Resource Discovery in Distributed Networks

Mor Harchol-Balter* Tom Leighton† Daniel Lewin‡

Abstract

In large distributed networks of computers, it is often the case that a subset of machines wants to cooperate to perform a task. Before they can do so, these machines need to learn of the existence of each other. In this paper we are interested in distributed algorithms whereby machines in a network learn of other machines in the network by making queries to machines they already know. The algorithms should be efficient both in terms of the time required and in terms of the total network communication required until all machines have discovered all other machines. We propose a very simple algorithm called Name-Dropper whereby all machines learn about each other within $O(\log^2 n)$ rounds with high probability, where n is the number of machines in the network. The total number of connections required is $O(n \log^2 n)$ and the total number of pointers which must be communicated is $O(n^2 \log^2 n)$, with high probability. Each of the preceding bounds is optimal to within polylogarithmic factors.

1 Introduction

In large distributed networks of computers, it is often the case that a subset of machines want to cooperate to perform a common task. For example, machines may cooperate to implement a distributed web caching protocol, to form a distributed file system, or to do some distributed computation. A first step in any of these applications is for the machines to learn about the existence of each other. In other words, machines need to know "Who on the network wants to cooperate with me?" This common first step is what we call the *Resource Discovery Problem.*

Resource discovery algorithms need to be efficient in terms of time and network communication. That is, machines should learn about each other quickly, without using an inordinate amount of communication. This is particularly important in applications where the algorithm may be used repeatedly to obtain updated information about the status of machines in the system.

An instance of the resource discovery problem is modeled as a directed graph. Each machine is represented by a node of the graph and edges represent the relation "machine A knows about machine B." As machines learn about each other, new edges are added to the graph. Communication can only take place between machines that know about each other, in contrast to, say, a "global ping" on the network, that every machine responds to. In terms of the graph, a node u can only communicate with another node v if there is a directed edge from u to v. In this case v is considered to be a *neighbor* or u. We will use the terms "network" and "graph" and the terms "machine" and "node" interchangably.

We are interested in *distributed* resource discovery algorithms, where there is no central control in the network and machines operate independently of each other, making local queries to their neighbors and transferring information about part or all of their neighbor lists. Figure 1 gives an example of a connection between adjacent nodes where one node, A, makes a connection with node B and is sent B's entire neighbor list, causing A's neighbor list to increase by two nodes.

We model resource discovery algorithms as proceeding

*Email: harchol@theory.lcs.mit.edu. MIT Laboratory for Computer Science. Supported by the NSF Postdoctoral Fellowship in the Mathematical Sciences.

†Email: ftl@math.mit.edu. Department of Mathematics and Laboratory for Computer Science. Supported by DARPA N00014-95-1-1246 and ARMY DAAH 04-95-1-0607

‡Email: danl@theory.lcs.mit.edu. Laboratory for Computer Science. Supported by DARPA N00014-95-1-1246 and ARMY DAAH 04-95-1-0607

in synchronous parallel rounds. We define one *round* as the time for each machine in the network to contact one or more of its neighbors and exchange some subset of its neighbor list. The running time of a resource discovery algorithm is the number of rounds required until every machine knows about every other machine, i.e., a complete graph is formed. Observe that different rounds can have different running times. One difference is caused by the fact that the "neighbors" of a node may be of different distances from the node in the underlying physical network. However, it can also turn out that communicating between two nodes which are physically close may be more costly than communicating between two nodes which are far apart because of differences in the speeds of the routers and speeds of the machines. Thus we simply use the number of rounds as our run time metric. In practice there is enough time between rounds that each round is able to complete.

Another performance measure of resource discovery algorithms is the amount of network communication they require. We measure network communication in two ways: The *pointer communication complexity* is defined to be the total number of pointers communicated during the course of the algorithm. The *connection communication complexity* is defined to be the total number of connections which are opened during the course of the algorithm, where a connection between u and v is created when u contacts v. During the connection, u is allowed to transmit as much information as it likes to v. The operation shown in Figure 1 for example has connection communication complexity of 1 and pointer communication complexity of two, since one connection is openned and two pointers are transferred. As another example, suppose we have a d-regular graph and each node picks one of its neighbors and sends it its entire neighbor list (including itself). Then the total pointer complexity for that round is $n(d+1)$, whereas the communication complexity for the round is n. We consider both the pointer communication complexity and the connection communication complexity because, in practice, network communication is a linear combination of the two.

Our only assumption about the network is that it is initially *weakly connected*. That is, if we ignore edge directions then the graph is connected. In practice, this boils down to giving every newly added machine a pointer to at least one machine in the network. Weak connectivity is a necessary assumption because otherwise we allow disconnected networks where there is no hope of ever evolving into one component.

