Asynchronous Resource Discovery

Ittai Abraham * Danny Dolev †

January 22, 2004

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 [2] 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 [3]. 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 [2]. We show a reduction from the classic Union-Find problem proving a $\Omega(n\alpha(n,n))$ lower bound for the Ad-hoc Resource Discovery Problem. Thus our Ad-hoc Resource Discovery 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 [2] defined the distributed problem of learning the ids of the nodes of a system and named it *Resource Discovery*. Apparently, the question was important in developing systems like Akamai (http://www.akamai.com).

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 get to 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 [7, 8, 11, 6]). 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 virtual directed graph $G = (V, E)$, where an edge $(u \rightarrow v) \in E$ denotes

*This research was supported in part by NDS PhD fellowship program.School of Engineering and Computer Science. The Hebrew University. Jerusalem, Israel. Email :ittaia@cs.huji.ac.il

†This research was supported in part by Intel COMM Grant - Internet Network/Transport Layer & QoS Environment (IXA). School of Engineering and Computer Science. The Hebrew University. Jerusalem, Israel. Email: dolev@cs.huji.ac.il

that node u knows $id(v)$. A node u can send messages to node v only if $(u \to v) \in E$. 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 $E := E \cup \{(v \to w)\}$.

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.

Recall that two nodes belong to the same weakly connected component if there is a path between them in the induced undirected graph 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.

Resource Discovery A distributed algorithm for Resource Discovery 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 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 [1] may be used. Once a leader is chosen, another $O(n)$ messages are needed to fulfill 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 [1]. For sparse networks in which $|E_0| = O(n)$ this is asymptotically message optimal. Accordingly, the algorithmic challenge for Resource Discovery 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.* [2] 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 [5] presented a randomized algorithm that combined with elements of the algorithm of [2] 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 [2] and of [5] relay heavily on the fact that the number of nodes in the network is known. Kutten, Peleg, and Vishkin [4] 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 [4] 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 [3] studied Resource Discovery 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 Problem Definition

Following Kutten and Peleg [3], 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 a node u to a node v maintain a FIFO (first in first out) ordering when arriving at v.

When defining the asynchronous resource discovery problem there is a subtle issue regarding the detection of termination. Generally, we cannot require an algorithm do detect that it has reached its goal and may terminate. This is true even for the easier problem of distributed leader election on 'knowledge graphs'. A distributed leader election algorithm begins when all nodes are in a leader state and must terminate with exactly one node with leader state and all others with non-leader state. In asynchronous weakly connected networks, leader election algorithms cannot detect that the termination condition is satisfied. 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. Thus we must define the asynchronous resource discovery problem without requiring the participants to detect termination explicitly.

Asynchronous Resource Discovery The following safety requirements should hold 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 liveness requirement:

4. When all nodes are awake, in a state that will never send any more messages, and all message queues are empty, exactly one leader remains in each weakly connected component.

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

We further refine Asynchronous Resource Discovery and define and study three variations:

Oblivious Resource Discovery Nodes do not know the size of their weakly connected component.

Bounded Resource Discovery Every node knows the number of nodes in its weakly connected component.

Ad-hoc Resource Discovery This variation is a novel relaxation of the original problem. An algorithm for Ad-hoc Resource Discovery must have properties (1), (2), and (4) of Asynchronous Resource Discovery, and the following relaxation of property (3):

3a. Each non-leader node has an identified pointer;

3b. These pointers induce a directed path from any non-leader node to its leader.

Note that Ad-hoc Resource Discovery captures the natural case in which at any given time only some of the nodes may need to know the ids of all other nodes. Whenever a node wishes to get all the ids it can prob its leader to get the information.

1.3 Our Results

We present both upper bounds and lower bounds for Asynchronous Resource Discovery and its variations.

Oblivious Resource Discovery We present an $\Omega(n \log n)$ message complexity lower bound. This result shows that Resource Discovery on directed graphs is in a different complexity class than on undirected graphs, for which an $O(n)$ upper bound is known [1].

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 [3].

