

1 Introduction

This lecture notes is based on the presentation of the paper “Concurrent online tracking of mobile users”(Awerbuch and Peleg) and “A Scalable Location Service for Geographic Ad Hoc Routing”(Li et al) by Bibudh Lahiri.

The main difference between the two papers is that the first one deals with tracking the location of a single/multiple mobile *object(s)/user(s)* in a distributed network, whereas the second problem deals with a network where the *nodes are themselves mobile*, and therefore locations of all the nodes are to be tracked by the network.

2 Concurrent online tracking of mobile users

2.1 Problem overview

This paper deals with the problem of maintaining a distributed directory server, that enables us to keep track of mobile users in a distributed network in the presence of concurrent requests. The paper uses the graph-theoretic concept of regional matching for implementing efficient tracking mechanisms. The tracking mechanism has to support two operations: a *move* operation, causing a user to move to a new destination, and a *find* operation, enabling one to contact a specified user at its current address. However, the tasks of minimizing the communication overhead of the *move* and *find* operations appear to be contradictory to each other.

The *full-information strategy* requires every vertex in the network to maintain a complete directory containing up-to-date information on the whereabouts of every user. This makes the *find* operations cheap. On the other hand, *move* operations are very expensive, since it is necessary to update the directories of all vertices. Thus this strategy is appropriate only for a near static setting, where users move relatively rarely, but frequently converse with each other.

The other extreme is the *no-information strategy*, which opts not to perform any updates following a *move*, thus abolishing altogether the concept of directories and making the *move* operations cheap. However, establishing a connection via a *find* operation becomes very expensive, as it requires a global search over the entire network.

This paper makes a trade-off between the costs of *find* and *move*. The intuitive idea behind the protocol suggested in this paper is that, moves to a near-by location, or searches for near-by users, are made to cost less. Their strategy is based on a hierarchy of regional directories, where each regional directory is based on a decomposition of the network into regions. Intuitively, the purpose of the i^{th} level regional directory is to enable any searcher to track any user residing within distance 2^i from it. This structure is augmented with an elaborate mechanism of forwarding pointers.

The organization of a regional directory is based on the graph-theoretic structure of a *regional matching*. An m -regional matching is a collection of sets of vertices, consisting of a *read* set $Read(v)$ and a *write* set $Write(v)$ for each vertex v , with the property that $Read(v)$ intersects with $Write(w)$ for any pair of vertices v, w within distance m of each other. These structures are used to enable localized updates and searches at the regional directories.

2.2 Model

We consider the standard model of a point-to-point communication network. The network is described by a connected undirected graph $G = (V, E)$, $|V| = n$. The vertices of the graph represent the processors of the network and the edges

represent bidirectional communication channels between the vertices. We assume the existence of a weight function $w : E \rightarrow R$, assigning an arbitrary non-negative weight $w(e)$ to each edge $e \in E$. For two vertices u, w in G , let $dist(u, w)$ denote the length of a shortest path in G between those vertices, where the length of a path (e_1, \dots, e_s) is $\sum_{i=1}^s w(e_i)$. Let $D(G)$ denote the weighted diameter of the network G , namely, the maximal shortest distance between any two vertices in G . Throughout we denote $\delta = \lceil \log D(G) \rceil$.

2.3 Problem statement

We denote by $Addr(\xi)$ the current address of the user ξ . A directory server \mathcal{D} is a distributed data structure, that supports the following operations:

Find(ξ, v): invoked at the vertex v , this operation delivers a search message from v to the current location $s = Addr(\xi)$ of the user ξ .

Move(ξ, s, t): invoked at the current location $s = Addr(\xi)$ of the user ξ , this operation moves ξ to a new location t and performs the necessary updates in the directory. The paper provides some guarantees on the stretch of *move* and *find* operations that this directory provides (stretch can be defined as the ratio of communication cost for an operation according to some algorithm to the optimal communication cost for the same operation). Since the proofs are beyond the scope of this paper, they were not included in the presentation, and hence are not included in the notes as well.

2.4 Solution overview

The solution is based on a distributed data structure storing pointers to the location of the users in various vertices. These pointers are updated as users move in the network. In order to localize the update operations on the pointers, we allow some of these pointers to be inaccurate. Intuitively, *pointers at locations nearby to the user, whose update by the user is relatively cheap, are required to be more accurate, whereas pointers at distant locations are updated less often.*

The hierarchical directory server \mathcal{D} is composed of a hierarchy of $\delta = \lceil \log D(G) \rceil$ regional directories \mathcal{RD}_i , with regional directories on higher levels of the hierarchy based on coarser decompositions of the network (i.e., decompositions into larger regions). The purpose of the regional directory \mathcal{RD}_i at level i of the hierarchy is to enable a potential searcher to track any user residing within distance 2^i from it.

