An Efficient Deterministic Algorithm for the Resource Discovery Problem

Vijay K. Garg [*] Adnan Aziz [†]

Electrical and Computer Engineering

The University of Texas at Austin

Abstract

We address the problem of how best to get a group of machines on a network to learn of each others existence; this is referred to as the *Resource Discovery Problem* (RDP). Straightforward algorithms for RDP are slow or have high communication cost. Harchol *et al.* [3] recently presented *name-dropper*, a randomized distributed algorithm for RDP which has low time and communication complexity. However, *name-dropper* has significant limitations — (1.) the use of randomization precludes it from providing a guaranteed bound on runtime, and, more significantly, (2.) it has no mechanism by which convergence can be detected; in order to provide a high probability of convergence, the number of machines on the network must be known *a priori* to all machines. We present *fast-leader*, a deterministic distributed algorithm for RDP which overcomes these limitations while matching *name-dropper*'s time and communication costs.

`DISC 2000 Submission: regular paper track`

Vijay K. Garg will serve as the contact author:

`Email: garg@ece.utexas.edu`

`Phone: (512) 475 9424`

`Address: ENS 527, The University of Texas, Austin TX 78712`

[*]Supported in part by the NSF Grants ECS-9907213, CCR-9520540, a General Motors Fellowship, and an IBM grant.

[†]Supported in part by an NSF Career Award and an IBM research partnership award.

1 The Resource Discovery Problem — Introduction

In large distributed networks, it is often the case that a group of machines need to work together in a coordinated fashion. This is the case, for example, in implementations of distributed web caching protocols [4], distributed file systems [10], and the Domain Name Service protocol [6]. In this context, as a first step, it is essential that machines be able to learn about the existence of each other. Harchol *et al.* [3] refer to this as the *Resource Discovery Problem (RDP)*.

A key requirement of any algorithm for RDP is that it be efficient, both in terms of the time taken for machines to learn about each other as well as the amount of communication performed. Time efficiency is critical, since the faster the ensemble converges, the sooner machines can get to work. Communication efficiency is also imperative, because of the cost of bandwidth and the fact that excessive communication leads to greater latencies and consequently higher runtimes. Indeed, the motivation for the work of Harchol *et al.* [3] is the fact that naive algorithms (which we review in Section 2) for RDP are either slow to converge or overload the network.

Our work has been largely inspired by the paper of Harchol *et al.* [3] who present *name-dropper*, a distributed randomized algorithm for RDP. We use the same formulation of the problem as they do. Each machine has a unique ID (which can be thought of as an IP address); for a machine to send a message to another machine, it is necessary that it knows the latter's ID. Furthermore, these ID's are assumed to be elements of a totally-ordered set. The system can be viewed as a directed graph whose vertices correspond to machine ID's; an edge exists from i to j when i knows of the existence of j, in which case we will refer to j as being a *neighbor* of i. An example of this representation is shown in Figure 1(a). The graph evolves as machines discover new machines. Figure 1(b) shows the edges added when machine 4 has sent its neighbor list to 3.

It is natural to restrict attention to *distributed* algorithms for RDP, in which there is no central control and individual machines operate independently, each running its own copy of the same program. Harchol *et al.* [3] model such algorithms as proceeding in synchronous parallel *rounds*; a round is defined as the time taken for each machine in the network to communicate with one or more other machines. They measure the runtime for an algorithm for RDP by the number of rounds needed for each machine to know of every other machine. When this happens, the algorithm is said to have *converged*.

Their metric is fairly crude, as it neglects that fact that communication time is a function of network topology, and varies with load. However, this measure is suitable for asymptotic analysis. They define two other complexity measures, namely the the *connection communication complexity*,

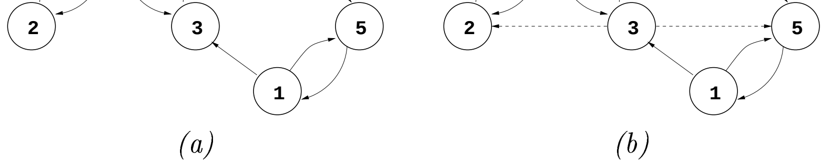

Figure 1: Examples

which is the total number of messages communicated over the duration of the algorithm, and the *pointer communication complexity*, which is the total amount of data communicated over the duration of the algorithm.

