Middleware

Readings:
Middleware: Walter, Welch, Vaidya mutual exclusion
Walter, Cao, Mohanty k-exclusion
Malpani, Welch, Vaidya leader election
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
Next:

1 Last time: Mutual exclusion using Gafni-Bertsekas

From Walter, Welch, Vaidya paper: A mutual exclusion algorithm for ad hoc mobile networks

Recall, their network model, changing graph, with links coming and going at any time.
Managed mutual exclusion using a token traveling through the changing network.
Used Gafni-Bertsekas link reversal ideas to maintain a token-oriented DAG (rather than a destination-oriented DAG as in GB).

A node that receives the token may lower its height (differs from GB, where nodes only raise their heights), to ensure that its new height is lower than that of the neighbor who sent the token. Nodes still raise their heights, as in GB, to maintain the token-directed DAG property.

Last time I described the algorithm in some detail—how the nodes respond to various input events, including requests to obtain and release the critical section, message arrivals, and LinkUp and LinkDown notifications.
I won't repeat this—refer to my notes or the paper.

Highlights:

Requesting the critical section:
Enqueues its own identifier i on its *queue*.
If it has the token, then calls GiveTokenToNext().
If not, calls ForwardRequest().

GiveTokenToNext():
Dequeues first element of queue.
If it's i itself, it 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():
Tries to find the token.
Sets *next* to the lowest height neighbor, and sends a Request message to it.

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

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().
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 $h1$ plus 1, and then sets $h2$ to be smaller than all the neighbors with the same $h1$.

Token arrives from neighbor j:
Lower height to be less than $height[j]$. Use the same rule that j did before it sent the token to i.
Then, to handle the token, call GiveTokenToNext().

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

Link failure, Link formation:...

1.1 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-token-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 having the token 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.

1.2 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!

2 Extension to k-exclusion

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

2.1 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).
—Nodes maintain queues of requests.
—Each node keeps track, in a *next* variable, of a preferred outgoing edge—normally, the one that leads to a neighbor with lowest height.

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 held 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.
(Compare: In the mutex paper, no node failures were considered.)

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.

2.2 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.

2.3 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: k-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".)

2.4 KRL algorithm

2.4.1 Overview

This is the main content of the paper—code is in appendix.
(Must be good, since it's been run through extensive simulations.

Maintain DAG, via height triples (h_1, h_2, 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 here 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 dequeue the first element.
—If $= i$, grant the resource to the local application.
—If it's a neighbor ℓ, send the token to ℓ.
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 h_1 and h_2, 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—requests are supposed to flow downward.)

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.

As before, they remark 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.
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 raise-height mechanism, from GB, is used to accommodate to dynamic links.
The lower-height mechanism is to adjust the tokens' link directions.

2.4.2 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.

2.5 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 ever 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.

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). ???

2.6 Simulation results

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

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.

The Leader Election Algorithm Papers are useful because they are mobility aware and they may elect a new leader to hold a token at any time.

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.
A version of the second algorithm is given as well but is not very easy to understand.
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.
There are only a few changes that occur to the TORA algorithm to get to the Leader Election algorithm.
That is, 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. Essentially, the levels are set to (0, 0, 0)

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 neighbors 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.

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 that is maintained by each node i.

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. It seems, however, that this case occurs only where there might be another failure when all of the nodes are trying to recover from a previous downtime.

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.
That is, there was never the case when the originating leader node was not the node itself when it came reflected back.
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 Token Circulation

We continue the study of middleware papers by Welch, Vaidya, and their students, now switching to token-circulation-based algorithms, rather than TORA-style algorithm.
The particular problems studied are:

Token circulation itself, using an interesting strategy.

Mutual exclusion using token circulation.
This rather technical paper introduces self-stabilization, and some interesting mechanisms to achieve it.

The third paper is about group membership, group communication, and (again) mutual exclusion, implemented using token circulation. Here, the token-circulation strategy is a random walk, unlike the previous two papers, which used deterministic strategies.

From Malpani, Chen, Vaidya, Welch: Distributed Token Circulation in Mobile ad hoc Networks

4.1 Overview

This paper describes a collection of algorithms for circulating a "token" repeatedly through all the nodes of a mobile ad hoc network.

The motivation for this comes from work on "group communication": they intend that the circulating token can be used to establish sequence numbers for messages, so that when the messages are later sent, in the background, everyone can establish a consistent order of message delivery.
They return to this motivation in the third paper today.

They study several algorithms, some of which use global information and some only local information.
The most interesting ones for the mobile setting are the local ones, because it is not so reasonable in that setting to assume that reliable global information is available everywhere.

