Fast Construction of Overlay Networks

Dana Angluin
Dept. Comp. Sci.
Yale University
New Haven, CT 06520
dana.angluin@yale.edu

James Aspnes *
Dept. Comp. Sci.
Yale University
New Haven, CT 06520
aspnes@cs.yale.edu

Jiang Chen
Dept. Comp. Sci.
Yale University
New Haven, CT 06520
jiang.chen@yale.edu

Yinghua Wu †
Dept. Comp. Sci.
Yale University
New Haven, CT 06520
y.wu@yale.edu

Yitong Yin ‡
Dept. Comp. Sci.
Yale University
New Haven, CT 06520
yitong.yin@yale.edu

ABSTRACT

An asynchronous algorithm is described for rapidly constructing an overlay network in a peer-to-peer system where all nodes can in principle communicate with each other directly through an underlying network, but each participating node initially has pointers to only a handful of other participants. The output of the mechanism is a linked list of all participants sorted by their identifiers, which can be used as a foundation for building various linear overlay networks such as Chord or skip graphs. Assuming the initial pointer graph is weakly-connected with maximum degree d and the length of a node identifier is W, the mechanism constructs a binary search tree of nodes of depth $O(W)$ in expected $O(W \log n)$ time using expected $O((d+W)n \log n)$ messages of size $O(W)$ each. Furthermore, the algorithm has low *contention*: at any time there are only $O(d)$ undelivered messages for any given recipient. A lower bound of $\Omega(d + \log n)$ is given for the running time of any procedure in a related synchronous model that yields a sorted list from a degree-d weakly-connected graph of n nodes. We conjecture that this lower bound is tight and could be attained by further improvements to our algorithms.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms Theory

Keywords

Overlay networks, asynchronous, merging, Patricia trees

*Supported in part by NSF grants CCR-0098078, CNS-0305258, and CNS-0435201.

†Supported by NSF grants CCR-0098078 and CNS-0305258.

‡Supported by NSF grants CCR-0098078 and CNS-0305258.

SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.

1. INTRODUCTION

Consider the problem of rapidly building a peer-to-peer system with a ring or line structure such as Chord [19] or skip graphs [3]. The default assumption in both systems appears to be that nodes will be inserted sequentially, giving a construction time of $O(n \log^2 n)$ for Chord and $O(n \log n)$ for skip graphs. But how quickly can we build such a system if we do so in parallel, assuming that initially each node only knows about a few other nodes in the system? At minimum, we need to be able to construct the bottom ring of the system, which consists of all of the nodes sorted by their identifiers (randomly chosen in Chord, based on keys in skip graphs). Constructing such a system thus depends on being able to sort nodes quickly; having done so, rebuilding the rest of the system takes little additional time. If building a peer-to-peer system from scratch can be done quickly enough, the payoff is high: we can instantly deploy peer-to-peer networks as needed as a tool in more complex distributed algorithms, and we can drop the cumbersome repair mechanisms found in many practical structured peer-to-peer systems in favor of a policy of periodic mass destruction and renewal.

In our model, we assume that any node can communicate with any other node once it knows the other's address, and that initially, the nodes are organized into a weakly-connected **knowledge graph** of bounded degree d, where a (directed) edge from u to v means that u knows v's address. Our algorithm proceeds by reorganizing this weakly-connected graph as a low-depth binary search tree where each node supplies both a leaf and at most one internal node; the sorted list can then be read off of the leaves. Our algorithm has low **contention**: each node receives at most $O(d)$ messages in a single round (in the synchronous version of the algorithm) or has at most $O(d)$ pending undelivered messages at any time (in the asynchronous version). It also uses only short messages, of size proportional to the node

identities, and requires storing only $O(d)$ identities per node.

Our algorithm is constructed from three components:

1. A randomized **pairing** algorithm that, starting from a degree-d weakly-connected graph, pairs off a constant fraction of the nodes on average in $O(1)$ time.[1] This algorithm is described in Section 3.1. The problem solved is similar to the problem of constructing a distributed matching [11, 20], except that there is no requirement that paired nodes be joined by an edge in the original knowledge graph. A complication is that since the original knowledge graph is directed, at any time a node may learn of the existence of new neighbors, and care needs to be taken to prevent deadlocks.

   The output of the pairing algorithm is used to join individual nodes into simulated **supernodes** that then participate in subsequent iterations of the pairing algorithm. These supernodes are in turn joined into larger supernodes, until only a single supernode (consisting of all the nodes in the system) remains, after an expected $O(\log n)$ iterations. For this construction to work, we need two additional mechanisms.

2. A **distributed merging** algorithm for combining balanced trees of nodes. In a synchronous model, this algorithm can be very simple: because all the trees constructed after k rounds will have depth $O(k)$, it is enough to recruit a new root node to join two trees together into a tree of depth $O(k+1)$. In an asynchronous model, an adversarial scheduler can arrange for particularly fast nodes to merge early followed by a slower succession of singleton nodes, leading to a tree of depth $\Theta(n)$ using the simple algorithm. Instead, we describe an algorithm obtained by parallelizing the sequential Patricia tree merge procedure of Okasaki and Gill [18]; this algorithm, described in Section 3.2, assigns a single leaf node and at most one internal node of the Patricia tree to each physical node in the system, and merges two Patricia trees of depth W in time $O(W)$, where W is the length of a node identity. Though we do not use this fact in our main result, our merging algorithm can be pipelined: a depth-k tree of up to 2^k merge operations can be carried out in parallel in $O(k+W)$ time with $O(1)$ contention.

   Note that because Patricia trees are a form of binary *search* tree, a consequence of using Patricia trees to represent supernodes is that the leaves are automatically sorted. We use this fact to generate the sorted ring of physical nodes that our main result promises, but the ability to rapidly generate a binary search tree with low contention starting from a weakly-connected knowledge graph may also be useful for other applications.

3. A **supernode simulation** that allows trees of ordinary physical nodes to simulate a single supernode in the pairing algorithm (Section 3.3). Though the essential idea of this simulation is simple—have the root of each tree simulate the supernode—some care needs to be taken to keep the root from being swamped with information. The actual simulation algorithm uses the leaves of the tree to communicate with other supernodes, with internal nodes aggregating the incoming data for delivery to the root. This increases the tree's effective bandwidth proportional to the size of the tree while keeping the contention down to the minimum $O(d)$ necessitated by the degree of the original knowledge graph. The cost of this strategy is that incoming messages are effectively delayed by the depth W of the Patricia tree, adding a factor of W slowdown to the algorithm. Using the entire tree also means that the simulated supernode must wait for a merge to complete before starting a new iteration of the pairing algorithm, which prevents its from taking advantage of pipelined merges. We discuss some ideas for how such bottlenecks might be avoided in Section 5.

