Neighbor Discovery in Mobile Ad Hoc Networks Using an Abstract MAC Layer

Alejandro Cornejo* Nancy Lynch† Saira Viqar‡ Jennifer L. Welch§

November 20, 2009

Abstract

We explore the problem of neighbor discovery in a mobile ad hoc network environment. We describe a protocol for learning about neighboring nodes in such an environment. The protocol is used for establishing and tearing down communication links with neighboring nodes as they move from one region of the network to another. The protocol is implemented on top of the abstract MAC layer service presented in [4], which provides reliable message delivery within the local neighborhood and also provides the sender with an acknowledgment when all neighboring nodes have received a message. There is an upper bound, guaranteed by the abstract MAC layer service, on the worst case delay that a message can experience before it is received or acknowledged. We determine the time complexity of the neighbor discovery protocol in terms of the bounded delays provided by the underlying abstract MAC layer.

*MIT CSAIL, acornejo@csail.mit.edu
†MIT CSAIL, lynch@csail.mit.edu
‡Texas A&M, viqar@cse.tamu.edu
§Texas A&M, welch@cse.tamu.edu

1 Introduction

Neighbor discovery is an important aspect of many algorithms in mobile wireless ad hoc networks (cf. [9], [8], [1]). For example, knowledge about neighboring nodes can be used to route, cluster and broadcast in an efficient manner.

Neighborhood knowledge is assumed in many routing protocols used in wireless sensor networks. For example in [9] the authors assume that nodes know the location of one- and two-hop neighbors. This information is used to implement a coordinate based routing algorithm. In [8] nodes are assumed to maintain information about their one-hop neighbors in order to perform routing in multi-hop wireless networks. In [1] the authors assume that each node knows its own location and its neighbors' locations, in order to develop a locality-aware location service.

We wish to take advantage of the reliability of the abstract MAC layer described in [4] to design an efficient neighbor discovery protocol. The abstract MAC layer hides the lower level details of collision detection and contention while providing bounds on the amount of delay incurred in the reception of a message and the receipt of an acknowledgment.

We assume that the network is divided into static regions and this division is known to all the nodes. If a node that enters a particular region of the network and remains there for sufficiently long, it should learn about and establish communication links with nodes which are in regions up to k-hops apart. If a node leaves a particular region, other nodes that no longer lie within its neighborhood should be notified and should be able to take down their communication links with the leaving node.

We also assume that each node can query its trajectory information for some constant time into the future (i.e. it can predict where it will go in the immediate future). Based on this information, and the bounds provided by the abstract MAC layer, we give a protocol in which nodes exchange notification messages at appropriate times, before exiting a particular region or upon entering a new region of the network. The protocol allows nodes to gain information about neighbors and exchange application messages reliably with neighbors. We also give a proof of correctness for our protocol.

2 Related Work

A deterministic distributed algorithm for neighbor discovery is suggested in [2] and uses TDMA slots. It has a running time of $MN/r + O(\max(M, N) \log r)$. It is assumed that there are n nodes and each node is assigned a unique identifier from the range $[1, N]$. M is the maximum number of channels available for communication, or possible channels all the nodes are capable of operating on, and r is the number of receivers at a node. It is suggested that the running time is so large because of the oblivious nature of the algorithm (which means that a node transmits based solely on its label and the time slot number). Our neighbor discovery protocol is different from the one given above since it is built on top of an underlying reliable MAC layer.

A protocol for secure neighbor discovery in the presence of compromised nodes is given in [5]. The protocol achieves secure discovery of the local neighborhood by taking advantage of the sensor deployment phase. It is assumed that sensor nodes can be trusted for a short time after deployment. This period of time is used to ensure that neighborhood information is not compromised. The protocol also takes advantage of the fact that usually neighboring nodes have a large number of common neighbors. Although the protocol tries to handle malicious nodes, it assumes that nodes remain static and do not change their location after they have been deployed. Our neighbor discovery protocol deals with mobile nodes which can move from region to region.

In [10] the authors give a neighbor discovery algorithm which is similar to ALOHA. The algorithm works without a collision detection mechanism. An extension of the algorithm is given which works in the absence of clock synchronization. Nodes are allowed to wake up at different times. The authors show that each node is able to find out about all its neighbors in expected time $ne(\log n + c)$ where c is constant. However, it is assumed that all n nodes form a clique throughout the execution of the algorithm, whereas in our algorithm the neighborhood topology need not be a complete graph and in fact can change over time.

3 System Model

The timed I/O automata modeling framework [11] is used in order to model the mobile ad hoc network. There are six components in the system: the *network layer automaton*, the *abstract MAC layer automaton*, the *queue layer automaton*, the *neighbor discovery layer automaton*, the *point-to-point layer automaton*, and the *user automaton* (see Figure 3.3). We give a description here of these components.

3.1 The Network Layer Automaton

The network layer automaton models the real world in terms of time, location, physical layer behavior and it also encapsulates mobility of nodes. It is assumed that location and time are accurately provided by the network layer.

