Mutual Exclusion, Leader Election, etc. in Mobile Ad Hoc Networks

Readings:
*** Malpani, Vaidya, Welch. Distributed token circulation on mobile ad hoc networks. Focus on the LR (Local Recency) algorithm.
Malpani, Welch, Vaidya. Leader election algorithms for mobile ad hoc networks.
Walter, Welch, Vaidya. A mutual exclusion algorithm for ad hoc mobile networks.
*** Walter, Cao, Mohanty. A k-mutual exclusion algorithm for wireless ad hoc networks.

1 Introduction

Today's material is based on the work of Welch, Vaidya, and their students and collaborators at Texas A&M, on algorithms for ad hoc mobile networks.
They consider an ad hoc mobile environment, in which the nodes can move. In some of the papers, the nodes can also join, leave, fail.
Assumptions common to all four of their papers:

1. The nodes have unique identifiers.
2. Communication is point-to-point. Communication links are reliable, bidirectional, and FIFO. Each node knows the nodes that it can communicate with directly.
   This is assumed to be implemented by means of a link-level protocol, which uses the broadcast capabilities of the underlying ad hoc radio network.

They do not assume that everyone can communicate with everyone—just that they can communicate with neighbors. Thus, they are not working on top of a routing layer. Their viewpoint is that many good distributed algorithms in the mobile setting will have a local flavor, and will use just neighbor communication.

In this setting, they focus on rather standard distributed computing problems such as leader election and mutual exclusion.

Their work basically involves coming up with algorithms that seem to work well. They present reasonable correctness arguments. For performance/complexity, they obtain simulation results.
There is plenty of additional work to be done to obtain analysis results that predict the behavior of the simulations, and to obtain improved algorithms and lower bounds.

I'll describe the key ideas of their algorithms.
The last three are based on ideas from Gafni, Bertsekas. The first is quite different.

2 Distributed token circulation

This simple paper addresses the problem of routing a token through a mobile network, so that it visits every node frequently.
Such an algorithm could be used for many purposes, e.g.:

1. Implement totally-ordered broadcast, either by piggybacking messages on the token as suggested by Cristian and Schmuck, or by piggybacking tags to establish the ordering (the actual messages could be obtained from elsewhere).
2. Implement mutual exclusion—the token embodies the critical section.

The model considered in this paper allows nodes to move, as usual, as long as they don't disconnect the network. It also allows nodes to join. Nodes can leave gracefully, in particular, when they don't have the token (they can wait to pass it on before leaving). They can't fail without warning, because that could happen when they have the token and result in the loss of the token.

Malpani et al. propose several algorithms, and compare them using simulations. Comparison is based on:

1. The time to complete a "round", which means that all the "current" nodes have been visited. (The definition isn't completely clear when the network is dynamic.)
2. Number of bytes of communication per round.
3. Number of nodes visited per round.

Example: Consider a graph with nodes arranged in a ring, with ids 1,3,5,2,6,4,1.
Visiting the nodes in the order 1,2,3,4,5,6,1,...yields high costs, but visiting in the order 1,3,5,2,6,4,1,...yields minimum costs.
They use this example to show why it's important to take topology into account when deciding on the traversal order.

Their conclusions show that one of their algorithms clearly dominates the others. It's called the "Local Recency" (LR) algorithm. So I'll just describe that one.

(Local) A node i that has the token chooses the next node to send the token to, from among the nodes that it currently believes are its neighbors.

(Recency) From among the possible neighbors, it sends the token to the one that the token has visited least recently.

A nice heuristic. An alternative they consider, which isn't as good, is to send to the one that the token has visited least frequently.

One way to implement Local Recency is for the token to keep a count of how many hops it has taken. Each node it visits keeps track of the latest count it saw. To decide where to pass the token, a node needs to learn about its neighbors' counts. Alternatively, the token itself could keep track of the latest count for each node in the system. Then the information that a node needs about its neighbors' counts is available right there in the token.

Of course, that isn't very practical...

Claim: If the network stabilizes to a (finite) connected graph, then every node will get visited infinitely often.

Proof: Suppose not. Let N be the set of nodes visited infinitely often and F the set of nodes visited finitely often.
We know that N is nonempty, since the token keeps moving.
If F is empty we are done, so assume that it is also nonempty.
Some node i in N is adjacent to at least one node in N.

Eventually, the execution reaches a point where:

1. No node in F is visited again, and
2. The counts for all the nodes in N are greater than the counts for all the nodes in F (that is, all the nodes in N have been visited more recently than any of the nodes in F).