Bounded Resource Discovery 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 [2].

Ad-hoc Resource Discovery We allow nodes to indirectly know the id of their leader, while still requiring the leader to directly know all ids. This leads to a problem whose message complexity is $\Theta(n\alpha(n,n))$. Notice that this result circumvent the classical lower bound of $\Theta(n \log n)$ messages for choosing a leader. We show that Ad-hoc Resource Discovery is tightly related to the classic Union-Find problem. We prove an $\Omega(n\alpha(n,n))$ message complexity lower bound for this model using reduction techniques.

Our algorithm for Ad-hoc Resource Discovery 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 cost of leader election and reply messages to all the requests is $O((m+n)\alpha(m,n))$.

Another open question raised by [2] 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? For Ad-hoc Resource Discovery, we show that our algorithm efficiently deals with dynamic node and link additions.

2 Message Complexity Lower Bound for Oblivious Resource Discovery

In this section we prove that any algorithm for Resource Discovery 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 that solves Oblivious Resource Discovery 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.

[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 $O(n\alpha(n,n))$.

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 weakly 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 Ad-hoc Resource Discovery.

3 Message Complexity Lower Bound for Ad-hoc Resource Discovery

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

Lemma 3.1 *Given any $h(n)$ message complexity algorithm for Ad-hoc Resource Discovery, 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.

Assume a distributed algorithm for Ad-hoc Resource Discovery 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 Ad-hoc Resource Discovery, 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 [9], 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 that solves Ad-hoc Resource Discovery 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. It is designed for the Oblivious model in which the network size is unknown and the network may not be strongly connected. Later we show how two variations: (1) for the Bounded model, when the network size is known; (2) for Ad-hoc Resource Discovery.

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 'explore' and during its execution may change its state to 'wait' and then 'conqueror', 'passive', 'conquered' or 'inactive'. A diagram with all the state transition is shown in Figure 1. We will call a node *leader* if its state is not 'conquered' or 'inactive' or 'passive'. Thus the state of a leader node is 'explore' or 'wait' or 'conqueror'. 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 unique id. Initially *local* holds the set of ids the node initially knows. The set *more* initially contains the element $\{id\}$, $next = id$, $phase = 1$, the sets *done*, *unaware*, *unexplored*, and the queue *previous* are empty. These data structures and their initial values are shown in Figure 2.

4.1 Finding an unexplored node

A leader node v first needs to find an unexplored node that does not belong to its cluster. The pseudo code for this operation is shown in Figure 3. The set of nodes that are already part of node v's cluster are divided into two sets *more* and *done*. Nodes in the *done* set have already sent to v all the neighbors they know of. Nodes in *more* are nodes that did not send (yet) all the nodes they know of to their leader.

The nodes that are not part of cluster are stored in the *unexplored* set. In order to conquer other nodes, the leader v must search within its cluster for unexplored nodes. If $v.unexplored$ is not empty then just choose any id u from $v.unexplored$ and continue to Section 4.2.

Otherwise, leader v must try to ask nodes in its cluster if they know of any unexplored nodes. If $v.more$ is not empty then v chooses any id w from $v.more$ and sends 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. The pseudo code of an inactive node receiving a 'query' message appears in Figure 5.

If $|v.more| + |v.done| + 1 \geq |w.local|$ then v now knows all the information that w has. In such a case w announces this in its 'query reply' message and v then moves w from $v.more$ to $v.done$.

Since v has only $|v.more|+|v.done|$ ids in its cluster 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).

