$O(\log n)$-time Overlay Network Construction from Graphs with Out-degree 1

James Aspnes[*] Yinghua Wu[*†]

July 15, 2007

Abstract

A fast self-stabilizing algorithm is described to rapidly construct a balanced overlay network from a directed graph initially with out-degree 1, a natural starting case that arises in peer-to-peer systems where each node attempts to join by contacting some single other node. This algorithm constructs a balanced search tree in time $O(W + \log n)$, where W is the key length and n is the number of nodes, improving by a factor of $\log n$ on the previous bound starting from a general graph, while retaining the properties of low contention and short messages. Our construction includes an improved version of the distributed Patricia tree structure of Angluin *et al.* [2], which we call a *double-headed radix tree.* This data structure responds gracefully to node failures and supports search, predecessor, and successor operations in $O(W)$ time with smoothly distributed load for predecessor and successor operations. Though the resulting tree data structure is highly vulnerable to disconnection due to failures, the fast predecessor and successor operations (as shown in previous work) can be used to quickly construct standard overlay networks with more redundancy.

Keywords: Overlay network, balanced search tree, pipeline, randomization, self-stabilizing, fault tolerance.

[*]Yale University Department of Computer Science, 51 Prospect St, New Haven CT 06520-8285, USA.
[†]Contact author. Email: `y.wu@yale.edu`. Telephone: +1 203 432-1239. Fax: +1 203 432-0593.

1 Introduction

Much work has been done recently on rapidly building a peer-to-peer system with a ring or line structure such as Chord [17] or skip graphs [3]. The naive approach of sequential insertion performs quite poorly for large networks: the time complexity is $\Theta(n \log^2 n)$ for Chord and $\Theta(n \log n)$ for skip graphs. So there is an incentive to find ways to exploit the parallelism of the system to build a network more quickly. Such a fast construction algorithm could allow rapid deployment of overlay networks or serve as a substitute for more complex self-repair mechanisms.

Several heuristic algorithms have been proposed that appear to converge in time $O(\log n)$ [16, 9, 10, 15]. But it is difficult to prove that this bound in fact holds, and the question of obtaining theoretical results justifying the observed practical performance remains open.

In previous work [2], we showed how to quickly sort nodes in a weakly-connected graph of bounded degree d with a provable time bound $O(W \log n)$, where n is the number of nodes and W is the length of node identifiers. This running time, which is $O(\log^2 n)$ under the reasonable assumption that $W = O(\log n)$, is much higher than both the lower bound of $\Omega(d + \log n)$ shown in the same paper and the observed behavior of practical methods. The algorithm contains three components: a randomized pairing algorithm that constructs a distributed matching from a degree-d weakly-connected graph; a distributed merging algorithm for combining balanced trees of nodes, i.e., distributed Patricia trees; and a supernode simulation that allows a tree to simulate a single supernode in the pairing algorithm. In each iteration, the output of the pairing algorithm is used to join nodes/supernodes into larger supernodes that then participate in subsequent iterations of the pairing algorithm, until a single supernode remains. The ultimate supernode is actually a distributed Patricia tree consisting of all the nodes, which supports efficient search, predecessor and successor operations. As observed in [2], having fast predecessor and successor operations can then be used to quickly construct other more robust distributed data structures, such as Chord rings or skip graphs.

In this paper, we present an even faster algorithm with expected time complexity of only $O(W + \log n)$ (which is $O(\log n)$ if node identifiers are small) and expected message complexity of $O(n \log n)$, which preserves the properties of low contention and short messages in our previous work [2]. The algorithm assumes that it starts with a directed graph initially with out-degree 1, an important special case that arises in practice. For example, a node joining an overlay network will typically connect to a single existing node, yielding a directed tree. If we relax the restriction that nodes attempt to connect to nodes already in the network, then in full generality we get a graph with out-degree 1, which may contain a cycle. Producing a sorted list quickly in this model then allows the construction of more complex data structures as in [2].

Despite the possibility of having a cycle, we use tree terminology: each node points to a parent, and we assume that each node also knows its children, which can be achieved by children's initial probes. There is no restriction on the diameter of the input structure, and unlike the output tree, children in the input graph are unordered. Our algorithm first restructures the input graph into a child-sibling graph, which can be viewed as consisting of a network of horizontal links (the sibling pointers) and vertical links (the parent and child pointers). By using a randomized pairing algorithm alternately along the horizontal and vertical links, we quickly pair off nodes and merge them to form distributed tree structures called **double-headed radix trees** (DHR trees), ultimately obtaining a single DHR tree.

What is important and different from our previous work [2] is that tree merges are pipelined, so when the roots of any pair of DHR trees start to merge, the lower layers of these trees may not be fully formed yet. This eliminates the overhead of internal communication within supernodes as in [2] and explains the reduction in cost from the previous algorithm by a factor of $O(W)$. The degree limitation on the input graph is carefully maintained to ensure that no supernode is given more than a constant number of outgoing edges so that merges will not create high contention.

