Middleware

Readings:
Middleware: C̄hen, Welch Self-stabilizing dynamic mutual exclusion paper
Dolev, Schiller, Welch Random walk for self-stabilizing group communication
Virtual objects D̄olev, Gilbert, et al. Geoquorums paper
Compulsory protocols H̄atzis, Pentaris, Spirakis, Tampakas, Tan. Fundamental control algorithms in mobile networks.
Chatzigiannakis, Nikoletseas, Spirakis. On the average and worse-case efficiency of some new distributed communication and control algorithms for ad hoc networks.
Chatzigiannakis, Nikoletseas, Spirakis. An efficient communication strategy for ad-hoc mobile networks.
Chatzigiannakis, Nikoletseas, Spirakis. An efficient routing protocol for hierarchical ad-hoc mobile networks. Next:

1 Last time: Self-stabilizing dynamic mutual exclusion

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

Goal is to produce a mutual exclusion algorithm that runs on a very badly behaved dynamic network.
With continual "churn", as before.
Also self-stabilizing, that is, it should tolerate occasional total system state corruption failures, and manage to recover back to normal operation, before too much time elapses.

I found the paper quite confusing. Many ideas intermingled, presented in a confusing order. Still, there's some ideas worth seeing.

Main protocol ideas:

1. 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 in the group, they keep getting chances to enter the critical region.
2. Use a token-circulation sub-protocol (similar to the LR protocol of Monday's paper) to circulate a token around the currently active processes (members of the group), and use this token to control access to the critical region.
3. Mechanisms to achieve self-stabilization:
   (a) Send messages, tokens repeatedly.
   (b) Enforce bounds on sizes of variables (e.g., counters).
   (c) Use timers, to time out old information.

1.1 System model

Digraph network (this is different from the others we've been considering, which have been undirected—I'm not sure what the significance of this change is).
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 place where we've seen the use of time in this series of papers. It's because of the self-stabilization.

$Assum_3$: Upper bound g on number of messages generated by a processor in each unit time interval.
(Maybe they don't need this assumption: Since processors have clocks or timers, perhaps they could just schedule their sending steps to satisfy this condition.)
And variables are all bounded-size (needed for self-stabilization).

The combination of assumptions ensures that, in stable operation, the total number of messages in the system is bounded; this makes it reasonable to assign messages ids from a bounded range.

Clocks:
They assume timers that decrease at the rate that real time increases.
They could just as well have said that they have logical clocks that increase at the same rate as real time, though the values aren't necessarily 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 consider local computation events and topology change events.
A topology change event instantaneously changes the topology; however, they make no assumption about notifications.
Processes will presumably have to execute sub-protocols to decide who their neighbors are.

$Assum_4$: 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—so why include it?).

Executions: The usual kind of definition, except that they don't have to begin in an initial state. Because the paper deals with self-stabilization, so any state can be the initial state of an execution.

1.2 The problem

Their variant of mutual exclusion involves join and leave requests, which announce that a process is or isn't interested in the critical section.
This suggests that it can remain a member of the group if it wants to submit a new request immediately after it leaves the critical region.

They want mutual exclusion, but not always: just after stabilization occurs.

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 automatically false.)

They list other, stronger progress properties:
Lockout-freedom (no starvation), or bounded waiting.
(These do not have the same problem in their statements, since they involve a particular process i that is assumed to be active.)

They claim that they can achieve any of the three progress properties, based on what they assume about the underlying token circulation algorithm.

1.3 Background: Dijkstra's and Varghese's algorithms

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, Varghese's for a static network with FIFO links.

In these algorithms, processes 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 it remembers in its current state, whereas everyone else checks that it is different.
When p_0 leaves 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 would like to adapt these algorithms to their dynamic network setting. Problems:

1. The simple local check used by D and V fails inthe current situation, because of the facts that links can be non-FIFO.
2. They don't have fixed ring—they have to construct a virtual ring, and that will change dynamically, based on changing topology and also join/leave requests.

3. Partitions—they don't want a process that gets disconnected from p_0 to prevent others from entering Crit.

1.4 The algorithm

1.4.1 Some motivation

To achieve self-stabilization, the distinguished process p_0 keeps generating mutual exclusion tokens (m-tokens), periodically.
In case one has been lost.

That's dangerous for achieving mutual exclusion: In token-based mutex algorithms, there's generally just one token, and only the token-holder can enter the critical section.
So they need to do something else here:

Tokens are issued with id numbers, and 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 will 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.
The check they do will be something like D and V's.

1.4.2 Some details

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?
It's based on the time that it takes a token to make one pass through the active processes in the network, giving everyone a chance to enter the critical region.
So it sounds like the lifetime of a token is really just one pass through the active processes in the network—it's not circulated repeatedly as in the original LF paper.

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, p_0 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 $ring$. It sounds like they use LRV, but they allocate the critical section in the order given by $ring$.)

