Virtual Nodes

Readings:
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.
Virtual nodes:
Virtual mobile nodes paper
Virtual stationary nodes paper
Home location paper (optional)

Next time:
Jim Aspnes talk. Two papers by Aspnes et al, from the schedule, on population protocols. Other three worth skimming too: Nath et al synopsis diffusion, Srivastava et el. aggregation methods; Patt-Shamir.

1 Compulsory Protocols

1.1 Overview

Four papers covering compulsory protocols. The first paper, by Hatzis et al, briefly introduced the idea, along with two non-compulsory algorithms. The other three 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.
More precisely, 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.

They focus on semi-compulsory protocols for simple message routing.

Network model: m mobile nodes moving in 3D space, with uids.

They consider the space partitioned into n 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 negligible.

Network graph $G = (V, E)$, where the vertices V are the cells (so $|V| = n$), and E gives adjacency relationships between cells.

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.
So $|E| = \epsilon$ is linear in n.

Thus, n is a good measure of the size of the space.

1.2 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.

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.3 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.

1.3.1 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.

1.3.2 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.

1.3.3 The Snake protocol

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.

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.

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.

1.3.4 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.

1.3.5 Analysis of the Snake protocol

Best source here: the CNS DISC paper, on just the Snake protocol.

The major point of the analysis is that the non-support nodes can move arbitrarily, "provided

that they don't deliberately try to avoid the support".

What exactly does this mean?
I will take this to mean that each non-support node moves according to a predefined, oblivious, deterministic or randomized strategy.
Thus, its movement doesn't adapt to what it encounters during execution.

They prove guaranteed expected time bounds for communication 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 the 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.

In analyzing random walks on graphs, it's convenient to define π, the stationary distribution.
Markov theory says that, after a "sufficiently long" time, 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.)

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

Theorem 1: Their main theorem.
Its statement is a bit confusing.
It seems to be saying, first, that the algorithm "guarantees" communication from S to R in finite time; but presumably, they mean that with probability 1, it eventually succeeds.
However, this seems to be simply a consequence of 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" n.

Proof of Theorem 1: I 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, they define EM, the expected time of the first meeting.
This expectation is taken over the random choices in the random walk (and also, over any random choices made by S in its walk strategy).

It is not randomizing over the starting positions for the head and S, nor over the S's possible walk strategies—they want to allow all possibilities here.
So, they define $m* = sup(EM)$, that is, the worst case expected meeting time, taking the worst case over all starting positions for the head and S, and all walk strategies for S.

For the analysis, we may fix the starting positions and the S strategy, and try to bound EM for this combination.

They use the stationary distribution π.
I don't completely follow this analysis, so I'll just try to give a high-level idea.

Note that the distribution of the head's position approaches the stationary distribution π in the limit.
They consider approximations to π: 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, $p_{i,j}(u)$, is at least $(1-1/e)\pi(j)$, that is, it is within $1/e$ of the correct stationary probability $\pi(j)$.
It follows from the independence assumption, top of p. 11, that the probability that the head is on the same vertex as S at time u is at least $(1-1/e)min_j(\pi(j))$ (since this min captures the minimum stationary probability anywhere in the graph).

This is at least plausible: if j is the vertex that S happens to be visiting at time u, 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.)

Then they consider the sequence of times $u, 2u, 3u, \ldots$.
They conclude that, at each of the times in this sequence, the probability that the head is on the same vertex as S is at least $(1-1/e)min_j(\pi(j))$.

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 $1/(this)$.
Thus, the expected time is at most $u/(this) = (u(e/(e-1)))/min_j(\pi(j))$.
Defining $c = u(e/(e-1))$, they rewrite this 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 $2c\epsilon$.
END OF PROOF OF Theorem 1

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 bound depends, 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.

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 is incomplete—see below.

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 $2c\epsilon/k$.
This is kind of rough...

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.

1.4 Hierarchical routing protocol

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

1.4.1 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.

1.4.2 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. ???

1.4.3 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.

1.4.4 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.)

1.4.5 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.

2 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.

2.1 Introduction

Uses 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, performing topology control, etc., building these services on top of the leader-election and counting services.

2.2 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.

Fixed network of MSSs is 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).

2.3 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: 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.

   Some activity goes on concurrently with this phase, and continues afterwards:
   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 $< count_{me} >$.

No mobile node sends more than one $< count_{me} >$, thus preventing double-counting.
(But 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; after it does so, it no longer sends any more $< count >$ messages (and presumably, doesn't accept any more $< count_{me} >$ messages.

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

   Question: Can a mobile node be missed because it moves from cell to cell during the execution of the protocol?
   Could it arrive in some cell after the MSS has already sent its size upward, yet not have been counted at its previous cell? Depends on communication assumptions?

3. Echo phase 3:
   The initiator MSS broadcasts an $< inform_{tok} >$ message in another Echo wave, containing the final determined size.
   Each MSS broadcasts this within its cell.

