Bootstrapping *Chord* in Ad Hoc Networks: Not Going Anywhere for a While.

Curt Cramer* Thomas Fuhrmann

cramer@ira.uka.de
System Architecture Group
Universität Karlsruhe (TH), Germany

Abstract

With the growing prevalence of wireless devices, infrastructure-less ad hoc networking is coming closer to reality. Research in this field has mainly been concerned with routing. However, to justify the relevance of ad hoc networks, there have to be applications.

Distributed applications require basic services such as naming. In an ad hoc network, these services have to be provided in a decentralized way. We believe that structured peer-to-peer overlays are a good basis for their design.

Prior work has been focused on the long-run performance of virtual peer-to-peer overlays over ad hoc networks. In this paper, we consider a vital functionality of any peer-to-peer network: bootstrapping.

We formally show that the self-configuration process of a spontaneously deployed Chord network has a time complexity linear in the network size. In addition to that, its centralized bootstrapping procedure causes an unfavorable traffic load imbalance.

1. Introduction

An ad hoc network is a multi-hop wireless network operating without an infrastructure. That is, all nodes have to cooperatively perform tasks such as routing or service discovery using distributed approaches.

Structured peer-to-peer overlays such as Pastry [9] and Chord [11] have originally been proposed as large-scale virtual overlays on top of the Internet. Yet, they may also be a good basis for implementing distributed applications in ad hoc networks: They operate in a decentralized way, they are self-organizing, and they offer a versatile key-based routing primitive.

The interest in applying structured peer-to-peer overlays to ad hoc networks is recently growing (cf. e.g. refs. [2, 5, 8, 12]), mostly being focused at the performance of key lookup routing. While some authors rebut the efficiency of structured overlays in ad hoc networks altogether [2, 5] and others describe working examples [8, 12], no definitive answer can be given yet.

Besides routing performance, an important, yet often neglected phase in the life cycle of an overlay network is *bootstrapping*. Since ad hoc networks are spontaneously deployed and operate without any infrastructure, the overlay protocol has to be able to quickly bootstrap itself in a decentralized manner.

In this paper, we analyze the bootstrapping protocol of the well-known overlay *Chord* with respect to its applicability to ad hoc networks (cf. Section 2). We show that the protocol has a time complexity linear in the size of the network, rendering it unsuitable. Section 3 discusses related work, and Section 4 concludes the paper.

2. Bootstrapping Chord in Ad Hoc Networks

2.1. Background

The basis of our analysis is the latest published version of Chord [11] (cf. pseudocode in Fig. 1). We included an optimization from the text in form of the *notify_predecessor()* function: "*When the predecessor of a node n changes, n notifies its old predecessor p about the new predecessor p′. This allows p to set its successor to p′ without waiting for the next stabilization round.*" [11].

Bootstrapping is not explicitly addressed in the sense that Chord does not further specify 1) which nodes create a network or 2) how joining nodes find members of the Chord network to connect with. The description in ref. [11] only covers a *join protocol* which simply assumes that there is a consistent Chord network into which new nodes join. In this setting, a new node is easily integrated since the initial call to *find_successor()* already yields the node's correct

*Funding was provided by Deutsche Forschungsgemeinschaft under grant FU448-1.

```
n.create()
  predecessor = nil;
  successor = n;
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
  successor.notify(n);
n.notify(n')
  if (predecessor is nil)
    predecessor = n';
  else if (n' ∈ (predecessor, n))
    predecessor.notify_predecessor(n')
    predecessor = n';
n.notify_predecessor(n')
  if (successor is nil or n' ∈ (n, successor))
    successor = n';
```

Figure 1. Pseudocode for Chord (cf. ref. [11])

successor. The successor is notified in the next stabilization period, resulting in a consistent network again.

A node creates a new Chord network by calling *create()*, or joins an existing Chord network by calling *join()*. After creating or joining a Chord network, a node periodically executes *stabilize()* every t_{stab} time units to update its successor and its successor's predecessor.

