Location Services

Readings:
Awerbuch, Peleg paper
Li, Jannotti, de Couto, Karger, Morris GRID paper
Abraham, Dolev, Malkhi LLS paper
Demirbas, Arora, Nolte, Lynch, Stalk paper
Next:
Mittal, Demirbas, Arora LOCI paper
Malpani, Welch, Vaidya Leader election paper
Walter, Welch, Vaidya
Walter, Cao, Mohanty k-exclusion paper

1 Introduction

Today we will study "location services", which maintain location about mobile users.
They generally support two operations:

- $MOVE(\xi, s, t)$: A user ξ may move from node s to node t.
- $FIND(\xi, v)$: Anyone at any node v in the network can try to locate, and get a message to, any user ξ anywhere in the network.

$FIND$ is typically used to obtain an address, which can be used repeatedly for communicating with the target node.
We will cover four papers...

2 Awerbuch, Peleg

2.1 Overview

This highly-cited paper develops a location service for mobile users traveling around a fixed network.
Operations supported:
$MOVE(\xi, s, t)$: A user ξ may jump from node s to node t (not necessarily along graph edges).
$FIND(\xi, v)$: Anyone at any node v in the network can try to get a message to any user ξ.

The paper was written for traditional wired networks.
For wireless networks, the results apply in static settings; it's interesting to think about how they could be extended to more dynamic cases.

Goals:
Make both kinds of operations fast and communication-efficient (polylog in the number n of nodes in the network), and store only polylog info per user at each node.

The main idea: Implement the service over a hierarchical directory service.

A single-level directory provides operations:

- $INSERT(\xi, s)$: Insert ξ associated with location s.
- $DELETE(\xi, t)$: Delete ξ from location t.
- $FIND(\xi, v)$: Starting from v, try to find an associated location; may return a location or may fail.

A hierarchical directory service consists of a series of directory services, each associated with successively larger "radii" (use successive doubling).
A directory service associated with a certain radius is guaranteed to return an answer for a $FIND$ for any node that is within the given radius of the location v at which the $FIND$ is invoked.

The implementation of the location service over the hierarchical directory service isn't obvious:
It doesn't try to keep all the directories up-to-date with the latest locations of all the users.
Rather, it keeps "low-level" directories—those with small radii—more up-to-date, while allowing higher-level directories to lag further behind.

However, if a directory entry is out of date, it points to some node where the user ξ previously visited, and that node has a forwarding pointer to a node that ξ visited more recently.
So, by following forwarding pointers, the $FIND$ query can eventually reach the user.

More specifically, if the level i directory entry for user ξ indicates node v, then node v has a forwarding pointer to the same node w that the level $i-1$ directory contains for ξ.
That is supposed to be a lot closer.
The length of the chain of forwarding pointers is kept low (logarithmic bound).

They also describe how to implement the hierarchical directory service.
This is a variant on standard read/write quorum system algorithms, fitted to the type of network environment they assume.

Everything is presented as if all the $MOVE$ and $FIND$ operations are done one-at-a-time, sequentially; in a final section, they describe some ideas about how to extend it to allow more concurrency.

2.2 The network model

Undirected graph $G = (V, E)$, n nodes.
Bidirectional communication channels.
Nodes may communicate directly only with neighbors in the graph.
But they sometimes want to communicate with other nodes: for this, they assume "efficient" routing is available (shortest path?)

They also allow edges to carry weights, and use these as part of their path cost measure (sum the weights).
However, for this presentation, I'll just assume that each edge has weight 1—so we are just considering hop counts.

$dist(u, v)$ = length of shortest path (I'll say in hops)
D, diameter of network

2.3 The problem

Each user ξ, at each time, is located at some node $Addr(\xi)$ of G.

Directory is a global service providing operations: $MOVE(\xi, s, t)$:
Invoked at node $s = Addr(\xi)$.
User ξ moves from node s to another node t.

$FIND(\xi, v)$:
Invoked at node v.
Delivers a message submitted (by some external client) at node v, to $Addr(\xi)$. Guaranteed.

Some extreme solutions:
Full-information strategy:
Every node in the network keeps a complete table of where all the users are.
Every time a user moves, immediately updates all the nodes.
Then $FIND$ operations are cheap—route the message directly to the intended user.
But $MOVE$ opearions are very expensive (e.g., in amount of communication).
This is useful only in settings where few $MOVE$s occur.

No-information strategy:
No information maintained anywhere.
$MOVE$s are now very cheap.
But $FIND$s are very expensive: involve a global search
Useful only in settings where few $FIND$s occur (occasional location queries).

Complete chain of forwarding pointers:
When user moves, leave a pointer from the previous node to the new one.
$MOVE$s cheap, but $FIND$s can involve following a length-n chain of pointers.

