A Virtual Node-Based Tracking Algorithm for Mobile Networks

Tina Nolte and Nancy Lynch
MIT CSAIL
Cambridge, MA 02139

Abstract— **We introduce a virtual-node based mobile object tracking algorithm for mobile sensor networks, VINESTALK. The algorithm uses the *Virtual Stationary Automata* programming layer, consisting of mobile clients, virtual timed machines distributed at known locations in the plane, called virtual stationary automata (VSAs), and a communication service connecting VSAs and mobile clients.**

VINESTALK maintains a data structure on top of an underlying hierarchical partitioning of the network. In a grid partitioning, operations to *find* a mobile object distance d away take $O(d)$ time and communication to complete. Updates to the tracking structure after the object has moved a total of d distance take $O(d * log$ network diameter$)$ amortized time and communication to complete. The tracked object may relocate without waiting for VINESTALK to complete updates for prior moves, and while a *find* is in progress.

Keywords: Virtual nodes, sensor networks, hierarchical partitioning, tracking, distributed data structures.

Technical areas: Algorithms and theory, Wireless and mobile computing, Sensor networks and ubiquitous computing

I. INTRODUCTION

A system with no fixed infrastructure in which mobile clients may wander in the plane and assist each other in forwarding messages is called an ad-hoc network. The task of designing algorithms for constantly changing networks is difficult. Highly dynamic networks, however, are becoming increasingly prevalent, especially in the context of pervasive and ubiquitous computing, and it is therefore important to develop and use techniques that simplify this task.

In this paper we describe a tracking algorithm for a mobile sensor network, VINESTALK (VIrtual NodE STALK). To simplify the implementation, we mask the unpredictable behavior of mobile nodes by using a *virtual* infrastructure, consisting of mobile client automata, timing-aware and location-aware machines at fixed locations, called *Virtual Stationary Automata* (VSAs) [7], [6], that mobile clients can interact with and use to coordinate their actions, and a communication service connecting VSAs and clients.

VINESTALK maintains a tracking path over a hierarchical partitioning of the network, providing low cost find and move operations. We implement updates to the tracking structure by means of two local actions, *grow* and *shrink*. A grow enables a path to grow from the new location of the mobile object to increasingly higher levels of the hierarchy and connect to the original path. A shrink cleans branches deserted by the object. Shrinking also starts at the lowest level and climbs to increasingly higher levels. Despite the fact that grow and shrink occur concurrently, we complete the move operation successfully by using suitably-chosen timers to determine when these actions are performed.

VINESTALK provides good locality guarantees in a grid-based hierarchy; a *move* of the object being tracked to distance d away requires $O(d * logD)$ time and communication (work) to update the tracking structure, where D is the network diameter. We also describe a *find* operation using the tracking structure. A find operation invoked at a process queries neighboring processes at increasingly higher levels of the clustering hierarchy until it encounters a process on the tracking path. Once the path is found, the find operation follows it to its leaf to reach the mobile object. We show that a *find* invoked within distance d of the mobile object requires $O(d)$ work to reach the object. Furthermore, VINESTALK achieves seamless tracking of a continuously moving object by allowing concurrent tracking and finding operations.

Virtual Stationary Automata programming layer. In prior work [7], [9], [8], we developed a notion of "virtual nodes" for mobile ad hoc networks. A virtual node is an abstract, relatively well-behaved active node that is implemented using less well-behaved real physical nodes. The static infrastructure we use in this paper includes fixed, timed virtual machines with an explicit notion of real time, called *Virtual Stationary Automata* (VSAs), distributed at known locations over the plane [7], [6]. Each VSA represents a predetermined geographic area and has broadcast capabilities similar to those of the mobile physical nodes. Many algorithms depend significantly on timing, and it is reasonable to assume that many mobile nodes have access to reasonably synchronized clocks. In the VSA layer, VSAs also have access to *virtual* clocks, guaranteed to not drift too far from real time. The layer provides mobile nodes with a fixed virtual infrastructure, reminiscent of more traditional and better understood wired networks, with which to coordinate their actions.

Our VSA layer is emulated by physical mobile nodes in the network. Each physical node is periodically told its region by the GPS. A VSA for a particular region is then emulated by a subset of the mobile nodes in its region: the VSA state is maintained in the memory of the physical nodes emulating it, and the physical nodes perform VSA actions on behalf of the VSA. If no physical nodes are in the region, the VSA fails; if physical nodes later arrive, the VSA restarts.

Tracking and location management. An algorithm for tracking on sensor motes, called STALK (Stabilizing Tracking viA Layered linKs), was presented in [2]. It also maintains a tracking path imposed on an underlying hierarchical partitioning of the sensor network into clusters.

VINESTALK is a more robust approach, for mobile networks, implementing a similar algorithm over a stationary network of VSAs. The clients (sensor nodes) broadcast object detections to their associated local VSAs. The tracking path of STALK is then maintained by the VSAs in the network, rather than directly by the client nodes themselves. The predetermined locations of the stationary VSAs allow the use of a *static hierarchy* of VSA regions, which circumvents the need for a difficult-to-provide dynamic global clustering of the clients in the network, as was required in STALK.

In addition to moving tracking path maintenance to replicated VSAs, we generalize the hierarchical clusterings allowed. The cluster definitions assumed for STALK were very restrictive in terms of the cluster geometries; for example, they disallowed hierarchical grid clusters such as those in [14], [1]. Our generalization of the cluster definitions allows grid clusters, while still guaranteeing time and work bounds similar to those of the original STALK algorithm. In generalizing the cluster definitions, we had to augment the original algorithm to include secondary pointers to the tracking path, to help guarantee low-work find operations.

Hierarchies have long been used to facilitate design of efficient and scalable protocols. For example, Awerbuch and Peleg [4] described distributed directory servers to store location information for mobile objects. A hierarchy of regional directories, based on read/write quorums of stationary nodes, is constructed so that each level l directory enables a node to find a mobile object within 2^l distance. The communication cost of a find for an object d away is $O(d * log^2 N)$ and that of a move of distance d is $O(d * logD * logN + log^2 D/logN)$ (where N is the number of nodes and D is network diameter). However, a topology change, such as a node failure, necessitates a global reset of the system since the regional directories depend on a non-local clustering program [3] that constructs a sparse cover of a graph.

The *Grid location service (GLS)* [14] maps each tracked client id to geographic coordinates where clients in the area are responsible for saving the tracked client's location. However, it suffers from the "dithering problem", as does the distributed *Arrow* protocol [12], wherein an object moving back and forth across a multi-level hierarchy boundary may lead to nonlocal updates. The *Locality-aware Location Service (LLS)* [1] is based on a grid hierarchy of lattice points for tracked clients, published with locations of associated nodes. Lattice points can be queried for the desired location, with a query traversing a spiral path of lattice nodes increasingly distant from the source until it reaches the destination. The protocols in [5] do not exploit the hierarchy idea and are not scalable for large networks. In [11], using a hierarchy of location servers, a stabilizing location management protocol is presented, but does not ensure locality of finds.