Double-headed radix trees can be thought of as radix trees in which the leaves have been removed (with their keys propagated up to some ancestor) and the root has been split into a left and right root (the "double head"); these changes eliminate the need to allocate new internal nodes during merges and allow DHR trees to respond more gracefully to node failures than the distributed Patricia trees of [2] from which they are ultimately derived. From the point of view of network construction, the key property is that despite these optimizations they continue to support the fast predecessor and successor operations needed to extract (for example) a sorted ring.

The paper is organized as the following: we first introduce our model in Section 2 and then double-headed radix trees in Section 3. Section 4 gives the synchronous contraction algorithm. We show how our algorithm can be adapted to an asynchronous environment in Section 5. Finally, we conclude our work in Section 6.

1.1 Other related work

In addition to work specifically aimed at building overlay networks, there are several strains of work in the literature on problems that are similar to the fast construction problem. These include **resource discovery** [8, 13, 14, 12, 1], **leader election** [6], and **parallel sorting** [7], which will be described briefly below; for a detailed discussion of the relation between these problems and the fast construction problem see the discussion in [2].

The Resource Discovery Problem was introduced by Harchol-Balter *et al.* [8], in which all the processes in an initial weakly connected knowledge graph learn the identities of all the other processes. The problem was then relaxed to require that one process becomes the leader with the knowledge of all the other process identities, and the leader's identity is known to the whole system. In the related papers [8, 13, 14, 12, 1] addressing this problem, the final knowledge graph usually contains a star on all the vertices and messages may contain the whole list of all the processes. Cidon *et al.* [6] gave a deterministic algorithm for leader election in an initially connected knowledge graph with $O(n)$ messages and time $O(n)$, in which each non-leader must finally have an identified path to its leader, rather than a direct edge.

Goodrich *et al.* [7] introduced a parallel sorting algorithm for a parallel pointer machine that may be the closest work to ours. It builds a binary tree over nodes and then merges components according to the tree. Consecutive merging phases are pipelined to give an $O(\log n)$ total time. However, our algorithm achieves this time complexity in a far more difficult and dynamic distributed environment.

2 Model

We assume that in the initial state, n processes form a directed graph G with maximum out-degree 1, with each process running as a node in G. Such a graph naturally forms a tree, and each node u in G knows the identifier of its unique parent $u.parent$, which can be *null* if u is a root, and also the identifiers of its (set of) children $u.children$ learned from children's initial probes, which will be the empty set if u is a leaf. Using tree terminology, we will describe edges as parent and child pointers instead of incoming and outgoing edges. Furthermore, we assume the initial graph G has a maximum in-degree $d = O(\log n)$.[1]

Following [2], we assume throughout that a process u can only send a message to another process v if u knows v's identifier, i.e., if v is in $u.parent \cup u.children$. Formally, we assume that messages are of the form (s, t, σ) or (s, t, σ, u), where s is the sender, t is the receiver, σ is a message type, and u (if present) is a *single* process identifier.

We first assume that our algorithms run in a synchronous model. The computation proceeds in rounds, and all messages sent to a process s in round i are delivered simultaneously in round $i + 1$. In other words, we assume the standard synchronous message-passing model with the added restrictions that processes can only communicate with known processes and can only send $O(1)$ messages per round. This follows the synchronous model used in [2]. Though this assumption might seem to limit the applicability of our results, we show (in Section 5) that a suitably adapted synchronizer will allow our algorithm to run in an asynchronous environment without introducing too much additional cost.

3 Double-headed radix trees

We first introduce an improved version of the distributed Patricia tree structure of Angluin *et al.* [2], which we call a **double-headed radix tree** or **DHR tree**. A DHR tree with at least two nodes has two roots: a left root and a right root. (For a singleton, there is only one root, which we think of as being both the left and right root.) Its height is bounded by the length of a node identifier, W, and any node has at most two children. Each internal node stores the longest common prefix of the subtree of which it is the root and pointers to its parent and children. The two roots also store pointers to each other.

An example is shown in Figure 1(a). The node identifiers are listed in the table, and their prefixes within parentheses. To support searching, the tree must have the following property:

Property 1 *The left and right roots have incomparable prefixes, as do the two children of any node.*

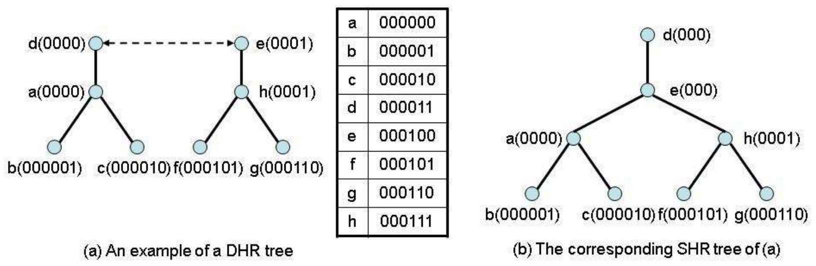
This property guarantees the correctness of searching. While searching for a particular node identifier, we start from the roots and follow the path that leads to longer prefix match. If the node with such an identifier exists somewhere in the DHR tree, its prefix has been stored in all its ancestors. Since Property 1 holds, the searching path is uniquely determined.