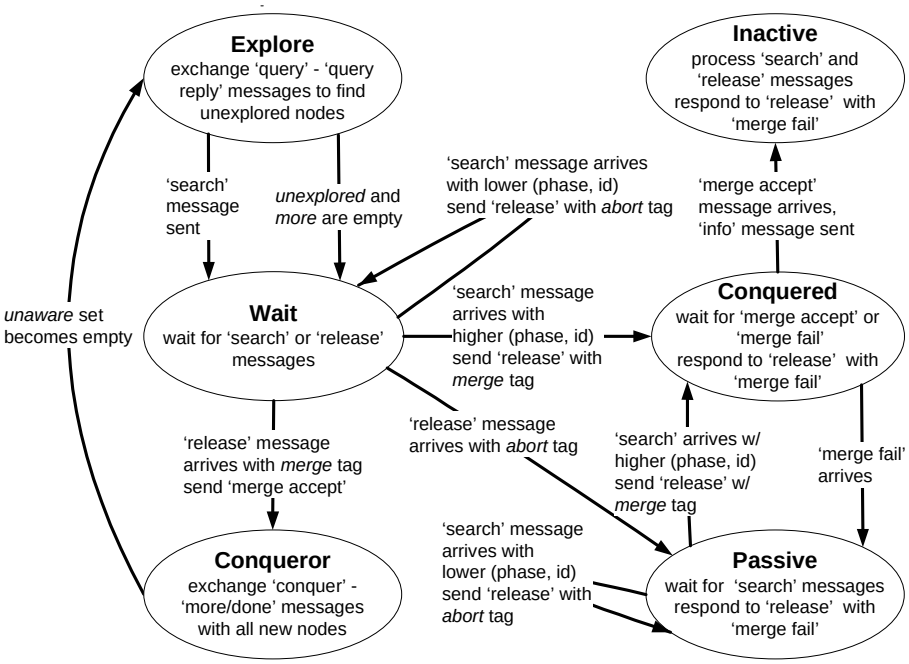

Figure 1: Node's state transition diagram.

Types:
$STATUS = \{explore, wait, passive, conqueror, conquered, inactive\}$;
ID; //the set of all identifiers
$IDS = \{x | x \subseteq ID\}$; //the set of all sets of identifiers
$FIFO$; // a fifo object with operations enqueue,dequeue,peek

Constants:
ID ID; //the node's unique identifier
IDS LOCAL; //the set of identifiers that the node initially knows

Variables and initial values:
$STATUS\ status := explore$;
$ID\ id :=$ ID;
$IDS\ local :=$ LOCAL;
$ID\ next := id$;
$int\ phase := 1$;
$IDS\ unaware := \emptyset,\ unexplored := \emptyset$;
$IDS\ done := \emptyset,\ more := \{id\}$;
$FIFO\ previous$; // initially empty

Figure 2: The data structure of each node.

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 inactive node until the leader is reached. The pseudo code of an inactive node receiving a 'search' message appears in Figure 5. The search message consists of $\langle v.id, v.phase, u.id, false \rangle$. 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

```
EXPLORE:
    if unexplored ≠ ∅ then
        choose u ∈ unexplored;
        unexplored := unexplored \ {u};
        send u a 'search' message ⟨v.id, v.phase, u.id, false⟩;
        status := wait; goto WAIT;
    while (more ≠ ∅) {
        choose w ∈ more;
        send w a 'query' message ⟨|v.more| + |v.done| + 1⟩;
        wait for a 'query reply' message ⟨set, doneFlag⟩;
        if doneFlag == true then
            more := more \ {w};
            done := done ∪ {w};
        if set \ (more ∪ done) ≠ ∅ then
            unexplored := unexplored ∪ (set \ (more ∪ done));
            goto EXPLORE;  };
    status := wait; goto WAIT;
```

Figure 3: Finding an unexplored node for leader node v.

$x.next$ only if M is the only message in the queue, that is $|x.previous| = 1$.

Once the search message is sent, the leader enters a wait loop, waiting for either a 'search' message or a 'release' message, the pseudo code for this stage appears in Figure 4.

When a 'search' message reaches the current leader l of u in a wait loop, a path compression 'release' message containing l will eventually be sent. This message is sent on the opposite direction along the path using the pointers stored in the *previous* queues. The pseudo code of an inactive node receiving a 'release' message appears in Figure 5. When a 'release' message arrives at an inactive node x, node x sets $(M, y) := x.previous.dequeue()$, $(M', y') := x.previous.peek()$. It forwards the 'release' message to y, sets $x.next := l$ and releases the search message M', if M' exists, by forwarding it to $x.next$.

