Location-Free Routing

Readings:
Rao et al. Geographical routing without location info
Fonseca et al. Beacon vector routing
Fang et al. GLIDER: Gradient Landmark-based distributed routing
Next time:
Location-based routing: Ko, Vaidya LAR paper
Ko, Vaidya multicast paper (skim only)
Kranakis, Singh, Urrutia (optional) Karp, Kung GPSR paper

1 Introduction

We continue our study of the problem of routing a message from a particular source to a named destination, in a mobile ad hoc network.

Today's material is based on three papers, which describe related routing methods.
All of these methods proceed by defining *virtual coordinates* for the nodes, which are approximations to real Euclidean coordinates but are not actually Euclidean coordinates.
These virtual coordinates are defined by means of hop-count distances from certain "landmark" (or "beacon", or "perimeter") nodes.
The nodes are not assumed to have GPS, or other sources of real coordinates—except possibly for the special perimeter nodes.
Once the coordinates are defined, they are used for routing in simple greedy-style algorithms: keep forwarding the packet in the direction of the (virtual) coordinates of the destination.

This type of greedy algorithm has been proposed based on actual geographical coordinates, assuming they are available to all nodes.
We will see examples of such algorithms next week.
Actually, we may be presenting the papers out-of-order: the geographical routing papers came earlier, and today's papers are attempts to improve on those algorithms.

The problem of determining good virtual coordinates is closely related to the localization problem we studied earlier this term.
Only, in these papers, they do not evaluate their choices of virtual coordinates as the earlier papers did—by how close they were to real Euclidean coordinates.
Rather, evaluation is done in these papers by seeing how well routing works over the virtual coordinates.

The high-level results are: Routing works just as well with these various forms of virtual coordinates as it does with real coordinates.
In fact, perhaps surprising at first: Routing with virtual coordinates works notably better than

with real coordinates in the presence of obstacles!
The reason is that the virtual coordinates are defined in terms of hop counts rather than geographical coordinates.
So they represent actual network connectivity better.
Two nodes on opposite sides of a long wall may be close in distance, but they cannot communicate.
The hop count measure would not say they were close.

The three papers have slightly different variants of these ideas.

2 The setting and some prior work

Network and application characteristics:
Large number of nodes, high density.
General communication pattern, with many communicating pairs of hosts.
Want low-latency packet delivery, even for the first packet of a message.
Would like to keep only a small amount of state info, e.g., per neighboring node, rather than per destination or per route.
Adaptability to network changes.

Distance Vector, Link State, DSDV:
Compute shortest paths between all possible pairs.
Every node stores next hop to every destination.
$O(n^2)$ message exchanges for route discovery, $O(n^2)$ routing state at each node.
Scales poorly.

On-demand routing:
Establishes routes when needed, by flooding route requests.
This works well for small/moderate-sized systems, and for some large systems with relatively stable routes, and some locality (nearby neighbors want to send to nearby neighbors, so can share routes).
But it incurs high latency on first packets, and keeps state info per-destination.

Gafni-Bertsekas link reversal: Adaptable, with just local changes.
But, need not yield shortest paths, and keeps state info per destination.

Hierarchical addressing:
Used in Internet.
Careful address allocation, allows route aggregation.
Hard to do in mobile nets: configuration not known a priori, may change.

Landmark Routing (LR) (Tsuchiya):
Allows nodes to self-configure their addresses.
Uses hierarchy of landmark nodes, which send periodic route-discovery messages.
Node's address is concatenation of closest landmark at each level in the hierarchy.
Reduces overhead of route setup.
But needs a protocol to create/maintain the landmark hierarchy.
Implementation could be complicated.

Geographic routing (Preview for next week): GPSR, etc.
Nodes identified by their geographical coordinates; requires GPS or other source of real coordinates.
Use greedy routing on geographical coordinates.
But sometimes greedy mode doesn't work—can reach a "dead end", from which it's not possible to continue in the direction of the destination; that is, every neighbor has a higher distance to the destination than the current node does.
Then we need to do something else:
Idea 1: Flood in a limited neighborhood looking for a better node.
Idea 2: Go into "perimeter mode", where the "right-hard rule" is used to forward a packet along a planar subgraph until it reaches a node closer to the destination than the starting point of perimeter mode.
In any case, need a special protocol to handle dead ends.
Also needs a protocol to route around large voids.

Advantages:
Low latency. Keeps state per-neighbor only.
Scalable, adaptable.

Problems:
Requires GPS or other localization method.
GPS is expensive, not always available, doesn't work everywhere,...
Typical localization algorithms may not be precise enough for geographic routing.
They mention the Saviddes paper in this context—perhaps other algorithms, like robust quads, would be more accurate.

Algorithm to extract planar subgraph relies on a unit-graph assumption, under which nodes here exactly the messages in a fixed radio range; not realistic.

May not be the best because it ignores real connectivity info (obstacles, etc.)
Goal: Achieve same benefits of geographic routing without location info, or with location available only at certain "perimeter" nodes.

3 Rao et al., Geographic Routing without Location Information

They consider the 2D case only, though they claim the ideas extend to 3D.
They assume special "perimeter" nodes, which may have real location information; no other nodes have any location info.
Perimeter nodes don't actually have to be on the perimeter—the algorithm would work fine with any collection of nodes, provided they are sufficiently well distributed.