Harchol *et al.* [3] prove that the *name-dropper* algorithm is within a poly-logarithmic factor of optimal time and communication complexities. However, *name-dropper* has significant limitations. First, because of randomization, it is intrinsically incapable of providing a guaranteed bound on the runtime; in real-time applications this is unacceptable. More significantly, as Harchol *et al.* [3] acknowledge, *name-dropper* has no mechanism by which it can detect convergence. Knowing when to stop, even when all that can be guaranteed is a high probability of convergence, requires that the number of machines on the network must be *a priori* known by all machines.

In this paper we present *fast-leader*, a deterministic algorithm for RDP whose runtime and communication complexities match those of *name-dropper*. Being deterministic, *fast-leader* algorithm overcomes the first limitation of *name-dropper* described above. Furthermore, we will see that convergence detection is built into the algorithm, thereby overcoming the second limitation of *name-dropper*. Consequently, *fast-leader* halts as soon as convergence is achieved.

One difference between *fast-leader* and *name-dropper* is that unlike *name-dropper*, *fast-leader* requires that the initial graph be strongly connected. In practice, this is easy to ensure: whenever a new machine is added to the system, it is sufficient for the new machine to know some existing machine, and tell that machine about its presence.

The remainder of this paper is organized as follows: In Section 2 we describe previously proposed algorithms and their shortcomings. This is followed in Section 3 with an exposition of *fast-leader*, our new algorithm for RDP. We conclude in Section 4 with a discussion of related problems and future work.

2 Prior work

2.1 The flooding and swamping algorithms

The *flooding* algorithm, currently implemented in Internet routers [7], can be used to solve RDP. In this algorithm, each machine begins with an initial set of neighbors; machines are restricted to communicate with their initial neighbors. At each round, each machine sends to all its initial neighbors the set of machines it has learned of since the previous round. The number of rounds to convergence is $\Theta(d_{\text{initial}})$, where d_{initial} is the diameter of the initial graph. Note that the diameter of a graph can be $\Theta(n)$, where n is the number of vertices present. Hence the runtime for a flooding-based algorithm for RDP can be very high.

It immediately follows from the $\Theta(d_{\text{initial}})$ bound on the number of rounds, that the connection communication complexity of the flooding algorithm is $\Theta(d_{\text{initial}} \cdot m_{\text{initial}})$, where m_{initial} is the number of edges present initially in the graph. Harchol *et al.* [3] demonstrate that the pointer communication complexity for the flooding algorithm is $\Theta(n \cdot m_{\text{initial}})$. In summary, using the flooding algorithm for RDP is not appealing as it has potentially large runtime and high network usage.

The *swamping* algorithm is identical to the flooding algorithm, except that at each round, each machine communicates with its *entire* neighbor set at that round. The swamping algorithm is very fast — it never takes more than $\log n$ rounds to converge. However, it uses excessive communication resources: by simply considering the last round, when the graph is complete, we see that its connection communication complexity is $\Omega(n^2)$, and pointer communication complexity is $\Omega(n^3)$.

2.2 The name-dropper algorithm

Harchol *et al.* [3] proposed the *name-dropper* algorithm for RDP. It proceeds as follows: at each round, each machine selects a neighbor at random, and transmits its own neighbor list to the selected machine. An elaborate proof, based on union-bound and bounded Markov arguments, shows that, with high probability, *name-dropper* will converge in $O(\log^2 n)$ rounds. Since each machine initiates a single connection at each round, *name-dropper* as a connection communication complexity of $O(n \cdot \log^2 n)$, and a pointer communication complexity of $O(n \cdot \log^2 n)$.

Consideration of the diameter of the graph yields a lower bound of $\Omega(\log n)$ rounds on any algorithm for RDP. The bounds for *name-dropper* are poly-logarithmic of optimal [3]. However, *name-dropper* has the limitations of randomness and lack of convergence detection (cf. Page 2).