Finding overlay members to initially connect with (called *join points* in the following by us) requires external means (e.g., pre-configured unicast, multicast, or anycast addresses). All of these solutions are cumbersome since they either depend upon central infrastructure components or contradict the notion of self-organization by relying on a static set of addresses. A decentralized solution for finding overlay members—for example, using MAC-layer broadcasts in the ad hoc scenario—is preferable, yet non-trivial since it has to ensure connectedness of the Chord network.

In our analysis, we neglect the maintenance of additional state for the nodes' virtual links (the so-called *fingers*) pointing at nodes in exponentially spaced distances, as this information is not used for updating successor and predecessor relations.

2.2. Problem Definition and Notation

Bootstrapping can either be simultaneous or sequential. Sequential bootstrapping (cf. also ref. [6]), in which all nodes join one by one, makes the idealistic assumption that there is a global coordinator. This method has at least linear time complexity.

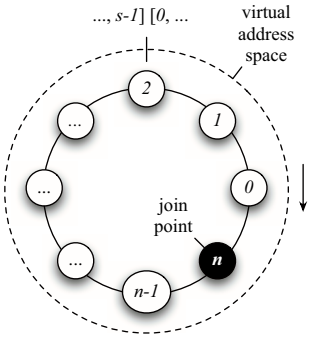


Figure 2. The employed node labeling scheme

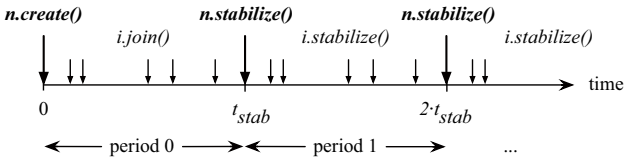


Figure 3. Timeline of bootstrapping

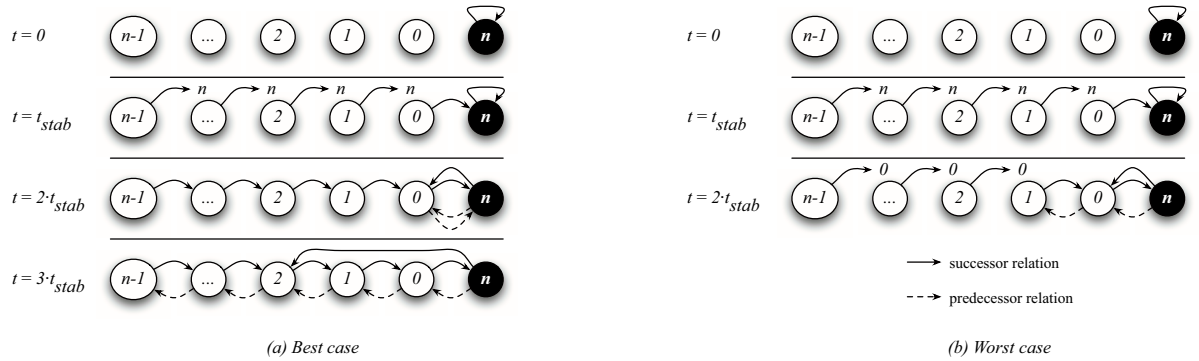
Especially in an ad hoc network, a more realistic assumption is that the network is spontaneously created. That is, the nodes are almost simultaneously powered up. This case is the subject of the analysis in the remainder of the paper.

Let the size of the network with the node set $\{0,1,\ldots,n-1,n\}$ be denoted by N. We assume that every node $i \in \{0,1,\ldots,n-1,n\}$ boots at a time $t_i \in [0,t_{stab}]$. Immediately thereafter, the nodes begin with bootstrapping Chord. Each node periodically executes *stabilize()* at times $t_i + k \cdot t_{stab}$ with $k \geq 1$; we call $[t_i + k \cdot t_{stab}, t_i + (k+1) \cdot t_{stab}]$ the k^{th} *stabilization period of* i.