3.1 Overview

Approach: Assign "virtual coordinates", then use known geographic routing methods.
The virtual coordinates need not be very accurate.
Rather, they should reflect the underlying node connectivity.
They propose a particular method for assigning virtual coordinates to nodes.

The routing (forwarding) algorithm is very simple:
Every node maintains a "routing table", in which it remembers its own virtual coordinates, plus those of its neighbors and the neighbors' neighbors (2-hop neighbors).

To route a packet, follow three Rules:

- Stop: If you are the destination, stop.
- Greedy: If you are not the destination, and some node in your routing table is closer than you are to the destination, then forward the packet to the node in the routing table closest (in virtual coordinates) to the destination.
- Dead-End: If you are not the destination, and you can't make progress with the Greedy Rule, then you have reached a "dead-end". Then perform an expanding ring search until a closer node is found (or a maximum time-to-live (TTL) for the packet has been exceeded).

Here, "closer" is measured in terms of the virtual coordinates—we will see how in a minute.

Distributed Hash Table (DHT): An application of routing.
Building a DHT over a routing service:
Each object to be stored in the network has an object identifier, which hashes to a geographical location.
It gets stored at some node near that location.
The object supports put and get operations, which work by forwarding requests to the object's location, using a routing algorithm; the location, of course, is determined by applying the hash function.

3.2 Coordinate construction

Three scenarios with decreasing degrees of location info for perimeter nodes:

1. They know their exact 2D coordinates.
2. They know that they are perimeter nodes, but don't know anything about their locations.
3. They don't even know they are perimeter nodes (and of course, don't know anything about their locations).

With less information, the algorithm becomes increasingly complex.

In the examples they will test:
3200 nodes spread uniformly throughout 200 x 200 unit square area.
64 perimeter nodes.
Nodes have radio range of 8 units, have average of 16 neighbors.
Simplifying assumptions about communication: reliable, succeed iff in nominal range.

3.2.1 Perimeter nodes know their locations

Use a simple iterative relaxation technique.
Perimeter nodes start with their actual positions, and other nodes start with default starting position (could be all the same).

Then repeatedly, presumably in synchronous steps, each node resets its x-coordinate to the average of its neighbors' x-coordinates, and the same for its y-coordinate.

That's it.
This performs well, according to the following metrics:
Packet routing success rate—the fraction of packets that reach their destinations using purely greedy routing.
Average path length—average number of hops taken along the path.
Their evaluation compared the success with what happens using real coordinates.

The algorithm seems to be robust, in that it also works well if we start with coordinates for only a subset of the perimeter nodes.
Or if locations aren't exact.

3.2.2 Perimeter nodes know who they are, but don't know their locations

Then they do some preprocessing to allow the perimeter nodes to determine candidate locations:

1. Each perimeter node floods a HELLO message through the network, allowing other perimeter nodes to determine their best distance to it, in hops. (Bellman-Ford style, or synchronous BFS).
   At this point, every perimeter node knows who all the other perimeter nodes are, and knows its best distance, in hops, to every other perimeter node.
2. Each perimeter node floods its vector of distances through the network.
   At this point, every perimeter node knows the distance, in hops, between each pair of perimeter nodes.
3. Now every perimeter node uses what they call a "triangulation algorithm" to determine coordinates of all the perimeter nodes; the algorithm is the same as one we saw in the Savvides localization paper.
   Namely, they optimize the sum-of-squares of all the pairwise distance errors, where the error between two perimeter nodes is the difference between the previously-measured distances (in hops) and the distances calculated from the proposed coordinates.
   Each node does the same calculation.

   Of course, since no real coordinates are available, the resulting coordinates can only be unique up to rotation, translation, reflection, so they must do something to "normalize".
   They could, for example, pre-select some perimeter nodes, for which everyone will determine coordinates in a consistent way, e.g., one at the origin, the second along the positive x-axis, a third in the positive y-direction.
   Actually, that not exactly what they do—they use just two perimeter nodes and the calculated "center of gravity" of all the perimeter node positions. They place the CG at the origin and use the two perimter nodes to resolve the rotation and reflection.

   At this point, every perimeter node has computed the same set of virtual coordinates for all the perimeter nodes, including itself.

Now, having computed coordinates for themselves, they can proceed as before.

They also give a physical interpretation for the Stage 3 calculation, in terms of springs between all pairs of perimeter nodes, with lengths proportional to the hop count distances.

Note: Non-perimeter nodes also acquire information during this preprocessing; in particular, they should now know their own shortest distances to all the perimeter nodes.
They could use these to come up with good proposals for their own initial coordinates, rather than all choosing the same default:
Each non-perimeter node could do this by running the same triangulation algorithm, but including itself in addition to the perimeter nodes.
This improved choice of initial coordinates causes the later relaxation to converge (to high routing success percentage) much faster: only one iteration in many cases!
They compare this with everyone starting with the same default initial coordinates—sometimes there it takes 1000 iterations—this sounds very poor.

An issues with the Stage 3 calculation:
Message loss and node failure can cause the nodes to have incomplete and inconsistent notions of the inter-perimeter hop-count distances.
They claim that their policy of pre-selecting two perimeter nodes and calculating the CG is resilient to some amount of such failures:
If someone doesn't get the info for the two selected perimeter nodes, they simply drop out from being a perimeter node.
They claim that the CG calculation is resilient to having partial info.

