of a DHT after a massive failure, where each node still has the old routing table. However, it is certainly not applicable to bootstrapping a DHT from scratch, which is the scenario that the proposals claim to address.

We take a step back and examine how the standard join protocols behave under a large number of concurrent join requests. We will use the term "JOIN protocol" to refer to the standard join protocol of a DHT, and "JOIN call" to refer to the remote procedure call to implement the protocol. When the meaning is clear from context, we will simply use "JOIN". Our system model is simple and realistic: we assume that every node knows a single global bootstrap server which is responsible for handing out a contact node when

asked. Each node asks the bootstrap server for a contact node, and then sends a message to the contact node to start a JOIN. The bootstrap server keeps a list of every node that has successfully joined the overlay, and picks one at random when it is asked to provide a contact node. We simulate bootstrapping a Chord [1] and a Kademlia [9] DHT where a large number of nodes join the network nearly simultaneously, at a rate of 1,000 nodes per second, and measure the time it takes to form a stable overlay. In contrast to the claim made by the existing proposals, our results indicate that both Chord and Kademlia handle a large number of concurrent JOINs quite well. For 10,000 nodes joining at a rate of 1,000 nodes per second, Chord and Kademlia took less than 20 and 15 seconds, respectively, to form a stable overlay. It should be noted, however, that Chord requires a slight modification to its join protocol, proposed by Baumgart *et al.* [10], to achieve the fast bootstrapping performance. We discuss this modification in detail in Section IV-C.

This paper makes two contributions to p2p research. First, we call into question the claim that JOIN protocols cannot handle a large number of nodes joining simultaneously, and provide empirical evidence disputing that claim. Second, we elucidate the reason why the Chord JOIN protocol needs a modification to deliver fast bootstrapping. We demonstrate the failure mode of the unmodified Chord JOIN protocol, which is not only interesting, but also edifying for the designers of distributed systems that may be subject to extreme conditions.

The rest of this paper is organized as follows. Section II summarizes the existing proposals and discusses the common assumption that they all seem to make. Section III describes our system model. Section IV presents our simulation results and discusses the modification to the Chord JOIN protocol. Finally, we conclude and discuss future work in Section V.

II. Related work

A. DHT construction proposals

Angluin *et al.* [4] present a series of distributed algorithms that build a linked list of all nodes sorted by their identifiers. The sorted list can then be used as a basis to construct DHTs that are based on linear overlay topologies such as Chord [1] or SkipNet [11].

For DHTs using fixed key space partitioning schemes such as P-Grid [12] or Pastry [13], Aberer *et al.* [5] propose a decentralized algorithm motivated by the requirements of peer-to-peer database applications.

Montresor, Jelasity and Babaoglu [6] describe a gossiping algorithm called T-MAN. Each node starts out with a *view*, which is just a handful of pointers to other nodes, and keeps exchanging its view with the neighboring nodes. The overlay topology defined by the views converges to that of Chord as each node refines its view to bring it closer to a Chord routing table when exchanging views with neighbors. Jelasity, Montresor and Babaoglu [7], and Voulgaris and van Steen [8], present similar gossiping algorithms that build a Pastry overlay instead.

B. Common assumptions

In [4], it is assumed that the nodes are initially organized as a weakly-connected *knowledge graph* of bounded degree d, where an edge from node u to node v indicates that u knows v's network address. A knowledge graph is therefore a model for an unstructured overlay network where each node has a handful of pointers to other nodes. In fact, all other proposals described in Section II-A make the same assumption about the initial state of the nodes, although the others do not use the term knowledge graph. Random walks on a pre-existing unstructured overlay network start the algorithm in [5]. The starting point for the gossiping algorithms in [6]–[8] is an unstructured overlay built by running another gossiping protocol (similar but separate from the one that builds DHT) called NEWSCAST [14] proposed by some of the same authors.

