A Hierarchy-based Fault-local Stabilizing Algorithm for Tracking in Sensor Networks

Murat Demirbas Anish Arora

Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA

Tina Nolte

MIT Computer Science &
Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

Abstract

In this paper, we present STALK*, a hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. Starting from an arbitrarily corrupted state,* STALK *satisfies its specification within time and communication cost proportional to the size of the faulty region instead of the network size. Local stabilization is achieved by slowing propagation of information as the levels of the hierarchy underlying* STALK *increase, enabling the more recent information propagated by lower levels to override misinformation at higher levels. While achieving fault-local stabilization,* STALK *also adheres to the locality of tracking operations: an operation to* find *a mobile object at a distance d away requires $O(d)$ amount of time and communication cost to intercept the moving object, and a* move *of an object to a distance d away requires $O(d*log(\text{network diameter}))$ amount of time and communication cost to update the tracking structure. Furthermore,* STALK *achieves seamless tracking of a continuously moving object by enabling concurrent executions of move and find operations.*

Keywords: Sensor networks, self-stabilization, fault-containment, tracking, distributed data structures.

> "Everything is related to everything else, but near things are more related than distant things".
>
> *Waldo Tobler's First Law of Geography*

This version supercedes OSU-CISRC-4/03-TR19

1 Introduction

Due to applications in mobile computing, cellular telephony, and military contexts, tracking of mobile objects had received significant attention [5, 7, 10, 20, 23]. More recently, the DARPA Network Embedded Software Technology (NEST) program posed tracking as a challenge problem in wireless sensor networks, and several groups have delivered small-scale (using 100 node networks) tracking demonstrations: pursuer-evader tracking with 1 human controlled evader and 3 autonomous pursuers is showcased in [22], and detection, classification, and tracking of various intruders, such as persons and cars, are demonstrated in [2].

Besides the opportunities they provide for tracking of objects, wireless sensor networks also impose additional challenges. Sensor nodes are energy constrained; algorithms that require an excessive communication burden are unacceptable since they drain battery power quickly. Sensor networks are fault-prone, message losses and corruptions and node failures are frequent; nodes can lose synchrony and programs can reach arbitrary states [16]. On-site maintenance is infeasible; sensor networks should be self-healing. Moreover self-healing should achieve fault-containment; otherwise a fault in one region of the network may contaminate the entire network and require a global correction, wasting the energy of the nodes and hindering the availability of the tracking service.

Contributions. Our novel contribution is to present a hierarchy-based fault-local stabilizing algorithm, namely STALK (Stabilizing TrAcking via Layered linKs), for tracking in sensor networks. Starting from an arbitrarily corrupted state STALK satisfies its specification in time and work proportional to perturbation size instead of network size. This implies fault-containment: fault contamination is confined to an area proportional to the perturbation size. We achieve fault-containment by slowing propagation of information as the levels of the hierarchy underlying STALK increase, enabling the more recent information propagated by lower levels to override misinformation at higher levels.

Our scheme for achieving fault-containment does not interfere with the efficiency of tracking operations in the absence of faults. While achieving fault-local stabilization STALK also adheres to the locality of tracking operations: a *find* invoked within distance d of the mobile object requires $O(d)$ time and communication cost (work) to reach the object, and a *move* of the object to distance d away requires $O(d * log(\text{network diameter}))$ time and work to update the tracking structure. Furthermore, STALK achieves seamless tracking of a continuously moving object by enabling concurrent executions of move and find operations.

Overview of STALK. For achieving scalability, STALK employs a hierarchical structure. For ensuring the locality of both find and move operations, STALK adopts a partial information strategy. The tracking information is maintained with accuracy related to the distance from the mobile object: Nearby nodes that are relatively cheap to update have more recent and accurate information about the object, whereas far away nodes that are relatively expensive to update have older and more approximate information about the object.

Tracking structure. We assume a hierarchical partitioning of the sensor network into clusters based on radius. The tracking structure is a path rooted at the highest level of the hierarchy. Each process in the *tracking path* has at most one child, either at its level or one below it in the hierarchy, and the mobile object resides at the leaf of the tracking path, at the lowest level. Each process in the path points to a process that is generally closer to the object and has more recent information about its location.

Find operation. 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 tracking path is found, the find operation follows it to its leaf to reach the mobile object.

Move operation. We implement move-triggered updates by means of two local actions, *grow* and *shrink*. The grow action enables a path to grow from the new location of the object to increasingly higher levels of the hierarchy and connect to the original path. The shrink action cleans branches deserted by the object. Shrinking also starts at the lowest level and climbs to increasingly higher levels. Despite that grow and shrink occur concurrently, we achieve the move operation successfully by using suitable values for the process timers, which actuates the execution of these actions.

Fault-local stabilization. We use two concepts

for achieving fault-locality: hierarchical partitioning and level-based timeouts for execution of actions. The key idea is to wait for more time before updating a wider region's view. We employ larger timeouts when propagating an update to a higher level of the hierarchy, and thus, more recent updates coming from lower levels can catch-up to misinformed updates at higher levels. The latency imposed by waiting is a constant factor of the communication delay and does not affect the accessibility of the tracking structure.

Concurrent move and find operations. STALK achieves seamless tracking of a continuously moving object: An object can relocate before the effects of its previous move operations finish updating the tracking path, and a find operation may be concurrently in progress with these move operations. During concurrent move operations, it is not possible to achieve a complete tracking path; there will be discontinuities in the path. By giving an upperbound on the speed of the object, we prove a reachability condition on the tracking path and ensure that if a find encounters a dead-end while following a path, there is always an available newer path nearby.

Related work. STALK provides a "network middleware support" for tracking: it assumes an underlying service for detection of a mobile object [15,17,25] and provides a basis for higher level applications such as multiple target tracking [21] and pursuer-evader applications [12].

The idea of employing a hierarchical structure for achieving scalability of tracking has been extensively researched [1, 24] in the context of personal communication systems and mobile Internet Protocol, and the idea of using a partial information strategy to optimize both finds and moves has been investigated in [5,8].

In [5], a hierarchy of regional directories is constructed so that each level l directory enables a node to find a mobile object within 2^l distance from itself. 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). 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 [4] that constructs a sparse cover of a graph. In [8], the tracking problem is considered for a geometric network model similar to ours, and cost complexity similar to ours is achieved[1].

STALK offers properties that these protocols lack, such as fault-tolerance and seamless tracking of a continuously moving object. STALK is not only fault-tolerant but also stabilizing and fault-containing as well: Starting from an arbitrarily corrupted state, STALK recovers within work and time proportional to the size of the faulty region. STALK achieves seamless tracking of continuously moving objects: An object can relocate before the effects of its previous move operations finish updating the tracking path, and a find operation may be concurrently in progress with these move operations.

There has been some work on self-stabilizing tracking algorithms [9,10,13]. The distributed arrow protocol [13] suffers from the dithering problem —where an object moving back and forth across a multi-level hierarchy boundary may lead to nonlocal updates. The protocols in [9] do not exploit the hierarchy idea and are not scalable for large networks. In [10], using a hierarchy of location servers, a stabilizing location management protocol is presented. However, in contrast to STALK, the protocol in [10] fails to ensure locality of finds. Also, none of these protocols enjoy fault-containment.

The area of fault-containment of self-stabilizing algorithms has received growing interest [3, 6, 11, 19]. The notion of fault containment within the context of stabilization is formalized first in [11]; algorithms were proposed to contain state-corruption of a single node in a stabilizing spanning tree protocol. In [19] fault-containment of Byzantine nodes have been studied for dining philosophers and graph coloring algorithms; this work requires the range of contamination to be constant and is too limiting for problems such as tracking and routing whose locality are not constant. In [6], a broadcast protocol is proposed to contain observable variables in the presence of state corruptions, but the protocol allows for global propagation of internal protocol variables.

A protocol that achieves fault-local stabilization

[1]The move operation in [8] costs O(d * log d) work (where d is the distance moved by the object), but their interpretation of "amortized cost" is more permissive than ours. Using the same interpretation for "amortized cost", the move operation in STALK also costs O(d * log d) work.

in shortest path routing is presented in [3]. To achieve fault-containment the protocol uses containment actions that are a constant time faster than the fault-intolerant program actions. In contrast to [3], we do not have a privileged set of containment actions in STALK; the program actions serve to this end. We enable fault-containment in a hierarchy-based manner by suitably varying the speed of actions (through the use of process timers) as per the level of the hierarchy they are executed at.

Organization of the paper. After presenting the model in the next section, we present specifications of STALK in Section 3. In Section 4, we present the move operation. Fault local stabilization of the tracking path is discussed in Section 5. The find operation is in Section 6. In Section 7 we discuss concurrent execution of move operations, where the mobile object may relocate while previous move operations are still updating the tracking structure. In Section 8 we consider execution of find operations while moves are concurrently updating the tracking structure. Finally we conclude our paper in Section 9. For space reasons, we relegate code and detailed proofs to the Appendix.

2 Model

We consider a sensor network consisting of multiple sensor locations. Each sensor location plays host to (possibly) multiple processes with identifiers from a set P. In this paper, as a convention, i and j refer to process identifiers, and $i.x$ refers to the value of variable x at i.

We denote the location of a process i with $loc(i)$ (and for convenience the set of locations of process set I with $loc(I)$). The Euclidean distance between the locations of i and j is denoted by $dist(i, j)$.

Hierarchical partitioning. Assume a hierarchical partitioning of processes over locations. Consider a tree with levels 0 through MAX of all processes P. For each process i we define:

1. $lvl(i)$, the level of process i in the tree,
2. $h(i)$, i's parent in the tree (for convenience, we define $h(i)$ to be i if $lvl(i) = MAX$),
3. $h^n(i)$, the iterated parent, defined as $h(i)$ if $n = 1$ and $h(h^{n-1}(i))$ otherwise,
4. $children(i)$, i's children in the tree.

We assume a one-to-one correspondence between the level 0 processes in the tree and sensor locations. For a location v we denote the level 0 process residing at v as $proc_0(v)$. We also assume that for any i such that $lvl(i) > 0$, i's location $loc(i)$ is equal to $loc(j)$ of one of its children j.

This partitioning yields *clusters*. For i such that $lvl(i) = k+1$, $0 \leq k < MAX$, $children(i)$ together form a cluster C at level k whose clusterhead, $head(C)$, is i. $Radius(C)$ is the maximum distance from $head(C)$ to any process in C.

Next we introduce the symmetric neighbor relation. For level 0 processes i, j, $i \neq j$, $j \in nbr(i) \iff dist(i, j) \leq 1$. For level $k > 0$ processes i, j, that are clusterheads of level $k-1$ clusters C_i and C_j, i and j are neighbors if C_i and C_j contain two processes that are neighbors.

