Online Tracking of Mobile Users

BARUCH AWERBUCH

Johns Hopkins University, Baltimore, Maryland

AND

DAVID PELEG

The Weizmann Institute, Rehovot, Israel

Abstract. 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. The paper introduces the graph-theoretic concept of *regional matching*, and demonstrates how finding a regional matching with certain parameters enables efficient tracking. The communication overhead of our tracking mechanism is within a polylogarithmic factor of the lower bound.

Categories and Subject Descriptors: B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids—*formal models, verification, worst-case analysis*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture, protocol verification*; D.4.4. [**Operating Systems**]: Communications Management—*network communications*; F.1.2 [**Computation by Abstract Devices**]: Models of Computation—*parallelism and concurrency*

General Terms: Design, Protocols, Reliability, Theory, Verification

Additional Key Words and Phrases: Bounded packet header, bounded protocol, ideal transmission cost, lookahead, non-FIFO channels, receiver-driven protocol, recoverable protocol, recovery cost, sequence transmission problem

1. *Introduction*

Since the primary function of a communication network is to provide communication facilities between users and processes in the system, one of the key problems such a network faces is the need to be able to locate the whereabout

B. Awerbuch was supported by Air Force contract TNDGAFOSR-86-0078, ARO contract DAAL 03-86-K-0171, National Science Foundation contract CCR 86-11442, DARPA contract N0014-89-J-1988, and a special grant from IBM.

D. Peleg was supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award.

Authors' present addresses: B. Awerbuch, Department of Electrical Engineering and Computer Science, The Johns Hopkins University, Baltimore, MD 21218, e-mail: baruch@blaze.cs.jhu.edu; D. Peleg, Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel, e-mail: peleg@wisdom.bitnet.

of various entities in it. This problem becomes noticeable especially in large networks, and is handled by tools such as name servers and distributed directories (cf. Lantz et al. [1985] and Peleg [1989b]).

The location problem manifests itself to its fullest extent when users are allowed to relocate themselves from one network site to another frequently and at will, or when processes and servers may occasionally migrate between processors. In this case, it is necessary to have a dynamic mechanism enabling one to keep track of such users and contact them at their current residence. The purpose of this work is to design efficient tracking mechanisms, based on distributed directory structures, minimizing the communication redundancy involved.

Networks with mobile users are by no means far-fetched. A prime example is that of cellular telephone networks. In fact, one may expect that in the future, all telephone systems will be based on "mobile telephone numbers," that is, ones that are not bound to any specific physical location. Another possible application is a system one may call "distributed yellow pages," or "distributed match-making" [Mullender and Vitányi 1988; Kranakis and Vitányi 1988]. Such a system is necessary in an environment consisting of mobile "servers" and "clients." The system has to provide means for enabling clients in need of some service to locate the whereabouts of the server they are looking for. (Our results are easier to present assuming the servers are distinct. However, they are applicable also to the case when a client is actually looking for one of the closest among a set of identical servers.) The method may also find engineering applications in the area of concurrent programming languages and related areas.

In essence, the tracking mechanism has to support two operations: a "move" operation, facilitating the move of a user 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. This can be realized by examining the following two extreme strategies (considered also in Mullender and Vitányi [1988]).

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.

In contrast, the *no-information* strategy 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. Alternatively, trying to eliminate this search, it is possible to require that whenever a user moves, it leaves a "forwarding" pointer at the old address, pointing to the new address. Unfortunately, this heuristic still does not guarantee any good worst-case bound for the "find" operations.

Our purpose is to design some intermediate "partial-information" strategy, that will perform well for any communication/travel pattern, making the costs of both "move" and "find" operations relatively cheap. This problem was tackled also by Mullender and Vitányi [1988] and Kranakis and Vitányi [1988]. However, their approach was to consider only the global worst-case performance. Consequently, the schemes designed there treat all requests alike, and ignore considerations such as locality.

Our goal is to design more refined strategies that take into account the inherent costs of the particular requests at hand. It is clear that in many cases these costs may be lower than implied by the global worst-case analysis. In particular, we would like moves to a near-by location, or searches for near-by users, to cost less. (Indeed, consider the case of a person who moves to a different room in the same hotel. Clearly, it is wasteful to update the telephone directories from coast to coast; notifying the hotel operator should normally suffice.) Thus, we are interested in the worst case *overhead* incurred by a particular strategy. This overhead is evaluated by comparing the total cost invested in a sequence of "move" and "find" operations against the inherent cost (namely, the cost incurred by the operations themselves, assuming full information is available for free). This comparison is done over all sequences of "move" and "find" operations. The strategy proposed in this paper guarantees overheads that are *polylogarithmic* in the size and diameter of the network.

Our 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 ith level regional directory is to enable any searcher to track any user residing within distance 2^i from it. This limitation implies that the cost of tracking by such a searcher can be made proportional to 2^i. This, combined with the hierarchical structure, enables us to bound the cost of the "find" operation as a function of the user's distance from the searcher. On the other hand, another implication of limiting the operational range of the ith level regional directory to radius 2^i is that it is possible to avoid constant updates due to "move" operations, relying instead on a mechanism of forwarding pointers. A more comprehensive update needs to be performed only once the movements of the user accumulate to a total distance of 2^i or more. The cost of such an update can also be made proportional to 2^i, which enables us to bound the amortized cost of the "move" operations.

The organization of a regional directory is based on the novel 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.

In a more general context, regional matchings provide a tool for constructing cheap locality preserving representations for arbitrary networks. For instance, this structure has recently been used in another application; namely, the construction of a network synchronizer with polylogarithmic time and communication overheads [Awerbuch and Peleg 1990b].

The construction of regional matchings is based on the concept of sparse graph covers [Peleg 1989a; Awerbuch and Peleg 1990a]. Such covers seem to play a fundamental role in the design of several types of locality preserving network representations. Indeed, cover-based network representations have already found several applications in the area of distributed network algorithms.[1] The construction of sparse covers and partitions can be achieved using clustering and decomposition techniques.[2]

Recently, the problem of tracking mobile users has received some attention in the model of wireless networks. In that model, the problem is complicated by the fact that, in order to save costly wireless communication, mobile users do not always inform the network nodes about their exact location. Hence, a host node might not be aware of a mobile user that is currently residing at it. This necessitates some additional tracking mechanisms in such systems. See Bar-Noy and Kessler [1993] and Bar-Noy et al. [1994].

The rest of the paper is organized as follows: The next section contains a precise definition of the model and the problem. In Section 3, we give an overview of the proposed solution. The regional directory servers (and the regional matching structure upon which they are based) are described in Section 4. The main, hierarchical directory server is described in Section 5. The mechanism is described under the assumption that "move" and "find" requests arrive sequentially. Section 6 describes how to extend our solution to allow concurrent accesses. Finally, Section 7 concludes with a discussion.

2. *The Problem*

2.1 THE 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. A vertex may communicate directly only with its neighbors, and messages to nonneighboring vertices are sent along some path connecting them in the graph. It is assumed that efficient routing facilities are provided by the system.

We assume the existence of a *weight* function $\omega: E \rightarrow \mathscr{R}$, assigning an arbitrary nonnegative weight $\omega(e)$ to each edge $e \in E$. The weight $\omega(e)$ represents the length of the edge, or the cost of transmitting a message on it.

2.2 CODING CONVENTIONS. Our algorithms are described based on what may be called "sequential" representation. That is, the code describes a single process, which is active at any given moment in a single vertex in the network. This vertex is referred to as the protocol's "center of activity," and is denoted in the code by $\otimes$. The center of activity may move around the network, via messages issued by the protocol.

Our distributed protocols make use of certain variables. These variables may belong to one of a number of types, differing in the nature of their ownership and physical location. The first type is a *local variable*, which is a variable stored in one particular vertex. We shall denote a local variable `var` stored in the vertex u by $\mathtt{var}^u$. A number of vertices may possess an identically-named local variable, for example, there may be a variable named $\mathtt{var}^u$ at every vertex u in the network. Our code may occasionally refer to "the local variable `var` stored at the current center of activity." This will be denoted by $\mathtt{var}^{\otimes}$.

[1]For example, see Peleg [1989b], Peleg [1989a; 1989b], Awerbuch et al. [1989; 1991b; 1992c; 1992d], and Awerbuch and Peleg [1990b; 1990c].
[2]These techniques were developed in Awerbuch [1985], Peleg and Schäffer [1989], Peleg [1989a], Awerbuch and Peleg [1990a; 1990d], Linial and Saks [1991], and Awerbuch et al. [1991a; 1992a; 1992b].

A second type of variables consists of those variables belonging to the user itself. More specifically, the representation of the user ξ (for our tracking purposes) consists of a "mobile data structure" containing a number of variables. These variables are special in that they "travel" along with the user, rather than stay at one vertex, and we shall superscript them by ξ.

A third type of variables consists of those variables carried along by the process itself. That is, the process executing the algorithm may own certain variables, and carry them along with it whenever its center of activity moves from one vertex to another. We shall superscript such a variable by $\hookrightarrow$.

Finally, in certain cases, the code makes use of a temporary variable whose ownership is of no particular consequence. We will simply leave such a variable unsuperscripted.

Our code uses several commands suitable for a distributed environment. The first is "**transfer-control-to** v," which means that the center of activity is moved to vertex v. When we mention a variable of the protocol, we refer to the variable at the current location of the center of activity.

When we write $\texttt{Local\_var}^{\otimes} \leftarrow$ **remote-read** $\texttt{Remote\_var}^{v}$, while the center of activity is located at the vertex u, we mean the following: go from u to v, read variable $\texttt{Remote\_var}^{v}$, return to u and write the retrieved value into variable $\texttt{Local\_var}^{u}$ at u. At the end of this operation, the center of activity remains at u.

Similarly, $\texttt{Remote\_var}^{v} \leftarrow$ **remote-write** $\texttt{Local\_var}^{\otimes}$, means that the value of the variable $\texttt{Local\_var}^{u}$ at the current center of activity u is retrieved, and the center of activity carries it from u to v and writes it into the variable $\texttt{Remote\_var}^{v}$ at v. The center of activity then returns to u.

2.3 Graph Notation. 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 $(e_1, \ldots, e_s)$ is $\sum_{i=1}^{s} \omega(e_i)$. Let $D(G)$ denote the (weighted) *diameter* of the network G, namely, the maximal distance between any two vertices in G. Throughout, we denote $\delta = \lceil \log D(G) \rceil$.

For a vertex $v \in V$, let $r(v, G) = \max_{w \in V} (dist(v, w))$. Let $R(G)$ denote the *radius* of the network, that is, $\min_{v \in V} (r(v, G))$. A *center* of G is any vertex v realizing the radius of G (i.e., such that $r(v, G) = R(G)$).

Let us now introduce some definitions relevant for covers. Given a set of vertices $S \subseteq V$, let $G(S)$ denote the subgraph induced by S in G. A *cluster* is a subset of vertices $S \subseteq V$ such that $G(S)$ is connected. A *cover* is a collection of clusters $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $\bigcup_i S_i = V$. Given a cover $\mathcal{S}$, let $\hat{R}(\mathcal{S}) = \max_i R(G(S_i))$. For every vertex $v \in V$, let $deg_{\mathcal{S}}(v)$ denote the degree of v in the hypergraph $(V, \mathcal{S})$, that is, the number of occurrences of v in clusters $S \in \mathcal{S}$. The *maximum degree* of a cover $\mathcal{S}$ is defined as $\Delta(\mathcal{S}) = \max_{v \in V} \deg(v, \mathcal{S})$.

Given two covers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and $\mathcal{T} = \{T_1, \ldots, T_k\}$, we say that $\mathcal{T}$ *coarsens* $\mathcal{S}$ if for every $S_i \in \mathcal{S}$ there exists a $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

The *j-neighborhood* of a vertex $v \in V$ is defined as $N_j(v) = \{w \mid dist(w, v) \leq j\}$. The *j-neighborhood cover* of the graph G is the collection of all j-neighborhoods in the graph,

$$\mathcal{N}_j(V) = \{N_j(v) \mid v \in V\}.$$

2.4 STATEMENT OF THE PROBLEM. Consider a user ξ moving about in the network G. Denote by $Addr(\xi)$ the current address of the user ξ. A *directory server* $\mathscr{D}$ is a distributed data structure (the *directory*), combined with access protocols that enable one to keep track of the users' movements and to find them whenever needed. Namely, the access protocols enable their users to perform the following two operations.

FIND$(\mathscr{D}, \xi, v)$. 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$(\mathscr{D}, \xi, s, t)$. 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.

(We may omit the reference to $\mathscr{D}$ when it is clear from the context.)

It is assumed that each vertex maintains a list of the users currently residing at it, so once the search process performed by the FIND operation has reached s, the fact that s is $Addr(\xi)$ can be readily verified.

For simplicity, we assume at first that individual activations of the operations FIND and MOVE do not interleave in time, that is, are performed in an "atomic" fashion. This enables us to avoid issues of concurrency control, namely, questions regarding the simultaneous execution of multiple FIND / MOVE operations. The necessary modifications for handling the concurrent case are outlined in Section 6.

2.5 COMPLEXITY MEASURES. Communication complexity is measured as follows: The basic message length is $O(\log n)$ bits. Longer messages are charged proportionally to their length (i.e., a message of length $l > \log n$ is viewed as $\lceil l/\log n \rceil$ basic messages). The *communication cost* of transmitting a basic message over an edge e is the weight $\omega(e)$ of that edge. For a *protocol* π (which could be a single operation such as FIND or MOVE, a sequence of operations or a procedure consisting of operations plus other control messages), the communication cost of π, denoted $Cost(\pi)$, is the sum of the communication costs of all message transmissions performed during the execution of the protocol.

The assumption of efficient routing facilities in the system is interpreted in this context as follows: Suppose that processor v has to send a message to processor u. Then the message will be sent along a route as short as possible in the network, and the cost of the routing is $O(dist(u, v))$.

We are interested in measuring the communication complexity of the FIND and MOVE operations in our directories. More specifically, we study the overheads incurred by our algorithms, compared to the minimal "inherent" costs associated with each FIND and MOVE operation. Consequently, let us first identify these optimal costs.

Consider a FIND instruction $F =$ FIND$(\mathscr{D}, \xi, v)$. Recall that $Cost(F)$ denotes the actual communication cost of F. Define the *optimal cost* of F as $Opt\_cost(F) = dist(v, Addr(\xi))$.