The regional directory \mathcal{RD}_i is based on intersecting *read* and *write* sets. A vertex v reports about every user it hosts to all vertices in some specified write set, $Write_i(v)$. While looking for a particular user, the searching vertex w queries all the vertices in some specified read set, $Read_i(w)$. These sets have the property that the *read set of a vertex w is guaranteed to intersect the write set of the vertex v whenever v and w are within distance 2^i of each other.*

How do the **Move** and **Find** operations take place in this hierarchical directory server? Ideally, whenever the user ξ moves, it should update its address listing in the regional directory \mathcal{RD}_i on all levels $1 \leq i \leq \delta$. Unfortunately, such an update is too costly. To prevent waste in performing a **Move** operation we use a mechanism of *forwarding addresses*. Whenever a user ξ moves to a new location at distance d away, only the $\log d$ lowest levels of the hierarchy of regional directories are updated to point directly at the new address. Regional directories of higher levels continue pointing at the old location. In order to help searchers that use these directories (and thus get to the old location), a forwarding pointer is left at the old location, directing the search to the new one.

The search procedure thus becomes more involved. Nearby searchers would be able to locate ξ 's correct address $Addr(\xi)$ directly, by inspecting the appropriate, low-level regional directory. However, searchers from distant locations that invoke a **Find** operation will fail in locating ξ using the lower-level regional directories (since on that level they keep track of users in the nearby locations only). Consequently, the searchers have to use higher levels of the hierarchy. The directories on these levels will indeed have some information on ξ , but this information may be out of date, and lead to some old location $Addr'(\xi)$. Upon reaching $Addr'(\xi)$, the searcher will be redirected to the new location $Addr(\xi)$ through a chain of forwarding pointers. The crucial point is that *updates at the low levels are local, and thus require low communication complexity.*

2.5 Construction of hierarchical directory servers

Each user ξ has a regional address $R_Addr_i(\xi)$ stored for it in each level \mathcal{RD}_i of the regional directory, $1 \leq i \leq \delta$. We denote the tuple of regional addresses of the user ξ by $\bar{A}(\xi) = \langle R_Addr_1(\xi), \dots, R_Addr_\delta(\xi) \rangle$.

The regional address $v = R_Addr_i(\xi)$ does not necessarily reflect the true location of the user ξ , since ξ may have moved in the meantime to a new location v . Thus, for every $1 \leq i \leq \delta$ and every user ξ , at any time, the regional address $R_Addr_i(\xi)$ is either ξ 's current address, $Addr(\xi)$, or one of its previous residences. The only variable that is guaranteed to maintain the true current address of ξ is its lowest level regional address, i.e., $R_Addr_1(\xi) = Addr(\xi)$. To address this problem, this algorithm maintains at each regional address $R_Addr_i(\xi)$ a *forwarding pointer* **Forward**(ξ) pointing at some more recent address of ξ . But *even the vertex pointed at by **Forward**(ξ) may not be the true location of ξ , since ξ may in the meantime have moved further.*

The hierarchical directory server maintains the following invariants:

- **The Reachability Invariant:** The tuple of regional addresses $\bar{A}(\xi)$ satisfies the reachability invariant if for every level $1 \leq i \leq \delta$, at any time, $R_Addr_i(\xi)$ stores a pointer **Forward**(ξ) pointing to the vertex $R_Addr_{i-1}(\xi)$. Thus, the reachability invariant essentially implies that *anyone starting at some regional address $R_Addr_i(\xi)$ and attempting to follow the forwarding pointers will indeed reach the current location of ξ , and moreover, will do so along a path going through all lower-level regional addresses of ξ .*

We associate with each regional address $R_Addr_i(\xi)$ a path denoted by $Migrate_i(\xi)$, which is the actual migration path traversed by ξ in its migration from $R_Addr_i(\xi)$ to its current location, $Addr(\xi)$. As users move about in the network, the system attempts to maintain its information as accurate as possible, and avoid having chains of long forwarding traces. This is controlled by designing the updating algorithm so that it updates the regional addresses frequently enough so as to guarantee the following invariant.

