Asynchronous Resource Discovery

Ittai Abraham [*]
School of Engineering and Computer Science
The Hebrew University
Jerusalem, Israel
ittaia@cs.huji.ac.il

Danny Dolev [†]
School of Engineering and Computer Science
The Hebrew University
Jerusalem, Israel
dolev@cs.huji.ac.il

ABSTRACT

Consider a dynamic, large-scale communication infrastructure (e.g., the Internet) where nodes (e.g., in a peer to peer system) can communicate only with nodes whose id (e.g., IP address) are known to them. One of the basic building blocks of such a distributed system is resource discovery - efficiently discovering the ids of the nodes that currently exist in the system. We present both upper and lower bounds for the resource discovery problem. For the original problem raised by Harchol-Balter, Leighton, and Lewin [3] we present an $\Omega(n \log n)$ message complexity lower bound for asynchronous networks whose size is unknown. For this model, we give an asymptotically message optimal algorithm that improves the bit complexity of Kutten and Peleg [4]. When each node knows the size of its connected component, we provide a novel and highly efficient algorithm with near linear $O(n\alpha(n,n))$ message complexity (where α is the inverse of Ackerman's function). In addition, we define and study the Ad-hoc Resource Discovery Problem, which is a practical relaxation of the original problem. Our algorithm for ad-hoc resource discovery has near linear $O(n\alpha(n,n))$ message complexity. The algorithm efficiently deals with dynamic node additions to the system, thus addressing an open question of [3]. We present a $\Omega(n\alpha(n,n))$ lower bound for the Ad-hoc Resource Discovery Problem, showing that our algorithm is asymptotically message optimal.

1. INTRODUCTION

When multiple nodes of a dynamic distributed system want to interact and cooperate, one of the basic building blocks needed is a mechanism for nodes to discover the existence of each other. This mechanism must also manage the dynamism of rapid node additions and node removals. Originally, Harchol-Balter, Leighton, and Lewin [3] defined the distributed problem of learning the ids of the nodes of a system and named it the *Resource Discovery Problem*.

The problem arises in many peer-to-peer systems when peers across the Internet initially know only a small number of peers. In such cases resource discovery algorithms may be used in order to efficiently discover all the peers that are weakly connected to each other. Once all peers that are interested know of each other they may cooperate on joint tasks (for example: peers may divide work on a complex computation or may build an overlay network and form a distributed hash table [8, 1, 9, 12, 7]). Resource discovery may be used as a key building block for repairing damaged peer to peer systems. Consider a system in which many of the nodes were either reset or totally removed from the system. The first step toward rebuilding such a system is discovering and regrouping all the currently online nodes.

We begin by formally defining "knowledge graphs". Consider a set V of n nodes. Each node v has a unique identifier $id(v)$ consisting of $O(\log n)$ bits. This identifier can be thought of as the node's IP address. The network is modelled as a directed graph $G = (V, E)$, where an edge $(u \rightarrow v) \in E$ denotes that node u knows $id(v)$. A node u can send messages to node v only if $(u \rightarrow v) \in E$. Denote the initial edge set as E_0; unless otherwise noted, we do not assume that (V, E_0) is strongly connected. Messages sent can be of arbitrary length and may contain ids of other nodes. The edge set E grows each time a node receives an id of a node it did not know of. Thus, when a node v receives a message containing $id(w)$ then $E := E \cup \{(v \rightarrow w)\}$.

This formal "knowledge graph" system may model Internet computations. In the Internet, each node has a unique IP address. The TCP/IP routing protocol creates a fully connected underlying network. Once an IP address is known to a node, the node may use the TCP/IP protocol to route messages to the destination whose IP address it knows. Thus after node u learns of the IP address of node v, node u may send messages to v. The "knowledge graph" model assumes that message costs are equal, this may represent the constant cost of building up a TCP/IP connection, while disregarding the variation in the number of hops caused by TCP/IP.

[*]This research was supported in part by NDS PhD fellowship program.

[†]This research was supported in part by Intel COMM Grant - Internet Network/Transport Layer & QoS Environment (IXA).

Recall that two nodes belong to the same weakly connected component if there is path between them in the undirected graph induced by G in which we ignore edge direction. Two nodes u, v belong to the same strongly connected component if there is a directed path from u to v, and a directed path from v to u.

A distributed algorithm for the Resource Discovery Problem strives to achieve the following:

1. Exactly one node in every weakly connected component is designated as leader.
2. The leader node knows the ids of all the nodes in its component.
3. All non-leader nodes know the id of their leader.