They want some intermediate "partial information" strategy, in which both operations are cheap, for any series of $MOVE$ and $FIND$ operations.

How to measure the costs of the operations?
They would like a measure in which moves to nearby locations, or searches for nearby users, cost less than long-distance moves and searches.
Essentially, would like to compare the actual costs of the operations to an "ideal cost" that would be incurred if full information were available for free.

They measure communication cost, for each $FIND$ and $MOVE$ operation, in terms of number of

hops that messages traverse in the implementation.
(Actually, they use weighted costs. And they charge based on the number of bits in the message. But let's avoid worrying about these issues right now.)

They compare these costs to the "optimal costs", which are defined by: $FIND(\xi, v)$: $dist(v, Addr(\xi))$, the distance, in terms of smallest number of hops, from the location at which the operation is invoked to the actual location of ξ.
$MOVE(\xi, s, t)$: $dist(s, t)$

In their analyses, they don't give worst-case bounds for all the individual operations, but rather, amortized bounds over the entire sequence.
Thus, for an entire sequence of $FIND$ and $MOVE$ operations, they pick out the $FIND$ subsequence and the $MOVE$ subsequence.
Then, e.g., for the $FIND$ subsequence, they compare the sum of the actual costs of all the $FIND$ operations to the sum of the optimal costs for those same $FIND$ operations.
They call this ratio the "FIND-stretch".
Similarly for the MOVE subsequence.

Their main result is an algorithm that guarantees:
Find-stretch $O(\log^2(n))$ (but that is counting $\log(n)$ for each message, so probably this should be just $O(\log(n))$.
Move-stretch $O((\log D)(\log n) + (\log D)^2/(\log n))$.
And low communication.

2.4 Hierarchical directory service

Hierarchy of "regional directories", $RD_i, i = 1, ..., \log D$.

RD_i enables a searcher to find any user residing within distance 2^i of the searching location.
Thus, higher levels represent coarser decompositions of the network.

Each RD_i is a single-level directory.
It maintains (conceptually) some global state, which is a set of pairs (ξ, s) consisting of users ξ and addresses s.
Call s the level i "regional address" of ξ.

RD_i provides operations:
$INSERT(\xi, s)$: Invoked from location s. Associate ξ with location s.
$DELETE(\xi, t)$: Invoked from location t, where ξ is associated with t. Undo the association of ξ with location t.
$FIND(\xi, v)$: Starting from v, try to find the location associated with ξ. May return a location or may fail. Guaranteed to succeed if ξ has an associated location s with $dist(v, s) \leq 2^i$.

2.5 Implementing the location service over hierarchical directory service

This is the interesting part of the paper.
Suppose we have a hierarchical directory service—we'll see later how they implement it.

Now, how to use it to implement a location service?

Each RD_i is used to maintain associations between users and locations (regional addresses).
Ideally, these would always be maintained as up-to-date locations, so when a $MOVE(\xi, s, t)$ occurs, all the directories should be updated to remove the association of ξ with s and add the association of ξ with t.
But that would be costly.

Instead, they use another mechanism, involving "forwarding pointers" stored at some nodes.
The forwarding pointers are maintained completely outside the directory service—they are just pointers of the form $forward(\xi)$, stored at some of the nodes.
The node at which a user currently resides also knows this fact—we could encode this be setting the forwarding pointer to the same node.

How this combination of mechanisms is used by a $FIND$: $FIND$ queries $RD_1, RD_2, ...$ about ξ, until it gets some address as an answer from some regional directory.
That will eventually happen, because it's guaranteed at the top level.
Then it goes to the node indicated by RD_i for ξ, and sees if the user is at that node—if not, it follows forwarding pointers from node to node until it reaches the user's current node.

The forwarding pointers and RD_i directores are not changed during the execution of $FIND$ operations; they are maintained during MOVEs.
So consider a MOVE(xi,s,t):

First, this modifies some of the (lower level) RD_i entries for ξ, always including RD_1, but sometimes some higher-level ones also, so that they contain t, the actual new location of ξ.

To calculate when to update higher-level directories, they maintain a cumulative count of how far the user has moved (in all its moves) since updating each directory.
Specifically, when the user has moved a distance 2^{i-1} or greater, regional address $RD_i(\xi)$ is guaranteed to be updated, to point to the actual new location t of ξ.
Any time any directory is updated for ξ, all the lower-level ones are also updated at the same time, to point to t.
Let i denote the highest-level directory that is updated.

Then, if i is not the top-level directory, set a forwarding pointer $forward(\xi) = t$ at the node pointed to in $RD_{i+1}(\xi)$.

They give a "Reachability Invariant", which describes an interesting relationship between the regional addresses for ξ stored in the regional directories, and the forwarding pointers:
If $i \geq 2$, $RD_i(\xi) = s$ and $RD_{i-1}(\xi) = t$, then $forward(\xi)$ at s contains t.