[1] In a synchronous model, time is measured in rounds. In an asynchronous model, time is measured by assuming a constant maximum message delay, and assigning all events the latest possible time consistent with this assumption. Details are given in Section 2.

We also consider how to use the tree to build a ring (Section 3.4) and the effects of churn (Section 3.5).

In an asynchronous model, the total time for the expected $O(\log n)$ iterations of pairing multiplied by the $O(W)$ merging and communications costs of each iteration is $O(W \log n)$. In a synchronous model, this can be improved by using the simple pairing algorithm described above to construct a balanced tree of depth $O(\log n)$ in $O((d+\log n)\log n)$ time (the d factor vanishes if nodes are allowed to send more than one outgoing message per time unit), and the nodes can then be sorted using pipelined Patricia tree merges in an additional $O(W+\log n)$ time for a total of $O(W+(d+\log n)\log n)$ time. All of these algorithms have contention at most $O(d)$, use messages of size $O(W)$, and store $O(dW)$ bits of state per node.

These limits compare favorably to previously known algorithms in this model for **resource discovery** [2,10,13,14,16] or **leader election** [4], which also construct trees over nodes that initially form a weakly-connected graph. In these algorithms, a single participant may receive messages from a very large number of other participants in a short amount of time; messages are often impractically long, conveying in the worst case the identities of every node in the system; and the resulting trees have very high degree, which not only leads to high contention in any algorithm that uses them but limits performance if nodes are also limited in how many messages they can send per time unit. We discuss these results further in Section 1.1.

Though our algorithm is reasonably efficient, we do not believe that it is optimal. The best lower bound we know how to prove for constructing a sorted list of nodes starting from a weakly-connected graph with maximum degree d in the synchronous model is $\Omega(d+\log n)$; here the d term comes from the assumption that a node can only send to one recipient per round, and the $\log n$ term comes from the time to communicate from one end of a length-n line to the other using pointer jumping (see Section 4 for details). Our suspicion is that this lower bound is in close to the true upper bound, and that an algorithm that interleaved pairing and merging operations more tightly could achieve something very close to it with high probability.

Due to space limitations, most proofs in this extended abstract are omitted or only briefly sketched.

1.1 Related Work

The problem of organizing a weakly-connected set of nodes into a useful data structure combines aspects of both sorting and resource discovery. We discuss the extensive prior literature on resource discovery first, and then consider some other algorithms that solve problems closer to ours.

1.1.1 The Resource Discovery Problem

Harchol-Balter, Leighton, and Lewin [10] introduced the Resource Discovery Problem to model the situation in which all the processes in an initial weakly connected knowledge graph learn the identities of all the other processes, preliminary to cooperating in a distributed computation [10]. In terms of the knowledge graph, the goal is to construct the complete graph from an arbitrary weakly connected graph. Subsequently, the problem definition was relaxed to require that one process become the leader and learn the identities of all the other processes, which each learn the identity of the leader [14]. This implies that the final knowledge graph contains a star (with bi-directional edges) on all the vertices.

The problem has been considered in both synchronous and asynchronous models of computation. In the synchronous model, in each round each node may contact one or more of its neighbors in the current knowledge graph and exchange some subset of its neighbor list. Harchol-Balter, Leighton and Lewin give a simple randomized algorithm to solve the complete-graph version of the problem with high probability in $O(\log^2 n)$ rounds, $O(n \log^2 n)$ messages, and $O(n^2 \log^3 n)$ bit complexity. Law and Siu give another randomized algorithm that improves each of these bounds by a factor of $\log n$ [16].

Kutten, Peleg, and Vishkin give a deterministic algorithm to solve the star-graph version of the problem in $O(\log n)$ rounds, $O(n \log n)$ messages, and $O(|E_0| \log^2 n)$ bit complexity, where E_0 is the set of edges of the initial knowledge graph [14]. A single additional round in which the leader sends the whole list to all processes achieves a complete graph, and adds $O(n)$ to message complexity and $O(n^2 \log n)$ to bit complexity.

In the asynchronous model of computation, all messages sent will eventually arrive after a finite but unbounded time, and messages from u to v are delivered to v in the order in which u sent them. Kutten and Peleg give a deterministic algorithm to solve the star-graph version of the problem in $O(n \log n)$ messages and $O(|E_0| \log^2 n)$ bit complexity [13]. Abraham and Dolev also consider asynchronous computation, and generalize the problem to finding a leader in each weakly connected component of the initial knowledge graph [2]. They show that knowledge of n, the size of a node's component, affects the achievable bounds. In particular, when n is unknown, they give a lower bound of $\Omega(n \log n)$ on message complexity, and a deterministic algorithm with message complexity $O(n \log n)$ and bit complexity $O(|E_0| \log n + n \log^2 n)$. When n is known, they give a deterministic algorithm with $O(n\alpha(n))$ message complexity and $O(|E_0| \log n + n \log^2 n)$ bit complexity, where $\alpha(n)$ is the inverse Ackermann function. They also define the Ad-hoc Resource Discovery Problem, in which each non-leader must only have an identified path to its leader, rather than a direct edge. For this problem, they show a lower bound on message complexity of $\Omega(n\alpha(n))$, and an algorithm that achieves message complexity $O(n\alpha(n))$ and bit complexity $O(|E_0| \log n + n \log^2 n)$. This algorithm also deals efficiently with dynamic additions of nodes and links to the network.

It is worth mentioning that our trees solve the Ad-hoc Resource Discovery Problem, which means the Abraham and Dolev $\Omega(n\alpha(n))$ lower bound for messages applies.

1.1.2 Leader Election

Cidon, Gopal and Kutten [4] introduce a detailed and technologically motivated model in which processes in a network may use newly learned routes to send messages at cost $O(1)$, although the physical route may consist of several links, analogous to the addition of edges in a knowledge graph. Assuming n nodes in an initial knowledge graph that is connected and undirected, they give a deterministic algorithm for leader election in $O(n)$ messages and time $O(n)$. Upon termination, the corresponding knowledge graph contains an edge from the leader to every process in the network, and a path of length at most $O(\log n)$ from each non-leader to the leader, solving the Ad-hoc Resource Discovery Problem as defined by Abraham and Dolev [2]. The time bound reflects the fact that there may be long chains of merges that happen sequentially.

