Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 19
April 26, 2006

Global Structures and Middleware

Readings:
Mittal, Demirbas, Arora LOCI paper
Malpani, Welch, Vaidya Leader election paper
Walter, Welch, Vaidya
Walter, Cao, Mohanty k-exclusion paper
Next:
Malpani, Chen, Vaidya, Welch Token Circulation paper
Chen, Welch Self-stabilizing dynamic mutual exclusion paper
Dolev, Schiller, Welch Random walk for self-stabilizing group communication

1 LOCI

[[Draw bunch of dots, with Voronoi tesselation, radius r circle around leader.]]

Problem: Given a 2D network of stationary nodes without GPS, some r, and some $m \geq 2$, partition the nodes into clusters such that:

- Each cluster has a unique leader.
- All nodes within r distance of a leader belong to the leader's cluster.
- The maximum distance from any node to its leader is mr.
- Each node belongs to the cluster of the nearest leader to it (Voronoi).
- There is a path from every node to its leader consisting only of nodes in the same cluster.

Distance refers to Cartesian distance, but it seems we would prefer hop count.
They mention in the conclusions that we can use hop count if we know that every node has a neighbor in each 60 deg cone around it.

Want distributed, local, scalable, self-stabilizing solution.

Assume:

- Nodes can fail-stop and recover.
- UIDs.
- Unit communication radius.
- Distance estimation: Somehow, nodes need to be able to calculate distance between points they've heard about. (Seems to sort of defeat the purpose of no coordinate assumption.)

Why is the m factor at least 2?
Stability. We're considering local solutions.
Clusters are determined locally to fit radius requirements (cluster consists of all nodes in a disc of radius r around its leader and some nodes up to distance mr from the leader).
What happens when a node appears or wakes up?
If m is too small, the node could be just out of range of any leaders of clusters around him, but too close to the clusters to start one himself.
This could require reconfiguration across the network to accomodate.
See the 1D example in Figures 1-3 of their paper using example of $m = 1$.

[[Figures 1-3. Draw Fig 1, add node j.]]

Node clusters are indicated with parentheses.
Node j joins, but is more than distance 1 from nearby clusters' leaders, so can't join their clusters, but is too close to members of those cluster leaders' r-disc members to have a non-overlapping r-disc if it became a leader.
Hence, the neighboring clusters would have to be destroyed to accomodate j, propagating through the entire network.

What value is big enough to prevent nonlocal reconfig? $m \geq 2$.
Why? Consider any partition of any subset of individual nodes into clusters satisfying our rules.
Now consider the addition of another node anywhere in the network.
Either new node within $2r$ of some cluster leader, so can join the leader's cluster, or can itself become a leader since the disc of radius r around it will not overlap the radius r discs of any other cluster leaders.
In second case, might wind up stealing nodes outside the radius r disc of nearby clusters to become part of its radius r disc.
Will not have a propagation effect, as does not affect leader status of surrounding leaders.

WIth this kind of scheme, the worst case ratio of number of clusters formed vs. the theoretical optimal can be seen to be 7, which is a constant.

[[Figure 9a. Draw circle of radius 2r, then 6 infringing circles on perimeter.]]

1.1 Algorithm

The algorithm at process j has three tasks:

1. *Start-cluster*(j):
   - Pre: j does not belong to a cluster, the distance from j to clusterheads of any neighboring nodes is more than mr.
   - Eff: j makes himself leader, starts growing a cluster iteratively, filling out his cluster with all nodes within his radius, then incrementing radius and accumulating more nodes.
2. *Join*(j, S), S the clusterhead of a neighbor k's cluster:
   - Pre: (1) j does not belong to a cluster and is within mr distance of S, and it's not closer to S than k is (guarantees that we have constant number of overlaps later),
     OR (2) j is farther than r from its clusterhead and within r of S,
     OR (3) j is farther than r from his clusterhead, k's cluster contains all nodes within r of S, and j is closer to S than to its own clusterhead.

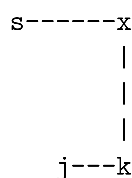
- Eff: j joins S.

3. *Resolve-overlap*(j, k):

   - Pre: j, k are clusterheads and j's r-nbrhood overlaps k's.
   - Eff: If j has a larger radius than k, or both j and k have same radius, but j has more overlaps, or if j and k are the same on both those counts, but j has a lower id, then j has lower priority than k.
     If j has lower priority than k then j reduces its cluster radius by one, telling periphery nodes in its cluster to reset their values, and letting others know the new radius.

There are some missing pieces here, which are alluded to in the paper.