Anyway, when a process is visited, it checks its local state against the token to see if it really has access to Crit.
The rule it usees here 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.
Moreover, the process has to be the first one on $ring$.

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 (long enough for everyone to have gotten access if things are behaving normally—note that this requires a bound on the time in the critical section).

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, confusing.

1.5 Conclusions

A key piece, self-stabilizing leader election, isn't here. How to do this?

2 Self-stabilizing group communication using random walks

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

2.1 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:

2.2 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.

2.3 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.

2.4 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.

2.5 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.

2.6 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 guarantees.
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.)

2.7 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").

3 Virtual Objects

[[[Seth to provide something here.]]]

4 Compulsory protocols overview

We have four papers on a common theme—compulsory protocols. The first paper briefly introduced the idea, along with two non-compulsory algorithms. The other three, by the same three authors, develop the idea in several directions. The contents of the papers overlap quite a bit.

A compulsory protocol, for an ad hoc network, is one in which some or all of the mobile nodes move as directed, under algorithmic control, rather than just going where they would like.

They focus on compulsory protocols for simple message routing.

4.1 The network model

They consider m mobile nodes moving in 3D space, with uids.

They consider the space partitioned into n cells that they call "cubes". Let's call them "cells".
Cells are chosen to be small enough so that, if a mobile host transmits while it is within a cell, its message is guaranteed to be received by any other hosts in the same cell.
Transmission delays are regarded as negligible.

They construct a network graph $G = (V, E)$, where the vertices V are the cells, so $|V| = n$.
And E gives adjacency relationships between cells.
In this way, they abstract away from the particular geometry.

They assume that the cells are arranged in a regular pattern in 3-space, such as a grid.
Moreover, the degree of the graph is bounded by a constant; for a typical 3D grid, the number of neighbors would be 6, one per face.
That implies that $|E| = \epsilon$ is linear in n.

They intend that n should approximate the ratio between the volume of the total space and the volume covered by the transmission radius of a single mobile node.
n is a good measure of the size of the space.

4.2 Compulsory protocols

They classify MANET protocols as:

- Non-compulsory: The mobile nodes travel anywhere they like.
- Compulsory: The mobile nodes travel where the algorithm says they should.
- Semi-compulsory: A subset of the mobile nodes, called the "support nodes", or just the "support", travel according to algorithm control; the non-support nodes go where they want.

Having some nodes under algorithm control can help achieve needed connectivity, help with tasks like message delivery.

Related work: Li and Rus: Algorithm for message delivery.
Forces all the mobile hosts to deviate slightly from their planned trajectories.

So, in CNS terms, their protocol is compulsory, but it limits the moving directions that the algorithm gives to the mobile nodes.
Their algorithm works for "deterministic host routes" (meaning?)
They also show an optimality result for message transmission times.

Another influence: The "two-tier principle" articulated by Imielinski and Korth in their early book on mobile computing.
It says take advantage of (move computation and communication to) fixed parts of network whenever possible.
They regard the support as analogous to a fixed part of the network.

For this group, the idea of compulsory protocols seems to have arisen in the first paper, by Hatzis et al., as an afterthought to a non-compulsory algorithm. So we'll cover this result next.

5 A non-compulsory protocol for leader election and node counting

From Hatzis paper.

The problem is to elect a unique leader among the m mobile nodes.
It should learn it is the leader, and others should learn they are not the leader.
Moreover, the leader should learn the total number n of mobile nodes.

In addition to the model assumptions listed earlier, they also assume the nodes know their geographical location, and hence, can tell which cell they are in.

5.1 The protocol

Very simple.
Every mobile node keeps a local counter, initally 1.
Whenever two nodes meet, they engage in a pairwise protocol, in which the one with the higher id wins and the other loses.
The winner remains active, whereas the loser becomes inactive.
The winner absorbs the loser's count by adding it to its own count.

They also augment the protocol to keep track of lists of ids of nodes that have been collected, instead of just the count. Of course, that leads to larger messages, hence more communication.

For some (not completely convincing) reason, they restrict the nodes so they participate in such an encounter only when the nodes "enter a new cell".
The idea here is to conserve battery power, but I'm not exactly sure how that would work: Should the nodes stay in sleep mode except occasionally, when they first enter a cell?
How would two nodes know when to wake up in synchrony, then?

Protocol correctness seems obvious, as long as messages can't get lost, nodes can't fail, etc.
However, the protocol has no resilience at all.

5.2 Protocol analysis using the random walk assumption

Now they want to analyze the expected time for the algorithm to complete.
Of course, it's impossible to bound this, because it depends entirely on how the nodes are moving.
So, you see, they would like to be able to tell the nodes how to move...so they could obtain some guarantees.

But here, for their analysis, they simple assume that each mobile node performs an independent random walk on the network graph G.
They claim that this is not such an unreasonable assumption, given that lots of other papers (esp. systems papers) assume the random waypoint model (which is a kind of random walk).