Thus, for a node s that appears as a regional address of ξ at any level of the directory hierarchy, following the forwarding pointer from s gives the same result as would be obtained by looking for ξ one level lower in the hierarchy.
So, following a path of forwarding pointers to ξ starting from any regional address for ξ in the directory hierarchy, is the same as following a path through all the successively lower-level regional addresses for ξ.

To capture the property that the directory entries are updated sufficiently often, they state a "Proximity Invariant":
For any i, the total distance traveled by user ξ since the last time $RD_i(\xi)$ was updated (to its exact location) is $\leq 2^{i-1} - 1$.

That's pretty much it; the use of powers of 2 in this way gives them the logarithmic bounds they claim; they don't actually do the analysis in the paper. (It probably appears in the journal version.)

2.6 Implementing the hierarchical directory service over the basic network model

For each level, they use a kind of quorum configuration.

The nicest way to see this is by defining a notion of "m-regional-matching", or "m-quorum-configuration", where m is a distance (nonnegative real).
This assigns to each vertex v, two sets of vertices, $Read(v)$ and $Write(v)$, such that, if two vertices v and w are within distance m of each other, then $Read(v) \cap Write(w) \neq \emptyset$.

Thus, if two operations are performed, one involving a $Read(v)$, and the other involving $Write(w)$, for nearby nodes v and w, the two operations would have to encounter a vertex in common.
So, for example, if a write-quorum is used to write an update to several copies of a directory, then accessing a read-quorum from a nearby node is guaranteed to get an up-to-date copy of othe directory.

They cite a result, Theorem 4.1, without proof, saying that certain efficient m-regional matchings exist.
"Efficient" here refers to:
Small size read-quorums and write-quorums, and
Short distance between each v and each member of $Read(v)$ and $Write(v)$; since the regional matching is parameterized by a distance m, this measure is normalized by dividing by m; thus, larger m are allowed to have proportionally larger distances between the quorum nodes, with no penalty.

To implement the level i regional directory RD_i:
Use a 2^i-regional matching.
Information being kept in RD_i for node v is actually recorded at all the nodes in $Write(v)$.
To read the information starting from node w, read everything in $Read(w)$.
If $dist(v, w) \leq 2^i$, this is guaranteed to return the last thing written in the directory at v.

More specifically, in terms of actual directory operations:
$INSERT(\xi, s)$: Implemented using writes to the nodes in $Write(s)$.
$DELETE(\xi, t)$: Same
$FIND(\xi, v)$: Implemented using reads of the nodes in $Read(v)$.

2.7 Extensions to allow concurrency

We can presume that $MOVE$s for the same user are done sequentially—that's easy enough to ensure, since the processing can be "centralized" at the user's location.

2 $FIND$s don't interfere.
So, the real problem is what happens if a $FIND$ for ξ proceeds concurrently with a $MOVE$ of ξ.

They would like to try to guarantee, under reasonable conditions, that the $FIND$ will succeed in tracking down the moving user.

Some tricks they use:
Slowing down the user's moves enough to ensure that concurrent $FIND$s can catch it.
Requiring stronger conditions for their quorum systems, so that a Read set is guaranteed to intersect all the "relevant" Write sets, for nodes within the distance covered by concurrent $MOVE$s.
Coordination using synchronized clocks.

3 The GRID Location Service

3.1 Overview

Grid Location Service (GLS), intended for mobile ad hoc networks.
In this paper, node addresses are 2D geographical cordinates, unlike in the Awerbuch-Peleg paper, in which they were just graph nodes.
GLS allows a mobile node x anywhere in the network to submit a $FIND(y)$ request, asking for the address (coordinates) of any other mobile node y.
The service is supposed to respond to x with reasonably up-to-date coordinates for y.
GLS also supports MOVE operations.

GLS is intended to be used with a standard geographical routing method, like the ones we have already studied.
Thus, after x looks up the address of y, x can use geographical routing to send an actual data message to y.
Moreover, the GLS implementation itself uses geographical routing as part of the processing of $FIND$ operations.

The key ideas of this work are three kinds of structures:
1. A static, hierarchical subdivision of the plane into nested squares.
2. A static assignment of unique ids to mobile nodes, from a circularly-ordered set of node ids.
3. A dynamic assignment of a table consisting of particular (node-id, address) pairs to each mobile node.

The pairs in the tables are supposed to contain relatively up-to-date coordinates.
So they have to be maintained in the face of mobility.

Also, the selection of which pairs reside in each table depends on the current distribution of mobile nodes in the plane, in particular, on where they are w.r.t. the hierarchical subdivision of the plane.

The selection of pairs is adapted from a "consistent hashing"
technique of Karger and others, which has appeared elsewhere.