We also define a **single-headed radix tree** (SHR tree) to facilitate merging procedures. A DHR tree can be transformed into a SHR tree by promoting its left root to a new super-root with only one child and the children of the two roots are assigned to the former right root. The corresponding SHR tree of Fig. 1(a) is shown in Fig. 1(b). We can also think of the left and right trees of a DHR tree as SHR trees.

The procedure to merge two DHR trees to form a larger DHR tree is quite straightforward. Any merge of two DHR trees can be reduced to merging two corresponding SHR trees, since any

[1]The assumption of bounded in-degree may not always hold in practice. For example, if nodes attempt to join an existing peer-to-peer system by attaching themselves to a known current member of the system, a single well-known member might end up with a very large in-degree. However, assuming low in-degree is necessary for any algorithm that must guarantee low contention, and can (at the cost of damaging connectivity) be enforced by having each node reject excess would-be children.

DHR tree can be transformed into a SHR tree. The prefixes of the two merging roots are compared and the new root can be determined immediately, i.e. in time $O(1)$, which means that the new root can represent the combined DHR tree without waiting for the whole merge to be finished. Details of the merging procedures are given in Appendix A.2.



a	000000
b	000001
c	000010
d	000011
e	000100
f	000101
g	000110
h	000111

Figure 1: An example of a DHR tree and its corresponding SHR tree

Given multiple DHR trees, pipelined merges to combine all these trees into a single tree can be viewed as multiple merging waves that propagate down the tree, with each consecutive wave following a few steps later. A partially complete tree can participate in another merge as soon as its root is determined, so that the extra time cost for an additional merge is constant. The result is that a tree of merges of maximum depth k can be completed in $O(W + k)$ time.

For the pairing algorithm given in Section 4, the depth k is given by the number of rounds of pairing, which is $O(\log n)$ with high probability. It follows that the running time of the full construction algorithm is $O(W + \log n)$.

4 Algorithms

This section contains a family of algorithms for quickly constructing an overlay network starting with a directed graph with maximum out-degree 1 and bounded in-degree d. The structure of our algorithm is as follows:

1. In the **pre-stage**, described in Section 4.1, the initial graph is converted into a child-sibling graph we call the **contraction graph**. Each node in the contraction graph is always the left root of some DHR (initially all singleton trees).

2. In the **merging stage**, described in Section 4.2, we alternate between contracting the child-sibling graph vertically (along the parent-child axis) and horizontally (along the sibling axis). Each contraction involves merging two DHR trees and replacing their left roots in the contraction tree with the single left root of the combined tree. An additional fix-up procedure is used to prevent each merged node from ending up with more than one child (or right sibling), by "kicking" such extra neighbors into the list of siblings (or descendants) one level down on the child-parent axis (or on the sibling axis) of the graph.

The pre-stage takes $O(d)$ time to construct the child-sibling graph, where we assume $d = O(\log n)$. It uses a total of $O(n)$ messages of length $O(W)$.

The merging stage, described in Section 4.2, takes advantage of pipelined merges of DHR trees to allow each merging operation to appear to complete in $O(1)$ time from the point of view of the contraction graph. This is the time needed to merge the top layers of two DHR trees and obtain the identifier of the new roots. Though the first merge operation continues to propagate downwards and will not finish for an additional $O(W)$ time, it is nonetheless possible to start a new merge operation immediately (which will then propagate through the merged trees behind the first merge operation). This pipelining means that each additional layer of merging adds only $O(1)$ time to the $O(W)$ cost of the first merge. The total cost of the merging stage is $O(W + \log n)$ with high probability, with a total of $O(n \log n)$ messages of size $O(W)$ each. This dominates the cost of the pre-stage and gives the overall asymptotic complexity of the algorithm.

4.1 Pre-stage

In the pre-stage, the original graph G is transformed into a child-sibling graph C. For each node u in G, it sequences the nodes in $u.children$ in arbitrary order and notifies each of its children $v \in u.children$ of v's left sibling and right sibling, denoted as v.leftsibling and v.rightsibling respectively. At the end, u only keeps a child pointer to its leftmost child, denoted as $u.child$. Figure 2 gives an example of how an ordinary graph (in this case a tree) can be transformed into a child-sibling graph. The pre-stage takes $O(d)$ rounds, as each node can notify only one of its children per round. At round $(d + 1)$, all the nodes will proceed to the merging stage as described in Section 4.2. Note that this description assumes that the in-degree bound d is a fixed parameter of the algorithm known to all nodes.

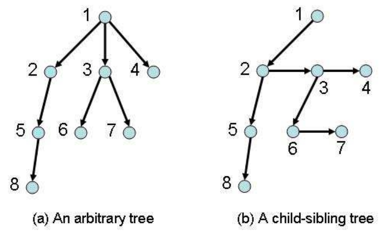


Figure 2: Pre-stage: Transform a tree into a child-sibling tree

The child-sibling graph C can be thought as a graph of many crossing directional links embedded in two dimensions. In Figure 2(b), rightward arrows indicate horizontal links generated by sibling pointers and downward arrows indicate vertical links generated by child pointers. The child-sibling graph will have a root if and only if there is some node with out-degree 0. Conversely, if there is a cycle in this graph, it can only appear as a vertical cycle along the (rootless) child-parent axis.