2.3 Leader election and broadcast

There has been prior work on related problems, such as *leader election* [1, Chapters 3, 14] and *broadcast* [1, Chapter 8]. Research on the leader election problem has focussed on specific topologies, on minimizing the number of messages passed, on obtaining lower bounds, and impossibility results. For example, there have been several papers on on obtaining lower bounds for leader election in ring networks; a tight lower bound of $\Omega(n \cdot \log n)$ has been shown for the number of messages in the average case that need to be passed for leader election [1, page 59]. Research on broadcast has focused on ensuring reliability and preserving the ordering of messages. As such, there appears to have been little work on reducing the number of rounds and the communication complexity for these problems.

One fundamental difference is that in the previous work a message could traverse only the initial set of edges; whereas in our model once a processor P_i learns about (the identity of) P_j, it can directly send a message to P_j. Thus, all the algorithms discussed in literature for leader election would take at least $O(n)$ rounds if they are adapted to our model.

2.4 Parallel Algorithms for Determining Connected Components

There has been considerable work on parallel algorithms for determining connected components in a graph. The problem requires the algorithm to assign a unique label to all nodes in the same connected component. Usually the identity of one of the nodes in the component is chosen as the label. Therefore, a connected components algorithm can also be used for leader election by viewing the label of the component as its leader. There are many parallel algorithms for determining connected components. The algorithm by Chin, Lam and Chen [2] determines connected components in $O(\log^2 n)$ time with $O(n^2)$ work on a common CRCW PRAM model. The algorithm by Shiloach and Vishkin [8] determines connected component in $O(\log n)$ time with $O((m + n) \cdot \log n)$ on an arbitrary CRCW model where m is the number of edges in the graph. Both of these algorithms assume that in one time step every edge can be examined (in parallel). While this is true in PRAM models, it does not apply to our framework. We have assumed that every node initiates contact with at most one node in every round. With this constraint one time step of the CRCW PRAM may require $O(n)$ rounds in a distributed system. Thus, these algorithms have time complexity of $O(n \cdot \log n)$ rounds when used in a distributed system.

3 Resource Discovery Algorithm

3.1 Notation

We model the evolution of the graph by means of the "knows" predicate $K(i,j,r)$. At round r the predicate $K(i,j,r)$ denotes that machine i knows machine j at the beginning of the round r. By convention, rounds are numbered starting from 1. The predicate $K(i,j,1)$ is simply the initial graph. By definition of "knows", we assume that $K(i,i,r)$ is true for all i and r. Clearly, the predicate K is monotone in r, i.e.,

$$K(i,j,r) \Rightarrow \forall r' \geq r \, : \, K(i,j,r').$$

The *parent* of machine i at round r is the machine with the highest ID that i knows of at the beginning of round r. Formally, we define the function

$$\mathit{parent}(i,r) \;=\; \max\{\, j \mid K(i,j,r)\}.$$

Observe that the parent always exists because $K(i,i,r)$ for all i and r.

It may be that $\mathit{parent}(i,r)$ equals i. Machines for which this condition holds will be referred to as *leaders* at round r. Formally,

$$\mathit{leader}(i,r) \;\equiv\; (\mathit{parent}(i,r) = i).$$

Note that the *leader* predicate is monotonically decreasing in r, i.e.,

$$\neg \mathit{leader}(i,r) \Rightarrow \forall r' \geq r \, : \, \neg \mathit{leader}(i,r').$$

We associate a *level* number with each machine at round r. Conceptually, the level number of i is the maximum round number up to which i was a leader. Formally, we define the function

$$\mathit{level}(i,r) \;=\; \max\left(\{0\} \cup \{\, k \mid \mathit{leader}(i,k) \wedge k \leq r\}\right).$$

Note that if the machine was never a leader, then its level number is 0 and will remain 0. Furthermore, the level number of a machine continues to increase with each round while it is a leader. Once it ceases to be a leader, it can never become a leader and hence its level number will not change after that.