- "The clusters are built iteratively, with each iteration incrementing the radius of the circular disc by 1."
  How do we do this?
  I think what it means is that each node in the cluster propagates information to the clusterhead about its distance, and that of all its neighbors.
  Once the clusterhead finds out information from all its neighbors, etc., within the radius limit indicating that all their neighbors are in the cluster or farther away, then the clusterhead can increment its radius and disseminate this information out to all of the nodes of the cluster.
- The above sounds simple, but it doesn't really work with their algo, unless we make more assumptions about the density of the network.
  Otherwise, consider the case where some node k is several hops from its clusterhead, node j is closer to k's clusterhead than k is, but j is only connected to k, not any other nodes.
  In this case, k must have been added to the cluster before j, but now j can't be added since it is closer to the clusterhead than k.
  Actually we should have already had a problem, somehow, since k shouldn't have been added before j since j was closer.

  [[Figure 4. Edge from s to x, and from x to k is length r. Right angles at each bend.]]

  Hence, I *think* we just need to assume that for any two nodes x, y, either they are within communication radius of each other, or there is some neighbor of x s.t. the distance from this neighbor to y is less than the distance from x to y.
  With this assumption it seems we can't have the kind of counterexample case I was just talking about.

I think this leads to a deeper question about this algorithm and its assumptions.
It seems sketchy to assume that nodes have access to ranging information to other nodes.
It would be nice to move to hopcounts instead.
It might be possible, but I think it's probably the case that hopcount distancing doesn't lead to stable solutions, even under the density assumption I suggested.

It seems the authors realized there was a problem at some point.
They then decided to simplify the algorithm, just using it for singlehop (FLOC).

They sketch correctness and stabilization time, showing the time not to be dependent on network size.
I'm not going to go through this, since we know that there are problems someplace (perhaps not surprisingly, those portions are finessed in the proof).

They also did an analysis of smaller values of m than 2, to see how many nodes get missed.
Makes a big difference.

They ran simulations to see what the number of clusters were that were constructed relative to optimal, and observed that it was much better than the theoretical worst case they sketch.

1.2 Hierarchy Construction

They explain how LOCI could be bootstrapped into a hierarchy:

Run LOCI on the network.
The resulting partitioning is level 0 of the hierarchy.
Clusterheads of level 0 clusters run LOCI amongst themselves to come up with level 1 clusters, and so on.
Clusters are grown so that at each level i, a clusterhead's cluster is guaranteed to have all level $i-1$ clusterheads within r level $i-1$ cluster hops, and some number up to mr cluster hops away.

Radius of clusters at that level are $[r(2mr)^i, mr\frac{(2mr)^{i+1}-1}{2mr-1}]$.

1.3 Self-stabilization

Can actually be made to handle corruption failures, though a good number of the details were not explicitly written.
I do believe it, though, assuming you could fix the problems I mentioned above.

Corruption failure is one in which a node's non-program state can become arbitrary without warning.
Can be useful feature in fault-prone sensor nets.

2 Middleware

The next several papers are by Welch, Vaidya, and their students.

They aim at producing "middleware" for mobile networks—helpful services for building applications.
E.g., mutual exclusion, k-exclusion, leader-election, group membership, reliable multicast,...

However, they don't build these upon a routing layer, in contrast to most proposals at the time they did their work.
They describe the state-of-the-art for MANET algorithm design as consisting of first implementing a routing layer, and they running a distributed algorithm designed for a static network on top of it.

Quite rightly, they say this might not be very efficient, so they propose another approach.

They start with a rather realistic model of a changing mobile network, a graph with frequent topology changes.
They implement their services using algorithms that run directly on that model; their algorithms are designed to adapt easily to frequent topology changes.
They call these algorithms "mobility-aware".
They don't use point-to-point routing.

They also happen not to use any information about geography (though I think that might be useful.

2.1 The network model

n mobile nodes with uids

Organized into undirected graph with edges that come and go, for unspecified reasons.
Presumably, the reasons have to do with mobility and properties of the underlying wireless broadcast service.
(Also possible due to nodes joining, leaving, failing.)
But this is all abstracted away nicely.
The different papers have some variations, e.g., one paper uses a directed graph.

The nodes send point-to-point messages over the graph edges only; they do not take advantage of the underlying local broadcast capability.

An issue with the model:
The atomicity of link change events isn't clear.
Sometimes it seems that they are assuming that, in one atomic step, a link forms, and both endpoints instantaneously become aware of it.
And similarly for link removal.
This isn't completely clearly stated.
Recall in TORA, a delay was allowed in when the endpoints of a link learn about the change.
(Actually, I don't think the TORA paper explicitly mentioned a graph abstraction—it just presupposed opinions about the existence of an edge, which could be different at the two endpoints.