This selection is designed to:
Load-balance, keeping the tables at different nodes of similar sizes.
Support an efficient $FIND$ operation, which follows a greedy strategy, repeatedly moving to the node whose id is "closest" to that of the intended target, among those whose coordinates appear in the local table.
The table organization is intended to ensure that the number of steps performed during each $FIND$ is proportional to the log of the Euclidean distance from the source to the target.
Moreover, the number of hops that are required for the first such step (using geographical routing) is approximately the distance between the source and target, and the number of hops for each successive step is approximately half of the number for the previous step.

Thus, the total number of hops used by a $FIND$ should be proportional to the geographical distance between source and target.
Queries for locations of nearby nodes are satisfied with correspondingly local communication.

Experiments show: Scalability, tolerance to failures, recoveries, motion Analysis of correctness and efficiency (in static case).

Their main emphasis is on scalability.
They talk about spreading the network around a campus, or even a large metropolitan area.

3.2 The structures

3.2.1 Node identifiers

We assume that the mobile nodes have unique ids, chosen from a discrete circularly-ordered set, e.g., integers mod something, going clockwise around a circle.

When we talk about a node "closest" to node x in some set X, we mean the one in X whose id is first encountered, starting from x and moving in the clockwise direction; we count x itself if $x \in X$.

3.2.2 The planar subdivision

They use nested squares, of order $1, 2, ..., k$.
Order-1 squares are the basic unit squares.
The entire plane is partitioned into $2^{k-1} \times 2^{k-1}$ order 1 squares; it is an order-k square.
In general, an order-$(i+1)$ square is composed of four order-i squares.

We assume that every mobile node, at any time, is at some location in the overall square.
However, we want to assume it is in only one square at any time.
So, we could use some kind of tiebreaker for boundaries, or rule out the nodes from residing exactly on the boundaries.

3.2.3 Tables

Now, what pairs should be stored in a node's table?
We describe this for the static case, assuming nodes are in fixed locations.

Namely, for each node x at geographical location ℓ, and each node y:
Put pair (x, ℓ) in y's table if one of the following hold:

1. x and y are in the same order-1 square.
2. For some $i, 1 \leq i \leq k-1$:
   x and y are in the same order-$(i+1)$ square but in different order-i squares, and y's id is the closest to x's id (in clockwise order), among the nodes in y's order-i square.
   (That is, there is no y' in the same order-i square as y such that $x < y' < y$.)

Thus, assuming no order-1 squares are vacant, a single (x, ℓ) will appear in $3(k-1)$ tables, 3 corresponding to each order, plus all the tables of the nodes in the same order-1 square.
It's 3 because each square has 3 "sibling" order-i squares within the same order-$(i+1)$ square.

Also, for each node y, the entries in its table will correspond to different levels.
The entries for lower levels correspond to nodes that are probably closer geographically to y.
(It's not clear that there are more entries for higher levels, though: although the "load" of potential nodes that COULD have their entries stored at y is greater at higher levels, it is not so likely that y is the chosen node—it has many other neighbors to share the load.)

3.3 FIND queries using the tables

3.3.1 The FIND protocol

Suppose that $FIND$ is invoked at node s, to locate node t. Then:

- Node s sends the query message to the node x whose id is closest to that of t among those whose coordinates appear in s's local table.
- If node $w \neq t$ receives a query message destined for t, then w sends the query message to the node x whose id is closest to that of t among those whose coordinates appear in w's local table.
- If node t receives a query message destined for itself, then t sends back (geographical routing) its own coordinates to the site s that initiated the $FIND$.

3.3.2 Termination and complexity bound

Why does this work? The key is an invariant that holds after any number j of steps of the $FIND$ algorithm (invoked from s for target t):

Claim: After at most j steps of $FIND$, $1 \leq j \leq k$, the query is at the closest node to t, from among those in the order-j square containing s.

Thus, the query progresses systematically, finding the closest node to t in successively larger squares containing the source s.
Of course, that means that, by the time $j = k$, we have found the closest node to t in the entire square, which of course is t itself.

Proof is inductive: See p. 125; it's the heart of the paper.

Base case: $j = 1$
This says that, after either 0 or 1 steps, the query has reached the closest node to t from among those in the order-1 square containing s.
If s is itself the best, then we're done, with 0 steps.
If not, then note that (we have assumed) that s knows about everyone in its order-1 square.
So in 1 step, it can certainly route the query to the best one, say u.

But we could have a problem: suppose that s happened to have something else in its table, say, v, that isn't in the same order-1 square, but that is closer to t than u is (in id-space).