Resource discovery algorithms are measured by three common complexity measures: total number of messages, total number of bits sent, and the number of rounds to completion (for synchronous models).

On a strongly connected network, the $O(n)$ message complexity leader election algorithm of Cidon, Gopal, and Kutten [2] may be used. Once a leader is chosen, another $O(n)$ messages are needed to fulfil the resource discovery termination properties. For weakly connected, synchronous networks, it is possible to make the graph strongly connected by adding to every directed edge an edge in the opposite direction, thus sending $|E_0|$ message. Therefore, an $O(n + |E_0|)$ message complexity algorithm can be achieved based on the work of [2]. For sparse networks in which $|E_0| = O(n)$ this is asymptotically message optimal. Accordingly, the algorithmic challenge for the Resource Discovery Problem is for networks that are weakly connected and non-sparse (where $|E_0| = \Omega(n \log n)$).

1.1 Previous Results

For the synchronous model, Harchol-Balter, *et al.* [3] presented a randomized algorithm that, with high probability, achieves $O(n \log^2 n)$ message complexity, $O(n^2 \log^3 n)$ bit complexity, and $O(\log^2 n)$ time complexity. Law and Siu [6] presented a randomized algorithm that combined with elements of the algorithm of [3] achieves, with high probability, $O(n \log n)$ message complexity, $O(n^2 \log^2 n)$ bit complexity, and $O(\log n)$ time complexity on weakly connected graphs. Note that both randomized algorithms of [3] and of [6] relay heavily on the fact that the number of nodes in the network is known. Kutten, Peleg, and Vishkin [5] presented a deterministic algorithm that achieves $O(n \log n)$ message complexity, $O(|E_0| \log^2 n)$ bit complexity, and $O(\log n)$ time complexity that does not need to know in advance the network size. In addition [5] shows that when there exists an upper bound on the network size, termination detection is possible.

All the above algorithms were built for synchronous networks and do not maintain their correctness and complexity measures in asynchronous settings. Real communication systems tend to have a variable and non-deterministic delay time on messages sent. Thus seeking to bring a solution a step closer to realistic models, Kutten and Peleg [4] studied the Resource Discovery Problem in asynchronous networks, and presented a deterministic algorithm that achieves $O(n \log n)$ message complexity, $O(|E_0| \log^2 n)$ bit complexity.

1.2 Our Results

Following Kutten and Peleg [4], we study the asynchronous model, where network communication is asynchronous and reliable; messages sent will eventually arrive after a finite but unbounded time. There is no global initialization time; nodes begin asynchronously and may wake-up nearby neighbors. Thus the wake-up time complexity is $\Omega(n)$. We assume that all messages sent from u to v maintain a FIFO (first in first out) ordering when arriving at v.

A distributed leader election algorithm terminates when all nodes decide whether they are in leader or non-leader state. In asynchronous weakly connected networks, leader election algorithms cannot terminate. Suppose a leader election algorithm has a terminating execution on a network G, then combine two G's and a single node u. Add a directed edge from u to both copies of G. Now wake up all nodes except node u. Each copy of G will elect a leader and terminate. This will cause a termination with two leaders. Therefore, instead of termination requirements we would like to have the following ad-hoc requirements at any phase during execution:

1. Each node is either a leader, or belongs to a single leader.
2. The leader node knows the ids of all the nodes that belong to it.
3. All non-leader nodes know the id of their leader.

In addition we have the following progress requirement:

- When all nodes have no more messages to send, and all message queues are empty, exactly one leader will remain in each weakly connected component.

The complexity of an algorithm for resource discovery measures the total number of bits and messages used until this steady state is reached.

We present both upper bounds and lower bounds for the asynchronous Resource Discovery Problem. For this problem, we study three models. In the first model, the *oblivious* model, nodes do not know the network size. In the second, the *bounded* model, each node knows the number of nodes in its weakly connected component. In the third model we study a novel variation, the *Ad-hoc Resource Discovery Problem*, in which we relax the third requirement for resource discovery.

We present an $\Omega(n \log n)$ message complexity lower bound for asynchronous networks whose size is unknown. This result shows that the Resource Discovery Problem on directed

graphs is in a different complexity class than on undirected graphs, for which [2] have an $O(n)$ upper bound.

For this model we also show a deterministic $O(n \log n)$ message complexity, $O(|E_0| \log n + n \log^2 n)$ bit complexity algorithm that improves the bit complexity of [4].