4.2 Merging stage

The merging stage consists of a sequence of contraction operations that alternate between contracting horizontal and vertical links generated by the pre-stage. We start with a *horizontal contraction* operation (in Section 4.2.1) along all horizontal links and then a *vertical contraction* operation (in Section 4.2.2) along all vertical links, and then repeat the procedure until only a single DHR tree is left. Some care must be taken during the contractions to ensure that all nodes maintain bounded degree; this is done by having the nodes push extra edges down towards their children (or rightward towards their right siblings). If the graph is a tree, this pushes extra edges toward the leaves. If instead the graph contains a cycle, it instead shuffles the extra edges between levels. The main purpose of pushing downwards is not so much to grab extra space (since there is none in the cycle case) as it is to prevent extra edges from piling up as the root contracts downward in the tree case.

4.2.1 Horizontal contraction

A horizontal contraction operation proceeds in two parts.

First, we do a pairing, which consists of (a) using a randomized algorithm to pair off nodes along horizontal links; (b) merging each pair of paired nodes (both being the left roots of two DHR trees) to form a new DHR tree; and (c) replacing each such pair in the contraction graph with the new left root of the resulting DHR tree.

Second, we move edges within the contraction graph. After merging some node may contain two child pointers with each coming from the previously paired nodes, so we must readjust the contraction graph C to retain the child-sibling property.

In detail, the horizontal contraction operation proceeds as follows:

```
For each node $u$, so long as $u$ is still within the contraction tree, it performs:
    *Pairing*:
    **round** $i$**:** Let *chosen* be picked uniformly from $\{u.\text{leftsibling}, u.\text{rightsibling}\}$; if *chosen*
        is *null*, choose another value; if still *null*, wait for Vertical Contraction.
    **round** $i+1$**:** Send $(u, \textit{chosen}, \text{pair})$ to *chosen*.
    **round** $i+2$**:** **Upon** receiving $(v, u, \text{pair})$ from $v$ **do**:
            If $v = \textit{chosen}$, send $(u, v, \text{accept})$ to $v$;
            otherwise, send $(u, v, \text{reject})$ to $v$.
    **round** $i+3$**:** **Upon** receiving $(v, u, \text{accept})$ from $v$ **do**:
            $u$ merges with $v$ and w.l.o.g, assume $u$ becomes the new left root and $v$ disconnects from the contraction graph.
            **Upon** receiving $(v, u, \text{reject})$ from $v$ **do**:
              Do nothing.
    *Readjusting*:
    **round** $i+4$**:** If either $u.\text{rightsibling} \neq \textit{null}$, denoted as $u_1$, or $u$ has a second child from
        previous round, denoted as $c_1$, $u$ pushes both $u_1$ and $c_1$ to a lower level:
        1. If $u.\textit{child}$ is *null*, let $u.\textit{child} = u_1$ (as in Figure 3 (a));
        2. otherwise, if $u$ has only one child, let $u.\textit{child}.\text{rightsibling} = u_1$ (as in Figure 3
           (b));
        3. otherwise, $u$ has two children $c$ and $c_1$. Assume $u$ keeps $c$ as $u.\textit{child}$.
         (a) If $u.\text{rightsibling} \neq \textit{null}$, let $c.\text{rightsibling} = u_1$ and $u_1.\text{rightsibling} = c_1$ (as in
             Figure 3 (c));
         (b) otherwise, let $c.\text{rightsibling} = c_1$ (as in Figure 3 (d)).
```

In round $i+3$, u takes over v's outgoing edges, and v is disconnected from the contraction graph and no longer participates in subsequent pairing procedures. Since all of these operations can be finished in one round, we can take advantage of pipelining to force every newly generated root to enter the next round of the pairing procedure at once, which is depicted in Section 3 as pipelined merges.

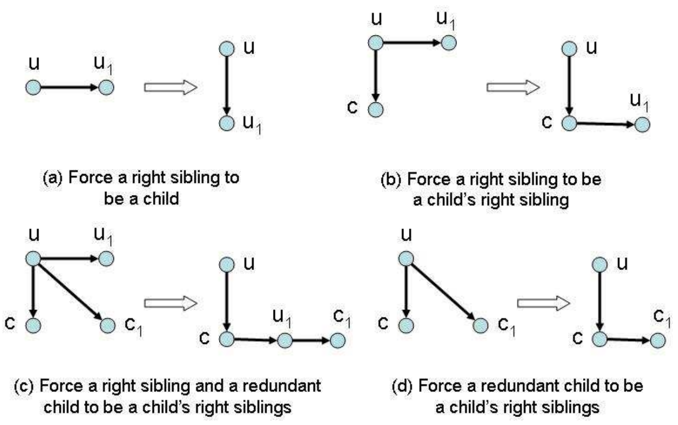

u
u1
u
u1
(a) Force a right sibling to be a child
u
u1
c
u
c
u1
(b) Force a right sibling to be a child's right sibling
u
u1
c
c1
u
c
u1
c1
(c) Force a right sibling and a redundant child to be a child's right siblings
u
c
c1
u
c
c1
(d) Force a redundant child to be a child's right siblings

Figure 3: Possible adjustments after a horizontal contraction

After a horizontal contraction operation, the graph C may not be a child-sibling graph any more. But notice that (as long as the previous graph is a child-sibling graph) the worst case is that some nodes have two children. We only need to do a local adjustment to retain the child-sibling structure as in round $i+4$. Here all the nodes simultaneously push their right siblings to a lower level (to be their children or to be their children's right siblings) so that each node still keeps a constant number of outgoing edges.