They use the term "mobility aware" to refer to algorithms that use their model.
This seems to mean just that the nodes learn about existence/nonexistence of edges and have the opportunity to react accordingly.
Probably this is being contrasted with point-to-point routing algorithms for mobile networks (DSR, AODV,...) that try to hide the geographical location information.

2.2 The papers

The papers fall into two batches: three on algorithms that use TORA-like, Gafni-Bertsekas-like strategies, and three that use token circulation.
Today we will cover the TORA-GB papers and next week the token circulation papers.

3 Leader election using a TORA-like strategy

From Malpani, Welch, Vaidya Leader election paper
Interesting, but looks somewhat preliminary.

3.1 Overview

The basic idea is to use Gafni-Bertsekas and TORA algorithm ideas to maintain leaders within connected components in a changing mobile network.
I found the original TORA paper rather confusing; the algorithm was presented without proof.
The algorithm given here modifies TORA in small ways, but does have at least a partial proof.
However, the proof contribution is limited, because it deals with only a static case of the algorithm.

The algorithm is presented in two stages:
1. A first version, designed to work in a restricted setting where each change is fully handled before another change occurs, and processing and communication are completely synchronous.
2. The second, full version, where changes can occur at any time, including when previous changes are still being handled, and where processing is asynchronous. This is the model I described above.

The second case is the more interesting one for real networks; unfortunately, it seems messy—I found it hard to understand—and has no correctness proof.
So, I will just summarize the high-level ideas here.

Why might one want leader election in a mobile network?
They note that, if some recovery protocol is needed, a leader can take charge.
E.g., in a token-circulation algorithm, if the token gets lost, a leader can regenerate a new one.
A leader could also take charge of coordination of normal processing, e.g., in a mutual exclusion algorithm (as we'll see in the Chen paper next time).

Their definition of leader-election:
If the network topology stabilizes for sufficiently long (no edge changes), then each connected component should stabilize to a situation in which exactly one node in the component is designated as the leader; moreover, every node in the component should know who the leader is.

Thus, their protocol will have to deal with issues such as:
Electing a new leader in a component that has become separated from its leader.
Killing of all but one one of the leaders when several components merge.

Recall Gafni-Bertsekas: Manipulates heights in such as way as to produce and maintain a destination-oriented DAG.

Recall TORA: Modifies this to detect partitions and stop sending messages.
Allows edges to be undirected when no routes are needed, or when the adjacent nodes are disconnected from the destination.
The new algorithm(s) of this paper:
Each component forms a DAG directed toward its own leader (instead of some global "destination").

They could have added these DAGs as part of the problem statement, as in GB and TORA, but instead, they just state the problem as leader election, regarding the DAGs as byproducts.

Related work:
Leader election algorithm of Hatzis, Pentaris, SPAA99.
We will be covering some of their papers in class 21.
Their algorithms use geographical knowledge (which is considered a disadvantage here).
Also, they don't always work, some of their algorithms require compulsory motion, and they don't address partitioning and merging issues.

3.2 Definitions

n mobile nodes, communicate by point-to-point messages over changing network, as above.
In this paper, the network graph isn't necessarily always connected.

Nodes have uids
Links are bidirectional, reliable, FIFO.
Each node knows its current neighbors. (Instantly? Do the two endpoints always agree?)

Leader election problem:
Each node has lid variable.
If the network stabilizes from some point on, then eventually, for
every connected component C of the network, there is some node l in C such that $lid_i = l$ for all nodes $i \in C$.

Moreover, every component has a leader-oriented DAG.

3.3 Review of GB and TORA

3.3.1 Gafni-Bertsekas partial reversal

Heights, partial reversal.

Correctness proofs, which we discussed in some detail.
Note that the GB proof works for a network in which changes may happen concurrently, before other changes have been completely handled.
So it does cover the concurrent case, unlike the proof in this paper.

(However, recall that, to show stabilization, GB assume that the network changes stop from some point on.)

3.3.2 TORA

No correctness proof was given, but the TORA paper refers to the generic correctness proof in GB and claims that this carries over—I haven't checked all the claims, though; remains to be worked out.

They give a nice summary of TORA.

The height is a 5-tuple $(\tau_i, oid_i, r_i, \delta_i, i)$, where (τ_i, oid_i, r_i) is the "reference level".