When the network size is known, we present a deterministic near linear $O(n\alpha(n,n))$[1] message complexity algorithm with bit complexity of $O(|E_0| \log n + n \log^2 n)$. In addition, our algorithm knows when to terminate, thus addressing a question raised by [3].

Another open question raised by [3] asked how to efficiently execute resource discovery on dynamic networks. Explicitly, can the complexity of fully incorporating a new node be reduced to less than the complexity of running the whole algorithm again? We positively answer this question using the following model.

We define the *Ad-hoc Resource Discovery Problem* in which the algorithm must have properties (1) and (2) of the Resource Discovery Problem, and the following relaxation of property (3):

- Each non-leader node has an identified pointer.
- These pointers induce a directed path from any non-leader node to its leader.

This relaxation enables interested nodes to query the leader without having a direct connection to it. An $\Omega(n\alpha(n,n))$ message complexity lower bound for this model is presented.

Our algorithm for the ad-hoc resource discovery problem has $O(n\alpha(n,n))$ message complexity (asymptotically optimal), and $O(|E_0| \log n + n \log^2 n)$ bit complexity. Ideally, we would like the length of the path between any non-leader node to the leader to be bounded by $O(1)$. Our algorithm achieves an amortized bound: for any m requests to reach the leader the total length of all paths is $O(m\alpha(m,n))$. In addition, our algorithm efficiently deals with dynamic node and link additions.

2. MESSAGE COMPLEXITY BOUND RESOURCE DISCOVERY

In this section we prove that any algorithm for the Resource Discovery Problem on an asynchronous system with n nodes, where n is unknown to the participating nodes, must send at least $\Omega(n \log n)$ messages[2]. The proof's idea is to show that on a certain network topology, an adversary that controls the time that each message arrives can force any algorithm to spend messages by causing temporary leaders to send messages to all the nodes they know of and then to reveal to them that there are more nodes in their weakly connected component.

THEOREM 1. *For any distributed algorithm for the Resource Discovery Problem there exists an execution in which at least* $0.5n \log n - 2$ *messages are sent.*

PROOF. Consider a complete rooted binary tree $T(i)$ with $n = 2^i - 1$ nodes, were all edges are directed toward the leaves. We prove by induction that on the network $T(i)$ any algorithm can be forced to send at least $i2^{i-1} - 2$ messages. For $i = 2$, $T(2)$ is a tree with 3 nodes, and any leader that any algorithm chooses must know the ids of the two other nodes so at least 2 messages need to be sent.

Suppose that the theorem holds for i, so for $T(i+1)$ an adversary can stall all messages sent by the root until both subtrees have no more messages to send. Note that before the root sends its message no subtree can learn about the rest of the tree and thus each subtree is forced to believe that its subtree is the whole connected component. By the induction hypothesis every algorithm can be forced to send at least $i2^{i-1} - 2$ messages for each subtree.

Now consider two cases: (1) The root is eventually chosen as the leader: in this case all other $2^{i+1} - 2$ nodes must receive a message to know the leader's id. (2) A non-root node is eventually chosen as the leader. In this case the root must send one message, at least one other message must be sent to the leader with the ids of the nodes of the other side, and 2^i nodes (the root and the nodes of the other side) must each receive the leader's id. Thus in both cases at least $2(i2^{i-1} - 2) + 1 + 1 + 2^i = (i+1)2^i - 2$ messages are sent on $T(i+1)$. □

Note that the request that all nodes will learn of the leader's id is the key source to the logarithmic factor in the proof. Thus this bound does not hold for the Ad-hoc Resource Discovery Problem.

3. MESSAGE COMPLEXITY BOUND AD-HOC RESOURCE DISCOVERY

In this section we show a reduction from the classic Union-Find Problem on disjoint sets to the asynchronous Ad-hoc Resource Discovery Problem. This reduction bounds the message complexity of the ad-hoc resource discovery to the running time of the Union-Find Problem.

LEMMA 1. *Given any $h(n)$ message complexity algorithm for the asynchronous Ad-hoc Resource Discovery Problem, a Union-Find Algorithm on a universe of n sets, can be built that has a $h(2n - 1 + m)$ time bound for any sequence of $n - 1$ merges and m finds.*

PROOF. Consider a universe of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Denote $F(i)$ the operation that finds the representative of S_i's , and denote $U(i,j)$ the union operation that unites the two sets that contained S_i and S_j. We assume that the two sets were disjoint prior to this operation. Let $\mathcal{U}$ be any sequence of $n - 1$ union operations and m find operations.