Figure 4 shows a possible merging result from the contraction graph C in Figure 2(b). Here, a double-circled node indicates that a pair of nodes have merged with each other. The subsequent edge readjustment is shown in Figure 4(b).

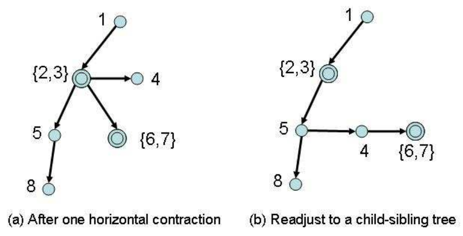

1
{2,3}
4
5
{6,7}
8
(a) After one horizontal contraction
1
{2,3}
5
4
{6,7}
8
(b) Readjust to a child-sibling tree

Figure 4: Merging stage: Horizontal contraction operation and its postadjustment

4.2.2 Vertical contraction

A vertical contraction operation is executed in exactly the same way as a horizontal contraction operation, except that it uses vertical rather than horizontal links. The main effects of this are (a) it is possible that we may have to contract a cycle rather than a path, and (b) so the pairing and pushing directions are switched.

Horizontal contractions do not contract cycles, but just "kick" edges along the cycles. Vertical contractions shrink cycles by merging adjacent nodes; when the cycle is reduced to two nodes, they can detect this and merge into a single node. Aside from this last optimization, there is no need to distinguish cycles from paths during the merging stage.

In pairing steps, for each node u, *chosen* is picked uniformly from $\{u.parent, u.child\}$, and merging is carried out along vertical links. In readjusting steps, u first pushes its child rightward to be either u.rightsibling's child or just its right sibling, and then u pushes its second right sibling (if any) to be either u.rightsibling's child or the child of u.rightsibling.*child*. These adjustments are shown in Figure 5.

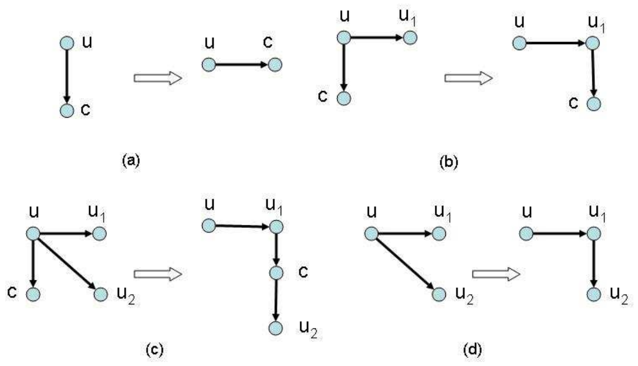


Figure 5: Possible adjustments after a vertical contraction

It is not hard to see that the algorithm does not partition the graph, so as long as it continues to merge nodes, there will be only one DHR tree left. The time complexity of the contraction algorithm is given in the theorem below. The proof appears in Appendix A.1.

Theorem 2 *The contraction algorithm finishes in* $O(W + \log n)$ *rounds on average and with high probability.*

4.3 Fault tolerance issues

In this section, we assume the underlying network is reliable and there is no message loss, but that nodes are subject to crash failures that can be detected by the node's neighbors. We give a very brief sketch of how DHR trees respond to node failures and how the contraction algorithm can be used to reconnect fragmented DHR trees.

When a node u in a DHR tree fails, the tree will separate into at most three well-formed DHR or SHR trees. The tree containing u's parent is a valid DHR tree itself even if u's ancestors may not store the longest prefixes of their subtrees any more. But this tree can still correctly perform searching and merging operations. Each of u's subtrees can become a valid DHR tree by promoting one of its root's own children to be the other root, an operation that takes $O(1)$ time. If these two trees need to connect back to the contraction tree after being separated, only their roots will try to repair these connections.

Since network partitions are irreversible in the absence of extra edges, we make a reasonable assumption that every node will keep information about several other nodes, although it chooses only one as its initial outgoing edge in the contraction graph. Thus, whenever some node fails, the root of a separated tree will try to contact nodes still in the contraction graph through its unused links. These contacts are propagated up through the DHR trees (possibly merging with other incoming contacts as they propagate up). Since two trees T_1 and T_2 may simultaneously attempt to join each other, this may lead to a cycle in the contraction graph. But since our algorithm can gracefully handle cycles, after node failures stop and the contraction graph correctly reforms, our algorithm will ultimately stabilize.