Note that it may be that although v knows u, node u may not know of v, in such a case u needs to update its local set accordingly. Moreover it may be that u's leader l has already put u in its $l.done$ set, in such a case l needs to remove u back to $l.more$. This process allows node l or future leaders that will conquer node l to have the possibility to reach and conquer node v. This is performed in our algorithm by adding an indicator flag as the last field in the 'search' message. A value of *true* indicates that u does not know of v.

4.3 Merging of two leaders

Whenever a 'search' message M initiated at a leader node v reaches a waiting leader l, it needs to decide whether to send a request to merge into v or to abort the merger. This decision is based on which leader has a lexicographically higher (phase,id) pair.

- If $v.phase > l.phase$ or if $v.phase == l.phase$ and $v.id > l.id$ then node l will request to merge. Specifically, node l will send a 'release - merge' message towards v and change its state to 'conquered'. The pseudo code for the conquered state appears in Figure 6.

  When v receives a release message with a *merge* value if may either:

  1. send a 'merge fail' message if its state is 'conquered' (see Figure 6) or 'passive' (see Figure 4) or 'inactive' (see Figure 5).
  2. send a 'merge accept' message if its state is 'wait' (see Figure 4).

  If l receives a 'merge fail' message from v then l sets its state to 'passive'. Once in the passive state, node l will never initiate any new 'search' messages, it will only passively wait until a 'search' message finds it.

```
WAIT:
    wait for message;

    if arrived 'search' message M = ⟨v.id, v.phase, u.id, new⟩ from node y
        if new == true and u ∈ done then
            done := done \ {u};
            more := more ∪ {u};
        if (v.phase > phase) or (v.phase == phase and v.id > id) then
            status := conquered;
            send 'release' message ⟨id, merge, v⟩ to y;
            state := conquered; goto CONQUERED;
        else
            send 'release' message ⟨id, abort, v⟩ to y;
            goto WAIT;

    if arrived 'release' message M = ⟨v, abort, id⟩
        state := passive; goto PASSIVE; //this node will never receive any more 'release' messages

    if arrived 'release' message M = ⟨v, merge, id⟩
        state := conqueror;
        send 'merge accept' message to v;
        state := conquered; goto CONQUEROR;

PASSIVE:
    wait for message;

    if arrived 'search' message M = ⟨v.id, v.phase, u.id, new⟩ from node y
        if new == true and u ∈ done then
            done := done \ {u};
            more := more ∪ {u};
        if (v.phase > phase) or (v.phase == phase and v.id > id) then
            status := conquered;
            send 'release' message ⟨id, merge, v⟩ to y;
            state := conquered; goto CONQUERED;
        else
            send 'release' message ⟨id, abort, v⟩ to y;
            goto PASSIVE;

    if arrived 'release' message M = ⟨l, answer, id⟩
        if answer == merge then send 'merge fail' message to l;
        goto PASSIVE;
```

Figure 4: Pseudo code for wait/passive node l .

Otherwise, if l receives a 'merge accept' message from v then $l.next$ is set to v and l sends an 'info' message M to v containing all the information l has gathered:

$$M = \langle l.phase, l.more, l.done, l.unaware, l.unexplored \rangle .$$

Finally, l becomes a non-leader node by setting its state to '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 a 'release - abort' message is sent to v with value $M = \langle l, abort \rangle$. A leader receiving 'release' message with an *abort* value stops sending new search messages. It will enter the 'passive' state and will loop until some other leader's 'search' message will find and conquer it.

```
INACTIVE:
    forever { wait for message; }

    if arrived 'query' message M = ⟨k⟩ from node v
        if |local| ≤ k then
            send 'query reply' message ⟨local, true⟩ to v;
            local := ∅;
        else
            choose set ⊂ local such that |set| = k;
            send 'query reply' message ⟨set, false⟩ to v;
            local := local \ set;

    if arrived 'search' message M = ⟨v.id, v.phase, u.id, new⟩ from node y
        if id == u.id and v.id ∉ LOCAL then
            local := local ∪ {v};
            M.new := true;
        previous.enqueue(M, y);
        if |previous| == 1 then
            send 'search' message M to next;

    if arrived 'release' message M = ⟨l, answer, v⟩ such that v == id
        if answer == merge then send send 'merge fail' to l;

    if arrived 'release' message M = ⟨l, answer, v⟩ such that v ≠ id
        (M, y) := previous.dequeue();
        next := l;
        send 'release' message M to y;
        if |previous| > 0 then
            (M', y') := previous.peek();
            send 'search' message M' to next;

    if arrived 'conquer' message M = ⟨l⟩
        next := l;
        if local == ∅ then
            send 'done' message to l;
        else
            send 'more' message to l;
```

Figure 5: pseudo code for inactive node x.

4.4 Conquering unaware nodes

If a leader node v receives a 'release - merge' message and it is in the 'wait' state then it enters the 'conqueror' state, responds with a 'merge accept' message, and waits for an 'info' message (see Figure 4 and Figure 6).

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

```
CONQUERED:
    wait for message;

    if arrived 'release' message M = ⟨l, answer, id⟩
        if answer == merge then send 'merge fail' message to l;
        goto CONQUERED;

    if arrived 'merge fail' message
        state := passive; goto PASSIVE;

    if arrived 'merge accept' message M = ⟨v⟩
        next := v;
        send 'info' message ⟨phase, more, done, unaware, unexplored⟩ to v;
        state := inactive; goto INACTIVE;

CONQUEROR:
    do {
        wait for message;
            }   until (unaware == ∅);
                state := explore; goto EXPLORE;

    if arrived 'info' message M = ⟨l.phase, l.more, l.done, l.unaware, l.unexplored⟩
        unaware := unaware ∪ l.more ∪ l.done ∪ l.unaware;
        unexplored := unexplored ∪ l.unexplored \ (l.more ∪ l.done ∪ l.unaware);
        if phase == l.phase or |more| + |done| + |unaware| ≥ 2^(phase+1) then
            phase := phase + 1;
        send 'conquer' message to all nodes in unawere;
        goto CONQUEROR;

    if arrived 'more' message from u ∈ unawere
        more := more ∪ {u};
        unawre := unawre \ {u};
        goto CONQUEROR;

    if arrived 'done' message from u ∈ unawere
        done := done ∪ {u};
        unawre := unawre \ {u};
        goto CONQUEROR;
```

Figure 6: pseudo code for conquered/conqueror node v.

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 (see Figure 5 for the pseudo code of receiving 'conquer' messages).

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 its state to 'explore' and restarts the algorithm's loop by finding a new unexplored node (see Figure 3).

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* data structure 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 Bounded Resource Discovery

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 Ad-hoc Resource Discovery

For Ad-hoc Resource Discovery, 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).

Section 6 discuss how extend the Ad-hoc algorithm to handle online link and nodes addition.

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 of all awaked nodes, and the requirements of Asynchronous Resource Discovery are fulfilled.

Lemma 5.1 *At any stage of execution, at least one node in every weakly connected component remains in a leader state.*

Proof: A node is not in a leader state once it is 'conquered' or 'passive' or 'inactive'. This happens in the following cases: (1) becomes 'passive' when it receives a 'release - abort' message, (2) becomes 'conquered' and later 'passive' when it sends a 'release - merge' message and receives a 'merge fail', or (3) becomes 'conquered' and later 'inactive' when it sends a 'release - merge' message and receives a 'merge accept'. At any given moment, the node with the lexicographically highest (phase, id) pair will always stay in one of the leader states, since it will neither receive an abort, nor become conquered. □

Another concern is that the algorithm would become deadlocked while some nodes have messages in their queues.

Lemma 5.2 *If the algorithm is deadlocked, then there are no messages in any message queue.*

Proof: If the algorithm is deadlocked, then no messages are being sent. Assume that some nodes have non-empty message queue.

