Asynchronous Resource Discovery in Peer to Peer Networks[1]

Shay Kutten*
Faculty of Industrial Eng. & Management
The Technion
Haifa 32000, Israel
kutten@ie.technion.ac.il

David Peleg†
Dept. of Computer Science and Applied Math.
The Weizmann Institute
Rehovot 76100, Israel
peleg@wisdom.weizmann.ac.il

Abstract

The resource discovery *problem arises in the context of* peer to peer (P2P) *networks, where at any point of time a peer may be placed at or removed from any location over a general purpose network (e.g., an Internet site). A vertex (peer) can communicate with another vertex directly if and only if it knows a certain routing information to that other vertex. Hence, a critical task is for the peers to convey this routing information to each other.*

The problem was formalized by Harchol-Balter, Leighton and Lewin [13]. The routing information needed for a vertex to reach another peer is that peer's identifier (e.g., IP address). A logical directed edge represents the fact that the peer at the tail of the edge knows the IP address of the one at its head. A number of algorithms were developed in [13] for this problem in the model of a synchronous *network over a weakly connected directed graph. The best of these algorithms was randomized. Subsequently, a deterministic algorithm for the problem on synchronous networks with improved complexity was presented in [15].*

The current paper extends the deterministic algorithm of [15] to the environment of asynchronous *networks, maintaining similar complexities (translated to the asynchronous model). These are lower than the complexities that would be needed to synchronize the system. The main technical difficulty in a directed, weakly connected system is to ensure that vertices take consistent steps, even if their knowledge about each other is not symmetric, and even if there is no timeout mechanism (which does exist in synchronous systems) to assist in that. (In particular, as opposed to the case in synchronous systems, here an algorithm* cannot *first transforming every directed edge to be bidirectional and second, apply an algorithm for bidirectional graph.) Thus our result takes another step towards representing the actual setting in a realistic manner.*

Keywords: *Peer to Peer, P2P, Topology changes, Topology knowledge loss, Distributed algorithms, Asynchronous networks.*

1 Introduction

The *resource discovery* problem arises in the context of *peer-to-peer (P2P)* networks, where at any point in time, a new peer may be placed at any location over a general-purpose network (e.g., an Internet site). Similarly, peers may be removed. A vertex (peer) can communicate with another vertex directly if and only if it knows a certain routing information to that other vertex. Hence, a critical task is for the peers to convey this routing information to each other.

The problem was formalized by Harchol-Balter, Leighton and Lewin in [13]. The specific P2P application they had in mind was the logical networks of servers placed by Akamai Technologies in various Internet sites. The routing information needed for a vertex to reach another peer is that peer's identifier (e.g., IP address). A logical directed edge represents the fact that the peer at the tail of the edge knows the IP address of the one at its head.

Recently, many other distributed peer-to-peer applications were introduced that require the same task. For example, in Gnutella [5] users on the Internet wish to share files with each other, without using a central server. Each user may tell its local Gnutella process (which is both a client and a server) the Internet addresses of some other client/server processes. Those may know other such processes, and so forth. There is no way for any process to know all the other processes in advance, since users appear and disappear without consulting any central server. Thus, to be able to share and swap files, the client-server

*Supported in part by a grant from the Israel Ministry of Science and Art and by the Technion Fund for the Promotion of Research.

†Supported in part by grants from the Israel Science Foundation and the Israel Ministry of Science and Art.

[1] May this paper serve as a modest tribute to the memory of Daniel Lewin, a co-author of a previous paper on the subject, who died tragically on September 11th, 2001.

processes need to cooperate to locate each other. A very partial list of papers concerning related P2P applications includes [6, 7, 8, 9]. A system of intelligent agents (in the context of Artificial Intelligence), by the name Retsina, was developed at CMU, and its description can be found in [10]. In that system, the agents need to locate each other. A general platform for P2P applications (together with distributed location protocols) is being developed in a project led by SUN Microsystems [11]. Since the actual set of P2P-related projects grows very fast, it is hard to list them here. A more up to date partial listing may be found in [12].