For our analysis, we assume a single join point which creates the Chord network at $t = 0s$, and all other nodes join the network through this join point (cf. Sec. 2.1). An increase in the number of join points at best results in a constant-factor improvement of the time complexity, as we will discuss later.

Without loss of generality, we assign ordered integer labels to the nodes as depicted in Figure 2: the join point is denoted by n, and starting at n, the (ordered) nodes are assigned *decreasing* labels $n-1, n-2, \ldots, 0$ in *increasing* direction of the circular address space. For example, n is preceded by 0, and n's successor is $n-1$.

Figure 3 depicts the timeline of bootstrapping Chord as described above, where the join point's stabilization periods are highlighted. By definition, all nodes $i \neq n$ boot and hence join in n's period 0. Their k^{th} stabilization period begins in n's k^{th} period.



(a) Best case *(b) Worst case*

Figure 4. Evolution of the network state

2.3. Time Complexity Analysis

Bootstrapping starts with the creation of the network and ends when consistency is reached, that is when all nodes know their correct predecessor and successor.

The time complexity is determined by the *sequence* in which nodes join the system. Since all nodes call *stabilize()* in the same order in which they join the system, the best case occurs if all nodes join in-order $((n-1),(n-2),\ldots,0)$. Conversely, the worst case occurs if they join in reverse order $(0,1,\ldots,(n-1))$, and the average case occurs if they join in a random order.

In the following, we describe the node actions during bootstrapping in detail for deriving the best-case result. Figure 4 (a) illustrates the description. As will be seen below, the time complexity is dominated by the stabilization actions of the join point. Therefore, we examine the network state at discrete points in time, namely, at the beginnings or endings of 0's stabilization periods ($t = k \cdot t_{stab}$ with $k \geq 0$):

$t = 0$: Node n calls *create()*, setting its successor to n, while all other nodes have not yet booted.

$t = t_{stab}$: All nodes $i \neq n$ have booted and called *i.join(n)*. The call to *n.find_successor(i)* therein yields n in all cases, since n did not change its successor in the meantime. Thus, all nodes $i \neq n$ set their successor to n. The only node that calls *stabilize()* in period 0 is n, not changing its state however.

$t = 2 \cdot t_{stab}$: In period 1, all nodes $i \neq n$ call *stabilize()* in the order of their join times, that is first $(n-1)$, then $(n-2)$, ..., and last 0. Thus, first $(n-1)$ becomes n's predecessor, then $(n-2)$, ..., and finally 0.

As a result of the optimization in the stabilization step, if n's predecessor changes from j to $(j-1)$, n notifies j about $(j-1)$. Node j in turn sets $(j-1)$ as its successor, not calling *stabilize()* yet. Hence, after all nodes $i \neq n$ have called *stabilize()*, their successors are correct due to the order in which they joined the network.

Finally, n calls *stabilize()*. Node n's successor is n itself, and its predecessor is 0. Therefore, n selects 0 as its successor and notifies 0. Node 0 at that time has no predecessor and thus accepts n as its predecessor. At that time, all nodes except 0 and n do not have a predecessor.

$t = 3 \cdot t_{stab}$: The nodes $i \neq n$ each call *stabilize()* in the order $(n-1),(n-2),\ldots,0$. Since in the last period these node's successors already became correct, the predecessor relations of nodes $(n-2),(n-3),\ldots,0$ now are correct, too.

The moment node 0 changes its predecessor to 1, it notifies its old predecessor n of 1's existence due to the stabilization optimization. Thus, n changes its successor to 1.

Finally, n stabilizes, setting its successor to be 1's predecessor, that is 2.

$t = k \cdot t_{stab}, 3 < k < n$: The successor and predecessor relations of nodes $(n-2),(n-3),\ldots,0$ do not change as they are already correct. Node $(n-1)$'s successor is also correct, but the node has no predecessor since no other node selected it as its successor yet.