For every network automaton there is a function f_G that maps from states to directed node interaction graphs. Fix an execution α of the MANET system. Suppose s gives the state of the network at some point in α. $G_{comm} = f_G(s)$ is the dynamic *directed communication graph* which captures nodes in communication range in network state s. Assume that the position of node i in state s is given by $pos(i)$. Then there is an edge (i, j) between nodes i and j in G_{comm} if and only if the Euclidean distance between $pos(i)$ and $pos(j)$ in state s is less than or equal to the broadcast radius of node i. At any point in α, E is the edge set corresponding to $G_{comm} = f_G(s)$ where s is the network state at that point. Note that two nodes may have different broadcast ranges. Let r_{min} be the minimum broadcast range among all the nodes.

Fix R to be a closed, bounded and connected subset of $\mathbb{R}^2$. R models the physical space in which the nodes reside; we call it the deployment space. Let U be the index set for regions in R used by the participating agents. We now define a region partitioning scheme.

Definition 1. *A region partitioning scheme divides R into a set of regions $\{R_u\}_{u \in U}$ such that: (i) each $u \in U$, R_u is a connected subset of R, (ii) for any $u, v \in U$, R_u and R_v are disjoint, (iii) the deployment space is equal to the union of all regions, $R = \bigcup_{u \in U} R_u$.*

For any $u, v \in U$, R_u and R_v are neighboring regions if there exists a closed linear trajectory that starts at R_u and ends in R_v and does not pass through any other regions.

We refer to the graph induced by the neighborhood relation of the region partition scheme as the *region graph*. We say region X and region Y (or node a in region X and node b in region Y) are k-hops apart if the shortest path between X and Y in the region graph is of length k. Throughout we assume the maximum distance between two points which are in regions k-hops apart is bounded by r_{min}. Finally each node has access to the function $getregion : \mathbb{R}^2 \to U$ which maps any point in the plane to a region.

3.2 The Abstract MAC Layer Automaton

The MAC layer automaton provides reliable message delivery to all recipients as well as feedback to the sender in the form of an acknowledgment which indicates that the message has been delivered to all intended receivers. It provides guaranteed time bounds on message delivery as well as the receipt of acknowledgments. These time bounds are functions of the current level of contention. The cost of implementing this abstract MAC layer exactly as described might be prohibitively large. However, it is possible to provide similar guarantees with a high probability.

The MAC layer provides the following interface actions $bcast(m)_i$, $abort(m)_i$, $rcv(m)_i$, $ack(m)_i$. The first two are input actions and the other two are output actions. In addition it imposes upper bounds on the time elapsing between $bcast(m)_i$ and corresponding $ack(m)_i$ and $rcv(m)_j$. These bounds depend on the contention involving the sending node, denoted above by i, and the receiver node, denoted above by j, during the broadcast interval. These time bounds can be summarized as follows:

- F_{rcv}^+ : upper bound on a specific message being delivered.
- F_{ack}^+ : upper bound on an acknowledgment being received.

These functions are monotonically non-decreasing with the level of contention present at the receiver or both the sender and receivers (for F_{ack}^+).

We assume that the values of the time bounds F_{rcv}^+ and F_{ack}^+ (described in [4]) are constant and available to algorithms implemented on top of the abstract MAC layer. Thus a node can use these bounds to determine when to transmit notification messages when entering or leaving a geographic region. Note that this implies that the dynamic communication graph (G_{comm}) induced by the motion of the nodes has a constant upper bound on the maximum degree.

The MAC layer assumes some well-formedness conditions for upper layers. In particular, it assumes that a user process does not submit a *bcast* until after its previous *bcast* has had a matching *ack* returned. There are constraints on message behavior. In particular, if a $bcast(m)_i$ event causes a $rcv(m)_j$ event, then at some point between these events nodes i and j have to be within interference range. If a $bcast(m)_i$ event causes an $ack(m)_i$ event and for every point in between these two event nodes i and j are in communication range, then a $rcv(m)_j$ caused by the *bcast* is guaranteed to precede the *ack*.

3.3 The Queue Layer Automaton

The queue layer automaton has inputs $bcast\_msg\_ndp(m)_i$, $bcast\_msg\_app(m)_i$, $ack(m)_i$, and output $bcast(m)_i$.

The queue layer automaton provides a message queue at each node which ensures that a node does not submit a *bcast* request to the abstract MAC layer until after its previous *bcast* has ended with a matching *ack* being returned. This is part of the well-formedness constraints placed on the upper layers using the abstract MAC layer service. The queue layer automaton actually provides two queues for each node. The application queue buffers messages from applications and the NDP queue buffers messages from the neighbor discovery layer. Preference is given to messages received from the neighbor discovery layer, which are broadcast first, even if there are pending application messages. The maximum size of the queues is fixed (given by k). It is assumed that messages are received from the application layer at a rate such that the queues do not overflow. Both the neighbor discovery layer and the user layer should be such that the number of messages they send does not overflow the queue, which is emptied at a rate of one element per F_{ack}^+ time units.