After this point, when node i subsequently gets the token, its neighbor in F has a lower count than any of its neighbors in S. So, i sends the token to a neighbor in F. Contradiction.

Simulation results for this algorithm fall into two categories: static and dynamic networks.
For static networks: After one round, the LR algorithm settles down to a repeated pattern, visiting the same nodes in the same order, and the number of hops is close to optimal. On the average, this bound is linear in the number of nodes, in fact, less than $2n$; however, a complicated counterexample graph has been produced (by hand) in which the number of hops is actually exponential in the number of nodes. I don't recall if the counterexample works for only the first round, or for subsequent rounds as well.
For dynamic networks: With bounded mobility, LR continues to perform well.

The algorithm tolerates mobility easily (results in links forming and failing), as long as the network doesn't become disconnected.
It tolerates joining (start with count of 0).
Also leaving, as long at the leaving node doesn't have the token (and doesn't disconnect the graph).
Stopping failures wouldn't hurt either, unless the failing node happened to have the token. But we say that this doesn't handle stopping failures, because it doesn't always handle them.

3 Leader election

3.1 Overview

The token circulation algorithm doesn't handle the loss of a token. Regenerating a token is a traditional task of a leader node. This leads naturally to considering leader election algorithms.

The problem is for the entire mobile network to elect a single leader, who learns, and can announce to higher layers, that it is the leader. Other nodes also learn the leader's id.
If the network becomes partitioned, then a leader should be elected within each component.
Because of the dynamic nature of the system, we might have to live with no leaders, or more than

one leader, in unstable situations (e.g., when two components have just merged).
So the problem is stated in terms of gravitating to a single-leader-per-component situation in cases where the network (or, just that component), stabilizes.

They consider mobility, joining, leaving. Link failures.

Leader election can be used:

1. To regenerate a token after a failure in a token-based algorithm, e.g., a token circulation algorithm, or a token-based mutual exclusion algorithm.
2. To choose a coordinator for managing a group communication service.

Malpani, Welch, and Vaidya present a leader election algorithm based on the partial reversal algorithm of (Gafni, Bertsekas), and on TORA. Now, instead of creating a DAG directed to a fixed destination site, they create a DAG directed to a node that is called the "leader"; that is, a "leader-oriented" DAG rather than a "destination-oriented" DAG.
When a partition is detected (using the TORA mechanism), the algorithm elects a new leader and propagates its id throughout the component.
When two components merge, the two leaders compete and the id of the winner (that is, the smaller id) is propagated throughout the merged component.

In addition to just electing a leader, their algorithm creates the directed DAGs.

3.2 Review of GB and TORA

Review Gafni-Bertsekas partial reversal algorithm, using (α, β, i) triples.
This strategy ensures that, after a node does a partial reversal, it will have at least one outgoing link (that is, it sets its height to be higher than at least one of its neighbors).
This strategy tries to limit the number of reversed links, which helps to reduce the propagation of changes.

TORA's biggest addition was a mechanism for detecting when the destination is no longer reachable—the original GB algorithm could loop forever in this case, sending infinitely many messages.

TORA's height is of the form $((\tau, j, r), \delta, i)$.
The last component is the local id, which is used as in GB as a tiebreaker.
(τ, j, r) is the "reference level".
Here, τ is a timestamp, j a node id, and r a Boolean.
This new reference level gets started by node j if it loses its last outgoing link due to a link failure.
It sets τ to the current time (which makes this reference level larger than any defined earlier), j is its own index, and $r = 0$, which means "unreflected".
(A reference level that has $r = 1$ is "reflected"—this is part of the mechanism to detect partitions.)

δ is another tiebreaker. This is all a little complicated...

There are several rules that can be followed by node i when it has no outgoing links, to carry out a partial reversal:

1. If not all of i's neighbors have the same reference level, then i sets its reference level to the largest among all its neighbors but sets its δ to one less than the minimum δ value among those with the same reference level. Thus, it is reversing edges to all those with lower reference levels.
2. If all of i's neighbors have the same reference level, then:
   (a) If it is an unreflected reference level, then i starts a reflection of this reference level by setting its reference level to be the same as its nbrs but with $r = 1$; also, it sets δ to 0.
   (b) If it is a reflected reference level, then (a) if i is the originator, then i has detected a partition and does something... (b) if i isn't the originator, then i starts a new reference level. (Occurs when a link fails during recovery.)

This looks a little complicated and ad hoc. Basically, TORA allows nodes to detect possible partitions, and provides a "reflection" mechanism that allows them to avoid sending messages forever, when it will do no good.