3.2.3 Perimeter nodes don't even know who they are

They use another preprocessing stage where a subset of the nodes elect themselves as perimeter nodes.

E.g., a heuristic:
Start with a predetermined bootstrap perimeter node, bcasts HELLO, every node learns its hop-count distance to the perimeter node.
Then each node elects itself to be a perimeter node if it is the furthest from the bootstrap perimeter node, among all its 2-hop neighbors.

Note that these nodes might not actually be on the "perimeter" of the area in 2-space, but it doesn't matter.
The use of "perimeter nodes" in this paper seems to be a misnomer—any nodes could be used (provided they aren't clustered, or in some "special position").

Projecting coordinates on a circle: ???
Support for DHT applications, which are often based on a circular configuration.
But they also mention that this helps in supporting mobility—I don't see this.

Mobility: Not clear. If all nodes, including perimeter nodes, can move around arbitrarily, nodes can join and leave, then it seems we would have to rerun the entire algorithm periodically.
On the other hand, if the perimeter nodes don't move much but the others do, then we could maintain the needed into by having just the other nodes periodically run the relaxation algorithm (each with the perimeter nodes plus itself).

3.3 Performance evaluation

Metrics: Success rate of greedy routing, and path length (in hops).

3.3.1 Experimental design

Packet-level simulator, for 10,000s of nodes.

Simplified MAC-layer model:
No message loss or signal propagation characteristics.
Radios have precise, circular radio range.

In some experiments they consider:
Losses: Nodes drop incoming packets with probability p.
Mobility: Random waypoint model.
Obstacles: Straight walls parallel to x or y axis. Nodes can't communicate if the line connected them passes through a wall.

Typically, 3200 nodes, in 200 x 200 area, radio range 8 units, so nodes have average of 16 neighbors. About 60 perimeter nodes.

Timers should really be adaptive to network parameters, but they just fix particular values.

3.3.2 Scalability

Measure success rate, path length, for greedy routing, as a function of the network size and node density.
Network size: They try 3200 and 12800 nodes.

Impact of number of iterations of relaxation algorithm:
Algorithm converges very fast to acceptable coordinates, for both sizes.

Success rates:
Are very close to those for greedy routing with true positions.
Are very consistent across the runs.
Decrease very little as network size increases.

Path length:
Of routes followed in the routing algorithm:
Virtual coordinates have virtually no impact—results almost identical to those for true coordinates.

Distribution of routing load:
Virtual coordinates do not introduce any hot spots.
Distribution very similar to that for true coordinates.

Impact of density:
At different node densities, performance with virtual coordinates still tracks that of real coordinates.

Overhead:
Initial setup phase: Communication load depends on number of perimeter nodes. Not too bad.
After bootstrapping is completed, periodic flooding of beacon info, refreshing neighborhood info, etc. Not too bad.

3.3.3 Other considerations

Obstacles:
Success rate decreases as obstacles increase.
But virtual coordinates work better than real coordinates; this is because the virtual coordinates reflect network connectivity better than real positions.

Mobility:
Random waypoint model.
Behavior in presence of mobility isn't clear to me...It sounds like the perimeter nodes don't recalculate their coordinates when they move—they just keep them fixed until they drop out from being perimeter nodes. ??? Unclear what effect this policy has...
Success rate of routing decreases with increased mobility, of course.

Losses and collisions:
Randomly drop control packets, not data packets, with probability p.
Routing success rate drops as loss rate increases, but the drop isn't severe.
This is probably because the algorithm has a lot of built-in robustness, because it does periodic refreshing and recalculation.

Irregular shapes:
Algorithm works well with irregularly-shaped network, including networks with large voids. Remarks similar to those for obstacles, above.

3.4 Conclusions

Key contribution: A relaxation algorithm that associates virtual coordinates to each node.
And a way of using these for geographic routing.

Simulations show:
Algorithm consistently matches performance of greedy routing with true positions, over a wide range of simulation scenarios.
In fact, outperforms greedy routing with true coordinates when the network connectivity is inconsistent with the network geography.

Scalable:
They use local info only for routing, but the set-up, and background, periodic maintenance, involve global interactions. Not too bad, though (they claim).
The actual message forwarding is fast, even for the first packet of a communication. Not too much state: per-perimeter node, and they don't need too many perimeter nodes. About 60/3200, or 2%.
Adaptability isn't so clear.

4 Fonseca, et al., Beacon vector routing

They propose an algorithm that is quite similar to the previous one, called Beacon Vector Routing (BVR).
Again, coordinates of nodes are defined based on the vector of hop count distances to a small set of nodes, here called "beacons" instead of perimeter nodes.
Again, they define a "distance metric" on the resulting coordinates and use it for greedy routing; but the metric and its use are a bit different.
They carry out simulations and an actual implementation.