We follow the model of [13, 15, 14], except for the assumption of synchronous communication. That is, the system consists of a set V of n vertices, where each $v \in V$ has a distinct $O(\log n)$ bit identifier $ID(v)$ representing its address. The vertices are logically connected via a directed graph $G(V, E)$, where a directed arc $\langle u, v\rangle$ represents the fact that u knows $ID(v)$ (meaning, intuitively, that it knows v's IP address). A vertex u can send a message to another vertex v if and only if the arc $\langle u, v\rangle$ exists in G.

The knowledge of vertices at the beginning of the process is represented by some *initial* directed graph $G_0(V, E_0)$ (i.e., each v knows only $ID(v)$ and $\{ID(u) \mid \langle v, u\rangle \in E_0\}$). The graph G grows as vertices learn the addresses of nonneighboring peers. The vertex v can learn $ID(u)$ by receiving a message from u itself or from some other vertex w who knows $ID(u)$. Conceptually, this adds the directed arc $\langle v, u\rangle$ to E. (The case of edge removals by the environment can be handled by rerunning the algorithm from time to time.)

As in [13, 15], the underlying communication network is modeled as a complete undirected graph over the set V of vertices. However, in this paper we assume the asynchronous communication model. Specifically, a vertex can send an arbitrary size message to any of its outgoing neighbors in G, and such a message eventually arrives after a finite but unbounded time. Another difference from [13, 15, 14] is that we discard the assumption that the algorithm is started simultaneously at all the vertices. We maintain the assumption that messages sent by a vertex u to its outgoing neighbor v obey the First-In First-Out (FIFO) discipline. The notion of time complexity we use here is the standard one used in the asynchronous model, that is, for the sake of complexity analysis only, we assume that every message is delivered within *at most* one unit of time. In other words, our algorithm operates at the application layer of the network model, in which the communication network can be thought of as a complete graph.

As defined in [13], the *resource discovery* problem is to compute the connected components in the underlying graph of G_0 (namely, the *undirected* graph obtained from G_0 by removing edge directionality). Formally, a distributed algorithm for the resource discovery problem must satisfy the following property upon termination: For every weakly connected component C of G, there is a *root* vertex $v \in C$, such that G contains a directed arc $\langle v, u\rangle$ for every u in C, and v is designated as the root by every vertex $u \in C$ (by setting a pointer $PTR(u)$ to $ID(v)$).

The problem arises in the wider context of the topology update problem, inherent in distributed networks. In a dynamic environment, the topology may change from time to time, and it is required to learn the new topology (see, e.g., [3, 16]). Separating the logical graph from the underlying communication graph is common in today's fast networks. The logical graph represents the knowledge of the vertices about the topology of the underlying communication network. The simplified notion of topological knowledge used in [13] models it as knowing the ID's of other vertices. More generally, such knowledge may include routing and access-related information (such as passwords). Our algorithm is required to increase the connectivity of the logical graph by learning more about the topology. At the same time, the environment tends to decrease the logical connectivity by introducing topological changes such as link/vertex additions and deletions, cooperative interconnections between different domains, and so on.

It is assumed (as in [13, 15]) that the logical graph G is weakly connected (which is a necessary condition for the solvability of the problem). In [13] it is suggested that one way to ensure weak connectivity [13] is to supply every newly added peer with a pointer to at least one previously added peer. This will not suffice if peers may also leave the graph. This motivates the use of an algorithm that increases the connectivity. On the other hand, the use of such an algorithm explains our assumption that the connectivity of the network may be high, and thus the message complexity may be high if one uses a less careful algorithm. Note that, in more general settings, topological knowledge can be lost not only by a peer leaving the network. In many cases the loss of information enhances the directional nature of the network (and thus moving farther from strong connectivity to a weak one). For example, knowledge may be lost due to losing the connection to a name server, or because of environment changes that make it irrelevant. Gaining or losing knowledge is not uniform or coordinated over the peers, since it is caused by environmental changes and asynchronous distributed algorithms. Consequently, this knowledge is also not necessarily symmetric.