5 Extension to an asynchronous model

So far we have assumed a synchronous model, which (a) simplifies the analysis of the contraction algorithm, (b) allows extra edges to be pushed down through the contraction tree without piling up, and (c) eliminates the need for explicit coordination of changes between nodes. The price of this assumption may, however, be too high in a practical setting, and it makes it difficult to compare our algorithm with previous algorithms (such as that of [2]) that work in asynchronous environments.

To address this issue, we show how the α synchronizer of [4] can be adapted to our setting. Recall that the α synchronizer issues a sequence of increasing timestamps $0, 1, \ldots$ at each node in an asynchronous system, with the guarantee that any message sent by a process after obtaining timestamp t is received before the recipient receives timestamp $t + 1$. The mechanism for doing so is to have each node notify its neighbors in the (static) communication graph when it has finished sending all messages for round t. Once a node has collected all such notifications (and with the assumption of FIFO message delivery), it can safely proceed to round $t + 1$. It is shown in [4] that an asynchronous simulation using the α synchronizer of a synchronous algorithm with time complexity T and message complexity M has time complexity T and message complexity bounded by $M + Tm$, where m is the number of edges in the communication graph and we make the usual assumption that asynchronous messages are delivered in at most unit time.

Proper functioning of the α synchronizer depends on each node being able to predict which nodes will be notifying it in each round. If we assume in our algorithm that all nodes know their neighbors in the initial communication graph (a strong assumption), that no new nodes arrive, and that no failures occur, then we can observe by induction that every node can compute its neighbors at round t based solely on locally available information. It follows that the natural modification of the α synchronizer where a node waits at round t only for its round-t neighbors in the dynamic communication graph. The same complexity analysis for the unmodified α synchronizer applies.

We thus have:

Theorem 3 *Starting from an initial tree with consistent parent and child pointers, the tree contraction algorithm running under an α synchronizer produces a single DHR containing all nodes in $O(W + \log n)$ time using $O(n(W + \log n))$ messages with high probability in an asynchronous system.*

The assumption that all nodes know their neighbors initially is troubling. For an initial tree, we can eliminate this assumption by inserting previously-unknown children as new nodes appearing at the first round at which they are known to their parent. However, the cost of propagating round numbers through the (possibly very unbalanced) initial tree can be as much as $O(n)$, greatly increasing the time complexity of the algorithm. A better approach may be to adapt the dynamic graph synchronizer ζ of Awerbuch and Sipser [5] to our setting. As this likely to be quite involved, we leave it to future work.

6 Conclusion

We have described a fast self-stabilizing algorithm to rapidly construct a balanced overlay network from a directed graph initially with out-degree 1. This algorithm organizes all the nodes in the network into a novel balanced search tree data structure that responds gracefully to node failures and supports quick search, predecessor and successor operations. And by applying predecessor and successor operations, the nodes can quickly form into a sorted link, which turns out to be a fundamental structure for many linear overlay networks. Our analysis shows that the expected running time of the algorithm is $O(W + \log n)$, which improves by a factor $\log n$ on our previous work [2], while still preserving low contention and using messages with length proportional to the length W of node identifiers.

Our algorithm is designed for in a synchronous model, but applying a synchronizer can extend the algorithm to work in an asynchronous environment. The key difficulty here is how to incorporate late arriving nodes into the ongoing procedure; more work needs to be done in this area.

In building our data structure, we developed methods for pushing extra edges downward to maintain small degree and for pipelining sequential merges efficiently. It is an interesting open problem whether these tools may be applied to algorithms for more general classes of initial graphs.

References

[1] I. Abraham and D. Dolev. Asynchronous resource discovery. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 143–150. ACM Press, 2003.

[2] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. Fast construction of overlay networks. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures(SPAA05)*, Las Vegas, NV, USA, July 2005.

[3] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 384–393, Baltimore, MD, USA, Jan. 2003. Submitted to a special issue of *Journal of Algorithms* dedicated to select papers of SODA 2003.

[4] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[5] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks (preliminary version). In *29th Annual Symposium on Foundations of Computer Science, 24-26 October 1988, White Plains, New York, USA*, pages 206–220, 1988.

[6] I. Cidon, I. Gopal, and S. Kutten. New models and algorithms for future networks. *IEEE Transactions on Information Theory*, 41(3):769–780, May 1995.

[7] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, 43(2):331–361, 1996.

[8] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 229–237. ACM Press, 1999.

[9] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[10] M. Jelasity, A. Montresor, and O. Babaoglu. The bootstrapping service. In *Proceedings of the 26th International Conference on Distributed Computing Systems Workshops (ICDCS WORKSHOPS): International Workshop on Dynamic Distributed Systems (IWDDS)*, Lisboa, Portugal, 2006. IEEE Computer Society.

[11] R. M. Karp. Probabilistic recurrence relations. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 190–197, New York, NY, USA, 1991. ACM Press.

[12] S. Kutten and D. Peleg. Asynchronous resource discovery in peer to peer networks. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 224–231, October 13–16, 2002.

[13] S. Kutten, D. Peleg, and U. Vishkin. Deterministic resource discovery in distributed networks. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 77–83. ACM Press, 2001.