Our goal is to design a resource discovery algorithm which requires few rounds and requires low network communication. For practical reasons it is more important that the algorithm be *very simple* than that it achieve absolutely optimal performance. In this paper we propose a randomized resource discovery algorithm with the following properties:

- Our algorithm is distributed and each machine executes the same simple local protocol.
- Machines learn about each other quickly: If there are n machines, then with high probability within $O(\log^2 n)$ rounds, all machines know about each other.
- Our algorithm does not flood the network with communication. In particular the connection communication complexity is $O(n \log^2 n)$ and the pointer communication complexity is $O(n^2 \log^2 n)$, with high probability.

Each of the preceding bounds is optimal to within polylogarithmic factors. [2]

1.1 Some Candidate Algorithms

Before describing our algorithm, we first describe a few natural algorithms for resource discovery, some of which are used in real systems. Our algorithm uses ideas from each of these. The performance of these algorithms is summarized in Table 1.

1.1.1 The Flooding Algorithm

The Flooding algorithm is used by Internet routers today [8].

In the *Flooding algorithm*, a machine is initially configured to have a fixed set of neighboring machines, and direct communication is only allowed with machines in this set. In terms of the graph, a node only communicates over the edges that were *initially* in the graph; new edges that are added to the graph are not used for communication. Observe that those edges that constitute the "initial neighbors" are not necessarily the links in the underlying physical network, but rather they are virtual links, each possibly corresponding to a path in the underlying network.

We denote by $\Gamma(v)$ the set consisting of v and of all the nodes that v points to. In every round of the Flooding algorithm, each node v contacts all of its *initial* neighbors and transmits to them the updates to $\Gamma(v)$, (denoted by $\Gamma(v)^{Updates}$), i.e., those nodes in $\Gamma(v)$ that are new since the last time v sent information. A node u that receives $\Gamma(v)^{Updates}$ then updates its set of neighbors by merging $\Gamma(u)$ and $\Gamma(v)^{Updates}$, ($\Gamma(u) \longleftarrow \Gamma(u) \cup \Gamma(v)^{Updates}$).

The number of rounds required for the Flooding algorithm to converge to a complete graph is equal to the diameter, $d_{initial}$, of the *initial graph*. If $d_{initial}$ is small then the algorithm is fast; however $d_{initial}$ could be large: $\Theta(n)$.

[2]Following this paper, a deterministic algorithm has been proposed for the problem with running time $O(\log n \log^* n)$, [11].

Thus the Flooding algorithm could be very slow if we are not careful to start with an initial graph that has small diameter.

The communication complexity of the Flooding algorithm also depends on the initial graph. Let $m_{initial}$ be the number of edges in the initial graph. The pointer communication complexity of the Flooding algorithm is $\Theta(n \cdot m_{initial})$ because every pointer must be sent over every edge during the algorithm. The connection communication complexity of the Flooding algorithm is $\Theta(d_{initial} \cdot m_{initial})$. The above bound can be obtained by the following argument: If the diameter is $d_{initial}$, then every point s is at least $d/2$ distance away from some other point, which means that there is a shortest path to s of length at least $d/2$. Thus s learns new pointer information in one of the next $\geq d/2$ rounds. Thus s must open up a connection with *all* of its initial neighbors during each of the new rounds, so that it can communicate this new information to them. Hence every one of the $m_{initial}$ edges is used during at least the next $d/2$ rounds. Thus the lower bound is $\Omega(d_{initial} \cdot m_{initial})$. Observe that this is also an upper bound since there are $d_{initial}$ rounds, and during each round at most $m_{initial}$ connections can be open.

The bottom line is that the network complexity depends on both $d_{initial}$ and $m_{initial}$, and often at least one of these will be high. Observe that $m_{initial}$ is always $\geq n$, since the inital graph is weakly connected.

1.1.2 The Swamping Algorithm

As we mentioned above, the Flooding algorithm is used by Internet routers today. However the Internet routers are in fact designed with the capability of opening connections to any machine they know about, not just machines in the "initial set." Although this capability is not currently being used in the context of the Internet, it is available in case future algorithms require it.

The *Swamping algorithm* is identical to the Flooding algorithm except that machines may now open connections with *all* their current neighbors, not just their *initial* neighbors. Also since the neighbor sets change, *all* of the current neighbor set is transfered, not just the updates. That is, a machine v now sends $\Gamma(v)$ to *every machine* in $\Gamma(v)$, instead of only sending to its initial set of neighbors.

The advantage of the Swamping algorithm is that the graph always converges to a complete graph in $O(\log n)$ steps, irrespective of the initial configuration.

The disadvantage of the Swamping algorithm is that the network communication complexity is increased. The pointer communication complexity of the Swamping algorithm is $\Omega(n^3)$, since during the last round, when the graph is almost complete, each of the n machines sends each of its n pointers to each of its n neighbors. The connection communication complexity of the Swamping algorithm is $\Omega(n^2)$ since during the last round, when the graph is almost complete, each of n machines makes connections with each of its n neighbors.