4.1 Introduction

This is written from a practical sensor net implementation viewpoint.
They begin with some interesting comments on sensor net communication:
So far, because all sensor nets have been doing is data-collection and aggregation, they have used limited forms of communication: one-to-many and many-to-one, along the edges of a tree that is built ahead of time.
For more interesting applications (e.g., more complicated data management, tracking, coordinated control), they will also need one-to-one routing.
But so far, there has been no practical current implementation of point-to-point routing in sensor nets—many proposed designs, but no usable implementations.
This lack is hindering the development of new sensor net applications.

What they want:
Strong scaling, robustness guarantees.
Algorithms should work with the limited resources of sensor nodes: low energy usage, small memory, small packet length, no GPS.
Algorithms should work over networks with weak communication guarantees, e.g., about radio quality.
Simple algorithms

Simplicity:
They insist that algorithms must be very simple. Why?
They complain that even simple algorithms are hard to implement in real sensor nets, e.g., trivial flooding and tree construction algorithms took years to get right.
That sounds like the algorithms they implemented weren't sufficiently robust, so had to be modified by the implementers to add extra resiliency, etc. (but then they should have started with different, more robust algorithms).
Maybe they are afraid that more complicated algorithms will be even harder for implementers to understand and modify.

They are happy to reuse algorithms they understand, like the ones that build trees routed at designated sources.
They will use this to build trees from the beacon nodes.

Scaling:
They complain about the scaling properties of previous algorithms:
Distance Vector, Link State, DSDV

On-demand route discovery: DSR, AODV

Hierarchical addressing: Hard to do in sensor nets: configuration not known a priori, may change. Implementation could be complicated.

Geographic coordinates:
Same advantages and problems as we discussed earlier.
Scalable, but requires GPS or other localization method.
And may not be the best in the presence of obstacles.

So they decide on Virtual Coordinates.
They refer to the Rao paper, but say the setup parts of this are expensive:
Flooding of distance info costs a lot of communication, since they use $O(\sqrt{n})$ perimeter nodes.
Also uses a lot of storage, since everyone has to record too many distances.
They also seem to be complaining that it's complicated (?).
Has not been implemented on real hardware (and by implication, the complexity of the algorithm would make this hard).

Which leads them to Beacon Vector Routing:
Small routing state (constant); the main reason seems to be simply that they use fewer beacons!
BVR's beacons are chosen randomly; needn't be on a perimeter or have any other special properties.

They use the beacons to construct node coordinates; no real geographical info required.
Different kinds of coordinates from Rao.
Easier to construct, no relaxation algorithm required.
In constructing these, they use trees as in prior sensor network (Madden?)—in fact, they regard this as the "core mechanism" of BVR.

Once they have these coordinates, they use greedy forwarding as in Rao, but based on the new kinds of coordinates.

4.2 The BVR Algorithm

BVR does greedy forwarding, using a new metric.
Assume a (very) small set of randomly-chosen beacon nodes.
Each of these constructs a tree spanning the network, allowing every node to learn its shortest distance, in hops, to each beacon.
So far, this is similar to Rao.

But now, they do something different:
They don't bother to determine virtual 2D coordinates for all the nodes; instead, they let the nodes work directly using their vectors of beacon distances!

But how should they do this?
Specifically, fix the destination d.
Suppose we have two nodes p and q, each with a vector of beacon distances, and we want to know which one is "closer to" d.
Suppose we know the vector of beacon distances for d.

How do we compare the two vectors, for p and q, and say which one is better?

They define a distance metric $\delta(p, d)$, for measuring how close a node p is to destination d.
Their metric looks rather ad hoc, though.

Definition of $\delta(p, d)$, using parameter k (a number of beacons, not necesarily all the beacons):
First, identify the set C_k consisting of the k beacons that are closest to d; this is deducible from d's distance vector (with some tiebreaking mechanisms).
Among these, consider only the subset I of beacons i such that $p_i \geq d_i$, that is, the beacons for which the destination is closer than the current node p is.
The intuition is that they want to use only beacons near d for routing—those that are far away should be less helpful.

For each $i \in I$, calculate how much closer d is than p: $p_i - d_i$.
Add these terms up, for all beacons in I.

This gives some kind of metric: See which beacons d is closer to than you are. And add up the numbers saying how much closer d is. It is some kind of measure of how far away you are from d. Sort of.
That's the main metric they use. In routing, they will try to find a neighbor with a smaller value of this metric, and think that will mean the neighbor is actually closer. Anyway, the metric is a decreasing integer if we route in this way, so it seems this policy should eventually get us close to d.

They also use a second metric as a tiebreaker:
This one involves the other beacons, in $C_k - I$, that is, the ones for which the current node p is closer than the destination is. This metric (which also adds up the differences to the individual beacons) measures how much closer p is than d.
Again, it seems good to move so as to reduce this metric.
However, they have the intuition that this metric is less reliable, which is why they use it only as a tiebreaker.
The reason is that it involves situations where the packet is trying to get closer to d by moving away from a beacon that is far from d.
But, it could be moving in the wrong direction!

Packet forwarding then proceeds as follows:
The packet carries with it the history of the best distance vector that has been encountered so far in the search, according to the given metric (normally, should be the the current location's vector).
Carry out a greedy search for a neighbor that improves on this metric value.
When greedy forwarding fails, use fall-back mode:
Forwards the packet towards the beacon closest to d, but this time using a different mechanism: send it to your parent on that beacon's rooted spanning tree (!)
When the parent receives it, it tries going back to normal greedy mode.
If the packet gets all the way up a beacon's tree, without being able to reach d, the beacon i at the root initiates a controlled flood.
This makes sense because b should be close to d.
Also, b knows how many hops are needed in the flood, since it knows d's distance vector, which says how many hops away d is from b.