The NEWSCAST protocol can build a knowledge graph from scratch. The authors show that NEWSCAST will produce a sufficiently randomized knowledge graph even when every node is initialized with only a pointer to a single well-known node. (This is clearly the minimal assumption one can make.) There are a few other proposals that construct networks which can be considered knowledge graphs as well [15]–[17].

Thus, with all the proposals described in Section II-A, building a DHT from scratch–that is, making no assumption except the existence of a single well-known node–actually becomes a two-step process that requires a knowledge graph to be built first using one of the knowledge graph construction proposals. This escalates the complexity and cost of the algorithms. And at the root of all the proposals, there is the premise that JOIN cannot handle massive concurrency involved in DHT bootstrapping. Our results challenge that premise.

III. System model

We simulate a scenario where a large number of nodes join a Chord or Kademlia overlay nearly simultaneously. Each node starts the bootstrapping process by contacting the single well-known bootstrap server. The bootstrap server, which has been keeping a list of every node that has successfully joined the overlay, responds to the newly joining node by handing out a *contact node* selected at random from the list. The newly joining node then initiates a JOIN call by sending a message to the contact node that was handed out by the bootstrap server. When the new node receives a response from the contact node indicating that the JOIN was successful, the new node contacts the bootstrap server once again, so that the new node can be added to the list of successfully bootstrapped nodes. We assume that initially the bootstrap server has one node in its list of successfully bootstrapped nodes, so that it has something to hand out from the very beginning. That is, we start our simulation with an overlay already containing a single node. In practice, the initial node can be the bootstrap server itself if the bootstrap server participates in the overlay.

The exact meaning of “a large number of nodes joining nearly simultaneously” is defined by setting the *join rate*, which is the average rate at which the nodes contact the bootstrap server to start JOIN. This is a reasonable model for

the real-life situation where the incoming requests would be throttled by the maximum throughput of the bootstrap server. We use the join rate of 1,000 nodes per second, which will generate 2,000 hits per second on the bootstrap server because every joining node contacts the bootstrap server twice. This hit rate is well within the capability of today's servers. Root name servers were required to handle 1,200 requests per second more than a decade ago [18], and a measurement in 2002 reports an average rate of nearly 2,000 requests per second, with much higher loads at peak times [19].

Note that there is a significant latency from the time that a node starts a JOIN protocol until the time that the node is considered successfully bootstrapped. The latency consists of the initial round trip to the bootstrap server, the JOIN remote procedure call (which can involve multiple hops), and the final notification to the bootstrap server. This means that, in the beginning, all newly joining nodes will send their JOIN calls to the single first node, because that node is the only contact node that the bootstrap server can hand out until the list of bootstrapped nodes starts growing after the initial bootstrapping latency. This seems to create another potential bottleneck in addition to the bootstrap server. However, unlike the load on the bootstrap server which is an integral part of our system model, the load on the single first node can be controlled easily by starting out with an overlay containing many nodes rather than a single node. We start with an overlay containing a single node for simplicity.

After initiating a JOIN protocol, each node begins to send a test query to a random key every second on average. The interval between the successive test queries by a node is drawn from the normal distribution with a mean of 1 second and a standard deviation of 0.1 second, truncated to non-negative values. At another fixed interval, we collect the total number of test messages sent by all nodes during that interval and the total number of messages that have been delivered successfully during the same interval, and calculate the ratio which we call *delivery rate*. We measure the *convergence time* of an overlay, which we define as the time it takes for the delivery rate to reach 95%. We also make sure that the average hop count of the delivered messages is $O(\log N)$ because an overlay should not be considered stable if messages are delivered in $O(N)$ hops. Since we focus on the bootstrapping stage where all nodes are just joining the overlay, we assume that the nodes stay alive after they join the overlay; i.e., we assume that there is no churn.