Now consider a MOVE instruction $M =$ MOVE$(\mathscr{D}, \xi, s, t)$. Its actual cost is denoted $Cost(M)$. Let $Reloc(\xi, s, t)$ denote the relocation cost of the user ξ from s to t, namely, the cost of moving the stored information associated with the user ξ itself (containing its own private variables, etc.) from s to t. We define the *optimal cost* of the operation M as $Opt\_cost(M) = Reloc(\xi, s, t)$, which is the inherent cost assuming no extra operations, such as directory updates, are taken. This cost depends on the distance between the old and new location, and we assume it satisfies $Reloc(\xi, s, t) \geq dist(s, t)$. (In fact, the relocation of a server is typically much more expensive than just $dist(s, t)$, which is the cost of sending a single basic message between the two locations.)

We would like to define the "amortized overhead" of our operations, compared to their optimal cost. For that purpose we consider mixed sequences of MOVE and FIND operations. Given such a sequence $\bar{\sigma} = \sigma_1, \ldots, \sigma_l$, let $\mathscr{F}(\bar{\sigma})$ denote the subsequence obtained by picking only the FIND operations from $\bar{\sigma}$, and similarly let $\mathscr{M}(\bar{\sigma})$ denote the subsequence obtained by picking only the MOVE operations from $\bar{\sigma}$ (i.e., $\bar{\sigma}$ consists of some shuffle of these two subsequences).

Define the cost and optimal cost of the subsequence $\mathscr{F}(\bar{\sigma}) = (F_1, \ldots, F_k)$ in the natural way, setting

$$Cost(\mathscr{F}(\bar{\sigma})) = \sum_{i=1}^{k} Cost(F_i).$$

$$Opt\_cost(\mathscr{F}(\bar{\sigma})) = \sum_{i=1}^{k} Opt\_cost(F_i).$$

The *find-stretch* of the directory server with respect to a given sequence of operations $\bar{\sigma}$ is defined as

$$Stretch_{find}(\bar{\sigma}) = \frac{Cost(\mathscr{F}(\bar{\sigma}))}{Opt\_cost(\mathscr{F}(\bar{\sigma}))}.$$

The find-stretch of the directory server, denoted $Stretch_{find}$, is the least upper bound on $Stretch_{find}(\bar{\sigma})$, taken over all finite sequences $\bar{\sigma}$.

For the subsequence $\mathscr{M}(\bar{\sigma})$, define the cost $Cost(\mathscr{M}(\bar{\sigma}))$, the optimal cost $Opt\_cost(\mathscr{M}(\bar{\sigma}))$, and the *move-stretch* factors $Stretch_{move}(\bar{\sigma})$ and $Stretch_{move}$ analogously.

We comment that our definitions ignore the initial set-up costs involved in organizing the directory when the user first enters the system. These costs are discussed further in Section 7.1.

Finally, define the memory requirement of a directory as the total amount of memory bits it uses in the processors of the network.

2.6 MAIN RESULTS. Our main result is the construction of a *hierarchical directory server*, $\mathscr{D}$, guaranteeing $Stretch_{find} = O(\log^2 n)$ and $Stretch_{move} = O(\delta \cdot \log n + \delta^2/\log n)$ and requiring a total of $O(N \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot \delta \cdot \log^2 n)$ bits of memory (including both data and bookkeeping information) throughout the network, for handling N users, where $\delta = \lceil \log D(G) \rceil$.

3. *Overview of the Solution*

3.1 OUTLINE. Consider the tour taken by the user ξ beginning at its point of origin and ending at its current location $Addr(\xi)$. Let $\bar{H}(\xi) = (Host_1, \ldots, Host_L)$ be the sequence, in chronological order, of vertices visited by the user ξ. That is, $Host_1$ is the user's origin, and $Host_L$ is the user's current location, that is, $Host_L = Addr(\xi)$.

A natural scheme for keeping track of the user's whereabouts is based on maintaining a global distributed data structure, storing in various vertices pointers to the location of the user. These pointers are updated as the user moves in the network.

Ideally, these pointers should always store the exact current address, $Addr(\xi)$, at any given moment. However, such a structure may be too costly to maintain and update. The approach proposed in this paper is based on the idea that in order to localize the update operations on the pointers, we may allow some of these pointers to be inaccurate some of the time. 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.

This idea naturally leads to a hierarchy-based solution. Our *hierarchical directory server* $\mathscr{D}$ is composed of a hierarchy of $\delta = \lceil \log D(G) \rceil$ *regional directories* $\mathscr{RD}_i$, $1 \le i \le \delta$, with regional directories on higher levels of the hierarchy based on coarser decompositions of the network (i.e., decompositions into larger regions). Intuitively, the purpose of the regional directory $\mathscr{RD}_i$ at level i of the hierarchy is to enable a potential searcher to track any user residing within distance $O(2^i)$ from it.

The regional directory operates by associating with the user ξ a pointer, or an "address listing." This address is called the user's ith level *regional address*, and is denoted $R\_addr_i(\xi)$.

Again, it would be best if whenever the user ξ moves, it could update its regional address listings $R\_addr_i(\xi)$ in the regional directories $\mathscr{RD}_i$ on *all* levels $1 \le i \le \delta$. Unfortunately, such an update may be too costly. Therefore, our policy is based on updating the regional addresses only every once in a while. Hence, the user's regional address at a given moment is not necessarily identical to the user's true current address, $Addr(\xi)$, but might be some previous location in the host sequence, that is, $R\_addr_i(\xi) = Host_j$ for some $1 \le j \le L$. Nevertheless, the inaccuracy of user's regional address $R\_addr_i(\xi)$ is guaranteed to be bounded by 2^i, that is, $dist(Addr(\xi), R\_addr_i(\xi)) \le 2^i$.

In order to overcome this imprecision in the regional addresses, we use a mechanism of "forwarding addresses." This mechanism ensures that each old location $u = Host_j$ of the user maintains a pointer directed at some more recent location $Host_{j'}$, for $j' > j$.

The ith level regional directory $\mathscr{RD}_i$ stores the regional address $R\_addr_i(\xi)$ at various vertices in the network, and thus enables potential searchers to find it. The *operational range* of $\mathscr{RD}_i$ is 2^i, that is, it is guaranteed to supply the user's regional address $w = R\_addr_i(\xi)$ to any searcher at distance up to 2^i from w.

The fact that $\mathscr{RD}_i$ has operational range 2^i means that the cost of updating the directory is proportional to 2^i. To compensate for that, the fact that the regional address may be inaccurate allows us to update the directory only once in a while, specifically, whenever the user has moved a distance of 2^i or more since the last update.

Figure 1 illustrates the general structure of the scheme.

Let us now turn to outline the MOVE and FIND operations of the main, hierarchical directory server.

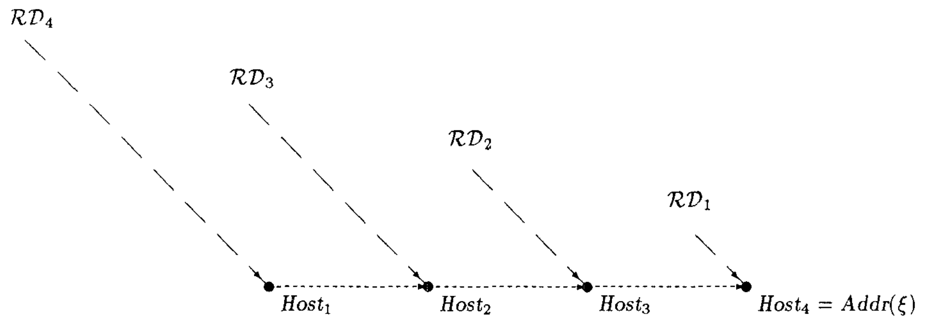

FIG. 1. Schematic description of the $\overline{H}$ sequence and the logical pointers maintained in the hierarchical directory. The dashed arrows represent the "regional address" pointers maintained by the regional directories $\mathscr{RD}_l$. The dotted arrows represent the forwarding pointers leading to ξ's current residence.

Our update policy can be schematically described as follows: Whenever a user ξ moves, a directory of level l is updated if the distance ("mileage") traversed by the user since the last update of regional address on that level exceeds 2^l. As a result of this update, regional addresses of all directories of levels $l' \leq l$ will point directly at the new address. Note that if the distance of the move (distance between current location $Host_L$ to a new location $Host_{L+1}$) is d, then at least $\log d$ lowest level addresses will be updated.

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 $Host_L$, directing the search to the new location, $Host_{L+1}$.

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 (or its regional address) using the lower-level regional directories (since on that level they belong to a different region). Consequently, they have to use higher levels of the hierarchy. The directories on these levels will indeed have the sought information on ξ (namely, its regional address), but this information may be out of date, that is, this regional address may be some old location $Host_j$. Upon reaching $Host_j$, the searcher will be redirected to the new location $Addr(\xi)$ through a chain of forwarding pointers. (This chain will not correspond to the original sequence of addresses occupied by the user, $\overline{H}(\xi)$; rather, it will be "compacted," as described later on, for efficiency purposes.)

Remark 3.1.1. In our description of the mechanism of forwarding addresses, we tacitly make the assumption that the hosts in the sequence $\overline{H}(\xi)$ are all *distinct*. Indeed, note that if the same vertex u occurs a number of times in the sequence $\overline{H}(\xi)$, then it may suffice for it to store only one pointer to a more recent location; and if an old location $u = Host_j$ is identical to the *current* location $Addr(\xi)$, then no forwarding pointer is necessary for it at all.

For simplicity of presentation, we shall ignore this special case of returning to a previously visited location in the sequel. In other words, we will think of each such "repeat visit" as if it visits a "new copy" of the same location, and require each of the copies to participate in the protocols on its own. This will have no adverse implications on the overall complexity of our mechanism.

It is clear, though, that any practical implementation based on the ideas described in this paper will have to accommodate for this special case, both because maintaining multiple "copies" of each location leads to considerably more complex control, and because whenever such repeat visits occur, it may be possible to introduce certain optimizations that can further reduce costs, even though the asymptotic complexity will remain unaffected. We shall generally ignore this optimization issue in the sequel.

3.2 IMPLEMENTATION. The regional directory $\mathcal{RD}_i$ is implemented as follows: As in the match-making strategy of Mullender and Vitányi [1988], the mechanism 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; see Figure 2. The underlying graph-theoretic structure at the basis of this construction is called a 2^i*-regional matching*. (In contrast, the match-making functions of Mullender and Vitányi [1988] and Kranakis and Vitányi [1988] do not have any distance limitation, and they insist on having exactly one element in each intersection.)

It turns out that the parameters of a regional matching relevant for our purposes are its *radius*, which is the maximal distance from a vertex to any other vertex in its read or write set, and its *degree*, which is the maximal number of vertices in any read or write set. In particular, the communication overhead of performing the FIND and MOVE operations in a regional directory $\mathcal{RD}_i$ grows as the product of the degree and the radius of the related 2^i-regional matching. There appears to be a trade-off between these two parameters, making simultaneous minimization of both of them a nontrivial task.

A related graph-theoretic problem is that of designing a "sparse" graph cover $\mathcal{S}$, that is, covering the graph by low-radius clusters with little overlap. The parameters of interest in that problem are thus the radius $\hat{R}(\mathcal{S})$ and degree $\Delta(\mathcal{S})$ of the cover $\mathcal{S}$, as defined in 2.3. Here, too, it is possible to trade-off radius for degree; for example, merging together a number of clusters, we may reduce the maximal degree at the expense of increasing the radius. Although the relationship between matchings and covers is not immediate (and in particular, the definitions of degree seem to measure different quantities), it turns out that there is a strong connection between the two constructs, and we show how given a cover it is possible to obtain a matching with the same degree and radius.

We shall now proceed with a more detailed treatment of the solution. The example at the end of Section 5 may be of further assistance in clarifying the overall structure of the directory server.

4. *Regional Directories*

4.1 THE CONCEPT OF A REGIONAL MATCHING. The basic components of our construction are a read set $Read(v) \subseteq V$ and a write set $Write(v) \subseteq V$,

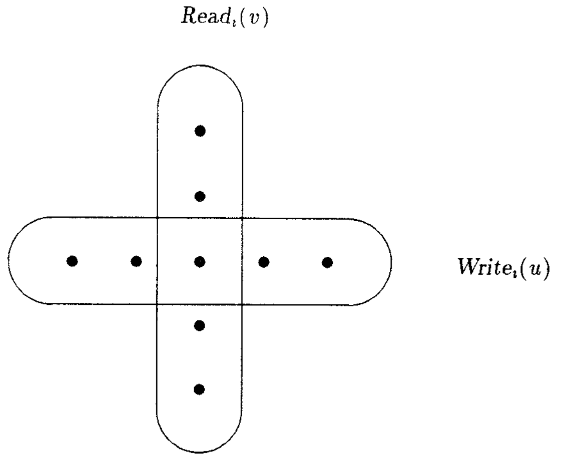

FIG. 2. The sets $Write_i(u)$ and $Read_i(v)$ for u and v satisfying $dist(u, v) \leq 2^i$.

defined for every vertex v. Consider the collection $\mathscr{RW}$ of all pairs of sets, namely

$$\mathscr{RW} = \{Read(v), Write(v) \mid v \in V\}.$$

Regional Matching. The collection $\mathscr{RW}$ is an *m-regional matching* (for some integer $m \geq 1$) if for all v, $u \in V$ such that $dist(u, v) \leq m$, $Write(v) \cap Read(u) \neq \emptyset$.

For any m-regional matching $\mathscr{RW}$, define the following four parameters:

$$Deg_{write}(\mathscr{RW}) = \max_{v \in V} |Write(v)|$$

$$Rad_{write}(\mathscr{RW}) = \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u, v) \mid u \in Write(v)\}$$

$$Deg_{read}(\mathscr{RW}) = \max_{v \in V} |Read(v)|$$

$$Rad_{read}(\mathscr{RW}) = \frac{1}{m} \cdot \max_{u,v \in V} \{dist(u, v) \mid u \in Read(v)\}.$$

Our construction of regional matchings is based on covers, and makes use of the following lemma of Awerbuch and Peleg [1990a; 1990c].