3.3 The new leader election algorithm

The height is now $(lid, reference - level, \delta, i)$. The first component is just the identity of the proposed leader, the rest are as in TORA.
Reference level $(-1, -1, -1)$ used by leader of a component, to ensure that it's always a sink (if the heights are based on its own identifier as the lid).

A node that thinks it has detected a partition proposes itself to be the leader of the new component.
To do this, it sets its height to start with its own identifier in the lid field, and sets its reference level to $(-1, -1, -1)$. Information about the new leader is propagated throughout the component.

When more than one id is being used in heights within a component, (which can happen, e.g., when components merge), the smaller lid will "defeat" the larger lid. Does this happen automatically, because the lid is in the high-order position in the height tuple?

The specific rules are given on p. 4 for the single-topology-update case, and then augmented with other rules (p. 7) to handle multiple topology changes, which might interfere with each other. They look rather complicated and detailed.

Basically, the idea seems to be: Have each node independently decide if it thinks it is the leader of its component. Use some strategy that ensures that if a component has no leader, someone will eventually decide that it should be the leader. Then sort out duplicates by means of the heights, where the proposed leader with the smallest id will win out.
For the particular winning id, the DAG gest set up as in TORA.

4 Mutual exclusion

4.1 Overview

Now they consider the mutual exclusion problem in mobile ad hoc networks. They assume a fixed number n of processes; no joins, leaves, or failures, just mobility, which can cause links to fail or recover. As before, they assume reliable notification of link status, up or down. They assume no partitions. Their algorithm uses neighbor communication only.

They want to achieve low latency for entry to the critical region, without incurring a huge communication cost.

They simulate their algorithm and compare it with a static mutual exclusion algorithm (in which anyone can communicate with anyone) running over a routing protocol.
Their algorithm achieves notably better latency, without adding any communication cost.

Their algorithm is token-based; only the node with the token can grant access to the critical section to its client.
However, it does not depend on systematic token circulation.
Rather, it involves maintaining a DAG directed toward the node with the token.

Using ideas similar to those of Gafni and Bertsekas, they set up a DAG with a dynamic destination node—the node with the token.
Requests are routed along the edges of this DAG, toward the destination node.
Actually, the request isn't sent all the way to the destination node—it just goes to its neighbor on the way to the destination node.
Each node maintains a FIFO request queue that may contain its own id and the ids of its neighbors, indicating priority for sending the token.

As in GB, the DAG is maintained using heights, where the token holder has the lowest height.
Each node chooses its lowest height neighbor as its preferred link to the token holder (tends to get the request there faster).

4.2 The Reverse Link (RL) mutual exclusion algorithm

A node keeps track of:

- status: Where it is in the mutual exclusion algorithm.
- N: Its set of current neighbors.
- myHeight: This time, the height is a triple $(h1, h2, i)$. As usual, links are considered to be directed from higher height to lower height nodes.
- height[j]: records heights of neighbors
- tokenHolder: says if you have the token
- next: This is supposed to indicate where the token is—either at your own node, or else which neighbor is the first one along a path to the token
- Q: FIFO queue containing identifiers of self and requesting neighbors, in order of receipt of requests
- And some bookkeeping stuff.

Event-driven. Events are arrivals of requests and returns at client boundary, arrivals of messages, and link status notifications.

When request arrives from client:
Update m.e. status.
Enqueue your own identifier locally, on your request queue.
If you don't have the token, and you haven't already done so on behalf of another current request (you can tell this simply by seeing if the queue now contains exactly one element), then send a request message to "next".
On the other hand, if you already have the token, then send the token to whomever is first on your queue—either your client or whichever neighbor is first.
(They claim that your own client must be first in this class; but if this is so, it seems like a fluke. I prefer more generality here.)
If you pass it along, you have to lower the neighbor's height to reflect that it is now the token-holder, not you.
If you send the token to someone and your queue isn't empty, then follow it with a request.

When the resource is returned by your client:
Update m.e. status.
If there is any active request on your queue, then pass the token to whomever is first.

When a request arrives from a neighbor:
Record the height (piggybacked on all messages).
If the link is incoming (and another technical condition is satisfied) then enqueue the neighbor's identifier.
Now, if you have the token, pass it to whomever is first on your queue, following it with a request if necessary.
(They are repeating this call in a lot of places—there must be a nice way to separate it out into an action on its own, which gets triggered whenever you hold the token and your queue is nonempty.)
On the other hand, if you don't have the token: If this request is first on your queue then forward it to next; unless next is invalid (e.g., link has reversed), in which case you might need to first recalculate next, even raising your height if necessary.

(This event-driven code is really awkward. I bet it can be made much simpler by allowing various actions to be triggered by predicates (locally-controlled) rather than just by inputs.)