They assume that the nodes run a low-level protocol to determine who their neighbors are at any time.
This protocol involves periodic "Hello" messages; timeouts of several Hello message intervals are used to remove a neighbor from the neighbor table.

Although neighbors are determined separately by each node, their examples use undirected graphs (which suggest that the two endpoints of an edge agree that they are neighbors).
However, the examples are used to illustrate behavior of the algorithms in static networks only.
They intend that the algorithm should work in dynamic networks as well; their simulations appear to include the independent neighbor determination.

The algorithm that behaves best in the static case is called Iterative Search (IS).
I don't completely follow this one, although it looks interesting.
It seems to be searching the network looking for a Hamiltonian path, recording distances to neighbors that should wind up being 0 if the neighbor is on the Hamiltonian path and greater than 0 if

not.
Since the Hamiltonian path problem is NP-complete, the algorithm can't always work.
But apparently it worked well in all their simulation scenarios (which probably consistent of rather few nodes).
No correctness proof or analysis is provided.

More interesting is their algorithm that behaves best in the dynamic case: LR (Least Recency).
In this algorithm, the token carries a count that gets incremented each time it visits a node. The token (or the nodes) keep track of the value of the count when each node was last visited.
In this way, a node that has the token can decide which of its neighbors was visited least recently, and send the token on to that neighbor.

They also study a variant of this algorithm, which instead sends the token to the neighbor that was visited least frequently in the past, rather than least recently. Their simulations determined that this algorithm, LF, behaved notably worse than LR, in the dynamic setting.
So, the point of the paper is really the LR strategy.

Interesting theoretical results include:
An upper bound on the length of a round of LR, that is, on the number of hops that it takes to traverse all the nodes.
A lower bound on this round length, obtained by an interesting graph construction.

4.2 Performance measures

They compare their algorithms based on:
Round length (number of node visits made by the token in one round.
Message overhead (number of bytes sent per round).
Time overhead (time required to complete a round).

4.3 The algorithms

They describe the "hello message" discipline here, with hello intervals.
If a certain number (called the "hello threshold") of hello intervals pass, then the node decides the timed-out node isn't its neighbor (for now).

4.3.1 Algorithm Local-Frequency (LF)

Keeps track of how many times each node has been visited, sends the token to the least frequently visited neighbor of the token holder.

What happens if a node decides to send the token to another node, and that node is no longer its neighbor?
The token could get lost in this case.
To guard against this, they use a TCP-like reliable communication strategy, on top of a DSR-like routing protocol, to reach the node.
Of course, the node could have failed, in which case this wouldn't work anyway...

Theorem: If the network is static and connected, then LF ensures every node is visited infinitely often.

Proof: Assume not, consider an execution in which someone gets the token only finitely many times.
Let F be the set of nodes that get the token finitely many times, I the others, who get it infinitely many times.
Now, surely the total number of node visits is infinite, since the LF rule always passes the token to someone.
So I is nonempty.
We have assumed that F is nonempty.
That means there must be two neighbors, $i \in I$ and $f \in F$.

But this yields a contradiction:
Consider what happens after f has stopped receiving the token.
Then i gets the token infinitely many times, keeps passing it to its neighbors.
Eventually f will become the neighbor who received it least frequently.
QED

However, the round length can be unbounded in a single execution, even for a static graph.
They show an example, Figure 2.
Trace the execution:

```
1,2,3,2,4,5,4,5,2,  1,2,3,2,1,2,3,2,4,5,4,5,4,5,4,5,4,5,2,    1,2,3,2,...

1     1         1                3                3
2     2         3                7                8
3     1         1                3                3
4     0         2                2                7
5     0         2                2                7
```

So this doesn't look very good—in fact, it represents exponential growth (approximately successive doubling).

4.3.2 Algorithm Local-Recency (LR)

The interesting algorithm.
Pass token to least recently visited neighbor.
In the above graph, we don't get the "exponential growth" problem. Now trace:

1,2,3,2,4,5,2, 1,2,3,2,4,5,2,...

Essentially the same argument as before shows that LR passes the token to everyone infinitely many times.
Now, you might expect that it would have a provable upper bound on round length that is much better than that of LF.

In most cases that arise in simulations, an upper bound of $2n$ is realized, where n is the total number of nodes.
Can that be proved as an upper bound?
No—there do exist graphs in which the round length is exponential in the number of nodes. We'll come back to this.