Geometry assumptions. We fix the following assumptions about the hierarchical partitioning:

1. We define a real constant $r \geq 3$ to denote the cluster dilation factor; the radius of a level l cluster is at least r^l,
2. We define a real maximum cluster radius constant $m \geq 2/\sqrt{3}$ to bound the radius of a level l cluster to be at most mr^l,
3. We define a real minimum cluster breadth constant q satisfying $\frac{2m+r-1}{r-1} \leq q \leq 2m$ to restricts the locations in *non-neighboring* level l clusters to be greater than qr^l apart.

The constraints imply a bound, ω, on the number of neighbors at any level $l > 0$. They also imply that, for $l > 0$, the distance between two neighboring level l processes is within $2r^{l-1}$-to-$2mr^{l-1}$, and the distance between a level l process and its children in the hierarchy is at most mr^{l-1}. This clustering does not necessarily imply a uniform tiling of the network, as radii of clusters at the same level are not required to be the same. The network diameter, D, is the maximum distance between any two locations in the network. Each node in the network is deployed with $O(MAX)$ storage where $MAX \leq log_r D$.

An example of the clustering geometry with $r = 3$ can be found in Section 4. Our hierarchical partitioning constraints can be realized by using a distributed and fault-local stabilizing clustering protocol, LOCI [18].

3 System specification

Here we describe the specification for STALK (modeled as IO automata in the Appendix).

Mobile object. The mobile object **Evader** resides at exactly one sensor location. An **object**$_i$

occurs at all processes residing at the object's current location and **no_object**$_j$ occurs for all other locations. When moving, the object nondeterministically moves to a neighboring location.

STALK. STALK consists of two parts, **Tracker** and **Finder**, as seen in Figure 1. **Tracker** maintains a tracking structure by propagating mobile object information obtained through **object** and **no_object** inputs. **Finder** answers client **find**s by outputting **found** at the mobile object's current location. **Finder** queries **Tracker** for location information through **cpq** requests and **Tracker** answers with **cpointer** responses.

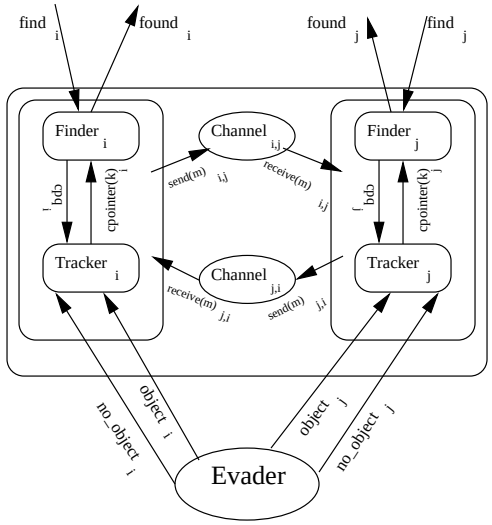


Figure 1. STALK **Architecture**

Both parts are implemented distributedly by individual processes communicating through channels. Each process is assumed to have access to its own local timer, that advances at the same rate at all processes. We do not assume time synchronization across processes.

Channels. We use a communication abstraction of a (possibly) multi-hop channel **Channel**$_{i,j}$ between any two processes i and j. Such channels are accessed using **send(m)**$_{i,j}$ to send from i and **receive(m)**$_{i,j}$ to receive at j. The cost of sending a message through **Channel**$_{i,j}$ is $dist(i,j)$, and in the absence of faults a message is removed from the channel by at most $\delta * dist(i,j)$ time where δ is a known message delay factor.

Fault model and tolerance specification. Processes can suffer from arbitrary state corruption. These faults may occur at any time and in any finite number and order. Channels may suffer faults that corrupt, manufacture, duplicate, or lose messages.

We say a system is *self-stabilizing* iff starting from an arbitrary state the system eventually recovers to a consistent state, a state from where its specification is satisfied. In Section 4 we characterize consistent states for our implementation.

A perturbation count for a given system state is the minimum number of processes whose state must change to achieve a consistent state of the system. For work and time calculations the level of "perturbed" processes are important; a fault hitting a level l process affects the entire level l cluster and hence its size is r^l. We define *perturbation size* to be a weighted sum of the levels of perturbed processes. A stabilizing system is *fault local stabilizing* if the time and work required for stabilization are bounded by functions of perturbation size rather than system size.

Complete system. The complete system is the composition of all channels, **Evader** and STALK.

We require the system be fault-local stabilizing to a consistent state. Starting from a consistent state with no outstanding **find** requests and no process or channel corruptions, we require that:

1. A **find** is eventually followed by a **found** at a location hosting the mobile object,
2. Each **found** is in response to a prior unanswered **find**,
3. If a **find** is initiated at a process Euclidean distance d from the mobile object, the time and work (communication) performed to service it is at most $O(d)$,
4. If the object moves d distance, the amortized time and work to update the tracking structure is $O(d * log(D))$.

4 Tracker

Here we describe how **Tracker** updates the tracking path after a move, assuming that the mobile object does not relocate until the updates are completed. In Section 7, we relax this restriction and allow the object to relocate while effects of its previous moves are still rippling through the 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 inevitably results in multi-level cluster boundaries: even though two processes are neighbors they 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 cluster boundary could result in work proportional to the size of the network rather than the distance of the move. To resolve this "dithering" problem, we allow one *lateral link* per level in our tracking path. A process occasionally connects to the original path with a lateral link to a neighboring process rather than by propagating a link to its parent in the hierarchy. We limit the lateral link count per level in order not to upset the locality properties of the find operation.

To implement **Tracker**, each process i maintains a child pointer c, a parent pointer p, a grow timer $gtime$, and a shrink timer $stime$. In the initial states, $i.c = i.p = \perp$ and $i.gtime = i.stime = \infty$ for all i. We assume the use of grow and shrink constants g and s that satisfy:

$$s \geq 10.5\delta m \quad (1)$$

$$\frac{s + \delta m}{r} < g \leq s - \delta m \quad (2)$$

A grow or shrink timer is set at i for $g * r^{lvl(i)}$ or $s * r^{lvl(i)}$ time respectively. The values for the timers are chosen to satisfy the requirements on both the work calculations in Section 4.4 and the fault-containment proofs in Section 5.

$\mathbf{Tracker}_i$ has four inputs: $\mathbf{object}_i$, $\mathbf{no\_object}_i$, $\mathbf{cpq}_i$, and $\mathbf{receive}(msg)_{j,i}$, and two outputs: $\mathbf{cpointer}(j)_i$ and $\mathbf{send}(msg)_{i,j}$ (msg can be gquery, ack_gquery, grow, or shrink).

$\mathbf{Tracker}_i$ answers a $\mathbf{cpq}_i$ input (an information request from $\mathbf{Finder}_i$) with a $\mathbf{cpointer}(i.c)_i$ output, providing the value of its child pointer. The **send**s and **receive**s propagate grows and shrinks as explained in detail below for process i.

4.1 Grow action

A grow updates a path to point to the new location of the object.

If i is at level 0, the object is at the same location as i, and i's child pointer c does not point to itself, then i becomes the leaf of the tracking path by setting c to i and setting its grow timer, $gtime$, scheduling a **grow** to be sent when $gtime$ expires.

If i is above level 0 and receives a **grow** message, it sets its c pointer to the sender, sets $gtime$ scheduling a **grow** to be sent to its prospective parent. i also sends a **gquery** message to its neighbors to check if the tracking path is reachable through a neighbor. The tracking path allows the use of one lateral link per level. A neighbor j that receives the **gquery** sends an **ack_gquery** back if j is on the tracking path and there isn't already a lateral link pointing to j, i.e., if $j.p$ points to its own clusterhead, $h(j)$. If i receives such an **ack_gquery** from j then it sets p to point to j, in preparation for adding a lateral link at j.

When $gtime$ expires, if c is still non-$\perp$, meaning that the path has not shrunk while i's grow timer was counting down, then a **send (grow)** is performed to extend the tracking path. If $i.p$ points to a neighbor j then the grow message is sent to j, inserting a lateral link. Otherwise, if $p = \perp$, i sets p to point to its own clusterhead $h(i)$ and sends a **grow** message to $h(i)$, propagating the grow one level up in the hierarchy. In either case $gtime$ is set to ∞, and i's role in updating the tracking path is complete.

If a **grow** message is received at i but i already has a parent in the tracking path or is the MAX level process, then i does not propagate the grow (it is already on the tracking path).

4.2 Shrink action

A shrink cleans old, deserted branches of the tracking path.

If i is at level 0 and has a non-$\perp$ child pointer, but the mobile object is not at i's location, then i removes itself from the leaf of the tracking path. It sets its child pointer c to $\perp$ and sets the shrink timer $stime$, scheduling a **shrink** to be sent upon expiration of $stime$.

If i receives a **shrink** message from another process j, i checks to see whether its child pointer c points to j (c might not point to j; it may have been updated to point to a process on a newer path). If $c = j$ then i removes itself from the path by setting c to $\perp$ and then sets its shrink timer, scheduling a **shrink** message to be sent to its parent p. Otherwise, if $c \neq j$, i ignores the message, ensuring that shrink actions clean only deadwood and not the entire tracking path.

When $stime$ expires, if c is still $\perp$, meaning no newer path has connected at i while $stime$ was

counting down, i sends a **shrink** message to its parent p in the path and then sets p to $\perp$.

Example. Figure 2 depicts a sample tracking path. The path is seen pointing to a level 2 clusterhead, which points to one of its hierarchy children, a level 1 clusterhead. That clusterhead has a lateral link to another level 1 clusterhead that points to a level 0 cluster where the object **e** is located. Deadwood is denoted by the dotted path.

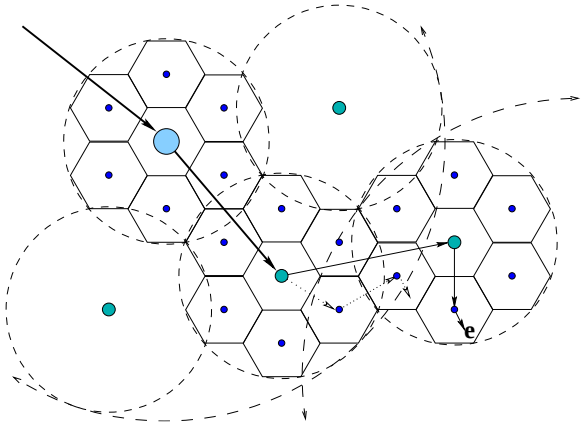


Figure 2. Tracking path example

4.3 Correctness

Here we present system invariants and define consistent states of the system.