The bottom line is that the Swamping algorithm is very fast (only $O(\log n)$ rounds), but this speed is obtained at the cost of wasted communication where many machines are being told of machines they already know about.

1.1.3 The Random Pointer Jump Algorithm

The disadvantage of the Swamping algorithm is that the communication complexity grows quickly. To reduce the communication complexity one might consider having each machine communicate with only *one* randomly-chosen neighbor during each round.

The *Random Pointer Jump Algorithm* works as follows: In each round, each machine v contacts a random neighbor $u \in \Gamma(v)$. The chosen neighbor u then sends $\Gamma(u)$ to v, who then merges $\Gamma(u)$ with $\Gamma(v)$. An example of the Random Pointer Jump operation is given in Figure 1. Note that this operation corresponds to the classical pointer jump operation, commonly used in parallel algorithm design, see [7].

The Random Pointer Jump Algorithm can only be applied to strongly connected networks (i.e., there must exist a path between every pair of machines), because otherwise the graph will never converge to a complete graph. Consider for example the graph with two nodes and a single directed edge between them: the remaining edge cannot be formed.

Given that the graph is strongly connected, the Random Pointer Jump might seem like a good idea. For example a ring of n nodes converges to a complete graph in $O(\log n)$ rounds with high probability and the total connection communication complexity is only $O(n \log n)$, since there are $\log n$ rounds during which each of n machines opens up one connection.

Interestingly, it turns out that this algorithm is *not* a good choice – even for strongly connected graphs. Figure 3 gives an example of a graph that is strongly connected and requires, with high probability $\Theta(n)$ rounds to converge to a complete graph.

Claim 1 *With high probability, the graph in figure 3 requires $\Omega(n)$ time to converge to a complete graph using the random jump local query.*

Proof: We only include a sketch here. Consider the time it takes for the outside ring to shrink to a clique. The problem is that at every round there are only a small number, $O(1)$, of ring nodes that choose their successor in the ring to run the protocol with (each ring node only has probability $O(1/n)$

	Num. Rounds	Pointer Communication	Connection Communication
Flooding	$d_{initial}$	$\Omega(n \cdot m_{initial})$	$\Omega(d_{initial} \cdot m_{initial})$
Swamping	$O(\log n)$	$\Omega(n^3)$	$\Omega(n^2)$
Random Pointer Jump	$\Omega(n)$ in worst case	(num. rounds)$\cdot m_{initial}$	(num. rounds)$\cdot n$
Name-Dropper	$O(\log^2 n)$	$O(n^2 \log^2 n)$	$O(n \log^2 n)$

Table 1: *This table shows the performance of 3 natural resource discovery algorithms, as compared with our own resource discovery algorithm, Name-Dropper. In the above notation: $m_{initial}$ is the number of edges in the initial graph, n is the number of nodes in the graph, and $d_{initial}$ is the initial diameter of the graph. The Name-Dropper algorithm outperforms the other 3 algorithms in terms of* worst-case *communication complexity by a factor of $\Omega(n/log^2 n)$, while the runtime of the Name-Dropper algorithm is close to optimal.*

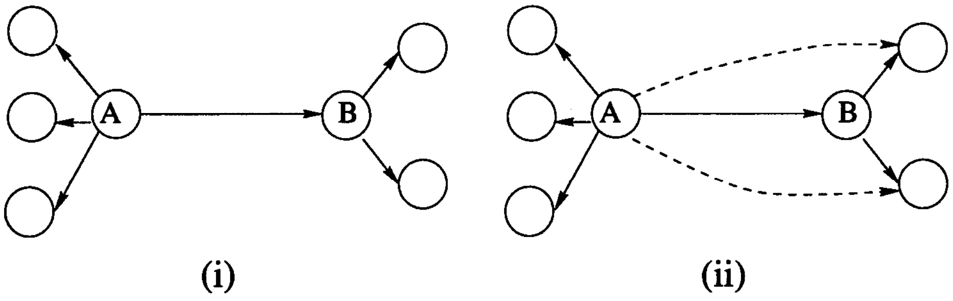

Figure 1: (i) Before the Random Pointer Jump. Node A chooses at random one of its neighbors and opens a connection with it. Here the chosen neighbor is labeled B. (ii) After the Random Pointer Jump. Node B has passed to node A all of its neighbors, and now A also points to them. The newly formed edges are shown with dashed lines.

of choosing its neighbor in the ring). Thus, in expectation, pointer jumping within the ring alone will only shrink the ring by a constant number of nodes at each round, thus requiring $\Theta(n)$ rounds to shrink the ring.