[1] Inverse Ackerman function:
$\alpha(m,n) = \min\{i \geq 1 | A(i, \lfloor m/n \rfloor) > \log n\}$,
where A is Ackerman's function:
for $m = 0$: $A(0,n) = n + 1$,
for $m > 0$, $n = 0$: $A(m,0) = A(m-1,1)$,
for $m > 0$, $n > 0$: $A(m,n) = A(m-1, A(m,n-1))$.

[2] We will later prove that if n is known in advance then the problem can be solved in message complexity $O(n\alpha(n,n))$

Assume a distributed algorithm for the asynchronous Ad-hoc Resource Discovery Problem with Message complexity $h(n)$ for a network with n nodes. We build the following network G: For each set S_i, add one node s_i. For each union operation $U(i,j)$ in $\mathcal{U}$, add a node $u_{i,j}$ and two directed edges: $(u_{i,j} \rightarrow s_i)$, and $(u_{i,j} \rightarrow s_j)$. For each find operation $F(i)$ in $\mathcal{U}$, add a node f_i and a directed edge $(f_i \rightarrow s_i)$.

Since the distributed algorithm works for asynchronous networks, we may control the wake-up of nodes in the following manner: Start from the first operation in $\mathcal{U}$. If the operation is $U(i,j)$ then wake up node $u_{i,j}$, if the operation is $F(i)$ then wake up node f_i, wait until the algorithm has no more messages to send, move to the next operation in $\mathcal{U}$ and loop until all nodes are awaken.

The Union-Find Algorithm for the sequence $\mathcal{U}$ needs just to simulate the Distributed Resource Discovery Algorithm on G for the wake-up sequence described above. The simulation of the $h(2n-1+m)$ messages sent will take at most $h(2n-1+m)$ time.

Due to the requirements of the Ad-hoc Resource Discovery Problem, each time a node $u_{i,j}$ is woken up, the algorithm will continue its execution until both nodes s_i, and s_j have the same leader. Thus the wake-up faithfully simulates the union operation. Each time a node f_i is woken up, a computation starting from s_i will reach the leader (since the leader must know f_i's id). Thus the wake-up of f_i faithfully simulates the find operation. □

The work of Tarjan [11], shows that for any algorithm on a pointer machine that has the separation property, a sequence with $n-1$ merges intermixed with n finds can be built that will require $\Omega(n\alpha(n,n))$ time. Note that our simulation is of a pointer machine that has the separation property (nodes of one disjoint component never have pointers to nodes of another disjoint component). Thus we have proved:

THEOREM 2. *For any distributed algorithm for the Ad-hoc Resource Discovery Problem there exists an execution were the algorithm sends at least $\Omega(n\alpha(n,n))$ messages.*

4. THE GENERIC ALGORITHM

In this section we present the generic resource discovery algorithm for asynchronous networks when network size is unknown and the network may not be strongly connected. Then we show two variations: (1) when the network size is known, (2) for the Ad-hoc Resource Discovery Problem.

Each node wakes up and begins its execution. Initially nodes begin as leaders and try to conquer other leader nodes. When a node is conquered it becomes inactive and only acts as a message router. When active, a leader node v seeks to enlarge its domain by the following steps:

1. Find an unexplored node u.
2. Reach the current leader, l, of node u.
3. Merge l into v.
4. Inform all of l's nodes of their new leader.

Each node begins in state 'leader' and may change its state to 'conquered', 'conqueror' or 'inactive'. Each node maintains five sets of ids: *local*, *done*, *more*, *unaware*, and *unexplored*, a FIFO queue *previous*, two id pointers: *id*, and *next*, and one integer: *phase*. The *id* field holds the node's id. Initially *local* holds the set of ids the node initially knows. If *local* is not empty then $more = \{id\}$ and $done = \emptyset$. Otherwise, if *local* is empty then $done = \{id\}$ and $more = \emptyset$. Initially, $next = id$, $phase = 1$, the sets *unaware*, *unexplored*, and the queue *previous* are empty.

4.1 Finding an unexplored node

A leader node v first needs to find a new node to expand its territory. The set $v.unexplored$ contains such nodes. $v.more$ contains nodes that may still have edges to unexplored nodes.

If $v.unexplored$ is not empty then just choose any id u from $v.unexplored$ and continue to subsection 4.2.