1.1.3 Parallel Sorting

Algorithms for sorting on parallel machines have been studied extensively. The closest such algorithm to our work is that of Goodrich and Kosaraju [9] for a **parallel pointer machine**, which also proceeds by building a binary tree over nodes and then merging components according to the tree. Their algorithm makes use of a clever parallel linked-list merge operation that allows consecutive merging phases to be pipelined, giving an $O(\log n)$ total time. We believe that essentially the same merging algorithm could be adapted to our model, although improvements in other parts of our algorithm would be necessary for this change to improve our overall running time.

1.1.4 Tree structures in previous work

A distributed trie is used as a search structure for P2P systems in several previous papers. Karl Aberer [1] designed a scalable access structure named P-Grid based on a multilevel trie with each node representing one specific interval of the key space. And at each level every peer is associated with exactly one tree node and maintains references to cover the other side of the search tree so that a search can be started at any peer. The convergence of P-Grid construction strongly depends on the assumption that peers meet frequently and randomly pairwise which is not so achievable in application. Although the paper provides some simulation results, it doesn't give further proof. Michael J. Freedman and Radek Vingralek [7] presented a similar approach while taking advantage of information piggybacking during lookups to achieve dynamic partitioning and lazy updates. However, the performance of the algorithm depends on the lookup pattern and the paper also lacks proof. Others have proposed using B-tree variants to index small numbers of nodes (hundreds) in distributed databases [12, 21].

2. MODEL

We assume that in the initial state each process knows the identifiers of some number of other processes. This information forms a **knowledge graph**, a directed graph with one vertex per process and an edge from u to v whenever u knows about v. The knowledge graph may evolve over time as processes tell each other about other processes; if u knows about both v and w, it may send a message to v containing w's identifier, and when v receives this message the edge vw is added to the graph. We assume throughout that a process u can only send a message to another process v if uv is present in the graph when the message is sent. We assume that the initial knowledge graph G_0 is weakly connected and has maximum degree d, where the **degree** $d(u)$ of a vertex u is defined to be the sum of its **indegree** $d^-(u)$ and its outdegree $d^+(u)$, which are the number of incoming and outgoing edges for u, respectively.

Processes communicate by passing messages along edges of the knowledge graph. 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 vector of $O(1)$ process identifiers. The state of each process consists of a state variable q together with a set of successors S. Upon receiving a set M of one or more messages, a process s adds all process identifiers that appear in M to S, and then executes a probabilistic transition function δ mapping (q, S, M) to (q', m), where q' is a new state and m is either $\perp$ (indicating no message sent) or a message (s, t, σ, u) where t is in S and u is in $S \cup \{\perp\}$. When and how this message is delivered depends on whether we are in a synchronous or asynchronous model; we discuss both variants below.

In the 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 one message per round. This yields a model essentially identical to the one used in the resource discovery literature, except that we have added a limitation on the number of identifiers that can be sent in a single message. We are also interested in minimizing **contention**, which we take to be the maximum number of messages received by any single process in any single round of the computation.

In the asynchronous model, messages arrive one at a time after a delay that may vary from message to message and that is controlled by an adversary scheduler. It is assumed, however, that no message is delayed by more than one time unit and that messages between any two nodes are delivered in FIFO order. Processing time is treated as zero.

Defining contention for an asynchronous model can be tricky, as the adversary could choose to deliver many messages in a short period of time; we adopt a simple measure of contention equal to the maximum number of distinct messages with the same recipient that are in transit at any point in time.[2] Assuming that each process sends at most one message to each neighbor in the knowledge graph before receiving a reply, the contention is trivially bounded by the degree of the knowledge graph in both the synchronous and asynchronous models.

3. ALGORITHMS

This section contains our main results, a family of algorithms for quickly constructing tree-structured overlay networks starting with a weakly-connected communication graph. We begin by describing (in Section 3.1) a randomized distributed algorithm for pairing nodes; this produces a matching on the set of nodes that includes a constant fraction of the nodes on average, in time $O(d)$, with $O(d)$ contention and $O(n)$ messages, each of size at most $O(W)$, where W is the maximum identifier size. Paired nodes are then joined together into simulated combined nodes that are internally organized as balanced trees (see Section 3.3). The participants in each combined node are carefully deployed so that the pairing and joining algorithms in later rounds still produce only $O(d)$ contention; however, communication within each subtree adds an factor to the cost of communication in the pairing algorithm that depends on the depth of the tree.

[2] An alternative assumption is that each process is only guaranteed to accept at least one message per time unit, with other messages waiting in a delivery queue. This yields similar time bounds, except that the running time must be multiplied by the contention to account for queuing delays.

In Section 3.2, we show that Patricia trees [17], using a parallelized version of the Patricia tree merge procedure of Okasaki and Gill [18], are a good choice for the balanced tree data structure. Using Patricia trees, we obtain a sorted final data structure in time $O(W \log n)$ (or $O((d+W) \log n)$ in the lower-bandwidth synchronous model), with $O(d)$ contention and $O((d + W)n \log n)$ messages.

Finally, we briefly discuss constructing a ring (Section 3.4) and the effects of node departures and arrivals (Section 3.5).

3.1 Pairing

The pairing problem has some similarities to the problem of finding a matching, but because we are not restricted in which nodes we pair off—except for the limits imposed by communication along edges of the knowledge graph—our algorithm can perform an initial pruning step that pairs off many of the nodes deterministically, leaving only a degree-2 surviving subgraph. We then run a simple randomized matching algorithm on this subgraph using a coin-flipping technique similar to that of Law and Siu [16] to resolve conflicts.

From a very high-level perspective, the algorithm proceeds as follows. Start with an directed graph G with maximum degree d. Each node starts by sending a **probe** to all of its successors. The recipient of such a probe responds by **accepting** the first one and **rejecting** subsequent probes; in this way every node has at most one designated predecessor, producing a graph of designated predecessors G_1 in which every node has in-degree at most 1. This graph is further pruned by having each node with two or more successors pair them off, leaving a graph G_2 in which every node has both in-degree and out-degree at most 1. Each component in such a graph is either a line or a cycle, and a constant fraction of the nodes can be matched along edges by simply having each node that is not at the end of a line flip a coin to decide whether to pair with its remaining predecessor or successor, and pairing those adjacent nodes whose choices are consistent. A simple calculation shows that on average half of the nodes in G_2 (and all nodes in $G - G_2$) are paired by this procedure, from which we can deduce that about half of the nodes are paired on average in the worst case where $G - G_2$ is empty.