Unfortunately, pointer jumping from within the central clique is not of much helpful use. The single edge pointing out of the center clique is replicated to all of the nodes in the clique in $O(\log n)$ time. At this point, this edge is chosen by one of the center nodes, and the single ring node pointed to by this edge effectively becomes part of the central clique. At this point, we are in a very similar situation to what we started with. Again we have a central clique and a ring (with one fewer nodes), where the central clique has one edge pointing out to the ring. This process repeats, but every $O(\log n)$ steps, the ring will only shrink by $O(1)$ nodes due to pointer jumping from the central clique. ∎

1.1.4 Random Pointer Jump with Back Edge

Consider what went wrong with the Random Pointer Jump algorithm of the previous section. The problem was that the central clique of Figure 3 was very slow to accumulate pointers out to the outside ring, and therefore we couldn't exercise the power of that central clique to quickly shrink the ring. A natural idea for improving the Random Pointer Jump algorithm of the previous section is to add a *back edge* every time a pointer jump is performed. Specifically, when node A chooses node B and node B passes to A all of its neighbors, node B also obtains a pointer back to A, as shown in Figure 2. This would cause the central clique in Figure 3 to almost immediately obtain pointers to all points on the outer ring. We call the new algorithm *Random Pointer Jump with Back Edge.*

We believe that Random Pointer Jump with Back Edge will have very good performance, however we have not been able to prove good bounds on the time for this algorithm to converge. The natural types of argument which one would like to make involve trying to show that every $u \longrightarrow v \longrightarrow w$ path gets jumped every $O(\log n)$ steps so that the diameter of the graph is cut by a factor of two every $O(\log n)$ steps. However when one tries to make this type of argument about Random Pointer Jump with Back Edge, one runs into the problem of not being able to ensure that a particular pointer will be jumped, i.e. a particular edge will be chosen. The difficulty is that a node may have many neighbors, i.e. many distractors, keeping it from choosing the particular edge that we would like it to choose. For this reason, we have instead chosen to study an algorithm which is very similar to Ran-

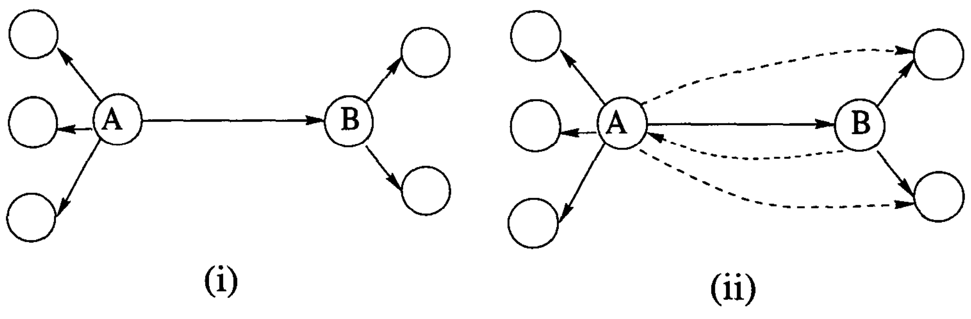

Figure 2: (i) Before the Random Pointer Jump with Back Edge. Node A chooses at random one of its neighbors and opens a connection with it. Here the chosen neighbor is labeled B. (ii) After the Random Pointer Jump. Node B has passed to node A all of its neighbors, and now A also points to them. In addition node B is also given a pointer to node A. The newly formed edges are shown with dashed lines.

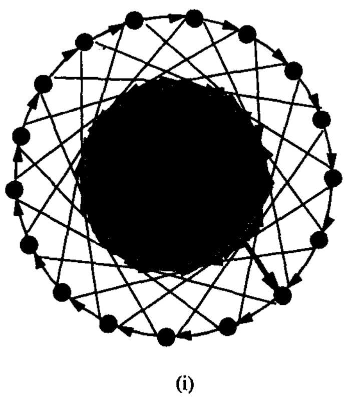

Figure 3: (i) A strongly connected graph with n nodes that takes time $\Omega(n)$ to converge to a complete graph using the random pointer jump local query. The "center" of the graph is a complete graph on $n/2$ nodes. Each node in the ring around the center is connected to *every* node in the central clique. There is a single node in the clique that points out to a single node on the ring - therefore making the graph strongly connected.

dom Pointer Jump with Back Edge, but where the direction of information transfer is *opposite* to Random Pointer Jump with Back Edge. We will describe this algorithm, which we call Name-Dropper, in the next section and see that it is in fact surprisingly easy to reason about.

1.2 Our Algorithm – Name-Dropper

The *Name-Dropper* algorithm looks very similar to the Random Pointer Jump with Back Edge algorithm presented in the previous section.