   At the end of Echo phase 3, all the MSSs and mobile nodes 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 do claim to 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 t_1 and t_2 respectively.
They describe one case where a host moves from S1 to S2 (but this is not a general case—just one possibility).
And to cope with this, it sounds like they are modifying the protocol. So this may be wrong...needs fixing.

They give a nice time bound, linear in the diameter of the fixed network.

3 Virtual Mobile Nodes

[[[Seth should contribute something here.]]]

4 Virtual Stationary Automata

Another approach to masking mobility is to impose a static network over the mobile nodes.
Instead of taking the view that taming mobile ad-hoc networks reduces to abstracting away location and just providing point-to-point routing, we embrace locality by providing a base station-like abstraction.
Our VSA layer overlays a stationary network of timed machines on top of the mobile nodes in the network, and allows interaction between the mobile nodes and the virtual machines.
This overlay network allows us to deploy a number of algorithms that were designed for base station networks and even wired networks, helping simplify the task of programming ad-hoc networks.
To simplify application design, we propose to impose on mobile networks a well-defined static infrastructure of programmable stationary virtual nodes with access to real-time clocks.

Mobile nodes, while doing their own thing, also cooperate to emulate well-defined stationary virtual nodes, which can then be used to build applications.

For example:

- Virtual Nodes can be used to route messages to a designated geographical location.
  The VNs know where they are, and how to send a message to another VN in the right direction.
- Virtual Nodes can collect data from sensors in their neighborhood, and send it to a designated VN.
  The designated VN can aggregate the data, and send results back.
- A VN can take charge of activity in a region, collecting information from all the nearby mobile nodes, and even surrounding virtual nodes, and then issuing instructions.
  As a more concrete example, imagine that a VN at a traffic intersection can provide a virtual traffic light service to the vehicles coming through the intersection.

4.1 Mobile node model

- Bounded speed.
- Synchronized real-time clocks.
- GPS service that can tell them which region of the deployment space theyre in.
- Reliable local broadcast service with a known bounded message delay and minimum comm radius that at least allows mobile nodes to communicate with other mobile nodes in neighboring regions.
- Nodes may crash, restart, or be corrupted, where corruption is an arbitrary perturbation of non-program state.

Corruption is a useful concept to capture strange node failure that isnt necessarily malicious.

4.2 VSA layer

So what exactly does the VSA layer provide?
The VSA layer includes:

(a) the mobile nodes,
(b) the timed virtual stationary machines (VSAs) for predefined regions of the network, and
(c) a local broadcast communication service to allow virtual machines and mobile nodes to communicate with other nearby virtual machines or mobile nodes.

So, what is a VSA?
-Arbitrary Timed IO Automata, each with access to a real time clock, and with a restricted external interface, allowing it to broadcast and receive messages.
(Hybrid, in the sense that, in addition to discrete state transitions, state variables can also develop along continuous trajectories.)
-They can crash, restart, or be corrupted, similar to the mobile nodes.

Unfortunately, a timed trace of an emulation of an abstract machine is not going to look exactly like a timed trace of the abstract machine itself.
The emulation is performed by failure-prone real nodes that have to coordinate their emulation and can suffer from message delay in that coordination.
Hence, the trace of an emulation of the VSA can be a little behind real time.
Our VSA layer provides delay-augmented VSAs, which look just like the abstract machines except that their broadcasts might get held up for an extra bounded additive amount of time.

((Layer figure: Figure 2 from paper.))

4.3 VSA implementation

TOBcast: For robustness, we use a replicated state machine approach to emulate VSAs.
We first implement a totally-ordered local broadcast service, to guarantee that emulators in a region see the same sequences of incoming messages.

Leader election: At most one mobile node in the region is chosen as a leader to perform the VSAs outputs, preventing multiple broadcasts from the VSA, and to handle the joining of new mobile nodes in the emulation, getting them up to speed on the current VSA state.
The leader election service we use is a very simple one, where nodes rely on heartbeats to tell if a leader is alive. If heartbeats timeout then the nodes in the region execute a protocol to select a new leader.
We do it by selecting the one with lowest UID, but we dont claim this algo is the best one.
All this is done while ensuring the virtual machine can be programmed as though it has a real-time clock.

Leader gaps: Due to hand-offs and failures, the VSA states clock value (the virtual clock) falls behind real time.
In order to satisfy the layer requirement that the emulated VSA traces look like those of the abstract VSA plus up to a constant extra delay, we have to introduce some machinery to the standard replicated state machine algorithm.
If a leader fails or ceases being leader, there can be a gap in time before the next process becomes leader.
During this time, the abstract VSA maybe was supposed to have transmitted a message.
We need the new leader to be able to catch up on transmissions that were not performed because there was no leader to perform them.
So, all emulators keep track of the VSA output messages generated by their local VSA emulation, even though nonleaders arent supposed to be responsible for them.
Whenever an emulator sees that the emulation leader has transmitted one of these messages, it

knows that it no longer has to worry about that message.
Then, if a process is chosen as a new leader, it broadcasts its local backlog of VSA messages that were apparently not sent by a prior leader.
From an external point of view, the abstract VSAs outputs then only look as though they were delayed by roughly the length of the gap between leaders.
(Theres an extra comm delay term in there to deal with an edge case, but it's a relatively minor detail.)