- **The Proximity Invariant:** The regional addresses $R_Addr_i(\xi)$ satisfy the proximity invariant if for every level $1 \leq i \leq \delta$, at any time, the distance travelled by ξ since the last time $R_Addr_i(\xi)$ was updated in \mathcal{RD}_i satisfies $|Migrate_i(\xi)| \leq 2^{i-1} - 1$

In order to guarantee the proximity invariant, the vertex $Addr(\xi)$ currently hosting the user ξ maintains also the following two data structures: the tuple of regional addresses $\bar{A}(\xi)$, and a tuple of migration counters $\bar{C}(\xi) = \langle C_1(\xi), \dots, C_\delta(\xi) \rangle$. Each counter $C_i(\xi)$ counts the distance travelled by ξ since the last time $R_Addr_i(\xi)$ was updated in \mathcal{RD}_i , i.e., $C_i(\xi) = |Migrate_i(\xi)|$. These counters are used in order to decide which regional addresses need to be updated after each move of the user.

2.6 Handling concurrent accesses

The algorithm handles concurrency issues as well, which means, if there is not enough time for the system to complete its operation on one **Find/Move** request before getting the next one, the system would be able to handle both, without leading to inconsistent states, with the following few modifications in the algorithm.

2.6.1 Modifications in the model

One modification that this paper suggests is to time the **Move** and **Find** operations, and adjust the definition of the communication cost for an operation/a series of operations taking the temporal factor into consideration. For any operation X , let $T_{start}(X)$ and $T_{end}(X)$ denote the start and termination times of X , respectively. At any given time τ , we define $Addr^\tau(\xi)$ to be the current residence of the user ξ at time τ . If at this time ξ is in transit on the way from s to t , then its current residence is considered to be node t . The optimal cost of a **Find** operation $F = \mathbf{Find}(\xi, v)$ is redefined to be the maximal distance from v to any location occupied by ξ throughout the duration of the operation, i.e., $\max_{T_{start}(F) \leq \tau \leq T_{end}(F)} \{dist(v, Addr^\tau(\xi))\}$.

2.6.2 Modifications in the algorithm

The concurrent case can be classified into two types. As discussed in the last paragraph of section 2.4, the first part involves the retrieval of some regional address $R_Addr_i(\xi)$ of the user. The second involves proceeding from that address to trace down the user along forwarding pointers. In order to prevent endless chases, the invariant that this modification to the algorithm preserves is that the searcher is allowed to *miss* the user while searching for it on level i only if the user is currently on transition to a new location farther away than distance 2^i .

The modification also allows a user to be *doubly-registered* at both the new and old addresses, when ξ performs $\mathbf{Move}(\xi, s, t)$. As a result, the $\mathbf{Pointer}(\xi)$ mechanism is not necessarily a single pointer any longer, but rather a collection possibly containing two pointers.

3 A Scalable Location Service for Geographic Ad Hoc Routing

3.1 Problem overview

This paper considers the problem of routing in large ad hoc networks of mobile hosts. There are a number of situations in which ad hoc networks are desirable. Users may be so sparse or dense that the appropriate level of fixed infrastructure is not an economical investment. Sometimes fixed infrastructure exists but cannot be relied upon, such as during disaster recovery. Finally, existing services may not provide adequate service, or may be too expensive.

Ad hoc networks are more difficult to implement than fixed networks. Fixed networks take advantage of their static nature in two ways. *First*, they proactively distribute network topology information among the nodes, and each node pre-computes routes through that topology using relatively inexpensive algorithms. *Second*, fixed networks embed routing hints in node addresses. Neither of these techniques works well for networks with mobile nodes because movement invalidates topology information and permanent node addresses cannot include dynamic location information.

This paper describes a system named *Grid* which uses geographical forwarding to take advantage of the similarity between physical and network proximity. A source must know the geographical positions of any destination to which it wishes to send, and must label packets for that destination with its position. An intermediate node only needs to know its own position and the positions of nearby nodes; that is enough information to relay each packet through the neighbor that is geographically closest to the ultimate destination.

Any location service based on geographic forwarding should also be scalable in the following senses:

- The workload of maintaining the location service should be spread evenly over the nodes.
- The failure of a node should not affect the reachability of many other nodes.
- Queries for the locations of nearby hosts should be satisfied with correspondingly local communication.
- The per-node storage and communication cost of the location service should grow as a small function of the total number of nodes.

3.2 The Grid Location Service

The Grid Location Service (GLS) discussed in this paper handles the issues of locality, load-balancing and fault-tolerance in a sense that a node maintains its current location in a number of *location servers* distributed throughout the network, and there is no specially designated location server node that has to store the location information of all nodes in the network. The location servers for a node are relatively dense near the node but sparse farther from node; this ensures that anyone near a destination can use a nearby location server to find the destination, while also limiting the number of location servers for each node. On the other hand long distance queries are not disproportionately penalized - query path lengths are proportional to data path lengths.