The Name-Dropper algorithm works as follows: During each round, each machine v transmits $\Gamma(v)$ to *one, randomly chosen* neighbor. A machine u that receives $\Gamma(v)$ merges $\Gamma(v)$ with $\Gamma(u)$ as in the previous algorithms. Figure 4 illustrates one connection in the Name-Dropper algorithm.[3]

In Section 2 we will prove that Name-Dropper terminates in $O(\log^2 n)$ rounds with high probability whereas the Random Pointer Jump algorithm can take $\Omega(n)$ rounds. As a consequence, we will also conclude that the network communication complexity of Name-Dropper is very low: The connection communication complexity is $O(n \log^2 n)$ and the pointer communication complexity is $O(n^2 \log^2 n)$. All these bounds are within polylogarithmic factors of optimal.

The Name-Dropper algorithm has been implemented at the Laboratory of Computer Science at MIT as part of a project to build a large-scale distributed cache. The project enables certain machines on the Internet to cooperate in caching information. In order for these machines to cooperate they must first locate each other – this is where the Name-Dropper resource discovery algorithm is used. The Name-Dropper algorithm has been licensed to Akamai Technologies, which is building an Internet-wide content-distribution system.

[3]Name-Dropper derives its name from the following social behavior commonly called "name dropping." A newcomer approaches a group of people and introduces himself. During the ensuing conversation, the newcomer inserts into the conversation all the names of the people he knows, usually in the belief that he will profit from his associations, however this aspect of the social behavior is irrelevant to our algorithm.

1.3 Previous Work

The Flooding algorithm which we describe in Section 1.1.1 is used today by routers on the Internet, see the Internet Request For Comments number 1583 [8].

Communication and broadcast by local queries has been extensively studied under the name "gossiping" [6, 4, 2, 10]. The "classical" gossiping problem assumes that either all machines know about each other, or that there is a fixed communication network. Gossiping is used to broadcast information from every machine to every machine. In [6], a survey of general lower bounds on the number of connections that need to be made to broadcast information from every machine to every other machine can be found. In addition, [6] describes tight upper bounds for gossiping on fixed specific communication networks.

In contrast to the classical gossiping problem, we address the problem of broadcasting in networks where machines may *not* initially know about each other. Our results show that if we allow machines to learn about other machines during the gossip process, then gossiping can be done efficiently starting from any weakly connected graph.

Gossip type algorithms have been used in various practical distributed systems and algorithms [1, 3, 5, 10, 9, 12]. For example, in [1, 3], gossiping is used to maintain consistency in a distributed replicated database. Recently [12], gossiping has been used to gather information about failures in a network of machines. Most of these gossiping algorithms assume that all the machines on the network are already aware of each other, and that information needs to be broadcast from one or more of the machine to the others. This broadcast is carried out by an algorithm similar to the Name-Dropper: choose a random neighbor and tell him your information.

In [12], it is assumed for simplicity that only a single machine gossips at every round, and the authors give empirical evidence that information is propagated to all machines in linear time. Our analysis proves that even if machines are not aware of each other at the start, after $O(\log^2 n)$ *parallel* rounds of gossiping, information has propagated to the whole network.

2 Performance Analysis of Name-Dropper

The main theorem in this paper is as follows:

Theorem 2 *Let G be any weakly connected directed graph on n vertices. Then, after $O(\log^2 n)$ rounds of the Name-Dropper algorithm, the graph evolves into a complete graph with probability greater than $1 - \frac{1}{n^{O(1)}}$.*

Observe that there is a lower bound of $\log n$ steps for any resource discovery algorithm since the diameter of the graph can at most halve with every round. We conjecture that this lower bound can be achieved with high probability using the Name-Dropper algorithm; however we have not been able to find a proof nor a counterexample to this conjecture.

As a corollary to Theorem 2, we obtain an upper bound on the network communication complexity required by the Name-Dropper algorithm.

Corollary 1 *The connection communication complexity of Name-Dropper is $O(n \log^2 n)$ with probability greater than $1 - \frac{1}{n^{O(1)}}$. The pointer communication complexity of the Name-Dropper algorithm is $O(n^2 \log^2 n)$ with probability greater than $1 - \frac{1}{n^{O(1)}}$.*

Proof: With respect to connection communication complexity, during each round, each of n machines only makes one connection, and there are $O(\log^2 n)$ rounds.

With respect to pointer communication complexity, each round may require each machine to transfer $O(n)$ pointers. Thus the pointer communication complexity is upper bounded by n^2 times the number of rounds. ∎

The rest of this section is devoted to proving Theorem 2. We begin with an overview of the proof.

We break the evolution of the graph into *stages* that are each $O(\log n)$ rounds long. We then show that, with high probability, the distance between every two nodes (measured after undirecting the edges) goes down by a constant factor every stage. Therefore, after $O(\log n)$ stages, the graph is complete since the distance between every two nodes is one with high probability.