The analysis looks quite straightforward, based on Markov chain analysis techniques.
First, they observe that they can restrict attention to the case where m, the number of mobile nodes, is exactly equal to n, the number of cells.
This is OK because:
1. If $m < n$, they could augment the algorithm with dummy nodes, thereby bringing the number of nodes up to exactly n.
And, they could assign all these nodes ids strictly less than all those of the real nodes.
Then the dummy nodes would not affect the behavior of the real nodes in the protocol.
So, the time for the real m nodes to complete is no greater than the time for all n real + dummy nodes to complete.

2. If $m > n$, then some of the m's start out in the same cell. They will meet immediately and become reduced to one node, thus reducing the cost to that for the cases where $m \leq n$.

So, restrict attention to $m = n$.

Lemma 3.2. calculates a bound on the probability that $M_{i,j}$, the meeting time of two mobile nodes that start in cells i and j, respectively, is bigger than some given time value (real number?) t.

The analysis says that that decreases exponentially in t; specifically, they get $1/e^{t/em*}$.

Here, e is the usual constant.
And $m*$ is the maximum expected value of $M_{i,j}$, where the maximum is taken over all pairs i, j.

So all they seem to be doing here is relating the value $M_{i,j}$ for any particular i, j to the expected value of $M_{i,j}$ for all i, j. Well, OK...

Theorem 1 does a bit more:
It bounds the expected time to finish, in terms of the maximum, over all i, j, of the expected time $E_i(T_j)$ for a host that starts in cell i to reach cell j.
Specifically, they bound the expected time to finish by $O(\log(n) max_{i,j} E_i(T_j))$.
They use a series of equations, LTTR.

Then in Corollary 3.1, which they give without any proof, they bound this further in terms of the number of edges in the graph.
Specifically, Corollary 3.1. says that the expected time to finish is $O(\log n\epsilon)$, that is, the log of the number of cells times the number of edges.

The key step here seems to be the claim that for every i and j, $E_i(T_j)$ is bounded linearly in terms of the number of edges.
Sounds plausible—maybe standard for random walks—but no proof here.

They also tighten their bound, in Theorem 2.
Apparently Theorem 1 allowed for the winner to meet all the other hosts one at a time.
But in most executions, the time it takes to meet the other hosts would overlap.
Taking this parallelism into account, they show that the expected time to finish is actually just $O(\epsilon)$, or $O(n)$ (linear).

Finally, they add termination detection, based on timeouts.
This implies that the nodes have clocks, and that they know when the protocol is likely to have completed.

5.3 Bells and whistles

Now they think of the idea of compulsory protocols.
Instead of assuming the nodes walk randomly, they might assume that the algorithm is allowed to tell the nodes where to go.
And of course, what it would tell them to do is to walk randomly.
Then of course the previous analysis results apply.

Anonymous networks:
If nodes have no ids, they can choose them randomly.
With high likelihood, they will choose different ones, and then the protocol works as before.
If two choose the same, then when they meet, they can "refine" their choices to choose different ones.

(E.g., they could add lower-order random choices, rather than choosing entirely new ids, and use lexicographic ordering on the pairs.
This seems like a good idea, because otherwise, they could ruin the property that the largest id wins (which could be interesting, though it's not parto of the problem requirements.

Simulation results: Nothing interesting—they just back up the theoretical results.

Further work: They suggest using similar strategies (by which they must mean random walks, and might mean compulsory protocols) for other problems like routing, coordination, termination detection, failure detection.

6 Semi-compulsory protocols for message routing

Working from the other three papers. They present a generic semi-compulsory protocol idea, then specialize it in two ways: to a Snake protocol and to independent Runners.

6.1 Some motivation

Problem definition: Send a message from some sender mobile node S to a receiver mobile node R.
In some places, they also mention that they want to notify S that its message has been delivered;

but then they seem to forget about this, so we will too.

They make a breezy claim that "No distributed algorithm can be implemented in ad-hoc mobile networks without solving this basic communication problem."
I'm not sure what they mean here; it suggests some impossibility claim...but it can't be right, in general...consider sensor nets in which the individual nodes have no importance—just the data they are collecting about the real world environment.

They discuss previous solutions to message routing in MANETs (DSR, AODV, TORA, LAR,...).
Some problems with these:
—Some require flooding.
—They require constructing and maintaining data structures, which might not work well if the network is changing rapidly.
—They are expensive in terms of communication.

They conjecture a kind of "impossibility result":
Any algorithm that tries to maintain a global structure with respect to the temporary network will be erroneous if the mobility rate is faster than the rate of updates of the algorithm.
Well, at a certain level, this sounds right; can we turn this into an actual impossibility result?

So they are looking for a better approach:
—Should work very well in a rapidly changing network.
—Require little overhead.
—Use local information only.
—Deliver messages fast.