Draw ordering diagram: t–v–u–s
But now they have to use the very precise way that the tables are constructed, to argue that s would in fact *not have* v in its table.
For, if v were in s's table, it would mean that s is the closest node to v among the nodes in some square (of some order)—that's why s was chosen to hold v's address info.
But u is in the same order-1 square as s, and has an id between v and s—so u would have been a better choice than s, and s would not have been chosen.
End of base case

Inductive step:
This is similar to the base case.
Now we can suppose that the query starts out at the closest node, s', to t among those in the order-j square containing s.
And we want to show that within either 0 or 1 more steps, it has reached the closest node to t from among those in the order-$(j + 1)$ square containing s.

If s' is itself the best, then we're done, with 0 more steps.

If not, then we claim that:
1. s' has, in its table, the closest node, u, to t in the order-$(j + 1)$ square, and
2. s' does not contain any other node that is closer to t.
That's enough to show that we get to the right node u in 1 more step.

For 1: s' has u in its table because s' is the closest node to u in the order-j square of s'.
Why? We have the ordering t–u–s', since u is closer to t than s' is.
We know we have nothing else between u and s' in the order-j square of s', since then it would be closer to t than s' is, contradicting our assumption about s'.

For 2: Suppose s' had something else in its table, say v, that isn't in the same order-$(j + 1)$ square, but nevertheless is closer to t than u is.
Ordering: t–v–u–s'

But then s' would in fact not have v in its table:
For, if it did, then s' would be the closest node to v among the nodes in some square (of order $\geq j+1$)—that's why s' was chosen to hold v's address info.
But u is in the same order-$(j+1)$ square as s', and has an id between v and s'—so u would have been a better choice than s'.
End of inductive step

3.3.3 Comments

Notice that we are explicitly relying on the tables NOT to contain certain elements.
That precludes solutions whereby we include extra location information as we discover it (by overhearing, e.g., in promiscuous mode).

What would go wrong if we did?
We'd still get termination.
But now we might have long paths—taking more steps than we need, and some might take us to greater distances than we need.

Clever algorithm

So far, the algorithm is presented as a static structure—we need to consider how it could be adapted to a dynamic setting.
They give some ideas, but more work could be done on this.

3.4 Initializing and maintaining the tables

3.4.1 Initialization

p. 123-124:
Suppose the network is static, but no one has anything in their tables.
Filling them appropriately sounds like a very expensive task—every node x would have to send its info to several chosen nodes, chosen as the closest ones in certain squares.
This sounds like it could involve searching through entire squares to find the closest one.

But it isn't that bad:
They show how to fill the tables reasonably efficiently—without global searches, flooding info, etc.
In fact, the procedure to fill the tables is similar to the $FIND$ procedure!

The method works bottom up, level by level.
Assume synchronous levels.
First, all the nodes place their information in all the nodes in their order-1 square.
(This much can be done by local broadcast, or a simple local flood.)

Then, for each level i in turn, $i = 1, ..., k-1$:
All nodes x place their information in the desired "servers" in the order-$(i+1)$ square (the ones that aren't in their order-i square).

So, when the nodes are trying to find their order-$(i+1)$ servers, they can assume that everyone already has set up their order-i servers.
Thus, every node within an order-i square already has all its table entries for nodes within that same order-i square.

For example, suppose node x wants to recruit a server in an order-i sibling within its order-$(i+1)$ square.
Then it sends its location info to the area of that square (anywhere), using geographic forwarding.
Someone in that square should get it, say, node y.
Then let y begin a PSEUDO-FIND for the target x, which proceeds just like an ordinary FIND.
That PSEUDO-FIND will remain within y's order-i square, and will eventually locate the closest node to y in the entire square.
That's what we wanted—this is the desired server.

Note that this works because we assume that all the nodes within that square already have the table entries set up for the nodes within that square.

3.4.2 Maintaining the tables

We have to worry about joins, leaves, mobility.
We would, at least, like the data structure to recover to its good state if the changes every stop.
Of course, we would really like it to work even during changes.

They use several ideas for maintenance, described in section 5:

For the geographic forwarding protocol:
Refresh neighbor information using Hello messages, timeouts.

Updating location servers when a node moves:
Not every time—that would be too much communication.
Rather, they use the Awerbuch-Peleg idea: Update order-i servers when you've moved 2^{i-1} distance.
Thus, updates are sent more frequently to local servers than more distant ones.

Invalidating location table entries:
They have a timeout value.
Even when a node doesn't move, it thus has to periodically send its location to its servers, in order to refresh the location info.

Caching:
Nodes can cache locations they hear about, to use in sending data via geographic routing—but these entries aren't put into the same table used by FIND—recall that it's important that certain entries NOT be in these tables.

In section 5.4, they describe an interesting scenario where a node y that was acting as a server moves away from its location.
If someone then routes a packet intended for target t to that node y, it will get lost!
So, it can leave "forwarding pointers" at ALL THE NODES in its old order-1 square, pointing to

its new order-1 square.