The main step in showing that a stage is successful (namely that the distance between every two nodes goes down by a constant factor) is to show that every node on a *shortest path* makes a "pointer jump" with high probability. More specifically, if $v \longrightarrow u \longrightarrow w$ is a subsequence of a shortest path, then we show that after $O(\log n)$ rounds, v has an edge to w with high probability.

In fact,we'll show that *every* triple $v \longrightarrow u \longrightarrow w$ has a very good chance of making a pointer jump within $O(\log n)$ rounds. Then we're done by the following argument: Pick one shortest path between each of the n^2 pairs of nodes. Each of these shortest paths has at most n triples. Thus we have n^3 triples which we care about, and each of these makes a pointer jump with probability $1 - \frac{1}{n^c}$, where c is a constant. Assuming c is high enough (in this case ≥ 4), then all the n^3 triples will make a pointer jump with probability at least $1 - \frac{1}{n^1}$.

We will use the above type of union-bound argument repeatedly in our analysis. To avoid extensive notation we will always denote the above constant by c, however it is impor-

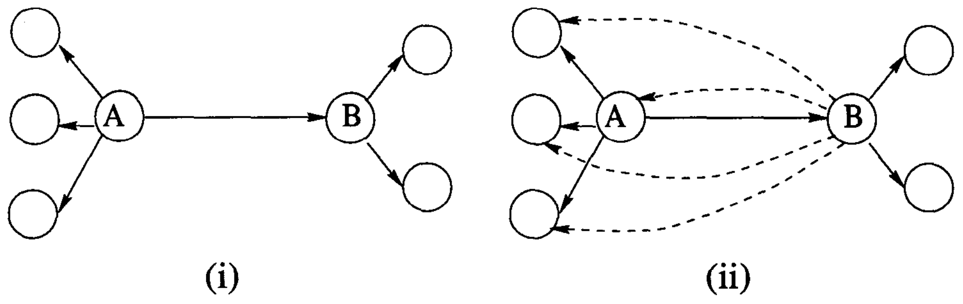

Figure 4: (i) Before Name-Dropper. Node A chooses a random neighbor, here the neighbor is labeled by B. (ii) After one round of the Name-Dropper algorithm. A has passed to B all of its neighbors and B has added edges to these neighbors. In addition, B learns about A. The edges added to the graph are dashed.

tant to realize that each time we use this type of argument the constant c needs to be chosen appropriately.

We now prove Theorem 2:

Let $G = (V, E)$ be the graph. We abuse notation and always denote the edge set of the graph by E, although it may increase at every step. A directed edge from u to v is denoted by (u, v). The following lemma is the main step in proving Theorem 2.

Lemma 2.1 *Given any $u, v, w \in V$ such that $(w, u) \in E$ and $(w, v) \in E$. Then for any constant c, with probability greater than $1 - \frac{1}{n^{O(c)}}$ after $O(c \log n)$ steps of the Name-Dropper algorithm we have $(u, v) \in E$ and $(v, u) \in E$.*

Figure 5 shows the setup of Lemma 2.1.

Figure 5: *(i) The setup of the nodes u, v, and w in Lemma 2.1. (ii) The configuration guaranteed to occur with high probability by Lemma 2.1 after $O(\log n)$ steps of the Name-Dropper algorithm.*

For now we simply assume Lemma 2.1 to be true and continue with our proof of Theorem 2. Our original graph $G = (V, E)$ is weakly connected. As a first step, we will prove that the graph will become strongly connected after $O(\log n)$ steps of the name dropper algorithm. In fact we will prove something stronger, namely that for every edge in E, an edge in the opposite direction will be created.

Lemma 2.2 *If $(u, v) \in E$, then with probability at least $1 - \frac{1}{n^{O(c)}}$, after $O(c \log n)$ steps the edge $(v, u) \in E$.*

Proof: We will apply Lemma 2.1. Suppose there exists a node $w \in V$ such that $(w, u) \in E$ and $(w, v) \in E$. In this case, Lemma 2.1 immediately applies and we are done. Now suppose there does not exist such a node w. In this case, either u contacts v directly in the next round, forming the edge (v, u), or else u contacts some other node w in the next round. In the latter case, w will now get a pointer back to u and a pointer to all of u's neighbors, including v. At this point, w will satisfy the condition for Lemma 2.1. ∎

So by running Name-Dropper for $O(c \log n)$ steps, the graph becomes strongly connected with probability at least $1 - \frac{1}{n^{O(c)}}$. After this first stage, we divide the evolution of the graph into stages consisting of $O(\log n)$ rounds each. At the beginning of each stage we measure the distance between each pair of nodes (note that since the graph is already strongly connected, there is a path between each pair of nodes). Our goal is to show that after $O(c \log n)$ rounds, or one stage, the distance between each pair of nodes is at most half of what it was at the beginning of the stage, with probability $1 - \frac{1}{n^{O(c)}}$. We will focus on one particular pair of nodes u and v and show that the above statement is true for that pair. Then, since there are only $O(n^2)$ paths, we can use a union bound to show that all the paths shrink by the desired amount with probability greater than $1 - \frac{1}{n^{O(c)}}$.