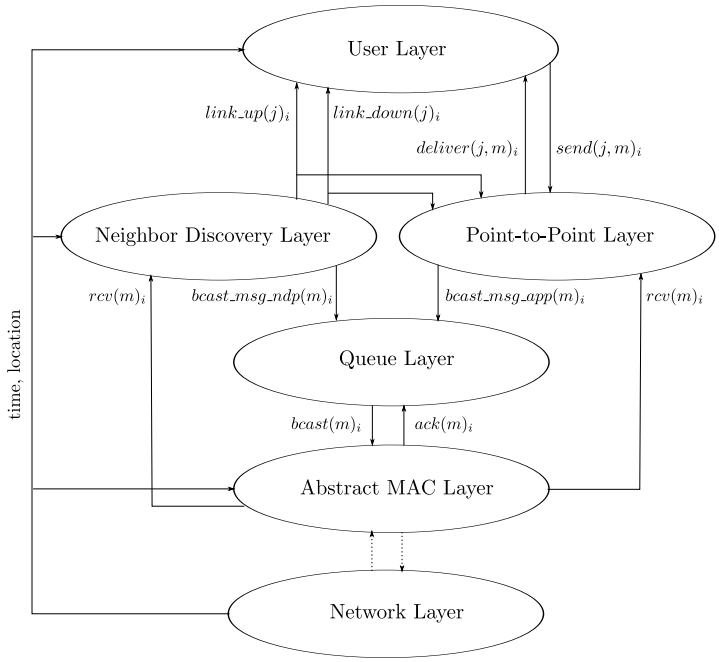

Figure 1: The MANET system.

3.4 The Neighbor Discovery Layer Automaton

The neighbor discovery layer automaton for node i defines three output actions, $bcast(m)_i$, $link\_up(j)_i$ and $link\_down(j)_i$ where $j \neq i$; in the following discussion we ignore $bcast(m)_i$ and focus on $link\_up(j)_i$ and $link\_down(j)_i$. The $link\_up(j)_i$ action signals that a *reliable* communication link has been established between node i and j *from the perspective of node* i. Similarly the $link\_down(j)_i$ action signals that a previously established communication link between node i and j is no longer available *from the perspective of node* i.

Consider any execution α, and let α_i be the projection of α onto the actions of node i, we impose the following restrictions on the executions.

Well-Formedness:

- For all i and j the $link\_up(j)_i$ and $link\_down(j)_i$ actions alternate in α_i.

The *link_up* and *link_down* events induce a *directed neighbor graph* G_{neigh} with vertex set equal to the node set. For any two nodes i and j, the directed edge (i, j) is in G_{neigh} if and only if the most recent link event at node i for node j is a *link_up*. If directed edge (i, j) is present in G_{neigh} we say its in the Up state, otherwise we say it is in the Dn state. We now define some synchronization conditions.

Synchronization:

1. While edge (i, j) is Up the edge (j, i) cannot go through the states $Up \rightarrow Dn \rightarrow Up$.
2. While edge (i, j) is Dn the edge (j, i) cannot go through the states $Dn \rightarrow Up \rightarrow Dn$.

To avoid the trivial solution where all edges remain Dn independent of the environment we define a progress condition.

Progress:

- There exist constants $a, b \in \mathbb{R}^+$, such that for all times times t_1 and t_2 where $t_2 \geq t_1 + a + b$, and for any nodes i and j: if i is in region X and j is in region Y throughout $[t_1, t_2]$, where X and Y are k-hops apart, the directed edges (i, j) and (j, i) are in G_{neigh} (that is they are in state Up) during the time interval $[t_1 + a, t_2 - b]$.

Similarly we now need a validity condition to avoid solutions where all edges are kept in the Up state independent of the environment.

Validity:

- If (i, j) is present in G_{neigh} (that is it is in state Up) then nodes i and j are in regions which are k-hops apart (and thus they are within distance r_{min}).

3.5 The Point-to-Point Layer Automaton

The point-to-point layer automaton has interface actions $send(j, m)_i$, and $deliver(j, m)_i$, which allow higher layers to send and receive point-to-point messages. It connects to the underlying neighbor discovery layer by receiving the neighbor discovery layer's $link\_up(j)_i$ and $link\_down(j)_i$ outputs as inputs.

Thus we have the following output action:

- $deliver(j, m)_i$ (node i receives message m from node j), for all i and j, $i \neq j$.

We have one input action:

- $send(j, m)_i$ (node i sends message m to node j), for all i and j, $i \neq j$.

Fix an execution α of the MANET system. We assume the existence of a *caused-by* function mapping every $deliver(i, m)_j$ event to a preceding $send(j, m)_i$ event, $i \neq j$. Below are additional constraints on the nature of the caused-by function.

Constraints on Message Behavior:

1. No duplicate receives: The caused-by function is one-to-one. That is, each *send* event causes at most one *deliver* event.

2. Termination: The caused-by function is onto. That is, each *send* event causes at least one *deliver* event.

3.6 The User Automaton

The user automaton is a composition of separate (and non-interacting) automata for the users $\{1, \ldots, n\}$. User i connects to the underlying neighbor discovery layer using the given interface.

Fix an execution α of a MANET system. The following properties define well-formedness:

1. α contains at most one *send* event for each message m.