Conceptually, they say, we can think of the forwarding pointers as being located "in the square" instead of at the nodes.
So, as nodes enter and leave a square, they should pick up the forwarding pointer information and pass it on to others.
They even give a little gossip mechanism, but limited to the boundaries of the square, to propagate and maintain this info.
That way, the information can persist beyond the time when any particular node is resident in the square.

But how do these get maintained? We don't want to set up long chains of forwarding pointers...
It would be better to pass on the information in the routing table then just the pointers...???

3.5 Simulations

They tested a lot of things, got excellent performance.
Small tables, low communication overhead.
Costs scale very well with the number of nodes.

Tradeoffs between cost of table maintenance and performance of FINDs.

Simulation scenario:
ns-2 with CMU wireless extensions
Random placement, random waypoint mobility
...

4 Abraham et al. LLS

[[[William to insert something. For now, these are just my own reading notes.]]]

1. Overview

A location service for the unit disk model.

They use an abstractly-defined cost measure, defined in terms of an abstract cost function on individual edges (actually, based on edge lengths).

They assume they are given a geographic routing capability, with worst case cost $O(d^2)$ and average cost O(d), where d is the Euclidean distance from source to destination. The average here is taken over all (source,destination) pairs. Implicit in this assumption is another assumption: that, on the average, the cost of a min-cost path from s to t is O(d). This assumption is reflected in the way they state their results—for the worst-case bound result, in terms of the min-cost path, but for the average-case bounds, in terms of Euclidean distances.

(Actually, they are not careful about their calculations of averages—assuming that routing cost is low for average pairs does not necessarily imply that it is low for the particular pairs that happen to be used in the implementations of FIND and MOVE—are these pairs also uniformly distributed?)

Their claims about LLS: 1. Cost of a FIND(t) from s, worst case: $O(d^2)$, where d is the min cost of a path from s to t. 2. Cost of FIND(t) from s, average case (average over all (s,t) pairs): O(d) where d is the Euclidean distance between s and t. 3. Cost of MOVE(s,t), average case: O(d log d), where d is the Euclidean distance between s and t.

They refer to FIND as "lookup", and MOVE as "publish".

2. Related work

Geographical routing: Kranakis, Bose, GPSR, Kuhn,... They assume a protocol (like Kuhn's?) with the bounds described above.

Location services:

Home location strategy: Every node has a "home location", obtained by a hash function. Keeps that location updated with its latest address. Requests to FIND go to the home location. This works best when there are some fixed nodes to act as home locations; more challenging to implement this approach in a fixed network.

Hubaux et al defined home as a geometric area, let all the mobile nodes in the area store the location info. Similar idea to Geoquorums.

Costs of home location strategies not local—can be high relative to actual distances involved.

Replicate home locations using quorums.

Awerbuch-Peleg: They use a hierarchy of partitions, as in this paper. But: They don't use geographical info. Not highly dynamic. Local bounds, but not as small as those here.

GRID Location service: Hierarchy similar to the one here. Clever hashing technique, load-balancing. But: Does not give as good bounds as in this paper. Their scheme doesn't try to proactively handle updates and out-of-date information. Boundary crossing leads to expensive operations: publish costs can be arbitrarily high w.r.t. distance travelled. Authors leave handling of mobility as open question. Nodes on opposite sides of boundary can be arbitrarily close, but be co-located only in very high levels of the hierarchy; leads to arbitrarily high cost lookups. Routing to a location server instead of directly to a destination can impose extra costs. Thus, no worst-case locality guarantees.

Iterative Bounded Flooding (IBF): Flood to successively larger distances 2^i. (Assuming low density?) They get worst-case cost $O(d^2)$ to reach the destination t, where d is min cost of path to t.

Demirbas, Nolte,... Similar algorithm, with O(d log d) MOVE and O(d) FIND (worst-case? under what assumptions?) Tina can discuss this briefly.

3. Model and Notation

n nodes in a bounded square of size M x M in 2D, coordinates ranging from 0 to M in both directions. Names v.id Unit disk graph network All nodes know their own locations, and the identities and locations of their neighbors.

Assume lower bound on how close nodes can be(!), but claim that the assumption could be removed using a connected dominating set backbone as in the Kuhn paper.

Cost function c, nondecreasing function of edge lengths Generalizes hop count, Euclidean distance, and energy bound Sum over edges in path

Virtual coordinates: The algorithm works in terms of geographical coordinates. However, there is no guarantee that there is a real mobile node exactly at a designated geographical coordinate. For

any point p in the plane, at any time, they define ell(p) to be the closest mobile node to point p. They assume they can adapt their underlying geographical routing algorithm so that, if it originally was supposed to reach p, it now actually reaches the node ell(p). They refer to the GHT paper for such a modification. The GHT algorithm adapts to node mobility, letting the real nodes exchange information periodically about which points they are "covering" (that is, which points are in their Voronoi polygons). So from now on, they will sometimes just assume that each p has a real node—a nice abstraction.