Otherwise, if $v.more$ is not empty then choose any id w from $v.more$ and send w a 'query' message informing it to remove $\min\{|v.more| + |v.done| + 1, |w.local|\}$ ids from its $w.local$ set and send them back in a 'query reply' message. Any new id (not in $v.more$ or $v.done$) that is received by v via a 'query reply' message is put into $v.unexplored$. Note that v itself may appear in $v.more$, in this case v simulates the message sending internally.

If the $w.local$ set is emptied, then w announces this in its 'query reply' message and v then moves w from $v.more$ to $v.done$.

Since v knows only $|v.more| + |v.done|$ ids then by receiving $\min\{|v.more|+|v.done|+1, |w.local|\}$ ids, either $v.unexplored$ is not empty or w has moved to $v.done$ (or both). The low bit complexity of the algorithm is due to this balance. Leader nodes receive just as many ids as needed in order to progress. The trivial solution of receiving all of w's ids would lead to a higher bit complexity $O(|E_0| \log^2 n)$.

If both $v.unexplored$ and $v.more$ are empty, the leader v waits until $v.more$ becomes non-empty (this may happen if a 'search' message arrives from a previously unknown node as explained at the next subsection).

4.2 Reaching the current leader of another node

Once a leader node v finds an unexplored node u it searches for u's leader. Node u's current leader is found by sending a 'search' message that follows the *next* pointer of each node until the leader is reached (note that the leader found may be inactive as explained at the end of the next subsection). The search message consists of $(v.id, v.phase, u.id)$. When a 'search' message M arrives from a node y to an inactive node x, then the pair (M, y) is enqueued, formally $x.previous.enqueue(M, y)$. Node x forwards message M to $x.next$ only if M is the only message in the queue, that is $|x.previous| = 1$.

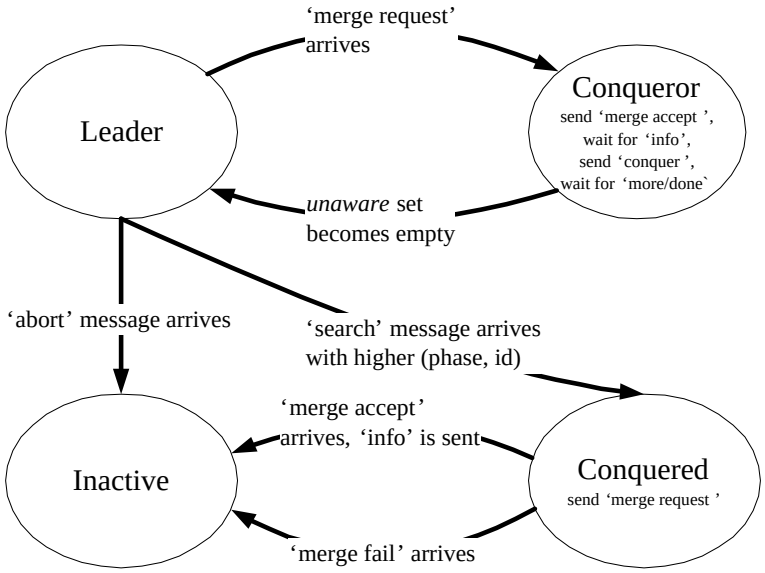

Figure 1: Node's state transition diagram.

When a 'search' message finally reaches the current leader l of u, a path compression 'release' message containing l is sent on the opposite direction along the path using the pointers stored in the *previous* queues. When a 'release' message arrives at an inactive node x, node x sets $(M, y) := x.previous.dequeue()$, $(M', z) := x.previous.peek()$. It forwards the 'release' message to y, sets $x.next := l$ and releases the search message M', if M' exists, by forwarding it to $x.next$.

Finally, if v is not in $l.unexplored$ and u is in $l.done$ then: (1) when the 'search' message reaches l, l moves u from $l.done$ to $l.more$, (2) when the 'release' message reaches u, node u adds v to $u.local$. Thus l (or future leaders that will conquer l) can later explore node v.

4.3 Merging of two leaders

Whenever a 'search' message M initiated at a leader node v reaches a leader l, node l enqueues (M, y) into $l.previous$ FIFO queue (where y is the node that forwarded M to l).

When $l.previous$ is non-empty, then l peeks at the first message in the queue, $M = (v.id, v.phase, u.id)$. Leader l needs to decide whether to merge into v.