Handling weakly connected graphs is the main technical challenge dealt with in [13, 15]. An alternative approach that was considered in the synchronous case is to first transform the graph into an undirected one and then solve the problem on that graph. Note that in an asynchronous setting such an operation is not that simple. To transform a directed edge (u, v) into an undirected (or bi-directed) one, the node u must send a message to v. In the synchronous

setting, this message arrives within one time unit, and hence any node that did not receive such a message may conclude that no edges directed to it exist. This is no longer the case in asynchronous networks, where the time for the message delivery is unknown and possibly unbounded. Thus, no algorithm can allow vertices to first wait for such messages, and act only afterwards (since this might cause a deadlock in some executions). Even in synchronous networks this approach may not lead to efficient solutions, in particular when E_0 might be large; this is often the case in practical distributed systems which handle network partitions by maintaining a large E_0.

A number of algorithms are presented in [13] for solving the problem in synchronous networks. These algorithms are evaluated in terms of the three standard distributed complexity measures, namely, time, messages (counting the overall number of messages sent throughout the execution) and communication (counting the overall number of bits sent), although the specific terminology of [13] is slightly different. The deterministic solutions for weakly connected directed networks presented therein require either time linear in the diameter of the initial network G_0 or communication complexity $O(n^3)$ (with message complexity $O(n^2)$). On the other hand, the *randomized* solutions of [13] require either time complexity $O(\log^2 n)$, message complexity $O(n \log^2 n)$ and communication complexity $O(n^2 \log^3 n)$ (or equivalently, $O(n^2 \log^2 n)$ pointer complexity), or communication complexity $O(n|E_0|)$ (again with message complexity $O(n^2)$). These complexity bounds hold with high probability. In [14] one algorithm has time complexity $O(\log n)$ and message complexity $O(n^2)$, and another algorithm has time complexity $O(\log^2 n)$ and message complexity $O(n)$.

A deterministic distributed algorithm for the weakly connected case (still for synchronous networks) is presented in [15]. This algorithm has time complexity $O(\log n)$, message complexity $O(n \log n)$ and communication complexity $O(|E_0| \log^2 n)$. However, the algorithm does not function correctly on asynchronous networks.

The main contribution of the current paper is an enhanced version of the algorithm of [15] that solves the resource discovery problem in the *asynchronous model* and is efficient simultaneously in all three complexity measures discussed above, namely, time, communication and messages. In particular, our algorithm has the same asymptotic message and communication complexities as those of the algorithm of [15] (which improved on the complexities of the algorithm of [13]). Moreover, executed in the synchronous model, the algorithm has the same time complexity as that of the algorithm of [15]. Since real P2P networks are asynchronous, this enhancement brings the solution a step closer to realistic models.

An overview of the algorithm is given in Section 2, followed by the algorithm itself in Section 3 and its analysis in Section 4.

2 Overview of the algorithm

The input to the problem consists of a directed graph $G_0(V, E_0)$, where each vertex knows all its outgoing neighbors. A vertex joins the algorithm either by being awakened spontaneously, or when it received the first message of the algorithm. Upon termination, for each weakly connected component of G_0 there is one root vertex which is known to every other vertex in the component, and this root knows the identifiers of all the vertices in its component.

The overall structure of our algorithm is the same as that of [15]; the technical difficulties lie in implementing its high-level operations asynchronously. We start with a high-level review of the (synchronous) algorithm of [15], and then discuss our asynchronous algorithm and the ways in which it differs from its synchronous counterpart.