Q: I don't understand the performance of the combination of the normal and fall-back mode. E.g., maybe it's better to just use the fall-back mode, simply routing on the tree to a beacon that is near d. ???

Beacon maintenance:
The algorithm is robust—can work when some beacons have failed, and when nodes' distance vectors record distances to different beacons.
To detect beacon failures: Use some kind of "I'm alive" announcements.
When there seem to be too few beacons (not enough), other nodes may nominate themselves as beacons, establish trees, etc.

Location directory:
A side issue: If you know a node's id and not its location, how can you find its location? Use consistent hashing.

4.3 Simulation results

High-level simulator that abstracts away technicalities of the wireless network.
Assumes fixed circular radio range, no packet losses.
Ignores capacity, congestion issues.

Nodes placed uniformly at random in square planar area, as in the Rao paper.
Vary total number r of beacons, and the number k of beacons used for routing to a particular destination.

Experiments compare routing using BVR to greedy geographic routing over true positions.

4.3.1 Metrics

- Greedy success rate: Fraction of packets delivered to destination without requiring flooding.
- Flood scope: The number of hops it takes to reach the destination in those cases where flooding is invoked.
- Path stretch: The ratio of the path length of BVR to the path length of greedy routing using true positions.

They want to understand the costs of achieving success according to these measures:

- Control overhead: Number of flooding messages generated to compute and maintain node coordinates; depends on r.
- Per-packet header overhead: Depends on k, the number of routing beacons.
- Routing state: Number of neighbors about which a node maintains information in its routing table.

4.3.2 Routing performance vs. Overhead

Routing success rate for different numbers r of beacons, and different numbers k of routing beacons.
Success rate increases with each (of course).
But, performance after $k = 10$ doesn't increase much.
They conclude that 10 routing beacons is enough.
And, using $k = 10$, $r \in [20, 30]$ is enough to match performance of true positions.
This is $< 1\%$ of the total number of nodes—very reasonable flooding overhead.

Path stretch is only 1.05, in pretty much all the cases they tried. Excellent.
That's counting only the non-flooding parts of paths; but even counting the flooding portions, it's still 1.1.

Distribution of routing load: Virtually identical to that for true positions—just a slightly higher load on the nodes in the immediate vicinity of beacons.

Summary: BVR can roughly match the performance of greedy geographic routing over true positions, using only a small numer of beacons, and considering only its one-hop neighbors for greedy forwarding.

Obstacles: As the number of obstacles and/or their length increases, the decrease in success rate using BVR isn't that big.
They say it's not significant—dropping in the worse case from 9691By comparison, routing with true coordinates drops from 98

4.3.3 Impact of node density

Has a high impact on the routing success rate.
At low density, BVR performs much better than greedy geographic routing; however, neither is really good (BVR is only 80
This was based on maintaining info about only 1-hop neighbors.
Adding information about 2-hop neighbors would made a very big difference in success rate for BVR (and presumably for geographic routing also).
Of course, this would increase the overhead (and complexity) of the neighborhood maintenance protocols.

Cute idea: Establish and maintain larger neighborhoods on-demand—triggered by a node being unable to forward a message greedily.
This can be a big win, because in a large graph, we expect that the number of nodes that are "local minima" will be only a small percentage of the total.
They did some experiments to evaluate this idea; it greatly improves the success rate (for both BVR and true positions).
And (for BVR) not too many of the nodes fetch 2-hop nbhds (up to 15% for the low densities they tried).

4.3.4 Scaling the network size

Set target success rate at 95%.
Fix $k = 10$.
Try to find what value of r can guarantee 95% success, as the number of total nodes grows.

They found that $r = 10$ is enough, using BVR with 1-hop nbhds plus on-demand 2-hop nbhds, even for very large numbers of nodes.
But with just BVR with 1-hop nbhds, the number r grows steadily (but it's not too huge—under 2
They conclude that the total number of beacons can remain small as the network grows.
Actually, this isn't so surprising—the beacons are being used as the localization algorithms, to determine positions for other nodes.
In the plane, some fixed number should suffice, independent of the number of nodes.

4.4 BVR Implementation

4.4.1 The implementation

They also do a real implementation, using TinyOS, mica2dot motes.
Always use $k = r$.
Don't implement some of the optimizations, including on-demand nbhd acquisition.
42 and 74 mote testbeds.
5-7 hop diameters

Several issues arise, each of which could be the subject of more research. They just do something reasonable in each case.