3.2 The *fast-leader* algorithm

We now describe *fast-leader* — a deterministic algorithm for resource discovery. We prove its correctness and demonstrate that it converges in $O(\log^2 n)$ rounds. As discussed on page 2, we assume that the initial graph is strongly connected.

The *fast-leader* algorithm proceeds in two steps:

1. *Leader Election*: This is the main component of the algorithm. A leader election algorithm converts a strongly connected graph to a star-graph in which one machine directly knows all other machines in the graph. This stage of the algorithm requires $O(\log^2 n)$ rounds.

2. *Broadcast*: This stage ensures that everyone knows about everyone. A star-graph can be converted to a complete graph in a single round by having the leader broadcast its information to all machines. This round adds $\Theta(n)$ to message complexity and $\Theta(n^2)$ to pointer communication complexity.

Our algorithm also consists of synchronous rounds as in [3]. At any round r, every machine i has for each of its neighbors k an estimate of $level(k, r)$; these estimates are initialized to 0 at the beginning of round 1.

In each round a machine, P_i, initiates contact with exactly one neighbor, P_j, and *exchanges* its neighbor information with P_j. Specifically, P_i creates a message containing its own ID and level number, as well as the ID's for all of its neighbors, together with its estimates of their level numbers. The machine P_j does the same; these two messages are then exchanged.

Note that the level number that P_i has for itself is exact. However, the level numbers that it knows for its neighbors are estimates: they are lower bounds for the neighbors current level numbers, as they may have been determined in previous rounds, or from other machines. When P_j receives the message, it records all new ID's and the level numbers sent for these ID's, as well as updates its own estimates for level numbers for machines whose ID's it already knows about but for whom its estimated level is lower than the corresponding value received from P_i. The machine P_i handles the message from P_j in a symmetric manner.

The idea of exchanging information with a neighbor is similar to that of [3]; the key difference is in the selection mechanism. Whereas *name-dropper* simply chooses one neighbor at random, our algorithm uses a deterministic strategy to choose a "useful" neighbor to communicate with.

Machines select a neighbor to communicate with as follows: any machine which is not a leader always communicates with its parent. A leader u, on the other hand, initiates contact with its *helper* machine for that round. Formally, we define a function $helper(i, r)$ that returns the helper of machine i in round r in Figure 2. We will later show that unless the algorithm has terminated, *helper* will always return non-null value.

Conceptually, if u is a leader, its helper is a machine k which is a neighbor of u, but which did not communicate with u in the previous round. There are two possibilities: either k changed its parent from u to someone else, or it did not. In the first case, u is guaranteed that k knows someone with a bigger identifier and therefore u contacts k. If the first case does not hold, u orders its neighbors by its estimates of their level numbers, breaking ties using ID's; it then contacts the machine with the highest position in this order which did not contact u in the last round.

```
function helper($r$)
    if ($\exists k : parent(k, r-1) \neq i \wedge parent(k, r-2) = i$) {   /* $parent(k,r) = -1$ for $r \leq 0$ */
        return $k$;
    else {
        order { $k \mid K(i,k,r)$ } by level number estimates, breaking ties using ID's
        return $\max\{\ k \mid K(i,k,r) \wedge (parent(k,r-1) \neq i)\}$   /* max returns $nil$ on $\emptyset$ */
    }


function fast-leader:round r
    if $\neg leader(i,r)$
         /*a machine in the neighbor list is given by $(level\ \ estimate, id)$*/
        exchange neighbor list with $parent(i,r)$;
    else
        if $helper(i,r) \neq nil$
            exchange neighbor list with $helper(i,r)$;
        else
            declare yourself as the global leader.
```

Figure 2: Fast Leader Election Algorithm at P_i

The intuition behind the strategy is as follows. The goal of *fast-leader* is to elect a global leader as quickly as possible. Thus it helps to reduce the number of leaders. The first choice of the helper machine eliminates u as the leader. If the first case does not hold, the leader chooses as helper the "largest" machine k it knows, where machines are ordered by level estimates, with ID's being used to break ties. If k knew about u and still did not report to u, then u will be eliminated as a leader. Otherwise, u will be successful in capturing k. Since the algorithm is based on comparison using *(level, id)* pairs, the algorithm favors capturing most recent leaders. Note that there is no guarantee that the helper will be a leader when it is captured.