IV. Simulation results

A. Simulation setup

We used the OverSim simulator [10], version 20080919 [20]. Conveniently, the DHT implementations included in OverSim follow a bootstrapping procedure very close to that of our system model. A singleton object called the BootstrapOracle assumes the role of our bootstrap server, except that the BootstrapOracle is a global C++ object accessed by the nodes without incurring any network delay. We modified the Chord and Kademlia implementations to include communication delays when nodes access the BootstrapOracle. The join rate is controlled by a configuration parameter, which we set to 1,000 nodes per second. OverSim also includes a test application called KBRTestApp. We modified it to produce our measurements.

Between any pair of nodes, we used a constant packet delay of 50 milliseconds, plus a random jitter of around 10%. The random jitter was drawn from the normal distribution with a mean of 0 and a standard deviation of 5 milliseconds, truncated to non-negative values. This is a conservative estimate of the average packet delay experienced in the Internet today. We do not explicitly model the processing delays in each node, which should be nearly negligible, but not zero. We consider that the processing delays are absorbed into the packet delays.

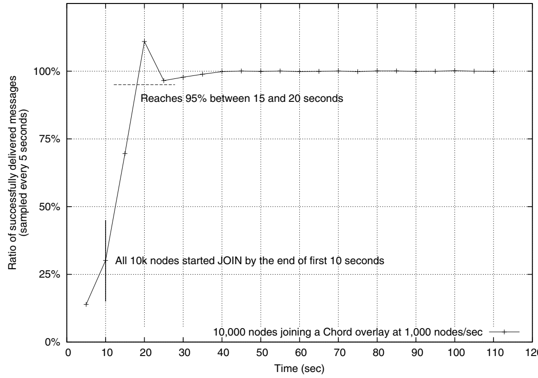
Unless noted otherwise, we used the following configuration settings for Chord. We used iterative routing, and the successor list size was set to 8. When a JOIN call fails, a node waits 5 seconds before it sends another. The *stabilize*() maintenance procedure (STABILIZE) and *fix_fingers*() maintenance procedure (FIXFINGERS) are run immediately after a node successfully joins the overlay, and afterwards they are run every 5 and 10 seconds, respectively. The STABILIZE procedure updates a node's immediate successor and the FIXFINGERS procedure updates all other routing table entries. We found the STABILIZE and FIXFINGERS intervals of 5 and 10 seconds to be optimal for bootstrapping, but in general, the intervals would be considered somewhat aggressive. A real-life implementation can use an adaptive approach, where it switches to longer intervals once the overlay reaches a stable state.

For Kademlia, we set $k = 16$, $\alpha = 1$, and $b = 1$. That is, the k-bucket size was set to 16, no parallel queries were used, and the optimization technique of considering IDs b bits at a time described in Section 4.2 of [9] was not used. OverSim's Kademlia implementation uses a sibling list, an extension described in [21]. We set the sibling list size to 8.

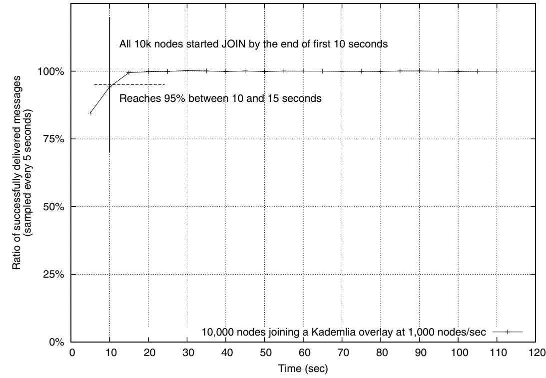
The simulations were performed on a PC equipped with Intel Core 2 Duo CPU and 4 GB RAM running Ubuntu Linux 8.04, except the simulations involving more than 50,000 nodes, which required more RAM. Those were run on a similar PC with 16 GB RAM.