II. MODEL ASSUMPTIONS

A. Network tiling

The deployment space is assumed to be a fixed, closed, and bounded connected region of the 2-D plane, divided into known connected *regions*, with unique ids drawn from an ordered set of *region identifiers*, U. Each region contains its boundaries, and the only overlap of points permitted at distinct regions is at their shared boundaries. For simplicity a boundary point's region is the one with minimum id.

We define a relation nbr, on ids from U, such that two regions are neighboring iff they share any boundary points.

We define the distance between regions u and v to be the length of the shortest path between u and v in the neighbor graph induced by the nbr relation. The network diameter, D, is the maximum distance between any two regions.

B. Cluster hierarchy

The regions are organized into a cluster hierarchy, expressed as a four-tuple $(C, L, cluster : (U \times L) \rightarrow C, h : C \rightarrow U)$, where a cluster is a connected set of regions:

- C is a set of *cluster ids*,
- L is a set of *levels* $\{0, \ldots, MAX\}, MAX > 0$,
- Total, onto function $cluster$ maps a region u and level l to the name of the level l cluster containing u,
- Total function h maps a cluster id to the name of the region that is the *head* of that cluster.

The following are derived terminology for $c, c' \in C, l \in L$:

- $level(c) = l \Leftrightarrow \exists u \in U : cluster(u, l) = c$,
- Cluster c's member regions: $members(c) = \{u \in U | cluster(u, level(c)) = c\}$,
- Neighboring clusters at c's level: $nbrs(c) = \{c' \in C | c \neq c' \wedge level(c) = level(c') \wedge \exists u \in members(c), v \in members(c') : nbr(u, v)\}$,
- Cluster children: $children(c) = \{c' \in C | level(c') = level(c) - 1 \wedge members(c') \subseteq members(c)\}$,
- $parent(c) = c' \Leftrightarrow c \in children(c')$.

We require that:

1) Each cluster belongs to exactly one level.
2) There is exactly one level MAX cluster.
3) Each region is the only member of its level 0 cluster.
4) Distinct clusters at the same level don't overlap.
5) Any members of the same level l cluster, $l \neq MAX$, are members of the same level $l + 1$ cluster.
6) The head, $h(c)$, of a cluster c is a member of c.

We make several geometry assumptions about the clustering, given functions n, p, q, ω, each from L to $Nat^{\geq 0}$:

1) (Proximity): Consider any sequence of clusters $\{c_l, c_{l-1}, \cdots, c_k\}$, $0 \leq k \leq l \leq MAX$, such that $level(c_l) = l$ and for each $c_{j,j \neq l}, level(c_j) = j$ and $\exists c \in \{c_j\} \cup nbrs(c_j) : members(c) \subseteq members(c_{j+1})$. Then for any region v neighboring a region in c_k, $cluster(v, l) \in \{c_l\} \cup nbrs(c_l)$.
2) A cluster at level l has at most $\omega(l)$ neighbors.
3) Any member of a level l cluster, $l \neq MAX$, is at most $n(l)$ distance from any member of a cluster neighbor.
4) Any member of a level l cluster, $l \neq MAX$, is at most $p(l)$ distance from any member of its level $l+1$ cluster.
5) Any region up to $q(l)$ distance from a region in a level l cluster is either in that cluster or one of its neighbors.

Notice that $q(0) = 1$ and $q(l) \leq n(l)$ for all $l \in L$. Also, for any level $l + 1$ cluster c, any neighbors of neighbors of level l clusters contained in c are either contained in c or a neighbor. This, in turn, implies that $2q(l-1) \leq q(l)$.

We also assume the following relationships. While these conditions are not implied for each selection of n, p, q, for a clustering satisfying our earlier conditions, there exist n, p, q (namely the tight ones) that describe the geometries of the clustering while satisfying these enumerated assumptions:

1) $n(l) \leq n(l+1)$
2) $p(l) \leq p(l+1)$
3) $p(l) \leq n(l+1)$

Example: Grid hierarchy: One hierarchy is a base r grid, where regions of size 1 are partitioned into $r \times r$ square level 1 clusters, which are themselves partitioned into square level 2 clusters, etc. Squares that share edges or are diagonal from one another, sharing a single border point, are neighbors. Any region in a cluster can be the clusterhead.

It is easy to verify that this satisfies the proximity requirement, and that we can describe MAX, n, p, q, ω as follows: (1) $MAX = \lceil \log_r(D+1) \rceil$, (2) $n(l) = 2r^l - 1$, (3) $p(l) = r^{l+1} - 1$, (4) $q(l) = r^l$, and (5) $\omega(l) = 8$.

C. Virtual Stationary Automata layer with C-gcast

We describe the VSA layer components. Figure 1 depicts the VSA layer, together with cluster geocast (Section II-C.3).

1) Client nodes: For each p in the set of physical node identifiers P, we assume a mobile timed I/O automaton [13] client C_p, with access to an accurate local clock, now. A $\mathsf{GPSupdate}(u)_p$ happens at a client C_p whenever it enters the system or changes region, indicating to the client the region u where it is currently located.

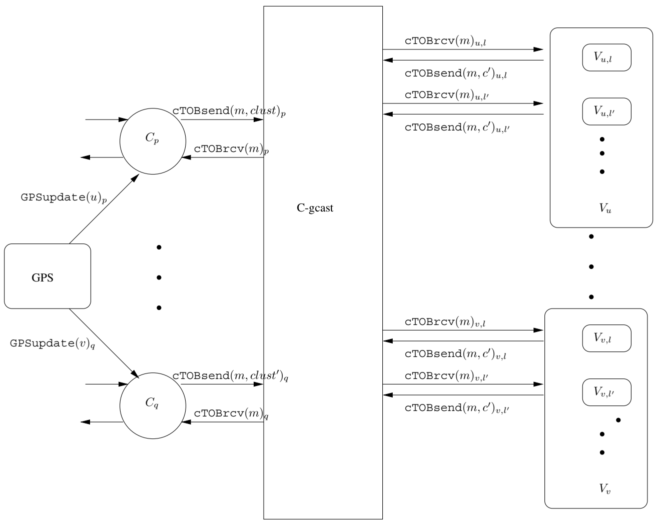
Each client C_p has access to a communication service C-gcast, allowing it to communicate with its region's VSA through $\mathsf{cTOBsend}(m, clust(u,0))_p$ and $\mathsf{cTOBrcv}(m)_p$.

Clients are susceptible to stopping failures. After a stopping failure, a client performs no additional local steps until restarted. If restarted, it starts again from an initial state.

Additional arbitrary external interface actions and local state used by algorithms running at the client are allowed. For simplicity we assume that local steps take no time.

2) Virtual Stationary Automata (VSAs): An abstract VSA is a clock-equipped virtual machine that may be emulated by the physical mobile nodes in its region in the network (a self-stabilizing implementation of VSAs using a GPS oracle and the physical mobile nodes in the system can be found in [7], [6]). For each region u, we formally describe a VSA for the region as a timed I/O automaton V_u, with access to an accurate local clock, now. In the context of our clustering hierarchy, a VSA V_u is a union of subautomata $V_{u,l}$, for levels l where $h(cluster(u,l)) = u$, corresponding to one submachine per cluster its region is head of (hosting).