4.3.3 Global algorithms

Not too interesting—they study rules like "choose the node from the entire network that was visited least recently (or least frequently)".
And they distinguish cases based on whether you charge for the nodes in between or not.
We'll just skip these.

4.3.4 Algorithm Iterative Search (IS)

A heuristic, which seems to succeed in practice in finding a Hamiltonian path, by searching and looking for repeated visits to a node.
Notice that, in a static graph, even if the algorithm were exponential, it could be designed to eventually find a Hamiltonian path.
(There is no proof that the algorithm given here always does this—but presumably some correct searching algorithm could work, though not efficiently.)
Thereafter, the algorithm would work efficiently—n hops per round.
So, it shouldn't be too surprising that this comes out looking good in the static case.

4.4 Simulation results

ns-2, with CMU wireless extensions.
Random waypoint model.

They simulate both static and dynamic topologies.
Static topologies:
Iterative Search is best, after it stabilizes to a Hamiltonian path.
LR isn't much worse.
LF yields unbounded round length in some topologies (as in the example).

Dynamic topologies:
Discuss many aspects, but the upshot is that LR behaves best.

4.5 Analysis of LR

Here they focus on LR, having decided that it's the best algorithm they have, for the mobile case (and it's not too far from the best they have for the static case).

They tried to prove a good upper bound on the length of a route, e.g., $2n$, but were unsuccessful.
Jennifer Welch talked about this during a several-day visit to MIT a few years ago, and we tried to prove a bound.

Subsequently her student found a counterexample, showing that the worst-case length of a round is exponential in n, even in a static graph.
We'll see the counterexample, but first, let's see what kind of upper bound they can prove:

Theorem 2:
Every round of LR, on any static graph, has length $O(n\Delta^D)$, where Δ is the maximum degree of any node in the graph, and D is the diameter.

The proof of Theorem 2 relies on Lemma 1, which expresses some simple properties of the LR traversal:

Lemma 1:
(a) If node p is visited $degree(p) + 1$ times in a segment of an execution, then every neighbor of p is visited at least once during this segment.
(b) If p is visited no more than k times in an execution, then every neighbor q is visited no more than $(k + 1)degree(p)$ times in the execution.

Proof:
(b) actually follows fairly directly from (a), so look at (a).
(a) is fairly obvious—if you come back to p enough times, you will visit all neighbors, rather systematically.
QED Lemma 1

Then Theorem 1 follows by considering neighbors of neighbors, etc.
For example, using (b), twice, we see that, if p is visited no more than k times, then neighbors of neighbors of p are visited no more than $((k + 1)degree(p) + 1)degree(p)$ times. Thus, we get a blowup based on distance D, each time multiplying at most by the degree bound Δ.
The analysis involves a geometric progression in Δ.

Finally, they give the counterexample graph, showing path length that is exponential in n.
See Figure 14.
The graph is defined recursively, as $G_1, G_2, ..., G_k$, where each G_i is simply G_{i-1} plus a copy of a special graph S given in Fig. 13.
Unwinding these recursive description, we see that we basically have a cascade of copies of S, each connected to the next by a single edge as shown (r_{i-1} to p_i).

Now, the special graph S is designed to have some special traversal properties. Namely:
If the token starts at r, then before it reaches r again:
—p is visited at least twice, and —every neighbor of p occurs between each two consecutive occurrences of p in that segment, and before the first occurrence.
(Actually, this isn't what they say—they either before the first occurrence or after the last occurrence. I'm not sure why after the last occurrence is useful, though. Of course, maybe my strengthening isn't satisfied by the graph in Figure 13...a bug here?)

Now consider the traversal of the entire graph G_k starting from r_k and ending just before visiting r_k again.
As described above, the traversal must visit p_k at least twice, and before each visit, will visit all of p_k's neighbors in S_k.

The first time the traversal visits p_k, its neighbors in S_k have all been visited but r_{k-1} hasn't been.
So at that point, the traversal descends into G_{k-1}, from which it continues recursively, producing a long traversal.
Now it returns to r_{k-1} and then p_k again.
From there, we pick up the first traversal, which was moving through S_k.
It will reach p_k the next time (the third time in the overall traversal), having visited all of p_k's neighbors in S_k at least once in the meantime.
So again, each of its neighbors in S_k have been visited more recently than r_{k-1}.
So at that point again, the traversal descends into G_{k-1}, again producing a long traversal.

So, one traversal of G_k from r_k back to r_k must include two traversals of G_{k-1} from r_{k-1} back to r_{k-1}.
We recurse, getting four traversals through G_{k-2}¡ etc., which yields an exponential blowup.