Node n calls *stabilize()* and advances its successor from $(k-2)$ to $(k-1)$.

$t = n \cdot t_{stab}$: Node n advances its successor from $(n-2)$ to $(n-1)$, and $(n-1)$ sets its predecessor to n. Now, the successor and predecessor relations of all nodes are correct.

At $t = n \cdot t_{stab} = (N-1) \cdot t_{stab}$, all nodes have correct predecessors and successors, and bootstrapping has completed.

The worst case is similar to the best case with the major difference that the stabilization optimization is ineffective. Network state here is identical to that of the best case for $t = 0$ and $t = t_{stab}$, but differs for the following times (cf. Fig. 4 (b)):

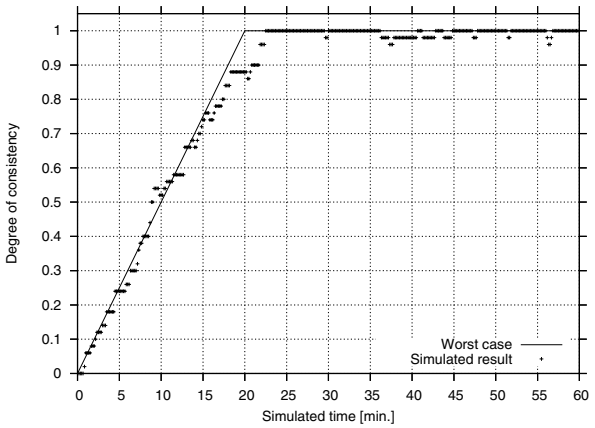


Figure 5. Exemplary run ($N = 40, t_{stab} = 30s$)

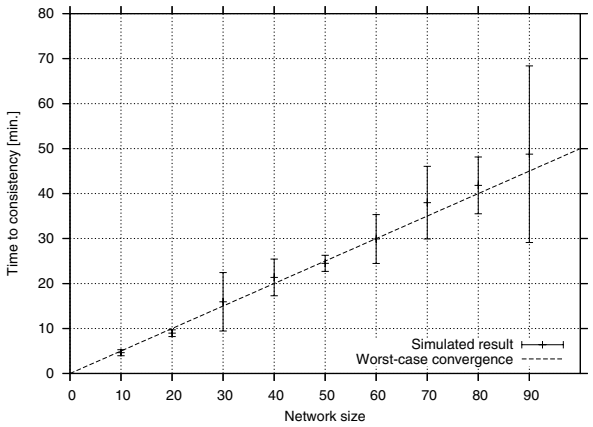


Figure 6. Convergence as a function of n

$t = 2 \cdot t_{stab}$: In period 1, all nodes $i \neq n$ call *stabilize()* in the order of their join times, that is first 0, then 1, ..., and last $(n-1)$. Thus, 0 immediately becomes n's predecessor, and all other nodes set 0 as their successor. Node 1 notifies its correct successor 0 and becomes its correct predecessor, while the notifications of the other nodes destined for 0 do not change anything.

Finally, n calls *stabilize()*. Node n's successor is n itself, and its predecessor is 0. Therefore, n selects 0 as its successor and notifies 0, which again does not change anything.

$t = k \cdot t_{stab}, 2 < k \leq n$: The successor relations of nodes $(k-2), (k-3), \ldots, 0$ do not change as they are already correct, as are the predecessor relations of nodes $n, 0, 1, \ldots, (k-3)$.

All nodes $i > (k-2)$ advance their successors from $(k-3)$ to $(k-2)$. Thus, $(k-1)$ has the correct successor $(k-2)$ and $(k-1)$ also becomes $(k-2)$'s correct predecessor.

$t = (n+1) \cdot t_{stab}$: The successor relations of nodes $(n-1), (n-2), \ldots, 0$ do not change as they are already correct, as are the predecessor relations of nodes $n, 0, 1, \ldots, (n-2)$.

Node n advances its successor from $(n-2)$ to $(n-1)$, and $(n-1)$ sets its predecessor to n. Now, the successor and predecessor relations of all nodes are correct.