B. Convergence time

Figure 1 shows our main result. It plots the delivery rates, sampled at every 5 seconds, when 10,000 nodes join a Chord and a Kademlia overlays at a rate of 1,000 nodes per second. Note that a delivery rate is simply the ratio of the number of successfully delivered messages to the number of messages sent in a given interval. Those messages that are sent and received across the interval boundaries add jitter to the delivery rates, and that is why some data points go over 100% in Figure 1. The join rate of 1,000 nodes/sec implies that all 10,000 nodes have started the JOIN calls after about 10 seconds. Figure 1(a) shows that it takes 10 more seconds for the nodes to form a stable Chord overlay that correctly routes over 95% of the queries, resulting in a fast convergence time of



(a) Delivery rate for 10,000 nodes joining a Chord overlay at the rate of 1,000 nodes per second. Convergence time is 20 seconds.



(b) Delivery rate for 10,000 nodes joining a Kademlia overlay at the rate of 1,000 nodes per second. Convergence time is 15 seconds.

Fig. 1.

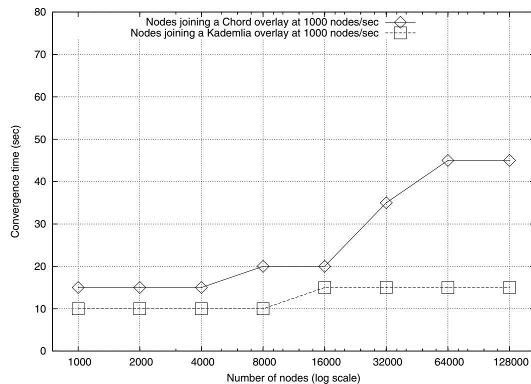


Fig. 2. Increase in convergence time as the overlay size doubles.

only 20 seconds. Figure 1(b) shows that Kademlia converges even faster, taking less than 15 seconds to reach the 95% delivery rate.

Convergence time remains fast when we increase the size of the overlay network. Figure 2 plots the increase in convergence time as we keep doubling the number of nodes. Convergence time seems to increase very slowly against the overlay size. The convergence time for Chord increases only to 45 seconds when the network size reaches 128,000 nodes. For Kademlia, the curve stays almost flat under 15 seconds.

The superior performance of Kademlia can be attributed to the symmetric nature of its routing architecture. In Kademlia, a node receives queries from the same nodes that it would contact when it wants to send a query. This enables Kademlia to use the information contained in regular queries to maintain the routing table, eliminating the need to have separate maintenance protocols. Every node sends a query every second in our simulation, resulting in a large number of messages from which the Kademlia nodes can extract useful information to adjust their routing tables. Chord does not have this symmetry, hence the need for the separate stabilization protocols, STABILIZE and FIXFINGERS.

For the large overlay sizes, a stable overlay can form before all nodes have started the JOIN calls. It takes 128 seconds for all 128,000 nodes to start the JOIN calls. In both Chord and

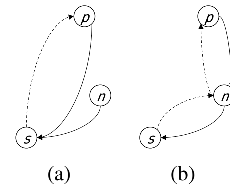


Fig. 3.

Kademlia, stable overlays form well before all nodes have joined, and the overlays remain stable as the rest of the nodes join them.

C. *Aggressive join*

Our Chord simulations in Section IV-B use a JOIN protocol called *aggressive join*, which contains a slight modification to the JOIN protocol described in the original Chord paper. Aggressive join was proposed by Baumgart *et al.* [10] and included in the OverSim's Chord implementation [20].

A node n joins a Chord ring by asking an existing node in the ring to find n's successor. In other words, n makes a JOIN call to an arbitrary contact node. Figure 3(a) shows the successor (solid line) and predecessor (dotted line) pointers among the three nodes, p, n and s, right after n has found its successor s. In the original Chord specification, the JOIN protocol stops here. Fixing up the rest of the successor and predecessor pointers to make the node n a full participant of the ring, as depicted in Figure 3(b), is delayed until the next STABILIZE cycle. The pseudocode for JOIN and STABILIZE from the Chord paper is reproduced in Figure 4.