To emulate a VSA using physical nodes, its interface must be emulatable by them. Hence, a VSA's external interface is restricted to only stopping failure, restarts, and the ability to send and receive messages through C-gcast. Since a VSA is emulated by physical nodes (corresponding to clients) in its region, its failures are defined in terms of clients: a clientless region's VSA is failed, a VSA only fails if clients fail or leave its region, and a failed VSA restarts if its region has some clients that don't fail/leave for some $t_{restart}$ time.



$\mathtt{cTOBrcv}(m)_{u,l}$
$\mathtt{cTOBsend}(m,c')_{u,l}$
$\mathtt{cTOBrcv}(m)_{u,l'}$
$\mathtt{cTOBsend}(m,c')_{u,l'}$
$V_{u,l}$
$V_{u,l'}$
V_u
$\mathtt{cTOBsend}(m,clust)_p$
$\mathtt{cTOBrcv}(m)_p$
C_p
$\mathtt{GPSupdate}(u)_p$
C-gcast
GPS
$\mathtt{GPSupdate}(v)_q$
$\mathtt{cTOBsend}(m,clust')_q$
$\mathtt{cTOBrcv}(m)_q$
C_q
$\mathtt{cTOBrcv}(m)_{v,l}$
$\mathtt{cTOBsend}(m,c')_{v,l}$
$\mathtt{cTOBrcv}(m)_{v,l'}$
$\mathtt{cTOBsend}(m,c')_{v,l'}$
$V_{v,l}$
$V_{v,l'}$
V_v

Fig. 1. Virtual Stationary Automata layer and C-gcast.

3) Cluster geocast service (C-gcast): We assume a cluster geocast service, allowing: (a) a VSA at a region u hosting a level l cluster to send a message m to local clients or a VSA hosting a cluster c' via a $\mathsf{cTOBsend}(m, c')_{u,l}$ action, or (b) a client to send a message to the level 0 VSA in its region or a neighboring one using a similar action. A message is received via a $\mathsf{cTOBrcv}(m)_{u,l}$ action. We assume that a received message was previously sent.

If no VSAs are failed over the broadcast period, a message sent will be received at exactly the following amounts of time later (δ and e are explained in "Preliminaries" below):
(a) level l cluster to neighboring cluster: $(\delta + e)n(l)$,
(b) level l cluster to parent cluster, or level $l+1$ cluster to a child cluster: $(\delta + e)p(l)$,
(c) level l cluster to a neighbor of a neighbor: $(\delta + e)2n(l)$,
(d) level 0 cluster to own or neighbor region clients: $\delta + e$,
(e) client to its current or neighboring region's cluster: δ.

Preliminaries: The VSA layer in [7], [6] includes clients and VSAs as described here, and a reliable local broadcast service, V-bcast, instead of C-gcast. V-bcast allows communication between clients and VSAs in the same or neighboring regions with message delay δ, the max delay of the underlying physical nodes' broadcast service. Physical node emulation of VSAs also leads to a requirement that the supremum distance between points in neighboring regions be at most the broadcast radius of the physical nodes. Also, the abstraction reflects that, due to failures or message delays, a VSA emulation might be behind real time, appearing to be delayed in performing outputs by up to some time e.

Afterwards, a communication service for non-neighboring VSAs was built in [10]: a DFS-based algorithm over V-bcast provided reliable bounded-delay geocast between VSAs. Our C-gcast is implemented by sending a cluster's messages, tagged with cluster information, to the cluster's VSA via this service, and delaying processing of received messages until the above described amounts of time (which account for δ and e terms imposed by the VSA layer) have transpired.

III. Problem statement

We describe the specification for the system and the structure of a solution using the VSA layer.

A. Tracking service specification

The mobile object tracking problem, for a system with a mobile object and clients with a mobile object detector, requires that queries to clients for the mobile object's region eventually produce responses at the object's region.

Mobile object. The mobile object being tracked, *Evader*, resides at exactly one region, and can nondeterministically move to a neighboring region. It is modeled with the GPS service; we assume the GPS service is augmented to provide a move input to a client exactly when the evader enters its region, and a left when the evader leaves.

Clients. In addition to the capabilities described in the system model, clients receive move, left inputs, as above. They also receive find inputs from the outside (queries for evader region) and output founds (at a current evader region).

Tracking service. The tracking service is a composition of the GPS service (modeling the mobile object) and clients such that an execution starting from an initial state with no mobile object detections and no outstanding find requests, where the first move input precedes the first find, and there is at least one alive client in a region whenever the mobile object enters or leaves it, satisfies the following:

1. A find is eventually followed by a found,
2. Each found occurs in a region hosting the mobile object, and is in response to a prior find.

B. Implementation using the VSA layer

Our tracking service implementation, VineStalk, described in the next two sections, is built on the VSA layer. For each region u, VSA V_u runs a *Tracker*$_{u,l}$ subautomaton for each level l cluster the region heads. (We refer to $Tracker_{u,l}$ as process $clust$, where $clust = cluster(u, l)$.)

The VSAs maintain a tracking structure by propagating mobile object information obtained through client broadcasts of object detections, and use this structure to help clients answer find queries. We show that, assuming each VSA is always alive, VineStalk implements the tracking service with the following work/time complexity on a grid:

1. A find initiated distance d from the mobile object takes $O(d)$ time and work (communication) to service,
2. If the object moves d distance, the amortized time and work to update the tracking structure is $O(d * log(D))$.

IV. VineStalk: Atomic move operations

Here we describe VineStalk, assuming that the mobile object does not relocate until the updates are completed.

A. Client algorithm

Any client that receives a move input, indicating the evader has just arrived in the region, sends a grow message to its local level 0 cluster. A client receiving a left sends a shrink message to its level 0 cluster. Since this implementation is so simple, we do not include the code for it.

B. VSA algorithm

The *Tracker*s together maintain a tracking path, rooted at the level MAX cluster, with pointers to successively lower levels of the hierarchy, and terminating at the mobile object's region. When a client grow or shrink message about an evader move is received at a VSA, it triggers updates to the tracking path. Updates to the tracking path are implemented by two local actions, grow and shrink. The grow action enables a new path to grow to increasingly higher levels of the clustering hierarchy and connect to the original path at some level. The shrink action cleans old branches deserted by the mobile object starting from the lowest levels.

A hierarchical partitioning of a network results in multi-level cluster boundaries: two neighboring regions might be contained in different clusters at all levels (except the top) of the hierarchy. If a process were to always propagate grows and shrinks to its clusterhead, a small movement of the object back and forth across a multi-level boundary could result in work proportional to network size. To resolve this "dithering" problem, but still guarantee locality of finds, we allow up to one *lateral link* per level in our tracking path. A process occasionally connects to the path with a lateral link to a neighboring process rather than a link to its hierarchy parent.