For all of this, they are willing to require certain designated nodes to move under algorithm control.

6.2 The generic protocol

Support = the nodes whose motion is controlled.
They assume some number k of support nodes.

In general, the support nodes move somehow through the network graph.
CNS abstract this behavior into a "support motion subprotocol" P1.

When a sender S is near a support node, it gives its message to the support node (using another protocol P2).
The message is stored "somewhere within the support structure".
When a receiver R is near a support node, it gets notified about a message waiting for it, and gets the message.

How are the messages managed within the support (that is, among the support nodes)?
They are propagated among the support nodes when two or more support nodes come within communication range.
How the support nodes exchange and maintain this information is controlled by a "synchronization subprotocol" P3.

So, in general, the support is some kind of moving skeleton subnetwork.
They leave things flexible, allowing different protocols for P1, P2, P3.

6.3 The Snake protocol

6.3.1 Protocol description

The protocol involves a collection of k nodes, for some (parameter) k, with an established ordering.
The first node, called the "head", does a random walk, at each step visiting a neighboring cell chosen uniformly at random.
The remaining nodes follow along behind the head, in order.
They all move at the same speed (in discrete steps).
Thus, after each step, each node occupies the cell that its predecessor did after the last step.
(They don't need common sense of orientation—they just need to be able to choose a random direction, or move to someone else's cell.)

This is used to send and deliver messages, as described above for the generic protocol:
S waits until any node of the snake is within range, then transfers its message to the snake.
The snake transfers the message among its own nodes, according to some protocol, e.g., full replication.
Since the snake is always connected, these nodes can manage data however they like.
Then, when any node of the snake is within range of the message's target R, it tells R message for it, and actually delivers it.

They also describe how to set up the snake initially.
They assume that the k support nodes who they are.
Somehow they have to coordinate to elect one to be a leader, who becomes the head of the snake.
It takes charge of ordering the other $k-1$ support nodes into a list, and telling them their positions.

6.3.2 Evaluation

They claim that this algorithm ensures coverage of the whole network (with high probability), within bounded time; they analyze this time, based on random walk analysis results.
The analysis does not assume that the non-support nodes are also doing random walks.
Rather, they can move any way they like, as long as they aren't behaving "adversarially", trying to avoid the support nodes; more precisely, they require that the non-support nodes' motion is independent of that of the snake.
Then just the fact that the snake is doing a random walk is enough to get their guarantees.

We'll come back to this analysis in a minute, after introducing the other special case: the Runners protocol.

Advantages of the Snake algorithm:
They claim it achieves very fast communication between any two mobile users.
With low communication overhead, no elaborate state, simple local processing.
The nodes don't actually use any location information.

Disadvantage:
It requires compulsory motion of the support nodes.

6.3.3 Bells and whistles

Significant modification:
The head does a random walk on just a spanning subgraph of the network graph.
Since the network graph is fixed, it's possible to define such a subgraph once and for all—it doesn't have to be maintained in the presence of changes.
This seems to improve performance, as determined by simulations and also by analysis.

Robustness: They also give a robustness "theorem" for the Snake protocol, but it requires modifications to the algorithm.

Theorem: A revised version of Snake tolerates failure of up to one support host.

The revised version allows the snake to split into two, then when the head H2 of one snake, snake2, happens to encounter any node of the other, snake1, H2 splices its entire snake2 inside snake1 at the point of the encounter.

They note that this trick doesn't work for more than one failure—since a cycle could form.

6.4 The Runners protocol

Simply allow k support nodes to perform k independent random walks, synchronizing whenever they meet.

Apparently this performs well—some of their experiments say it performs better than the Snake.
Apparently increasing k has a strong impact on the expected delivery time—that seems to make sense, since there are more changes for S (or R) to encounter a support node if they are moving independently than if they are moving together...

In contrast, increasing k beyond a certain point—around $\sqrt(n)$—doesn't seem to help much in the Snake protocol.

They discuss a 2-phase commit protocol, which is used for runners that encounter each other to exchange their information.
This suggests that they think some kind of consistency will be needed for this synchronization—so they are using a fairly heavyweight consensus (commit) protocol here.
But it is not clear what consistency guarantees they require and why.
(The commit-style subprotocol is standard: one of the runners that has met (e.g., the one with the lowest id) takes charge of collecting everyone's information and then broadcasting it to everyone.)

Robustness:
Theorem: Runners tolerates failure of up to $k - 1$ support hosts.
Because the remaining nodes just continue their traversals as before.

Experiments: They compare Snake and Runners.
Only a small support is required for each.
However, Runners did better than Snake in almost all cases—so maybe it's just a better algorithm.

6.5 Analysis of the Snake protocol

The best source here is CNS's DISC paper—it's on just the Snake protocol.
The major point of the paper is that the non-support nodes can move arbitrarily, "provided that they don't deliberately try to avoid the support", and their analysis results still hold.