At $t = (n+1) \cdot t_{stab} = N \cdot t_{stab}$, all nodes have correct predecessors and successors, and bootstrapping has completed. Thus, the upper bound on the time complexity is $O(N)$. Since both, the lower and the upper bound, are $O(N)$, there is a *tight bound of* $\Theta(N)$ on the time complexity.

2.4. Simulation Experiments

To validate our theoretical results, we have implemented Chord in the GloMoSim ad hoc network simulator [1]. The nodes maintain a list of four consecutive successors, and run the stabilization protocol every $30s$ after they have booted. All nodes boot within the first $50s$ of simulated time, except for the join point which starts at $t = 0s$.

Chord is run on top of the routing protocol AODV in its draft version 13 (cf. ref. [7]). AODV is configured with the standard parameter values, does not use optimizations, and relies on link-layer notifications to detect link failures. The MAC protocol is IEEE 802.11 DFWMAC with RTS/CTS extension, using a 2.4-GHz radio interface with a speed of 2 Mbps. Radio propagation follows the two-ray model with a nominal transmission range of $250m$.

We simulate networks of $N \in \{10, 20, \ldots, 100\}$ where the nodes are uniformly distributed on a square with an average of 13 neighbors per node. Since node mobility does not affect the asymptotic complexity of the bootstrapping protocol, the networks are static. Unless stated otherwise, for each data point, ten independent simulations are made, with each run simulating one hour of time.

The linear convergence behavior of the bootstrapping behavior can be observed on two scales. Figure 5 shows it on a microscopic scale, that is within a single simulation run. It depicts the degree of consistency of a 40-node Chord overlay as a function of time. The observed convergence behavior agrees with our analysis, its variations are only due to packet drops.

Packet drops occurring as a result of MAC-layer collisions or physical transmission errors further delay convergence which can be observed on a macroscopic scale: For each network size, we determined the average of the first point in time when all nodes have selected their correct successor.

Figure 6 depicts the result. Each data point is accompanied by a 95%-confidence interval approximated using Student's t distribution. As a result of erroneous transmissions, the larger the networks get, the less probable they are to converge into a fully consistent state, thus the increasing

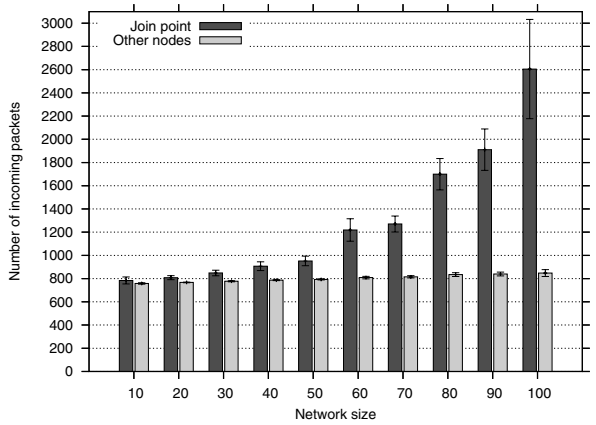


Figure 7. Traffic load

length of the confidence intervals. As a result of packet loss, none of the 100-node networks managed to converge into a consistent state within one hour of simulated time. Their expected worst-case convergence time is $100 \cdot 30s = 50min$.

Last, we measured the traffic load on the join point and the other nodes, respectively (cf. Fig. 7; "load" here is defined as the number of packets a node receives with itself being the destination). The load at the join point steeply increases with the network size, whereas only a slight increase occurs at the other nodes.

2.5. Discussion

We have shown that the time complexity of bootstrapping Chord to a consistent state is $\Theta(N)$. One might be apt to argue that the linear convergence behavior is only due to the assumption that there is a *single* join point. However, the time complexity of bootstrapping from a set of k join points is $\Theta(k + \lceil (N-k)/k \rceil) = \Theta(N)$, and thus the *asymptotic* time complexity of the bootstrapping protocol remains unchanged.