The algorithm sketched above can be implemented directly in a synchronous system where all nodes start simultaneously, because after the initial probing phase there is no confusion about the structure of the graph, and after a phase consisting of a known number of rounds any unmatched nodes can simply restart the protocol along with any supernodes resulting from merges in the previous phase. But in an asynchronous setting the situation is more complicated. While some of the early pruning steps can still be used (in particular, we still have each node accept and respond to a single designated predecessor), the final matching stages require more care.

There are two main problems. The first is that no node can detect when a phase of the pairing protocol has finished, so that an unmatched node cannot detect the end of a pairing phase and restart the protocol. Instead, the best an unmatched node can hope for is that the faithless suitor who spurned it initially will return to accept its advances after it finishes digesting luckier candidates. But this creates the possibility of creating very long chains of nodes, each waiting for the next to finish a merge that is itself delayed by waiting for nodes further down the chain.

This problem is compounded by the fact that a node that has not yet received a probe cannot tell whether it has no

predecessors or merely slow predecessors. If an unprobed node simply assumes that it has no predecessors and that it should pair with any successor that accepts it, the adversary can schedule events so that every node in a long chain only learns of its unrequited predecessor after it has committed to its successor, creating the linear-time backlog described above. On the other hand, if a node chooses to wait for a predecessor to come, it may be left stuck in this state forever.

The solution is to retain the coin-flip by which a node chooses its orientation, but let the presence of a successor who wants to pair now override the wait for a predecessor that may never arrive. In addition, the successor-pairing procedure is modified slightly: instead of pairing all successors, possibly leaving none, a process always saves the first successor for itself and pairs off only subsequent pairs. This may leave an odd successor that is not paired, but there is at most one such node left out for each node that participates in the (now very implicit) randomized matching protocol. If this left-out node is waiting for its predecessor, it will eventually be picked up after the predecessor merges with its preferred successor.

What makes all of this work is that the randomization breaks up long waiting chains: it is unlikely that a long chain of nodes will all be pointed the same way by their coin-flips. At the same time, opportunistic merging by nodes with the first available suitor prevents deadlocks in cycles, even if all nodes are pointed in the same direction, as some node's proposal gets in first.

3.1.1 Details of the Algorithm

Formally, each node can be in one of four different states, depending on what messages it has received. The four different states and their attitudes towards incoming pairing proposals are described as follows:

- ISOLATED: The node has not yet received any probes and has no predecessor. So once it receives a proposal from its successor, it can accept it immediately.

- PROBED: The node has been probed, but not yet been told whether it is paired off by its predecessor, nor it has a pairable successor. In this case, it waits to find out what its predecessor will do with it. If it receives a proposal from its successor and its coin is also pointed to its successor, it enters the PROPOSED state. If the proposal conflicts with its coin, it refuses the proposal immediately.

- PROPOSED: The node has a waiting proposal, but has not yet been told whether it is paired off by its predecessor. It defers responding to the proposal until its state changes due to the notice from its predecessor.

- PROPOSING: The node has a predecessor and has been told by the predecessor that it is not paired off. So the node should actively send out a proposal in the direction indicated by its coin and accept immediately any proposal that does not conflict with its coin. Proposals that conflict with its coin are refused immediately.

- PAIRED: The node has been paired, either by its predecessor or due to coins. It refuses any proposals (although it may later be available for new proposals once it has completed a merge operation with its partner).

For each node u, we assume that it maintains a **neighbors** set N_u, which induces an underlying graph G with edges (u, v) for all pairs with $v \in N_u$. Initially N_u consists of those nodes whose identity u knows at the start of the protocol. It is updated by adding any node that sends u a probe message. Neighbor sets are merged when two nodes join into a single supernode. A neighbor that refuses a proposal is removed: this prevents a slow node from being pestered by arbitrarily large numbers of duplicate proposals from a faster neighbor, since the faster neighbor will only try again after the slow node has rejoined its neighbor set (by sending out a probe message after completing a merge).

The algorithm is described below. It has a main thread which is responsible for the main function of the pairing, and four daemon threads which are triggered by messages and responsible for state transitions. The execution of the daemon threads should be implemented to be atomic, which is quite reasonable because there is no waiting in the daemon threads and our model ignores the running time of a process.

For each node u:

1. Let *state*←ISOLATED; let *chosen* be picked uniformly at random from $\{pred, succ\}$.
2. For each $v \in$*neighbors*, send a message $(u, v,$ probe$)$ to v and wait for all replies;
3. Let $v_1, v_2, \ldots v_k$ be the nodes that reply with 'accept.' For each odd i less than k, send a message $(u, v_i,$ pair$, v_{i+1})$ to v_i and $(u, v_{i+1},$ pair$, v_i)$ to v_{i+1}. If k is odd, let *succ*← v_k, and send a message $(u, succ,$ no_pair$)$ to the node *succ*; else let *chosen*←*pred*;
4. While (*state*=ISOLATED or *state*=PROBED) wait;
5. If *state*=PROPOSING then:

   Send a message $(u, chosen,$ propose$)$ to the node *chosen*;

   If reply is $(chosen, u,$ accept$)$ then let *state*←PAIRED and *obj*←*chosen*;

   else if reply is $(chosen, u,$ reject_propose$)$ then let *neighbors*←*neighbors*−$\{chosen\}$

   else if reply is $(chosen, u,$ paired$)$ then do nothing but proceed;
6. If *state*=PAIRED, then merge with *obj*;
7. Go to line 1.

Upon receiving message $(v, u,$ probe$)$ from node v **do**:

Let *neighbors*←*neighbors*∪$\{v\}$;
If *state*=ISOLATED then:
  Let *pred*← v and *state*←PROBED;
  Send a message $(u, v,$ accept$)$ to node v;
else:
  Send a message $(u, v,$ reject$)$ to node v.