Secondary tracking pointers, indicating if a neighbor is on the tracking path, are also maintained. When a process joins the path, it checks its secondary pointers to determine whether its path parent should be a neighbor on the path or its hierarchy parent. It then informs its neighbors and path parent of its addition and whether it connected via a lateral link. Neighbors store this data in a secondary pointer.

Grows and shrinks are explained below, with TIOA-style code for *Tracker*$_{u,lvl}$, or cluster process $clust$, in Figure 2. Each process has a child pointer c, a parent pointer p, secondary tracking pointers $nbrptup$ and $nbrptdown$, and a $timer$. Initially, pointers are $\perp$ and $timer$s$= \infty$. We assume grow and shrink timers $g, s : L - \{MAX\} \rightarrow \mathbb{R}$ that satisfy:

$$\sum_{j=0}^{l}[s(j) - g(j)] > (\delta + e)n(l) \qquad (1)$$

1) Grow action at Tracker$_{u,l}$ (process clust): A grow updates a path to point to the new location of the object.

If process $clust$ receives a grow, its c gets set to the sender's cluster. If $p \neq \perp$ or $level(clust) = MAX$, the grow is done since $clust$ is already on the tracking path. Otherwise, $timer \leftarrow now + g(l)$, scheduling a grow.

When $timer$ expires, if $c \neq \perp$ still, meaning a shrink did not remove the pointer, and $p = \perp$, then a grow is sent to extend the tracking path. If $nbrptup \neq \perp$, meaning $nbrptup$ is a neighbor on the tracking path not connected by a lateral

```
Signature:
  Input cTOBrcv($(\langle m, v\rangle)_{u,lvl}$, $v \in C$, $m \in$ {grow,growNbr,
        growPar,shrink,shrinkUpd,find,findQuery,findAck}
  Output cTOBsend($(\langle m, clust\rangle, v)_{u,lvl}$, $v \in parent(clust) \cup$
        $nbrs(clust)$, $m \in$ {grow, growNbr, growPar, shrink,
                    shrinkUpd, find, findQuery, found}

State variables:
  $c \in children(clust) \cup nbrs(clust) \cup \{clust, \perp\}$, initially $\perp$
  $p \in nbrs(clust) \cup \{parent(clust), \perp\}$, initially $\perp$
  $nbrptup, nbrptdown \in nbrs(clust) \cup \{\perp\}$, initially $\perp$
  $sendq \in C \times$ {growNbr, growPar, shrinkUpd, find,
        findQuery}, initially $\emptyset$
  $timer \in \mathbb{R}$, a timer, initially $\infty$
  analog $now \in \mathbb{R}$, clock indicating current time

Move-related actions:
  Output cTOBsend($(\langle m, clust\rangle, dest)_{u,lvl}$
  Precondition
    $\langle dest, m\rangle \in sendq$
  Effect
    $sendq \leftarrow sendq - \{\langle dest, m\rangle\}$

  Input cTOBrcv($(\langle$grow$, cid\rangle)_{u,lvl}$
  Effect
    if $c = p = \perp \wedge lvl \neq MAX$ then
      $timer \leftarrow now + g(lvl)$
    $c \leftarrow cid$

  Output cTOBsend($(\langle$grow$, clust\rangle, par)_{u,lvl}$
  Precondition
    $now = timer \wedge c \neq \perp \wedge p = \perp$
    $par = nbrptup \neq \perp \vee (par = parent(clust) \wedge nbrptup = \perp)$
  Effect
    $p \leftarrow par$
    if $par = nbrptup$ then
      for each $b \in nbrs(clust)$
        $sendq \leftarrow sendq \cup \{\langle b,$ growNbr$\rangle\}$
    else for each $b \in nbrs(clust)$
        $sendq \leftarrow sendq \cup \{\langle b,$ growPar$\rangle\}$

  Input cTOBrcv($(\langle$growPar$, cid\rangle)_{u,lvl}$
  Effect
    $nbrptup \leftarrow cid$

  Input cTOBrcv($(\langle$growNbr$, cid\rangle)_{u,lvl}$
  Effect
    $nbrptdown \leftarrow cid$

  Input cTOBrcv($(\langle$shrink$, cid\rangle)_{u,lvl}$
  Effect
    if $c = cid$ then
      $c \leftarrow \perp$
      if $lvl \neq MAX$ then
        $timer \leftarrow now + s(lvl)$

  Output cTOBsend($(\langle$shrink$, clust\rangle, p)_{u,lvl}$
  Precondition
    $now = timer \wedge c = \perp \wedge p \neq \perp$
  Effect
    $p \leftarrow \perp$
    for each $b \in nbrs(clust)$
      $sendq \leftarrow sendq \cup \{\langle b,$ shrinkUpd$\rangle\}$

  Input cTOBrcv($(\langle$shrinkUpd$, cid\rangle)_{u,lvl}$
  Effect
    if $nbrptup = cid$ then
      $nbrptup \leftarrow \perp$
    if $nbrptdown = cid$ then
      $nbrptdown \leftarrow \perp$

Additional find-related Signature:
  Output cTOBsend($(\langle$findAck$, u'\rangle, v)_{u,lvl}$, $u', v \in C$
        cTOBsend($(\langle$found$, clust\rangle, clust)_{u,lvl}$
  Internal findQuery$_{u,lvl}$

Additional find-related State variables:
  $nbrtimeout \in \mathbb{R}$, a timer, initially $\infty$
  $findAckq \subseteq C \times C$, initially $\emptyset$
  $finding \in Boolean$, initially false

Additional find-related actions:
  Output cTOBsend($(\langle$findAck$, x\rangle, dest)_{u,lvl}$
  Precondition
    $\langle dest, x\rangle \in findAckq$
  Effect
    $findAckq \leftarrow findAckq - \{\langle dest, x\rangle\}$

  Input cTOBrcv($(\langle$find$, cid\rangle)_{u,lvl}$
  Effect
    $finding \leftarrow$ true
    $nbrtimeout \leftarrow \infty$

  Output cTOBsend($(\langle$found$, clust\rangle, clust)_{u,lvl}$
  Precondition
    $finding \wedge c = clust$
  Effect
    for each $j \in nbrs(clust)$
      $sendq \leftarrow sendq \cup \{\langle j,$ found$\rangle\}$
    $finding \leftarrow$ false

  Internal findquery$_{u,lvl}$
  Precondition
    $finding \wedge c = nbrptdown = \perp \wedge nbrptup \in \{\perp, p\}$
    $nbrtimeout > now + 2(\delta + e)n(lvl)$
  Effect
    $nbrtimeout \leftarrow now + 2(\delta + e)n(lvl)$
    for each $j \in nbrs(clust) - \{p\}$
      $sendq \leftarrow sendq \cup \{\langle j,$ findQuery$\rangle\}$

  Input cTOBrcv($(\langle$findQuery$, cid\rangle)_{u,lvl}$
  Effect
    if $c \neq \perp$ then
      $findAckq \leftarrow findAckq \cup \{\langle cid, c\rangle\}$
    else if $nbrptdown \neq \perp$ then
      $findAckq \leftarrow findAckq \cup \{\langle cid, nbrptdown\rangle\}$
    else if $nbrptup \neq \perp$ then
      $findAckq \leftarrow findAckq \cup \{\langle cid, nbrptup\rangle\}$

  Input cTOBrcv($(\langle$findAck$, dest\rangle)_{u,lvl}$
  Effect
    if $finding \wedge dest \neq clust \wedge c = nbrptdown = \perp \wedge nbrptup \in \{\perp, p\}$
                                                        then
      $sendq \leftarrow sendq \cup \{\langle dest,$ find$\rangle\}$
      $finding \leftarrow$ false

  Output cTOBsend($(\langle$find$, clust\rangle, dest)_{u,lvl}$
  Precondition
    $finding \wedge dest \notin \{clust, \perp\} \wedge [c = dest \vee (c = \perp \wedge dest = nbrptdown)$
      $\vee (c = nbrptdown = \perp \wedge (dest = nbrptup \neq p \vee (nbrtimeout \leq now$
        $\wedge [(dest = parent(clust) \wedge nbrptup = \perp) \vee dest = nbprtup])))]$
  Effect
    $finding \leftarrow$ false

Trajectories
  Evolves
    $\mathbf{d}(now) = 1$
    All other variables constant.
  Stops when
    Any precondition is satisfied.
```