A node i begins a new reference level if it loses its last outgoing link because of a link failure.
τ_i is then set to the current time (actually, time of the failure).
oid_i is set to i, the originator of this reference level.
$r_i = 0$, to indicate this is the "unreflected" sublevel of this reference level.

The increment (δ_i, i) is used to induce the directions on the links, among nodes with the same reference level.
It is these two components that are actually used to form the destination-oriented DAG.
The originator of a new reference level sets its $\delta = 0$.

When node i creates a new reference level, that level is strictly larger than any previously-created one, since it's based on the current time.

The new level propagates to all nodes for which i was on their only paths to d.
These nodes must try to form new paths to d (or discover that there aren't any).

The accommodations work as follows.
A node i can lose all its outgoing links due to a neighbors's height change under several different circumstances:

1. If the neighbors don't all have the same reference level, then i sets its ref level to the largest, but then sets its δ to the minimum δ among all neighbors with the largest reference level, minus 1.

This is a partial reversal—keep the links pointed inward, for the neighbors with the same reference level.
But the node does at least acquire one outgoing edge in this way, since its new reference level is greater than some neighbor's ref level.

Notice that δ is set to be negative. The further away the nodes get from the originator, the more negative they will become.

2. If the nbrs all have the same ref level, and it's unreflected ($r = 0$), then i starts a reflection of the reference level by setting its reference level to be the same but with $r = 1$. And it sets $\delta = 0$.

This is a full reversal, then.
This is supposed to help in detecting a partition.

3. If the nbrs all have the same ref level, and it's reflected, and i is the originator, then i has detected a partition (and it starts a cleanup).

(4. Like 3, but if i isn't the originator, then this is like detecting the previous failure of a link, so then i starts an entirely new reference level as before.)

3.4 Algorithm for a single topology change

They present their algorithm in two pieces—the version that can assume that no concurrent changes occur is simpler than the concurrent one.
Not so pleasing...

In the first case, where one change is handled at a time, the changes to TORA are fairly minimal:
Height becomes a 6-tuple $(lid_i, \tau_i, oid_i, r_i, \delta_i, i)$, where the last 5 components are exactly as in TORA, and the first is a leader id.

The algorithm will favor leaders that have smaller ids.
Leader uses the special reference level $(-1, -1, -1)$ to ensure that it is a sink.

Now a node that detects a partition, instead of "cleaning up" as in TORA, instead elects itself as a leader.

3.4.1 Algorithm details

They give nice, simple pseudocode, p. 99 col. 2.
But (caution), this has to be changed later to accommodate concurrent changes—and then it isn't so simple.

Pseudocode:

A. describes what happens when a node detects the failure of its last outgoing link.

TORA used to start a new reference level here.
This algorithm does the same if i still has some incoming links.
Otherwise (that is, if all the adjacent links are undirected), this is a signal for i to elect itself as leader.

B-D are cases in which i has just learned that it has no outgoing links, because it just received a link reversal UPDATE message from some neighbor j.
Similar actions to TORA.
But here, it only takes actions if $lid_j = lid_i$.
The case where $lid_j \neq lid_i$ is handled by case E.

B. If nbrs' reference levels aren't all the same, then act as in TORA (adopt the highest ref level, do partial reversal using the δs), as in TORA case 1.

C. If nbrs' ref levels are all the same, and it's unreflected ($r = 0$), then reflect, as in TORA case 2.

D. If nbrs' ref levels are the same, and it's reflected, and i is the originator, then i elects itself the leader (analogous to TORA case 3).

Now, what happens if the new information comes from a neighbor j with a different lid?

E. If $lid_j < lid_i$, then adopt lid_j as your leader too (so, the smaller leader id wins).

??? I'm not sure of the rest of what is happening here.
Also adopt lid_j in case it's the same as the originator id oid_i of your current, reflected, reference level; this seems to be indicating that you have recognized that lid_j is someone who, for a good reason, has elected itself as leader, and may be about to detect a partition (?)

This part isn't clearly explained. And anyway, it changes for the concurrent case, getting a lot more complicated.

3.4.2 Correctness

The correctness proof is an invariant-style proof, based on the number of rounds (this is assumed to be synchronous).
It considers three cases separately:
A link loss that doesn't cause a disconnection; a link appearance; and a link loss that does cause a disconnection.

Case 1: A link loss that doesn't disconnect a previously-connected component:

The proof basically indicates the progress that is made in adjusting the link directions so they still point to the same leader.
Let i be the node that detects the link loss.
They identify the set V_i of nodes that lose their paths to the leader—these are like the "bad nodes" in Busch's paper.