In the absence of faults, every process i satisfies I, the following five conditions, at all times:

I0. If $lvl(i) = 0$ and $\mathbf{object}_i$ occurs then $i.c = i$,

I1. If $i.c \neq \perp$ then one of the following holds:
(a) $i.c = i$ and the object is at i,
(b) $i.c$ points to one of its children in the clustering hierarchy, or
(c) $i.c$ points to a neighbor and $i.p$ points to its parent in the clustering hierarchy,

I2. If $i.p \neq \perp$ then either $i.c \neq \perp$ or i is executing a shrink action and will send a **shrink** to $i.p$,

I3. The dual: if $i.c \neq \perp$ then $i.p \neq \perp$ or i is executing a grow action and will send a **grow** to its prospective parent,

I4. If $i.c \neq i$ and $i.c \neq \perp$ then $(i.c).p$ is either i or $\perp$. In the latter case a **shrink** from $i.c$ is in transit to i. □

A *tracking path* is a sequence $\{i_x, \ldots, i_1\}$ where

1. i_1 is a leaf and contains the object,
2. Every process but i_1 points to the next process as its child, and
3. I is satisfied at all processes in the sequence.

A *complete tracking path* is a tracking path $\{i_x, \ldots, i_1\}$ where $lvl(i_x) = MAX$ and $i_x.p = i_x$.

A *consistent state* is a state where a complete tracking path exists and $i.c = i.p = \perp$ for every process i not in the tracking path.

Using invariant I it follows from the program actions that an execution starting from an initial state eventually reaches a consistent state (Lemma 4.6 in Appendix) and that consistent states are closed under moves of the object (Theorem 4.7).

4.4 Work

In order to prove our work claims, we must show that the timing of changes to the new and old tracking paths satisfy certain relationships to ensure that the old path is reused (via insertion of a lateral link) to the extent possible. More specifically, it follows from the assumptions on timer constants s and g that an old path being cleaned bottom-up from level 0 will not clean one of its level l pointers before a grow starting at level 0 in the new path reaches level l and has an opportunity to query one of those pointers, allowing for the addition of a lateral link.

This allows us to reason that the new path (which grows by propagating pointers straight up the hierarchy until it connects to the old path) connects to the pre-shrink old path at the lowest level process that is either an iterated clusterhead of the new object location or a neighbor of such a clusterhead that is not itself connected to the tracking path via a lateral link. In the latter case, the new path would connect via a lateral link.

We then prove the following theorem.

Theorem 4.10 *Starting from a consistent state, move operations of the mobile object to a total of distance d away require at most $O(d*\omega mr*MAX)$ amortized work and $O(d*gr^2*MAX)$ amortized time to update the tracking path.*

Proof sketch. The above reasoning implies a level l pointer in the path is updated as often as every $\sum_{j=1}^{l-2} qr^j$ distance because of the required use of lateral links at all levels below l (note that qr^l is the minimum distance between two non-neighboring level l clusters). An $O(mr^{l-1})$ work and $O(gr^l)$ time cost is incurred each time a level l pointer is updated. The costs, multiplied by frequency of updates, are summed for each level for the result. □

5 Fault-containment

After state corruption of a region of (potentially all) processes, our tracking path heals itself in a fault-local manner within work proportional to perturbation size. Here we discuss correction actions enabling fault-local stabilization of the path.

Through faults a shrink action can be mistakenly initiated. For example, when a portion of a tracking path is hit by faults, higher level processes of the path, unaware a healthy lower path exists, start a shrink action. If "growth" at lower levels lags behind "shrinking" of upper levels, faults can propagate through the entire upper path. For fault-containment, grow actions started at lower levels must contain shrink actions.

Similarly, grow actions can be mistakenly initiated. Consider a garbage path with no object at its leaf. The topmost process of this path, unaware that the path does not lead to the object, starts a grow action. If "shrinking" from lower levels lags behind "growing" of upper levels, faults can contaminate the entire network. Thus shrinks started at lower levels must contain grows.

The above requirements are both satisfied by giving priority to actions with more recent information regarding the path; actions from lower levels are privileged over ones at higher levels. We achieve this by delaying shrink/grow for longer periods as the level of the process executing the action increases. This way, propagation actions coming from below are subject to lesser delays and can arrest mistakenly initiated propagation actions; fault-local stabilization is achieved. We note that the latency imposed by delaying is a constant factor of the communication delay to higher levels and does not affect the quality of tracking.

Stabilization. Here we present correction actions for re-establishing the tracking path invariant I starting from an arbitrarily corrupted state.

Correction of $I0$ and $I1$. $I0$ is established trivially by **object** and **no_object** inputs. The correction of $I1$ follows from the domain assumptions we make on non-$\perp$ c, p and $gnbrquery$ variables for $i \in P$. We require that $i.c \neq \perp \Rightarrow i.c \in \{nbr(i) \cup children(i)\}$: $i.c$ points to either a neighbor of i or to a child of i. Similarly, we restrict the domain of non-$\perp$ $i.p$ variables to $\{nbr(i) \cup \{h(i)\}\}$ and $i.gnbrquery$ to subsets of $nbr(i)$. These assumptions are reasonable since the clustering provides a process with the identifiers of its neighbors, children, and clusterhead; a process can locally check and set these variables to $\perp$ if their values are outside their respective domains.

Correction action for $I2$. If i has a valid parent but no valid child, then $I2$ is corrected at i by setting $i.c = \perp$ and scheduling a **shrink** message to be sent to $i.p$.

Correction action for $I3$. If i has a valid child but no parent, then a **gquery** message is sent to i's neighbors and a **grow** message is scheduled to be sent to the future parent of i.

Correction actions for $I4$. To correct $I4$ we use heartbeat messages and two timers: *next* for periodically sending heartbeats to the parent and a *timeout* for dissociating a child if no heartbeat is heard. The correction actions use a constant b for calculating the frequency of heartbeat messages, whose periodicity are tunable to achieve less communication or faster detection. We require that b is more than twice s, the shrink timer constant:

$$b \geq 2s \tag{3}$$

Intuitively, this condition serves to prevent a scenario where aggressively scheduled heartbeats shrink the original path before a new growing path can reconnect to the original.

Every i with a non-$\perp$ valued parent sends a **heartbeat** message to its parent every $b * r^{lvl(i)}$ time by setting *next*. Every time i receives a **heartbeat** or **grow** message from its child, $i.c$, i resets its *timeout* variable to $(b+2\delta m/r)*r^{lvl(i)}$ (it is also reset upon receipt of a **grow** to prevent the scenario where the heartbeat timeout of i expires scheduling a shrink just after i receives a **grow** message from a process in a newly growing path). If i receives a heartbeat from j but $i.c = \perp$ then i sets $i.c := j$. Otherwise, a **heartbeat** message received from a process other than $i.c$ is ignored.

If i has a non-$\perp$ valued child, is not a leaf, and has not received a **heartbeat** message in a $(b + 2\delta m/r) * r^{lvl(i)}$ time interval, then $i.c$ is set to $\perp$.

Stabilization of the *next* and *timeout* variables of the corrector is ensured by keeping their values within their respective domains.

Using the correction actions described above, we prove in Theorem 5.2, that STALK is self-stabilizing to a consistent state, where a complete tracking path exists.

Theorem 5.2 STALK *is self-stabilizing.* □

Fault-local stabilization. To prove fault-local stabilization we first give a bound on arresting distance of grow/shrink actions in Lemmas 5.3 and 5.4. In these lemmas, l_1+1 and l_2 are respectively the lowest and highest perturbed levels: faults occur only from level l_1+1 through level l_2. We prove fault containment by showing that due to our timing assumptions, a correction propagated from l_1 catches propagation of bad information at a level $l > l_2$, leaving levels above l untouched by faults. The proof is done by comparing the maximum time the propagation of a lower wave takes to reach l versus the minimum time the higher wave takes to pass it.

Lemma 5.3 *Propagation of a shrink action started at level l_1+1 catches propagation of a grow action started at level l_2 by level l where*

$$l = l_2 + \lceil \log_r \tfrac{br-b+sr+gr-2s+3\delta m}{gr-s-\delta m} \rceil. \quad \square$$

Lemma 5.4 *Propagation of a grow action started at level l_1 catches propagation of a shrink action started at level l_2 by level l where*

$$l = l_2 + \lceil \log_r \tfrac{br-b+sr^2-gr-\delta m}{sr-gr-3\delta m} \rceil. \quad \square$$

The size, $l - l_2$, of contamination due to fault propagation is independent of the network size and is tunable via grow and shrink timer settings. In Section 7 we give sample values for these.

Finally, the above two lemmas allow us to prove the following theorem:

Theorem 5.5 (Fault-local stabilization) *For a perturbation size S, our program self-stabilizes in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.* $\square$

6 Finder

Here we describe **Finder** assuming find operations are interleaved with move operations. We relax this restriction in Section 8 and allow the object to relocate while a find is in progress.

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

A client initiates the operation with a **find** input. The level 0 process at that location starts servicing the find.

A find is serviced at a process i by first querying the local $\mathbf{Tracker}_i$ using $\mathbf{cpq}_i$. $\mathbf{Tracker}_i$ will then return its child pointer c' through $\mathbf{cpointer}(c')_i$.

If $c' = i$, the object is found at i and the tracing phase is over, so i outputs $\mathbf{found}_i$.

If $c' \neq i$ and $c' \neq \perp$, the tracing phase is continuing, and i sends a **find** to process c'.

If $c' = \perp$ it is still the search phase, and i sends an **fquery** message to its neighbors and sets a timeout equal to the maximum time for roundtrip neighbor communication at $lvl(i)$. Neighbors answer the query with an **fqack** message and start servicing the find if they are on the tracking path, and ignore it otherwise. If such an **fqack** is received before the timeout period expires at i, i knows the tracking path has been found and tracing has started at j; i is done. If the timeout period expires with no reply from a neighbor, the search phase is continuing. In this case i sends a **find** to its clusterhead and hands over the responsibility for servicing the find to $h(i)$.