They define cost measures, corresponding to the three results listed above. They express these in terms of a parameterized notion of "locality-awareness"; I'll skip these.

4. Hierarchical lattice structure

For any node t, they use a home location H(t.id). This is a static location, independent of where the node t actually moves. Next, they consider all the squares, of successively larger sizes 2^i, that contain this home location at their centers. These are nested squares. Let $L_k(t.id)$ denote the set of 4 corner points of the square centered at H(t.id), of side 2^k. Thus, different nodes t yield different lattices $L_k(t.id)$—the division of the plane is determined by a particular center point H(t.id).

Now consider any arbitrary point x in the plane. It is contained is some of the nested squares of t's hierarchy. Define $W_k(t.id, x)$ to be the set of 4 corner points of the square of t's hierarchy at level k (side length 2^k) that contains x. That is, $W_k(t.id, x)$ locates x within one square of the lattice $L_k(t.id)$.

5. Spiral and Spiral-Flood

These two algorithms yield efficient FIND, but inefficient MOVE; LLS adds an efficient MOVE strategy.

5.1. Spiral

Each t maintains its location info at all the nodes in the corners of all the squares of t's lattice that are centered at t's current location. That is, at all the nodes $W_k(t.id, x)$, for all k, where x is t's current location.

MOVE(t,x,y): This is heavy-weight: Removes all the info from the previous $W_k(t.id, x)$ nodes and reinstates the new info in all the $W_k(t.id, y)$ nodes.

They depict the strategy, in Figure 2, as "following a spiral". Really, they are just updating the corner nodes of these nested squares; the spiral pattern is just a convenient way to order these points.

FIND(t,s): Invoked from s. This queries points in the squares of t's lattice that contain s, that is, the nodes in $W_k(t.id, s)$. The query proceeds outward from s's closest-containing square. The order is like that of the MOVE processing—outward along a spiral.

Eventually, the spiral being used by this FIND will intersect the spiral used by the most recent MOVE, that is, will hit locations $W_k(t.id, x)$ where x is t's current location. This is guaranteed to happen at the first hierarchy level where the s and x are in the same square, of t's hierarchy. Then the FIND will learn the actual location.

Their main result here says that Spiral has property 2 in the overview above. It assumes the efficient geographical routing discussed earlier.

Theorem 5.3: Expected cost of FIND(t) invoked at s is O(d), where d is the Euclidean distance between s and t.

Proof: The FIND will get the info it needs by the time it reaches the smallest square contining both s and t. In general, s and t could be arbitrarily close together, and yet this smallest square could correspond to a very high level in the hierarchy. They avoid this problem simply by averaging over all (s,t) location possibilities.

5.2. Spiral-Flood

Spiral obviously has a problem near the boundaries, which prevents worst-case FIND behavior from depending only on geographical distance. What they do here is simply "back up" the Spiral FIND algorithm by combining it with IBF. Namely, they consider the particular spiral-shaped path followed by the Spiral algorithm. When the Spiral FIND has accumulated sufficient cost (2^{2i}), they switch to flooding to depth 2^i; switching according to this policy ensures that the combined cost is not more than a constant times the cost of Spiral alone. But also, the cost doesn't exceed the cost of IBF alone, which is $O(d^2)$ in the worst case.

So, Spiral-Flood gives both properties 1 and 2 in the overview. It circumvents the grid-boundary problem.

6. LLS

Modifies Spiral-Flood to get good MOVE behavior.

First, for each node t, and for each level k, they store location info not only in the (corners of the) one square containing the current location of t (the $W_k(t.id, x)$ points, where x is t's current location), but also in the 8 squares around it—a total of 16 nodes, called here $Z_k(t.id, x)$. This will allow a little slack in updating the information—the information will be updated slightly "lazily".

Thus, when t moves within such a 9-square boundary, nothing happens. FINDs could then lead to a square not containing t, but then one of the adjacent squares does. Update location info when t leave the 9-square boundary, at which time we know that the distance traversed by the node is already geq the side of one square; thus, amortized cost of updating is kept low, as a function of actual distance traveled.

The information stored at the $Z_k(t.id, y)$ nodes is now different from before: instead of actual location info, they store forwarding pointers to the $W_{k-1}(t.id, y)$ nodes—these nodes are lower in the hierarchy, and are supposed to contain better information for reaching t. The $Z_0(t.id, y)$ nodes, of course, contain the actual location y.