They define "frontier nodes" to be the bad nodes that are adjacent (in the undirected graph) to good nodes.
A node j on an undirected path from a frontier node to node i adjusts its height appropriately, in a number of rounds that depends on the number of edges between j and i (on a longest path).
Other nodes take additional time to readjust their heights—depending on how far they are from the nodes in the first category above.

The proof gives an inductive presentation of these ideas.

Case 2: A link appearance.

There were previously two leaders; now the smaller will win.
Nodes in the component of the smaller leader don't have to do anything.
A node j in the component of the larger leader adjusts within a number of rounds depending on its shortest distance to the node in its own component that is an endpoint of the newly-added edge.

Case 3: A link loss that disconnects.

Similar to case 1. Everyone adjusts within a number of rounds depending on the length of the longest simple path (?).

3.5 The general algorithm

Unfortunately, here the algorithm gets complicated.
They replace case E—the one where you hear from someone with a different lid—with a much longer piece of code.
And they add a new case F.

E: ???
It seems like lines 3-6 address the case where node i hears directly about a new leader j, though i is already well advanced in processing an old reference level.
In this case, now i adopts the new leader j (instead of killing the new leader in favor of i's reference level as before).

But there is also something else going on in lines 8-17—it sounds like a techicality. I don't follow this.

F: Like 4., in TORA, only now the node elects itself the leader instead of cleaning up.
Apparently, this TORA case doesn't arise in the sequental setting discussed above, which is why we didn't see it handled there.

Initialization: Every node can start out electing itself a leader, and this should sort itself out.

Node recoveries after failure: Recover by electing itself a leader.

4 Mutual exclusion using a TORA-like strategy

From Walter, Welch, Vaidya paper: A mutual exclusion algorithm for ad hoc mobile networks
This paper is much simpler and clearer than the previous one.

4.1 Overview

Uses ideas like GB (not the partition-management ideas of TORA, just GB), this time for solving mutual exclusion.

Basic ideas:
Enforce mutual exclusion by means of a token: only the node that holds the token can use the critical section.
Have the nodes use heights as in GB to configure themselves into a token-directed DAG.

Differences from GB:
1. In GB, d is static; here, the token moves around, in order to grant the resource to different nodes.
As it does, the edges of the DAG have to reorient themselves to accommodate.
The DAG is used for sending requests in the direction of the resource.

2. Here, the nodes sometimes lower their heights as well as raising them.
This happens only when a node receives the token.
At that point, that node should have the lowest height, so it lowers its height to try to ensure this.

They prove that, if the network ever stabilizes, then every request eventually gets granted.
They don't analyze time bounds, but they probably could.

They perform simulations, comparing their algorithm to one by Raymond.
Raymond's algorithm works by:
1. Building and maintaining point-to-point routes.
2. Using those routes to build a static spanning tree, with links directed toward the token-holder.
3. Sending the token around on the static spanning tree.

The simulations seem to show that the time complexity is better for the algorithm in this paper, although the message complexity might be somewhat worse.
Raymond's algorithm apparently isn't very resilient—it doesn't tolerate link failures.
(A later variant by Chang does tolerate link failures, but doesn't provide for reintegrating recovered links, which is equally important in mobile networks.)

Their contributions include some modifications to GB's partial reversal method:
1. To maintain a token-oriented DAG, with a moving token, they allow token-holders to raise their heights.
2. Each node maintains a "request queue" that records requests from neighboring nodes, so it can help in granting these.
3. A node that has several outgoing edges chooses one as the "preferred" one—it's the one that has the smallest height.
(When this changes, the node will reissue a request on the new preferred edge.)

4.2 Definitions

n nodes, unique ids, as before.
In this paper, no node failures.
Communication links: bidirectional, FIFO.
Links can appear and disappear, as before, but they assume no partitions.

Link-level protocol ensures each node knows its neighbors.
(But do both endpoints learn about each other atomically?)
Reliable communication on links that are up.

The protocol is written in interrupt-driven style: each node's responds to:
Requests to obtain and release the critical section
Messages arrivals
LinkUp and LinkDown notifications

The response consists of state change plus triggering of some outputs:
Grants of the critical section
Message sends

The definitions of execution, etc. are standard.

They make some strong assumptions about link failure, however:

1. They seem to be saying that both endpoints hear about LinkUp and LinkDown events at the same time; thus, they seem to be considering strong atomic notions of link failure/recovery.

2. They assume that a LinkDown event cannot occur if any messages are in transit on the link; that seems like a strong reliability assumption for the links.

I don't know how dependent their results are on these very strong assumptions—perhaps they can be modified to handle more problems.