What exactly does this mean? That they move according to a pre-defined, oblivious deterministic or randomized strategy?
The authors don't actually say whether the non-support nodes' movement strategy is allowed to adapt to what they encounter during execution—so I will presume that it can't.
Thus, I will assume that each non-support node has a predefined walk strategy, that doesn't adapt at all to what it sees during execution.
To anything, not just to the support's movement—since I don't know how to formalize dependence on the support's movement.

Their analysis results consist of guaranteed expected time bounds for comunication from S to R—guaranteed for arbitrary motion patterns for the non-support nodes.
These time bounds don't depend on the number of non-support nodes, nor on the initial placements, nor on the movement strategy for S and R—just on the size of the network graph.

Their proofs are rather heavy on the Markov analysis.
In particular, they use a fundamental notion of "strong stationary times" of reversible Markov chains.

They restrict attention to properties of the random walk performed by the head, without worrying about the rest of the support hosts.

(They define:
$p(i, j)$, transition probabilities for the head (probability that, from i, it moves directly to j)
$P_i(E)$, the probability that the walk satisfies a property E, given that it starts at vertex i.
T_j, first "hitting time" when the walk reaches vertex j.
$E_i(T_j)$, expected value of T_j, for walks that begin at cell (vertex) i.)

In analyzing random walks on graphs, it's convenient to define π, the stationary distribution.
Markov theory says that, after a "sufficiently long" time t, a walk starting anywhere will reach the various vertices with probabilities that are given by the stationary probabilities (actually, these probabilities approach the stationary probabilities in the limit).

Moreover, the stationary distribution has a nice formula: $\pi(i)$ for any vertex i is just $deg(i)/2\epsilon$ (recall ϵ is the number of edges).
(Check that these sum to 1, as they should for a probability measure.)
(This says that the vertices have probabilities of being visited that are strictly proportional to their degrees. More neighbors, more chances of being visited.)

$p_{i,j}(t)$, the probability that a walk started at i will be at j at time t.

Now they state their main theorem, Theorem 1:
The statement is a bit confusing.
It seems to be saying, first, that the algorithm "guarantees" communication from S to R in finite time.
Presumably, they must mean that with probability 1, it eventually succeeds.
But this is a consequence of what I understand to be the second statement, which gives a bound on the expected time for this communication.
Actually, they don't give the bound in the theorem statement—they just say that some bound exists, and that it's a "function of the motion space size" (equivalently, of the number n of cells).

Proof: I found it confusing—didn't completely follow.
They analyze the time it would take for a randomly-walking snake head to meet the node S.
Then the time to meet R is symmetric.
And they can also add in some time for communicating the message among the snake members.

To analyze the time for a randomly-walking head to meet S, then define EM, the expected time of the first meeting.
What is this expectation taken over? It must be over the random choices in the random walk, and also, any random choices made by S in its walk strategy (this is allowed).
But it is not randomizing over the starting positions for the head and S, nor over the choice of walk strategies S might be using; they want to allow all possibilities here.
So, they define $m* = sup(EM)$, taking the worst case expected meeting time over all starting positions for the head and S, and all walk strategies for S.

So, fix the starting positions and the S strategy; try to bound EM for this combination.

They again use the stationary distribution π.

I don't completely follow this analysis.
So I'll just try to give a high-level idea of what I think I understand.

Note that the distribution of the head's position approaches the stationary distribution in the limit.
They consider approximations to this. Namely, they claim there exists a (sufficiently large) time u such that, for all vertices i, j, the probability that the head's walk, starting from vertex i, is at j at time u is at least $(1-1/e)\pi(j)$, that is, it is within $1/e$ of the correct stationary probability $\pi(j)$.

They consider the sequence of times $u, 2u, 3u, \ldots$.
They invoke the independence assumption, top of p. 11, to conclude (somehow) that, at each of the times in this sequence, the probability that the head is on the same vertex as S is at least $(1 - 1/e)min_j(\pi(j))$, an approximation to the minimum stationary probability of any vertex.

This is at least plausible: if j is the vertex that S happens to be visiting at one of these times du, then the stationary probability $\pi(j)$ is at least this min expression.
And the random walk by the head has approximately this stationary probability's chance of being there at the same time.
(The analysis here seems to be neglecting the approximation issue—I'm not sure if it's a mistake.)

Since the head has at least probability $(1 - 1/e)min_j(\pi(j))$ of meeting S at every time that is a multiple of u, the expected number of multiples of u that have to elapse before the head meets S is at most $1/(thisexpression)$.
Thus, the expected time is at most $(u(e/(e-1)))/min_j(\pi(j))$.
Now they define $c = u(e/(e-1))$, and thus rewrite the bound as just $c/min_j(\pi(j))$.