Let us look at the graph induced by the *next* pointers. The graph is a forest, since updating the *next* pointer is always done toward a node with a higher *phase*.

Note that no deadlocked node can be in states 'explore' or 'conqueror'. This is true since nodes in these states wait for replies from inactive nodes that immediately answer.

Moreover, no deadlocked node can be in state 'conquered'. Nodes in this state wait for either a 'merge accept' or 'merge fail'. These messages are sent when a 'release - merge' message is received. Notice that the 'release - merge' message sent by the 'conquered' node is never delayed in intermediate 'inactive' nodes. Their return path is maintained by the pointers and the *previous* queue. The message will eventually arrive to the initiator of the appropriate 'search' message, which will answer immediately.

Now, consider a tree in the forest in which one of the nodes has a non-empty message queue. Let v be the node closest to the root with non-empty message queue. Assume first that v is the root. In this case its state cannot be 'inactive', because such nodes have their *next* pointer set to another node. Thus its state should be either 'wait' or 'passive'. But a node cannot be deadlocked with non-empty queues in either state.

Now assume that v is an 'inactive' intermediate node. If its queue is not empty, it is waiting for a reply to a search message it had sent toward its root. None of the intermediate nodes hold any queue, thus the search message should have resulted in a reply of a release message from the root node - a contradiction. □

Lemma 5.3 *If all nodes are awake, then as long as there is more than a single leader node in each weakly connected component, at least one leader's computation is free to continue its execution.*

Proof: Assume that the system is deadlocked and that we have more that a single leader node in some weakly connected component. As proved in Lemma 5.2, leader nodes can be deadlocked only in state 'wait' and in this state there are no pending search messages left.

Since there is more than a single leader node, and all nodes are awake, then there is at least one leader node whose *unexplored* set is not empty. That node, should have invoked a search message. A contradiction. □

Lemma 5.4 *If all nodes are awake, and there is a single leader, then eventually it knows the ids of all the nodes in its weakly connected component.*

Proof: Assume that the single leader node, l, is deadlocked without knowing the ids of some of the nodes in its weakly connected component. Such a deadlock can occur only when all *more* and *unexplored* sets are empty. This implies that all nodes reachable from l are in $l.done$.

Thus, the nodes that l doesn't know off are not reachable from l. Removing $l.done$ from its weakly connected component creates one or more residual weakly connected components. All the edges between these components and $l.done$ are directed towards $l.done$.

Consider such a residual weakly connected component, $G\prime$. $G\prime$ does not contain a leader node. Lemma 5.1 implies that the lexicographically highest node in $G\prime$ must have received a 'release' message from one of the nodes in $l.done$. Such a message should have made the edge it traversed between $G\prime$ and $l.done$ component be bidirectional. A contradiction.

Thus, there is a single leader in each weakly connected component. □

Combining the lemmas we have proven the following:

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

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

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

5.2 Message complexity

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

Lemma 5.5 *The number of 'query' and 'query reply' messages is at most* $4n$.

Proof: 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) move from 'wait' to a non-leader state.

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 each occur at most n times, since a node may conquer at most n other nodes and n nodes may move once from 'wait' to a non-leader state. □

Lemma 5.6 *The number of 'search' and 'release' messages is* $O(n\alpha(n,n))$.

Proof: Each 'search' - 'release' computation simulates a *find* operation. Such computation either causes a *merge* of the leader node with another, or causes the leader node to receive a 'release - abort' message and never to return to a leader state. 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 5.7 *The total number of 'merge accept', 'merge fail', and 'info' messages is at most* $2n$.

Proof: Following a 'release - merge' the sender will never return to a leader state. This occurs at most n times. Each such 'release - merge' message results in a sequence of 'release - merge','merge fail' messages or a sequence of 'release - merge', 'merge accept', and 'info' messages. □

Lemma 5.8 *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.*

Proof: In the Generic Algorithm, each time an inactive node is sent a 'conquer' message its phase increases. Due to the analysis of the Union-Find analogy, the maximum phase of any leader is $logn$. 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. □

The above lemmas implies the following theorems.