Work. Finds are local: a find initiated at process i distance d from the mobile object requires $O(d)$ work to complete. To see this we first note that geometry assumptions imply:

Theorem 6.1 (Proximity) *In a consistent state, for a process j that is at most d distance from the mobile object, one of the following holds:*

- *$h^{\lceil log_r d \rceil + 1}(j)$ is in the tracking path or*
- *$\exists i \in nbr(h^{\lceil log_r d \rceil + 1}(j))$ in the tracking path.*

Proof sketch. Say $d = r^l$. For this theorem to be false, it must be that level l cluster j is represented by does not neighbor any level l cluster in the tracking path, implying the distance between j and any process represented by a level l cluster on the tracking path is more than qr^l. However, using the fact that there is at most 1 lateral link per level and that the maximum radius of a level l cluster is mr^l, we can conclude that the distance between the leaf of the tracking path and any process represented by a level l cluster in the tracking path is at most $\sum_{j=0}^{l-1} 2mr^j$. This plus the distance r^l is less than qr^l, by assumptions in Section 2. $\square$

Theorem 6.2 *A* find *operation invoked at distance d from a mobile object results in $O(d * \omega rm)$ work and takes $O(d * \delta rm)$ time.*

Proof sketch. The previous theorem implies a find operation will find the path by level $\lceil log_r d \rceil +$

1. We add this cost of searching to the cost of following tracking path links from that level. □

7 Concurrent move operations

In this section we relax the atomic move restriction and consider concurrent execution of move operations, where the mobile object may relocate while effects of previous move operations are still rippling through the tracking structure.

We showed that a complete tracking path was preserved by atomic move operations. However, during concurrent move operations, we can not guarantee a complete tracking path: at any given instant, there may be a new path growing, older deadwood shrinking, and new deadwood being produced. Hence, we provide a looser definition of a *tracking structure* consisting of several path segments that satisfy a reachability condition.

A *path segment* is a piece of a tracking path. The piece is maximal in that the first process in the segment has no parent pointer or has a parent pointer to itself (for the topmost level of hierarchy) and the last pointer in the segment (the endpoint) points to itself or a process without a pointer. A sequence of path segments $\{\{i_{x,y_x}, \ldots, i_{x,1}\}, \ldots, \{i_{1,y_1}, \ldots, i_{1,1}\}\}$ is a *tracking structure* if $i_{1,1}$ contains the object and every endpoint i, s.t. $i = i_{y,1}, y \neq 1$, satisfies a 3-part *reachability condition*: *(1)* If $i.c$ is i's hierarchy child, $lvl(i) > 1$ implies the next path segment contains a neighbor of i, and $lvl(i) = 1$ implies the next segment contains a process neighboring $i.c$. *(2)* If $i.c$ is i's neighbor, the next segment contains a process neighboring $i.c$. *(3)* If $lvl(i) > 1$, the next segment's endpoint is at least 2 levels below $lvl(i)$.

A *complete tracking structure* is a tracking structure that reaches the top level of the hierarchy. We also define a weaker version of a consistent state: A *good state* is a program state where a complete tracking structure exists and $i.c = i.p = \perp$ for all processes i not in the tracking structure.

We assume the object takes at least e time at a level 0 process (that is, within the coverage area of its sensor) before moving to a neighboring level 0 process, and the minimum time the object takes to move a total of d distance is $e * d$ where

$$e \geq 2sr^3 \tag{4}$$

Theorem 7.1 *Starting from a good state, a move of the object leads to another good state.*

Proof sketch. The reachability condition is implied because the time a mobile object takes moving far enough to require a level $l-2$ update and then propagating a shrink to remove the level $l-2$ pointers is more than the time to delete level l pointers in a prior segment. □

The following two theorems have proofs very similar to those of the non-concurrent case.

Theorem 7.4 *Starting from a good state, object moves to distance d away take $O(d*\omega rm*MAX)$ work and $O(d*gr^2*MAX)$ time to complete.*

Proof sketch. Newer segments do not outgrow older segments so Theorem 4.10 still holds. □

Theorem 7.6 (Fault-local stabilization) *For concurrent moves and perturbation size S the system self-stabilizes to a good state in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.* □

Sample timer constants. Consider $g = 5\delta m, s = 11\delta m, e = 23\delta mr^3$, and $b = 11\delta mr$. Tracking structure inequalities are satisfied, grow actions catch faulty shrink actions in 2 levels, and shrinks catch faulty grows within 4 levels.

8 Concurrent find and move operations

Given our prior timing assumptions, find operations are successful even when move operations are still in progress on the tracking structure.

In the searching phase a find invoked within d distance of a mobile object hits the tracking structure by level $\lceil log_r d \rceil + 1$ as before; the object can not move fast enough to result in a propagation of a shrink to level $\lceil log_r d \rceil + 1$ before the find operation gets there.

In a tracing phase that is concurrent with a move, a complete tracking path may not be available, and a find may reach a process with $c = \perp$ while tracing the tracking structure. If a find reaches such a dead end it re-executes the searching phase. The reachability condition of the tracking structure ensures the find will reach a newer path segment by searching neighboring processes at the current level or one level higher. The mobility of the object only results in a constant factor difference in time and work to complete a find.

Theorem 8.4 *A find operation invoked within d distance of a mobile object requires $O(d\omega rm)$ work and $O(d\delta rm)$ time to reach the object.* □

9 Concluding remarks

We presented STALK, a fault-local stabilizing tracking service for sensor networks. We use two concepts to achieve fault locality: hierarchical partitioning and level-based timeouts for execution of actions. The key idea is to wait longer before updating a wider region's view by employing larger timeouts when propagating an update to higher levels of the hierarchy. This way, more recent updates from lower levels can catch-up to and override the misinformed updates at higher levels. While achieving fault-local stabilization STALK also adheres to the locality of tracking operations. Moreover, by enabling concurrent move and concurrent find operations STALK achieves seamless and continuous tracking of the mobile object.

In this paper we focused on the analytical worst-case performance of STALK. In the appendix, we provide simulations for the average-case performance of STALK by considering a random movement model for the object. Simulation results show that work for move scales better than linearly due to the locality of movements featured by the random model. The code is available at `www.cse.ohio-state.edu/~demirbas/track/`.

STALK has applications in message routing to mobile units and in pursuer/evader games. As part of our efforts to develop sensor network services in the DARPA/NEST program, we are implementing STALK on the Mica mote platform [14]. For future work, we are examining other problems that could benefit from our hierarchy-based local stabilization technique.

References

[1] I.F. Akyildiz, J. McNair, J.S.M. Ho, H. Uzunalioglu, and W. Wang. Mobility management in next-generation wireless systems. *Proceedings of the IEEE*, 87:1347–1384, 1999.

[2] A. Arora and et. al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *To appear in Computer Networks (Elsevier)*, 2004.

[3] A. Arora and H. Zhang. LSRP: Local stabilization in shortest path routing. In *IEEE-IFIP DSN*, pages 139–148, June 2003.

[4] B. Awerbuch and D. Peleg. Sparse partitions (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[5] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the Association for Computing Machinery*, 42:1021–1058, 1995.

[6] Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *ACM PODC*, pages 33–42, 2003.

[7] A. Bar-Noy and I. Kessler. Tracking mobile users in wireless communication networks. In *INFOCOM*, pages 1232–1239, 1993.

[8] Y. Bejerano and I. Cidon. An efficient mobility management strategy for personal communication systems. *MOBICOM*, pages 215–222, 1998.

[9] M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. *Sixth Symposium on Self-Stabilizing Systems(SSS'03)*, 2003.

[10] S. Dolev, D. Pradhan, and J. Welch. Modified tree structure for location management in mobile environments. In *INFOCOM (2)*, pages 530–537, 1995.

[11] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *ACM PODC*, pages 45–54, 1996.

[12] L. J. Guibas. Sensing, tracking, and reasoning with relations. *IEEE Signal Processing Magazine*, March 2002.

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

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, pages 93–104, 2000.

[15] Y. H. Hu, D. Li, K. Wong, and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), March 2002.

[16] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *ACM Symposium on Principles of Distributed Computing*, 1996.

[17] J. Liu, P. Cheung, F. Zhao, and L. J. Guibas. A dual-space approach to tracking and sensor management in wireless sensor networks. *MOBICOM*, pages 131–139, 2002.

[18] V. Mittal, M. Demirbas, and A. Arora. LOCI: Local clustering in large scale wireless networks. Technical Report OSU-CISRC-2/03-TR07, The Ohio State University, February 2003.

[19] M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults. In *IEEE SRDS*, pages 22–31, 2002.

[20] E. Pitoura and G. Samaras. Locating objects in mobile computing. *Knowledge and Data Engineering*, 13(4):571–592, 2001.

[21] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target identities in wireless ad-hoc sensor networks. *IPSN*, April 2003.

[22] B. Sinopoli, C. Sharp, L. Schenato, S. Schaffert, and S. Sastry. Distributed control applications within sensor networks. *Proceeding of the IEEE, Special Issue on Sensor Networks and Applications*, August 2003.

[23] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

[24] J. Xie and I. F. Akyildiz. A distributed dynamic regional location management scheme for mobile ip. *IEEE INFOCOM*, pages 1069–1078, 2002.

[25] F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, March 2002.

Appendix

4 Move operation

Signature:
Input: **object**$_i$
no_object$_i$
cpq$_i$
receive$(msg)_{j,i}, j \in P$,
$msg \in$ {gquery, ack_gquery, grow, shrink}
Output: **send**$(msg)_{i,j}, j \in P$,
$msg \in$ {gquery, ack_gquery, grow, shrink}
cpointer$(j)_i, j \in P \cup \{\perp\}$
State:
$c \in P \cup \{\perp\}$, initially $\perp$
$p \in P \cup \{\perp\}$, initially $\perp$
$gqack \in P \cup \{\perp\}$, initially $\perp$
$gnbrquery \subseteq P$, initially $\emptyset$
$update$, a Boolean, initially $false$
$gtime \in \Re$, a timer, initially ∞
$stime \in \Re$, a timer, initially ∞
$now \in \Re$, a timer indicating current time at i

Figure 3. Signature and state of Tracker$_i$

Lemma 4.1 *$I1$ is an invariant.*

Proof. In the initial states, since $(\forall i : i.c = \perp)$ holds, $I1$ is trivially satisfied.

In our program there are only two places where $i.c$ is set to a non $\perp$ value. The first is the **object**$_i$ action that sets $i.c$ to be i when an object is present at i, hence $I1$ is preserved by this case.

The second is the **receive (grow)**$_{j,i}$ action that sets $i.c$ to j, the sender of the **grow** message. Observe from the **send (grow)** actions that a process sends a **grow** message to either (1) its clusterhead, or (2) its same level neighbor in the clustering hierarchy. It follows from (1) that $i.c \in$ children(i), hence $I1$ is preserved by this case.

For (2), note that j sends a **grow** message to a neighboring process i only if i had previously sent an **ack_gquery** to j. Since **send (ack_gquery)**i, j implies $i.p = h(i)$, it follows that $(i.c \in nbr(i) \ \wedge \ i.p = h(i))$, $I1$ is preserved. □

Lemma 4.2 *$I2$ is an invariant.*

Proof. In the initial states, since $(\forall i : i.p = \perp)$ holds, $I2$ is trivially satisfied.