2.1 A Review of the synchronous algorithm

Let us first review the synchronous algorithm [15]. The main (distributed) data-structure maintained by the algorithm is a directed *pointer graph* $\mathcal{G} = (V, P)$, where $P = \{PTR(v) \mid v \in V\}$ is a set of pointers, one per each vertex. Throughout the execution, the algorithm maintains the invariant that every vertex in $\mathcal{G}$ has out-degree 1, and $\mathcal{G}$ forms a directed forest with self-loops at the roots. Hence at any given time, each vertex v belongs to exactly one tree, denoted $Tree(v)$. Also, $PTR(v)$ always points to some peer known to v, and thus the arcs of $\mathcal{G}$ are a subset of those of the current graph G.

Initially $PTR(v) = ID(v)$ for every vertex v, or in other words, every vertex is the root of a singleton component. Upon termination, the pointer graph consists of a single *star*, namely, exactly one vertex t is the root and all other vertices point to it. The height of each tree T is defined as the larger among 1 and the length (in hops) of the longest leaf-to-root path over T. (Hence a singleton vertex and a star are both defined to have height 1.)

The algorithm requires vertices to change their pointer value. First, a non-root vertex which currently points at a non-root vertex may advance its pointer to an ancestor in its tree in order to reduce its height. A second pointer changing rule is intended to enable merging two trees together. A root u may change its pointer to point to another vertex (and thus stop being a root). This can be done if u has an outgoing edge in G leading to a vertex in another tree. Such a pointer change happens in the algorithm in one of two situations. Either an outgoing edge of some vertex in u's tree leads to a vertex in another tree, or an incoming edge in G from a vertex w, whose root is not u, leads into either u, or into

another vertex in $Tree(u)$, and an offer, sent by w, to join w's tree has been received by u.

Throughout the execution of the algorithm, each tree root is in either *active* or *passive* state. Initially all the vertices are active. A root u becomes passive once its tree stops changing. This may happen in the situation where all three of the following conditions hold:
(a) $Tree(u)$ is a star and u already knows the outgoing edges of all its vertices;
(b) all the outgoing edges of u in G lead to vertices which point to u (so there are no suggestions for new trees to merge with);
(c) all offers to join received so far were handled.

Once a root became passive it stays so until it receives an offer to join some other (active) tree. All non-root vertices in a passive tree (which are necessarily leaves and children of the root) are also considered passive.

During the algorithm, at the first opportunity in which a vertex u points to another vertex w which is a root (and only at that time), u forwards to w its list of outgoing edges (including those previously forwarded to u, during the period in which it has been a root).

The algorithm operates in phases. The number of messages sent by each root may not exceed an integer bound a_i which increases geometrically with the phase number i.

2.2 Overall structure of the asynchronous algorithm

We now give a detailed description of the asynchronous algorithm, focusing on explaining the main changes with respect to the synchronous solution of [15]. Note that in the asynchronous setting, different phases can coexist. A star root v moves to phase i after it is done with all the operations it needs to perform for phase $i-1$.

Phase i consists of the following steps.

Step 1

Each active star-root r tries to form a larger tree by hooking on (i.e., becoming a child of) another vertex, or by helping others to hook upon itself.

Step 1.1 To save messages, a star root r only invites (by "join" messages) some γ_i of its adjacent vertices in phase i, where γ_i grows exponentially with i.

In the algorithm of [15], in order to maintain the property that the set of pointers is always a forest, the star root r attempted to hook itself on the smallest-ID active vertex which could be obtained with the help of the sets of adjacent vertices mentioned above. (That ID did not necessarily have to be smaller than $ID(r)$.)

Unfortunately, in a system that is both asynchronous and weakly connected, this is not easy to implement. Since the system is weakly connected, an intended recipient of a message (e.g. a "join") may not know of the existence of the "join" message sender, and may not know to expect such a message. In a synchronous environment, it suffices for such an intended recipient to wait for one time unit, after which all the "join" messages that were on their way to it must arrive. However, in an asynchronous system this simple solution is not possible. Hence instead, r acts as follows:

A star root r selects a hooking *candidate* as soon as it has any (either in response to r's own "join" messages, or by receiving a "join" message initiated by another star root). At that time we say that the star root r is *matched*.