LEMMA 4.1.1 [AWERBUCH AND PELEG 1990a; 1990c]. *Given a graph $G = (V, E)$, $|V| = n$, a cover $\mathscr{S}$ and an integer $k \geq 1$, it is possible to construct a coarsening cover $\mathscr{T}$ that satisfies the following properties*:

(1) $\hat{R}(\mathscr{T}) \leq (2k - 1)\hat{R}(\mathscr{S})$, *and*
(2) $\Delta(\mathscr{T}) \leq 2k|\mathscr{S}|^{1/k}$.

Using the lemma, we get the following result:

THEOREM 4.1.2. *For all m, $k \geq 1$, it is possible to construct an m-regional matching $\mathscr{RW}_{m,k}$ with*

$$Deg_{read}(\mathscr{RW}_{m,k}) \leq 2k \cdot n^{1/k}$$

$$Deg_{write}(\mathscr{RW}_{m,k}) = 1$$

$$Rad_{read}(\mathscr{RW}_{m,k}) \leq 2k - 1$$

$$Rad_{write}(\mathscr{RW}_{m,k}) \leq 2k - 1.$$

PROOF. Given the graph G and integers k, $m \geq 1$, construct the regional matching $\mathscr{RW}_{m,k}$ as follows. Start by setting $\mathscr{S} = \mathscr{N}_m(V)$, the m-neighborhood cover of the graph, and constructing a coarsening cover $\mathscr{T}$ for $\mathscr{S}$ as in Lemma 4.1. Next, select a center $l(T)$ (in the graph-theoretic sense) in each cluster $T \in \mathscr{T}$. Since $\mathscr{T}$ coarsens $\mathscr{N}_m(V)$, for every vertex $v \in V$ there is a cluster $T \in \mathscr{T}$ such that $N_m(v) \subseteq T$. Consequently, select for every vertex v one such cluster T_v (breaking ties arbitrarily) and set

$$Write(v) = \{l(T_v)\}$$

and

$$Read(v) = \{l(T) \mid v \in T\}.$$

We need to argue that the defined collection of sets constitutes an m-regional matching. Suppose that $dist(u, v) \leq m$ for some processors u, v. Consider the cluster T_v such that $Write(v) = \{l(T_v)\}$. By definition, this cluster satisfies $N_m(v) \subseteq T_v$. Since $dist(u, v) \leq m$, necessarily $u \in N_m(v)$, hence also $u \in T_v$. Therefore, by definition of $Read(u)$, $l(T_v) \in Read(u)$. It follows that $Read(u) \cap Write(v) \neq \emptyset$, as required.

The bounds on Rad_{write}, Deg_{write}, Rad_{read}, and Deg_{read} follow directly from Lemma 4.1.1 (noting that $|\mathscr{S}| = |\mathscr{N}_m(V)| = n$). □

In what follows, we use a hierarchy of 2^i-regional matchings in order to design our regional directories, and show that the complexities of the MOVE and FIND operations in these directories depend on the above parameters of the matchings.

4.2 THE CONSTRUCTION OF REGIONAL DIRECTORIES

4.2.1 *Concepts.* Our constructions are based on hierarchically organizing the tracking information in *regional directories*. A regional directory is based on defining a "regional address" $R\_addr(\xi)$ for every user ξ. The regional address $R\_addr(\xi)$ is simply the name of a vertex w where the user is currently expected to be.

In the hierarchical context, to be discussed in Section 5, the regional addresses represent the most updated *local* knowledge regarding the whereabouts of the user. In particular, the regional address $R\_addr(\xi)$ may be outdated, as ξ may have moved in the meantime to a new location without bothering to update the regional directory.

The basic tasks for which we use the regional directory are similar to those of a regular (global) directory, namely, the retrieval of the regional address, and its change whenever needed. Nevertheless, it is important to note that the semantics of the R_FIND operation is different from that of the FIND operation given in Section 2.4 (in which a message is sent to the user itself). Also, for technical reasons, the modification tasks are easier to represent in the form of "insert" and "delete" operations, rather than the more natural "move" operation. Thus, an *m-regional* directory $\mathscr{RD}$ supports the operations R_FIND($\mathscr{RD}$,

ξ, v), R_DEL($\mathscr{RD}$, ξ, s) and R_INS($\mathscr{RD}$, ξ, t). However, let us stress that the normal operation of the directory will always be based on "moves," namely, pairs of "delete" immediately followed by "insert," so at any given moment (with the exception of transitory states), the user ξ has exactly one valid regional address.

The basic operations of the regional directory are defined as follows:

R_DEL($\mathscr{RD}$, ξ, s). Invoked at the vertex $s = R\_addr(\xi)$, this operation nullifies the current regional address of ξ.

R_INS($\mathscr{RD}$, ξ, t). Invoked at the location t, this operation sets the regional address of ξ, $R\_addr(\xi)$, to be t.

R_FIND($\mathscr{RD}$, ξ, v). Invoked at the vertex v, this operation returns (to vertex v) the regional address $R\_addr(\xi)$ of the user ξ. This operation is guaranteed to succeed only if $dist(v, R\_addr(\xi)) \leq m$. Otherwise, the search may end in *failure*, that is, it may be that the user's regional address is not found. If that happens, then an appropriate message is returned to v.

4.2.2 *Implementation.* The regional address $s = R\_addr(\xi)$ is maintained in the regional directory by means of pointers to s, stored at some vertices in the network. Every vertex u in the network has a variable $\texttt{R\_addr}^u(\xi)$ (for each user ξ in the system). At any given moment, if the regional address of ξ is $R\_addr(\xi) = s$, then the variable $\texttt{R\_addr}^u(\xi)$ at each vertex u is set to either s or *nil*.

The construction of an m-regional directory is based on an m-regional matching $\mathscr{RW}$. The basic idea is the following:

Regional Address Representation. Suppose that at some given moment, the regional address of the user ξ is $s = R\_addr(\xi)$. This fact is represented in the regional directory as follows:

— At each vertex u in the write set $Write(s)$, $\texttt{R\_addr}^u(\xi) = s$.
— At every other vertex w in the network, $\texttt{R\_addr}^w(\xi) = nil$.

Implementation of R_FIND($\mathscr{RD}$, ξ, v). In order to implement the R_FIND operation, the searcher v successively queries the vertices in its read set, $Read(v)$, until hitting a vertex u whose pointer $\texttt{R\_addr}^u(\xi)$ is set to some value $s \neq nil$. This s is the current regional address of the user ξ. In case none of the vertices in $Read(v)$ has the desired pointer (namely, all of them have *nil* pointers), the operation is said to end in failure. Note that by definition of an m-regional matching, this might happen only if $dist(v, s) > m$ for ξ's current regional address $s = R\_addr(\xi)$.

Implementation of R_DEL($\mathscr{RD}$, ξ, s). Operation R_DEL, invoked at the vertex $s = R\_addr(\xi)$, consists of deleting the pointers $\texttt{R\_addr}^u(\xi)$ pointing to s at all the vertices $u \in Write(s)$ (i.e., setting them to *nil*).

Implementation of R_INS($\mathscr{RD}$, ξ, t). Similarly, operation R_INS, invoked at the vertex t, consist of setting the variables $\texttt{R\_addr}^u(\xi)$ to point to t at all

```
R_FIND($\mathcal{RD}, \xi, v$):                          /* invoked at a vertex $v$ */
    For all $u \in Read(v)$ do:
        id ← remote-read R_addr$^u(\xi)$
        If id $\neq$ nil then Return(id)
    End-for
    Return("failure")

R_DEL($\mathcal{RD}, \xi, s$):              /* invoked at the vertex $s = R\_addr(\xi)$ */
    For all $u \in Write(s)$ do:
        R_addr$^u(\xi)$ ← remote-write nil
    End-for

R_INS($\mathcal{RD}, \xi, t$):                          /* invoked at a vertex $t$ */
    For all $u \in Write(t)$ do:
        R_addr$^u(\xi)$ ← remote-write "$t$"
    End-for
```

FIG. 3. The three operations of the m-regional directory $\mathcal{RD}$, based on an m-regional matching $\mathcal{RW}$.

the vertices $u \in Write(t)$, thus effectively setting the regional address $R\_addr(\xi)$ to be t.

As discussed earlier, the two operations will always be performed together, so ξ cannot end up having more than one address in the regional directory.

A formal presentation of operations R_FIND, R_DEL, and R_INS is given in Figure 3.

4.3 ANALYSIS. Let us first verify the correctness of the above implementation for an m-regional directory.

LEMMA 4.3.1. *If $\mathcal{RW}$ is an m-regional matching, then the three procedures described above (Figure 3) correctly implement the operations* R_FIND, R_DEL, *and* R_INS *of an m-regional directory.*

PROOF. The correctness of the R_DEL and R_INS procedures is immediate from the definition of the representation of a regional address in the directory.

The correctness of the R_FIND procedure can be verified in a straightforward manner from the definition of m-regional matchings. In particular, suppose a vertex v issues an R_FIND($\mathcal{RD}$, ξ, v) operation, where currently $R\_add(\xi) = s$ and $dist(s, v) \leq m$. Then the properties of the regional matching $\mathcal{RW}$ guarantee that there is some vertex w in the intersection of $Read(v)$ and $Write(s)$. This w stores a pointer R_addr$^w(\xi) = s$, and therefore v's search will end successfully. □

LEMMA 4.3.2. *The implementation of the regional operations in Figure* 3 *has the following complexities*:

(1) $Cost(\text{R\_FIND}(\mathcal{RD}, \xi, v)) \leq 2m \cdot Deg_{read}(\mathcal{RW}) \cdot Rad_{read}(\mathcal{RW})$,
(2) $Cost(\text{R\_DEL}(\mathcal{RD}, \xi, s)) \leq 2m \cdot Deg_{write}(\mathcal{RW}) \cdot Rad_{write}(\mathcal{RW})$,
(3) $Cost(\text{R\_INS}(\mathcal{RD}, \xi, t)) \leq 2m \cdot Deg_{write}(\mathcal{RW}) \cdot Rad_{write}(\mathcal{RW})$.

PROOF. To prove (1), observe that each R_FIND operation is answered after at most $Deg_{read}(\mathcal{RW})$ searches, each involving sending a query and getting a reply along a path of length at most $m \cdot Rad_{read}(\mathcal{RW})$.

Let us now turn to (2). Note that the operation R_DEL$(\mathcal{RD}, \xi, s)$ involves deleting pointers to s at all the vertices of $Write(s)$. These deletions require sending an appropriate **remote-write** message from s to all vertices in $Write(s)$. The number of such messages is at most $Deg_{write}(\mathcal{RW})$, and each of them traverses a path of length at most $m \cdot Rad_{write}(\mathcal{RW})$ in both directions. A similar argument applies to (3). □

5. *Hierarchical Directory Servers*

In this section we define our *hierarchical directory server* $\mathcal{D}$, and analyze its properties and complexity.

5.1 THE CONSTRUCTION

5.1.1 *Concepts*. The hierarchical directory server $\mathcal{D}$ is defined as follows. For every $1 \leq i \leq \delta$, we construct a 2^i-regional directory $\mathcal{RD}_i$. The user ξ is tracked by each regional directory separately, that is, it has a regional address $R\_addr_i(\xi)$ stored for it in each $\mathcal{RD}_i$. We denote the tuple of regional addresses of the user ξ by

$$\bar{A}(\xi) = \langle R\_addr_1(\xi), \ldots, R\_addr_\delta(\xi) \rangle.$$

As discussed earlier, the regional address $v = R\_addr_i(\xi)$ (stored at the regional directory of level i) 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, $Host_{j_i}$, for some $j_i < L$. The only regional address that is guaranteed to be identical to the true current address of ξ is its lowest level regional address, that is, $R\_addr_1(\xi) = Host_L = Addr(\xi)$. (As a rule, the lower the level, the more up-to-date is the regional address, i.e., $j_{i-1} \geq j_i$.)

This situation implies that finding a regional address of the user ξ alone is not sufficient for locating the user itself. This potential problem is rectified by maintaining at each regional address $h = Host_{j_i} = R\_addr_i(\xi)$ a *forwarding pointer* $\texttt{Forward}^h(\xi)$ pointing at some more recent address of ξ, namely, some $Host_{j'}$, where $j' > j_i$. (It should be clear that the user may in the meantime have moved further, and is no longer at the vertex $Host_{j'}$, but in this case, $Host_{j'}$ will itself have a forwarding pointer for ξ.)

The invariant maintained by the hierarchical directory server regarding the relationships between the regional addresses stored at the various levels and the forwarding pointers is expressed by the following definition:

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, the vertex

$h = Host_{j_i} = R\_addr_i(\xi)$ stores a pointer $\mathtt{Forward}^h(\xi)$ pointing to the vertex $Host_{j_{i-1}} = R\_addr_{i-1}(\xi)$, unless $R\_addr_i(\xi) = Addr(\xi)$.

Thus, the reachability invariant essentially implies that anyone starting at some regional address $R\_addr_i(\xi)$ and attempting to trace the user along the forwarding pointers will indeed reach the current location of ξ, $Addr(\xi)$.

The remaining problem that should concern us involves bounding the length of the resulting forwarding chains. Let $Migrate(\xi)$ denote the actual *migration path* traversed by ξ in its migration from its place of origin $Host_1$ to its current location, $Host_L = Addr(\xi)$. For $1 \le j_1 \le j_2 \le L$, let $Migrate(\xi, j_1, j_2)$ denote the segment of the migration path from $Host_{j_1}$ to $Host_{j_2}$.

Let us further associate with each *regional address* $Host_{j_i} = R\_addr_i(\xi)$ a subpath of $Migrate(\xi)$ denoted $Migrate_i(\xi)$, which is the subpath from $Host_{j_i}$ to the current location, that is, $Migrate_i(\xi) = Migrate(\xi, j_i, L)$.

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 address $R\_addr_i(\xi)$ satisfies the *proximity invariant* if the distance traveled by ξ since the last time the value of $R\_addr_i(\xi)$ was updated in $\mathscr{RD}_i$ satisfies

$$|Migrate_i(\xi)| \le 2^{i-1} - 1.$$

So far, we have not specified the precise value of the regional addresses of the user at any given moment, except for requiring them to obey the proximity invariant. Let us now fix our policy by formally defining the regional addresses associated with ξ.