Let $u \longrightarrow w_1 \longrightarrow w_2 \longrightarrow \ldots \longrightarrow v$ be the shortest path between u and v at the start of a stage. We show that with probability greater than $1 - \frac{1}{n^{O(c)}}$ every node on the path executes a "pointer jump" after $O(c \log n)$ steps, for every constant c. We focus on one particular node, s, in the path, and prove that it does a pointer jump with probability at least $1 - \frac{1}{n^{O(c)}}$ after $O(c \log n)$ rounds. Since there are at most n nodes in the path, a union bound suffices to show the result.

Let $s \longrightarrow t \longrightarrow z$ be three nodes in the path. We first invoke Lemma 2.2 to show that after $O(c \log n)$ rounds, with probability greater than $1 - \frac{1}{n^{O(c)}}$, the edge (t, s) is added to the graph. Now we have exactly the configuration of Lemma 2.1, and thus the edge (s, z) is added to the graph with probability greater than $1 - \frac{1}{n^{O(c)}}$ after $O(c \log n)$ steps.

Since there can be at most $O(\log n)$ stages where distances go down by a constant factor, the graph must be complete with probability greater than $1 - \frac{1}{n^{O(1)}}$ in $O(\log^2 n)$ steps.

The only thing that remains is proving Lemma 2.1.

Proof:[Proof of Lemma 2.1] Let A denote the set of nodes that have edges to both nodes u and v. The set A is not empty by the assumption of the lemma, $w \in A$. By symmetry it suffices to focus on one of the edges (u, v), (v, u), say (u, v). The overview of the proof is as follows: We will show that during every round one of two things are true: *Either*

1. The probability that the edge (u, v) is formed is at least some constant, *or*

2. The probability that the set A grows by some constant factor is at least some constant.

This suffices to prove the lemma since (2) can happen at most $O(\log n)$ times and if (1) happens $O(c \log n)$ times then the probability that the edge (u, v) is formed is at least $1 - \frac{1}{n^{O(c)}}$.

Observe that the only way that the edge (u, v) can be formed is if some node in A contacts u. Let d_i denote the degree of node i in A. Then,

$$\begin{aligned}
& \mathbf{Pr}\{\text{edge } (u, v) \text{ is formed}\} \\
= \; & 1 - \mathbf{Pr}\{\text{edge } (u, v) \text{ is not formed}\} \\
= \; & 1 - \Pi_{i \in A} \left(1 - \frac{1}{d_i}\right) \\
\geq \; & 1 - e^{-\sum_{i \in A} \frac{1}{d_i}}
\end{aligned}$$

So, if the probability that (u, v) is formed is less than $1 - e^{-\frac{1}{4}}$, then:

$$\sum_{i \in A} \frac{1}{d_i} \leq \frac{1}{4}$$

This in turn implies that at least half of the nodes in A have degree greater than $2|A|$. We now show this last statement in turn implies that the size of A will increase by a constant factor with constant probability at the next step. In particular we show that at least $\frac{|A|}{16}$ nodes that were *not* in A are contacted by a node in A with probability at least $\frac{1}{15}$.

Denote by the "special set" the set of $\frac{|A|}{2}$ nodes in A that have degree at least $2|A|$. Each of the nodes in the special set point to at least $|A|$ nodes that are not in A. Thus, each node in the special set contacts a node not in A that no other node in the special set contacts with probability at least $\frac{1}{4}$. Thus, the *expected* number of new nodes in A is at least $\frac{|A|}{2} \cdot \frac{1}{4} = \frac{|A|}{8}$.

We now use a "bounded Markov argument" to show that the probability that the number of new nodes in A is less than $\frac{|A|}{16}$ is less than $\frac{14}{15}$.

Bounded Markov Argument Let X be a random variable that is bounded from below by 0 and from above by U. That is, $0 \leq X \leq U$. Then for $t \leq U$,

$$\mathbf{Pr}\{X \leq t\} \leq \frac{U - \mathbf{E}\{X\}}{U - t}.$$

Thus by the bounded Markov argument we see that

$$\mathbf{Pr}\left\{\text{Number of new nodes in } A < \tfrac{|A|}{16}\right\} \leq \frac{|A| - \frac{|A|}{8}}{|A| - \frac{|A|}{16}} = \frac{14}{15}.$$

We have shown that at every round either there is a constant probability that the edge (u, v) is formed or a constant probability that the set A grows by a constant factor. A simple Chernoff argument will now show that if an event has a constant probability at every step of occurring and there is independence between the steps then after $O(c \log n)$ steps the event will happen $O(\log n)$ times with probability greater than $1 - \frac{1}{n^{O(c)}}$.