This means that many of the "join" messages sent through the process are not considered by their recipients. To preserve the loop freedom, we still need to maintain the following property:
"if v considered u as a hooking candidate, then u must consider v."

Thus we add the following mechanism:

Step 1.2 A *matched* star root r which receives a **join** message, replies by a **cancel** message to the star root that originated the **join**. If, however, a **join** arrives at r before it is *matched* then r becomes *matched*, and replies in the same way as in the synchronous algorithm of [15]. (This process is detailed more formally in the next section.) We stress that r does reply to each and every **join** message it receives. Thus, the sender of that **join** message can always know whether its **join** has arrived when the recipient was *unmatched* or not.

Step 1.3 When a star root r is *matched*, if it sent any **join** messages, then r waits until all of them are answered before it finalizes its hooking selection. That is, when all the **join** messages of r are answered, r selects as a hooking *candidate* the smallest-ID star root z such that either r received from z or sent to it a **not-cancelled** reply.

Step 1.4 A star root r hooks itself on its hooking *candidate*, *unless* one of the following three cases applies:

(a) Another active star root w hooks itself on r, w has the smallest ID among the active vertices which r found to be adjacent to it, but the ID of r is smaller than that of w.
In this case, r is the one that remains (a root and) active.

(b) All the outgoing edges of r lead to passive roots and all of them chose to hook on r. (Note that, in fact, had they known an active root s with lower ID than r's, then r itself would have hooked on s too).
In Case (b), r *may* become passive (see Step 4) if $\gamma_r = m_r$.

(c) The root r did send **join** messages to some active star roots, but all of these messages were answered by **cancel** replies. In this case, we refer to r as a *lone star*.
In this case, r chooses arbitrarily one of the active star roots that sent it a **cancel** message, and hooks itself on that star root. (This operation guarantees that every star that has a neighbor to hook into, will do so, providing the progress property needed to ensure the time complexity bound).

Step 1.5 If a vertex r hooked onto a new parent q (either the hooking *candidate*, or via the hooking rule of case (c) above), then r sends a message **new-child** to q, notifying it about the hooking.

Step 1.6 Finally, in any case, r sends a **cancellation-ack** to every vertex that sent it (in this phase) a **cancel** message. Moreover, r waits (before it continues to Step 2 below), until it receives a **cancellation-ack** from every vertex to which r sent a **cancel** message in this phase. Thus, if any lone star hooked on r, then r will learn of this fact before it continues to Step 2.

Step 2

Non-star trees halve their height from h to $\lceil h/2 \rceil$ by a process known as "pointer shortcut." In this operation, each vertex v assigns the value $PTR(PTR(v))$ into its pointer $PTR(v)$ (i.e., it adopts its parent's pointer).

Step 3

In this step, each parent learns about its new children and their state. In particular, the tree root learns the state of the tree after the hooking and shortcut operations.

Step 4

A tree becomes passive if its size did not increase geometrically, its height did not decrease geometrically, and it has considered all its outgoing edges. (The specific details are postponed to Lemma 4.5). A procedure for increasing the number of outgoing edges that a root may consider in each phase is given in Step 1 of the algorithm in Section 3.

The algorithm terminates once no active roots remain. The last active vertex will be the root of a star that includes all the vertices of the graph.

Termination detection can be added in the same way as in [15], so we omit a discussion of this feature here.

3 Detailed algorithmic implementation

Let us now present a precise description of the algorithm. Initially, $PTR(v) = v$ for every vertex v.