- Link estimation: How can they tell which nodes are connected? Should they use the same kind of graph model as in the simulations? Now they find it more convenient to characterize links by a probability of successful communication.
  Specifically, they use a "passive link estimator":
  Tag packets with sequence numbers, receiver calculates percentage received.
  Receiver transmits this info periodically, so the sender also knows the estimate.
- Neighbor selection: A technical issue. They don't have enough space to maintain state for all neighbors, because these are little motes. Nodes throw out info about neighbors to which they are connected with lower-quality links.
- Distance estimation: Need to pin down a particular protocol.
  Nodes build a tree to every beacon, with sequence numbers (like using time tags?) used to maintain the tree in the presence of changes.
  They have to make sure the tree contains good paths, so as not to give an impression of a low hop count when in fact the path isn't very usable.
  They do some estimation and optimization to accomplish this.
- Route selection: How does a node forward packets? What happens with lossy links?
  When selecting the next hop, BVR takes into account not just the progress using the distance function, but also the quality of the links.
  They use the product PRR x distance—the product of the link quality and the distance metric—to select the best link.

They play various tricks to try to get reliable delivery—link-level acks and some other kinds of retries (trying different neighbors).

With all these changes, the implementation is fairly complicated.
It seems they essentially have modified the algorithm in sigificant ways, so it's not really the same algorithm they described and simulated earlier.
The new algorithm should be analyzed/simulated on its own.

4.5 Evaluation

They show that:

- Nodes select high quality neighbors (neighbors to which it has high-quality links).
  But they aren't necessarily nearby—some are halfway across the network.
  This underscores that the usual circular radio assumption isn't good.
- Routing success is high (over 97gracefully under high loads).
  They believe that BVR can sustain a significant workload of routing messages, in a real implementation.
- Dynamics: BVR sustains high performance even under high node failure rate, both beacon and non-beacon.

  For non-beacon nodes: Extremely resilient to random node failure, even when up to 80% of the motes fail!
  This is because of redundancy in the coordinate system, the route selection mechanisms, and the adaptability of the coordinates to topology changes.

  For beacon nodes:
  Routing performance doesn't degrade with occasional beacon failures.
  Tolerates even high failure rates reasonably well: BVR routes well with even a partial beacon set.
  Also, beacons get replenished rapidly, the time depending on network diameter and frequency of neighbor exchanges.
  Extra candidate beacons are efficiently suppressed.
  No significant communication overhead.
- Coordinate stability: A node's coordinates don't change often, and when they do, the changes are small.

4.6 Conclusions

Advantages: Simplicity, scalability, resilience
Builds no large-scale structures.
Periodic flooding means that the network recovers fast from any failures and changes (seems self-stabilizing?)
Good performance in a wide range of settings, often much better than geographic routing.

5 Fang et al., GLIDER

GLIDER: Gradient Landmark-Based Distributed Routing for Sensor Networks
An interesting paper, along the same lines as the first two, but with an interesting new twist and some cool math.
It presents a new naming/addressing scheme and a new routing algorithm based on this.

They assume fixed nodes, in a configuration that isn't known a priori—their work remains to be extended to dynamic settings.

Their protocol starts with a preprocessing phase where it discovers global topology information about the sensor field (e.g., holes).
It also divides the nodes into overlapping clusters they call "tiles", where each individual tile is supposed to have no interesting topological features.
They assign an address to each node, containing the name of a tile containing it and some local coordinates.

For routing, they use the global topology knowledge to plan an abstract route consisting of tiles, to reach a tile containing the destination node.
Then they have to realize the route using real nodes.
Within each tile, routing should be easy, using greedy methods.
Similarly, jumping between adjacent tiles is easy.

A nice level of abstraction here—assumes that the topology is longer-lived than the low-level nodes, and does high-level route planning based on this info. Then, on a shorter time-scale, the lower-level nodes "implement" the path of tiles.

5.1 The idea: Topology-enabled routing

Depends on connectivity only, no actual locations.
Two phases:

1. Global preprocessing: Proactive
   Discover global topology of the sensor field, specifically, divide the nodes into "tiles", with trivial topology.
   Topological features reflect underlying structure of the environment, e.g., obstacles.
   Topology likely to remain stable.

   Define local coordinates for nodes within each tile, depending only on link connectivity of the nodes.
2. Local routing: Reactive
   Greedy forwarding based on local coordinates within each tile.
   Local decisions, made on short-term, node-by-node basis.
   Greedy forwarding works better within tiles than in general networks, because the tile has no topological features.

Features of the algorithm:
Extends to 3D.

Don't need to construct a planar graph to bypass local minima.
Don't need to explicitly discover holes.

Both phases use a distinguished subset of the nodes, here called "landmarks".
Use Voronoi/Delaunay methods to construct a complex whose vertices are the landmarks and whose topology captures the topology of the underlying sensor field.
Generate local coordinates based on link distances to nearby landmarks.
Also, use the landmarks in routing.

Difference from Rao and Fonseca:
Here, don't use all the landmarks to provide coordinates for all the nodes—coordinates are based just on nearby landmarks. So this scheme scales better.
They use a different method for generating the local coordinates (?).

Topology: Here refers not just to connectivity, but to all topological features, in the sense of algebraic topology.
2D: Holes; 3D: Tunnels, voids

Traditional algebraic topology is based on continuous spaces, not discrete collections of points or nodes.
So when they talk about topology here, they are thinking of the underlying set from which the points are sampled.
This thinking is reflected in their presentation—first for continuous spaces, then for discrete spaces, presented as an approximation to their treatment for continuous spaces.

5.2 Overview of GLIDER

$G = (V, E)$, communication graph
Distance = shortest hop count

Atlas M: A structure that describes the general topology of the sensor field, with nodes divided into routable tiles, and with an adjacency relation between the tiles.
This will become globally known in phase 1.