Aggressive join, on the other hand, proceeds to complete all the pointers between the three nodes immediately after JOIN:

$$s.predecessor = n$$
$$n.predecessor = p$$
$$p.successor = n$$

When s receives a join request from n (or when the FIND_SUCCESSOR call initiated by n's JOIN call terminates at s, to be more precise), s immediately sets its predecessor to n, s then includes p in its join response message to n, enabling n to set its predecessor to p, and finally s sends a message to

```
// create a new Chord ring
n.create() {
  predecessor = nil;
  successor = n;
}
// join a Chord ring containing node n′
n.join(n′) {
  predecessor = nil;
  successor = n′.find_successor(n);
}
// called periodically:
// verifies immediate successor of n
// and tells the successor about n
n.stabilize() {
  x = successor.predecessor;
  if (x in (n,successor))
    successor = x;
  successor.notify(n);
}
// n′ thinks it might be our predecessor
n.notify(n′) {
  if (predecessor is nil or n′ in (predecessor,n))
  predecessor = n′;
}
```

Fig. 4. The original Chord stabilization protocols, reproduced from the Chord paper.

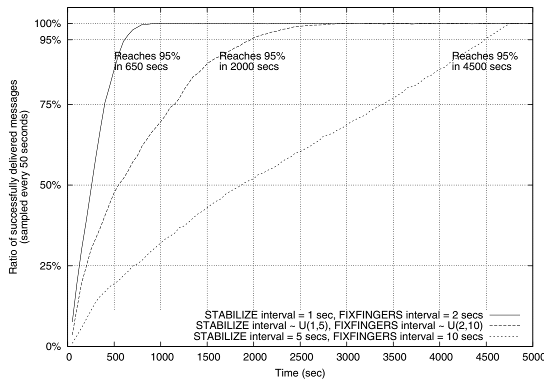


Fig. 5. Delivery rates for three different STABILIZE and FIXFINGERS intervals, with aggressive join disabled. (1,000 nodes at the rate of 1,000 nodes/sec in all three cases.)

its old predecessor p, indicating that the newly joining node n would be a better successor for p.

It turns out that aggressive join is essential for fast convergence. Figure 5 shows convergence times when we disabled aggressive join. With our standard STABILIZE and FIXFINGERS intervals of 5 and 10 seconds, it took 4,500 seconds to form a stable overlay. When we tried to compensate for the lack of aggressive join by running STABILIZE and FIXFINGERS extremely frequently–STABILIZE every second and FIXFINGERS every two seconds–we were able to bring the convergence time down to 650 seconds, but it is still orders of magnitude slower than the 15 seconds convergence time with aggressive join enabled. Randomizing the intervals did not improve the convergence time. When we set each STABILIZE interval from a uniform distribution between 1 and 5 seconds and each FIXFINGERS interval between 2 and 10 seconds, the convergence time was 2,000 seconds, indicating no significant benefit from randomization.

Figure 6 illustrates the circumstance under which bootstrapping a Chord network without aggressive join suffers from a slow convergence. It is a topology snapshot of 250 nodes trying to bootstrap a Chord network, taken at 50 seconds

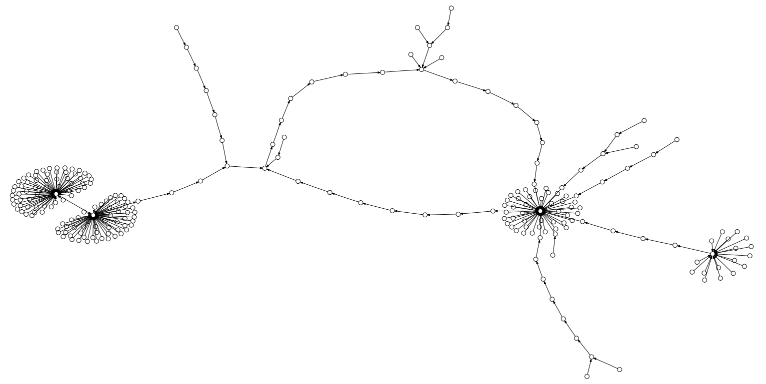


Fig. 6. Topology snapshot of 250 nodes forming a Chord network, taken at 50 seconds after they all started JOIN.