In each phase i (starting from $i = 0$):

Step 1

Step 1.1: Each active star root r becomes *unmatched*, and sends out **join** messages on its first γ_i outgoing edges, for $\gamma_i = \min\{m_r, (3/2)^i\}$, where m_r is the number of the outgoing edges from r to a vertex outside its star.
(Note that in the first iteration $\gamma_i = 1$.)

Step 1.2 A vertex z receiving such a **join** message responds as follows. Suppose z currently points to vertex v (possibly z itself). First, z forwards the **join** message to v.

(A1) If v is an active and *unmatched* star root, then v becomes *matched*, and responds with the message **active**, to which it appends its ID.

(A2) If v is an active vertex that is not a star root, then v responds with the message **active**, to which it appends its ID.

(A3) If v is a *matched* star root, then v sends a **cancel** message (appending v's ID) to the originator of the **join** message.

(B) If v is a passive root, it hooks itself on the active vertex w that sent the **join** message.
(Note that if another **join** message arrives at v at that very moment, then it is treated after the hooking is completed, and v is no longer a passive root.)

Step 1.3: The star root r waits until all the **join** messages it has sent are answered.

If an active star-root r found a non-empty set of other active vertices (either by ID's contained in the **active** messages it received, or through **join** messages it received from an active vertex when it was *unmatched*), then r chooses as its hooking *candidate* the smallest-ID active vertex v in this set.

Step 1.4:

(A) If r has chosen a hooking *candidate* v, and r is not the candidate of v, or r is the candidate of v but $ID(v) < ID(r)$, then r hooks into v.

(B) If r has no hooking candidate, but did receive **cancel** messages, then it hooks onto the smallest-ID sender of a **cancel** message.

Step 1.5: Each star root r that hooked on another vertex v, sends v a **new-child** message notifying v about the hooking.

Step 1.6: The vertex r sends a **cancellation-ack** message to every vertex that sent it a **cancel** message in this phase. Moreover, r waits (before it continues to Step 2 below), until it receives a **cancellation-ack** from every vertex to which r sent a **cancel** message in this phase.

Step 2

Every vertex v performs a pointer shortcut operation, setting

$$PTR(v) := PTR(PTR(v)).$$

To facilitate that, every vertex v sends its pointer value to its new children (i.e., those who notified v in Step 3.3 of the previous phase), and every non-root vertex assigns the pointer value received from the parent, to its own pointer variable.

Step 3

Step 3.1: A vertex u that for the first time points to a root w other than itself (i.e., such that prior to this step, $PTR(u)$ was either u or a non-root vertex), sends its list of neighbors to w.

Step 3.2: Every child of a root, that now has no children, notifies the root.

Step 3.3 Every (passive or active) vertex that now points at a new parent w, sends its ID to w.

Step 4

Every active root r declares itself passive, *unless* either:

(i) m_r was larger than γ_i in Step 1.1 of the current phase; or

(ii) an active vertex, which prior to Step 1 of the current phase did not point to r, points to r now (i.e., following Steps 1 or 2 of the current phase).

4 Analysis

The analysis is structured similar to that of [15]. However, the changes in the implementation details of various actions taken by the algorithm force some changes in the proof details.

To begin with, the algorithm immediately implies the following.

Lemma 4.1 *Each root always knows its children, all the neighbors (in E_0) of all its children, and whether it is a star-root.*

Lemma 4.2 *No cycles are created in the pointer graph.*

Proof: Since only a star root may hook itself, if a star root r hooks itself into a non-root v then no cycle can be created, as v's tree does not hook itself.

Note that hooking to another tree occurs in one of two places in the code, namely, either at Step 1.4(A) or at Step 1.4(B). The hooking actions performed in Step 1.4(A) are rather similar to those taken in Step 1 of the algorithm of [15], and therefore it is possible to prove the claim for this case along the same lines as the proof of the corresponding lemma in [15]. This requires us to establish the necessary symmetry property used in that proof, namely, to show that each edge (u, v) considered for hooking at the vertex u is also considered at the vertex v (if v is a star root). It is easy to see that the **cancel** mechanism added in the algorithm introduced above ensures this symmetry property.