This concludes the proof of Lemma 2.1 and thus the proof of Theorem 2 as well. ■

3 Conclusion and Future Work

In this paper we consider the problem of resource discovery in a distributed network and propose several natural and simple distributed algorithms to solve the problem. All of our algorithms involve machines making local queries to one or more neighboring machines, whereby a machine transfers its neighbor list, or part thereof to a neighboring machine. Our analysis shows that the Name-Dropper algorithm achieves near-optimal performance both with respect to time complexity and with respect to the network communication complexity.

One thing that makes this result peculiar is that the Name-Dropper algorithm is almost identical to the Random Pointer Jump algorithm, however the worst-case performance of the Random Pointer Jump algorithm is very poor ($\Omega(n)$ rounds are required by Random Pointer Jump as compared with $O(\log^2 n)$ rounds for Name-Dropper). In both the Name-Dropper and the Random Pointer Jump algorithms, during each round, each machine a chooses *one, random* machine b from its neighbor list at random. In the case of Name-Dropper, a sends to b a pointer to each machine on a's neighbor list (which includes itself). In the case of Random Pointer Jump, b sends to a a pointer to each machine on b's neighbor list.

As mentioned earlier, the Name-Dropper algorithm is currently being implemented within the Laboratory of Computer Science at the Massachusetts Institute of Technology as part of a project to build a large-scale distributed cache. The idea is to have certain machines on the Internet cooperate in caching information. In order for these machines to cooperate they must first locate each other, which is where the Name-Dropper resource discovery algorithm is used. The Name-Dropper algorithm has been licensed to an LCS startup company, Akamai Technologies, which is building an Internet-wide content-distribution system.

There are many issues which we have not explored in this paper. For one thing, our analysis has assumed that the network is static, i.e., that there are no machines being added or removed while the algorithm is running. In reality, a machine can unexpectedly crash and be brought back online, can change location, or can become temporarily unavailable. Such circumstances are likely to occur in large systems with no centralized control, and resource discovery algorithms should be capable of operating in such environments. Currently the time for one or more new machines to be fully incorporated into the network is the running time of the algorithm, but that assumes that no new machines are added to the network during the time the algorithm is running. We have some preliminary results for the running time of the algorithm in the presence of ongoing births. However we are finding the presence of deaths more difficult to ánalyze.

A second, somewhat related problem, is the question of how the cooperating machines know that they have all discovered each other (and can thus stop running the algorithm). In particular, there might be more machines out there which just haven not been discovered yet. Unfortunately, knowing when to stop depends on first knowing the number of machines which are out there, which is a related problem of interest proposed by Lipton [Personal communication, 1998].

Thirdly, one weakness of our current model is that it allows for the possibility that in some given round, many machines in the network might all choose to contact the same one particular host. In reality that host could only maintain a small number of simultaneous connections and would have to deny access to all the other machines trying to contact it. Even under those more stringent circumstances, however, we conjecture that the Name-Dropper algorithm would still converge to a complete graph in $O(\log^2 n)$ rounds.

Finally, an interesting open problem is whether the running time of the Name-Dropper algorithm can be shown to be $\Theta(\log n)$ rounds, or whether there exists some other equally simple algorithm which only requires $\Theta(\log n)$ rounds.

References

[1] Divyakant Agrawal, Amr El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–172, Tucson, Arizona, 12–15 May 1997.

[2] Susan Assmann and Daniel Kleitman. The number of rounds needed to exchange information within a graph. *SIAM Discrete Applied Math*, 6:117–125, 1983.

[3] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, British Columbia, Canada, 10–12 August 1987.

[4] S. Even and B. Monien. On the number of rounds necessary to disseminate information. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 261–266, 1989.

[5] Mark Hayden and Kenneth Birman. Probabilistic broadcast. *Cornell CS Technical Report TR96-1606*, 1998.

[6] Sandra Hedetniemi, Stephen Hedetniemi, and Arthur Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[7] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes.* Morgan Kaufmann Publishers, 1992.

[8] John Moy. Ospf version 2, rfc 1583. *Available from* `http://www.dsi.unive.it/Connected/RFC/1583/index.html`, 1994.

[9] Derek C. Oppen and Yogen K. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, July 1983.

[10] Andrzej Pelc. Fault-tolerant broadcasting and gossiping in communication. *Networks*, 28(3):143–156, October 1996.

[11] D. Peleg. Deterministic distributed resource discovery. *Unpublished manuscript*, January 29, 1999.

[12] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detector. 1998.