Upon receiving message $(v, u,$ propose$)$ from node v **do** the one of the following according to the value of *state*:

ISOLATED: Let *state*←PAIRED and *obj*←v; reply with $(u, v,$ accept$)$;

PROBED: If *chosen*=v, then let *waiting*← v and *state*←PROPOSED;
if otherwise, reply with $(u, v,$ reject_propose$)$;

PROPOSING: If *chosen*=v, then let
state←PAIRED and *obj*←v, and
reply with $(u, v,$ accept);
if otherwise, reply with $(u, v,$ reject_propose);
PROPOSED: Reply with $(u, v,$ reject_propose);
PAIRED: Reply with $(u, v,$ paired).

Upon receiving message $(pred, u,$ pair$, w)$ from node *pred* **do**:
If *state*≠PAIRED then:
Let *state*←PAIRED and *obj*←w;
If *state* = PROPOSED then:
Send a message $(u, waiting,$ paired) to node *waiting*.

Upon receiving message $(pred, u,$ no_pair) from node *pred* **do**:
If *state*=PROBED then:
Let *state*←PROPOSING;
else if *state* = PROPOSED then:
Let *state*←PAIRED and *obj*←*waiting*;
Send a message $(u, waiting,$ accept) to node *waiting*.

The algorithm terminates when there is only one node remaining.

3.1.2 Analysis

The analysis of the pairing algorithm is quite involved, and can be found in the full paper. We give some highlights of the argument here. The basic idea is to analyze the sequence of graphs G_t derived from the neighbor lists N_u at each time t. We use M for the time to perform a merge operation and D for the maximum message delay; note that since the nodes in the protocol may in fact be trees simulating single supernodes, D can be as large as the depth of the tree.

First, we show that for any edge (u, v) in the initial knowledge graph G_0, at least one of (u, v) or (v, u) (taking into account new identities assumed by supernodes that absorb them) appears in G_t until the nodes merge; this implies that the operation of the algorithm does not disconnect the graph.

Second, we define an **iteration** as an interval between times when the node enters the ISOLATED state, and show by case analysis that during each iteration, a node remains at most $O(D)$ time in the PROBED, PROPOSED, and PROPOSING states, including time waiting for a neighbor to respond in the PROPOSING state, and at most $M+O(D)$ time in the PAIRED state, since this state leads immediately to a merge. Bounding the time in the ISOLATED state requires a more detailed analysis, but by considering all possible states of the node's neighbors we can show that $T(\text{ISOLATED}) \leq T(\text{PROBED}) + T(\text{PROPOSING}) + T(\text{PROPOSED}) + T(\text{PAIRED}) + O(D) = O(M+D)$. Since each state can be entered at most once during a single iteration, the maximum time T_I for an iteration is $O(M + D)$.

Finally, we show by an argument similar to that sketched out for the simple pairing algorithm that during an iteration, for each node there is a probability of at least 1/2 that the node either is paired or is the unique unpaired successor of a predecessor that is paired. The situation is slightly complicated by the fact that iterations are not synchronized across nodes, but with some additional work it is possible to show that the expected number of surviving supernodes at the end of any interval of $2T_I$ time units is at most 8/9 of the number at the start. This suffices to prove:

THEOREM 1. *The expected running time of the pairing protocol is* $O((M + D) \log n)$.

For the Patricia trees described in Section 3.2, D and M are both proportional to the maximum depth W given by the length in bits of node identifiers, giving a cost of $O(W \log n)$. In the synchronous model, an additional delay of $O(d)$ time units is imposed on each probing step, because each leaf node may have to probe up to d neighbors and is limited to sending one message per time unit. This gives a synchronous running time of $O((d + W) \log n)$.

3.2 Merging

In this section, we describe a distributed implementation of a variant on **big-endian Patricia trees** [18]. This implementation permits two trees to be merged in time $O(W)$ with $O(1)$ contention and $O(\min(n+m, W \min(n, m)))$ messages of size $O(W)$ each, where W is the length in bits of an identifier and n and m are the number of elements in the two trees.

A Patricia tree [17] is similar to a binary **trie** [5,6] with all keys stored in the leaves, except that paths with no branches are compressed to single edges. We assume that all keys are bit-strings with fixed width W: shorter strings can be padded with zeros. Each node in the tree stores a prefix that is common to all of the keys in its subtree. The two children of a node with prefix x store prefixes that begin with $x0$ and $x1$; it is possible that their prefixes will be longer if all nodes with prefix xb have additional prefix bits in common.

In our implementation, keys are identifiers of processes, and there is a natural one-to-one correspondence between keys, processes, and the leaves of the tree. To allow operations to be performed in parallel on internal nodes, we must also assign processes to these nodes. Because Patricia trees are binary trees, there are exactly $n - 1$ internal nodes in a Patricia tree with n leaf nodes. Thus we can assign one internal node to each process, leaving one process left over. Each process is thus responsible for simulating at most two nodes in the tree. If we think of the nodes in the tree as simulated processes, the contention on any real process is at most twice the contention on any simulated process.

The unused "internal node" of the leftover process is kept in reserve as an **extra node** by the root of the tree. When two trees merge, the extra node from one of them is used to supply the new internal node required for the merge, and the other is kept in reserve for a subsequent merge. Note that in the initial state of any process, it is both leaf and root of a singleton tree, and thus acts as its own extra node. The use of such extra nodes to avoid the need for a global allocation mechanism for internal nodes is the main technical trick that distinguishes our merge algorithm from Okasaki-Gill.

Let us now describe the merge procedure. Intuitively, when two Patricia trees merge, either their roots share a common prefix, in which case the roots are combined and the matching subtrees of each tree are merged in parallel; or one root's prefix is a prefix of another, in which case the root with the longer prefix is merged into the appropriate child of the other; or the prefixes are incomparable, in which case the two old roots become children of a new root. What makes it possible to pipeline this procedure is that wave of merging proceeds down the trees one layer at a time, and as soon as the roots of merging subtrees have communicated they can determine immediately which node becomes the root of the new subtree and which nodes are denoted to extra nodes in inferior merges, merged with children, etc. This immediate determination of the new root means that

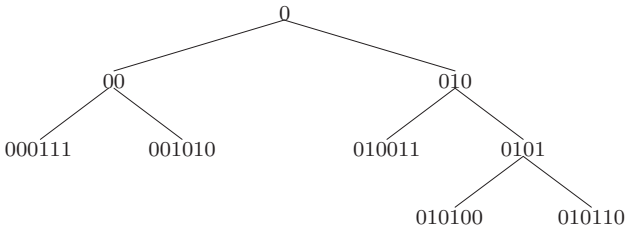

Figure 1: A Patricia tree storing the strings 000111, 001010, 010011, 010100, and 010110.

it can begin a second merge in $O(1)$ time, waiting only to fix the identities of its new children. It follows that the first merge takes $O(W)$ time, but that subsequent merges add only $O(1)$ additional time, allowing up to 2^k merge operations to be completed in $O(k+W)$ time if the tree of merges is perfectly balanced. Unfortunately, our pairing algorithm is not sophisticated enough to construct a balanced tree of merges in parallel with the merges occurring, but we can imagine situations where such pipelining may be useful.