In our program there are only two places where $i.p$ is set to a non $\perp$ value: the **send (grow)**$_{i,h(i)}$ action and the **receive (ack_gquery)**$_{j,i}$ action. Both cases require that $i.c \neq \perp$ to set $i.p$ to a non $\perp$ value.

Input: **object**$_i$
eff: if $c \neq i \ \wedge \ lvl(i) = 0$ then
$c := i$
$gtime := now + g$

Output: **send (gquery)**$_{i,j}$
pre: $j \in gnbrquery$
eff: $gnbrquery := gnbrquery - \{j\}$
if $gnbrquery = \emptyset$ then
$gtime := now + g * r^{lvl(i)}$

Input: **receive (gquery)**$_{j,i}$
eff: if $p = h(i)$ then
$gqack := j$

Output: **send (ack_gquery)**$_{i,j}$
pre: $gqack = j$
eff: $gqack := \perp$

Input: **receive (ack_gquery)**$_{j,i}$
eff: if $c \neq \perp \ \wedge \ p = \perp$ then
$p := j$

Output: **send (grow)**$_{i,j}$
pre: $now = gtime \ \wedge \ c \neq \perp \ \wedge$
$((j = p \ \wedge \ p \in nbr(i)) \ \vee \ (j = h(i) \ \wedge \ p = \perp))$
eff: if $p = \perp$ then
$p := h(i)$
$gtime := \infty$

Input: **receive (grow)**$_{j,i}$
eff: $c := j$
if $lvl(i) = MAX$ then
$p := i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Figure 4. Grow actions at process i

Input: **no_object**$_i$
eff: if $lvl(i) = 0 \ \wedge \ c \neq \perp$ then
$c := \perp$
$stime := now + s$

Output: **send (shrink)**$_{i,j}$
pre: $now = stime \ \wedge \ c = \perp \ \wedge \ j = p$
eff: $p := \perp$
$stime := \infty$

Input: **receive (shrink)**$_{j,i}$
eff: if $c = j$ then
$c := \perp$
$stime := now + s * r^{lvl(i)}$

Figure 5. Shrink actions at process i

i can set $c := \perp$ only by executing an **object-left**$_i$ or a **receive (shrink)** action. In both cases, $i.stime$ is set to be $now + s * r^{lvl(i)}$. When $now = i.stime$ and $c = \perp$ and $p \neq \perp$, **send (shrink)**$_{i,j}$ action sets $p := \perp$, so $I2$ is preserved. □

Lemma 4.3 *$I3$ is an invariant.*

Proof. In the initial states $I3$ is trivially satisfied.

In our program there are only two places where $i.c$ is set to a non $\perp$ value: the **object**$_i$ action and the **receive (grow)**$_{j,i}$ action. In both cases, either the grow timer at i is set so we have $gtime \in [now, now + g * r^{lvl(i)}]$ or we already have $p \neq \perp$.

Recall that the only way that i can change its non $\perp$ valued p variable is by executing a **send (shrink)** action. This action is enabled only when $i.c = \perp$ so $I3$ is trivially preserved in this case. □

Lemma 4.4 *$I4$ is an invariant.*

Proof. In the initial states $I4$ is trivially satisfied.

$i.c \neq \perp \ \wedge \ i.c \neq i$ implies that i has executed a **receive (grow)**$_{i.c,i}$ action. The corresponding **send (grow)**$_{i.c,i}$ action at process $(i.c)$ implies that $(i.c).p$ is set to i.

The only way that process $(i.c)$ can change its non $\perp$ valued p variable is by executing a **send (shrink)** action. Note that in this case a **shrink** message is inserted into the channel to i. Therefore, $i.c \neq \perp \ \wedge \ i.c \neq i$ implies that either $(i.c).p = i$ or $(i.c).p = \perp$ and a **shrink** message exists in $Channel_{(i.c),i}$. □

Theorem 4.5 *I is an invariant.*

Proof. $I0$ is an invariant due to inspection of the code. In the above lemmas we proved that $I1$ through $I4$ are invariants, thus I is also an invariant. □

Lemma 4.6 *Consider an execution α that starts from an initial state and contains* **object**$_i$ *actions at exactly one process i. Execution α reaches a consistent state.*

Proof. The proof is in two parts:

(1) First, we prove that the move operation terminates. Since **object**$_i$ actions occur at only one process i, no **no_object**$_i$ action is fired. Hence, no **send (shrink)** actions are enabled, and hence, no **receive (shrink)** actions are fired.

A **send (gquery)** is *always* enabled at the highest level process in the tracking path, except when the tracking path is complete. We prove that a **send (ack_gquery)** is never enabled as a result: at the base case, when the tracking path is of length 1, no **send (ack_gquery)** is enabled since all neighbors of i_1 have $c = \perp$. Assume that in a tracking path of up to length l no **send(ack_gquery)** is enabled. Then, the tracking path $\{i_l, \ldots, i_1\}$ is such that $\forall k : 1 < k \leq l : i_k.c \in children(i_k)$ and child pointers of all processes outside the path are $\perp$ since the only way to have $c \neq \perp$ is with a **receive (grow)** action. Thus, i_l sends a grow message to its clusterhead, say i_{l+1}, and since all neighbors of i_{l+1} have $c = \perp$, no **send (ack_gquery)** is enabled for a tracking path of length $l + 1$.

Therefore, in a state where a tracking path $\{i_l, \ldots, i_1\}$ of length l exists, the only enabled action (in addition to the input actions) may be a **send (grow)**$_{i_l,h(i_l)}$ action. Note that for $lvl(i_l) = MAX$, the **send (grow)** action is also disabled. Since the following lexicographic function always decreases the move operation is guaranteed to terminate:

- $MAX - lvl(i_l)$,
- $i_l.now - i_l.gtime$,
- remaining time for **grow** to be delivered at $h(i_l)$.

(2) We now show that a consistent state is reached in α. Since we have shown that for all j not in the tracking path $j.c = \perp \ \wedge \ j.p = \perp$ holds, here we only show that a complete tracking path exists when the move operation terminates. When **object**$_i$ is executed, condition 1 of the tracking path definition is satisfied. The **send (grow)** and the corresponding **receive (grow)** actions establish condition 2 of the tracking path. Since I is an invariant, condition 3 of the tracking path definition is satisfied. Since termination occurs when $lvl(i_x)$ of the first process i_x in the tracking path is MAX, the path is a complete tracking path at termination. □

Theorem 4.7 *Starting from a consistent state, a move operation of the mobile object leads to another consistent state.*

Proof. Since the starting state is a consistent state, the original tracking path is a complete tracking path and for every process i outside the path $i.c = i.p = \perp$.

(1) First, we prove that the move operation terminates. Let L be the level that the new growing path reconnects to the original path.

The highest level process in the new growing path always sends "gquery" to its neighbors, but never receives an "ack_gquery" till level L. Thus, till level L the new path always grows vertically, hence for the growth of the new path up to level L the variant function in part 1 of Lemma 4.6 applies.

Let i be the level L process where the new growing path first intersects the original path. Either i is the MAX level process, meaning there is a vertical tracking path, or i responds to the **receive(grow)** action by pointing to the highest level process in the new growing path. At this point it either does not propagate

this grow further since it already has a parent in the tracking path or it propagates a grow action to its clusterhead parent. This might be repeated for every level above L and below MAX. However, because a grow introduces at most one lateral link per level in a tracking path and then continues to the next level, use of a lexicographic function similar to that of Lemma 4.6 (but incorporating the possibility of 2 links per level) reveals the grow eventually will reach the top level.

Shrinking starts at the level 0 process in the old location of the evader and clears deadwood below level L via the **send(shrink)** and **receive(shrink)** actions. A variant function on <the number of hops in the deadwood, the shrink timers of the processes in the deadwood, and the time for shrink message to be delivered by the channels in the deadwood> can be used to show that the shrink reaches level L. At the point of connection, process i either now points to the new path and upon receipt of the shrink message, discards the message since it is not from its current child or it propagates the message to its parent. Again, this continues until either the shrink propagation is halted by encountering a process whose child pointer points to the new path or when the shrink reaches level MAX.

Note that when a grow installs a pointer in the new path it is not possible for the shrink to remove it; the shrink below level L cannot affect the new path below L and the remaining cases are either that the shrink has already been propagated to the next process on the old path or has not yet arrived at i and will be halted when it arrives. I guarantees there is no cycle in the path that allows the same shrink to visit a process more than once. The argument can be repeated for each intersection of the two paths above L.

(2) We now show that a consistent state is reached when the grow and shrink actions terminate. The new growing path grows vertically and satisfies conditions 1, 2, 3 of the tracking path definition and intersects the original path at level L. Since the grow action adds links that are not removed by the concurrent shrink action and either connects to a portion of the old complete path that reaches level MAX or climbs in level until it reaches MAX itself, a complete tracking path will result. Since the starting state was a consistent state, and our grow and shrink actions have no effect on processes other than those in the original path and the new growing path, for every process i that is in neither path $i.c = i.p = \perp$ is preserved. Finally, since the shrink actions set $i.c = i.p = \perp$ for every process i at which they complete before the grow arrives and halt when a new pointer is encountered, it follows that the resulting state is also a consistent state. □

Lemma 4.8 *A shrink action propagated from level 0 through a tracking path with full extension below l takes longer to erase a level l parent pointer than a grow action takes to propagate a vertical growth up to l and have a neighbor query delivered.*

Proof. The minimum time it takes for shrink messages to erase a level l pointer at a process i is at least the sum of shrink timer waits at each process in the tracking path up to i. At level 0, the timer wait is s and for levels $j : 0 < j < l$, a process's timer waits are sr^j, with 2 such processes at each level (one with a lateral link and one with a pointer to a hierarchy child). At level l, the shrink timer takes sr^l to expire and delete i's p pointer.