2. If $send(j, m)_i$ occurs then $(i, j) \in G_{neigh}$ (and the link (i, j) is Up).

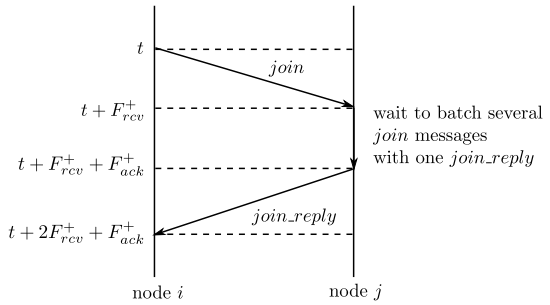

Figure 2: The maximum time required for setting up a link.

4 The Neighbor Discovery Protocol

The neighbor discovery protocol is based on nodes sending notification messages tagged with their UID and their current region whenever they enter or leave a region. In particular, there are three types of messages:

- *leave*
- *join*
- *join_reply*.

When a node is about to move into a new region, it broadcasts a *leave* message some time before leaving. This message indicates to its neighboring nodes that they should begin tearing down the corresponding link if appropriate. When a node enters a new region and determines that it is going to remain there for sufficiently long, it broadcasts a *join* message. This message indicates to the neighbors that they should start setting up the corresponding link if they don't have one already. It also serves as a request to learn the ids of neighbors. Nodes that receive a *join* message send a *join_reply* message in response so that the original node can learn their ids. The timing of these messages ensures that the proper semantics of the corresponding links are maintained. This means that the overhead for setting up and tearing down links is taken into account, and reliable message delivery is guaranteed when a link is in the Up state.

Suppose that the time overhead for setting up a link between two neighbors is given by δ_{LU}, and the time overhead for tearing down a link is given by δ_{LD}. A node broadcasts a *join* message upon entering a new region only if it is going to remain there for at least the amount of time required to set up a link and to tear it down. Thus a node broadcasts a *join* message if it is going to remain in its new region for at least $\delta_{LU} + \delta_{LD} + L$ time in the future where $L \geq 0$ is an application provided parameter.

The exact time overhead for setting up a link (δ_{LU}) can be determined in terms of the delays provided by the underlying MAC layer. This is the overhead incurred in sending the *join* message and getting back the corresponding *join_reply*. After this the link has been set up and application messages can be sent over it. Thus $\delta_{LU} = 2F^+_{rcv} + F^+_{ack}$ (see Figure 2).

- The first F^+_{rcv} time units are to allow the *join* message to get from the sender to the receiver. When the receiver gets the *join* message it will perform a *link_up* with the sender.
- It takes another F^+_{ack} time units for the receiver to process the *join* message. This is because when a node receives a *join* message it waits before broadcasting the corresponding

join_reply. It does so in order to process multiple *join* messages in batches. This prevents the receiver from being swamped with pending *join* messages. Consider the following scenario. Suppose that node i is present in region X of the network. Now suppose that $n-1$ nodes move into region X and send *join* messages. Node i will then have to send $n-1$ *join_replies*. This will result in overflow in the NDP message queue in the Queue layer. Thus i waits for F^+_{rcv} time and collects the *join* messages and responds with one *join_reply*. An interval of F^+_{ack} is used to also guarantee wellformedness (no more than one message every F^+_{ack} units of time is sent by the NDP layer).

- The last F^+_{rcv} units of time ensure that the *join_reply* gets back to the original node which sent the *join* message. After getting this *join_reply* the original node performs a *link_up* with the sender of the *join_reply*.

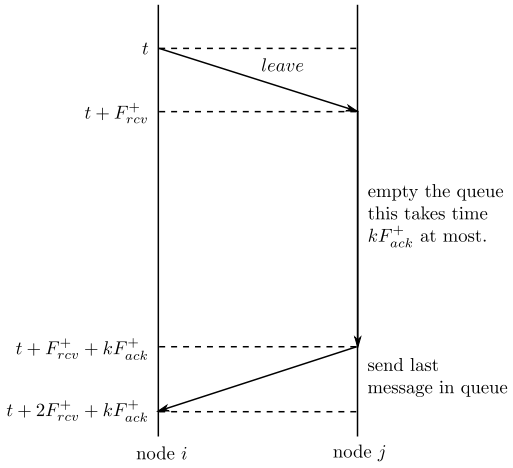

Figure 3: The maximum time required for taking down a link.

The overhead for tearing down a link (δ_{LD}) can similarly be determined in terms of the delays provided by the MAC layer and the size of the application message queue. This time bound guarantees that all neighbors get the *leave* message sent by a node before it leaves so that the neighbors have accurate information at all times about who their own neighbors are. The delay of δ_{LD} also ensures that the leaving node receives messages that were in transit when it sent the notification. The sender of a *leave* message performs a *link_down* with all its neighbors as soon as it send the *leave* message. Specifically $\delta_{LD} = 2F^+_{rcv} + kF^+_{ack}$ (see Figure 3).

- The first F^+_{rcv} time units allow the *leave* message to get to the receiver. At this point the receiver performs a *link_down* with the sending node. After this the receiver will not send any more messages. However, it will still receive messages that are already in the queue at the sender.

- The next kF^+_{ack} time units allow the receiver to empty out application messages in the queue. Note that the maximum queue size is given by k and each message can incur a maximum delay of F^+_{ack} before it is sent.

- The remaining F^+_{rcv} time units allows the last message in he queue to reach the sender.