Observing that every machine *initiates* contact with at most one other machine, it follows that exactly n exchanges take place in each round. Since our algorithm takes $O(\log^2 n)$ rounds, it follows that our algorithm uses at most $O(n \log^2 n)$ messages. Each message contains at most a constant amount of information per machine, so the size of a message is $O(n)$. Every machine needs to store a constant amount of information about each machine M that it knows of, specifically, M's ID and a level estimate for M; leader machines need to store in addition which machines reported to it in the previous two rounds. Thus the asymptotic space complexity of *fast-leader* is $O(n)$, matching that of *name-dropper*.

3.3 Correctness and Complexity of *fast-leader*

We show that whenever the algorithm terminates, i.e., there exists a process which is a leader and does not have a helper, then it is indeed the global leader. Later, we will show that our algorithm always terminates (in $O(log^2)$ rounds).

Theorem 1 Assuming strong connectivity, if i is a leader and has a null helper, then it must be a global leader. Formally,

$$leader(i,r) \wedge (helper(i,r) = nil) \Rightarrow \forall j : K(i,j,r)$$

Proof: First consider the case when $r = 1$. From definition of $helper(i,1)$, there does not exist any j different from i such that $K(i,j,1)$. From strong connectivity, this is possible only when the graph consists of a single vertex i. Since $K(i,i,1)$, the claim holds.

Now consider the case when r is greater than 1. We prove the contrapositive. Assume that there exists j such that $\neg K(i,j,r)$, but $leader(i,r) \wedge (helper(i,r) = nil)$. However, from strong connectivity we know that there exists a sequence of vertices $(a_0, a_1, \cdots, a_m)$ such that $a_0 = i$, $a_m = j$ and $K(a_k, a_{k+1}, 1)$ for all $0 \leq k < m$. Let j' be the first vertex in this path such that $\neg K(i,j',r)$; such a vertex must exist since by hypothesis $\neg K(i,j,r)$. The vertex j' is distinct from i because $K(i,i,r)$. Let u be the vertex preceding j' in this path. From the definition of j' and u, we see that the following must hold: $K(i,u,r) \wedge K(u,j',1) \wedge \neg K(i,j',r)$; we call this Condition 1.

We now do a case analysis.

Case 1: $parent(u, r-1) \neq i$.

We observe that the set $\{\ k \mid K(i,k,r) \wedge (parent(k,r-1) \neq i)\}$ is nonempty since it contains u. Therefore, $helper(i,r)$ is not null.

Case 2: $parent(u, r-1) = i$.

Since $K(u,j',1)$ and $r > 1$, we know that $K(u,j',r-1)$ from the monotonicity of K. Therefore, $K(i,j',r)$ from the algorithm. This contradicts Condition 1.

■

We now prove that *fast-leader* converges in $O(\log^2 n)$ rounds.

The argument is based on viewing the algorithm as $\log n$ *phases*, each consisting of $\lceil \log n \rceil + 3$ rounds. We will prove that in each phase the number of leaders reduce by at least a factor of 2. This implies that in at most $\log n$ phases the number of leaders will reduce to 1. We will show that one additional phase ensures that the leader knows everybody.

Conceptually, at any round the *parent* function induces a graph on the machines; the edge (u,v) exists in the induced graph exactly when $parent(u,r) = v$. This graph is a forest of trees, and the trees are rooted at leaders, as in Figure 3. In our proof we will analyze how the forest evolves.

It is important to note that the concept of phase is used only for the purpose of proof and is not required in the algorithm. If for a particular family of graphs convergence is achieved in $f(n)$ rounds, then *fast-leader* will terminate in $f(n)$ rounds, even if $f(n) = o(\log^2 n)$. In contrast, *name-dropper* has no mechanism to detect convergence; it can at most provide a bound on the probability of convergence, and that too only when n is *a priori* known to all machines.