Regional Address Assignment. The regional addresses of the user ξ are defined by induction on the length of the visiting sequence $\overline{H}(\xi)$, as follows:

—Initially (i.e., at the point of origin, $Host_1$), all regional addresses are set to $Host_1$.

—Let $R\_addr_i(\xi) = Host_{j_i}$ for $1 \le i \le \delta$, and suppose that the user is currently moving from $Host_L$ to a new location $Addr(\xi) = Host_{L+1}$, thus increasing its migration path. This may cause some of the regional addresses to violate the proximity invariant. Let J be the highest index for which the current value of $R\_addr_J(\xi)$ causes such violation, namely:

$$|Migrate_J(\xi)| = |Migrate(\xi, j_J, L+1)| > 2^{J-1} - 1$$

and

$$|Migrate_i(\xi)| = |Migrate(\xi, j_i, L+1)| \le 2^{i-1} - 1 \text{ for every } i > J.$$

Then the regional addresses of ξ at all levels 1 through J are redefined to be the user's current address, $Host_{L+1}$. The remaining regional addresses are left unchanged.

5.1.2 *Implementation.* For every $1 \le i \le \delta$, the 2^i-regional directory $\mathscr{RD}_i$ is based on a 2^i-regional matching as described in the previous subsection.

Further, the collection of 2^i-regional matchings used for these regional directories is constructed so that all of them have the same bounds on their Rad_{read}, Deg_{read}, Rad_{write}, and Deg_{write} values, namely, bounds that are independent of the distance parameter 2^i (the construction described earlier enjoys this independence property).

Each processor v participates in each of the 2^i-regional directories $\mathscr{RD}_i$, for $1 \le i \le \delta$. In particular, this means the following: Each vertex v has sets $Write_i(v)$ and $Read_i(v)$ in each $\mathscr{RD}_i$. Also, each vertex v has a pointer variable $\texttt{R\_addr}_i^v(\xi)$ for each $\mathscr{RD}_i$, and this variable is set to $s = R\_addr_i(\xi)$ at the vertices $v \in Write(s)$ and to *nil* elsewhere.

In addition, the user itself carries along with it a "mobile data structure" providing a complete picture of its tuple of regional addresses. That is, the representation of the user includes a structure

$$\texttt{MDS}^\xi = \langle \overline{\texttt{A}}^\xi, \overline{\texttt{C}}^\xi, J^\xi \rangle.$$

The first component is a tuple of variables

$$\overline{\texttt{A}}^\xi = \langle \texttt{R\_addr}_1^\xi, \ldots, \texttt{R\_addr}_\delta^\xi \rangle.$$

As the regional addresses are set by the user itself, it always knows their value, and therefore $\texttt{R\_addr}_i^\xi = R\_addr_i(\xi)$ for every i.

The second component in the mobile data structure is the tuple of *migration counters*

$$\overline{\texttt{C}}^\xi = \langle \texttt{C}_1^\xi, \ldots, \texttt{C}_\delta^\xi \rangle,$$

used in order to guarantee the proximity invariant. Each counter $\texttt{C}_i^\xi$ counts the distance traveled by ξ since the last time the regional $R\_addr_i(\xi)$ was updated in $\mathscr{RD}_i$, that is,

$$\texttt{C}_i^\xi = |Migrate_i(\xi)|.$$

These counters are used in order to decide which regional addresses need to be updated after each move of the user.

The third component, J^ξ, is a temporary variable used in the process of updating the directory after a move.

5.2 THE PROCEDURES. A FIND($\mathscr{D}$, ξ, v) instruction is performed in two stages: the *regional address retrieval* stage and the *tracing* stage. In the first stage, the querying vertex v successively issues instructions R_FIND($\mathscr{RD}_i$, ξ, v) in the regional directories $\mathscr{RD}_1$, $\mathscr{RD}_2$ etc., until it reaches the first level i on which it succeeds. (There must be such a level, since the highest level always succeeds.)

At this point, the searcher v switches to the second stage, and starts tracing the user through the network, starting from the vertex $R\_addr_i(\xi)$, and moving along forwarding pointers. This tracing process eventually leads to the real address of the user, $Addr(\xi)$, whereby (upon finding the user ξ at that vertex) the search terminates. The procedure FIND($\mathscr{D}$, ξ, v) is formally described in Figure 4.

A MOVE($\mathscr{D}$, ξ, s, t) operation is carried out as follows: All migration counters $\texttt{C}_i^\xi$ are increased by $dist(s, t)$. Let $\texttt{C}_J^\xi$ be the highest level counter that reaches or exceeds its upper limit $(2^{J-1} - 1)$ as a result of this increase. Then,

```
$i \leftarrow 0$                                        /* level in the hierarchy */
id $\leftarrow$ nil                                     /* potential pointer to $R\_addr_i(\xi)$ */
Repeat                  /* retrieve lowest possible regional address $R\_addr_i(\xi)$ */
    $i \leftarrow i + 1$
    id $\leftarrow$ R_FIND($\mathcal{RD}_i, \xi, v$)
Until id $\neq$ nil                                     /* id points to $R\_addr_i(\xi)$ */

transfer-control-to vertex id                          /* move to the vertex $R\_addr_i(\xi)$ */
Repeat                                                  /* trace $\xi$ along forwarding pointers */
    transfer-control-to vertex Forward$^{\otimes}(\xi)$
Until the user $\xi$ is found locally                   /* the vertex $Addr(\xi)$ is reached */
```

FIG. 4. *Procedure* FIND($\mathscr{D}$, ξ, v), *invoked at the vertex* v.

we elect to update the regional directories at levels 1 through J. This involves first eliminating outdated information, by erasing the old listing of ξ in these directories using procedure R_DEL and erasing the forwarding pointers to ξ from the old regional addresses at levels 1 through J. (Note that these pointers are still valid, but going on using them will gradually deteriorate performance, as the tracing process will have to follow longer and longer paths.) Next, we add the updated information, by inserting the appropriate new listing to ξ in these directories (pointing at t as the new regional address) using R_INS. It is also necessary to leave an appropriate forwarding pointer at the vertex $R\_addr_{J+1}(\xi)$ leading to the new location t, and of course perform the actual relocation of the user (along with its mobile data structure $\mathtt{MDS}^{\xi}$). The update procedure MOVE($\mathscr{D}$, ξ, s, t) is described in Figure 5.

5.3 AN EXAMPLE. Let us now consider an example case, illustrating the data structures held in the system and the way they are manipulated. The example concerns a searcher v, and a user ξ. The sequence of vertices visited by the user is $\overline{H} = (h_1, h_2, h_3, h_4)$. Namely, the user has initially resided at h_1, then migrated to h_2, then to h_3, and finally to h_4, which is its current address, $Addr(\xi) = h_4$. The initial situation is depicted in Figure 6, including ξ's migration path, $Migrate(\xi) = p_3 \cdot p_2 \cdot p_1$, the sets $Write_i(\xi)$ and forwarding pointers, and the sets $Read_i(v)$.

Let us first consider a FIND(ξ, v) request issued by v. Then v will fail to retrieve a pointer for ξ in its queries to the regional directories $\mathscr{RD}_i$ for $i = 1$, 2, 3, 4, since $Read_1(v)$, $Read_2(v)$, $Read_3(v)$, and $Read_4(v)$ do not intersect $Write_1(h_4)$, $Write_2(h_4)$, $Write_3(h_3)$, and $Write_4(h_2)$, respectively. However, it will retrieve the pointer $\mathtt{R\_addr}^z(\xi) = h_2$ stored at the vertex z, since this vertex belongs to $Read_5(v) \cap Write_5(h_2)$. The search will now proceed from h_2 along the forwarding pointers to h_3 and from there to h_4.

Now let us consider move operations. The regional addresses of the user ξ are as illustrated in the figure, and the migration subpaths corresponding to the regional addresses are $Migrate_1(\xi) = Migrate_2(\xi) = \emptyset$, $Migrate_3(\xi) = p_1$, $Migrate_4(\xi) = Migrate_5(\xi) = p_2 \cdot p_1$, and $Migrate_6(\xi) = p_3 \cdot p_2 \cdot p_1$, where the

```
For $1 \le i \le \delta$ do:                                    /* updating migration counters */
    $\mathtt{C}_i^\xi \leftarrow \mathtt{C}_i^\xi + dist(s,t)$
End-for
$J^\xi \leftarrow \max\{i \mid \mathtt{C}_i^\xi \ge 2^{i-1}\}$                          /* highest level to be updated */
$\mathtt{Forward}^u(\xi) \leftarrow$ remote-write "$t$", for vertex $u = \mathtt{R\_addr}_{J^\xi+1}^\xi$
                                                     /* forwarding pointer to new location */
For $1 \le i \le J^\xi$ do:
    transfer-control-to vertex $\mathtt{R\_addr}_i^\xi$
    $\mathtt{Forward}^\otimes(\xi) \leftarrow nil$
    R_DEL$(\mathcal{RD}_i, \xi, \otimes)$                     /* erasing pointers to old location at $\mathcal{RD}_i$ */
End-for
Relocate user $\xi$ to vertex $t$, along with its mobile data structure $\mathtt{MDS}^\xi$
For $1 \le i \le J^\xi$ do:
    $\mathtt{R\_addr}_i^\xi \leftarrow t$                                      /* updating address tuple */
    $\mathtt{C}_i^\xi \leftarrow 0$
    R_INS$(\mathcal{RD}_i, \xi, t)$                      /* adding pointers to new location at $\mathcal{RD}_i$ */
End-for
```

FIG. 5. *Procedure* MOVE($\mathscr{D}$, ξ, s, t), *invoked at the vertex* s = Addr(ξ).

lengths of the segments are $|p_1| = 3$, $|p_2| = 2$, and $|p_3| = 20$. Hence, the current values of the tuples of regional addresses and migration counters are

$$\bar{A}(\xi) = \langle h_4, h_4, h_3, h_2, h_2, h_1 \rangle, \qquad \bar{\mathtt{C}}^\xi = \langle 0, 0, 3, 5, 5, 25 \rangle.$$

(Note that the counters satisfy the proximity invariant.)

Now suppose that the user ξ performs three move operations as follows:

$$M_1 = \text{MOVE}(\xi, h_4, h_5), \qquad M_2 = \text{MOVE}(\xi, h_5, h_6),$$
$$M_3 = \text{MOVE}(\xi, h_6, h_7),$$

where $dist(h_4, h_5) = dist(h_5, h_6) = dist(h_6, h_7) = 1$.

Then, the data structures of the directory change as follows: In the first move M_1, the counter $\mathtt{C}_3^\xi$ is increased to 4, thus exceeding its allowed upper limit. This requires updates to the regional directories $\mathscr{RD}_i$ for $1 \le i \le 3$, which now all point to h_5. More specifically, the pointers $\mathtt{R\_addr}_1^u(\xi)$ and $\mathtt{R\_addr}_2^u(\xi)$ are set to *nil* at all vertices $u \in Write_1(h_4) \cup Write_2(h_4)$ (where h_4 is the previous regional address at levels 1 and 2), similarly the pointers $\mathtt{R\_addr}_3^u(\xi)$ are set to *nil* at all vertices $u \in Write_3(h_3)$, and the pointers $\mathtt{R\_addr}_i^u(\xi)$, $i = 1, 2, 3$, are set to h_5 at all vertices $u \in Write_1(h_5) \cup Write_2(h_5) \cup Write_3(h_5)$. Also, the forwarding pointer at h_3 is erased (since vertex h_3 ceases to play any role in the directory with respect to the user ξ), and the forwarding pointer at h_2 is now directed at h_5. The resulting address and counter tuples of ξ are now

$$\bar{A}(\xi) = \langle h_5, h_5, h_5, h_2, h_2, h_1 \rangle, \qquad \bar{\mathtt{C}}^\xi = \langle 0, 0, 0, 6, 6, 26 \rangle.$$

In the second move M_2, the only counter that exceeds its upper limit is $\mathtt{C}_1^\xi$. Therefore only $\mathscr{RD}_1$ is updated to lead to h_6, and a new forwarding pointer is

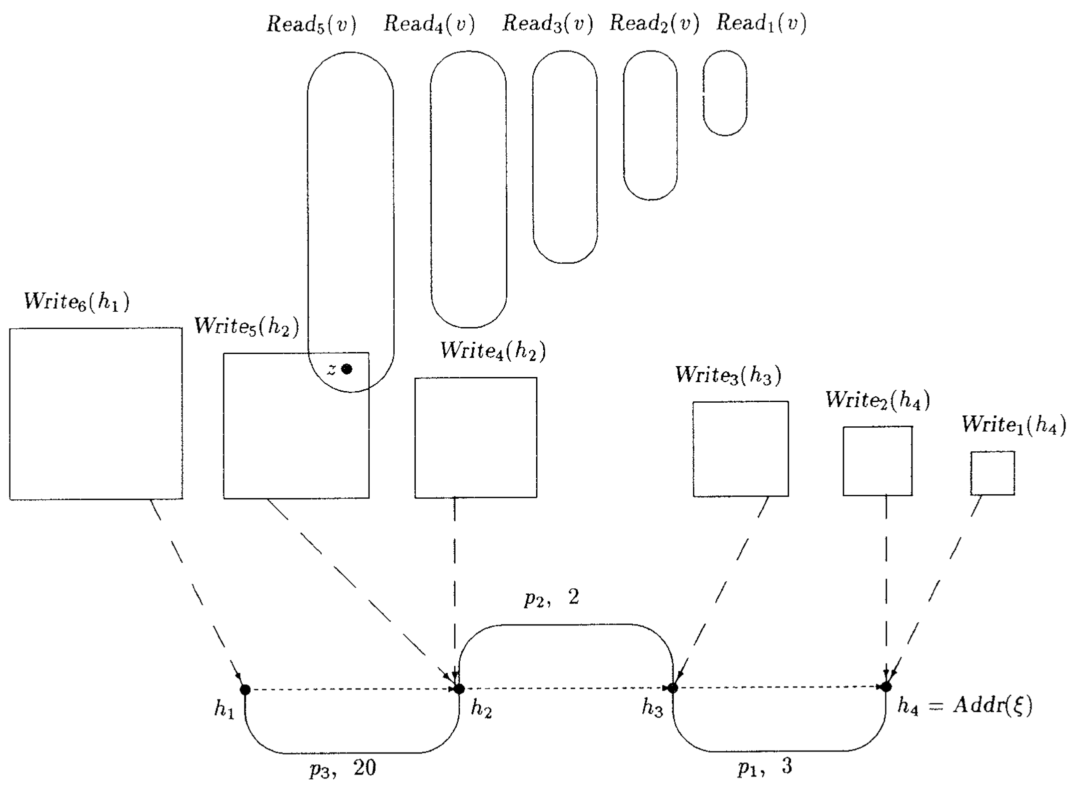

FIG. 6. Data structures involved in the example. The solid winding line represents the migration path of the user ξ, composed of three segments p_3, p_2, p_1. The number listed next to each segment represents its length. The dotted arrows represent the forwarding pointers leading to ξ's current residence. The dashed arrows represent the pointers to regional addresses, stored at the appropriate *Write* sets.