Mutual exclusion problem definition:
Mutual exclusion, as usual.
No starvation: If the link changes eventually stabilize, then every request is eventually granted.

4.3 Reverse Link (RL) mutual exclusion algorithm

4.3.1 Data structures

The interesting data maintained by node i:
Its mutual exclusion status
Whether or not it has the token.

N, its neighbors
$myHeight$, here a triple $(h1, h2, i)$; the handling is similar to GB triples, in their partial-reversal algorithm.
$height[j]$, to keep track of perceived heights of neighbors
$next$, a particular neighbor—supposedly the best one for routing towards the token
$queue$, containing ids of requesting neighbors
$receivedLI[j]$: ??? This has something to do with synchronization of link information. It is described as a Boolean indicating whether a Link-Info message has been received from node j, to which i recently sent a token. When this is false, height information from j will be ignored. (?)

And there are some other state components, $forming[j]$ and $formHeight[j]$, that seem to be also involved in synchronizing link information. ???

4.3.2 The RL algorithm

This says what process i does in response to each triggering event.

Requesting the critical section:
Enqueues its own identifier i on its $queue$ (nice, treating itself uniformly with its neighbors).
If it has the token, then calls a procedure GiveTokenToNext().
If not, calls ForwardRequest().

What do these two procedures do?

GiveTokenToNext():
This simply sees who is first on the queue (and dequeues it).

If it's i itself, it just grants the resource to the local application.

If not, it's a neighbor j; then i sends the Token to j, and resets $next := j$.
To accommodate the movement of the token to j:
—It lowers its estimate of $height[j]$, using a certain rule—the same rule j will use to lower its own height when it receives the token.
—If i's *queue* is still nonempty, it sends a Request message to follow the Token message, to help service these requests.

ForwardRequest():
This tries to find the token.
Sets *next* to the lowest height neighbor, and sends a Request message to it.

Exiting the critical section:
Just call GiveTokenToNext(), to pass the token on.

(In their code, the same procedure is called in many places, as a result of different triggers. Maybe this could be rewritten to avoid the multiple calls, in a precondition-effect style.)

Request from neighbor j:
If $height[j] > height[i]$ then i enqueues j on its *queue*.

Then, if i has the token and *queue* is nonempty, it calls GiveTokenToNext() (as before—repetitious).
On the other hand, if i doesn't have the token, then:
—If i now has no outgoing links, it calls RaiseHeight().
—Now i does have at least one outgoing link. Then if j is the only node on *queue*, or if the *queue* is nonempty and the link from *next* is now incoming, then call ForwardRequest(). As before, this tries to chase down the token.

Procedure RaiseHeight():
Raises height using the GB partial reversal rule: use min value of $h1$ minus 1, and then sets $h2$ to be smaller than all the neighbors with the same $h1$.
(Slight bug in the code: They don't set $h2$ in the case where S is empty. ??)

Token arrives from neighbor j:
Lower height to be less than $height[j]$. Use the same partial-reversal rule that j did before it sent the token to i. Now, what should i do with this token? Just what it did before—call GiveTokenToNext().

LinkInfo messages:
I don't understand this processing. Some type of synchronization for when links are changing?

Link failure, Link formation:...

4.3.3 Examples of algorithm operation

These might be worth tracing in class.

4.4 Correctness

Mutual exclusion is obvious, based on the token.

For "no starvation", they assume that link changes eventually cease, and then show that eventually the system settles down into a nice pattern.
Thereafter, they can use a progress function.

Lemma 3 is key: It says that, once the link changes stop, eventually the system settles down to where it thereafter always forms a token-oriented DAG.

Proof of Lemma 3:
They use the fact, which I proved for GB, that a DAG is token-oriented iff there are no non-token-holder sinks. Lemma 1.
So it's enough to show that eventually, there are no non-tolen-holder sinks.

To do this, they argue that there must be only a finite number of calls to RaiseHeight().
The argument is similar to the one I did for GB: if someone raises its height infinitely many times, then we can argue that every node does so.
But then the contradiction isn't so immediate as it was before—in GB, we get the contradiction simply because the destination never increases its height.

Now, they have to carry out a more detailed argument about why eventually someone must stop calling RaiseHeight().
What they do is consider any node that ever holds the token after the links have all settled down.
Then claim that such a node cannot thereafter call RaiseHeight().
The argument is by contradiction, assuming that i is the first node that calls RaiseHeight() after it has had the token after the links have settled down.
It's a rather technical argument about possible reasons why i might have called RaiseHeight() (because i lost its last outgoing link, because of receipt of some LinkInfo or Request message).
They consider both these possible message types, trace down how they could cause the problem to happen, and in each case reach a contradiction, by determining that the source j of the message must have earlier called RaiseHeight() after links settled down.
So, this violates the choice of i as the first...