4.6 Conclusion

They planned to study LR some more—trying to identify characteristics of graphs on which LR can be proved to have linear round length. I am not sure if anything came of this—don't think so.

Also, it would be nice to prove some theorem expressing properties of this algorithm in the presence of (limited) mobility.
They state that there is so far little if any analysis of algorithms in MANETs in the presence of mobility!
This is a 2005 paper.
Can we think of some such results? Any examples yet in this course?

They still need to work out the group communication thing...

How to tolerate token loss?
Develop self-stabilizing version of the algorithm?
Partitions?

5 Self-stabilizing dynamic mutual exclusion

From Chen, Welch: Self-Stabilizing Dynamic Mutual Exclusion for Mobile ad hoc Networks

5.1 Overview

This is a hard paper to read, because it is very detailed, and has many ideas intermingled.
The problem and solution are interesting, though.

Their goal is ambitious and worthwhile:
Produce a mutual exclusion algorithm that runs on a very badly behaved mobile network, with continual churn.
Moreover, it should be self-stabilizing, that is, it should tolerate occasional total system state corruption failures, and still manage to recover back to normal operation, before too much time has

elapsed.
And they want to state and prove real theorems expressing the algorithm's guarantees.

They have to solve many problems, and the solutions are interrelated.
There is little modularity in the presentation and proofs.

The main protocol ideas are:

1. Have the processes interested in accessing the critical region formally join a group of "active" processes, and when they are no longer interested, they leave the group.
   While they are in the group, they keep getting chances to enter the critical region.
2. Use a token-circulation protocol to circulate a mutual exclusion token around the currently active processes (members of the group), and use this token to control access to the critical region.
   The token is circulated using a variant of the LR protocol of the previous paper.
3. They use several mechanisms to achieve self-stabilization:
   Send messages, tokens, etc. repeatedly.
   Enforce bounds on the sizes of various variables (e.g., counters).
   Use timers liberally, e.g., to prevent too much from happening too quickly (while we're trying to stabilize the system).

Problems with the algorithm:

1. It relies on a known leader process, who is assumed to remain active at all times. They claim that they can remove this, by means of leader-election; however, they don't give any hints about where to find a good self-stabilizing leader-election algorithm for this setting.
2. It needs more modularity. For example, we might separate out a leader-election service, with well-defined guarantees;
   a token-circulation service;
   a group-membership service, which maintains information about who is in the group of currently-active processes;
3. It would be nice if the use of timing could be clearly articulated and abstracted in some way.

I'll give a rather quick tour of the rest of the paper.

5.2 Introduction

They guarantee mutual exclusion and progress conditions.
They can't guarantee mutual exclusion while the system is unstable, of course—just after stabilization occurs.

Their other properties are progress properties, of various strengths: deadlock-free, lockout-free, and bounded waiting.
They claim that they can achieve any of these, depending on what they assume about the underlying LRV algorithm.

To achieve self-stabilization, they send around multiple tokens (in case one has been lost).
That sounds dangerous for achieving mutual exclusion: the usual way token-based mutex works is that there's just one token, and only the token-holder can enter the critical section.

So they need to do something else here:
They have new tokens issued with id numbers, and have processes perform a local consistency check of their own state against the id number in the token.
In the "normal case", when the system is stabilized, the check should ensure that only one process will enter Crit at once.
Of course, while the system is unstable, anything can happen— more than one process can enter Crit.

Related work:
They refer to a "weakly" self-stabilizing leader election algorithm of Vasudevan et al.
I'm not sure what this means—I suppose it's weak enough that they can't use it in this setting.

5.3 Motivation

Distinguished processor p_0 keeps generating mutual exclusion tokens (m-tokens), periodically.
Local check needed, as noted above.

Their starting point is earlier work, by Dijkstra and by Varghese, on self-stabilizing mutual exclusion in a static ring of processes.
Dijkstra's is for a static shared memory model, which is quite different from what is being studied in this paper.
Varghese's is for a static network with FIFO links.

These algorithms use a local checking rule to decide when they can enter the critical section.
Namely, the token carries a value.
Process p_0 checks that the token's value is equal to the one in its current state, whereas everyone else checks that it is different.
When p_0 is done with the critical section, it increments the token's value and sends it to the next process. WHen anyone else is done, it just sends it along unchanged.

They show a little example of how a simple local check like the one used by Dijkstra and Varghese fails in the current situation—because of the fact that links can be non-FIFO.