[14] C. Law and K.-Y. Siu. An $O(\log n)$ randomized resource discovery algorithm. In *Brief Announcements of the 14th International Symposium on Distributed Computing, Technical University of Madrid, Technical Report FIM/110.1/DLSIIS/2000*, pages 5–8, Oct. 2000.

[15] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005)*, pages 87–94, Konstanz, Germany, Aug. 2005. IEEE Computer Society.

[16] M. Onus, A. Richa, and C. Scheideler. Linearization: Locally self-stabilizing sorting in graphs. To appear, Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX), 2007.

[17] 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 Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

A Appendix

A.1 Analysis of time complexity

Since the contraction graph C has at most n edges, there is a one-to-one mapping between each edge and some merge, except for the last merge that collapses the possible vertical cycle, which may consume two antiparallel edges. Thus after every successful merge, the total number of edges is reduced by one. In the following analysis, we will show that after one horizontal and vertical contraction operations, a constant fraction of edges will be eliminated on average.

Assume before the i^{th} contraction operations there are A_{i-1} horizontal edges and B_{i-1} vertical edges(e.g. $A_0 + B_0 = n - 1$ or n).

Lemma 4 *After the i^{th} horizontal and vertical contraction operations,* $\mathrm{E}[A_i+B_i] \leq \frac{7}{8}(A_{i-1}+B_{i-1})$.

Proof: Let H be the number of horizontal edges and V the number of vertical edges removed by the i^{th} horizontal and vertical contraction operations. Since the probability of choosing any edge is at least 1/4, we have $\mathrm{E}[H] \geq \frac{1}{4}A_{i-1}$.

After the horizontal contraction operation, we readjust the contraction graph to retain the child-sibling property. This adjustment may change some vertical edges to horizontal edges. But the number of such edges is no more than $B_{i-1}/2$, because such an adjustment happens only if a (consolidated) node has two children. Removing these edges leaves $B_{i-1}/2$ vertical edges to participate in the i-th vertical contraction, and so $\mathrm{E}[V] \geq \frac{1}{4}(B_{i-1}/2) = \frac{1}{8}B_{i-1}$. So we have

$$\mathrm{E}[A_i + B_i] = (A_{i-1} + B_{i-1}) - \mathrm{E}[H] - \mathrm{E}[V] \leq \frac{7}{8}(A_{i-1} + B_{i-1})$$

∎

We use a classic theorem regarding probabilistic recurrence relations, due to Karp [11]. If a process can be described as $T(x) = a(x) + T(h(x))$, where x is a nonnegative real variable, $a(x)$ is a nonnegative real-valued function of x and $h(x)$ is a random variable ranging over $[0, x]$ and having expectation less than or equal to $m(x)$, where m is a nonnegative real-valued function, then the following theorem holds

Theorem 5 ([11]) *Suppose there is a constant d such that $a(x) = 0$, $x < d$ and $a(x) = 1$, $x \geq d$. Let $c_t = \min\{x|u(x) \geq t\}$. Then, for every positive real x and every positive integer w,* $\Pr[T(x) \geq u(x) + w] \leq \left(\frac{m(x)}{x}\right)^{w-1} \frac{m(x)}{c_{u(x)}}$,

in which $u(x)$ denotes the least nonnegative solution of $\tau(x) = a(x) + \tau(m(x))$, a deterministic counterpart of the above process. $u(x)$ is uniquely given by the formula $u(x) = \sum_{i=0}^{\infty} a(m^{[i]}(x))$, where $m^{[0]}(x) = x$ and $m^{[i]}(x) = m(m^{[i-1]}(x))$ for $i = 1, 2, \ldots$.

Our contraction algorithm can be illustrated in the form of Theorem 5 as in [11]: $m(x) = \frac{7}{8}x$, $a(x) = 0, x < 1$, $a(x) = 1, x \geq 1$ for the time cost of each horizontal and vertical contraction operation is $O(1)$. Then $u(x) = 0$ for $x < 1$, $u(x) = \lfloor \log_{8/7}(x) \rfloor + 1$ for $x \geq 1$ and $c_t = \left(\frac{8}{7}\right)^{t-1}$. Then Theorem 5 gives the following result when we substitute x with $n-1, n \geq 3$ and let $w = \lceil c \log_{8/7}(n-1) \rceil$ for any fixed constant c:

$$\Pr[T(n-1) \geq \lfloor \log_{8/7}(n-1) \rfloor + w + 1] \leq \left(\frac{7}{8}\right)^{w-1} \frac{n-1}{\left(\frac{8}{7}\right)^{\lfloor \log_{8/7}(n-1) \rfloor + 1}} \leq \frac{1}{(n-1)^{c-1}}.$$

Therefore, with high probability we will only need $O(\log n)$ rounds to reduce all the edges. If this bound fails, we are left with a contraction graph which again collapses to a single node in an additional $O(\log n)$ rounds w.h.p. It follows that the expected number of rounds to contract all the edges is also $O(\log n)$.

To this must be added the (deterministic) time cost $O(W)$ for the final DHR tree to finish all the pipelined merges. We thus obtain the total time cost for our algorithms $O(W + \log n)$, and thus prove Theorem 2.

A.2 Merging two DHR trees

Here we describe the merge procedure for a pair of DHR trees.