From now on we use an additional argument with our functions to indicate the phase. Thus, $leader(i,r,p)$ means that i is a leader in round r of phase p. We do the same for the K and $parent$ functions.

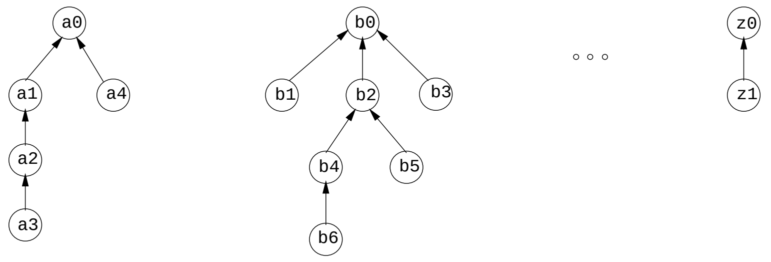

Figure 3: A forest of trees.

Let $pleaders(p)$ denote the set of leaders at the first round of phase p. In a round r of a phase p, we define the *rank* of i that captures the notion of how distant i is from knowing a member of $pleaders(p)$. To define the rank function formally, we first introduce an iterated version of the parent function:

$$\begin{aligned} parent^k(i,r,p) &= parent(parent^{k-1}(i,r,p),r,p) \text{ if } k > 0 \\ parent^0(i,r,p) &= i \text{ if } k = 0 \end{aligned}$$

Now, we define the rank function as:

$$rank(i,r,p) = min\{\ k \mid \exists u \in pleaders(p) : K(parent^k(i,r,p),u,i,p)\}$$

Informally, the rank of a machine is the distance to the least ancestor who knows one of the machines in $pleaders(p)$. Note that the rank of a machine which knows some machine in $pleaders(p)$ is 0. Since the knowledge of a machine can only increase, once the rank of a machine becomes 0, it stays 0 in that phase.

Note that the rank function is well defined: from any machine by following parents we will eventually arrive at a node that knows of a member in $pleaders(p)$. Let h be the maximum rank of any machine in the first round of phase p, i.e.,

$$h = \max\{rank(i,1,p) \mid i\}.$$

It is easy to see that h is most $n-1$. For notational convenience, we define $t = \lceil \log h \rceil$. We now show that the rank of any machine is 0 at the beginning of round $t+1$ in phase p.

Theorem 2 For any machine i and phase p, we have $rank(i,t+1,p) = 0$.

Proof: Let i be any machine. We first show that for any round k,

$$rank(i,k,p) \leq 1 \Rightarrow rank(i,k+1,p) = 0.$$

If the rank of the machine is 0, then it stays 0. If the rank of machine i is 1, then by definition of $rank$, the parent of i in round k knew of a machine $u \in pleaders(p)$. From our algorithm, i will learn about u in phase k. Therefore, its rank will become 0 in the next round.

We now show that

$$rank(i,k,p) > 1 \Rightarrow rank(i,k+1,p) \leq \lceil rank(i,k,p)/2 \rceil$$

Let v be the least ancestor of i which knows of some machine $u \in pleaders(p)$. Define W to be the set of machines along the path (based on the parent pointer) from i to the machine v. We take W to include i but not v. Note that the size of W is equal to the rank of i.

After round k, either all machines in W have their parent pointers in $W \cup \{v\}$ or some machine in W has a parent pointer outside of W. We consider both possibilities:

Case 1 $\forall w \in W : parent(w,k+1,p) \in W \cup \{v\}$.

In this case, for any $w \in W$ such that $parent(w,k,p) \neq v$, we have

$$parent(w,k+1,p) = parent(parent(w,k,p),k,p).$$

This implies $rank(i,k+1,p) \leq \lceil rank(i,k,p)/2 \rceil$. This is the standard pointer jumping technique used in design of parallel algorithms.

Case 2 $\exists w \in W : parent(w,k+1,p) \notin W \cup \{v\}$.