They seem to be mixing two things in this discussion: Non-FIFO links and dynamically-changing network.
Actually, these two issues are related, in that a dynamic network is one cause of out-of-order message delivery, since messages can travel along different paths.
But there are other issues with dynamic networks, e.g., we can't establish a fixed ring.

So, they must add new machinery to deal with:
Dynamic virtual ring, updated periodically to reflect changing topology and join/leave requests.
Non-FIFO virtual links on the virtual ring.
Partitions—they don't want a process that gets disconnected from p_0 to prevent others from entering Crit.

5.4 System model

Digraph network (this is different from the others we've been considering, which are undirected).
Point-to-point communication.

$Assum_0$: Distinguished processor p_0.

$Assum_1$: Known upper bound N on number of processors; Processors have uids in $[1, ..., N]$.

$Assum_2$: Known upper bound d on message delay between neighbors, for messages that are actually delivered.
But messages can be lost.

Notice that this is the first time they've introduced the use of time into these algorithms.

$Assum_3$: Upper bound g on number of messages generated by a processor in each time unit.
(I'm not sure they actually need an assumption for this. If the processors have clocks or timers, as they seem to here, perhaps they could just schedule their sending steps to satisfy this condition.)
Variables are all bounded-size (needed for self-stabilization).

The combination of assumptions ensures that, in stable operation, there is a bound on the total number of messages existing in the system; this allows messages to have ids assigned from a bounded range.

Clocks:
They assume timers that decrease at the rate that real time increases.
Kind of an odd assumption; why not just say that they have logical clocks that go at the rate of real time, though the values aren't synchronized?
Anyway, they assume that they can set timers, and detect when they expire; timeout expiration is used to trigger activities, just like other kinds of inputs.

They talk about local computation events and topology change events.
A topology change event instantaneously changes the topology; however, there is no assumption about notifications.
Thus, there are no instantaneous notification events like in some of the other papers. No link-level neighbor notification service.
Processes will presumably have to decide as part of their protocols who they think their neighbors are.

Note that LR, which they are using, doesn't assume atomic, reliable neighbor notification—it uses Hello messages.

$Assum_4$: At each processor no more than one event is activated at the same time.
And, the local computation time is negligible.
The time a processor stays in Crit is negligible (this is a funny assumption—but they say later that it isn't needed).

Executions: The usual kind of definition.
Since this paper deals with self-stabilization, they don't require that the system start in an initial state—any state can be a start state of an execution.

5.5 Problem definition

Here they describe their variant of mutual exclusion, which involves join and leave requests.

Q: To make an algorithm self-stabilizing, shouldn't they require that the application keeps re-submitting its last join or leave request...the last one it submitted could get lost.

Eventual stopping: Any processor that becomes inactive (its last request is to leave, or it never requested to join in the first place), eventually stops entering Crit.

No deadlock: Some active process enters Crit infinitely often, if the execution is infinite.

The above isn't quite stated correctly: They define a process to be "active" if it eventually stop submitting join/leave requests and the last request was a join.
But what about infinite executions in which each process submits join and leave requests infinitely many times? Then there are no active processes, so the no-deadlock property is false.

They list other, stronger progress properties, as usual.
THese do not have the same problem in their statements, since they involve a particular process i that is assumed to be active.

They define self-stabilization, as usual.

5.6 Algorithm overview

Two kinds of tokens: Join-request tokens (j-tokens), and Mutual exclusion tokens (m-tokens)

Both kinds are routed using LRV, which is claimed to be similar to LR, but self-stabilizing.

LRV doesn't sound all that similar to LR:
It uses bounded timestamps, and enforces a lifetime on tokens, by discarding any token that has been forwarded more than a certain number of hops.
How do they set the number of hops?
That sounds like the lifetime of a token is really just one pass through the active processes in the network—not repeated circulation as before.

Tokens get ids, from bounded range, incremented modulo the bound.

Execution of algorithm is divided into phases.
In one phase, the m-token is supposed to pass through all the active members exactly once.
p_0 maintains information about the membership, in two variables: $ring$: the actual members, and $new - set$: the processes that are currently trying to join (from which p_0 has recently received j-token messages.

Membership gets updated only at the beginning of a phase.

In each phase, p0 repeatedly generate m-tokens carrying the same *ring* and $new - set$ info.
Processes in $new - set$ initialize their local states upon receipt of an m-token (they will receive it since the routing is controlled by the underlying LRV protocol, which should visit everyone).
They say that the processes in *ring* are somehow visited in the order specified in *ring*.
But I don't understand this—I thought that LRV was being used to determine the order of token visits, not the order in *ring*.
Anyway, when a process is visited, it checks its local state against the token to see if it really has access to Crit.
(It should be first on *ring*? Should satisfy other consistency conditions w.r.t. token id and local id?)