In the meantime, propagation of **grow** messages to a level l process i' could suffer maximum message delay in addition to the time it takes for grow timers to expire, taking a total of up to $\sum_{j=0}^{l-1}[gr^j + \delta mr^j]$ time. Here i' would query its neighbors. This message could take $2\delta mr^{l-1}$ time to deliver at a neighboring level l process. Algebra reveals that this total time is less than the described shrink time: $\sum_{j=0}^{l-1}[gr^j + \delta mr^j] + 2\delta mr^{l-1} < s + \sum_{j=1}^{l-1} 2sr^j + sr^l$. □

Lemma 4.9 *Consider a complete tracking path $\{i_x, \ldots, i_1\}$ in a consistent state before a move operation and the resulting complete path $\{i'_{x'}, \ldots, i'_1\}$ in the consistent state after the operation. There exists an index j and level l where:*

1. *$lvl(i_j) = l$,*
2. *The old and new path share a prefix up to i_j: $\{i'_{x'}, \ldots, i'_1\} = \{i_x, \ldots, i_j\} \cdot \{i'_{j'}, \ldots, i'_1\}$,*
3. *Path $\{i'_{j'}, \ldots, i'_1\}$ has a vertical growth to l,*
4. *The old path $\{i_x, \ldots, i_j, \ldots, i_1\}$ has full extension below l,*
5. *Each process i below level l in the new path does not neighbor a process j such that $j.p = h(j)$ in the old path.*

Proof. Because there is only one level MAX process, the two tracking paths will satisfy condition 2. Consider the process i_j for which the new path shares the prefix of the old path up to i_j. Consider l to be $lvl(i_j)$. We want to show conditions 3 and 4. Since the conditions trivially hold for $l = 0$, consider $l > 0$.

Assume for contradiction that condition 3 doesn't hold. This can only occur if a lateral link was introduced at some level below l in the new path. Lateral links are introduced as a result of an **ack_gquery** message returned by a neighbor in response to a **gquery**. Such an acknowledgement is only returned by a neighboring process whose parent pointer p is to its own

clusterhead. Since the move operation was performed starting from a consistent state, such a process i must have been in the old path.

When this is the case, the lateral link would have been installed to the old tree at this point. This installation would take up to $gr^{l'} + 2\delta m r^{l'-1}$ time. In the meantime, i may have already sent a **shrink** message to its parent. However, our assumptions for s and g guarantee that even in the case where the **shrink** message does not suffer message delay, the **grow** message from i to its parent (which could take up to $gr^{l'} + \delta m r^{l'}$ time) will arrive before the **shrink** action has completed at $i.p$; the reduced expression described here is $2gr + \delta mr + 2\delta m < sr^2$, which is satisfied by the fact that $g \leq s - \delta m$. Hence, the new and old paths would share a prefix that extended beyond i_j, a contradiction.

Assume condition 4 does not hold. There is some $l' < l$ such that the old path only has one process at level l'. Since the new path does not connect at this level, it must be the case that either this process i is not a neighbor of the corresponding level l' process i' in the new path or this process's parent pointer was deleted by a shrink action before i' queried it.

Process i points to a hierarchy child. The maximum distance from the cluster boundary of process i's level $l'-1$ cluster that $i.c$'s pointer could be to is $mr^{l'-2}$. After this pointer, we could add as much as $m + 2m\sum_{j=1}^{l'-3} r^j$ distance, from following lateral links to the bottom level. After the evader moves a distance of 1, the total distance from the edge of process i's level $l'-1$ cluster to the evader could be as much as $1 + m + mr^{l'-2} + 2m\frac{r^{l'-2}-r}{r-1}$. For i to not be a neighbor of process i', this total distance would have to be more than $qr^{l'-1}$, the minimum distance separating non-neighboring $l'-1$ cluster boundaries. However, algebra reveals this is false.

Then it must be that i's parent pointer was deleted by a shrink action before i' had a chance to query it or that the grow timer at i' expired before a reply might have been received. However, Lemma 4.8 guarantees that the former cannot be and the latter is disproved by the fact that $g > \frac{s+\delta m}{r}$, implying $gr^l > 2\delta m r^{l-1}$. □

Theorem 4.10 *Starting from a consistent state, move operations of the mobile object to a total of distance d away require at most $O(d * \omega mr * MAX)$ amortized work and $O(d * gr^2 * MAX)$ amortized time to update the tracking path.*

Proof. A level l pointer in the tracking path is updated as often as every $q\sum_{j=1}^{l-2} r^j = q\frac{r^{l-1}-r}{r-1}$ distance because of full extension at lower levels, whose construction is guaranteed by the previous lemma. As a result, if an object has traveled a total of d distance then we must consider updates up to the MAX level, with up to $\frac{d(r-1)}{q(r^{l-1}-r)}$ updates at each level l.

A work cost of up to mr^{l-1} is incurred every time a level l pointer is updated. This results in up to $4m\omega r^{l-1}$ communication to query the neighbors. Additionally, in half the cases it will result in another $2mr^{l-1}$ communication to update one neighbor to have a lateral link. This brings the average communication for a level l clusterhead update to $(4\omega + 2)mr^{l-1}$.

Further, a time cost of up to δmr^{l-1} is incurred when updating a level l pointer, followed by a gr^l wait time for the grow. This cost in some cases must also accomodate lateral link insertions, resulting in additional $2\delta mr^{l-1}$ time for communication.

The worst case amortized cost for a move is then $\sum_{j=1}^{MAX}[\frac{d(r-1)}{q(r^{j-1}-r)} * (4\omega+2)mr^{j-1}]$, or $O(d\omega mr * MAX)$. The time cost is $\sum_{j=1}^{MAX}[\frac{d(r-1)}{q(r^{j-1}-r)} * (3\delta m + gr)r^{j-1}]$ or, given that $g > \frac{4\delta m}{r}$, $O(d * gr^2 * MAX)$. □

5 Fault-containment

Internal: **start-shrink**$_i$
pre: $(c = \perp \ \wedge \ p \neq \perp \ \wedge \ stime \notin [now, now + s * r^{lvl(i)}])$
$\vee [p \in nbr(i) \ \wedge \ c \in nbr(i)]$
eff: $c := \perp$
$stime := now + s * r^{lvl(i)}$

Internal: **start-grow**$_i$
pre: $c \neq \perp \ \wedge \ p = \perp \ \wedge \ gtime \notin [now, now + g * r^{lvl(i)}]$
eff: if $lvl(i) = MAX$ then
$p = i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Figure 6. Starting grow/shrink at process i

Lemma 5.1 *Starting from an arbitrary state of the processes and channels, our tracking program stabilizes to a state where for every process i, I holds and at most 1 message is in travel in every incoming channel to i within at most $2m\delta r^{lvl(i)-1}$.*

Proof. $I0$, $I1$, $I2$, and $I3$ (but not $I4$!) are all local to a process and are immediately established by the local process actions as discussed in Sections **??**. $I0$ follows from inspection of **object** and **no_object** actions. $I1$ is established due to our domain assumption, also the first disjunct in the **start-shrink** action corrects the third disjunct of $I1$. $I2$ is established by the **start-shrink** action, $I3$ is established by **start-grow**.

Next we show first how the channel contents are cleared once $I0...I3$ is established, and then consequently how $I4$ is established.

Output: **send (heartbeat)**$_{i,j}$
pre: $now = next \ \wedge \ j = p$
eff: $next := now + b * r^{lvl(i)}$

Input: **receive (heartbeat)**$_{j,i}$
eff: if $c = \perp$ then $c := j$
if $c = j$ then
$timeout := now + (b + 2\delta m/r) * r^{lvl(i)}$

Internal: **timeout_expire**$_i$
pre: $now = timeout \ \wedge \ c \neq \perp \ \wedge \ c \neq i$
eff: $c := \perp$

Internal: **heartbeat_set**$_i$
pre: $p \neq \perp \ \wedge \ next \notin [now, now + b * r^{lvl(i)}]$
eff: $next := now + b * r^{lvl(i)}$

Internal: **timeout_set**$_i$
pre: $c \neq \perp \ \wedge \ c \neq i$
$\wedge \ timeout \notin [now, now + (b + 2\delta m/r) * r^{lvl(i)}]$
eff: $timeout := now + (b + 2\delta m) * r^{lvl(i)}$

Figure 7. Heartbeat actions at process i

In the starting state, there can be a bounded number of messages in the channels. Since $I1$, $I2$, and $I3$ are invariants under receive actions, receiving these messages does not violate them.

Since after $I1$ is established there cannot be any cycles in the network with respect to message forwarding, we next show that the messages in the channels in the starting state are reduced to at most 1 message per channel within a bounded time $2m\delta r^{lvl-1}$.

The following messages are removed from the channels and by inspection of the code do not result in new messages being inserted into the channels:

- messages from processes outside $nbr(i)$ and $children(i)$, and messages other than **grow**, **shrink**, **gquery**, **ack_gquery**, and **heartbeat**,
- **heartbeat** messages,
- **ack_gquery** messages,
- and **grow/shrink** messages received by the highest level clusterhead.

The following messages result in limited forwarding:

- **gquery** can result in at most one **ack_query** message. Thus, its effects are diminished within $2m\delta * r^{l-1}$.
- A **grow/shrink** message can result in forwarding of a **grow/shrink** message but this time either to a higher level or to a same level process with no lateral link (which means the next time the forwarding has to be to a higher level).

  Since before propagation of a grow/shrink message we wait for grow/shrink timers to expire and a new grow/shrink message resets the grow/shrink timers, all messages in a channel are received and suppressed to at most 1 message forward. Only the last **grow/shrink** message in the channel is dominant.

$I4$ is a pair-wise link predicate on processes. After $I0$ through $I3$ are established, and junk messages in the channels are consumed as above, $I4$ is established by the **timeout_expire** action. □

Theorem 5.2 *Our tracking program stabilizes to a consistent state.*

Proof. From Lemma 5.1, it follows that I is established at all processes and the number of initial messages in channels is reduced to at most 1.

In a state where I holds at all processes, along with the tracking path there may be fake tracking paths, which fail to satisfy all conditions of a tracking path, for example in a fake tracking path it may be that $i_1.c \neq i_1$ and $(\exists k : (i_k.c).p \neq i_k)$. Due to **start-shrink** actions, a fake tracking path is cleaned starting from the lowermost process in the path. Using a lexicographic function similar to that in Theorem 4.7, we can argue that all fake paths are eventually cleared.

A **start-grow** from the topmost process in the "true" tracking path results in a complete tracking path (using a similar proof to that of Lemma 4.6) and hence in a consistent state. □

Lemma 5.3 *Propagation of a shrink action started at level l_1+1 catches propagation of a grow action started at level l_2 by level l where*

$$l = l_2 + \lceil \log_r \frac{br - b + sr + gr - 2s + 3\delta m}{gr - s - \delta m} \rceil.$$

Proof. Consider the worst case where faults corrupt the network in such a way as to introduce a lateral link at each faulty level. This "bad grow" could then quickly propagate upwards without use of lateral links.