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

Similarly to the message complexity analysis of the previous subsection, we bound the total number of bits by bounding the total number of bits for each message type and then sum up on all types.

Lemma 5.9 *The total number of bits in 'query reply' messages is at most* $2|E_0|\log n$.

Proof: 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 5.10 *The total number of bits in 'info' messages is at most* $4n\log^2 n$.

Proof: Every leader maintains the inequality $|more|+|done|+|unaware| < 2^{phase+1}$, and thus $|unexplored| \leq 2^{phase+1}$. So when a node gets conquered, the 'info' message it sends contains at most $2^{i+2}\log n$ bits. Since the number of leader nodes that reach phase i is at most $n/2^i$, summing over all leader nodes by phase results in:

$$\sum_{1\leq i\leq \log n} (n/2^i)(2^{i+2})\log n = 4n\log^2 n.$$

□

Theorem 7 *The bit complexity of the Generic Algorithm is* $O(|E_0|\log n + n\log^2 n)$.

Proof: 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 $O(n\log^2 n)$ □

6 Node and link additions in Ad-hoc Resource Discovery

In this section we extend our Ad-hoc Resource Discovery algorithm to work in a dynamic environment.

In the dynamic model we allow nodes and links to be added to the system while the resource discovery algorithm is still running. Essentially, there is no difference between a node joining the system at a certain time and a node that wakes up at that time. Thus no special operation is performed for node arrivals.

When a link is added, it may require the node to notify its leader of this change. Specifically, when a new edge $(u \to v)$ is added at time t there are two cases to consider:

1. u did not yet report all its edges in response to query messages.
2. u already reported all its edges ($u.local$ is empty).

In the first case, the edge can be considered as an edge that has not been reported until time t and nothing else needs to be done. In the second case, u initiates a *search* message towards its leader with the *new* flag set to true. The message will cause u's leader to move u from *done* back to *more*. This will cause the leader to know that u has more links and to query it in the future.

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 the initial state is* $O(m\alpha(m, n+\hat{n}))$ *where* $m = n + \hat{n} + \hat{e}$.

Proof: Adding $\hat{n}$ new nodes is the same as starting with $n + \hat{n}$ nodes. Adding $\hat{e}$ links causes at most $\hat{e}$ more find operations. Thus the total number of *find* operations is $n + \hat{n} + \hat{e}$. □

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.

7 Conclusion

A new message complexity lower bound for Asynchronous Resource Discovery is provided. We present an algorithm that improves the bit complexity of [3]. We explore two open questions raised in [2]: how to deal with networks that are not static, and can termination be detected. We partially answered the first question by providing an algorithm for Ad-hoc Resource Discovery that efficiently manages dynamic node and link additions. We prove a message complexity lower bounds for Oblivious Resource Discovery and for Ad-hoc Resource Discovery, 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, 9] and distributed algorithms.

Kutten and Peleg [3] 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.

Another interesting remaining open question is how to ideal efficiently with dynamic nodes' removals. This topic is related to increasing the robustness of Resource Discovery.

8 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 [1] may be applied for strongly connected networks.

References

[1] I. Cidon, I. S. Gopal, and S. Kutten. New models and algorithms for future networks. In *Symposium on Principles of Distributed Computing*, pages 79–89, 1988.

[2] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 229–237. ACM Press, 1999.

[3] S. Kutten and D. Peleg. Asynchronous resource discovery in peer to peer networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 224–, 2002.

[4] S. Kutten, D. Peleg, and U. Vishkin. Deterministic resource discovery in distributed networks. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 77–83. ACM Press, 2001.

[5] C. Law and K. Siu. An o(log n) randomized resource discovery algorithm. In *Brief Announcements of the 14th International Symposium on Distributed Computing, Technical Report, Technical University of Madrid*, pages 5–8.

[6] 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)*, pages 183–192, 2002.

[7] 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*, 2001.

[8] 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*, 2001.

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

[10] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[11] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.