If $v.phase > l.phase$ or if $v.phase = l.phase$ and $v.id > l.id$ then node l changes its state to 'conquered' and does the following: Reply to all 'merge request' messages with a 'merge fail' message. Send a 'merge request' message to v. If a 'merge fail' message arrives from v then set state to 'inactive'. Otherwise, if a 'merge accept' message arrives from v then set $l.next$ to v and send an 'info' message with $(l.phase, l.more, l.done, l.unaware, l.unexplored)$ to v and become 'inactive'.

Otherwise, if node v has a smaller phase or the same phase but a smaller id then the merge is aborted. In such a case an 'abort' message is sent to v, and leader node v becomes inactive. Node v will wait until some other leader's 'search' message will find and conquer it.

At the end of each of the cases above node l dequeues (M, y) from $l.previous$ and sends y a 'release' message containing l.

4.4 Conquering unaware nodes

If a leader node v receives a 'merge request' message and is not in the 'conquered' state then it enters the 'conqueror' state, responds with a 'merge accept' message, stops processing 'search' messages, and waits for an 'info' message.

Once an 'info' message arrives from a node l, node v sets $v.unaware := v.unaware \cup l.more \cup l.done \cup l.unaware$, and $v.unexplored := v.unexplored \cup l.unexplored \setminus \{l.more \cup l.done \cup l.unaware\}$.

Leader v checks whether $l.phase = v.phase$ or if $|v.more| + |v.done| + |v.unaware| \geq 2^{v.phase+1}$ in these cases node v increments its phase ($v.phase := v.phase + 1$).

The *unaware* set contains all the new nodes that do not yet know that their leader has changed. Now v informs the new nodes about its id by sending a 'conquer' message containing $v.id$ and $v.phase$ to each node in $v.unaware$.

An inactive node that receives a 'conquer' message from a phase higher than its current leader updates its *next* and *phase* pointers accordingly and responds with a 'more/done' message that indicates with a bit if the node's *local* set is empty. This is done, since the process we used for merging the sets of a conquer node loses that information.

When v receives a 'more/done' message it moves the node from the $v.unaware$ set into either $v.done$ or $v.more$ depending on the bit indicator. When $v.unaware$ becomes empty v resets it state to 'leader' and resumes processing

'search' messages from $v.previous$ and it restarts the algorithm's loop by finding a new unexplored node.

4.5 Variations on the Generic Algorithm

All the variations that follow change the behavior of the conquer phase of the algorithm. The variations do not maintain the *unaware* set at all. When a 'merge accept' message arrives from v to a node l in a 'conquered' state, node l responds by sending node v an 'info' message containing $(l.phase, l.more, l.done, l.unexplored)$ and l sets $l.next$ pointer to v. When v receives an 'info' message from l, it merges each set accordingly:
$v.done := v.done \cup l.done$,
$v.more := v.more \cup l.more$, and
$v.unexplored := v.unexplored \cup l.unexplored \setminus \{l.done \cup l.more\}$).

4.5.1 The Bounded Model

In the *bounded* model every node knows the size of the component it belongs to. Let n denote the component's size. When a leader node reaches $|done| = n$, it sends a 'conquer' message to all the nodes in *done* and terminates.

4.5.2 The Ad-hoc Resource Discovery Problem

For the Ad-hoc Resource Discovery Problem, leaders never send 'conquer' messages. Instead, when a node wants to know the current snapshot of the ids in the component, it sends a message to the leader (similar to the 'search' messages) and performs a path compression on the reply (similar to the 'release' messages). In the Ad-hoc model nodes may dynamically join the network, and new edges may be added online. New nodes begin as leaders and execute the algorithm as if they woke up just now. When a new edge is added to a node v, then if node v was in its leader's *done* set, then node v sends a message to its leader node l (and does path compression on the way back) informing leader l that node v should be moved back from $l.done$ to $l.more$.

5. ANALYSIS OF THE ALGORITHMS

5.1 Liveness and Termination

In this section we show that when all nodes have no more messages to send, and when all message queues are empty, exactly one leader will remain in each weakly connected component, and the requirements of the Resource Discovery Problem are fulfilled. A node is active if its state is not 'inactive'.

LEMMA 2. *At any stage of execution, at least one leader remains in an active state.*

PROOF. A leader becomes inactive in the following cases: (1) when it receives an 'abort', (2) when it sends a 'merge request' and receives a 'merge fail', or (3) when it sends a 'merge request' and receives a 'merge accept'. At any given moment, the leader with the lexicographically highest phase, id pair cannot become inactive since it will neither receive an abort, nor become conquered. □

Another concern is that all the computations initiated by all active nodes will enter an indefinite wait state and the algorithm will become deadlocked before fulfilling the requirements.