Anyway, once they know that the RaiseHeights() stop, they know that no node will ever be a non-token-holder sink (since then it would call Raiseheight(). So, we have a DAG, as needed for Lemma 3.

Once they have a DAG, they can define a simple progress function for each request based on who it is waiting for: this is captured by which nodes are waiting on requests to which others, and the lengths of all these nodes' queues.
That is used to prove that every request eventually gets granted.

4.5 Simulation results

They compare with Raymond.
They use a routing layer that magically always provides shortest path routes (a totally connected

digraph?)
And on top of this, they assume a static spanning tree.
And then they run Raymond's token-based tree traversal mutual exclusion algorithm on this tree.

Their results don't make it clear which is better.
It seems that the average time is better for RL than Raymond in the presence of mobility (frequent change), though not much difference is seen in the static case.
RL is also better under increasing load, particularly under low connectivity.

For number of messages, it seems that Raymond is better (if I'm reading the graphs right).
That makes sense, because RL is doing more to maintain the structure.
And, they aren't charging for Raymond's perfect-information routing!

4.6 Conclusions

Revise this algorithm to make the code neater and less repetitive. Would precondition-effect style help?

Complexity analysis should be possible.

It would be nice to remove the strong link-layer assumptions. What happens in the presence of partitions?

5 Extension to k-exclusion

From Walter, Cao, Mohanty: A k-mutual exclusion algorithm for wireless ad hoc networks

6 Overview

Continuing on the same theme... This one is about k-exclusion, whereby each node may request access to the critical section, and the system is supposed to guarantee that at most k are in Crit at once.

Another resource allocation problem.
They say that resource allocation may be quite important in mobile networks, where the nodes are resource-poor (so may need to share).

Their algorithm, KRL, is a generalization of the mutual exclusion algorithm, RL, of the previous paper.
Now they use k tokens instead of 1, and let someone go to Crit only when it has a token. Then k-exclusion obviously holds.

The tokens travel around the network in response to requests; in an improved algorithm, KRLF, later in the paper, the tokens are also forwarded when they are not in use, to try to find pending requests that are waiting for other tokens.

This is again based on GB partial reversal, with heights that are triples.
Differences from GB (same as in RL):
—Instead of just raising heights, as in GB, they also lower heights under certain conditions (when a node receives a token).
—Each node keeps track, in a *next* variable, of a preferred outgoing edge—normally, the one that leads to a neighbor with lowest height.
Etc.

Differences between KRL and the mutual exclusion algorithm RL:
—Techniques for maintaining the DAG with multiple tokens.
—Forwarding tokens to promote fair access.

They prove liveness claim: If the network stabilizes from some point onward, and not all the tokens are lost (by being hold by processes that fail), then all requests are eventually granted.
Thus, as long as one token stays alive, the algorithm will still grant all requests.
Though of course it might take longer than if there were more tokens.

They have extensive simulation results, showing that KRL and KRLF behave well in the face of network changes, including mobility.
And, KRLF indeed improves over basic KRL (in terms of delay to reach the CS), without significant extra cost in messages.
KRLF cuts the time for CS entry nearly in half.

6.1 Related work

They refer to distributed k-exclusion algorithms as either "permission-based" or "token-based".

Permission-based: Huang; Raymond
Everyone has to get explicit permission from all or some processes.

Token-based:
Better for dynamic ad hoc networks.
Seems to require less communication.
But existing token-based algorithms make strong reliability assumptions (network reliable, fully connected).

RL mutex algorithm is token based, maintains token-oriented DAG.
Each processor maintains a request queue of neighbors who have sent it requests for the token.
Maintains preferred *next* link.

KRL: Extends RL by using k tokens.
Multiple sinks allowed.
But every non-token-holding process should have a path to some token-holding process.
So, only token-holders should be allowed to be (remain) sinks.
Chooses lowest-height neighbor as *next* link.
Requests continually serviced in the presence of DAG changes.

KRLF: Distributes unused tokens around, preventing them from congregating in a small part of

the network.

6.2 System assumptions

n mobile nodes, uids
Communicate by point-to-point messages with neighbors.
Links are bidirectional, FIFO

Link-level protocol ensures each node knows its neighbors.
Reliable communication on links that are up.

Assume at least one current token-holder always remains alive.
Partitions may occur; components with at least one living token holder can continue granting the resource.