Constructing Atlas M: First construct a Voronoi complex, then take its combinatorial Delaunay triangulation. The atlas M is defined to be this CDT.
Landmark Voronoi Complex (LVC)
Select a small number of "landmark nodes".
The tiles are the Voronoi cells of the landmarks, based on nearest landmark in the hop-count metric.
Since there can be ties, a node is thus allowed to belong to more than one tile.
The Combinatorial Delaunay triangulation (CDT) is just the adjacency graph among the tiles.
Every node learns the CDT.

They also define a "name" for each node v, in their first phase. Name has 2 parts:
Global tile name, the uid of the closest landmark (now breaking ties, according to some default).
Local landmark coordinates, defined in terms of distances from the node to its own and nearby landmarks; the distances here are defined to be "centered squared-distance coordinates".

Packet forwarding:
A node u wants to forward a packet towards a destination d, which it knows by its name, as defined above.
Thus, it needs a rule to specify which of its neighbors it should forward to, based on its own local info, some info about its neighbors, and the name d. The rule is:
Calculate from the CDT a high-level sequence of tiles for the routing path.
Use greedy gradient descent towards the landmark node of the next tile in the path, or towards d if this is the last tile.

Success depends on a good choice of landmarks.
Should be small number, and should represent interesting topological features.

5.3 Landmark Voronoi complex

$G = (V, E)$
$\tau(u, v)$ = shortest-hop distance between nodes u and v in G.
$L \subseteq V$, landmarks

LVC and CDT are natural extensions of the corresponding concepts from ordinary geometry, to the case of a graph with shortest-path metric.

LVC definition:
For $v \in L$, define Voronoi cell $T(v)$ to be the set of nodes whose nearest landmark is v.
Thus, a node may be in more than one Voronoi cell.

Lemma 1: For any node $u \in T(v)$, every shortest path from u to v in G is completely contained in $T(v)$.

Thus, the spanning graph on each V-cell is connected.
So the V-cells provide a division (not a partition as they say) into connected "tiles".
It is desirable that each V-cell have trivial topology in all dimensions, not just connectivity (e.g., no holes), but this depends on a good choice of landmarks.

CDT definition:
They start talking about this is some topological generality, but then rapidly restrict their attention to a special case: $D(L)$, a simple connectivity graph for the landmarks.
Two landmarks are defined to be connected iff either their V-cells share a common node, or there are two "witness" nodes in the two V-cells that are directly connected in G.

The topological generalization involves a clique of nodes in G, of any size, that intersect all the cells; sounds interesting, but they don't use this later.

Theorem 2: If G is connected, then $D(L)$ is also connected.

In fact, every path in G can be "lifted" to a path in $D(L)$.
And conversely, every path in $D(L)$ can be "realized" as a path in G.
This correspondence suggest that $D(L)$ is a good simplification of G for use in routing.

$D(L)$ is the CDT that will serve as the atlas for global route-planning.

Landmark selection:
CDT should be small, so we should have as few landmarks as we can.
However, we need enough landmarks to ensure that each V-cell has a simple (hole-free) topology.
Thus, we want landmarks close to topological features, such as hole boundaries.
Can select the landmarks manually.
Can automatically discover hole boundaries (reference [6], by Fekete et al., sounds interesting), or at least some nodes on the boundary.

The number of landmarks should be proportional to the number of holes (or other topological features) in the sensor domain.
They think that typically this number should be small.

5.4 Local landmark coordinates

Define local virtual coordinates, for use in greedy routing.
Motivate the definition by describing a continuous version, which they show has no local minima—it always converges to the target.
Then they adapt this to the needed discrete version.

5.4.1 Continuous version

Fix a set $\{u_1, u_2, ...u_k\}$ of landmarks.
First, for each point p, define a k-vector of coordinates $C(p)$, one coordinate for each of the k landmarks.
This is supposed to describe the distances between p and all the specified landmarks.

Namely, start by defining $B(p)$ to be the k-vector of squared distances to the k landmarks: $B(p)_i = |p - u_i|^2$.
Now define $\bar{B}(p)$ to be the average such squared distance, taken over all the k landmarks.
Then define the actual coordinates to be the k-vector $C(p)$, where $C(p)_i = B(p)_i - \bar{B}(p)_i$, that is, the vector of "centered" squared distances—centered by comparing them to the mean squared distance for all the landmarks.

Next, they define the distance $\delta(p, q)$ between two points p and q in terms of these k-vectors: $\delta(p, q) = |C(p) - C(q)|^2$.
That is, they take the Euclidean distance between the two vectors, regarding each vector as a point in k-space, and then square it.
Q: Why another squaring step here?

They then consider doing gradient descent based on distance to the target node v, as measured by distance measure δ.

Lemma 3: In the continuous Euclidean plane, if $k \geq 3$, then gradient descent using δ distance to the target always converges to the target.

Proof: By the form of the functions involved.
They show that the function mapping p to $C(p)$ is an "affine linear transformation", and (therefore) is one-to-one.
So the gradient of the distance function is nowhere 0 except at the destination itself.