Clock slippage: We also have to worry about slippage in abstract machine emulation during the joining of a new emulator.
When a leader processes a join, it sends out the current VSA state to the joiner.
That means the joiner, by the time he receives the message and adopts the VSA state, is up to comm delay behind the other emulators in emulation.
To resynch emulators, we actually have all emulators adopt the sent-out state, effectively rolling back the state (there is a little bookkeeping to make sure all previously received messages are still seen, messages arent resent, etc., but thats a small detail).
But now the virtual clock is behind at all the emulators by a comm delay factor.
We need the virtual clock to be able to play catch-up so that the clock doesnt keep slipping.
The solution is cute:

((Figure 7 from paper.))

When a process is emulating a VSA, it advances the virtual clock at twice the rate of real time until it is caught up to real time (where it progresses at the rate of real time), while still performing actions of the emulated VSA as triggered by the virtual clock.
Remember, the virtual clock is part of the VSA state, and as far as an emulation is concerned, the virtual clock can be the only one that exists.
If the virtual clock slips behind real-time, the emulators real clocks make up the difference by speeding up the virtual clock.
At the beginning of one of those stretches, the virtual clock can be behind by as much as max comm delay time, and the gap between real and virtual time than closes over the next comm delay time.

Self-stabilization: I didnt talk about details, but the described algorithm has been made self-stabilizing through some local checking actions and checksums.
A self-stabilizing algorithm allows a system to recover from corruption faults, or alternatively, to start in an arbitrary initial configuration.
It is an interesting question what self-stabilization means here.
The idea is that if the mobile nodes suffer corruption failures, they might not produce a consistent emulation, possibly even having more than one leader, etc.
However, they eventually stabilize so that they do produce a consistent emulation of the VSA.
Of course, that consistent emulation of the VSA may start the VSA in an arbitrary state, something that is unavoidable.
As a result, if we allow mobile node corruption failure, we also need to allow VSAs to suffer corruption failures, implying that VSA programs should be self-stabilizing.

5 Home location service and routing using VSAs

We can use the VSA layer to build applications.
Well be defining and implementing layers that will then be used to define and implement other

layers. (See figure.)

The VSA layer is used to provide a GeoCast service.
The GeoCast service and VSA layer are used to provide a location management service.
Location management, together with GeoCast and the VSA layer, is used to provide an end-to-end routing service.
Notice these are applications that are not trivial to provide at the physical layer, as we've spent a good deal of time talking about.
This is especially true if you are concerned with providing fault-tolerance.

5.1 GeoCast

GeoCast provides a service allowing communication between VSAs at two different regions.
A VSA at region u can send a message to a VSA at region v via the Geocast action.
If both VSAs remain alive for ttlVtoV time, then the VSA v is guaranteed to receive the message by the end of that time.
(We also have to make some assumptions about failure patterns of VSAs, but I'm going to sweep that under a rug.)

The implementation of the service uses a greedy DFS strategy, forwarding messages using the VSA layers local broadcast.
If a VSA wishes to send a message to another VSA, it forwards that message to a neighbor via the VSA layer broadcast, to the neighbor closest to the intended destination VSA.
If a VSA receives such a forwarded message that is not intended for it, it forwards that message to the neighboring VSA closer to the destination, etc.
This is very similar to Glider.

5.2 Home location management

We can then provide the following location management service on top of the GeoCast and VSA layers.
Lets say some client p performs a HLquery for another client qs location.
Under some movement constraints and failure conditions, the location management service guarantees the client receives a response to its query in bounded time.

To provide this service, Each mobile nodes id hashes to a set of home locations, which here are a set of regions.
The VSAs at these regions are then responsible for maintaining information about the mobile node's current region.
A mobile node periodically informs its local VSAs that it is there through local heartbeat broadcasts.
Local VSAs use the geocast service to notify the mobile nodes home location VSAs of its location.
Other nodes looking for the mobile node also use the geocast, this time to send queries to the searched-for nodes home locations, receive responses, etc.

5.3 Point to point routing

Using geographic broadcast and location management, it was then almost trivial to provide point-to-point message routing.

Again, under certain mobility and failure assumptions, if a client p sends a message to another client q, then we can guarantee that within bounded time the client q receives the message.

To implement the service, One client p looking to send a message to client q looks up qs recent region using the location management service.
It then has its local VSA send the message for q, via geoCast, to the location (and nearby locations) returned by the location management service.
Those VSAs re-bcast the message for q to receive.
This is like GLIDER.

None of the algorithms above were very original, but the point is that the algorithms are easy to actually program/get working using VSAs.