Fig. 2. $Tracker_{u,lvl}$, where $clust = cluster(u, lvl)$ and $h(clust) = u$.

link, then $p \leftarrow nbrptup$ (inserting a lateral link), the grow is sent to p, and growNbr is sent to neighboring clusters. Otherwise, $p \leftarrow parent(clust)$, the grow is sent up to p, and growPar is sent to neighbors.

Neighbors that receive a growNbr from $clust$ set their $nbrptdown$ pointers to $clust$, indicating $clust$ is connected to the tracking path via a lateral link. Neighbors that receive a growPar set their $nbrptup$ pointers, indicating $clust$ is connected to the tracking path via its hierarchy parent.

2) Shrink action at Tracker$_{u,l}$: A shrink cleans old, deserted branches of the tracking path.

If $clust$ receives a shrink from some $clust'$, it checks if $c = clust'$ (c might not point to $clust'$; it may have been updated to a process on a newer path). If $c = clust'$ then $clust$ removes itself from the path by setting $c \leftarrow \perp$ and, if $p \neq \perp$, sets $timer \leftarrow now + s(l)$, scheduling a shrink to be sent to its tracking path parent p. Otherwise, if $c \neq clust'$, $clust$ ignores the message, ensuring that shrinks clean only deadwood and not the entire tracking path.

When $timer$ expires, if $c = \perp$ still, meaning no new path connected at $clust$ while $timer$ was counting down, $clust$ sends a shrink to p and sets $p \leftarrow \perp$. It also sends each of its neighbors a shrinkUpd. A neighbor receiving the shrinkUpd erases its $nbrptdown$ or $nbrptup$ pointer if it is set to $clust$, removing the defunct secondary tracking pointer.

C. Correctness

We assume executions start in an initial state with no mobile object moves and no outstanding finds, the first move precedes the first find, and there is at least one alive client in a region whenever the object enters or leaves it. We also assume each VSA is alive throughout the execution.

1) Terminology: We start with some useful terminology:

• A move(c_0) occurs when move inputs occur at clients in a region u where $c_0 = cluster(u, 0)$.

•A *path segment* $\{c_x, c_{x-1}, \ldots, c_0\}$ is a cluster sequence s.t.:

1) If $level(c_x) = MAX$, then $c_x.p = \perp$ and $c_x.c \in children(c_x) \cup \{\perp\}$,
2) For each $k \in \{x, \ldots, 1\}$, $c_k.c = c_{k-1} \wedge (c_k.c).p = c_k$,
3) For each $k \in \{x, \ldots, 0\}$, if $c_k.p \in nbrs(c_k)$, then:
    a) If $k \neq 0 \vee level(c_k) > 0$, then $c_k.c \in children(c_k) \cup \{\perp\}$,
    b) If $k = 0 \wedge level(c_k) = 0$, then $c_k.c \in \{\perp, c_k\}$,
4) For each $k \in \{x, \ldots, 0\}$, if $c_k.p = parent(c_k)$, then:
    a) If $k \neq 0 \vee level(c_k) > 0$, then $c_k.c \in children(c_k) \cup nbrs(c_k) \cup \{\perp\}$,
    b) If $k = 0 \wedge level(c_k) = 0$, then $c_k.c \in nbrs(c_k) \cup \{\perp, c_k\}$.

• A *tracking path* $\{c_x, c_{x-1}, \ldots, c_0\}$ is a path segment s.t.:

1) $level(c_x) = MAX$,
2) *Evader* is in region u where $c_0 = c_0.c = cluster(u, 0)$.

• A *consistent state* is a state where:

1) One tracking path $\{c_x, c_{x-1}, \ldots, c_0\}$ exists,
2) For each c_k not in the tracking path, $c_k.c = \perp = c_k.p$,
3) For each c_k, c_n, $c_k.nbrptup = c_n \neq \perp \Leftrightarrow c_n \in nbrs(c_k) \wedge c_n.p = parent(c_n)$,
4) For each c_k, c_n, $c_k.nbrptdown = c_n \neq \perp \Leftrightarrow c_n \in nbrs(c_k) \wedge c_n.p \in nbrs(c_n)$,
5) There are no grow, growPar, growNbr, shrink, shrinkUpd messages in transit or queued.

• The *level of a grow* is defined as $level(m)$, for process m such that $m.c \neq \perp$, $m.p = \perp$, and $level(m) < MAX$, or such that a grow message is in transit to m. If no such m exists, we say the grow is done. We define the level of a shrink in a similar manner, with $m.c = \perp$ and $m.p \neq \perp$.

• A path segment $\{c_x, c_{x-1}, \ldots, c_0\}$ is a *vertical growth to* c_x if for each $k \in \{x-1, \ldots, 0\}$, $c_k.p = parent(c_k)$.

• Function init maps any level 0 cluster c_0 to a consistent state with a tracking path terminating in c_0 that is a vertical growth to level MAX.

• Function atomicMove maps a consistent state with tracking path $\{c_x, c_{x-1}, \ldots, c_0\}$ and a new level 0 cluster location $c'_0 \in nbrs(c_0)$ to a new consistent state with tracking path $\{c'_{x'}, c'_{x'-1}, \ldots, c'_0\}$ such that there exists an index j where:

1) The old and new path share a prefix up to c_j: $\{c'_{x'}, \ldots, c'_0\} = \{c_x, \ldots, c_j\} \cdot \{c'_{j'}, \ldots, c'_0\}$,
2) $\{c'_{j'}, \cdots, c'_0\}$ and $\{c_{j-1}, \cdots, c_0\}$ share no elements,
3) Path segment $\{c'_{j'}, \ldots, c'_0\}$ is a vertical growth to $c'_{j'}$,
4) No cluster in $\{c'_{j'}, \ldots, c'_0\}$ neighbors a cluster c_k in $\{c_{j-1}, \ldots, c_0\}$ such that $c_k.p = parent(c_k)$.