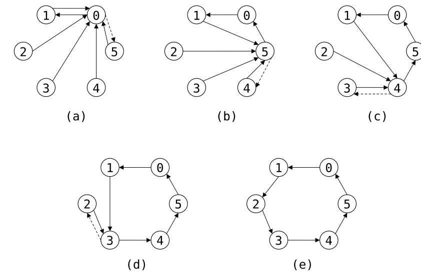


Fig. 7. An extremely slow process by which a hub turns into a ring.

after they all started JOIN nearly simultaneously (at the rate of 1,000 nodes/sec.) The arrows indicate successor pointers. The massive number of concurrent join requests, combined with the fact that the standard Chord JOIN protocol delays fixing successor and predecessor pointers, resulted in a tree-like topology containing long chains and high in-degree hubs.

Without aggressive join, when fixing successors and predecessors are left to the STABILIZE procedure, such chains and hubs converge to a ring extremely slowly. Figure 7 illustrate the process for a hub. Nodes 1 to 5 all have their successors set to node 0, and thus send NOTIFY calls to node 0, telling node 0 that they might be 0's predecessors. As node 0 receives NOTIFY calls from different nodes, it keeps updating its predecessor, eventually settling on node 5, since node 5 is the best predecessor for node 0 among nodes 1 to 5. This is shown in Figure 7(a). During the next STABILIZE cycle, nodes 1 to 4 discover that node 0 has a new predecessor, and they all set their successors to node 5 because node 5 is a better successor for them than node 0, as shown in Figure 7(b). Node 5 then receives NOTIFY calls from node 1 to 4, and sets its predecessor to node 4, which in turn causes node 1 to 3 to flock to node 4 during the next STABILIZE cycle, as shown in Figure 7(c). This process continues until the nodes forms a ring, as in Figure 7(e). From the STABILIZE algorithm in Figure 4, we see that a successor is changed only when the previous successor acquires a new predecessor, and that happens when NOTIFY is called. But in a hub topology, of all the NOTIFY calls sent by the nodes pointing to the central node, only one of them will have an effect, yielding just one chance for a successor change. Since the size of the ring can never be greater than the number of unique successors, at each iteration, the size of the ring increases only by one.

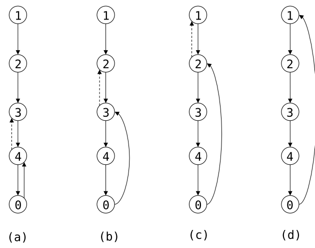


Fig. 8. An extremely slow process by which a chain turns into a ring.

Figure 8 illustrates a similar process by a long chain rather than a hub. As the sequence of predecessor changes works its way backward towards the end of the chain, the ring gets expanded one node at a time. Note that a chain will necessarily have the nodes lined up in the order of their IDs, due to the fact that a JOIN process is finding the successor node and setting the successor pointer to it.

Aggressive join sets successors and predecessors immediately to the freshly joined nodes, therefore increasing the number of unique successors as quickly as possible. This behavior is enough to break the previous pattern, as evidenced by the fast bootstrapping performance.

The fact that aggressive join effectively solves the problem of the unmodified Chord JOIN protocol provides a clue as to why Kademlia does not suffer from the same problem. Recall that Kademlia does not use separate protocols for maintaining routing tables; instead it simply uses the regular query traffic for the purpose. Kademlia's JOIN protocol is simply a series of lookup queries, from which the corresponding nodes update their routing tables as needed. In effect, Kademlia's JOIN protocol is already as "aggressive" as it can be. The lesson for a distributed systems designer seems to be that a decision to defer topology corrections to a later time should be carefully vetted to make sure there is no adverse effect, especially when the system may be subject to a high churn environment.