When the token arrives:
Set token-holder to true.
Lower your height to be less than your neighbors'.
Inform all your neighbors of the height change (sending a LinkInfo message).
And now, pass the token to whomever is first on your queue.

When you receive a LinkInfo message:
Record the neighbor's new height.
Do various bookkeeping, e.g., if the neighbor is now an outgoing neighbor but it appears on your queue, then remove it from your queue.
Now if you have no outgoing links and you don't have the token, then raise your height to produce outgoing links. This uses the partial reversal strategy, as in GB.
This might require you to resend a request...

When you sense that a link has failed:
Bookkeeping: Remove from N, delete any request from that neighbor from your queue, possibly raise height, possibly resend request.
(As before. Same stuff happens whenever topological changes occur.)

When you see a new link:
Just send LinkInfo message and do minor bookkeeping.

Basically, this is using heights to maintain a DAG pointed toward the current token-holder.
Requests are sent in the direction of the token-holder, but are only queued at the immediate neighbor. That neighbor sends its own request to the next node, and so on.
Each node's request queue is kept in FIFO order, for reasonable fairness.

A lot of detailed code to maintain this structure, could be simplified.

The algorithm obviously ensures mutual exclusion, because of the use of the token.
They also prove that, if topological changes ever cease, then eventually any request will be granted.
They do this by first showing that if changes stop, then eventually the structure stabilizes to a correct DAG, with all the pointers set correctly.
Then, a simple termination argument based on a mapping to a well-founded set shows that each request eventually gets satisfied.
The set is a set of vectors, giving the request's positions on the path of queues between the originator and the token-holder.

Sounds like a nice algorithm.
They don't prove any time bounds. Such bounds could presumably be extracted from the termination proof, but might be very pessimistic.
They also give simulation results, which show how the strategy works "in practice".

They compare the latency results to those for a standard static mutual exclusion algorithm based on circulating a token over a spanning tree. The tree is assumed to be a shortest-paths spanning tree, assumed to be produced by an underlying "routing" layer.

They simulated only a 30-node system.
Poisson arrivals for requests.
Link changes also Poisson.
Changes to graph consist of deletion of a randomly chosen link followed by insertion of randomly chosen link.

They compare based on average waiting time for entry to CS, and amount of communication generated.
Their RL algorithm does very well, compared to RR, especially when load and mobility is high.
The message communication is better for RR, but the comparison isn't fair because they aren't counting the messages within the routing layer. When these are taken into account, RL does comparably.

4.3 k-exclusion

They generalize this work to produce an algorithm, KRL, for k-exclusion, wherein $k \geq 1$ nodes can be in the CS at once.

KRL constructs a DAG based on heights, as in the mutual exclusion algorithm just discussed.
The DAG adjusts to the physical topology, as before.
But instead of having only one sink, this DAG is allowed to have up to k sinks.
KRL maintains k tokens in the DAG instead of one, and these are supposed to be at the sinks.
Every non-token-holder is supposed to always have a path of directed links to some sink.

One difference is a new "token forwarding strategy", by which unused tokens don't just sit there, but are circulated continuously throughout the network.
This is shown (by simulations) to improve latency quite a bit (approximately a factor of 2).

The heights are triples, as in the mutual exclusion algorithm.
When a node receives a token, it modifies the first two numbers in its height triple if necessary so that its height is lower than the height of the neighbor that sent the token.
Non-token-holding nodes must ensure that they have at least one lower height neighbor at all times.
If it doesn't, it uses partial reversal, raising its height above at least one of its neighbors and thus creating at least one outgoing link.
Also, a token holder ensures that it has at least one higher height neighbor, by doing a "reverse" of the GB partial reversal, lowering its height to be less than at least one of its neighbors, thus creating at least one incoming link.

Again, they get simulation results that yield good performance.
It would be nice to have more definite claims of guarantees under certain situations.
E.g., an important problem for k-exclusion algorithms is that, even if $k-1$ nodes go into their CS and don't leave, any other request will still be able to get into the CS.
Does this algorithm guarantee some version of this property?

5 Discussion

I like their neighbor-oriented approach. It looks much more promising to me than the standard approach, which builds a routing layer first and then runs algorithms on top of that layer. That approach does not seem to take appropriate advantage of physical topology.

They have only studied a few specific problems. More can be done with these, and more problems can be studied.
Their algorithms look nice, but so far they have mainly demonstrated their attractiveness using simulations. I would like to see analysis under various kinds of assumptions.
No lower bounds have been proved yet.

Clearly, this is a wide-open area for future research.