Since $c'_0 \in nbrs(c_0)$, our clustering assumptions imply each cluster below $level(c_j)$ in $\{c'_{j'}, \ldots, c'_0\}$ neighbors some cluster in segment $\{c_{j-1}, \ldots, c_0\}$ such that $c_k.p \in nbrs(c_k)$.

• Derived function atomicMoveSeq maps a sequence of level 0 clusters $\{c_0, c_1, \cdots, c_x\}$ to the consistent state that is the result of starting with consistent state init(c_0), applying atomicMove to the result together with location c_1, applying atomicMove to that result together with c_2, and so on.

2) Proof sketches: Code examination allows us to easily show the following two lemmas:

Lemma 4.1: The number of grow messages from clusters in transit plus $|\{clust \in C | clust.c \neq \perp \wedge clust.p = \perp \wedge level(clust) \neq MAX\}| \leq 1$. Similarly, the number of shrink messages in transit plus $|\{clust \in C | clust.c = \perp \wedge clust.p \neq \perp \wedge level(clust) \neq MAX\}| \leq 1$. □

Lemma 4.2: A grow message is sent laterally at most once per level per move. □

We then show the following lemma, which captures the key to the argument that a grow is never affected by a shrink:

Lemma 4.3: If a grow is in transit from process $clust$ to a neighboring process $clust'$, then $clust'.p = parent(clust')$.

Proof sketch: For contradiction, assume that there is some execution where a grow is forwarded between an $clust$ and $clust'$ as in the lemma statement, but $clust'.p \neq parent(clust')$. Consider the lowest level l in the execution where this happens. By Lemma 4.2, this grow is the only one forwarded between neighbors at this level. Also, no growPar messages could have been sent at this level for this grow (or the grow would have been forwarded to a hierarchy parent instead of a neighbor). Since we had a consistent state before the move, $clust'.p$ must have been $parent(clust')$. If $clust'.p \neq parent(clust')$ now, a shrink arrived at $clust'$

and erased $clust'.p$. A shrink is forwarded from level 0 and takes at least $\sum_{j=0}^{l-1}[s(j)+(\delta+e)p(j)]+s(l)$ time to be forwarded to level l and complete at process $clust'$.

Since $clust'.p \neq parent(clust')$, the time for a grow to be propagated from level 0 to process $clust$, have $clust.timer$ expire, and deliver a grow at $clust'$ must be more than the shrink time above. If the grow is forwarded only to cluster parents in levels below l, the grow time would be at most $\sum_{j=0}^{l-1}[g(j)+(\delta+e)p(j)]+g(l)+(\delta+e)n(l)$. By (1), this is less than the shrink total, so the grow must also have been propagated between neighbors cl and cl' at some level below l. The grow reached level l, so it must not have terminated at cl', implying $cl'.p$ was $\perp$ when the grow arrived, contradicting that l was the lowest level where this could happen. ■

Using the above lemma, the following is easy to show:

Lemma 4.4: Consider a state with a process $clust$ where $c=\perp, p\neq\perp$, and $timer = now$. Either the grow is done or the level of the grow is greater than $level(clust)$ and no grow-related messages are in transit at $level(clust)$. □

Inspection reveals shrinks and grows to be weakly increasing in level. Shrinks are slower than grows, and there are a finite number of pointers per level, so we can show:

Theorem 4.5: Updates terminate after a move. □

To show VINESTALK implements a tracking service, we define function lookAhead that takes a state of VINESTALK and produces a new system state (Figure 3), the "future state", where outstanding grow-related updates have been applied, followed by the shrink-related ones. We then show that future state equals atomicMove's consistent state.

As a first step, we note that lookAhead after the first move returns the same result as init at that region:

Lemma 4.6: From an initial state, apply a move(c_0) input to get state s'. lookAhead(s') = init(c_0). □

Now we show that lookAhead applied immediately after a move equals the consistent state of atomicMove:

Lemma 4.7: Consider the state s' that results from a move(c'_0) input on a consistent state s with tracking path $\{c_x, c_{x-1}, \ldots, c_0\}$, where $c'_0 \in nbrs(c_0)$. lookAhead(s') = atomicMove(s, c'_0).

Proof sketch: lookAhead$(s) = s$, since a consistent state has no messages in transit and has no grow or shrink related actions enabled. When a move occurs, grow and shrink messages are put in transit to level 0 clusters, but no other changes are made to the state, and no other messages are in transit. Inspection of lookAhead shows that in this case, the function returns a consistent state (old pointers and their neighbors' pointers to them are removed after the new grow pointers are added and propagated to neighbors, and the grow and shrink messages are removed from channels).

The grow in lookAhead is propagated to parents in the hierarchy until a $p\neq\perp$ or $nbrptup$ is encountered or level MAX is reached. By definition of a consistent state, a $nbrptup \neq \perp$ will be encountered iff that neighbor has p to its own parent. Hence, lookAhead has the grow travel vertically, setting c, p, and $nbrptup$ pointers, until it reaches a cluster in the old tracking path or neighboring one in the old tracking path that has its p set to its parent, just as in atomicMove. In this case, lookAhead would set p to that neighbor, update corresponding $nbrptdown$ pointers, and be done with the grow, just as in atomicMove. lookAhead will then remove pointers and neighbors' pointers bottom up from the old tracking path until it encounters a place where the c pointer is different, just as in atomicMove. Since it arrives at the same tracking path as in atomicMove and the system state is consistent, lookAhead(s') = atomicMove(s, c'_0). ■

```
function lookAhead($s$: system state): system state :=
  for each growNbr in transit, from a process $clust$ to $clust'$
    $clust'.nbrptdown \leftarrow clust$
    Remove growNbr from message channel
  for each growPar in transit, from a process $clust$ to $clust'$
    $clust'.nbrptup \leftarrow clust$
    Remove growPar from message channel
  for each grow in transit, from a process $clust$ to $clust'$
    $clust'.c \leftarrow clust$
    Remove grow from message channel
  $clust \leftarrow$ process $cl$ where $cl.c \neq \perp \wedge cl.p = \perp$
  while $clust.p = \perp \wedge level(clust) \neq MAX$    %Propagate grow
    if $clust.nbrptup \neq \perp$ then
      $clust.p \leftarrow clust.nbrptup$
      for each $clust' \in nbrs(clust)$
        $clust'.nbrptdown \leftarrow clust$
    if $clust.nbrptup = \perp$ then
      $clust.p \leftarrow parent(clust)$
      for each $clust' \in nbrs(clust)$
        $clust'.nbrptup \leftarrow clust$
    $(clust.p).c \leftarrow clust$
    $clust \leftarrow clust.p$
  for each shrinkUpd in transit, from a process $clust$ to $clust'$
    if $clust'.nbrptup = clust$ then
      $clust'.nbrptup \leftarrow \perp$
    if $clust'.nbrptdown = clust$ then
      $clust'.nbrptdown \leftarrow \perp$
    Remove shrinkUpd from message channel
  for each shrink in transit, from a process $clust$ to $clust'$
    if $clust'.c = clust$ then
      $clust'.c \leftarrow \perp$
    Remove shrink from message channel
  $clust \leftarrow$ process $cl$ where $cl.c = \perp \wedge cl.p \neq \perp$
  while $clust.p \neq \perp \wedge level(clust) \neq MAX$    %Propagate shrink
    for each $clust' \in nbrs(clust)$
      if $clust'.nbrptup = clust$ then
        $clust'.nbrptup \leftarrow \perp$
      if $clust'.nbrptdown = clust$ then
        $clust'.nbrptdown \leftarrow \perp$
    if $(clust.p).c = clust$ then
      $clust \leftarrow clust.p$
      $(clust.c).p$, $clust.c \leftarrow \perp$
    else $clust.p \leftarrow \perp$
  return($s$)
```