Formally, a non-root node x in the tree stores a bit string x.prefix and two child pointers x.child[0] and x.child[1]. For leaf nodes, x.prefix is a key of width W and both child pointers are null ($\perp$). For internal nodes, x.prefix is the common prefix of all keys in the subtree rooted at x and both child pointers are non-null. A root node stores in addition to these values a pointer x.extra to the extra node for the tree.

We write $x \sqsubseteq y$ if x is a prefix of y. The invariant for the tree is: For each $b \in \{0,1\}$, if x.child[b] $\neq \perp$, then x.child[$1-b$] $\neq \perp$ and x.prefix $+ b \sqsubseteq x$.child[b].prefix. See Figure 1 for an example.

3.2.1 Merging Two Patricia Trees: Global View

Below is a global description of the merging algorithm. Though this is given as pseudocode for a centralized controller, the reader should not be misled into thinking that a centralized controller is required; instead, all steps of the merge can be carried out by direct communication between nodes in the trees, as we will describe later.

```
// Merge two Patricia trees in parallel, returning the new root
// extra is the unused node needed for this merge procedure
procedure Merge(x, y, extra)
  if x.prefix = y.prefix                                   (Case 1)
    // combine roots and merge matching subtrees in parallel:
    x.child[0] ←Merge(x.child[0], y.child[0], extra)
    // y is freed and used as the extra node for submerge
    x.child[1] ←Merge(x.child[1], y.child[1], y)
    return x
  else if x.prefix ⊑ y.prefix                              (Case 2)
    // merge y with appropriate subtree of x
    let b be the first bit in y.prefix that is not in x.prefix
    x.child[b] ←Merge(x.child[b], y, extra)
    return x
  else if y.prefix ⊑ x.prefix                              (Case 3)
    // merge x with appropriate subtree of y
    let b be the first bit in x.prefix that is not in y.prefix
    y.child[b] ←Merge(y.child[b], x, extra)
    return y
  else if x.prefix and y.prefix are incomparable           (Case 4)
    // use the extra node to create a new node that holds the common prefix
    let p be the longest common prefix of x.prefix and y.prefix
    let b be the first bit of x.prefix that differs from y.prefix
    allocate a new node z = extra with:
      z.prefix = p
      z.child[b] = x
      z.child[1 − b] = y
    return z
```

3.2.2 Merging Two Patricia Trees: Local View

As described above, we think of each node in the tree as represented by a "virtual process" whose operations are carried out by one of the actual processes in the system. Allocating a new node consists of allocating a new virtual process at some physical process. We extend our message-passing model to allow virtual processes to communicate with each other; this involves the corresponding physical processes sending messages on the virtual processes' behalf, with appropriate tags to distinguish between a physical process's multiple virtual processes.

Each call to Merge is handled by the process that holds node x, and is triggered by a message (v, x, merge, y) from some **initiator** v, typically x's parent. In response, x first queries y for its state with a message $(x, y, \text{getState})$ and y responds with $(y, x, \text{returnState}, (y.\text{prefix}, y.\text{child}[0], y.\text{child}[1]))$.[3] Upon receiving y's state, x applies one of the four cases of the Merge procedure, issuing up to two merge messages and waiting for corresponding return messages before sending its own return back to the initiator v.

Some additional machinery is needed to avoid sending merge operations to children before they are ready; details are given in the full paper.

THEOREM 2. *Merging two Patricia trees of size n and m with W-bit keys requires $O(W)$ time, $O(1)$ contention, and $O(\min(n+m, W\min(n,m)))$ messages of $O(W)$ bits each.*

PROOF. Start with the time bound. Consider a single call to Merge as executed by some process x. The time for x to complete this merge operation is the maximum time of any recursive merges, plus $O(1)$ time to send and receive all the messages at the top level. Note that except for possibly receiving a return message, x is idle during the recursive calls, so its physical process can simulate other virtual processes with at most one time unit of delay (to receive the return) during this time without increasing the contention beyond $O(1)$ as long as it simulates only one per branch of the tree. The running time increases by $O(1)$ for each level of the tree, giving a total time of $O(W)$.

Now let us consider the total number of messages. Each recursive call to Merge generates $O(1)$ additional messages, so we just need to bound the number of such calls. Each call returns a distinct node in the combined tree, and the number of internal nodes in the combined tree is bounded by $n+m$, the number of leaves; this gives a bound of $n+m$. To get the other side of the bound, let $T(w,n,m)$ be the number of calls needed to merge two trees where w is the number of bits that are not in the common prefix of both trees, and n and m are the number of elements in the two trees. Let n_0 and n_1 be the number of elements in the left and right subtrees of the first tree, and m_0 and m_1 be the number of elements in the corresponding subtrees of the second tree. Then we have

$$T(w,n,m) \leq 1 + T(w-1, n_0, m_0) + T(w-1, n_1, m_1),$$

[3]To escape the one-identifier-per-message restriction, this can be sent as three consecutive messages

with $T(0,\cdot,\cdot) = T(\cdot,0,\cdot) = T(\cdot,\cdot,0) = 0$.

We will now show by induction on $w+m+n$ that $T(w,m,n) \leq w\min(m,n)$. Clearly this holds for the base cases. Now consider $T(w,n,m)$ where w, n, and m are all nonzero, and suppose the bound holds for $w'+n'+m' < w+n+m$. Then

$$\begin{aligned} T(w,n,m) &\leq 1 + T(w-1,n_0,m_0) + T(w-1,n_1,m_1) \\ &\leq 1 + (w-1)\min(n_0,m_0) + (w-1)\min(n_1,m_1) \\ &\leq 1 + (w-1)\min(n,m) \\ &\leq w\min(n,m). \end{aligned}$$

□

3.2.3 Merging Many Patricia trees

To merge many Patricia trees, observe first that in the pairwise merge procedure above a new root is determined immediately—it is not necessary to wait for the recursive merges of subtrees to complete. It is thus possible to start a new merge between the combined tree (as represented by its new root) and another Patricia tree (which may also be the result of a recently-initiated merge) without waiting for the subtree merges to complete. If we think of the first merge operation as a wave propagating down through the trees, the second merge operation propagates as a wave just a few steps behind it. Subsequent merge operations can be similarly pipelined, so long as we have enough processes to handle the operations at the individual tree nodes. The result is that a tree of merges of maximum depth k can be completed in $O(k+W)$ time.