When a node i receives a *join* message it checks if it is going to remain in its current region long enough to send a *join_reply* back and then tear down its link with the initiator of the *join*

message. Thus i checks if it is going to remain in its current region for at least the next $\delta_{LU} + \delta_{LD}$ time units.

In the neighbor discovery protocol, nodes include their ids and their current region in notification messages. For ease of exposition we assume the existence of a function $hops : \{R_u\}_{u \in U} \times \{R_u\}_{u \in U} \rightarrow \mathbb{N}$ which receives two regions and returns the number of hops between them, if one of the regions is null it returns ∞.

5 TIOA Code

We describe the algorithms using the TIOA formalism [11]. In the neighbor discovery protocol we assume *TIOA trajectory* that stops time whenever a precondition is enabled.

Algorithm 1 Neighbor Discovery Protocol

automaton NDP(i:$\mathbb{N}$, traj:Traj, F^+_{ack}:$\mathbb{R}$, L:$\mathbb{R}$, k:$\mathbb{N}$, δ_{LU}:$\mathbb{R}$, δ_{LD}:$\mathbb{R}$)
states
active:Bool := $false$;
sendbuffer:Seq[M] := $\varnothing$;
recvbuffer:Seq[M] := $\varnothing$;
eventqueue:Seq[Ev] := $\varnothing$;
S:Set[$\mathbb{N}$] := $\varnothing$;
regs:Map[$\mathbb{N}$, Region] := $empty$;
curreg:Null[Region] := nil;
newreg:Null[Region] := nil;
jointrigger:$\mathbb{R}$:= -1;
now:$\mathbb{R}$:= 0;

transitions
output bcast(m, i)
pre $m = head(sendbuffer)$
eff
sendbuffer := $tail(sendbuffer)$;

input rcv(m, i)
eff
recvbuffer := $recvbuffer \vdash m$;

internal enter_region(i)
pre $eventqueue = \varnothing \wedge getregion(traj_{now}) \neq val(curreg)$
eff
curreg := $embed(getregion(traj_{now}))$;
if $\forall t : \mathbb{R}(t \geq now \wedge t \leq now + \delta_{LU} + L + \delta_{LD} \Rightarrow getregion(traj_t) = val(curreg))$ **then**
sendbuffer := $sendbuffer \vdash [[join, val(curreg), nil], i]$;
active := $true$;

internal leave_region(i)
pre $eventqueue = \varnothing \wedge active \wedge getregion(traj_{now+\delta_{LD}}) \neq val(curreg)$
eff
newreg := $embed(getregion(traj_{now+\delta_{LD}}))$;
active := $false$;
if $\exists t : \mathbb{R}(t \geq now + \delta_{LD} \wedge t \leq now + \delta_{LD} + \delta_{LU} + L + \delta_{LD} \Rightarrow getregion(traj_t) \neq val(newreg))$ **then**
newreg := nil;
sendbuffer := $sendbuffer \vdash [[leave, val(curreg), newreg], i]$;
for j **in** S
if $hops(regs_j, val(newreg)) > k$ **then**
eventqueue := $eventqueue \vdash [down, j, regs_j]$;

internal process_message(m, i)
pre $eventqueue = \varnothing \wedge m = head(recvbuffer) \wedge getregion(traj_{now}) = val(curreg)$
eff
recvbuffer := $tail(recvbuffer)$;
if $m.sender \in S$ **then**
regs := $update(regs, m.sender, m.msg.reg)$;
if $hops(m.msg.reg, val(curreg)) \leq k$ **then**
if $m.msg.type = join \wedge m.sender \notin S \wedge \forall t : \mathbb{R}(t \geq now \wedge t \leq now + \delta_{LU} + \delta_{LD} \Rightarrow$
$getregion(traj_t) = val(curreg))$ **then**
if $jointrigger = -1$ **then**
jointrigger := $now + F^+_{ack}$;
eventqueue := $eventqueue \vdash [up, m.sender, m.msg.reg]$;
if $m.msg.type = leave \wedge m.sender \in S \wedge hops(val(m.msg.dest), val(curreg)) > k$ **then**
eventqueue := $eventqueue \vdash [down, m.sender, regs_{m.sender}]$;
if $m.msg.type = join\_reply \wedge m.sender \notin S \wedge active$ **then**
eventqueue := $eventqueue \vdash [up, m.sender, m.msg.reg]$;

internal send_join_reply(i)
pre $eventqueue = \varnothing \wedge jointrigger = now$
eff
jointrigger := -1;
sendbuffer := $sendbuffer \vdash [[join\_reply, val(curreg), nil], i]$;

output link_down(j, i)
pre $\exists reg : Region(head(eventqueue) = [down, j, reg])$
eff
S := $S - \{j\}$;
regs := $remove(regs, j)$;
eventqueue := $tail(eventqueue)$;

output link_up(j, i)
pre $\exists reg : Region(head(eventqueue) = [up, j, reg])$
eff
S := $S \cup \{j\}$;
regs := $update(regs, j, head(eventqueue).reg)$;
eventqueue := $tail(eventqueue)$;