Event-driven protocol, same events as for RL:
Application events: Request, Release, Grant
Network I/O events: Recv, Send, LinkUp, LinkDown

They guarantee (same as for the RL paper): Mutual exclusion
No starvation: If link changes ever stop, then every request is eventually granted.
(They say "link failure", but I think they mean "link changes".)

6.3 KRL algorithm

6.3.1 Overview

This is the main content of the paper—code is in appendix, probably less enlightening.
Though, the code must be good, since it's been run through extensive simulations.

Maintain DAG, via height triples $(h1, h2, i)$ as in GB, links directed from higher to lower heights.

Initially:
Link directions form a DAG in which every non-token-holder has a directed path to some token-holder.
Every token-holder has at least one incoming link.
Tokens at nodes 0,...,k-1

Messages: Request, Token, LinkInfo

Process i acts as follows (written in something more like precondition/effect style):

Local request for the critical section:
Enqueues i on its *queue*.

When Request message arrives from neighbor j:
If $height[j] > height[i]$ then i enqueues j on its *queue*.

Send Request message to neighbor j:
When $j = next$ and *queue* is (first becomes) nonempty.

When Token message arrives from neighbor j:
If *queue* is nonempty, then process the first element:
—Dequeue it.
—If $= i$, grant the resource to the local application.
—If it's a neighbor j, send the token to j.
LOWER i's height, if necessary to make it less than $height[j]$.

RAISE height:
When a non-token holder finds itself with no lower neighbor.
Use partial reversal as in GB, RL, to change $h1$ and $j2$, thus ensuring that $height[i]$ becomes greater than height of at least one of its neighbors; creates at least one outgoing link.
When link to a requester j on i's request queue reverses to point outward, delete j from *queue*, if it's there.

When a link to neighbor j fails:
Delete j from *queue*, if it's there.

(Requests don't get lost as a result of these deletions, because the original request always remains in its own *queue*.)

LOWER height:
When you're a token-holder with no incoming edges (all neighbors have lower heights).
Then use the opposite of GB to change $h1$, $h2$ so that your height becomes less than that of at least one neighbor; creates at least one incoming link.

6.3.2 Example of static KRL operation

Straightforward, could trace in class.

An interesting remark:
They claim that, in a static network, no node will ever raise its height.
This can be verified by looking at the rules above—the only time a node RAISEs its height is when it's a sink and a non-token-holder.
So it can do so if necessary.
But suppose we reach a system state with no non-token-holder sinks.
We claim that this situation always persists (and so, no RAISEs can occur).
How can we create a non-token-holder sink?
1. A token-holder node becomes a non-token-holder sink, or
2. A non-token-holder remains a non-token-holder and becomes a sink.

In case 1, the node sends its token away on an outgoing edge—so it can't be a sink, and remains not a sink.

Case 2 can't occur because a neighbor raises its height, since we're assuming no one is doing

this so far.
So it must be that the node itself lowers its height.
But the only nodes that lower their heights are those that are (or become) token-holders. Contradiction.

This shows us that the raising-height mechanism, from GB, is used to accommodate to dynamic links.
The lower-height mechanism is to adjust the destinations' (tokens') link directions.

6.3.3 Example of dynamic KRL operation

Similar.

6.3.4 KRL with token-forwarding

It's possible for some node i to have a token, which another, j, is waiting for another token.
So they let each token-holder forward the token to other parts of the network in case it doesn't know anyone nearby who wants it (presumably, that means itself or any neighbors that happen to be on its queue).

Heuristic: Forward to token on the preferred edge—to whichever neighbor has the lowest height.
That is supposed to cause a smaller number of accommodating link reversals.

In fact, their forwarding policy seems a bit more involved than this—it seems that they are doing a kind of token circulation here, by keeping track of which nodes have been sent the token recently.
This might be a source of the LR token-forwarding algorithm, which we will study next time.

6.4 Correctness of KRL algorithm

k-exclusion follows from k tokens.

No starvation: If topology changes eventually stop.

Proof idea:
Once topology changes stop, eventually the network "stabilizes" to a situation where no one every RAISEs its height again (argument as above).
Then they argue somehow that any chain of propagated requests will eventually reach some token-holder (?).
Once this happens, they are able to define a progress function based on the length of the chain, that measures progress toward satisfying the given request.
Plausible, but not convincing—worth tracking down sometime.

They claim that the forwarding idea doesn't upset the progress argument—the key is that, even when tokens are moving, a request will eventually have to reach one (since there are only finitely many total nodes). ???

6.5 Simulation results

Average waiting time for KRL is good, KRLF is better.
Works very well in the presence of mobility.