LEMMA 3. *As long as there is more than a single active leader in each weakly connected component, at least one active nodes's computation is free to continue its execution.*

PROOF. A node cannot indefinitely wait at 'conquered' or 'conqueror' states, since the messages it waits for will always arrive. A node in state 'leader' enters a wait state only when its search messages queue is empty.

Thus, leaders in state 'leader' 'conquered' or 'conqueror' that receive a 'search' message eventually send a 'release' message. Therefore, all search messages at all nodes will eventually get processed.

Hence, eventually there will be no pending search messages left, and no node remains at state 'conquered' or 'conqueror'. All remaining leaders are in state 'leader' (and as we proved there is at least one such leader in every weakly connected component). A deadlock in this case can occur only when all *more* and *unexplored* sets at all leaders are empty. This implies that the remaining leaders are disconnected. Thus, there is a single leader in each weakly connected component.

In the ad-hoc model, that sole leader is the root of a directed tree containing all the nodes in its weakly connected component.

In the bounded model, the number of nodes in the weakly connected component is known in advance. A leader that sends a 'conquer' message to all its nodes, knows that it is the sole leader and terminates. □

Combining the lemmas we have proven the following:

THEOREM 3. *The Generic Algorithm (respectively, the Ad-hoc Algorithm) fulfils the Resource Discovery (respectively, the Ad-hoc Resource Discovery) requirements.*

THEOREM 4. *The algorithm for the Bounded model never enters a deadlock state and upon termination fulfills the Resource Discovery requirements.*

5.2 Message complexity

We bound the message complexity by bounding each message type and then summing up.

We begin with the 'query' and 'query reply' message types. Each time such a pair is sent then either (1) the queried node moves to *done*, or (2) the leader node receives enough ids to ensure that its *unexplored* set is not empty. In this case the leader will eventually either (2a) conquer another leader, or (2b) become inactive.

Case (1) may occur at most $2n$ times. Note that nodes may move more than once from *more* to *done*. The first n counts the number of nodes moving for the first time. The second

n is due to the fact that every time a node moves from *done* to *more*, the leader that initiated the 'search' message that caused the change becomes inactive, and that can happen at most n times.

Cases (2a) and (2b) may occur at most n times each, since a node may conquer at most n other nodes and at most n nodes may become inactive.

LEMMA 4. *The number of 'query' and 'query reply' messages is at most* $4n$.

Each 'search' - 'release' computation simulates a *find* operation. Such computation either ends with a *merge* of the leader with another, or with the leader becoming inactive due to an 'abort' message. Thus, at most $2n$ *find* operations and n *merge* operations occur. Our algorithm simulates a sequential execution of Tarjan's classical union/find algorithm for disjoint sets. By the analysis of Tarjan and van Leeuwen [10], a total of $O(n\alpha(n,n))$ 'search' - 'release' messages are sent.

LEMMA 5. *The number of 'search' and 'release' messages is* $O(n\alpha(n,n))$. *The number of 'abort' messages is at most* n.

Following a 'merge request' the sender becomes inactive, this occurs at most n times. This results in a sequence 'merge request','merge fail' or a sequence 'merge request', 'merge accept', and 'info'.

PROPOSITION 1. *The number of 'merge request', 'merge accept', 'merge fail', and 'info' messages is at most* $3n$.

In the Generic Algorithm, each time an inactive node is sent a 'conquer' message its phase increases. The maximum phase is $logn$, since each time the phases increases the size of $|more| + |done|$ is doubled; thus, there are at most $2n \log n$ messages of type 'conquer', 'more/done'. In the Bounded Model, these messages are sent only at the last phase, thus $2n$ such messages are sent.

PROPOSITION 2. *The number of 'conquer', 'more/done' messages is at most* $2n \log n$ *in the Generic Algorithm, and at most* $2n$ *in the Bounded model.*

THEOREM 5. *The message complexity of the Generic Algorithm is* $O(n \log n)$.

THEOREM 6. *The message complexity of the Bounded, and the Ad-hoc algorithms is* $O(n\alpha(n,n))$.

5.3 Bit complexity

As above, we bound the total number of bits by bounding each message type and summing up.

For each edge of $(u \to v)$ in E_0, id v may appear in the 'query reply' messages at most once. This occurs when it is first requested from u by a query message. Note that during the execution, directed edges of E_0 may cause an opposite edge to be added to a node's *local* set (see the end of Section 4.2 for details). Thus, in addition, for each edge of $(u \to v)$ in E_0 the id u may appear in the 'query reply' messages at most once.