It takes at most $(b + 2\delta m/r) * r^{l_1+1}$ for a heartbeat timer to expire at level $l_1 + 1$ and trigger a shrink action. After this it can take the maximal time to propagate shrink actions up to level l_2 to correct corrupted portions of the network, $\sum_{j=l_1+1}^{l_2}[sr^j + \delta m r^j + sr^j + 2\delta m r^{j-1}]$, followed by the maximal time to propagate shrink actions through the non-corrupted levels of the network that had received **grow** messages, $\sum_{j=l_2+1}^{l-1}[sr^j + \delta m r^j]$.

To have fault containment, this total time has to be less than the minimum time that a grow could take to be propagated past level l from level l_2+1, the level above the last initially faulty level:

$(b+2\delta m/r)r^{l_1+1}+\sum_{j=l_1+1}^{l_2}[2sr^j+\delta mr^j+2\delta mr^{j-1}]+\sum_{j=l_2+1}^{l-1}[sr^j+\delta mr^j]<\sum_{j=l_2+1}^{l}gr^j$.

With simplification the above inequality becomes:

$r^{l_1}(br^2-br-2sr+\delta mr-4\delta m)+r^{l_2}(sr+gr+2\delta m)<r^l(gr-s-\delta m)$. □

Lemma 5.4 *Propagation of a grow action started at level l_1 catches propagation of a shrink action started at level l_2 by level l where*

$$l=l_2+\lceil\log_r\frac{br-b+sr^2-gr-\delta m}{sr-gr-3\delta m}\rceil.$$

Proof. Consider the worst case where the correcting **grow** must be propagated vertically and connect with a lateral link at level l.

After the heartbeat timer at level l_1 expires it will send a heartbeat to the lowest faulty level, taking up to $br^{l_1}+\delta mr^{l_1}$. This will trigger propagation of a **grow** message. Propagation of this grow action might suffer maximal message delay of $\sum_{j=l_1+1}^{l-1}[gr^j+\delta mr^j]$ to reach level l and then insert a lateral link at level l, taking up to $gr^l+2\delta mr^{l-1}$, but must still take less time than for a shrink to be quickly propagated past level l:

$br^{l_1}+\delta mr^{l_1}+gr^l+2\delta mr^{l-1}+\sum_{j=l_1+1}^{l-1}[gr^j+\delta mr^j]<\sum_{j=l_2+1}^{l}sr^j$.

Simplification gives:

$$r^{l_1}(br-b-gr-\delta m)+r^{l_2}(sr)<r^l(sr-gr-3\delta m+2\delta m/r).$$

□

The difference $l-l_2$, the size of contamination due to fault propagation, is tunable via grow and shrink timer settings. Later, after imposing additional constraints on grow and shrink timers for handling of concurrent operations, we give sample values for the timers and the size of contamination for those values.

Theorem 5.5 (Fault-local stabilization) *For a perturbation size S, our program self-stabilizes in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.*

Proof. Even though there may be many different scenarios for corruption, since they all lead to either mispropagation of a shrink or a grow, they all can be cast to the below two cases for a perturbed process i: 1) i can be corrupted to think it has a child and i grows up, 2) i can be corrupted to think it has no child and i shrinks up.

In either case i learns the correct information within at most $O(r^{lvl(i)})$ time and from the containment arguments in Lemmas 5.3 and 5.4 this correction wave contains previous misinformed waves within $O(r^{lvl(i)})$ time and work.

The work for fault-containment is additive: summation of the work for all perturbed processes results in $O(S)$ work overall. However, fault-containment takes place concurrently for all perturbed processes, the fault-containment time $O(L)$ for the highest level perturbed process dominates. □

6 Find operation

Each process maintains a child pointer c' and a *nbrtimeout* timer. Boolean $cpqflag$ indicates whether the process should query the **Tracker** automata for the latest tracking structure pointer. Boolean fqn indicates if the process has up-to-date information and is executing a **find**. Variable *nbrqset* is used to query neighbors while *nbrack* is used to answer neighbors' queries. We assume that $nbrack \in nbr(i)\cup\{\perp\}$ and that $nbrqset \subseteq nbr(i)$. Additionally, we add the local correction that if $cpqflag = false \ \wedge \ fqn = false$ then $nbrtimeout = \infty$.

Signature:
- *Input:* **find**$_i$
  receive$(msg)_{j,i}, j \in P$, $msg \in$ {find, fqack, fquery}
  cpointer$(j)_i, j \in P\cup\{\perp\}$
- *Output:* **found**$_i$
  cpq$_i$
  send$(msg)_{i,j}, j \in P$, $msg \in$ {find, fqack, fquery}
- *Internal:* **fqset**$_i$

State:
- $c' \in P\cup\{\perp\}$
- fqn, a Boolean
- $nbrack \in P\cup\{\perp\}$
- $nbrqset \subseteq P$
- $cpqflag$, a Boolean
- $nbrtimeout \in \Re$, a timer
- $now \in \Re$, a timer indicating current time at i

Figure 8. Finder signature/state at process i

Theorem 6.1 (Proximity) *In a consistent state, for any process j that is at most d distance away from the mobile object, one of the following holds:*

- *$h^{\lceil log_r d\rceil+1}(j)$ is in the tracking path or*

Input: $\mathbf{find}_i$
eff: if $lvl(i) = 0$ then
$cpqflag := true$
$nbrtimeout := \infty$

Output: $\mathbf{cpq}_i$
pre: $cpqflag = true$
eff: none

Input: $\mathbf{cpointer}(j)_i$
eff: $cpqflag := false$
$fqn := true$
$c' := j$
if $nbrack \neq \perp \ \wedge \ c' = \perp$ then
$fqn := false$
$nbrack := \perp$

Output: $\mathbf{found}_i$
pre: $fqn = true \ \wedge \ c' = i$
eff: $fqn := false$

Output: $\mathbf{send(find)}_{i,j}$
pre: $fqn = true \ \wedge \ ((j = c' \ \wedge \ c' \neq i) \ \vee$
$(j = h(i) \ \wedge \ c' = \perp \ \wedge \ nbrtimeout \leq now))$
eff: $fqn := false$

Input: $\mathbf{receive(find)}_{j,i}$
eff: $cpqflag := true$
$nbrtimeout := \infty$

Input: $\mathbf{receive(fqack)}_{j,i}$
eff: $fqn := false$

Input: $\mathbf{receive(fquery)}_{j,i}$
eff: $cpqflag := true$
$nbrtimeout := \infty$
$nbrack := j$

Output: $\mathbf{send(fqack)}_{i,j}$
pre: $nbrack = j$
eff: $nbrack := \perp$

Internal: $\mathbf{fqset}_i$
pre: $fqn = true \ \wedge \ c' = \perp \ \wedge \ nbrtimeout = \infty$
eff: $nbrqset := nbr(i)$

Output: $\mathbf{send(fquery)}_{i,j}$
pre: $j \in nbrqset$
eff: $nbrqset := nbrqset - \{j\}$
if $nbrqset = \emptyset$ then
$nbrtimeout := now + 4\delta m r^{lvl(i)}$

Figure 9. Finder actions at process i

- $\exists i \in nbr(h^{\lceil log_r d \rceil + 1}(j))$ *that is in the tracking path.*

Proof. Let j be r^l distance from i_1 where the mobile object resides. In order to violate the proximity invariant, we need to show that neither $j.h^{l+1}$ nor any neighbor of $j.h^{l+1}$ is in the tracking path. This is only possible if the distance between j and all boundary processes in a level l cluster on the tracking path is more than qr^l, the minimum distance between processes subsumed by two non-neighboring level l clusters. However, for a contradiction, we show below that there exists a process in a level l cluster on the tracking path whose distance to j is less than qr^l.

Since there is at most 1 lateral link at every level, a level l process in the tracking path is at most $m(r^l + r^{l-1})$ distance from a level $l+1$ process in the path (mr^l is the maximum radius of a level l cluster and a lateral link at level l is of length $2m * r^{l-1}$). Therefore, we conclude that the distance between the leaf process i_1 of the tracking path and a process on the boundary of a level l cluster in the tracking path is at most: $2\sum_{j=0}^{l-1} mr^j$. This distance is always less than qr^l, hence the proximity invariant holds.

$$
\begin{aligned}
&(2\textstyle\sum_{j=0}^{l-1} mr^j) + r^l < qr^l \\
\equiv\ & 2m\tfrac{r^l-1}{r-1} + r^l \quad < qr^l \\
\equiv\ & 2m\tfrac{1}{r-1} + 1 \quad \leq q \\
\equiv\ & true \ \{\text{since } q \geq \tfrac{2m+r-1}{r-1}\}.
\end{aligned}
$$

□

Theorem 6.2 *A* find *operation invoked at distance d from a mobile object results in $O(d * \omega rm)$ work and takes $O(d * \delta rm)$ time.*

Proof. The cost of find operation is calculated as follows: At every level l all ω neighbors are consulted, and a find operation will be answered at level $\lceil log_r d \rceil + 1$ in the worst case (follows from Theorem 6.1). Thus, given propagation costs and 2 communications with each of ω neighbors that are at a distance of at most $2m * r^l$ at each level l, we have:

$$\sum_{j=0}^{\lceil log_r d \rceil + 1} (4\omega m + m) * r^j = (4\omega m + m)\frac{dr^2 - 1}{r - 1}.$$

We add the cost of following the tracking path links, at most $2m + \sum_{j=1}^{\lceil log_r d \rceil + 1}(2mr^j + mr^{j-1})$, to this number. This gives us that the total work of a find is $O(d\omega rm)$, linearly proportional to the initial distance between the evader and the initiator of the find.

The time for a find to reach this level is expressed by $\sum_{j=0}^{\lceil log_r d \rceil + 1} 5\delta mr^j = 5\delta m\frac{dr^2-1}{r-1}$, for transmission up the hierarchy and queries of neighbors at each level.

Following tracking path links after the path is found takes up to $2\delta m + \sum_{j=1}^{\lceil log_r d \rceil + 1}(2\delta m r^j + \delta m r^{j-1})$ for a total time of $O(d * \delta r m)$. □

7 Concurrent move operations

Lemma 7.1 *In the absence of faults, there exists a tracking structure.*

Proof. When the evader first enters the system there is a singleton tracking structure, $\{i_1\}$, where the evader is located. Initially the requirements trivially hold.

Now we assume that there exists a tracking structure and show that, regardless of evader movements and allowable timing uncertainty, reachability conditions are maintained. For contradiction, assume that in one state they are and in the next they are not.