Fig. 3. lookAhead function.

Correctness of VINESTALK follows from this theorem:

Theorem 4.8: Given an execution fragment starting at an initial state, ending at some state s, and experiencing the evader move sequence $\{c_0, c_1, \cdots, c_x\}$, where for each $k \in \{1, ldotsx\}$: $c_k \in nbrs(c_{k-1})$, the following holds: lookAhead(s) = atomicMoveSeq$(\{c_0, c_1, \cdots, c_x\})$.

Proof sketch: By induction on executions. Initially this holds since no pointers are set in the system and none are set in the atomicMove object. Next, assume this holds for the execution up to s, and show it still holds at s' after any action occurs. If the action is a move, then the relationship holds by Lemmas 4.6 and 4.7, and by the induction hypothesis.

For non-move actions, we must show lookAhead(s) = lookAhead(s'). We include the most interesting cases here:

• cTOBsend$(\langle$grow, $clust\rangle, par)_{u,lvl}$: For this to be sent, $c \neq \perp$ and $p = \perp$, and lookAhead(s) would have started by delivering any outstanding grow update messages and then simulating effects of this action. The only update message that could affect the outcome of this action is growPar, which is only sent by a process forwarding the grow to a hierarchy parent, meaning it could not be outstanding. Hence, lookAhead(s') would set p and neighbor pointers based on the same values as before, and the result would not change.

• cTOBrcv$(\langle$shrink, $cid\rangle)_{u,lvl}$: This message must have been in the channel in s and accounted for in lookAhead(s). The only way for lookAhead to differ now is if the corresponding grow was to have changed c from cid, or if c was something else and was going to be changed to cid. In the first case, lookAhead(s') will calculate the result based on grows being first again, and the value of c will still be updated to the new value from the grow, and the shrink will not be further propagated, as in s. In the second case, c must have changed since the move due to receipt of a grow, and either $lvl = MAX$ or $p \neq \perp$, since the move started from a consistent state and the shrink just arrived (a grow never changes p unless it was $\perp$, and a shrink is only passed along non-$\perp$ ps). This implies the grow terminated upon arrival, and will not update c back to cid.

• cTOBsend$(\langle$shrink, $clust\rangle, p)_{u,lvl}$: lookAhead(s) accounted for this shrink since $timer$ was running. By Lemma 4.4, the associated grow is either done or at a higher level, and no messages related to it are still in transit at this level, so the state of this process will not be changed due to a grow. Hence, lookAhead(s') will simulate finishing the grows, which won't change since they are at a higher level, then simulate the receipt of the new shrinkUpd and shrink messages, whose effects were modeled in lookAhead(s). ∎

D. Work

To prove work claims, we use the fact that a new path segment connects to the old path at the first process that is an iterated cluster of the new object region, or a neighbor of such a cluster that is not connected via a lateral link:

Theorem 4.9 (Work): Updates for mobile object moves to a total distance d away take amortized work and time:

$$O(d[\omega(0) + \sum_{j=1}^{MAX} \frac{n(j)(1+\omega(j))}{q(j-1)}]), \text{ and}$$

$$O(d[s(0) + \sum_{j=1}^{MAX} \frac{s(j)+(\delta+e)n(j)}{q(j-1)}]).$$

Proof sketch: By our clustering, a level 0 pointer is updated as often as every step. A level 1 pointer is updated as often as every two steps. A level l pointer, $l \geq 2$, is updated as often as every $q(l-1)$ steps, since a level l pointer is only updated after a non-neighboring level $l-1$ cluster is reached.

For $l \geq 1$, $O(p(l-1))$ cost is incurred every time a level l pointer is updated (for shrink and grow propagation from level $l-1$). This can result in $O(\omega(l)n(l))$ communication to update neighbors and propagate a shrink between neighbors. Since $p(l-1) \leq n(l)$, the worst case amortized cost is then: $O(d[\omega(0) + p(0) + n(1)\omega(1) + \sum_{j=2}^{MAX} \frac{p(j-1)+n(j)\omega(j)}{q(j-1)}])$.

The shrink takes longer than the grow by Lemma 4.4. If a level l pointer is updated, this could be associated with a shrink arriving from level $l-1$ (taking $(\delta+e)p(l-1)$ time), a shrink through two neighbors at level l (taking $2s(l) + (\delta+e)n(l)$ time), and then a shrink update to neighbors at the level. Using similar computations to those above, the worst case amortized time for a move is: $O(d[s(0) + \sum_{j=1}^{MAX} \frac{s(j)+(\delta+e)(p(j-1)+n(j))}{q(j-1)}])$. ∎

As a corollary, in the grid hierarchy under the assumption that $s(l) = sr^l$ for some constant s, the amortized work and time is $O(dr \log_r D)$ and $O(dr(s+\delta+e)\log_r D)$.

V. VineStalk: Atomic find operations

Here we describe finds, assuming find operations are executed in consistent states.

If a client receives a find input, it informs its region's VSA with a find broadcast. The level 0 process at the VSA starts servicing the find once it receives the message.

A find over VSAs consists of two phases: *searching* and *tracing*. Searching queries neighboring processes at increasing levels of the hierarchy until a tracking path is found. Tracing follows pointers in the path to the mobile object.

Each $Tracker_{u,l}$ maintains a flag *finding*, indicating if it is currently engaged in a find, and a $nbrtimeout$ timer.

If a find message is received and $c = \perp$, then $clust$ is in the search phase. If $nbrptdown \neq \perp$, then $clust$ forwards find to $nbrptdown$. If $nbrptdown = \perp$ but $nbrptup \neq \perp$, then $clust$ forwards the find to $nbrptup$. (In these cases, $clust$ is forwarding the find along secondary pointers to the tracking path.) If $nbrptup = nbrptdown = \perp$, $clust$ has neither primary nor secondary pointers to the tracking path. Hence, $clust$ sends a findQuery message to its neighbors, and sets $nbrtimeout$ to the roundtrip neighbor communication time. Neighbors answer the query with a findAck message and a pointer to themselves or a neighbor if they are on the tracking path or have a secondary pointer to the tracking path, and ignore it otherwise. If such a findAck is received before $nbrtimeout$ expires at $clust$, $clust$ forwards the find to the process indicated by the findAck. If $nbrtimeout$ expires with no reply from a neighbor, $clust$ forwards the find to its hierarchy parent, $parent(clust)$.