Since each stationary probability $\pi(j)$ is equal to $degree(j)/(2\epsilon)$, it is $\geq 1/(2\epsilon)$.
Plugging this into their bound expression yields a bound of $2c\epsilon$.
Their final conclusion, restated in Corollary 1, is this upper bound on expected time of $2c\epsilon$.

In interpreting this bound, notice that ϵ is a parameter of the graph (number of edges), and so is $c = u(e/(e-1))$.
(This depends on the Markov stationary probabilities for this particular graph).
So, the dependency is as they say—only on the graph parameters, and not on the starting locations or S's strategy.

But, they don't give a bound for u here—just say it's a parameter of the graph.

But all this just analyzes the expected meeting time between the head and S (or R).
It doesn't say anything about the time to communicate the message within the snake.
So this analysis seems incomplete—surely that involves some extra cost.

Protocol time efficiency properties:
Now they use the previous analysis of the meeting time for the head and S to get a bound on the communication time from S to R, taking the entire snake into account.
There are two additions here: getting a better bound on the meeting time, and actually incorporating the bound into a communication cost analysis.

First, consider the meeting time—for the snake to meet S (or symmetrically, R).
The snake has k nodes—so we would expect some improvement in the expected time to meet S, since now it has to meet any one of the nodes in the snake, not necessarily the head.
However, since the motion of the k support nodes is tightly coupled, the improvement might not be as pronounced as if the nodes were doing independent random walks.

What they get from the larger number k of nodes is essentially, a dynamically-changing reduced graph.
At any time, the (approximately) k vertices on which the support nodes reside can be regarded as one "super-vertex" in a reduced network graph.
All of these vertices' neighbors become neighbors of the one super-vertex.
The super-vertex's larger degree means it has a larger stationary probability.
Which should translate into a larger chance of meeting S.

They claim they can modify their previous analysis, to now use the randomly-moving super-vertex instead of the randomly-moving snake head.
They seem to be claiming that, for any super-vertex, the degree is at least k (instead of 1 as before).
???
So their upper bound on the meeting time now becomes $2c\epsilon/k$.

That's just for the meeting time.
To get the communication time in this setting, they have to add in an $O(k)$ term—because of the time needed to propagate the message through the support.
So they get something like $2(2c\epsilon/k) + O(k)$.

They claim to optimize this expression when $k = \sqrt{2c\epsilon}$, yielding an overall bound of $O(\sqrt{c\epsilon})$.

Spanning tree bound:
Finally, they also claim a (better) bound for the case where the snake traverses a spanning tree rather than the whole graph: $O(n)$.
Proofs not in this paper, though. LTTR.

6.6 Discussion

They also claim a lower bound on walk time, which looks interesting. But I'll leave it to the reader—anyway, the proof of a key lemma isn't here.
The conclusion is that the expected time for hitting S (for some choice of starting positions and S strategy) is at least $(n-1)^2/2\epsilon$.

Future work: Study some forms of constrained motion for the non-support hosts.

7 Hierarchical routing protocol

From Chatzigiannakis, Nikoletseas, Spirakis: An efficient routing protocol for hierarchical ad hoc mobile networks
And also, the POMC paper.

7.1 Overview

This paper presents an embellishment of the Snake protocol.

They introduce a special-case model for ad hoc networks that is based on areas they call "cities", connected by long-distance links called "highways".
In the cities, mobile nodes are dense and move fairly randomly.
On the highways, nodes are sparser but their motion is much more predictable.
The highways are traversed fairly frequently.
They claim that such networks are common.

They address the problem of routing a message from one mobile node to another, where the mobile nodes may be in different cities.
The mobile nodes that live in the cities (including S and R), are assumed *not* to travel on the roads—that capability is reserved for special "highway mobile nodes".

The main idea is to use the snake framework within cities, in order to route a message within the city, between specific S or R mobile nodes in the city and a particular "access port location" in the city.
Highway nodes carry the message from the access port of one city to access ports of the other cities.

The access port location is on the highway, and gets visited frequently enough so that, with high probability, a message arriving at the access port from within the city (via a local snake) will be synchronized with a highway node arriving at the port from the highway.
They require the other direction too: high probability that a message arriving there from the highway will be synchronized with the local snake, to allow the message to be picked up by the snake and delivered to the right recipient within the city.

Much of the paper deals with the 2-city case; if there are more cities, their protocol involves essentially flooding the message to all cities, where they will be circulated around via their snakes. That is, unless there is some global knowledge of which city a particular destination resides in.

They give simulation results, which they claim show really good performance. However, they are only comparing their protocol to the earlier snake protocol, when run throughout the whole network.

7.2 Model of hierarchical ad-hoc mobile networks

They assume 3D.

Dense city subnetworks, each with a "city graph", which is just like the overall network graph in the Snake paper.
In this graph, each vertex corresponds to a "cell", or "cube"; it must satisfy the condition that anyone in the cell who sends a message is heard by everyone else in the cell.