V. Conclusion and future work

It has been claimed that the standard DHT join protocols are not adequate for bootstrapping large DHT networks from scratch. We debunk the claim by showing that Chord and Kademlia DHT take less than 20 seconds to form a stable overlay of 10,000 nodes. Kademlia bootstraps quickly without any modification to its JOIN protocol. Chord, however, requires a slight modification to its JOIN protocol. The modified JOIN protocol, called aggressive join, fixes the successor and predecessor pointers immediately after a node has joined, as opposed to waiting until the next STABILIZE cycle. We analyze the behavior of the unmodified Chord JOIN protocol in detail and illustrate its failure mode, which reveals a danger inherent in designing a p2p system that may be subject to extreme churn.

As future work, we plan to extend our evaluation to other well-known DHTs such as Pastry. We also plan to validate our simulation results by taking measurements from a DHT deployed in a real-life environment such as the PlanetLab, albeit at a much smaller scale than our simulations.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *IEEE Transactions on Networking*, vol. 11, February 2003.

[2] What happened on August 16, 2007. [Online]. Available: http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environments," in *Proc. of SOSP '03*. New York, NY, USA: ACM, 2003, pp. 298–313.

[4] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin, "Fast construction of overlay networks," in *Proc. of SPAA '05*. New York, NY, USA: ACM Press, 2005, pp. 145–154.

[5] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt, "Indexing data-oriented overlay networks," in *Proc. of VLDB '05*. VLDB Endowment, 2005, pp. 685–696.

[6] A. Montresor, M. Jelasity, and O. Babaoglu, "Chord on demand," in *Proc. of P2P '05*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 87–94.

[7] M. Jelasity, A. Montresor, and O. Babaoglu, "The bootstrapping service," in *Proc. of ICDCSW '06*. Washington, DC, USA: IEEE Computer Society, 2006.

[8] S. Voulgaris and M. V. Steen, "An epidemic protocol for managing routing tables in very large peer-to-peer networks," in *Proc. of DSOM '03*. Springer, 2003, pp. 41–54.

[9] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *International workshop on Peer-To-Peer Systems (IPTPS)*, 2002, pp. 53–65.

[10] I. Baumgart, B. Heep, and S. Krause, "OverSim: A flexible overlay network simulation framework," in *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, May 2007, pp. 79–84.

[11] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[12] K. Aberer, "P-Grid: A self-organizing access structure for p2p information systems," in *CoopIS'01: Proceedings of the 9th International Conference on Cooperative Information Systems*. London, UK: Springer-Verlag, 2001, pp. 179–194.

[13] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[14] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "The peer sampling service: experimental evaluation of unstructured gossip-based implementations," in *Proc. of Middleware '04*. New York, NY, USA: Springer-Verlag New York, Inc., 2004, pp. 79–98.

[15] G. Pandurangan, P. Raghavan, and E. Upfal, "Building low-diameter peer-to-peer networks," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 6, pp. 995–1002, Aug. 2003.

[16] C. Law and K.-Y. Siu, "Distributed construction of random expander networks," *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 3, pp. 2133–2143, April 2003.

[17] V. Vishnumurthy and P. Francis, "On heterogeneous overlay construction and random node selection in unstructured p2p networks," *INFOCOM 2006. The 25th IEEE International Conference on Computer Communications*, pp. 1–12, April 2006.

[18] B. Manning and P. Vixie, "Operational criteria for root name servers," RFC 2010, October 1996. [Online]. Available: http://tools.ietf.org/html/rfc2010

[19] D. Wessels and M. Fomenkov, "Wow, that's a lot of packets," in *Proceedings of Passive and Active Measurement Workshop (PAM)*, April 2003.

[20] OverSim: The overlay simulation framework. [Online]. Available: http://www.oversim.org/

[21] I. Baumgart and S. Mies, "S/Kademlia: A practicable approach towards secure key-based routing," in *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–8.