added at h_5, directed at h_6. The resulting address and counter tuples of ξ are now

$$\bar{A}(\xi) = \langle h_6, h_5, h_5, h_2, h_2, h_1 \rangle, \qquad \bar{c}^{\xi} = \langle 0, 1, 1, 7, 7, 27 \rangle.$$

The third move M_3 causes c_4^{ξ} to overflow. This results in updates to the regional directories $\mathscr{RD}_i$ for $1 \le i \le 4$, which now all point to h_7. The forwarding pointer at h_2 is now directed at h_7. The resulting address and counter tuples of ξ are now

$$\bar{A}(\xi) = \langle h_7, h_7, h_7, h_7, h_2, h_1 \rangle, \qquad \bar{c}^{\xi} = \langle 0, 0, 0, 0, 8, 28 \rangle. \qquad \square$$

5.4 Analysis

LEMMA 5.4.1. *Procedure* MOVE *maintains the reachability and proximity invariants.*

PROOF. By direct inspection on the algorithm. Every time a migration counter c_i^{ξ} exceeds its allowed upper bound, it is reduced to zero, updating the corresponding regional address; and every time a regional address $R\_addr_i(\xi)$ is changed, the forwarding pointer at the vertex $R\_addr_{i+1}(\xi)$ is updated appropriately. $\square$

Let us now consider the execution of some FIND(ξ, v) operation. Suppose the procedure has completed its first stage, and succeeded in retrieving some regional address $R\_addr_i(\xi)$. The second stage of the search now starts a tracing process, proceeding along a path starting at the regional address $u_i = R\_addr_i(\xi)$ and tracing the forwarding pointers $\texttt{Forward}^{u_j}(\xi)$. Let us denote the path taken by the tracing process by $Trace_i(\xi)$.

LEMMA 5.4.2. *For every $1 \le i \le \delta$ and every user ξ, at any time, the path $Trace_i(\xi)$ has the following properties*:

(1) *$Trace_i(\xi)$ leads to $Addr(\xi)$.*
(2) *$|Trace_i(\xi)| \le |Migrate_i(\xi)| \le 2^{i-1} - 1$.*
(3) *$dist(R\_addr_i(\xi), Addr(\xi)) \le 2^{i-1} - 1$.*

PROOF. The first property follows from the reachability invariant, guaranteed by Lemma 5.4.1, as discussed earlier.

For the second property, note that the reachability invariant implies also that $Trace_i(\xi)$ is the concatenation of segments

$$Trace_i(\xi) = q_i \cdots q_2,$$

where q_j is the path used by the system for message routing from the vertex $R\_addr_j(\xi)$ to $R\_addr_{j-1}(\xi)$, for $2 \le j \le i$. The vertices $R\_addr_j(\xi)$ determining the trajectory of the path $Trace_i(\xi)$ all occur on the migration subpath $Migrate_i(\xi)$. By our assumption on the routing system, each segment q_j of $Trace_i(\xi)$ is a shortest path between its endpoints, that is, is of length $dist(R\_addr_j(\xi), R\_addr_{j-1}(\xi))$. Hence, the length of each segment q_j can only be smaller than or equal to the corresponding segment on $Migrate_i(\xi)$, and therefore $|Trace_i(\xi)| \le |Migrate_i(\xi)|$. The second inequality follows from the proximity invariant, guaranteed by Lemma 5.4.1.

The third property also follows from the proximity invariant, by the triangle inequality, noting that $R\_addr_i(\xi)$ and $Addr(\xi)$ are the endpoints of the path $Migrate_i(\xi)$. □

LEMMA 5.4.3. *The hierarchical directory server $\mathcal{D}$ satisfies*

$$Stretch_{find} = O(Deg_{read} \cdot Rad_{read}).$$

PROOF. Consider a sequence $\bar{\sigma}$ of operations, with a subsequence $\mathcal{F}(\bar{\sigma})$ of FIND operations. We shall actually prove a stronger claim than the lemma, namely, we will upper bound the worst-case stretch $Cost(F)/Opt\_cost(F)$ of any single FIND operation $F \in \mathcal{F}(\bar{\sigma})$, rather than just the amortized stretch $Stretch_{find}$. Clearly, this worst-case stretch upper-bounds the average stretch, that is,

$$Stretch_{find}(\bar{\sigma}) = \frac{\sum_{F \in \mathcal{F}(\bar{\sigma})} Cost(F)}{\sum_{F \in \mathcal{F}(\bar{\sigma})} Opt\_cost(F)} \le \max_{F \in \mathcal{F}(\bar{\sigma})} \frac{Cost(F)}{Opt\_cost(F)}.$$

Suppose that a processor v issues an instruction $F = \text{FIND}(\mathcal{D}, \xi, v)$ for some user ξ in the network. Recall that $Opt\_cost(F) = dist(v, Addr(\xi))$. Let $\beta = \lceil \log dist(v, Addr(\xi)) \rceil$, that is,

$$2^{\beta-1} < dist(v, Addr(\xi)) \le 2^{\beta}. \tag{1}$$

The FIND procedure operates as follows: The querying vertex v successively executes find operations $RF_i = \text{R\_FIND}(\mathcal{RD}_i, \xi, v)$ in the regional directory $\mathcal{RD}_i$, for $i = 1, 2, \ldots,$ until it reaches the first level $i_0 \le \delta$ on which it succeeds in fetching the identity of the vertex $R\_addr_{i_0}(\xi)$.

Recall that by the definition of a regional directory, success on level i is guaranteed whenever $dist(v, R\_addr_i(\xi)) \le 2^i$. It is thus necessary to bound these distances. Clearly, for every i

$$dist(v, R\_addr_i(\xi)) \le 2^{\delta}. \tag{2}$$

Also, by (1) and Property (3) of Lemma 5.4.2 combined with the triangle inequality, we have

$$dist(v, R\_addr_i(\xi)) \le dist(v, Addr(\xi)) \\ + dist(Addr(\xi), R\_addr_i(\xi)) < 2^{\beta} + 2^{i-1}. \tag{3}$$

It follows from (2) and (3) that for $i' = \min\{\beta + 1, \delta\}$, the regional directory $\mathcal{RD}_{i'}$ must succeed in performing the operation $RF_{i'} = \text{R\_FIND}(\mathcal{RD}_{i'}, \xi, v)$. Hence, the *first* level i_0 on which RF_{i_0} succeeds satisfies

$$i_0 \le \beta + 1. \tag{4}$$

After fetching the regional address $R\_addr_{i_0}(\xi)$, the FIND procedure starts its second stage, proceeding as follows. The searcher v starts tracing the user through the network, starting from the vertex $R\_addr_{i_0}(\xi)$, and moving along the path $Trace_{i_0}(\xi)$, as directed by the forwarding pointers. By Property (1) of Lemma 5.4.2, this tracing process leads to the vertex $Addr(\xi)$, and by Property (2) of the same lemma and (4), the path traversed satisfies

$$|Trace_{i_0}(\xi)| \le 2^{i_0-1} \le 2^{\beta}.$$

Therefore, the actual communication cost $Cost(F)$ for the entire search operation $F = \text{FIND}(\mathscr{D}, \xi, v)$, (which equals the total length of the combined path traversed by the search messages), satisfies by Lemma 4.3.2

$$\begin{aligned} Cost(F) &\le 2 \cdot 2^{\beta} + \sum_{i=1}^{\beta+1} Cost(RF_i) \\ &\le 2 \cdot 2^{\beta} + \sum_{i=1}^{\beta+1} \left(2 \cdot 2^{i} \cdot Deg_{read} \cdot Rad_{read}\right) \\ &\le \left(\frac{1}{2} + 2 \cdot Deg_{read} \cdot Rad_{read}\right) \cdot 2^{\beta+2} \\ &\le 8\left(\frac{1}{2} + 2 \cdot Deg_{read} \cdot Rad_{read}\right) \cdot dist(v, Addr(\xi)), \end{aligned}$$

establishing

$$\frac{Cost(F)}{Opt\_cost(F)} \le 8\left(\frac{1}{2} + 2 \cdot Deg_{read} \cdot Rad_{read}\right). \qquad \square$$

LEMMA 5.4.4. *The hierarchical directory server* $\mathscr{D}$ *satisfies*

$$Stretch_{move} = O\left(Rad_{write} \cdot Deg_{write} \cdot \delta + \frac{\delta^2}{\log n}\right).$$

PROOF. Consider any sequence $\bar{\sigma}$ of operations, and its subsequence $\mathscr{M}(\bar{\sigma}) = (M_1, \dots, M_k)$ of MOVE operations, where $M_i = (\mathscr{D}, \xi, s^i, s^{i+1})$. Let $\rho_{\mathscr{M}}$ denote the total length of the path taken by ξ since its insertion into the system,

$$\rho_{\mathscr{M}} = \sum_{1 \le i \le k} dist(s^i, s^{i+1}).$$

Observe that

$$Opt\_cost(\mathscr{M}(\bar{\sigma})) = \sum_{1 \le i \le k} Reloc(\xi, s^i, s^{i+1}) \ge \rho_{\mathscr{M}}.$$

To compute the amortized cost, we partition the sequence $\mathscr{M}(\bar{\sigma})$ of move operations into subsequences $\mathscr{M}_1, \dots, \mathscr{M}_\delta$ as follows. Consider the move operation M_i, and let l be the value assigned to the variable J^{ξ} in the execution of Procedure MOVE (i.e., l is the highest index of a migration counter that reached or exceeded its bound in this move). Then M_i is assigned to the subsequence M_l.

Let us now consider a move operation $M = (\mathscr{D}, \xi, s, t)$ in the subsequence $\mathscr{M}_l$ and analyze this cost. Denote $\alpha_i = R\_addr_i(\xi)$ prior to that move, for every $1 \le i \le l + 1$. In this update, the regional address $R\_addr_i(\xi)$ is changed from α_i to t, for every $1 \le i \le l$. The first step taken by Procedure MOVE is sending a message to vertex α_i and performing $\text{R\_DEL}(\mathscr{R}\mathscr{D}_i, \xi, \alpha_i)$ from α_i, for all $i \le l$. Observe that for every i, $dist(s, \alpha_i) \le 2^{i-1}$, which bounds the cost of these messages. By Lemma 4.3.2, the overall communication complexity of $\text{R\_DEL}(\mathscr{R}\mathscr{D}_i, \xi, \alpha_i)$ is at most $2 \cdot 2^{i} \cdot Rad_{write} \cdot Deg_{write}$. Thus the overall cost of the first step is at most

$$\sum_{1 \le i \le l} \left(2^{i-1} + 2^{i+1} \cdot Rad_{write} \cdot Deg_{write}\right) \le 2^{l+2}\left(\frac{1}{4} + Rad_{write} \cdot Deg_{write}\right).$$

Sending a **remote-write** message to vertex $u = \alpha_{l+1}$ (informing it to change the value of the pointer $\mathtt{Forward}^{u}(\xi)$ so that it points to t and not to α_l) costs at most 2^{l+1} by the same argument. Relocating the user ξ to vertex t cost $Reloc(\xi, s, t)$. It is also necessary to move its mobile data structure $\mathtt{MDS}^{\xi}$ containing essentially δ regional addresses $R\_addr_l(\xi)$ of $O(\log n)$ bits each, and δ migration counters $\mathtt{C}_l^{\xi}$ of $O(\delta)$ bits each, totaling an additional $dist(s, t) \cdot O(\delta + \delta^2/\log n) = O(Reloc(\xi, s, t)(\delta + \delta^2/\log n))$ cost. Finally, performing $\text{R\_INS}(\mathcal{RD}_i, \xi, t)$ from t, for all $i \le l$, costs at most $2^{l+2} \cdot Rad_{write} \cdot Deg_{write}$ by Lemma 4.3.2 again.