Let y be the first machine on the path from i to v for which $parent(y,k+1,p) \notin W \cup \{v\}$. There are two subcases. Either, y was contacted by some other machine and y changed its parent, or y was a child of v in round k and in round $k+1$ its parent became $parent(v,k,p)$. If y was contacted by some other machine, that machine must be a leader and it follows that $rank(y,k+1,p) = 0$. If y was a child of v, then its rank is 1 and it follows again that $rank(y,k+1,p) = 0$.

Now consider any machine i' on the path from i to y; for each such machine, its parent in round $k+1$ is in $W \cup \{v\}$ (since by hypothesis, y was the first machine such that $parent(y,k+1,p) \notin W \cup \{v\}$). Hence $parent(i',k+1,p) = parent(parent(i',k,p),k,p)$. Analogous to Case 1, the distance of i to y is halved. Since $rank(y,k+1,p) = 0$, it follows that $rank(i,k+1,p) \leq \lceil rank(i,k,p)/2 \rceil$.

By the arguments above, in t rounds, the rank of every machine is at most 1; in one more round, the rank of each machine becomes 0. ■

We now show that the number of leaders in every phase go down by at least a factor of 2. The main argument would show that in a phase every leader x will either become a nonleader or will capture another leader y. It is clear that once a leader is captured, from our algorithm y will continue reporting to x unless it finds a bigger identifier. In that case, from the definition of the helper, we are guaranteed that x will become a nonleader in the next round. Observe that the helper function is designed such that if y used to report to x and then stopped reporting to x, then y or a machine with a similar characteristic is used as a helper.

Theorem 3 If $|pleaders(p)| > 1$, then $|pleaders(p+1)| \leq |pleaders(p)|/2$.

Proof: It is clear that $pleaders(p+1) \subseteq pleaders(p)$. We show that for any machine $u \in pleaders(p+1)$ there exists a machine $x \in pleaders(p) - pleaders(p+1)$, such that $parent(x, t+$

$3, p) = u$. Since a machine cannot have two parents, it follows that the number of leaders is at least halved in each phase.

Consider the round $t+1$ for any machine u in $pleaders(p+1)$. Let v be the helper machine for u. We first claim that v knows a machine x, different from u, which is either bigger than u, or is a member of $pleaders(p)$. Formally,

$$\exists x : K(v, x, t+1, p) \wedge ((u < x) \vee (x \in pleader(p) \wedge (x \neq u))). \tag{1}$$

From Theorem 2, we know that $K(v, w, t+1, p)$ for some $w \in pleaders(p)$. If $w \neq u$, we are done because w discharges the existential requirement. Otherwise, $w = u$. Since $K(v, u, t+1, p)$ and $parent(v, t+1, p) \neq u$ (because v is the helper machine for u), we get that v must know some other machine with a strictly greater identifier. This proves our claim.

From the above claim, and the fact that the neighbor list of v is sent to u in round $t+1$, one of the following cases holds for round $t+2$.

Case 1 $\exists x : K(u, x, t+2, p) \wedge (u < x)$.
This implies $\neg leader(u, t+2, p)$. Therefore $u \notin pleaders(p+1)$.

Case 2 $\exists x : K(u, x, t+2, p) \wedge (x \in pleaders(p) \wedge (x \neq u))$.
If $(u < x)$, Case 1 holds. So now consider the case when $x < u$. In round $t+2$ due to our strategy for choosing helper, either machine u is not a leader or it will contact the machine with the highest *(level, id)*. This implies that u will contact the machine x or a machine which was a leader for at least as many rounds as x. Call this machine y. Clearly, $y \in pleaders(p)$. If $u = parent(y, t+3, p)$, u has managed to capture y and we are done; otherwise, $\exists z : K(u, z, t+3, p) \wedge (u < z)$. This implies that $u \notin pleaders(p+1)$, a contradiction.

Note that the level numbers known to a machine at a particular round are only estimates, and may not be the true level numbers at that round; however, the true level numbers are no less than the estimates. Let α be the total number of rounds which have taken place when phase p begins. Since children of the leader get their estimate updated in the first round of phase p, any machine in the tree of the leader will have its estimate at least equal to α. Thus in Theorem 3 the machine u's estimate of x's level number is at least α. Therefore, the node it contacts will have a level number at least α. ∎

We now have the following result.