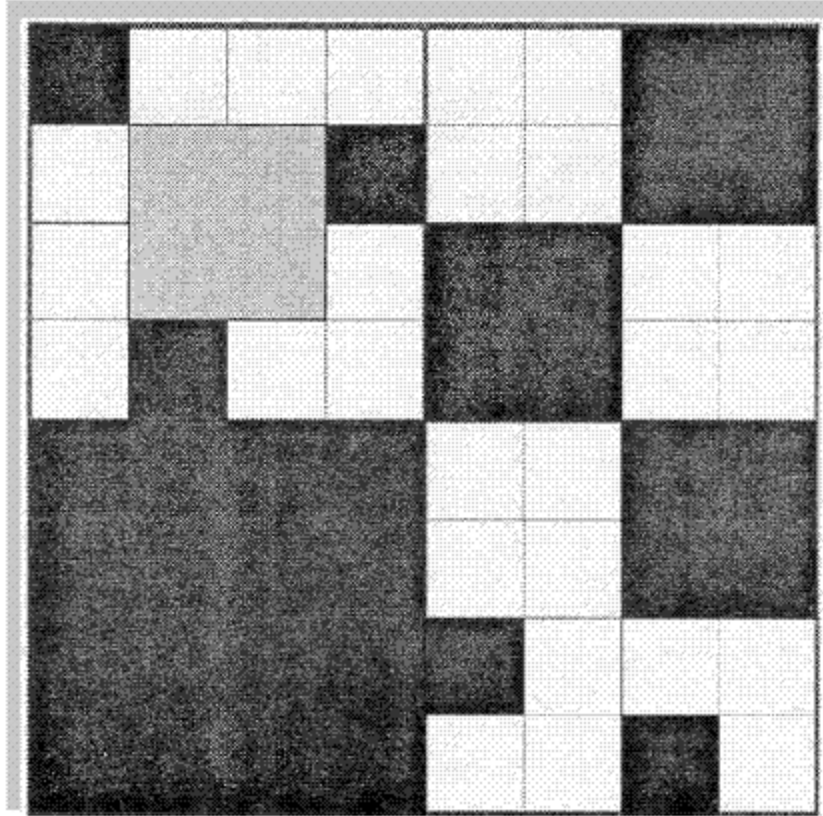


Figure 1: A piece of the global partitioning of the deployment region. A few example squares of various orders are shown with dark shading. The lightly shaded square is shown as an example of a 2x2 square which is not an order-2 square because of its location.

3.2.1 Selecting and Querying Location Servers

A node A hoping to contact node B can query one of a number of other nodes that know B 's location. Of course, A must be able to contact the nodes that know B 's location. This means that A 's search for B 's location servers and B 's original recruitment of location servers ought to lead to the same servers. When B recruits location servers it uses the same information that A will have when searching for B 's location servers - B 's name and certain information that all nodes have at startup.

At startup, all nodes know the same global partitioning of the deployment region into a hierarchy of grids with squares of increasing size, as shown in Figure 1. The smallest square is referred to as an order-1 square. Four order-1 squares make up an order-2 square, and so on. It is important to note that *not every square made up of four order- n squares is also an order- $(n + 1)$ square*. Rather, to avoid overlap, *a particular order- n square is part of only one order- $(n + 1)$ square, not four*. This maintains an important invariant - *a node is located in exactly one square of each size*. This system of increasing square sizes provides a context in which a node selects fewer and fewer location servers at greater distances.

How does a node B determine which nodes to update with its changing location? B knows that other nodes will want to locate it, but that they will have little knowledge beyond B 's ID. B 's strategy is *to recruit nodes with IDs close to its own ID to serve as its location servers*. The authors define the node closest to B in ID space to be the node with the least ID greater than B . The ID space is considered to be circular, so 2 is closer to 17 than 7 is to 17.

If we consider the tree corresponding to the grid decomposition, a node selects location servers in each sibling of a square that contains the node. The exact details of the selection are best understood with an example (see Figure 2). A node