Algorithm 2 Queue

automaton Queue(i:$\mathbb{N}$, k:$\mathbb{N}$, M:**Type**)
states
ptp_queue:Seq[M] := $\varnothing$;
ndp_queue:Seq[M] := $\varnothing$;
cts:Bool := $true$;

transitions
input bcast_msg_ptp(m, i)
eff
if $len(ptp\_queue) < k$ **then**
ptp_queue := $ptp\_queue \vdash m$;

input bcast_msg_ndp(m, i)
eff
if $len(ndp\_queue) < k$ **then**
ndp_queue := $ndp\_queue \vdash m$;

```
    output bcast(m, i)
        pre cts = true ∧ m = head(ndp_queue) ∨ m = head(ptp_queue) ∧ ndp_queue ≠ ∅
        eff
            cts := false;

            if len(ndp_queue) > 0 then
                ndp_queue := tail(ndp_queue);
            else
                ptp_queue := tail(ptp_queue);

    input ack(m, i)
        eff
            cts := true;
```

Algorithm 3 Point to Point Algorithm

```
automaton PTP(i:ℕ, P:Type)
    states
        sendbuffer:Seq[M] := ∅;
        recvbuffer:Seq[M] := ∅;
        S:Set[ℕ] := ∅;

    transitions
        input link_down(j, i)
            eff
                S := S − {j};

        input link_up(j, i)
            eff
                S := S ∪ {j};

        input send(j, p, i)
            eff
                if j ∈ S then
                    sendbuffer := sendbuffer ⊢ [p, i];

        output deliver(j, p, i)
            pre [p, j] = head(recvbuffer)
            eff
                recvbuffer := tail(recvbuffer);

        input rcv(m, i)
            eff
                recvbuffer := recvbuffer ⊢ m;

        output bcast_msg_app(m, i)
            pre m = head(sendbuffer)
            eff
                sendbuffer := tail(sendbuffer);
```

Technical Considerations

Since regions were defined as disjoint, technically there might not exist a "first" time when a node enters or leaves a region due to possible left-open intervals.

Concretely, the *enter_region* action requires a trajectory that stops time when a node first changes regions. To side step this problem we allow some slack in the stopping conditions; unfortunately this requires changing the definitions of the partitioning scheme to allow neighboring regions to overlap at their boundaries. We define an overlapping partition as follows:

Definition 2. *Fix R to be a closed, bounded and connected subset of $\mathbb{R}^2$. R models the physical space in which the nodes reside; we call it the deployment space. Let U be the index set for regions in R used by the participating agents. A region partitioning scheme divides R into a set of regions $\{R_u\}_{u \in U}$ such that: (i) each $u \in U$, R_u is a closed and connected subset of R, (ii) for any $u, v \in U$, R_u and R_v may overlap only at their boundaries.*

For any $u, v \in U$, R_u and R_v are neighboring regions if they intersect at their boundaries ($R_u \cap R_v \neq \varnothing$). Let the region graph *be the graph induced by the neighborhood relation on the set of regions.*

Observe that the *getregion* function would now a set of regions instead of a single region. In particular, when queried in a region boundary it returns the set of regions that share the boundary, otherwise it returns a singleton set with the current region. The algorithm itself would remain unchanged modulo some simple changes to handle the fact that *getregion* returns a set and not a single element which we omitted for readability. However the stopping condition in the TIOA trajectory for the *enter_region* action would become:

$$\begin{aligned} \exists u : Region\ (u \neq val(region)) \wedge \\ curreg \notin getregion(traj_{now}) \wedge u \in getregion(traj_{now}) \wedge \\ curreg \in getregion(traj_{now-\varepsilon}) \wedge u \in getregion(traj_{now-\varepsilon}) \end{aligned}$$

Here $\varepsilon > 0$ is a small constant describing the slack, and it depends on the motion of the agents with respect to the size of the regions. Observe that the same discussion applies for the *leave_region* action, and a similar predicate can be used for its TIOA stopping condition.

6 Proof of Correctness

Lemma 1. *The neighbor discovery algorithm satisfies the well-formedness condition.*

Proof. We have to show that for all i and j, $link\_up(j)_i$ and $link\_down(j)_i$ alternate.

Consider nodes i and j. Suppose that at time t node i performs a $link\_up(j)_i$. This means that node j is now in it's neighbor set. Node i can now perform another $link\_up(j)_i$ before a $link\_down(j)_i$ only if it receives a *join* or a *join_reply* message. In both cases it first checks if j is already in the neighbor set, and does not carry out a $link\_up(j)_i$ if it is.

Now suppose that node i performs a $link\_down(j)_i$ at time t'. This means that j is removed from the neighbor set. It can only perform another $link\_down(j)_i$ if it performs a *leave_region* action or it gets a *leave* message. For both cases it checks its neighbor set to see if j is present in it before doing a $link\_down(j)_i$.

□

Proposition 1. *A join message is always received before its corresponding leave message. The same holds for join and join_reply messages.*

Proof. If a *join* message is sent at time t the corresponding *leave* message will be sent at time $t + \delta_{LU}$ at the earliest. The MAC layer guarantees message delivery after F^+_{rcv}, an since $\delta_{LU} > F^+_{rcv}$ the *leave* message will be sent (and thus delivered) after the *join* message is delivered.