When a process wants to join, it sends j-tokens (repeatedly) to p_0 (somehow).

When a process wants to leave, it just says so by piggybacking the info on every m-token that visits it.

p_0 start a new phase:
When it gets the token back indicating that all the members in *ring* have gotten access to Crit on the previous phase.
Or, when a (nice, long) timeout expires.

At that point, p_0 updates *ring* to include the new processes in $new - set$ that have already initialized their states, and to exclude those who requested to leave.
p_0 gets to pick the ordering of the new processes that it adds to *ring*; it does this based on the order the m-token happened to traverse; that ordering was determined by the LRV algorithm. But of course there is no guarantee that the next time, the LRV algorithm will send the token along the same path! This is confusing...
$new - set$ is now updated to be all the new nodes from which p_0 has received j-tokens during the previous phase.

Those are the main ideas. The rest seems like details, and confusing...I'll try to pick out highlights.

5.7 Algorithm

5.7.1 Token circulation algorithm: LRV

Like LR, but the timestamps that are kept in the token are now in a bounded range $[0, L]$, where L is a large upper bound.
L should be chosen based on "normal" network behavior, to guarantee that, generally, the token can be forwarded to every processor within L hops.

Only tokens with max timestamp $< L$ are continued—others are just discarded (as being old, or corrupted).
Thus, it seems that tokens are intended to travel for only one circuit, which is different from LR, where they circulate constantly.

5.7.2 Data structures

They define:
Some time constants, e.g., bounds on token lifetime.
Some bounds on the sizes of token ids, on the differences between token ids in the system at the same time, etc.

Lots of token fields, and process variables...

5.7.3 Interfaces to the application

They impose a time bound between when a processor leaves the system until it can try to join again.
This is supposed to give the system time to erase all information related to that processor's previous activity.

5.7.4 Mutual exclusion algorithm

Details, details.
The basic rule by which a process determines whether it can enter Crit is given in lines 9.13-9.15 of the code, and discussed in the middle of p. 16.
The rule is supposed to be: For p_0, the token's id should be the same as p_0's current id. For others, the token's id should be one more than the process' current id.
Also, the process has to be the first one on *ring*.

The algorithm lets processes use timers to be sure that various information has been cleaned out, or that other things have been accomplished somewhere.
The exact uses of the timers are not articulated clearly.
Maybe we can establish a level of abstraction here, expressing what the timers do.

The code is formidable looking.

5.8 Correctness

Skimming...

They don't just start out proving mutual exclusion, based on properties of stable configurations.
Rather, they prove some liveness properties first—a pretty nonstandard proof organization.
That seems to be because they can't prove mutual exclusion until the system has stabilized.

What I would do instead is define the "good states" to be those satisfying a collection of nice properties.
I would argue that the set of good states is closed under all the algorithm steps. (But this is tricky because of the timer settings—we have to do some reasoning about the time deadlines.)
And of course, good states all satisfy mutual exclusion.

Then we need liveness arguments to show that we eventually reach good states.
For this, the argument should be done in stages, achieving better and better approximations to the good states (that is, more and more of the good properties).
That's kind of what they do—their diagram in Figure 6 suggests this type of successive approximation to good states.

What are the successive approximations they use?
Well, first, they reach a "Stopping" condition, where it seems that all processes that are going to end up inactive already have submitted their last leave request. And moreover, the execution has progress far enough so that never again will these processes ever enter their Crit regions.
Thereafter, they will reach a point where they satisfy a condition $conv_token$ (token convergence), after which the token ids will be in a narrow range ($token_safe$).
Again, thereafter, they reach a point where they satisfy another condition $conv_nmem$ (convergence for non-members), after which all information about non-members will be removed from the system ($nmem_safe$).
Finally, they get $conv_member$, which expresses good properties related to the members' states; this implies mutual exclusion.

Q: Are they assuming that everyone is eventually a member or eventually not a member? That no one joins and leaves infinitely many times?

Detailed arguments...
Some of the lists of properties look like the kinds of things that would made good definitions for successive approximation regions.

Q: Are they using time to the the extent that they might? Is there some way that making more extensive use of time could simplify all this?

This should all be expressed in terms of approximating regions...

Finally, they discuss the various liveness properties they defined for mutual exclusion, claiming that they can be proved based on particular properties that can be assumed for LRV. LTTR.