MOVE: Do a spiral search from the new location y, to find the lowest level k whose $W_k(t.id, y)$ nodes have location info for t. (This is supposed to imply that t has moved outside the 9-square region of level k-1.) Then replace the current location info at level Z_j node, j leq k, with a pointer to $W_{j-1}(t.id, y)$.

FIND: Start a spiral as before, but now the search ends when a pointer of the form (t,W(t.id,y)) is found somewhere. Once such a pointer is found, FIND can just follow the location pointers to locate t.

Q: How does the IBF flooding procedure integrate with this?

Analysis: LTTR. William: Can you say something intelligent about it? I'm not sure why it's OK to calculate averages as they do—what independence assumptions are needed? The rough idea is just that they are amortizing the costs of the updates over a series of MOVE operations, and making sure that before they actually do any work to update the tables, the node has moved correspondingly far.

They claim some fault-tolerance properties, because of: Redundancy provided by using multiple nodes. Locality of information-processing The use of flooding.

Simulations validate the theoretical average-case claims.

5 STALK (Demirbas, et al)

STALK, like LLS, borrows its core tracking ideas from Awerbuch/Peleg.
Its main difference from LLS is in its model assumptions.

Unlike LLS, STALK does not assume nodes have access to location information.
Because of this, STALK uses one node hierarchy for the network, rather than one per node.
We'll see in another lecture an example of an algorithm to help us compute such a hierarchy in a distributed fashion.
The hierarchy required by STALK is similar to that of LLS, in that clusters in the network have sizes exponential in level.

5.1 Model assumptions

Stationary sensors, one mobile node (evader).
No location info.
Access to timers (used to provide a fault-containment property).

Hierarchy (see Figure 1):
Sensors partitioned into level 0 clusters.
Each level 0 cluster would have one clusterhead.
Clusterheads of level 0 clusters would combine to create level 1 clusters, and so on.
Clusters at a particular level are non-overlapping and the resulting hierarchy would have the property that higher level clusters have larger radius.
A level l cluster has a radius (max distance between clusterhead and any node in cluster) between r^l and mr^l, for m and r constants.
We assume that the service provides a clusterhead at some level with information on who its children are, who its neighbors are, and who is its clusterhead at the next level.
The distance between two non-neighboring clusters at a particular level is assumed to be bounded from below.

5.2 Algorithm

Tracking path:
On top of the hierarchical partitioning service, we construct a tracking path (see Figure 2).
Each node of the tracking path is a mote serving as a clusterhead in the hierarchical partitioning service, pointing to one of its children from the hierarchical partitioning such that the path is rooted at the highest level clusterhead of the partitioning service and the mobile object resides at the end of the path.

In order to prevent dithering problems, where too much work is done when the mobile object moves to a new location that happens to be just across a cluster boundary, we allow one lateral link per level in the tracking path.
These are pointers from one level l cluster to a neighboring level l cluster.

Tracking path updates (grows and shrinks):
For now, consider the case of atomic moves, where a new move cannot occur until an old move is

complete.
When the evader reaches a new location, a grow action is initiated.
The action climbs the hierarchy, propagating pointers, always checking neighbors at each level to see if insertion of a lateral link is possible, and eventually joins the old tracking path.
In the meantime, a shrink action is cleaning up the old pointers bottom up.

Here we can see an example of tracking.
1. We start off with the evader and a complete tracking path (Figure 2).
2. The evader moves to a neighboring level 0 cluster (Figure 3).
The old level 0 link is cleaned up and a new path starts growing from the new location.
The level 0 clusterhead discovers the neighboring level 0 clusterhead that is both on the tracking path and without a lateral link.
It adds a lateral link between them and is done.
3. The evader moves again (Figure 4).
This time we cannot add a lateral link and so have to propagate a pointer to the next level in the hierarchy.
This level 1 clusterhead has a neighbor it can add a lateral link to and does so.
In the meantime, the old chain of pointers is being cleaned up.
This can continue.

Notice we could have cleaned up old pointers in a top down approach like in LLS.
However, since we assumed timers, we could have the clean-up be initiated by nodes on the old path themselves.
These timers have to be tuned though to not allow grows or shrinks to outrace the other.

Finds:
A find is initiated at some level 0 node, and would make its way up the hierarchy, querying parents and neighbors until the tracking path is found.
At that point, it would follow the tracking path pointers down to the leaf which would contain the mobile object.

For reasons similar to those in LLS, we can bound how far up the hierarchy a find would have to go in order to find pointers to the mobile node.

5.3 Some observations

Moves and finds operate with the same amount of work as in the LLS paper.

Moves and finds can operate concurrently with a couple simple tweaks, which also preserve the work bounds.

Algorithm is actually fault-containing self-stabilizing, meaning that if the node state were to get arbitrarily screwed up somehow, the screwup doesn't propagate very far in the hierachy before being corrected.