Similarly after a *join* message is sent, a *joint_reply* message cannot be sent before waiting F^+_{ack} in the send trigger, and thus by the time the *join_reply* message is sent the *join* message would have been delivered. □

This property is assumed throughout the rest of the proofs.

Lemma 2. *While (i, j) is Up, (j, i) cannot go through the states $Up \to Dn \to Up$.*

Proof. Fix nodes i and j where the directed edges (i, j) and (j, i) are both in the Up state. Suppose the edge (j, i) switches to the Dn state at time t while the edge (i, j) remains Up. The state change $Up \to Dn$ of edge (j, i) was caused when node j executed a *leave_region* action or processed a *leave* message sent by i.

It suffices to show that in either case the edge (j, i) can't switch back to Up before the edge (i, j) switches to the Dn state.

- If node j executed *leave_region* at time t it also sent a *leave* message at time t, and moreover when it compared its new region to the current region of node i it determined they were more than k hops apart. Node i receives this message at time $t \le t' \le t + F^+_{rcv}$, and since it will perform the same comparison it will change the state of edge (i, j) to Dn. Moreover the edge (j, i) cannot switch back to Up before $t'' = t + \delta_{LD}$, and since $\delta_{LD} > F^+_{rcv}$ then $t'' > t'$ and the statement follows.

- Suppose node j processed a *leave* message at time t and changed the edge to Dn, therefore when it compared the new region of node i to its current region it determined they were more than k hops apart. This message was sent by node i at time $t - F^+_{rcv} \le t' \le t$, and since it made the same comparison then at time t' the edge (i, j) was Dn. Moreover the edge (i, j) cannot switch back to the Up state before time $t' + \delta_{LD}$ and since $\delta_{LD} > F^+_{rcv}$ this contradicts the assumption that edge (i, j) was up at time t.

□

The proof of the second synchronization condition follows the same vein, but with some subtle differences.

Lemma 3. *While (i, j) is Dn (j, i) cannot go through the states $Dn \to Up \to Dn$.*

Proof. Fix nodes i and j where the directed edges (i, j) and (j, i) are both in the Dn state. Suppose the edge (j, i) switches to the Up state at time t while the edge (i, j) remains Dn. The state change $Dn \to Up$ of edge (j, i) was triggered when node j received a *join* or a *join_reply* message from node i. It suffices to show that in either case the edge (j, i) can't switch back to Dn before the edge (i, j) switches to the Up state.

- If node j processed a *join* message at time t, then node i will receive the corresponding *join_reply* message at time $t \le t' \le t + F^+_{rcv} + F^+_{ack}$ and will switch edge (i, j) to the Up state. Moreover the edge (j, i) cannot switch back to a Dn state before $t + \delta_{LU} > t'$ and the statement follows.

- Suppose node j processed a *join_reply* message at time t sent by node i at time $t - F^+_{rcv} \leq t' \leq t$. Let $\tau_i, \tau_j \leq t$ be the times at which nodes i and j executed the *enter_region* action, observe that by construction we know that node i will not execute the *leave_region* action until $t + \delta_{LU} - F^+_{ack} - F^+_{rcv} = t + F^+_{rcv}$ as the earliest.

  - If $\tau_j \leq \tau_i$ then the *join* message of node i would reach j before the *join_reply* message and while it is active, switching the edge (j, i) to the Up state. However this contradicts the assumption that the reception of a *join_reply* triggered the $Dn \rightarrow Up$ state change of edge (j, i).
  - If $\tau_i < \tau_j$ then the *join* message of node j would reach i before $\tau_j + F^+_{rcv}$ and while node i is active, switching the edge (i, j) to the Up state, moreover recall that node i won't switch edge (j, i) to Dn until $t + F^+_{rcv}$ as the earliest. On the other hand, node j switched the edge (i, j) to Up at time t and won't switch edge (i, j) to Dn until $\tau_j + \delta_{LU}$ at the earliest. Since $\tau_j + \delta_{LU} < \tau_j + F^+_{rcv}$ and $t < t + F^+_{rcv}$ the statement follows.

□

From the previous two lemmas, the following is a corollary.

Theorem 1. *The neighbor discovery algorithm satisfies the synchronization conditions.*

W now prove the neighbor discovery protocol satisfies the progress condition of the neighbor discovery specification with $a = \delta_{LU}$ and $b = \delta_{LD} + L$ for any $L \geq 0$.

Theorem 2. *The neighbor discovery algorithm satisfies the progress condition.*

Proof. Suppose that $a = 2F^+_{rcv} + F^+_{ack} = \delta_{LU}$, $b = 2F^+_{rcv} + kF^+_{ack} + L = \delta_{LD} + L$. Consider any two times t_1 and t_2, with $t_2 \geq t_1 + a + b$, such that i and j are in region X and Y respectively throughout $[t_1, t_2]$, moreover the hop distance between region X and Y is less than or equal to k. Let t be the earliest time equal to or before t_1, such that i and j are in regions X and Y throughout $[t, t_2]$.

Without loss of generality, suppose i enters region X at time t. Then i and j participate in the link establishment, which takes at most $a = \delta_{LU}$ time, at the end of which (i, j) and (j, i) are both Up. So starting at time $t_1 + a$, or earlier, the links are Up.