Each city has a special location called an "access port", which is its (unique?) point of connection to the highway.

They divide the mobile nodes into two categories: City nodes, who remain within cities, and highway nodes, who only traverse the highways.
They assume that highway nodes traverse the highways fairly frequently.
Highway nodes are not controllable—no compulsory motion; rather, the protocol should try to take advantage of their predictable motion.

They use a discrete time (slot) notion.
They assume a lower bound p on the probability that, at a given time (slot), some highway mobile user is at a city's access port.

This notion is a little unclear.
They say they assume probability p (a constant) that, at any given time, "the exchange of information by the higher layer is available" at an access port.
What does this actually mean? It seems like two different things:

1. A guarantee that, at any given time (slot), some highway node is at the access port (available to receive a message from the support). The reasonableness of this guarantee depends on the density of travel on the highways.
2. A guarantee that, at any given time (slot), the support is at the access point (available to receive a message from the highway node).

This is a different sort of guarantee from the first one above. Its reasonableness depends on the size of the city and the size k of the snake.)
They don't explicitly say this is what they mean. But the only other interpretation I can see is that some mobile nodeis sitting permanently at the access point. ???

7.3 Their protocol

They use a Snake within each city.
For the highways, they piggyback messages on the predictably-mobile highway nodes.
They don't need compulsory motion on the highways—they just rely on whoever happens to be traveling there.
They claim that the regular (not random) movement on the interconnection highway helps the communication time quite a bit, over random routing via a snake.

More details:
Each city has one snake, doing a random walk as before.
Assume that sender S has a message to send to another node R not in the same city.
(If in the same city, the usual snake protocol will work.)
Then:

1. When S is within transmission range of the local snake, it gives its message to the snake.
2. When the head of the snake arrives at the access port, then if it happens to meet a highway mobile node there (which it does, with probability at least p), it hands off the message to the highway mobile node.
   If not, then the snake keeps moving randomly until it happens to return to the access port, and keeps on doing this until it succeeds in meeting a highway node at the port.
   (So it doesn't sound as though anyone remains permanentaly at the access port.)
3. The highway mobile node moves according to its regular movements on the highway, to the other city's access point. (Here, they assume there is only one other city, though they claim that their ideas generalize to more cities.)
   When the highway mobile node reaches the other city, they say that, again with probability p, it meets the other city's snake.
4. The support in the new city delivers the message to R, during its usual random walk.

This is for two cities. They also talk about "modularity", by which they mean that their algorithm extends to any number of cities.
For finding a target node in an unknown location, they simply have the highway node drop off the message at all cities.

7.4 Analysis

They try to explain informally why the original snake protocol wouldn't behave well on the hierarchical network.

I'm not even sure how the snake is supposed to work in this case.
What happens when it reaches an access port? Does it count the highway as one of the adjacent

edges and include it in its uniform random choice of next place to go?
What if no highway node is there at the time? Then choose another direction?

They claim that there is only a small probability that the snake would pass through the access port and head to the right city. ??? They claim that, in contrast, the hierarchical algorithm guarantees, with high probability, that within a small number of visits to the access port, the message will be successfully handed off.

Another reason why the hierarchical protocol should behave better than the pure snake protocol is the much greater opportunities for concurrent processing in the hierarchical protocol.
The various snakes can continue their work traveling around cities, collecting and distributing messages, all in parallel, and in parallel with the useful work done by the highway nodes in conveying their messages between cities.
Another reason the highway protocol does better is that it brings the message to all cities, essentially in parallel, and then it gets distributed throught all the cities in parallel, whereas the snake would bring it to only one at a time. Again, a matter of concurrency.

They carry out an average analysis, assuming that all the snakes are performing random walks, and also that the other city nodes are doing random walks.

I'll skip the bounds—the analysis is somewhat different from the one in the other paper.
Using a support of size $\sqrt{n}$ for each city, where the city has n cells, is optimal; with this, they get linear average message delays, linear in n.
(In this analysis, they assume the delay on the highway is a constant.)

7.5 Experimental results

For hierarchical graphs, they claim that the hierarchical protocol is much better than the pure Snake protocol.

Their experiments involved one fixed S in one city and another fixed R node in another city.
One access port in each city, one highway connecting them.

Though S and R are fixed, they allow S to send many messages to R.
So, obviously, their algorithm can take advantage of all the concurrency, whereas the original algorithm had very little—only one snake!

For the new algorithm, the key factor in determining expected time turns out to be the probability p.
But, $p = .3$ is good enough.

8 Echo algorithms for leader election and counting

From Hatzis, Pentaris, Spirakis, Tampakas, Tan, 1999: Fundamental Control Algorithms in Mobile Networks.
This is for the cellular model.
But it might provide ideas for similar algorithms in mobile networks, if we use a sufficiently high

level of abstraction (e.g., use virtual base stations).