In k-dimensional Euclidean space, they would need $k+1$ landmarks instead of just 3, and they should not be contained in any $(k-1)$-dimensional affine subspace.
This sounds like the same considerations that arose in the localization papers.
It is related: What we are trying to guarantee is that the C function assigns unique coordinates to all points in the Euclidean plane.

5.4.2 Discrete version

They work entirely by analogy with the continuous case, using hop counts instead of Euclidean distances: $\tau(p, u_i)$, which is the minimum number of hops in G from node p to landmark node u_i.
Define $\bar{\tau}(p)$ to be the average of the squares of these hop-count distances, taken over the k landmarks.
Then define $C(p)$ by $C(p)_i = \tau(p, u_i)^2 - \bar{\tau}(p)$.
This is analogous to the continuous case (think of $B(p)_i = \tau(p, u_i)^2$)).
Then $\delta(p, q)$ is defined just as in the continuous case.
Now they carry out routing as before, using this new δ function. Local, efficient.

Local minima may occur in the discrete version!
Most likely where the continuous gradient is "shallow".
However, when the nodes are sufficiently dense, the shortest-distance metric approximates the Euclidean metric closely enough to "reduce the chance" of local minima.

Optimization:
Can define local landmark coordinates in terms of nearby landmarks only, rather than all the landmarks in the system.
Namely, each node u in a V-cell $T(v)$ can use just v and neighbors of v in $D(L)$.
(If u is in more than one V-cell, then resolve ambiguity by the default rule mentioned earlier.)

Different sets of landmarks:
When you compare distances between two nodes, you must use the same set of landmarks for both.
What if they actually have coordinates for different landmarks?
Consider only those in the intersection of the two sets?
This issue arises during gradient descent—when a node compares the distances of many neighbors, then it seems it should use the same landmarks for all of these.
But maybe this isn't much of a problem because gradient descent is generally being used WITHIN a tile only, where the landmark sets should be (mostly) the same. (?)

A technicality:
In calculating distances, they don't use distances in G, but rather in more restricted "Voronoi neighborhoods"—call these "neighborhood distances". Presumably, these can be different sometimes, but it's not clear why, and not clear if it's interesting.

5.5 Naming and routing

Nodes have unique ids.
Names aren't the same as ids; they are assigned after preprocessing, used for routing. They need not be unique.

Each node has a "resident tile" (ambiguity resolved by default rule), and the landmark for that tile is its "home landmark".

Name consists of:
id of its home landmark, and
list of neighborhood distances to its reference landmarks (those in its resident tile and neighboring tiles).

These distances are just (absolute) distances—not squared, not centered; they be squared and centered when needed.

Non-uniqueness of names:
Shouldn't occur often.
And when it does, they are likely to be nearby, so that local flooding can resolve the situation.
In some applications (e.g., sensor data collection), might not even have to resolve—might be good enough to reach ANY of the nodes with the given name.

Routing, from node u to node v:
Two parts, global and local.

Global: Identify the shortest path from $h(u)$ to $h(v)$, the two home landmarks, in the CDT.
This is a sequence of tiles, with known home landmarks.

Local:
Intra-tile: Use gradient descent based on the local landmark coordinates. If stuck, do local flooding, within the tile.
Inter-tile: Aim for the landmark node in the next tile in the global route.

Note the way this behaves: The packet heads toward the landmark for a tile. But as soon as it reaches the tile, it stops trying to reach the landmark and instead heads for the landmark of the *next* time in the tile route.

5.6 Detailed distributed algorithms

Naming protocol
After the landmarks are selected, this constructs the LVC and CDT, and assigns to each node its local coordinates.
Straightforward protocol, involving some flooding.
Landmarks initiate flooding to determine graph distances from each node to its nearby landmarks, and to compute the Voronoi cells.
(If start times for the different landmarks are synchronized, they may be able to prune out many messages.)

After this flood, everyone has learned which Voronoi tiles it belongs to.

Then someone (centralized) collects all the CDT information and distributes it to everyone.

Then each landmark computes the CDT shortest-path tree rooted at that landmark and bcasts it to all the nodes in its own V-cell.

Then each node computes the neighborhood distances between itself and its reference landmarks—can do this with a single flood from each landmark, just within its neighborhood.

Routing protocol: As described earlier.

5.7 Discussion

They did some naive simulations, assuming no packet loss, delay, no interesting timing issues.
Unit disk graph: Two nodes communicate directly iff their Euclidean distance is ≤ 1.
2000 nodes on a "perturbed grid".

They claim high success rate for delivery.
Packets can always make progress across intermediate tiles.
May get stuck at small holes, with frequency that depends on node density; need at least (approx) 5 neighbors.

Path length:
Without obstacles, GLIDER generates routes comparable to those generated by geographical routing algorithms.
With obstacles, GLIDER performs better:
Avoids routing directly around holes, which leads to longer routes. (Packets don't actually reach the landmark nodes in each tile along the way—they just use them as intermediate goals to help them reach the tile. Once they've reached the tile, they work toward the next tile. That tends to shorten the path.)

Routing directly around holes would also congest and deplete the nodes on the boundary.

Scheme works in 3D also.

Future work:
They wish they could utilize higher-dimensional topology info for something, not just connectivity.
Criteria and algorithms for landmark selection.
Multi-level hierarchies instead of just two-level.
Handling network dynamics (node addition and failure).