Given that, the proof for the hooking in Step 1.4(A) now proceeds as follows. To see that hooking on the smallest-ID neighbor avoids cycles, note that for every vertex v, the linked list starting at it and jumping two pointers at a time must be monotonically decreasing in ID size, namely, the ID of $PTR(PTR(v))$ is smaller than that of v; this is since the vertex $PTR(v)$ chose the vertex $PTR(PTR(v))$ over the vertex v. (We know that the vertex v was indeed considered by the vertex $PTR(v)$, since v did consider the vertex $PTR(v)$).

This leaves us with having to prove the claim for the hooking actions of Step 1.4(B). Note that if vertex r hooked on vertex x in Step 1.4(B), then v sent r a **cancel** message. This means that at that point v was *matched* to some other star root w. We know that $w \neq r$ since, if r has reached Step 1.4(B), then necessarily none of r's **join** messages found an *unmatched* star root. Thus r's hooking on v cannot create a cycle. ∎

Finally, the following lemma is proved in the same way as in [15].

Lemma 4.3 *As long as not all vertices belong to a single star, there is at least one active root.*

For the sake of analyzing time complexity, we use the term "*after phase i*" to designate the time after all the active star roots performed at least i phases.

The following lemma is needed to make the analysis of [15] go through in the asynchronous environment as well. Its proof is immediate from the lone star mechanism (Step 1.4(B)).

Lemma 4.4 *In every phase where an active star root sends out* **join** *messages, it either hooks on another tree or gets hooked upon by other trees.*

Lemma 4.5 *Following phase i, the size of an active tree is at least $(3/2)^i$.*

Proof: We rely on the following inductive claims.

(1) Following phase i, the sum of the heights of all the active trees in the pointer graph is at most $t/(2/3)^i$, where t is the total number of vertices in active trees.

(2) If at phase i, only passive vertices are hooked on an active root r, then either its tree size increases by $(3/2)^i$ vertices, or r becomes passive.

The proof uses Lemma 4.4 in the proof of claim (2). Except for that, it is the same as the proof of the analogous lemma in [15].

For active trees of height greater than 1, progress is provided by the pointer shortcut operation which decreases the height of non-star active trees. Note that the merging of a passive tree does not increase the height of an active tree T, since the passive root always becomes a child of the root of T (at Step 1). Since the passive tree is a star, this may temporarily increase the height of T to 2 (if it was 1 prior to the merge), but Step 2 will immediately reduce it back to 1. So the height of an active tree T can increase only by the merging of other active trees (in Step 1). However, this does not increase the sum of their heights. Then, in Step 2, the progress stated in claim (1) is achieved. Since the height of a tree is at least 1, claim (1) implies the desired lower bound on the size of each active tree; to see this, note that if we isolate the vertices and edges of one or more active trees, the same analysis would apply to the isolated part.

For active trees of height 1, the progress proof of claim (1) relies on claim (2). Note that if the active root remained active, then its contribution to the parameter t grew larger, while the height of the tree did not increase. On the other hand, if the tree $T(q)$, for some root q, becomes passive, then consider the execution of the algorithm on a different system of vertices, that does not include the vertices of $T(q)$ (but is identical to G otherwise). The execution up to (and including) phase i on both systems is identical. Thus, the induction hypothesis continues to hold for the set of active trees that does not include $T(q)$.

The lemma follows. ∎

Corollary 4.6 *Following phase i, there are at most $n/(3/2)^i$ active trees.*

In analyzing the time complexity of the algorithm, one needs to take into account the fact that in the asynchronous model assumed here, not all vertices start the algorithm at the same time. The last corollary implies the following.