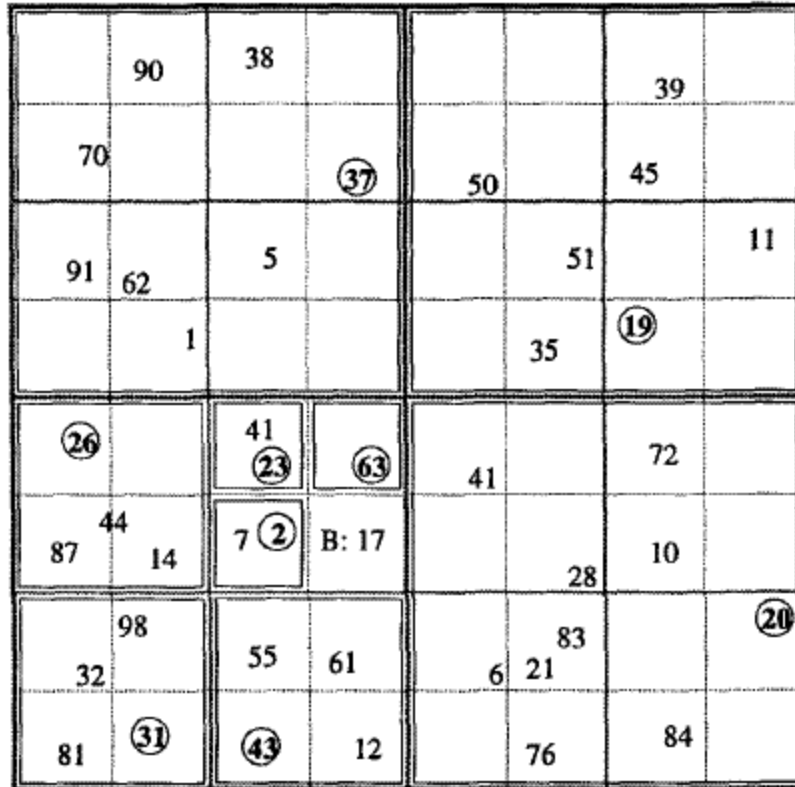


Figure 2: The inset squares are regions in which B will seek a location server. The nodes that become B 's location servers are circled and shown in bold.

chooses three location servers for each level of the grid hierarchy. For example, in the figure, B recruits three servers in order-1 squares (23,63 and 2), three servers in order-2 squares (26,31,43), and three servers in order-3 squares (19,20,37). In each of the three order-1 squares that, along with B 's own order-1 square, make up an order-2 square, B chooses the node closest to itself in ID space as a server. The same location server selection process occurs in higher order squares. In the three order-1 squares that combine with B 's order-1 square to make an order-2 square, B selects 2, 23, and 63 as location servers.

Figure 3 shows the state of a Grid network once all nodes have provided their coordinates to the nodes that will act as their location servers. With the complete network state as reference, we can return to the problem of how A finds the location of B .

To perform a location query, A sends a request (using geographic forwarding) to the least node greater than or equal to B for which A has location information. That node forwards the query in the same way, and so on. Eventually, the query will reach a location server of B which will forward the query to B itself. Since the query contains A 's location, B can respond directly using geographic forwarding. The location query is forwarded all the way to B so that B can respond with its latest location.

	70: 72,76,81 82,84,87	1,5,6,10,12 14,37,62,70 90,91				19,35,37,45 50,51,82
	A: 90	38				39
1,5,16,37,64 63,90,91			16(17): 19,21 23,26,28,31 32,33	19,35,39,45 51,82		39,41,43
70			37	50		45
1,62,70,90	1,5,16,37,39 41,43,45,50 51,55,61,91	1,2,16,37,62 70,90,91			35,39,45,50	19,35,39,45 50,51,55,61 62,63,70,72 76,81 11
91	62	5			51	
	62,91,98				19,20,21,23 26,28,31,32 51,82	1,2,5,6,10,12 14,16,17,82 84,87,90,91 98 19
	1				35	
14,17,19,20 21,23,26,87		2,17,23,63	2,17,23,26 31,32,43,55 61,62	28,31,32,35 37,39		10,20,21,28 41,43,45,50 51,55,61,62 63,70 72
26		23	63	41		
14,23,31,32 43,55,61,63 81,82,84	2,12,26,87 98	1,17,23,63,81 87,98	2,12,14,16 23,63		6,10,20,21 23,26,41,72 76,84	6,72,76,84
87	14	2	B: 17		28	10
31,81,98	31,32,81,87 90,91	12,43,45,50 51,61	12,43,55	1,2,5,21,76 84,87,90,91 98	6,10,20,76	6,10,12,14 16(17),19,84
32	98	55	61	6	21	20
31,32,43,55 61,63,70,72 76,98	2,12,14,17 23,26,28,32 81,98	12,14,17,23 26,31,32,35 37,39,41,55 61	2,5,6,10,43 55,61,63,81 87,98		6(21): 28,41 72	20,21,28,41 72,76,81,82
81	31	43	12		A: 76	84

Figure 3: An entire network's location server organization. Each node is shown with the list of nodes for which it has up to date location information; *B*'s location servers are shown in bold. Two possible queries by *A* for *B*'s location are shown.