5.9 Conclusions

A key piece, self-stabilizing leader election, seems to be missing. How to do this?

6 Self-stabilizing group communication using random walks

From Dolev, Schiller, Welch: Random walk for self-stabilizing group communication in ad hoc networks.

6.1 Overview

Puts together many ideas from different places.
They assume a changing undirected graph, like (most of) the previous papers in this set.

They assume nodes learn about their neighbors (atomically at both ends?).

They use the graph to perform a random walk, sending an "agent" around, randomly choosing each successive step from among the set of neighbors.
The agent is regarded as an active entity, but that really doesn't seem to matter for these applications—here, it acts just like a message—a repository for some data.

Anyway, they identify a subclass of "nice" executions: those in which there is just a single agent, and it happens to arrive at every processor in the system within at most M moves (for some value M that is fixed, as a known function of the network size).

They use this "nice" abstraction to split the problem into two pieces:
1. Eventually achieving a nice execution, starting from an arbitrary configuration.
2. Using the nice execution to solve some interesting problems.

The interesting problems they choose are:
Group membership maintenance.
Group communication (group multicast).
Resource allocation (mutual exclusion).

They attempt to achieve nice executions using random walks.
However, this doesn't always work—some patterns of change in network topology can prevent any strategy from yielding a nice execution.
But they identify a few cases where it does work, and give some bounds on M that work well in those cases.

To achieve nice executions, they basically have to create new agents if none exist, and throw out duplicate agents if more than one exist.

Creating new agents:
Use a timeout. A process that doesn't see an agent for a long time creates one (but many might do this—but then the duplicate removel should take care of these).

Removing duplicates: Whenever agents collide, remove them all and start up a new one in their place.

Assuming nice executions, they can implement group membership:
Have everyone set a flag saying whether they want to be in the group.
The agent wanders around collecting information from these flags, and forming a new group (new viewid, known members) each time it sees a change in the membership requests.
The agent keeps track of the time it last saw a membership request from each process, and times the process out, throwing it out of the group, if it doesn't see another one for a long while.
Also, it removes a member if it revisits the member and sees the flag unset.

Using token circulation and group memberwhip, then implement group communication:
By letting group members attach messages to the agent, in order.
Every member of the group that the agent visits can then read the latest messages.
After a while, the agent can delete the messages (after they get old enough).

Finally, they use token circ and group membership to implement mutual exclusion:
Let the agents carry around a queue of requests from the members for the critical region.
The resource gets granted to the first node on the queue.
When the node finishes with the resource, it must wait for the agent again in order to remove itself from the queue (?)

Those are the key ideas. Now, for a few details:

6.2 Introduction

The algorithm is "self-stabilizing", which means that it can be started in any configuration, and eventually will start working right.
Here, that means that eventually (with high probability, anyway) it will start acting like a nice execution, which then guarantees that it gives the correctness properties needed for the other problems.

Group communication and group membership:
Well-studied abstractions for programming changing networks.
Originally designed for networks that didn't change very frequently.
However, mobile networks are subject to more frequent changes.
(But, which processes want to belong to the group might not change so quickly—and that may be a key determinant of the performance of this algorithm.)

Group membership:
Nodes decide to join/leave a named group.
The group changes, forming a succession of "views".
Each view = (viewid, members)

Group multicast:
Each member can send a message to the group.
It should be delivered to all members of the group (often the requirement says "while they are in the same view", but that extra requirement doesn't seem to be used here).
The order of delivery should be the same everywhere, for those messages that are actually delivered.

Designed for a fairly rapidly changing mobile network.
Flooding isn't good (too many messages).
TORA-like structures aren't too good either—the system may change too quickly to allow effective maintenance of the structures.
They also compare with "compulsory algorithms", which we will study in class 21.

This algorithm doesn't require any flooding, doesn't build any structures, and doesn't require any compulsory motion.

6.3 The system settings

n processors, $n \leq N$, upper bound N known
unique ids

Agents are something like processes, but are sent from process to process like messages.
When an agent is at a process, the process can execute some of its steps, then send it on to a neighbor.

Reasonable definition of execution. Can start in an arbitrary state (since they are considering self-stabilization).

Nice execution: A single agent; visits every processor in at most every M consecutive moves.

6.4 Random walks of agents

Choose the next node randomly, uniformly, from among the neighbors.
Ensure a single agent as described above:
Using timeouts to start up new agents (if you haven't seen any for a while).
Detecting collisions and discarding all but one.

Impossibility result:
Assume that we do have only one agent.
Even so, if the topology changes are bad enough, we can't guarantee that the agent will visit everyone, using the assumed random strategy or any other.