Consider first the case where the reachability violation occurs at a level 1 endpoint. One of two things must have happened: either the endpoint's pointer c is to a hierarchy child and the next structure segment does not contain a neighbor of that child or the pointer is to neighbor and the next segment does not contain a process that neighbors that neighbor. For the first condition, it takes up to $s + \delta m$ time for the endpoint's pointer to be erased through propagation of shrink actions. In the meantime, the evader must have moved from the location of c to a neighboring process. Since the evader takes at least e time at this new location, we know that the condition is satisfied. For the second condition, it takes up to $s + \delta m + sr + 2\delta m$ time for a shrink action to erase c. In the meantime, the evader must have moved far enough to violate the reachability condition, which would require it to have travelled at least $2q - 2m$ distance and then propagated a shrink action to the appropriate level 1 process, which takes at least $s + sr$ time. Given our assumptions on s and e this is not possible.

Now consider the case where the violation of reachability condition occurs at an endpoint whose level l is greater than 1. The first case, where a pointer is to a hierarchy child, is subsumed by the subsequent cases, since code inspection and Theorem 4.10 reveal that the pointer here will be updated to the neighbor. Similarly, we reason that to violate either of the remaining conditions, it must be the case that the evader has moved far enough to trigger a shrink wave to ultimately clean the required neighboring pointer, and that the shrink has managed to progress beyond two levels below the endpoint in question.

It takes the evader at least $e[2qr^{l-3} - \sum_{j=1}^{l-4} 2mr^j]$ time to move out of range of a path segment's level $l-2$ pointer. Afterwards it takes at least $s + \sum_{j=1}^{l-3}[2sr^j] + sr^{l-2}$ time to delete the child pointer of the level $l-2$ process in this segment.

To violate the reachability condition, we require this time to be less than the maximum required to delete the level l pointers in the old path: $s + \delta m + \sum_{j=1}^{l-1}[2sr^j + \delta m r^j + 2\delta m r^{j-1}] + sr^l + 2\delta m r^{l-1}$. However, algebra reveals that this is not true, and hence the reachability condition holds. □

Theorem 7.2 *Consider a trace α of the program that contains an* **object**$_i$ *event. The trace α eventually reaches a good state.*

Proof sketch. In Theorem 7.1 we proved that good states are closed under move operations. The proof can be adapted for the case where the tracking structure is still growing into a complete tracking structure by noting that the longest a path segment that started to grow in response to the first **object**$_i$ event can take to reach the topmost level is $\sum_{j=0}^{MAX-1}[gr^j + \delta m r^j]$ time and a complete tracking structure is established. □

Lemma 7.3 *Consider a path segment (except the first) in the tracking structure. The preceding segment in the structure contains a neighbor of the first process.*

Proof. Consider the case where a move of distance d should be extending a segment to level $l = \lceil log_r(d\frac{r-1}{q} + r)\rceil + 1$, the highest level that might be updated after a move of distance d from the leaf of a straight tracking structure. In the worst case the prior segment suffers from maximum message delay when propagating a grow wave to level l, taking up to $\sum_{j=0}^{l-1}[gr^j + \delta m r^j]$ time, before installing the information, taking an additional gr^l time.

If messaging is fast for the new segment, after the move has completed in time ed, it will only take the total time for the grow timers to expire through level $l-1$ to reach level l of the hierarchy, taking $\sum_{j=0}^{l-1} gr^j$. This time must be more than the amount of time it would have taken for the old segment described above to have installed information at level l: $ed > gr^l + \sum_{j=0}^{l-1} \delta m r^j$, which is satisfied by our assumptions. □

Theorem 7.4 *The tracking service with concurrent moves is self-stabilizing to a good state.*

Proof. Similar to the proof of Theorem 5.2. From Lemma 5.1, it follows that I is established at all processes and the number of initial messages in channels is reduced to at most 1. Then fake tracking paths are cleared via **start-shrink** actions.

After this point, instead of a single path (as in Theorem 5.2), due to object mobility multiple path segments

may start to grow. Note that, since the invariant I is already established, the reachability condition is satisfied for the newly growing path segments. Thus, a complete tracking structure, and hence, a good state is reached eventually. □

Theorem 7.5 (Fault-local stabilization) *For concurrent moves and perturbation size S the system self-stabilizes in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level.*

Proof. Similar to the proof of Theorem 5.5. Any arbitrarily perturbed state can be captured in terms of mispropagations of a shrink or a grow for each perturbed process i. Since in either case i learns the correct information within, again, at most $O(r^{lvl(i)})$ time, we conclude using Lemmas 5.3 & 5.4 that the system self-stabilizes in $O(S)$ work and in $O(r^L)$ time where L denotes the highest perturbed level. □

8 Concurrent find and move operations

Theorem 8.1 *A find operation invoked within d distance of the mobile object intercepts the tracking structure by level $\lceil log_r d \rceil + 1$ of the hierarchy.*

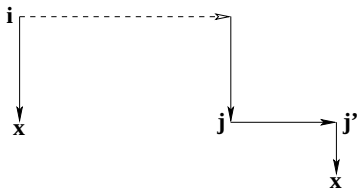
Proof. If the find starts d away from the mobile object then by the time it reaches the $\lceil log_r d \rceil + 1$ level, having spent the maximal roundtrip neighbor messaging time at each level on the way up, taking up to $\sum_{j=0}^{\lceil log_r d \rceil} [\delta m r^j + 4\delta m r^{j-1}]$ time, there should still be a pointer at a neighbor (it takes up to $2\delta m r^{\lceil log_r d \rceil}$ to get a query to all neighbors). For there to still be a pointer, it means that the evader has not had time to cross out of the $\lceil log_r d \rceil$ cluster and quickly propagate a shrink despite the d unit head start it had:

$\sum_{j=0}^{\lceil log_r d \rceil} [\delta m r^j + 4\delta m r^{j-1}] + 2\delta m r^{\lceil log_r d \rceil} < e[q r^{\lceil log_r d \rceil} - d] + s + 2\sum_{j=1}^{\lceil log_r d \rceil} s r^j$.

This is satisfied by our assumptions. □

Lemma 8.2 *If the find reaches a dead end by following a downward child pointer at a level l process i, then after searching neighbors of i the find will reach a level $l-1$ process j in a neighboring path segment.*

Proof. It follows from the reachability condition of the tracking structure that a newer path is available at a neighbor of i. Furthermore, the timing conditions in Lemma 7.1 ensure that in the worst case the newer path segment is structured as shown in the below figure. Note that the case where j has a downward child pointer to a level $l-2$ process, say j'', is not the worst



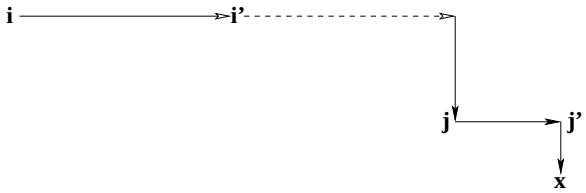
case: Lemma 4.9 ensures that by the time j'' sends a shrink message to j, a newly growing path (that of j') would have already connected to j.

The maximum time a find takes in following the downward pointer at i to the dead end process, querying neighbors there, returning to i, querying neighbors of i, and following the newer path to j is: $5\delta m r^{l-1} + 4\delta m r^{l-1} + 4\delta m r^{l-2}$.

For j to remain intact until the find operation reaches j, we require the shrink time for j' to be greater than the above term: $s r^{l-1} > 9\delta m r^{l-1} + 4\delta m r^{l-2}$, which is satisfied by our assumptions. □

Lemma 8.3 *If the find reaches a dead end by following a lateral pointer at level l process i, then after querying neighbors of the dead end process the find will reach a level $l-1$ process j in a newer path segment.*

Proof. The proof is similar to that of Lemma 8.2. It follows from the reachability condition of the tracking structure that a newer path is available at a neighbor of the dead end process. Furthermore, the timing conditions in Lemma 7.1 ensure that in the worst case the newer path segment is structured as shown below. The



maximum time a find takes in travelling to i', querying neighbors of the dead end process i', and following the newer path to j is: $5\delta m r^{l-1} + 4\delta m r^{l-1}$.

For j to remain intact until the find operation reaches j, we require the shrink time for j' to be greater than the above term: $s r^{l-1} > 9\delta m r^{l-1}$, which is satisfied by our assumptions. □

Theorem 8.4 *A find operation invoked within d distance of a mobile object requires $O(d\omega rm)$ work and $O(d\delta rm)$ time to reach the object.*

Proof. The cost of the searching phase is the same as that in Theorem 6.2: $(4\omega m + m)\frac{dr^2-1}{r-1}$. It follows from Lemmas 8.2 and 8.3 that work performed in the tracing phase is at most $\sum_{j=0}^{\lceil log_r d \rceil + 1} (4m r^{j-1} + 4\omega m r^{j-1} + 5m r^{j-1} + 4\omega m r^{j-1} + 4\omega m r^{j-2})$, or $O(d\omega rm)$.

Similarly, the time for the searching phase is the same as before: $5\delta m\frac{dr^2-1}{r-1}$. The tracking phase now has to take into account the possibilities that lemmas 8.2 and 8.3 come into play. This time is at most $\sum_{j=0}^{\lceil log_r d\rceil+1}(8\delta mr^{j-1}+9\delta mr^{j-1}+4\delta mr^{j-2})$. This plus the searching phase results in a total of $O(d\delta rm)$ time.

The ability of the evader to move during find operations results in only constant factor differences in the time and work it would take for a find to reach it. □

9 Simulations

Here we consider a random movement model for the object and investigate the average-case performance through simulations. For simulations we use Prowler, an event-driven wireless sensor networks simulator by Vanderbilt University (`www.isis.vanderbilt.edu/projects/nest/prowler/`). Besides STALK, we simulate two algorithms for comparison. The first algorithm always propagates grows/shrinks to the topmost clusterhead after each move of the object. The second algorithm is similar to STALK except that it does not use lateral links. The code is available at `www.cis.ohio-state.edu/~demirbas/track/`.

In our experiments, we use a grid topology and employ a static, 2-level clustering: the radius of a level 1 cluster is 1, and the radius of the level 2 cluster is r — hence the grid is $2r$-by-$2r$. The object moves randomly to any of the 8 (including the diagonals) neighbors.

The first figure shows work done by each algorithm for increasing values of r. We simulated up to networks of 2500-nodes and observed that in contrast to the first two, STALK scales well with respect to r. Comparing STALK to algorithm 2, it is clear lateral links have a large impact on scalability. Theoretical analysis shows that work of STALK scales linearly with respect to r in the worst case, but simulation reveals that work scales sublinearly for a randomly moving object.

The second figure shows work done by each algorithm with respect to distance the object moved: STALK outperforms the other algorithms. Theoretical results show STALK's work scales linearly with respect to distance, but simulation shows it scales better: the randomly moving object has locality to its movements that STALK captures via lateral links, avoiding the propagation of updates to upper levels as much as possible.