The link teardown is not initiated by either endpoint until δ_{LD} time before leaving their respective regions, which by assumption, is no earlier than $t_2 - b$ (since $b = \delta_{LD}$). So (i, j) and (j, i) remain up until at least $t_2 - b$. □

Lemma 4. *The neighbor discovery protocol satisfies the proximity condition.*

Proof. Consider nodes i and j. If (i, j) is in state Up then i and j are k-hops apart. By definition the maximum distance between any two points in the regions which are k-hops apart is at most r_{min}. Hence there is a communication channel between node i and j. □

Lemma 5. *The point to point algorithm satisfies the condition for no duplicate receives.*

Proof. The abstract MAC layer satisfies the no duplicate receives property. The point to point algorithm only invokes one *bcast* on behalf of any given user-level message. Hence each *send* event will cause at most one *deliver* event. □

Theorem 3. *The point to point algorithm satisfies the termination condition.*

Proof. We have to show that if a message is sent from i to j and (i, j) is in state Up, then the message is delivered.

Suppose that (i, j) is Up at time t. This means i and j are in regions X and Y (where X and Y are at most k-hops apart), and they remain there throughout $[t, t + \delta_{LD}]$. Hence, by definition of r_{min}, i and j are within each other's transmission radius throughout $[t, t + \delta_{LD}]$. Observe that the underlying MAC Layer guarantees that any message broadcast by node i at time t will be delivered before time $t + F^+_{rcv}$ to any node that remains within the communication radius during this time. The maximum overhead incurred at the queue layer is kF^+_{ack}. Since $\delta_{LD} = 2F^+_{rcv} + kF^+_{ack}$, it is guaranteed that any message sent when (i, j) is in state Up will be delivered. □

7 Conclusions

We have shown how a reactive neighbor discovery protocol can be implemented over a reliable MAC layer. Many algorithms, such as the token circulation algorithms given in [6], can use this protocol to keep track of neighborhood information.

It is of interest to explore the following areas of future work:

- Lower bounds or impossibility results related to the neighbor discovery problem in mobile ad hoc network (based on abstract MAC layer, or a reasonable physical network layer model).
- Determining the lower bounds or impossibility results for the neighbor discovery problem over probabilistic MAC layer variants.
- A different protocol with a leader based approach, where each region has a leader node associated with it, which transmits information on behalf of other nodes.
- Making the service fault-tolerant to message losses in the MAC layer, and exploring self-stabilization.

8 Acknowledgements

This work was supported in part by Texas Higher Education Coordinating Board grant NHARP-00512-0130-2007, AFOSR Award Number FA9550-08-1-0159, and NSF Award Numbers CCF-0726514 and CNS-0715397.

References

[1] Abraham, I., Dolev, D., Malkhi, D.: LLS: A Locality Aware Location Service for Mobile ad Hoc Networks. In: DIALM-POMC Joint Workshop on Foundations of Mobile Computing, pp. 75-84, (2004)

[2] Krishnamurthy, S., Chandrasekaran, R., Mittal, N., Venkatesan, S.: Brief Announcement: Synchronous Distributed Algorithms for Node Discovery and Configuration in Multi-channel Cognitive Radio Networks. In: The Symposium on Distributed Computing (DISC), (2006)

[3] Krishnamurthy, S., Thoppian, M.R., Kuppa, S., Chandrasekaran, R., Mittal, N.,Venkatesan, S., Prakash, R.: Time-efficient distributed layer-2 auto-configuration for cognitive radio networks. Computer Networks: The International Journal of Computer and Telecommunications Networking, 52(4), 831-849, (2008)

[4] Kuhn, F., Lynch, N., Newport, C.: The Abstract MAC Layer. In DISC 2009: 23rd International Symposium on Distributed Computing, (2009)

[5] Liu, D.: Protecting Neighbor Discovery Against Node Compromises in Sensor Networks. In ICDCS: IEEE International Conference on Distributed Computing Systems, (2009)

[6] Malpani, N., Chen, Y., Vaidya, N., Welch, J.: Distributed token circulation in mobile ad hoc networks. IEEE Transactions on Mobile Computing, 4(2), 154-165, (2005)

[7] Mittal N., Krishnamurthy S., Chandrasekaran R., Venkatesan, S.: A Fast Deterministic Algorithm for Neighbor Discovery in Multi-Channel Cognitive Radio Networks. Technical Report UTDCS-14-07,Department of Computer Science, The University of Texas at Dallas, (2007)

[8] Park, V., Corson, M.: A Highly Adaptive Distributed Routing Algorithm for Mobile Ad Hoc Networks. In IEEE INFOCOM, pp. 1405-1413, (1997)

[9] Rao, A., Ratnasamy, S., Papadimitriou, C., Shenker, S., Stoica, I.: Geographic Routing without Location Information. In Mobicom: The Ninth Annual International Conference on Mobile Computing and Networking, pp. 96-108, (2003)

[10] Vasudevan, S., Towsley, D., Goeckel, D.: Neighbor Discovery in Wireless Networks and the Coupon Collectors Problem. In Mobicom, (2009)

[11] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The Theory of Timed I/O Automata.Synthesis Lectures on Computer Science, Morgan Claypool Publishers, 2006.