The impossibility result is based on a nice example: 2 moves back and forth between 1 and 3, always keeping the graph connected.
The agent visits 1 when 2 isn't there, and similarly for 3. So the agent never visits 2.

So they have to rely on some special properties of the topology and topology changes. They list three that seem to work, but this looks fairly sketchy and preliminary:

1. Fixed communication: That is, a random walk of a fixed n-node graph. Then known results say that it takes time $O(n^3)$ to reduce to a single leader and likewise $O(n^3)$ for a single leader to visit the entire graph.
2. Randomly changing graph: Always connected, but changes completely in between two successive moves of the agent. They it's essentially a random walk on a complete graph. Then they get $O(n \log n)$ for both times above.
3. Neighborhood probability. ??? This is unclear.

6.5 Membership service by random walks

Group membership is described for nice executions, with two requirements:

4.1. For every i, if g_i (the flag set by process i to indicate that it wants to be a member) has a fixed value (true or false) throughout the execution, then eventually i is a member of the current view recorded in the unique agent iff $g_i = true$.
That is, if i consistently says it wants to be a member, then it is in all views from some point on, and if it consistently says it does not want to be a member, then it is in no views from some point on.

Notice that the view is allowed to change, though.

4.2. If every g_i has a fixed value throughout, then eventually the view becomes stable (stays fixed at some (viewid, members) pair).

Now we describe the group membership algorithm.
We have to say what it does when started in an arbitrary state.
The algorithm will ensure that the execution eventually becomes nice (has a nice suffix); however, it doesn't have to start out nice—so we have to say what it does even when it isn't nice.

The agent carries a view around: (viewid, membership).
Also, for each member, the agent has a counter value.
Whenever the agent visits p_i, and p_i wants to be a member, its counter gets set to a ttl (time to live) constant.
This counter is then decremented whenever the agent is received by any processor.
p_i remains an active member of the group as long as its counter is strictly greater than 0.

Now, before the execution becomes nice, there can be no agents or more than one.
If any processor p_i doesn't see any agent for too long, it just creates one, initializing it with a new view with members = $\{i\}$.

If two or more agents, each with its own view, arrive at a processor p_i, it kills them all, and starts up a new view, again with members = $\{i\}$.

Whenever a processor p_i that holds a single agent discovers that the set of members has changed, it forms a new view with a new viewid and the new membership.
The change can result from rhe current process changing its request flag g_i, or some process timing out (count reaches 0).

Lemma 4.3. says that, if the execution is in fact nice, then Properties 4.1 and 4.2 of group membership hold.
This doesn't seem hard to believe, informally: a traversal collects all the correct information, and forms new views containing anyone whose $g_i = true$. If all the g_i's are fixed, then the view never changes again (after an initial traversal that collects all the correct information. LTTR.

The protocol also ensures that eventually the execution becomes nice.

6.6 Group multicast

They describe two common token-based approaches to group multicast:
1. Token circulates, carrying a sequence number for messages, which processes assign to particular messages and then increment.
2. Messages themselves are put into the token; the order determines the order of delivery that everyone sees.

Here they use the second approach: Maintain a queue of messages in the agent state.

Group communication algorithms in the literature make various different communication guar-

antees.
Here they require (in nice executions):

5.1. If p_i is a member of every view in the execution, then any message sent by a member of the group (any view) during the execution is eventually delivered to p_i.
5.2. Same order everywhere, for the messages that are delivered.

To achieve these goals, they let the agent accumulate an ordered queue of all the messages that any visited process wants to multicast.
They keep the message queue length bounded, by throwing out old messages after long enough has passed that they should have been delivered. (? I'm not sure this is quite what they are describing—what about the case where one view persists forever? Its messages should be cleaned up after a while, but they don't seem to say that.)

6.7 Resource allocation

Mutual exclusion, really.
They assume that anyone who gets the resource releases it within some known time.

They require mutual exclusion and no-lockout.

They show how to build this using the basic agent traversal and the group membership service (not the multicast service).
The basic idea seems to be that everyone who wants the resource joins a group $g_{resource}$.
Then the agent orders the members in the order in which they join the group.
And the agent allocates the resource to the process that is at the head of the request queue.
The process presumably learns about this when the agent arrives with itself at the front of the request queue.
When the process is done with the resource, it simply leaves $g_{resource}$.

Also in some other situations, the resource gets released, e.g., when the process who has it leaves the system (?), or certain types of partitions occur (see the notes about "primary components").