The results have a twofold implication: First, Chord's bootstrapping protocol and its linear time complexity naturally contradict the requirement of a network being able to *quickly* configure itself. Second, when designing simulation experiments with Chord, researchers have to account for the potentially very long warm-up phase, since during this time, lookups will yield incorrect results.

An advantage of having $k > 1$ join points over a single join point is the accompanying load distribution. However, in either case, the join points can be viewed as a kind of infrastructure. They have to be available before any other node can join the system, and their addresses—be it unicast, anycast, or multicast addresses—have to be pre-configured in all nodes. Again, this is an indication of the unsuitability of Chord's bootstrapping protocol for the ad hoc scenario.

3. Related Work

T-Chord [6] and the *RN* protocol [10] were designed to improve lookup performance by quickly bootstrapping the full state of a Chord network. Both approaches rely on infrastructural means for discovering join points. Our work is the first to analyze the more crucial initial phase of establishing network consistency.

We have independently proposed *ISPRP* [3] for initializing the successor pointers in an ad hoc setting. Unlike other proposals, *ISPRP* is completely decentralized, asynchronous, and provably correct [4].

4. Conclusion

Bootstrapping is an often neglected, yet important aspect of any peer-to-peer overlay protocol. In this paper, we have analyzed the bootstrapping protocol of the well-known overlay *Chord*. It has a time complexity of $\Theta(\text{network size})$, rendering it unsuitable for ad hoc networking. Additionally, since the bootstrapping protocol relies on a central entity, an unfavorable load imbalance is caused.

Future work has to address these inefficiencies to enable structured peer-to-peer overlays for ad hoc networks.

References

[1] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. GloMoSim: A Scalable Network Simulation Environment. Technical Report 990027, UCLA Computer Science Department, 1999.

[2] L. Barbosa e Oliveira, I. G. Siqueira, and A. A. F. Loureiro. Evaluation of Ad-Hoc Routing Protocols under a Peer-to-Peer Application. In *Proceedings of the IEEE WCNC 2003*, pp. 1143–1148, New Orleans, LA, USA, Mar. 2003.

[3] C. Cramer and T. Fuhrmann. ISPRP: A Message-Efficient Protocol for Initializing Structured P2P Networks. In *Proceedings of the 24th IEEE International, Performance, Computing, and Communications Conference (IPCCC)*, pp. 365–370, Phoenix, AZ, USA, Apr. 2005.

[4] C. Cramer and T. Fuhrmann. Self-Stabilizing Ring Networks on Connected Graphs. Technical Report 2005-5, Fakultät für Informatik, Unversität Karlsruhe (TH), Germany, 2005.

[5] J. Eberspächer, R. Schollmeier, S. Zöls, and G. Kunzmann. Structured P2P Networks in Mobile and Fixed Environments. In *Proceedings of the International Working Conference on Performance Modeling and Evaluation of Heterogeneous Networks (HET-NETs '04)*, Ilkley, West Yorkshire, UK, July 2004.

[6] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on Demand. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pp. 87–94, Konstanz, Germany, Aug. 2005.

[7] C. E. Perkins, E. M. Belding-Royer, and S. R. Das. Ad hoc On-Demand Distance Vector (AODV) Routing, Feb. 2003. Internet draft version 13.

[8] H. Pucha, S. M. Das, and Y. C. Hu. Ekta: An Efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004)*, English Lake District, UK, Dec. 2004.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM Middleware Conference 2001*, Heidelberg, Germany, Nov. 2001.

[10] A. Shaker and D. S. Reeves. Self-Stabilizing Structured Ring Topology P2P Systems. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing*, pp. 39–46, Konstanz, Germany, Aug. 2005.

[11] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[12] T. Zahn and J. Schiller. MADPastry: A DHT Substrate for Practicably Sized MANETs. In *5th Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, Paris, France, June 2005.