A.2.1 Notation in DHR trees

We first denote a DHR tree as T_D, and similarly a SHR tree as T_S. When the distinction is not necessary, both a DHR tree and a SHR tree can be denoted as T and called a radix tree. We adopt the following definitions:

– For u and v, $T_D(u, v)$ indicates a DHR tree with left root u and right root v; $T_S(u)$ indicates a SHR tree rooted at u. When there is no possibility of confusion, $T(u)$ means a radix subtree rooted at u.

– For a node u, its identifier is denoted as $u.id$ and its prefix as $u.prefix$. For a radix tree T, its prefix means the longest common prefix of all the node identifiers in T, denoted as $T.prefix$.

– For two prefixes x and y, we use $x = y$ to indicate that x is equal to y, $x \neq y$ that x is incomparable to y, and $x \subset y$ that y is a short prefix of x respectively.

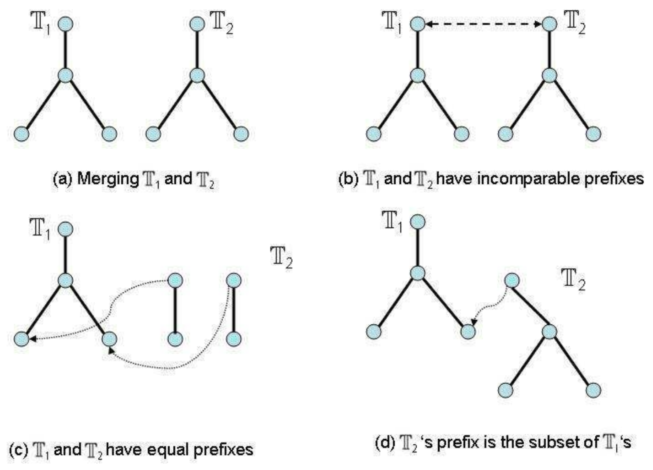
A.2.2 Merging algorithm

Any merge of two DHR trees can be reduced to merging two corresponding SHR trees, since any DHR tree can be transformed into a SHR tree. Assume T_1 is merging with T_2 as in Fig. 6(a) and let b be the first bit position at which $T_1.prefix$ differs from $T_2.prefix$. There are only three cases categorized by the relationship of T_1 and T_2's prefixes.

1. If $T_1.prefix \neq T_2.prefix$, we combine these two SHR trees into one DHR tree. The tree with the b-th bit equal to 0 will become the left tree of the merged DHR tree while the other becomes the right tree, as in Fig. 6(b).

2. If $T_1.prefix = T_2.prefix$, we can break the tie by choosing the tree with a head of a smaller node identifer, e.g. T_1, and another tree T_2 is decomposed into two smaller SHR trees, i.e. its left and right trees, which slide down along T_1 and carry on further merges with subtrees of T_1, as in Fig. 6.

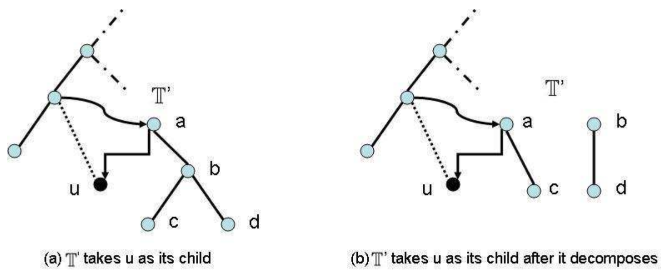
3. If $T_1.prefix \subset T_2.prefix$ or $T_2.prefix \subset T_1.prefix$, e.g. as in Fig. 6 (d), the tree with shorter prefix keeps its root while the other slides down for further merges.



$\mathbb{T}_1$
$\mathbb{T}_2$
(a) Merging $\mathbb{T}_1$ and $\mathbb{T}_2$
(b) $\mathbb{T}_1$ and $\mathbb{T}_2$ have incomparable prefixes
(c) $\mathbb{T}_1$ and $\mathbb{T}_2$ have equal prefixes
(d) $\mathbb{T}_2$'s prefix is the subset of $\mathbb{T}_1$'s

Figure 6: Merging two DHR trees T_1 and T_2

When a SHR tree is sliding down another tree, it either further decomposes into smaller SHR trees or settles down somewhere deep in the tree. For example, as in Fig. 7, T' is sliding down along another tree. If it encounters a subtree with the same prefix, it just decomposes into two smaller SHR trees, each of which slides down along the proper branch. If a subtree with incomparable prefix, assuming $T(u)$ as in Fig. 7(a), is encountered, T' will take u as one of its child and substitute $T(u)$ in the original tree. If $T' \subset T(u).prefix$, then T' can just keep sliding down along the proper branch. But if $T(u).prefix \subset T'$, T' needs to first decompose into two smaller SHR trees, one of which replaces u's position and takes u as its child while the other slides down, as in Fig. 7(b).



$\mathbb{T}'$
a
b
c
d
u
(a) $\mathbb{T}'$ takes u as its child
(b) $\mathbb{T}'$ takes u as its child after it decomposes

Figure 7: T' is sliding down along another tree