Overall, an operation $M = (\mathcal{D}, \xi, s, t)$ from subsequence $\mathcal{M}_l$ costs a total of

$$Cost(M) \le O\left(Reloc(\xi, s, t)\left(\delta + \frac{\delta^2}{\log n}\right)\right) + 2^{l+3} \cdot \left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right).$$

Next let us bound the number of move operations in each subsequence $\mathcal{M}_l$. This bound relies on the observation that between any two successive move operations in the subsequence $\mathcal{M}_l$, the migration counter $\mathtt{C}_l^{\xi}$ had to be increased by at least 2^{l-1}. Thus, the number of move operations from $\mathcal{M}(\bar{\sigma})$ that are assigned to $\mathcal{M}_l$ satisfies

$$|\mathcal{M}_l| \le \frac{\rho_{\mathcal{M}}}{2^{l-1}}.$$

The total cost of operations in the subsequence $\mathcal{M}_l$ is therefore bounded by

$$\begin{aligned}
&Cost(\mathcal{M}_l)\\
&\quad = \sum_{M_i \in \mathcal{M}_l} Cost(M_i)\\
&\quad \le \sum_{M_i \in \mathcal{M}_l} O\left(Reloc(\xi, s, t)\left(\delta + \frac{\delta^2}{\log n}\right)\right)\\
&\qquad + |\mathcal{M}_l| \cdot 2^{l+3} \cdot \left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right)\\
&\quad \le Opt\_cost(\mathcal{M}_l) \cdot O\left(\delta + \frac{\delta^2}{\log n}\right) + \frac{\rho_{\mathcal{M}}}{2^{l-1}} \cdot 2^{l+3} \cdot \left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right)\\
&\quad = Opt\_cost(\mathcal{M}_l) \cdot O\left(\delta + \frac{\delta^2}{\log n}\right) + 16 \cdot \rho_{\mathcal{M}} \cdot \left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right).
\end{aligned}$$

Summing over all δ subsequences, we get the following bounds.

$$\begin{aligned}
Cost(\mathcal{M}(\bar{\sigma})) &= \sum_{l=1}^{\delta} Cost(\mathcal{M}_l)\\
&\le Opt\_cost(\mathcal{M}(\bar{\sigma})) \cdot O\left(\delta + \frac{\delta^2}{\log n}\right)\\
&\quad + 16\delta \cdot \rho_{\mathcal{M}} \cdot \left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right).
\end{aligned}$$

Hence

$$\begin{aligned}&Stretch_{move}(\bar{\sigma})\\&= \frac{Cost(\mathcal{M}(\bar{\sigma}))}{Opt\text{-}cost(\mathcal{M}(\bar{\sigma}))}\\&\leq O\left(\delta + \frac{\delta^2}{\log n}\right) + 16\delta\left(\frac{3}{8} + Rad_{write} \cdot Deg_{write}\right). \qquad \square\end{aligned}$$

LEMMA 5.4.5. *The hierarchical directory server constructed as above can be implemented for N users using a total of* $O(N \cdot Deg_{write} \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot Deg_{read} \cdot \delta \cdot \log n)$ *memory bits throughout the network.*

PROOF. The regional directory implementation involves the following space requirements. Each user ξ stores its mobile data structure MDS^{ξ}, whose main components, the $\bar{A}^{\xi}$ and $\bar{C}^{\xi}$ tuples, contain $O(\delta^2 + \delta \log n)$ bits, in its current host. In addition, it posts its address $s = Addr(\xi)$ at the vertices of $Write(s)$. Summing over N users and δ levels, this gives a total memory of $O(N \cdot Deg_{write} \cdot \delta \cdot \log n + N \cdot \delta^2)$. In addition, each processor v needs to know the identity of the vertices in $Read(v)$. This requires a total additional amount of $O(Deg_{read} \cdot n \cdot \delta \cdot \log n)$ bits. $\square$

Summarizing the above three lemmas, we get:

LEMMA 5.4.6. *Given an appropriate family of regional matchings, the hierarchical directory server* $\mathcal{D}$ *constructed as above satisfies* $Stretch_{find} = O(Deg_{read} \cdot Rad_{read})$ *and* $Stretch_{move} = O(Rad_{write} \cdot Deg_{write} \cdot \delta + \delta^2/\log n)$, *and requires a total of* $O(N \cdot Deg_{write} \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot Deg_{read} \cdot \delta \cdot \log n)$ *memory bits throughout the network in order to handle N users.*

Using Lemma 5.4.6 and Theorem 4.1.2, we get

THEOREM 5.4.7. *Using a family of regional matchings* $\mathcal{RW}_{m,k}$ *as in Theorem* 4.1.2, *the hierarchical directory server* $\mathcal{D}$ *constructed as above satisfies* $Stretch_{find} = O(k^2 \cdot n^{1/k})$ *and* $Stretch_{move} = O(\delta \cdot k + \delta^2/\log n)$, *and requires a total of* $O(N \cdot \delta \cdot \log n + N \cdot \delta^2 + n^{1+1/k} \cdot k \cdot \delta \cdot \log n)$ *memory bits throughout the network in order to handle N users.*

Now, consider the hierarchical directory server $\mathcal{D}$ obtained by picking $k = \log n$.

COROLLARY 5.4.8. *The hierarchical directory server* $\mathcal{D}$ *satisfies* $Stretch_{find} = O(\log^2 n)$ *and* $Stretch_{move} = O(\delta \cdot \log n + \delta^2/\log n)$ *and uses a total of* $O(N \cdot \delta \cdot \log n + N \cdot \delta^2 + n \cdot \delta \cdot \log^2 n)$ *memory bits for handling N users.*

6. *Handling Concurrent Accesses*

Our solution, as described so far, completely ignores concurrency issues. It is based on the assumption that the FIND and MOVE requests arrive "sequentially," and are handled one at a time, that is, there is enough time for the system to complete its operation on one request before getting the next one. This assumption would be reasonable if all network communication, as well as all MOVE and FIND operations, were performed in negligible time.

However, in some practical applications, for example, for satellite links, communication suffers a significant latency. Also, the MOVE and FIND operations may in fact take a considerable amount of time. In such cases, concurrency issues can no longer be ignored.

Interesting problems arise when many operations are issued simultaneously. Specifically, problems may occur if someone attempts to contact a user *while* it is moving. It is necessary to ensure that the searcher will eventually be able to reach the moving user, even if that user repeatedly moves. In this section, we outline the particular modifications (both in the model and in the algorithms) needed to handle the case in which operations are performed concurrently and asynchronously.

6.1 MODIFICATIONS IN THE MODEL. We assume the static asynchronous network model, cf. Gallager et al. [1983]. (The case of a dynamically changing network is discussed later on, in Section 7.2.)

In order to facilitate reasoning about concurrent operations, it is necessary to address timing issues more explicitly. The input to the system now consists of a stream of (possibly concurrent) requests to perform MOVE and FIND operations, and the function of the system is to implement these operations. Both MOVE and FIND operations are viewed as occupying some time interval. A FIND(ξ, v) operation starts upon the requesting processor v issuing the request. Its implementation consists of the delivery of a message to the current location of the migrating process, and is terminated at the time of delivery. A MOVE(ξ, s, t) operation again starts upon the requesting user ξ issuing the request at s. Its implementation consists of the actual move of the process, followed by the updating of various data structures, followed by a signal indicating the termination of the operation. 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 vertex t.

The concurrent case poses some complications for our cost definitions. In particular, consider a request $F =$ FIND(ξ, v). It may so happen that while the directory server attempts to satisfy this request, and deliver the search message from v to ξ, the user ξ itself is busy migrating, in some arbitrary direction. In fact, ξ could possibly perform several moves while searched by v. How then should we define the *inherent* cost of the search?

The approach adopted here is the following. First, the correctness requirement of the directory server is that a FIND operation F always terminates successfully within finite time, that is, the "chase" cannot proceed forever, and $T_{end}(F) < \infty$.

The operation F takes place in the time interval $[T_{start}(F), T_{end}(F)]$. Since the user ξ may have moved (perhaps more than once) during this period, we redefine the optimal cost of this operation to be the maximal distance from v to any location occupied by ξ throughout the duration of the operation, namely

$$Opt\_cost(F) = \max_{T_{start}(F) \le \tau \le T_{end}(F)} \{dist(v, Addr^{\tau}(\xi))\}.$$

Despite its seeming permissiveness, this definition is in fact quite reasonable, as can be realized by considering the following simple scenario. Suppose that the user ξ, located at a vertex s neighboring v, starts migrating away from v to some distant location t at time T_0. Suppose further that v starts a search for ξ immediately after that, at time $T_0 + \epsilon$, while the user is still physically at s.

Now, the location at which v will get hold of ξ depends on the relative speeds of the search and the migration processes. In particular, if the migration process is very fast relative to the search process, it could happen that the latter is forced to chase ξ all the way to t. This might happen no matter what control policy is used, unless the system uses the extremely conservative (and expensive) policy of requiring the migrating process to query all processors and verify none of them currently needs it, before it is allowed to start moving.

6.2 OVERVIEW OF ALGORITHMIC MODIFICATIONS. The problems that arise in the concurrent case can be classified into two types, roughly corresponding to the two stages of procedure FIND($\mathscr{D}$, ξ, v), namely, the regional address retrieval stage and the tracing process. The idea is that in order to prevent endless chases, the invariant that we would like to preserve 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. If the user is currently moving *within* the 2^i vicinity of the searcher, then it must be found.

In order to enforce such invariant, it is necessary to make sure that both stages of the FIND procedure succeed. First, the searcher should be able to retrieve a regional address $R\_addr_i(\xi)$ of the user at level i. Secondly, once such an address is retrieved, it should suffice to lead the tracing process to the user within a "short" chase (where "short" is to be understood in accordance with our stretch bounds).

Intuitively, this is imposed by ensuring the following "clean move" requirement: The user ξ is not allowed to finish a new move that involves updating regional directories up to level l, before it is found by any "near-by" searcher that is already at the second stage of the search, that is, that has already retrieved some current regional address $R\_addr_i(\xi)$, for $i \leq l + 1$.

Our algorithms have the same general structure as before, except for three minor modifications, which mostly involve permuting and altering some of the steps in the implementation of MOVE(ξ, s, t).

6.3 FIRST STAGE: OBTAINING A REGIONAL ADDRESS. Let us first consider the problem of preventing unjustified failures of the first part of the FIND operation, performed by the R_FIND procedure, namely, ensuring that a searcher residing in the appropriate vicinity of the user always succeeds in obtaining the user's regional address.

More formally, we need to ensure that the R_FIND operation in a regional directory of level i guarantees the retrieval of the ith level regional address $R\_addr_i(\xi)$ of the user ξ even while this address is being changed, as long as both the new and the old addresses are within the 2^i-neighborhood of v, $N_{2^i}(v)$. (Otherwise, as mentioned earlier, the optimal cost of the FIND operation is considered to be higher than 2^i, and therefore the partial search R_FIND is allowed to fail at level i.)

Such a search might fail, for instance, because according to the current code of the MOVE operation, there are times when the user has *no* valid regional address at all. These are the times exactly in the middle of the MOVE procedure, *after* it has already performed the R_DEL($\mathscr{RD}$, ξ, s) procedure and deleted its old regional address s, and *before* it had the opportunity to insert its new regional address t. If the searcher queries the regional directory $\mathscr{RD}_i$ for ξ's regional address $R\_addr_i(\xi)$ precisely at this unfortunate moment, then the search will fail to retrieve either s or t, even if both locations are in the searcher's close vicinity, thereby violating the definition of the regional directory.

This type of failure can be overcome by a straightforward modification to the MOVE procedure, requiring the user ξ to *first* register in its new address t, *and only then* delete its registration at the old address s. That is, the R_INS($\mathscr{RD}$, ξ, t) procedure is executed before the R_DEL($\mathscr{RD}$, ξ, s) procedure. This ensures that at any given moment, the user has a valid regional address.

Note that this change in the order of operation requires also a small change in the R_DEL procedure itself: if the sets $Write(s)$ and $Write(t)$ intersect at some vertex w, then setting $\text{R\_addr}^w(\xi) = t$ at w by the R_INS procedure in effect takes care of the part of the R_DEL operation local to w as well.

Note also, that as a result of this change, ξ may be temporarily "doubly-registered" at both the new and old addresses. That is, there may be points in time when some vertices register s as the user's regional address, while some other vertices register t as that address. Such double registration cannot hurt the process of locating the user in any way since, as will be explained in the next subsection, the reachability invariant can still be maintained, so even if the tracing process arrives at the old address s, it will subsequently proceed along the forwarding pointers to the new address t.

Unfortunately, due to the intricate nature of concurrency and asynchronous communication, although the above change guarantees that at any given moment in time the user has a valid regional address, it still does not suffice in itself to guarantee that a searcher in the vicinity of both s and t will succeed in retrieving (at least) one of them as $R\_addr(\xi)$.

To see where a problem might arise, consider a vertex v invoking R_FIND($\mathscr{RD}_i$, ξ, v), while $R\_addr_i(\xi)$ is changed from s' to t', such that both $dist(v, s') \le 2^i$ and $dist(v, t') \le 2^i$. The implementation of this operation consists of the **remote-read** of $\text{R\_addr}^u(\xi)$ from u, for all $u \in Read_i(v)$. Even though deleting the pointers $\text{R\_addr}^u(\xi) = s'$ at all $u \in Write_i(s')$ (in suboperation R_DEL($\mathscr{RD}_i$, ξ, s')) is performed *after* adding the pointers to t', $\text{R\_addr}^u(\xi) = t'$, at all $u \in Write_i(t')$ (in suboperation R_INS($\mathscr{RD}_i$, ξ, t')), it is still possible that all of the **remote-read** operations done by v fail to detect a pointer to ξ. This might happen, for example, in the scenario in which the messages carrying the **remote-read** requests from v to vertices $u \in Read_i(v) \cap Write_i(s')$ are very slow, and hence reach each such vertex u only *after* it has already erased its pointer $\text{R\_addr}^u(\xi) = s'$ leading to ξ's old residence at s', while at the same time, the messages carrying the **remote-read** requests from v to vertices $u \in Read_i(v) \cap Write_i(t')$ are very fast, and hence reach each such vertex u *before* it established a pointer $\text{R\_addr}^u(\xi) = t'$ leading to ξ's new residence at t'.

This "atomicity problem" is typical to asynchronous systems and arises, for example, in the "distributed snapshot" problem [Chandy and Lamport 1985]. In our case, there are several ways to go about solving this problem. The one we adopt is based on strengthening the definition of m-regional matchings. Specifically, let us introduce the following additional requirement:

For every v, u_1 and u_2, if $dist(v, u_1) \le m$ and $dist(v, u_2) \le m$, then

$$Read(v) \cap Write(u_1) \cap Write(u_2) \neq \emptyset. \qquad (*)$$

This would eliminate the difficulty outlined above, since when v queries all vertices in $Read_i(v)$, it hits also some vertex $x \in Read_i(v) \cap Write_i(s') \cap Write_i(t')$, and at this vertex, the variable $\texttt{R\_addr}^x(\xi)$ must be set to either s' or t'.

In fact, a stronger requirement that can be imposed on an m-regional matching is the following:

For every v and u, if $dist(v, u) \leq m$, then

$$Read(v) \subseteq Write(u). \qquad (**)$$

This second requirement $(**)$ clearly implies the former requirement $(*)$, and in fact, it simplifies the search process, since under this requirement, *every* vertex in $Read(v)$ that previously pointed at s', is guaranteed to point to either s' or t' at any time during the transition performed by the MOVE procedure.

Finally, a way to impose requirement $(**)$, and at the same time simplify the FIND procedure even further, is by making the following (yet stronger) requirement:

$$\text{All read sets are of cardinality 1.} \qquad (***)$$

Under this requirement, each vertex v queries a single location during an R_FIND operation.

Luckily, this last requirement $(***)$ is easy to achieve. A straightforward way to do that is by switching the construction rules of the read and write sets in the proof of Theorem 4.1.2, that is, switching the sets $Write(v)$ and $Read(v)$ for every v. This does not change the basic properties of the regional matching. However, notice that in that construction, the write sets are of cardinality 1, and therefore the same applies to the read sets in the switched construction.