If a find is received and $c \neq \perp$, the find is tracing. If $c \neq clust$, the find is forwarded to c. If $c = clust$, tracing is over, and $clust$ broadcasts found. Clients in that and neighboring regions that receive the found and whose last move indicated the presence of the evader then perform a found output.

Work. Finds are local. To see this, we note the following:

Theorem 5.1: In a consistent state, if region u is at most $q(l)$ distance from the mobile object's region, there is some cluster in $\{cluster(u,l)\} \cup nbrs(cluster(u,l))$ on the tracking path or with a secondary pointer to the tracking path.

Proof sketch: By proximity, the mobile object region's level l cluster is on the tracking path, or has a secondary pointer to a neighbor on the path. If region u is at most $q(l)$ away from the object, $cluster(u, l)$ is either the object's level l cluster or a neighbor. So, $cluster(u, l)$ or a neighbor is on the tracking path or has a secondary pointer to the path. ■

Theorem 5.2: A find input invoked distance d from a mobile object leads to $O(\sum_{j=0}^{l} n(j)(\omega(j) + 1))$ work and $O((\delta + e) \sum_{j=0}^{l} n(j))$ time.

Proof sketch: For searching, at each level j, all $\omega(j)$ neighbors are queried, until the minimum level l such that $d \leq q(l)$, when the path will be found, by Theorem 5.1. With roundtrip communication with each neighbor at each level, we have $\sum_{j=0}^{l-1}[2\omega(j)n(j) + p(j)] + 2\omega(l)n(l)$ work, plus a possible $2n(l)$ to follow a secondary and then primary pointer to the tracking path. For tracing, the cost of following the path to the object is at most $\sum_{j=0}^{l-1}[n(j) + p(j)] + n(l)$. The total cost is then $\sum_{j=0}^{l-1}[2p(j) + n(j)(2\omega(j) + 1)] + 2\omega(l)n(l) + 3n(l)$, or $O(\sum_{j=0}^{l}[(1 + \omega(j))n(j)])$.

The time is $O((\delta + e)(n(l) + \sum_{j=0}^{l-1}[p(j) + n(j)]))$. ■

In the grid, the work is $O(d)$, and time is $O(d(\delta + e))$.

VI. Concurrent operations

We can relax move and find restrictions and consider concurrent execution of move operations, where the mobile object may relocate before updates to the tracking path are complete, and finds. The algorithms for move and find in Figure 2 remain the same, but it is necessary to introduce restrictions on the speed of the mobile object in order to continue to guarantee low-cost find and move operations.

During concurrent move operations, at any given instant, there may be several new paths growing, older deadwood shrinking, and new deadwood being produced. Under restrictions on mobile object speed, we can show that for each move, the triggered grows and shrinks are the same as in the atomic case, with the same time/work cost.

For finds, under mobile object speed assumptions, we can show that a search phase would at worst go up only one more level than in the atomic case. We can also show that in the tracing phase, a find that follows a defunct pointer will be able to make progress down the cluster hierarchy, and perform a found output within a constant factor of the time and work required in the atomic case in a grid hierarchy.

VII. Extensions

We are extending VineStalk to be self-stabilizing. The original Stalk algorithm is self-stabilizing and hierarchy-based fault-containing, preventing propagation of faults in the tracking structure beyond a small number of levels in the hierarchy. It achieves self-stabilization mainly through heartbeats. We also argued that our VSA algorithm in [6] is self-stabilizing: if the physical mobile nodes are started in arbitrary states, then they eventually converge to correct emulation of VSAs. Finally, the geocast service that was the basis of C-gcast used in this paper is also self-stabilizing [10]. Since each of the building blocks of VineStalk are self-stabilizing, the modifications to make it self-stabilizing should involve just minor tweaks to the techniques in Stalk.

We can also try to improve fault-tolerance of VineStalk by allowing multiple heads per cluster. Updates to the tracking path and queries of clusterheads would involve contacting multiple heads for each cluster. This quorum-like approach should result in only an additional constant factor overhead, but would allow for the failure of limited sets of VSAs.

Another extension to consider is to allow multiple finders and mobile objects, with the goal of coordinating the behaviour of the finders so as to minimize the time before all mobile objects are overtaken. VSAs doing the tracking might occasionally send information to data repository VSAs acting as command centers. These centers then direct finders to particular targets to eliminate as much overlap in pursuit as possible. The use of VSAs to coordinate motion of mobile units has been explored in a simple scenario in [15].

Lastly, we can examine the performance degradation that results if mobile objects occasionally move faster than we allow in our analysis. Such moves can result in suboptimal tracking path constructions, but if they occur infrequently enough the structure can still recover to something usable.

References

[1] I. Abraham, D. Dolev, and D. Malkhi. LLS: a locality aware location service for mobile ad hoc networks. *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, 2004.

[2] A. Arora, M. Demirbas, N. Lynch, and T. Nolte. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. *8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.

[3] B. Awerbuch and D. Peleg. Sparse partitions (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, 1990.

[4] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the Association for Computing Machinery*, 42, 1995.

[5] M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. *Symposium on Self-Stabilizing Systems (SSS)*, 2003.

[6] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Brief announcement: Virtual stationary automata for mobile networks. *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[7] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, and T. Nolte. Timed virtual stationary automata for mobile networks. *TR MIT-LCS-TR-979a*, 2005.

[8] S. Dolev, S. Gilbert, N. Lynch, E. Schiller, A. Shvartsman, J., and Welch. Virtual mobile nodes for mobile ad hoc networks. *International Conference on Principles of Distributed Computing (DISC)*, 2004.

[9] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in ad hoc networks. *17th International Conference on Principles of Distributed Computing (DISC)*, 2003.

[10] S. Dolev, L. Lahiani, N. Lynch, and T. Nolte. Self-stabilizing mobile node location management and message routing. *Symposium on Self Stabilizing Systems (SSS)*, 2005.

[11] S. Dolev, D. Pradhan, and J. Welch. Modified tree structure for location management in mobile environments. In *INFOCOM (2)*, 1995.

[12] M.P. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th International Symposium on Distributed Computing*, pages 209–219, October 2001.

[13] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan Claypool, 2006.

[14] J. Li, J. Jannotti, D.S.J. De Couto, D.R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. *Proceedings of Mobicom*, 2000.

[15] N. Lynch, S. Mitra, and T. Nolte. Motion coordination using virtual nodes. *IEEE Conference on Decision and Control*, 2005.

[16] V. Mittal, M. Demirbas, and A. Arora. Loci: Local clustering in large scale wireless networks. *TR OSU-CISRC-2/03-TR07*, 2003.