We do not use pipelined merges, as limitations of the pairing algorithm and supernode simulation require merges to be carried out sequentially. However, we can imagine an improved pairing algorithm that chooses new pairings while merges are still in progress. The ability to pipeline merges may also have other applications, such as building a Patricia tree from a pre-existing but unsorted balanced tree of nodes obtained by some other means.

3.3 Simulating Supernodes

Intuitively, the idea behind the tree-construction algorithm is to use the merging procedure to join each pair of nodes selected by the pairing algorithm into a single component that acts like a single "supernode" in the next round of pairing. If contention were not an issue, we could simulate such a supernode easily by choosing a single representative of each component, and have it handle all the edges that previously went into some member of the component.

The problem with this approach is that we will quickly raise the degree of the representative nodes; toward the end of the algorithm we this accumulation of edges could produce both linear contention and a linear slowdown in the pairing procedure. To avoid these problems, we organize the members of a component in a binary tree, and leave the task of communicating with other components to the leaves, with the root acting as a global coordinator. In this construction, the *neighbors* set is distributed across the leaf nodes, while the other components of the supernode's state reside at the root. The resulting protocol is similar in many ways to the classic distributed minimum spanning tree protocol of Gallager *et al.* [8], although the internal communication costs of components are reduced by our ability to construct a balanced tree by adding new edges to the communication graph.

In carrying out this strategy, we have to be very careful to ensure that the atomic operations of the daemon threads continue to appear atomic. If we were willing to accept substantial overhead, a natural approach would be to serialize the processing of incoming messages at the leaves using a parallel-prefix computation [15], taking advantage that the effects of processing the various incoming messages on the states can be summarized as simple state updates plus possible assignment to the variables *waiting* and *obj*, and such operations are composable. However, this approach could in the worst case require a parallel-prefix operation on the entire tree to collect and process a single incoming message, giving an $O(n)$ worst-case blow-up in message traffic. Instead, we will consider the structure of the pairing algorithm carefully, and show that many incoming messages can be sent directly to the root of the tree without significantly increasing contention, while others can be processed using a convergecast operations.

The key observation is that at any time a supernode has at most one designated predecessor and at most one designated successor node, and that only these nodes can send propose, pair, and no_pair messages. So these messages (and their responses) can be sent directly between roots, and the roots can update the state of the simulated supernode locally. (To enable this, we assume that all messages are extended by including the identity of the root node of the sending component, and all message-ids within.) For reasons of space, we do not discuss processing of these messages further. However, probe messages and their responses occur in much greater abundance, and thus require special handling.

3.3.1 Consolidating Probes and Probe Responses

Probe messages appear in the algorithm in two places: in the main thread, the supernode sends probes to all neighbors and waits for responses, and in the daemon thread handling received probes, the supernode must accept only the first probe. The main thread must also collect up to d responses per leaf node and pair off those that accept.

The task of sending probes and collecting responses is handled by a modified convergecast procedure, initiated by the root. Pseudocode for each node's role in this procedure is given below. A wrinkle that does not appear in the simple pairing algorithm is the same_component response; this allows a node to detect that its neighbor is in the same component and should not be troubled further.

Upon receiving message $(parent, u, \text{initiate\_probe})$ from node *parent* **do**:

If u is a leaf node:

For each $v \in neighbors$, send a message $(u, v, \text{probe}, root)$ to v and wait for all replies.

For each node v that replies with $(v, u, \text{same\_component})$, remove v from *neighbors*.

Let $v_1, \ldots v_k$ be the nodes that reply with 'accept.' For each odd i less than k, send a message $(u, v_i, \text{pair}, v_{i+1})$ to v_i and $(u, v_{i+1}, \text{pair}, v_i)$ to v_{i+1}. If k is odd, send $(u, parent, \text{respond\_probe}, v_k)$ to *parent*.

Else u is an internal node:

For each child node c, send $(u, c, \text{initiate\_probe})$ and wait for all replies.

Let V be the union of all sets of nodes appearing in the replies. If $V = v_1, v_2$, send messages $(u, v_1, \text{pair}, v_2)$ and $(u, v_2, \text{pair}, v_1)$ to v_1 and v_2, and send $(u, parent, \text{respond\_probe},)$ to *parent*. If instead $|V| < 1$, send $(u, parent, \text{respond\_probe}, V)$ to *parent*.

The code at the root is similar to the case for an internal node, except that a singleton surviving element of V becomes the chosen successor as in the original pairing algorithm.

Note the addition of the *root* variable to the probe message, which tracks the root of the component that u belongs to. We assume that this variable is updated as part of the merge protocol.

Incoming probes are similarly handled by a convergecast operation. The idea is that any node receiving one or more probes forwards the first to its parent and rejects any others. This allows the root to accept exactly one incoming probe on behalf of the simulated supernode. Enforcement of the reject-all-but-one strategy is handled using a flag *probed* that is reset during the merge procedure. Note that this flag appears in both leaf and internal nodes; a process simulating two such nodes must maintain two separate flags.

Upon receiving message $(v, u, \text{probe}, root')$ from node v **do**:

Let $neighbors \leftarrow neighbors \cup \{v\}$.

If $root' = root$ send $(u, v, \text{same\_component})$ to v.

Else If $probed = 0$: send $(u, parent, \text{probed}, u, root')$ to $parent$ and set $probed \leftarrow 1$.

Else Send (u, v, reject) to v.

Upon receiving message $(v, u, \text{probed}, leaf, root')$ from node v **do**:

If $probed = 0$: send $(u, parent, \text{probed}, leaf, root')$ to $parent$ and set $probed \leftarrow 1$.

Else Send $(leaf, v, \text{reject})$ to v.

The root responds to probed messages as if they were probes: accepting and switching to a PROBED state if in an ISOLATED state, and rejecting otherwise.