Theorem 4 *The Fast Leader Election algorithm terminates in $O(log^2 n)$ rounds with a unique leader.*

Proof: From Theorem 3, the number of leaders get at least halved in every phase. The total number of leaders in the first phase is bounded above by n. Therefore, at most $\log n$ phases are required for the number of leaders to reduce to 1. Each phase is $\log n + 3$; the result follows immediately.

Once there is only one leader, from Theorem 2 we know that in $O(\log n)$ rounds every machine will report to that leader. At that point, the leader will terminate declaring itself to be the global leader based on Theorem 1. ∎

Since a machine initiates at most one message per round it is clear that the message complexity is $O(n \log^2 n)$ and the pointer complexity (the number of pointers communicated) is $O(n^2 \log^2 n)$.

4 Conclusion

In summary we have developed an efficient algorithm for RDP which overcomes key limitations of previously offered solutions. For brevity, we have omitted experimental results; however, simulations on the Georgia Tech Internetwork Topology Models [11] indicate that *name-dropper* takes two to four times as many rounds as *fast-leader* to achieve convergence. More significantly, it was always the case that both achieved convergence in far fewer than $\log^2 n$ rounds; however, *name-dropper* had no way to determine this.

It is relatively straightforward to use the ideas underlying *fast-leader* to develop a convergence detection algorithm, which can run in conjunction with any resource discovery thus providing them with a stopping rule, without any increase in message complexity. Additionally, *fast-leader* can be modified to achieve fast solutions to other problems in distributed computing, such as the *topology discovery* problem. Because of space limitations, we do not present these applications.

One shortcoming of *fast-leader* is that it requires more memory, and more computation by each machine in a round. However, given the low cost of memory, and the relative speeds of computers and networks [9, page 73] this should not significantly affect performance.

Another shortcoming of *fast-leader* is that rounds are assumed to be synchronous. However, the individual transactions can be made asynchronous; we conjecture that even in the asynchronous case, *fast-leader* will perform well.

A more significant limitation of *fast-leader* is the fact that when it is close to convergence, there will be a small set of leaders, and each will communicate with a large number of followers. Consequently, leaders may get swamped. (The possibility of swamping exists with *name-dropper* too, e.g., in a star topology.) There are some obvious optimizations to *fast-leader*, which alleviate this problem. For example, instead of sending the entire neighbor list to the parent in every round, a machine can send only the new information it has learnt in the last round. With this optimization it is clear that no machine can ever receive more than n pointers from any machine. To implement this optimization, it is sufficient for every machine to remember the parent and the message it sent in the last round. More generally, there is a large body of literature on load-balancing, e.g., [5], which can be applied to this problem. We are investigating extensions to *fast-leader* in which leaders can transfer some of their transactions to their followers.

The *fast-leader* algorithm is the simplest algorithm that we were able to prove a poly-logarithmic time bound on; however, we believe that simpler versions of *fast-leader* exist which are also have poly-logarithmic time complexity.

References

[1] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw Hill, 1998.

[2] F. Y. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[3] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource Discovery in Distributed Networks. In *Symposium on Principles of Distributed Computing*, May 1999.

[4] Akamai Inc. `www.akamai.com`.

[5] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hotspots on the World Wide Web. In *Proc. ACM Symposium on the Theory of Computing*, May 1997.

[6] S. Keshav. *An Engineering Approach to Computer Networking* . Addison-Wesley, 1997.

[7] John Moy. RFC 1583. `http://www.dsi.unive.it/Connected/RFC/1583`, 1994.

[8] Y. Shiloach and U. Vishkin. An o(n log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[9] Mark Stemm. *A Network Measurement Architecture for Adaptive Applications.* PhD thesis, The University of California at Berkeley, 1999.

[10] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *ACM Symposium on Operating System Principles*, October 1997.

[11] Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE Infocom*, 1996.