The problem addressed here is leader election among mobile nodes.
As a secondary, closely related problem, they consider the accumulation of the exact total count of nodes in the system.

They give a simple protocol whereby the fixed network collects the counts using a fairly standard Echo-style protocol.

8.1 Introduction

Usefulness of leader election and node counting:

1. For applications: Inventory applications, animal population monitoring, traffic monitoring,...
2. For network control:
   Could use them in constructing routing protocols, managing data, etc., using a layer organization wherein these services are built on top of the leader-election and counting services.

   Maybe could also use them for low-level network monitoring and control, e.g., for topology control?
   This suggests a layer organization wherein leader election and counting run as "coroutines" with the network control services (since they may depend on each other).

8.2 The model, the problem

Mobile networks with fixed base stations (Mobile Service Stations—MSSs), each controlling a cell.
Mobile nodes move around to various cells.
While in a cell, a mobile node communicates only with that cell's MSS.

When a mobile node enters a cell, it sends a join message to the MSS.

They model the fixed network of MSSs as an undirected graph $G = (V, E)$, where $|V| = n$, here the number of MSSs, and $|E| = \epsilon = O(n^2)$.
The edges here represent fixed wired links.

They also assume m mobile nodes.

The problem:
One of the mobile nodes initiates the algorithm to find the total number of mobile nodes (or elect a leader).

8.3 The protocol

The protocol works in two "tiers", with the fixed coordinators at the higher tier, managing the mobile nodes at the lower tier.

The basic organization is "Echo" style, by which they mean that a central root node starts a broadcast on a fixed tree network, followed by a convergecast back.

The particular Echo protocol used here assumes a fixed tree of the wired network of MSSs. It involves four Echo phases.

To begin the algorithm, the initiator mobile node sends a message to its local MSS, telling it to be the coordinator of the algorithm.
The coordinator MSS broadcasts a ¡count¿ message in its cell.

1. Echo phase 1:
   The coordinator starts an Echo containing a $< count_tok >$ message, along the tree of MSSs.
   Each MSS that receives the $< count_tok >$ message continues the Echo by propagating the $< count_tok >$ message along the tree.
   In parallel, it broadcasts a $< count >$ message in its own cell.

   After the completion of Echo phase 1, every MSS knows about the execution of the counting algorithm and has broadcast a $< count >$ message in its cells.

   Now, some activity goes on concurrently with this phase, and continues afterwards.
   Namely, when a mobile node receives a $< count >$ message, it responds with a $< count_me >$ message containing its own id.
   Each MSS keeps track of the counts it collects in this way, in a variable $size$.
   Also, if a new mobile node arrives in a cell, it sends a $< join >$ message, to which the node responds with another $< count >$ message, so that the new mobile node will have a chance to respond with a $< count_me >$.

   Note that no mobile node sends more than one $< count_me >$, ever, thus preventing double-counting.
   But note that this isn't resilient to lost messages.
2. Echo phase 2:
   Now the initiator MSS sends a $< size_tok >$ message in an Echo wave.
   In the convergecast part of this phase, each MSS sends its current value of size up to its parent; when it does so, it no longer sends any more $< count >$ messages.

   At the end of Echo phase 2, the initiator MSS should know the total number of nodes in the network.

   Questions:
   A possible problem: A mobile node might have been missed because it moved from cell to cell during the execution of the protocol.
   At what point does an MSS stop accepting new $< count_me >$ messages?
   Presumably after it has sent its size value upwards, towards its parent.
   But then can't some arrive after this point?
3. Echo phase 3:
   Next, the initiator MSS broadcasts an $< inform_tok >$ message in another Echo wave.
   This message contains the determined size.

Each MSS broadcasts this within its cell.

At the end of Echo phase 3, all the mobile nodes, as well as the MSSs, are supposed to know the (same) size estimate for the network.

4. Echo phase 4:
   A final phase, in which the initator informs everyone about the completion of the counting.

The algorithm can be modified to elect a unique leader, e.g., the one with the max id.

Lemma 2.1. says that the algorithm correctly counts all the mobile nodes.
We already know that no one gets double-counted.
So the key issue here is showing that every host in fact gets counted somewhere.
They argue this, though I don't find the argument convincing.
It doesn't seem to be an actual proof, but just an example of a particular type of motion.

They talk about a host that moves from MSS S1 to S2, who "finish their execution of the protocol" (what does that mean?) at times t1 and t2 respectively.
They describe one case where a host moves from S1 to S2 (but this is not a general case—just one possibility).
But, to cope with this, it sounds like they are modifying the protocol. So this may be wrong...needs fixing.

HW exercise: Fix this?
Chandy-Lamport-style (or Fischer-Griffeth-Lynch-style) snapshot ideas may be useful here.

They give a nice time bound, linear in the diameter of the fixed network.
They also analyze a more abstract "cost" mesaure, which turns out to be linear in the number of edges in the fixed network.