It is not difficult to see that these procedures correctly simulate the behavior of the pairing algorithm in handling probe messages. The only tricky part of this analysis is to argue that message arrivals are serialized properly: in particular, responses to probes arrive at times that are consistent with the behavior of a single node running the pairing algorithm. But here the assumption that the system is asynchronous works for us, as the algorithm guarantees that a probe is rejected only if some other probe can be accepted, and the delay in propagating any probe that is ultimately accepted up the tree can be hidden behind asynchronous message delays. For a synchronous system, we can instead explicitly delay responding to any probes until the convergecast has had time to terminate. We omit the details.

3.4 Building a Ring

The preceding sections allow us to build a tree of nodes quickly, but do not quite achieve our original goal of building the sorted base ring of a ring-structured distributed data structure like Chord or a skip graph. Building such a ring is, fortunately, an easy extension of building a binary tree, as it is enough for each leaf node to know its successor, which can be computed quickly from the tree structure. A natural way to do this is to have each node in the tree keep track of its minimum and maximum leaf, values which can easily be updated during the merge procedure. To inform a leaf node of its successor, we pass with each recursive call to Merge the identity of the first node to the right of the trees being merged (the minimum-key leaf node in the case of the rightmost trees). This value is either obtained from the parent call, for the right subtrees, or by taking the minimum of the leftmost values of the right subtrees, for the left subtrees. Details are given in the full paper.

3.5 Failures, Latecomers, and Churn

An important issue in building peer-to-peer systems is **churn**, the rapid arrival and departure of component nodes. Arrival of new nodes is not a problem: we can simply treat them as very slow nodes in the asynchronous pairing algorithm. But our algorithms do not deal well with node departures; indeed, the failure of any node during the tree-construction procedure could in principle lead to deadlock of the entire system. We believe that a judicious use of timeouts combined with restarting parts of the algorithm could handle such difficulties, but avoiding deadlocks or inconsistencies will require substantial further work.

4. LOWER BOUNDS

The upper bound of $O((d+W)\log n)$ on time to sort nodes in a weakly-connected graph contrasts with our best current lower bound of $\Omega(d + \log n)$, which is proved in this section. We suspect that the lower bound is closer to the optimal time; this issue is discussed further in Section 5.

The model we use here is a simplification of the one defined in Section 2. During the algorithm, each vertex in a weakly connected directed graph G represents a process with some unique identifier u, maintaining a knowledge set K_u which contains the identifiers of endpoints of all its outgoing edges. A communication is denoted as a triple (u, v, w), where u, v and w are processes identifiers as well as $v, w \in K_u$, and after the communication, K_v becomes $K_v \cup \{w\}$, as well as a new directed edge from v to w is added to G. A procedure is defined as a sequence of such triples, which are arranged into different time units, where in each time unit there is at most one triple starting with u for any u in G. Besides, a total order of all identifiers is given and for each identifier u, there is a unique successor, denoted as $succ(u)$. A procedure is said to yield a sorted list from G, if after the procedure, for any u in G, it holds that $succ(u) \in K_u$.

One may easily notice that this model only captures the aspect of exchanges of knowledge of identifiers. However, since the knowledge availability is a necessary condition for the sorting problem, this model is sufficient to build the lower bounds.

In the full paper, we show that (a) any graph with diameter Δ requires $\Omega(\log \Delta)$ time to sort in the worst case, and (b) any graph that can be separated into d components by the removal of a single vertex requires $\Omega(d)$ time to sort in the worst case. Considering a graph consisting of a degree-d star with a chain of $n - d - 1$ nodes attached to one of its outer vertices then gives:

THEOREM 3. *For any $n > d > 0$, there exists a degree-d weakly-connected graph of n nodes with some identifier permutation, such that, for any procedure, yielding a sorted list from this graph requires $\Omega(d + \log(n - d + 1))$ time.*

PROOF. Let G be a graph whose underlying graph is a d-degree star with a chain of $n - d - 1$ vertices connected to one of its outer vertices. Notice that the diameter of G's underlying graph is $n-d+1$, and after removing the central vertex of star, G is separated into d components. Then by the above two lemmas, it is easy to obtain the lower bound on the running time $\Omega(d + \log(n - d + 1))$. □

5. CONCLUSIONS

We have described an asynchronous distributed algorithm for quickly converting the nodes in a weakly-connected pointer graph into the leaves of a Patricia tree with depth bounded by the length of node identifiers. Applications of this protocol include solving resource discovery or leader election subject to contention, message-size, or memory constraints that limit how many identifiers can be transmitted in a single message or stored in a single node; and the construction of sorted lists as a foundation for more sophisticated peer-to-peer data structures. The application to peer-to-peer data structures is of interest not only for building such structures quickly *ab initio*, but also as a repair mechanism for damaged structures; our protocol is fast enough that it is would not be unreasonable to use it to repair damaged structures by periodically rebuilding them from scratch.

Our analysis assumes that no failures occur during its execution, an assumption unlikely to hold in practice despite the speed of the algorithm. Handling failures is an interesting avenue for future work.

Finally, the optimal running time of self-sorting starting from a weakly-connected knowledge graph remains open. Our upper bound of $O((d + W) \log n)$ for the synchronous model is tantalizingly close to our lower bound of $\Omega(d + \log n)$, and it would not be surprising if the extra near-logarithmic factor of W could be eliminated by pipelining the effectively sequential phases of the pairing algorithm. The key difficulty is that until a merge reaches them, it is difficult for the nodes deep in a newly-combined component to distinguish between external edges and edges that are now internal to the component. Such an improvement would require either a mechanism for discarding internal edges from each component quickly, or for choosing candidate external edges in a way that produced good pairings with high probability. We plan to pursue such possibilities in future work.

6. REFERENCES

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. *Lecture Notes in Computer Science*, 2172:179–194, 2001.

[2] 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.

[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] 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.

[5] R. de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 295–298, Montvale, NJ, USA, 1959.

[6] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[7] M. J. Freedman and R. Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[8] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[9] 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.

[10] 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.

[11] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.

[12] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 337–346. ACM Press, 1993.

[13] 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.

[14] 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.

[15] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[16] 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.

[17] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[18] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[19] 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.

[20] R. Uehara and Z.-Z. Chen. Parallel approximation algorithms for maximum weighted matching in general graphs. In *Proceedings of IFIP TCS 2000*, pages 84–98. Springer-Verlag, LNCS 1872, 2000.

[21] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-btree: An update-conscious parallel directory structure. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 448–457. IEEE Computer Society, 1999.