LEMMA 6. *The total number of bits in 'query reply' messages is at most* $2|E_0| \log n$.

Every leader maintains the inequality $|more| + |done| + |unaware| < 2^{phase+1}$, and thus $|unexplored| < 2^{phase+1}$. So when a leader gets conquered, the 'info' message it sends contains at most $2^{i+2} \log n$ bits. Since the number of leaders that reach phase i is at most $n/2^i$, summing over all leaders by phase gives $\sum_{1 \le i \le \log n} (n/2^i)(2^{i+2}) \log n = 4n \log^2 n$.

LEMMA 7. *The total number of bits in 'info' messages is at most* $4n \log^2 n$.

All other messages are of length $O(\log n)$ and the message complexity is $O(n \log n)$. Thus the bit complexity of all other messages is $O(n \log^2 n)$

THEOREM 7. *The bit complexity of the resource discovery algorithms is* $O(|E_0| \log n + n \log^2 n)$.

5.4 Dynamic node and link additions in the Ad-hoc model

We will sketch the analysis of dynamic additions of nodes and links. Consider any new node that is added at time t as a node that wakes up at time t. When a new edge $(u \to v)$ is added at time t there are two cases: (1) u did not report all its edges in response to a query message; (2) u already reported all its edges ($u.local$ is empty). In case (1) the edge can be considered as an edge that has not been reported until time t. In case (2) the new edge causes a *find* operation, since u needs to inform its leader to move it from *done* back to *more* (see end of subsection 4.5.2).

THEOREM 8. *For any dynamic addition of* $\hat{n}$ *new nodes and* $\hat{e}$ *new edges to any network with* $|V| = n$, *the total message complexity from initial state is* $O(m\alpha(m,n))$ *where* $m = n + \hat{n} + \hat{e}$ *is the number of* find *operations needed.*

Thus there is no need to re-run the algorithm each time a new component is added. The additional messages due to network dynamism is almost linear in the number of additional nodes and edges.

6. CONCLUSION

A new message complexity lower bound for asynchronous resource discovery is provided. We present an algorithm that improves the bit complexity of [4]. We explore two open questions raised in [3]: how to deal with networks that are not static, and how detect that termination is possible. We partially answered the first question by providing an algorithm for the Ad-hoc model that efficiently manages dynamic node additions (for node removals, we have initial results where for any k node removals $O(k \log k + n)$ messages are sent). We prove a lower bound for this model, thus showing that our algorithm is asymptotically message optimal. For the second question, we show that termination detection is possible when the network size is known. In addition, we present the first connection known to us between the classic union/find data structure of [10, 11] and distributed algorithms.

Kutten and Peleg [4] describe a wake-up model in which some global broadcast mechanism takes ΔT time to wake-up all nodes, in such a model the time complexity of their algorithm when run in a synchronous settings is $O(\Delta T + \log n)$. Note that in such a model our algorithm's time complexity is $O(\Delta T + n)$, an open question is improving the bit complexity while maintaining the logarithmic time complexity.

7. ACKNOWLEDGEMENTS

The authors would like to thank Shay Kutten for helpful discussions and remarks, and for pointing out that the $O(n)$ message complexity leader election algorithm of [2] may be applied for strongly connected networks.

8. REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. August 2001.

[2] I. Cidon, I. Gopal, and S. Kutten, "New Models and Algorithms for Future Networks", IEEE Transactions on Information Theory, Vol. 41, No.3, pp. 769–780, May 1995.

[3] 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.

[4] S. Kutten, and D. Peleg, Asynchronous Resource Discovery in Peer to Peer Networks. In *21st Symp. on Reliable Distributed Systems*, October 2002 Japan, pp. 224-231.

[5] 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.

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

[7] D. Malkhi, M. Naor and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing* (PODC '02), August 2002.

[8] G. Pandurangan, P. Raghavan and E. Upfal. Building low-diameter p2p networks. In *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, 2001.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the SIGCOMM 2001*, August 2001.

[10] R.E. Tarjan, and J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. In *Journal of the ACM*, 31, 1984, pp. 245-281.

[11] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. In *Journal of Computer and System Sciences*, 18(2):110-127, April 1979.

[12] B. Y. Zhao, J. D. Kubiatowicz and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. U. C. Berkeley Technical Report UCB/CSD-01-1141, April, 2001.