Corollary 4.7 *The algorithm terminates $O(\log n)$ time units after the last vertex joined the execution.*

It is easy to see that if only some of the vertices are awaken by an external signal, then the time complexity of any algorithm is $\Omega(n)$, since $\Omega(n)$ time might elapse until the chain of messages arrives at the last vertex (say, when the initial graph G_0 is a line). Now let us adopt the assumption of [13] that all the vertices are awaken (e.g., periodically) by an external signal. Let ΔT denote the difference between the wake-up times of the first and last vertices to be awakened. From the above corollary we get the following.

Corollary 4.8 *The time complexity of the algorithm is $\Delta T + O(\log n)$.*

For analyzing the message complexity, we observe that in phase i, each root initiates $O((3/2)^i)$ message chains of length $O(1)$ (i.e., messages followed by answer messages, and so on). This implies the following.

Corollary 4.9 *The algorithm sends $O(n)$ messages per phase, and its overall message complexity is $O(n \log n)$.*

We also observe that per phase, each edge in E_0 is broadcast $O(1)$ times (plus some lower order terms). Therefore we have the following.

Corollary 4.10 *The algorithm sends $O(|E_0| \log n)$ bits per phase, and its overall communication complexity is $O(|E_0| \log^2 n)$ bits.*

Theorem 4.11 *When executed on a weakly connected graph $G = (V, |E_0|)$, the algorithm terminates in time $\Delta T + O(\log n)$ and uses $O(n \log n)$ messages totaling $O(|E_0| \log^2 n)$ bits. Upon termination, the pointer graph is a star tree, where the star root knows the ID's of all the vertices, every other vertex knows the ID of the star root, and the pointer PTR of every other vertex points at the star root.*

References

[1] B. Awerbuch and Y. Shiloach. New Connectivity and MSF Algorithms for Ultracomputer and PRAM. *IEEE Trans. on Computers* **36**, (1987), 1258–1263.

[2] I. Cidon, I. Gopal, and S. kutten. New Models and Algorithms for Future Networks. *IEEE Trans. on Information Theory* **41**, (1995), 769–780.

[3] I. Cidon, I. Gopal, M. Kaplan, and S. kutten. Distributed Control for Fast Networks. *IEEE Trans. on Communications*, **43**, (1995), 1950–1960.

[4] R.G. Gallager, P.A. Humblet, and P. M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. on Programming Lang. and Syst.* **5**, (1983), 66–77.

[5] Gnutella home page http://gnutella.wego.com.

[6] Freenet home page http://freenet.sourceforge.net.

[7] PPWAN: Peer-to-Peer Without a Name, http://dividuum.de/p/ppwan/.

[8] ALPINE: Adaptive Large-scale P2p Information Networking, http://cubicmetercrystal.com/alpine/.

[9] The GDNP: The G Diffuse Networking Protocol, http://web.g.diffuse-network.org/.

[10] Retsina: Discovery of Infrastructure in Multi-Agent Systems, http://www-2.cs.cmu.edu/~softagents/retsina.html.

[11] JXTA home page, http://www.jxta.org.

[12] P2P seminar home page. http://iew3.technion.ac.il/ kutten/p2p-seminar.html.

[13] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, May 1999, pp. 229-237.

[14] C. Law and K-Y. Siu. An $O(\log n)$ Randomized Resource Discovery Algorithm. Brief Announcements of the 14th Symp. on Distributed Computing, Tech. Report FIM/110.1/DLSIIS/2000, Technical University of Madrid, Oct. 2000, pp. 5–8.

[15] S. Kutten, D. Peleg, and U. Vishkin. Deterministic Resource Discovery in Distributed Networks. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures*, 2001 Crete, pp. 77–83.

[16] J. McQuillan, I. Richer, and E. Rosen. The New Routing Algorithm for the Arpanet. *IEEE Trans. on Communications* **28**, (1980), 711–719.

[17] Y. Shiloach and U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms* **3**, (1982), 57–67.