Consequently, in the sequel (and particularly in the code of Figure 8), we will refer to $Read(v)$ as a single vertex, rather than a set of vertices.

6.4 SECOND STAGE: TRACING THE USER.

Let us next consider the problem of preventing failures in the second stage of the FIND operation, once some regional address has been obtained. Recall that this stage involves proceeding from the obtained regional address to "trace down" the user, and can be thought of as sending a "tracing message" to chase the user, along the forwarding pointers. Our approach toward preventing endless chases is based on guaranteeing that the searcher is not allowed to "miss" the user while searching for it on level i if the user is currently moving *within* the 2^i vicinity of the searcher.

To formalize this property (and in fact, make a stronger requirement), we need the following definition. For a FIND operation F, let us define an intermediate time $T_{mid}(F)$, falling between $T_{start}(F)$ and $T_{end}(F)$, as the time in between the two parts of Procedure FIND, namely, the time by which the searcher has obtained a regional address for the user.

The Clean Move Requirement. Suppose that at time T_0, the user ξ wishes to perform a move M that involves updating regional directories up to level l. Then ξ is not allowed to finish M before it is found by any searcher that is already at the second stage of the search, that is, that has already retrieved some current regional address $R\_addr_i(\xi)$, for $i \leq l + 1$, at time $T_{mid}(F) \leq T_0$.

The main implication of this change is that it helps to prevent unbounded chases, since a FIND operation is guaranteed to retrieve a regional address of the user at some level i, and once it did so, it is guaranteed to reach the user at its current address, since the user is pinned to that location until the FIND procedure terminates.

The "clean move" requirement can be imposed in a standard way, by means of extensive locking. For instance, the R_FIND procedure can start by locking the $\texttt{R\_addr}_i^u(\xi)$ variable it accesses, and if its value is not *nil*, releasing it only after reaching the user. The MOVE procedure will have to get a lock on each variable $\texttt{R\_addr}_i^u(\xi)$ it wants to erase at a vertex u, and will be allowed to erase the forwarding pointers from the current level-i regional address $s = R\_addr_i(\xi)$ to the level $i - 1$ regional address only after erasing the variables $\texttt{R\_addr}_i^u(\xi)$ at all vertices $u \in Write(s)$.

Such a solution will not affect the communication complexity of the procedures, although it might affect the response time. Fairness can be guaranteed by means of queuing locking requests at each vertex, and granting locks according to the order in the queue.

Nevertheless, this solution might still be improved in terms of response time, by reducing the locking periods and thus increasing the degree of actual concurrency. The more detailed solution described below will employ minimal locking.

Our solution imposes the "clean move" requirement by introducing the following changes to our MOVE algorithm. First, the old regional addresses are deleted top-down, that is, starting from the highest level regional directory and ending with the lowest level one. Secondly, along its way, the deletion process also "sweeps" the route and verifies that there are no tracing processes for ξ in transit. A similar change is made in procedure R_DEL($\mathscr{RD}_i$, ξ, s), which is now required to sweep along the routes leading from all vertices in $Write_i(s)$ to s, and make sure there are no tracing processes for ξ in transit.

6.4.1 *The* SWEEP *Procedure.* In order to enable these "sweep" operations, it is required to ensure that "tracing processes" (following an `R_addr` or a `Forward` pointer) progress along unique fixed paths, so that they can be traced themselves. We formalize this requirement as follows:

The Unique Route Requirement. For any two vertices $u, w \in V$, fix a unique (shortest) path $Route(u, w)$ connecting u and w in the network. Now all the messages of Procedure FIND are required to be sent along these paths. This includes the messages sent from one regional address $R\_addr_i(\xi)$ to the regional address $R\_addr_{i-1}(\xi)$ in the level below it, in implementing the "**transfer-control-to**" command, as well as the messages sent from vertices $Write_i(s)$ to s, in implementing the "**remote-read**" commands of Procedure R_FIND.

The sweep operation now becomes trivial if message transition on the links is assumed to obey the first-in, first-out (FIFO) rule. In order to clean the route from u to w, $Route(u, w)$, and verify that there are no tracing processes currently in transition on it, it suffices to send a "clean-up" message (with no particular content) traversing the same path; when this message arrives, it is clear that all tracing processes destined to ξ that have been sent on this route before, have already reached w, and were not left "stuck" along the way.

```
For $i = 1$ to $q$ do:
    transfer-control-to vertex $v_i$
End-for
```

FIG. 7. *Procedure* SWEEP(u, w, ξ), *invoked at the vertex* u. *Assume* Route(u, w) = ($u = v_1, \ldots, u_q = w$).

If the edges do not provide FIFO transmission, then this property can be easily imposed, at least for the tracing processes and clean-up messages, in the standard way, as follows: Each vertex along a route *Route*(u, w) will forward tracing processes and clean-up messages reaching it in the same order it has received them. Furthermore, it will forward these messages one by one, and wait for an acknowledgment on one such message before sending the next.

Formally, let us define a procedure SWEEP(u, w, ξ), that invoked at vertex u, performs a sequence of control transfers along the path *Route*(u, w), effectively sending a message carrying the center of activity along the path and sweeping it as described above (see Figure 7).

6.4.2 *Locking the* `R_addr` *Pointers.* The last point to notice is that the SWEEP operation might still fail at its very beginning, if a searcher is still at the stage after it has obtained a regional address for the user, but before it has actually started the second stage, tracing the forwarding pointers towards the user, that is, the "tracing process" has not yet been generated.

This is the point where locking should still be used. A searcher about to examine an $\mathtt{R\_addr}^{w}(\xi)$ pointer at some vertex w in its read set, first locks this variable, and the process releases this variable only after generating and transmitting the tracing process from w towards the user's location.

We make use of the standard primitives **lock** and **unlock**. The "**lock** `var`" command waits until granted access to the variable `var` and then locks it. The "**unlock** `var`" operation releases the lock on `var`. (As discussed earlier, fairness can be guaranteed by queuing all incoming locking requests.)

For conventional simplicity, in the code we describe the operation of releasing the variable $\mathtt{R\_addr}^{w}(\xi)$ as performed remotely, by a **remote-unlock** command issued from vertex $s = R\_addr(\xi)$, that is, after the tracing process has already crossed the first segment of its trip and reached the regional address s. Clearly, better programming options exist, but we will not dwell on them any further.

6.5 THE MODIFIED PROCEDURES. In summary, the process of moving the user ξ from s to t is started by actually relocating ξ from s to t, and setting a forwarding pointer $\mathtt{Forward}^{s}(\xi)$ at s pointing to t. In fact, since a relocation operation might take some non-negligible time, a fast searcher arriving at s might follow the forwarding pointer and reach t before t is even aware of the relocation process taking place. It is therefore necessary to precede the relocation process itself by setting up a "control process" for the user ξ at t, that will manage a waiting queue for searchers that arrive at t looking for ξ in midst of the relocation.

The procedure then registers ξ as residing at t at all the regional directories up to the appropriate level. Only once this step is completed, the procedure starts deleting pointers to ξ at its old residence s from the regional directories. This is done top-down, while sweeping the routes to ensure all messages in transit succeed in finding ξ.

```
$i \leftarrow 0$                                           /* level in the hierarchy */
Repeat forever                /* retrieve lowest possible regional address $R\_addr_i(\xi)$ */
    $i \leftarrow i + 1$
    transfer-control-to vertex $u = Read_i(v)$
    lock $R\_addr_i^u(\xi)$
    If $R\_addr_i^u(\xi) \neq nil$ then goto Found
    unlock $R\_addr_i^u(\xi)$
    transfer-control-to $v$                /* No regional address found - try next level */
End-Repeat

Found: transfer-control-to vertex $R\_addr_i^u(\xi)$      /* move to the vertex $R\_addr_i(\xi)$ */
remote-unlock $R\_addr_i^u(\xi)$
Repeat                                          /* trace $\xi$ along forwarding pointers */
    transfer-control-to vertex $u = Forward^{\otimes}(\xi)$ along the fixed route $Route(\otimes, u)$
Until the user $\xi$ is found locally                  /* the vertex $Addr(\xi)$ is reached */
```

FIG. 8. *Procedure* FIND($\mathscr{D}$, ξ, v), *invoked at the vertex* v (*concurrent case*).

Formal specification for the implementation of the FIND and MOVE procedures is given in Figures 8 and 9, respectively. The (simplified) R_FIND procedure is canceled, and its actions appear directly inside the FIND procedure. Procedure R_INS is left unchanged. The implementation of the modified R_DEL operation is given in Figure 10. The formal code for procedure SWEEP is given in Figure 7.

Let us remark that the code of the modified MOVE procedure can be made somewhat more efficient by more careful design of the SWEEP procedure, particularly, avoiding the return to the point of origin after each invocation of the SWEEP procedure, but we will ignore such optimizations throughout the sequel.

6.6 CORRECTNESS AND COMPLEXITY. We need to prove that our modified procedures guarantee the requirements of the problem in the presence of concurrent accesses, that is, that the resulting directory correctly performs its basic find operation.

Let us first argue the correctness of the first part of the FIND procedure.

LEMMA 6.6.1. *No deadlocks can be caused by the* "**lock-unlock**" *commands in the* FIND *and* MOVE *procedures.*

PROOF. A deadlock might occur if (at least) two concurrent processes try to obtain locks on the same two (or more) variables (in which case it may happen that each process has been granted one lock, both wait for the other lock, and none releases the one it has already obtained).

In our case, each iteration in the first loop of the FIND procedure requests a lock on a single variable, and performs a constant number of steps before releasing the lock. Also, the MOVE procedure requests locks sequentially, that

```
Set up a control process (and waiting queue) for $\xi$ at $t$
For $1 \le i \le \delta$ do:
    $C_i^\xi \leftarrow C_i^\xi + dist(s,t)$
End-for
$J^\xi \leftarrow \max\{i \mid C_i^\xi \ge 2^{i-1}\}$
$Forward^s(\xi) \leftarrow$ remote-write "$t$"
Relocate user $\xi$ to vertex $t$, along with its mobile data structure $MDS^\xi$
transfer-control-to vertex $t$
$Forward^u(\xi) \leftarrow$ remote-write "$t$", for vertex $u = R\_addr^\xi_{J^\xi+1}(\xi)$
For $1 \le i \le J^\xi$ do:
    $chain_i^{\hookrightarrow} \leftarrow R\_addr_i^\xi$                /* save old regional addresses */
    $R\_addr_i^\xi \leftarrow$ "$t$"
    $C_i^\xi \leftarrow 0$
    $R\_INS(\mathcal{RD}_i, \xi, t)$              /* register new regional address in $\mathcal{RD}_i$ */
End-for
$chain^{\hookrightarrow}_{J^\xi+1} \leftarrow R\_addr^\xi_{J^\xi+1}$
transfer-control-to vertex $R\_addr^\xi_{J^\xi+1}(\xi)$
For $i = J^\xi$ to 1 do:
    $SWEEP(chain^{\hookrightarrow}_{i+1}, chain_i^{\hookrightarrow}, \xi)$
    $R\_DEL(\mathcal{RD}_i, \xi, \otimes)$         /* eliminate old regional address from $\mathcal{RD}_i$ */
    $Forward^{\otimes}(\xi) \leftarrow nil$
End-for
transfer-control-to $s$
$SWEEP(s, t, \xi)$
```

FIG. 9. *Procedure* MOVE($\mathscr{D}$, ξ, s, t), *invoked at* $s = Addr(\xi)$ *(concurrent case).*

```
$R\_DEL(\mathcal{RD}_i, \xi, s_i)$:              /* invoked at the vertex $s_i$, the old $R\_addr_i(\xi)$ */
    For all $u \in Write(s_i)$ do:
        transfer-control-to vertex $u$
        lock $R\_addr_i^u(\xi)$
        If $R\_addr_i^u(\xi) = s$ then $R\_addr_i^u(\xi) \leftarrow nil$
        unlock $R\_addr_i^u(\xi)$
        $SWEEP(u, s_i, \xi)$         /* verify all tracing processes for $\xi$ have arrived $s_i$ */
    End-for
```

FIG. 10. The modified R_DEL operation of the m-regional directory $\mathscr{RD}$, based on an m-regional matching $\mathscr{RW}$ (concurrent case).

is, at most one at a time, since it only requests a lock within the R_DEL procedure. Hence, no deadlocks can occur. □

LEMMA 6.6.2. *The first stage of the* FIND *procedure always terminates.*

PROOF. By the last lemma, the procedure never deadlocks. At level $i = \delta$, the regional address variables are non-*nil*, and the procedure will therefore obtain a regional address of the user on that level, if it did not obtain one on a lower level. □

LEMMA 6.6.3. *Consider a find request $F = $ FIND$(\mathcal{D}, \xi, v)$ issued by a vertex v. If all the hosts $Host_j$ visited by the user during the execution of the first stage of F are restricted to the 2^i neighborhood of the vertex v, then F will succeed in retrieving $R\_addr_i(\xi)$ (or a lower-level regional address).*

PROOF. Consider the segment $(Host_{j_1}, \ldots, Host_{j_2})$ of the $\overline{H}$ sequence, containing the hosts visited by the user during the time interval $[T_{start}(F), T_{mid}(F)]$. (This interval is finite by the last lemma.) For each vertex $Host_j$ on this segment, the set $Write_i(Host_j)$ intersects with the set $Read_i(v) = \{w\}$, that is, contains w. Therefore the variable $\mathtt{R\_addr}_i^w(\xi)$ at the vertex w always points at some vertex $Host_j$, at any time in $[T_{start}(F), T_{mid}(F)]$. Consequently, v will succeed in retrieving the regional address $R\_addr_i(\xi)$ on its ith iteration (if it did not obtain a regional address on an earlier iteration). □

Next, let us argue the correctness of the second part of the FIND procedure. By arguments similar to the sequential case (Lemma 5.4.1), we have:

LEMMA 6.6.4. *The concurrent* MOVE *procedure maintains the reachability and proximity invariants.*

LEMMA 6.6.5. *The modified directory guarantees the clean move requirement. Namely, if the searcher v has already obtained a regional address $R\_addr_i(\xi)$ of ξ at time T_0, then the user will not finish a move M that involves updating regional directories up to level $l \geq i - 1$ before it is found by v.*

PROOF. Suppose that at time T_0, the searcher v has obtained a regional address $s_i = R\_addr_i(\xi)$ of ξ, at some vertex $w \in Read_i(v)$. The FIND procedure then starts tracing the user from w to s_i, and then along forwarding pointers to lower-level regional addresses $s_{i-1}, \ldots, s_1 = Addr(\xi)$, going along the unique routes $Route(w, s_i)$ and then $Route(s_j, s_{j-1})$ in each segment of the way.

Now suppose that at some time $T_1 \geq T_0$ the user wishes to perform move M as in the lemma, and move from s_1 to some new location t. Note that the user is allowed to do that right away, without waiting for any searcher(s). However, it must set up a forwarding pointer from s to t, so if v's tracing process reaches s, it will be forwarded to t. Moreover, before the user is allowed to complete its move procedure, it is required to sweep the exact same routes traversed by v's tracing process from w through $s_i, s_{i-1}, \ldots, s_2$ to s_1 and finally to t, namely, $Route(w, s_i)$ followed by $Route(s_j, s_{j-1})$ for $j = i, i - 1, \ldots, 2$, followed by $Route(s, t)$. The sweeping process guarantees that once the move M is completed, v has already found the user (at either s or t). □

Let us finally say a word on the resulting complexity bounds.

LEMMA 6.6.6. *The stretch factor for the* FIND *operation in the concurrent case remains $Stretch_{find} = O(Deg_{read} \cdot Rad_{read})$.*

PROOF. In the implementation of a find operation $F = $ FIND$(\mathcal{D}, \xi, v)$, the number of messages required is of the same order as in the sequential case. Indeed, assuming $Opt\_cost(F) \leq 2^i$, namely, assuming that the user ξ does not leave $N_{2^i}(v)$, the 2^i-neighborhood of v, during this operation, one can easily verify that no R_FIND operations are invoked in regional directories of level $i + 2$ or higher, that is, the R_FIND operation succeeds at some level $j \leq i + 1$. Moreover, the total length of the paths traversed by the tracing process along forwarding pointers is $O(2^i)$, even if the user makes several moves (inside the 2^i-neighborhood of v) after the tracing stage has started. □

LEMMA 6.6.7. *The stretch factor for the* MOVE *operation in the concurrent case remains* $Stretch_{move} = O(Rad_{write} \cdot Deg_{write} \cdot \delta + \delta^2/\log n)$.

PROOF. In the implementation of a move operation $M = \text{MOVE}(\mathscr{D}, \xi, s, t)$, the number of messages required is of the same order as in the sequential case. In particular, the cost of the sweep operations is as follows: In Procedure R_DEL$(\mathscr{RD}_i, \xi, s_i)$, we perform a sweep from every $u \in Write(s_i)$ to s_i. Each such sweep costs $O(2^i \cdot Rad_{write})$, so the entire sweep operations in the procedure cost $O(2^i \cdot Rad_{write} \cdot Deg_{write})$. In addition, sweeping the chain of forwarding pointers in a move that updates regional directories of levels l and below costs $O(2^l)$. Hence, the total cost of sweep operations for a move that updates regional directories of levels l and below is $O(2^l) + \sum_{i=1}^{l} O(2^i \cdot Rad_{write} \cdot Deg_{write}) = O(2^l \cdot Rad_{write} \cdot Deg_{write})$. The rest of the analysis can thus follow the proof of Lemma 5.4.4. □

Again, considering the modified hierarchical directory server $\mathscr{D}$ obtained by picking $k = \log n$, we have

THEOREM 6.6.8. *The modified hierarchical directory server* $\mathscr{D}$ *allowing concurrent accesses satisfies* $Stretch_{find} = O(\log n)$ *and* $Stretch_{move} = O(\delta \cdot \log^2 n + \delta^2/\log n)$ *and uses a total of* $O(N \cdot \delta \cdot \log^2 n + N \cdot \delta^2 + n \cdot \delta \cdot \log n)$ *memory bits for handling N users.*

PROOF. The only change in complexity results from the modification in the construction of the regional matching. This modification causes (minor) changes in the stretch complexities of the "find" and "move" operations due to the switch between Deg_{read} and Deg_{write}. In particular, a $\log n$ factor is shifted from $Stretch_{find}$ to $Stretch_{move}$, and from one factor to the other in the memory requirements. □

Let us finally remark that there are other ways for solving the "race" problem handled in Subsection 6.3, which leave unchanged the complexity bounds for both operations, as well as the memory requirements, at the cost of somewhat more complicated code.

7. *Discussion*

This final section discusses several related aspects and possible generalizations of the proposed directory server.

7.1 PREPROCESSING. So far, we have not taken preprocessing costs into account. These costs are of two types. First, it is necessary to construct the necessary underlying construction, namely, the hierarchy of regional matchings. This can be done using the distributed clustering techniques of Awerbuch and Peleg [1990d] with a global communication cost of $O(E \cdot \log^4 n)$.

The second type of preprocessing occurs whenever a new user is introduced into the system, or an old one is removed. In the setting considered in this paper, the operation of inserting a new user into the system requires a considerably heavy initialization operation on the directories. In particular, such a user ξ has to be registered in the regional directories at all levels, by setting a pointer $\texttt{R\_addr}^{u}(\xi) = s$ to the user's new residence s at all vertices $u \in Write_i(s)$, for all $1 \le i \le \delta$. The total cost of this initialization operation is $O(D(G) \cdot Rad_{write} \cdot Deg_{write})$, which in our construction amounts to $O(D(G) \cdot \log n)$. Note that in our framework, this cost cannot be charged directly to any "inherent cost", since the user has never moved yet.

Although this preparation cost is rather heavy, it does not seem to be exceedingly expensive, considering the fact that it is incurred only once throughout the lifetime of the user in the system. In fact, it is possible to unify the cost analysis to cover also the preparation stage, by treating the introduction of a new user as a migration of that user from some external location to its new residence "across the entire network", and consequently defining its inherent cost as $D(G)$.

A similar discussion applies also to deletions of users from the system.

7.2 FAULT TOLERANCE. Our solution, as described, assumes a static network, and does not tolerate failures of network processes. Let us now briefly address the issue of handling faults. In most existing networks, link failures are detected via time-outs of low-level data link protocols. This enables us to use a "semi-static" approach to failures. The network can be viewed as a dynamic graph, where edges fail and recover arbitrarily, but failures are detectable. Hence, whenever the network topology changes, this fact is detected by the involved vertices, who then trigger the initiation of an adaptation stage, replacing the present (static) tracking structure with an entirely new (static) structure matching the current topology.

In this sense, the situation is analogous to that of most common algorithms for routing and other topology sensitive tasks (cf. McQuillan et al. [1980]) where the topological databases and delay estimates have to be recomputed periodically in an adaptive fashion. Such semi-static methods are successful as long as the topology of the network stays stable for sufficiently long time between successive restructurings. In the fiber-optic-technology-based networks of the 90's (cf. Cidon et al. [1988]), this is a quite realistic assumption.

More specifically, the transformation from the existing structure to the next is done in two steps. The first step is to wipe out all traces of the previous structure. This can be done via general protocol transformers, for example, reset protocols [Finn 1977; Afek et al. 1987]. These protocols "clean" the network from messages and data related to previous computations.

The second step involves recomputing the new structure (from scratch). This requires, in particular, recomputing all the necessary data structures, namely, the hierarchy of regional matchings. As mentioned earlier, such restructuring can be performed in communication cost $O(E \cdot \log^4 n)$ using the techniques of Awerbuch and Peleg [1990d]. It is also necessary to register each user ξ in the regional directories at all levels, by setting the appropriate pointers to its residence at all the vertices in its $Write_i$ sets. The total cost of this operation amounts to $O(D(G) \cdot \log n)$ per user. Thus the overall restructuring cost is $O(E \cdot \log^4 n + ND(G) \cdot \log n)$ for N users.

Since any topological change can completely change the distance metric, there is apparently no way to avoid such global restructuring in the worst case. Further research is needed in order to develop efficient *online* methods for updating our data structures, attempting to take advantage of the probable resemblance between the previous and current topology.

7.3 MIGRATION CONTROL. An interesting area of research concerns developing good policies for moving servers in the network, in response to a stream of requests, in order to achieve best performance. In the literature on sequential algorithms this problem is known as the "k-server" problem (cf. Manasse et al. 1988]). In the distributed setting there are additional complicating factors, having to do with the effects of locality and partial information. Even though this question is not dealt with in this paper, our contribution is in showing that the limitation of partial information is not very restrictive, in the sense that one can get by paying no more than a polylogarithmic factor in performance compared to an algorithm which obtains full information for free. Subsequent papers [Bartal et al. 1992; Bartal and Rosen 1992; Awerbuch et al. 1993] have extended our work by dealing with this generalized framework and providing efficient solutions for the control problem as well.

ACKNOWLEDGMENTS. The authors thank Michael Fischer for the stimulating discussion that triggered this research, and Richard Karp for his helpful comments. Thanks are also due to two anonymous referees whose comments greatly helped to improve the presentation of this paper.

REFERENCES

AFEK, Y., AWERBUCH, B., AND GAFNI, E. 1987. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Symposium on Foundations of Computer Science*. IEEE, New York.

AWERBUCH, B. 1985. Complexity of network synchronization. *J ACM 32*, 804–823.

AWERBUCH, B., BARTAL, Y., AND FIAT, A. 1993. Competitive distributed file allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing* (San Diego, Calif., May 16–18). ACM, New York, pp. 164–173.

AWERBUCH, B., BERGER, B., COWEN, L., AND PELEG, D. 1991a. Fast constructions of sparse neighborhood covers. Unpublished manuscript.

AWERBUCH, B., BERGER, B., COWEN, L., AND PELEG, D. 1992a. Low diameter graph decomposition is in NC. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory* (Helsinki, Finland, July). pp. 83–93.

AWERBUCH, B., BERGER, B., COWEN, L., AND PELEG, D. 1992b. Fast network decomposition. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 169–178.

AWERBUCH, B., GOLDBERG, A., LUBY, M., AND PLOTKIN, S. 1989. Network decomposition and locality in distributed computation. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*. IEEE, New York.

AWERBUCH, B., KUTTEN, S., AND PELEG, D. 1991b. On buffer-economical store-and-forward deadlock prevention. *Proceedings of INFOCOM* (Apr.). pp. 410–414.

AWERBUCH, B., KUTTEN, S., AND PELEG, D. 1992c. Competitive distributed job scheduling. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 571–580.

AWERBUCH, B., PATT-SHAMIR, B., PELEG, D., AND SAKS, M. 1992d. Adapting to asynchronous dynamic networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 557–570.

AWERBUCH, B., AND PELEG, D. 1990. Sparse partitions. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE, New York, pp. 503–513.

AWERBUCH, B., AND PELEG, D. 1990b. Network synchronization with polylogarithmic overhead. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*. IEEE, New York, pp. 514–522.

AWERBUCH, B., AND PELEG, D. 1990c. Routing with polynomial communication-space trade-off. *SIAM J. Disc. Math. 5*. 151–162.

AWERBUCH, B., AND PELEG, D. 1990d. Efficient distributed construction of sparse covers. Tech. Rep. CS90-17. The Weizmann Institute, Rehovot, Israel, July.

BAR-NOY, A., AND KESSLER, I. 1993. Tracking mobile users in wireless networks. *IEEE Trans. Inf. Theory 39*, 1877–1886.

BAR-NOY, A., KESSLER, I., AND SIDI, M. 1994. Mobile users: To update or not to update? In *Proceedings of the IEEE INFOCOM* (Toronto, Ont., Canada, June).

BARTAL, Y., FIAT, A., AND RABANI, Y. 1992. Competitive algorithms for distributed data management. In *Proceedings of the 24th ACM Symposium on Theory of Computing* (Victoria, B.C., Canada, May 4–6). ACM, New York, pp. 39–50.

BARTAL, Y., AND ROSEN, A. 1992. The distributed k-server problem—A competitive distributed translator for k-server algorithms. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science.*

CHANDY, K. M., AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 63–75.

CIDON, I., GOPAL, I., AND KUTTEN, S. 1988. New models and algorithms for future networks. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 75–89.

FINN, S. G. 1979 Resynch procedures and a fail-safe network protocol. *IEEE Trans Commun. COM-27*, 840–845.

GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. 1983. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. Syst. 5*, 66–77.

KRANAKIS, E., AND VITÁNYI, P. M B. 1988. A note on weighted distributed match-making. In *AWOC.* Lecture Notes in Computer Science, vol. 319. Springer-Verlag, New York, pp. 361–368.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 558–565.

LANTZ, K. A., EDIGHOFFER, J. L., AND HISTON, B. L. 1985. Towards a universal directory service. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 5–7). ACM, New York, pp. 250–260.

LINIAL, N., AND SAKS, M. 1991. Decomposing graphs into regions of small diameter. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, Calif.). ACM, New York, pp 320–330.

MANASSE, M. S., MCGEOCH, L. A., AND SLEATOR, D. D. 1988. Competitive algorithms or on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (Chicago, Ill., May 2–4). ACM, New York, pp. 322–333.

MCQUILLAN, J., RICHER, I., AND ROSEN, E. C. 1980. The new routing algorithm for the ARPANET. *IEEE Trans. Commun 28*, 711–719.

MULLENDER, S. J., AND VITÁNYI, P. M. B. 1988. Distributed match-making. *Algorithmica 3*, 367–391.

PELEG, D. 1989a. Sparse graph partitions. Rep. CS89-01, Dept. of Applied Math. The Weizmann Institute, Rehovot, Israel, Feb.

PELEG, D. 1989b. Distance-dependent distributed directories, Rep. CS89-10, Dept. of Applied Math. The Weizmann Institute, Rehovot, Israel, May.

PELEG, D., AND SCHÄFFER, A. A. 1989. Graph spanners. *J. Graph Theory 13*, 99–116.

PELEG, D., AND ULLMAN, J. D. 1989. An optimal synchronizer for the hypercube. *SIAM J. Comput. 18*, 740–747.

PELEG, D., AND UPFAL, E. 1989. A trade-off between size and efficiency for routing tables, *J. ACM 36*, 510–530.

RECEIVED OCTOBER 1990; REVISED MAY 1995; ACCEPTED JUNE 1995