STALK: A Self-Stabilizing Hierarchical Tracking Service for Sensor Networks

Murat Demirbas, Anish Arora

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

Tina Nolte, Nancy Lynch

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

Abstract

In this paper, we present STALK*, a hierarchical tracking service for sensor networks.* STALK *favors local 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. Moreover,* STALK *is fault-local stabilizing: starting from an arbitrarily corrupted state the tracking structure satisfies its specification within time and communication cost proportional to the perturbation size 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.*

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

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

Number of pages: 1 page cover + 10 page main + 1 page references + 10 page appendix
Contact information. Email: demirbas@cis.ohio-state.edu,
Tel: +1 614 688 4637, Fax: +1 614 292 2911,
Address: 395 Dreese Labs; The Ohio State University; Columbus, OH 43210

Eligible for Best Student Paper Award
Consider for BA if not accepted as regular presentation

1 Introduction

Due to applications in mobile computing, cellular telephony, and military contexts, tracking of mobile objects has received significant attention [3,4,6,7,16,17,21]. The DARPA Network Embedded Software Technology (NEST) program posed tracking as a challenge problem in wireless sensor networks, and several groups in the program 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 [20], and detection, classification, and tracking of various intruders, such as persons and cars, are demonstrated in [1].

Besides the opportunities they provide for tracking, such as easy deployment and ubiquitous sensing, wireless sensor networks impose the following constraints: Sensor nodes have limited computational resources (such as 8K RAM); centralized algorithms are unsuitable due to their large computational requirements. Sensor nodes are energy constrained; algorithms that impose 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 [12]. Lastly, on-site maintenance is infeasible; sensor networks should be self-healing.

Contributions. We present a tracking service for sensor networks, namely STALK (Self-stabilizing TrAcking via Layered linKs).

STALK is local and scalable; 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.

STALK is fault-local stabilizing: starting from an arbitrarily corrupted state it 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.

Overview of STALK. We achieve scalability of STALK by maintaining tracking information 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. Our tracking structure is a path rooted at the topmost 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 path. Each process in the path points to a process that is generally closer to the mobile object and has more recent information about its location.

There is a tradeoff between work for finds and work for updating a tracking structure after moves. In a 2-D network, one extreme is the full-information strategy where a find costs $O(1)$ and a move of distance d costs $O(d^2)$. The other extreme is the no-information strategy where a move is $O(1)$ and a find of an object d away is $O(d^2)$ [5]. We use a partial information strategy to optimize both finds and moves.

Find operation. A find operation invoked at a process i 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 mobile object to increasingly higher levels of the clustering hierarchy and connect to the original path. The shrink action cleans branches deserted by the mobile object. Shrinking also starts at the lowest level and climbs to increasingly higher levels of the hierarchy.

Fault-local stabilization. We use two concepts for achieving fault-locality: hierarchical partitioning and arresting waves. 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 accessibility of the tracking structure: it still seamlessly tracks continuously moving objects.

Related work. STALK provides a "network middleware support" for tracking: it assumes an underlying service for detection of a mobile object [11,14,22] and provides a basis for higher level applications such as multiple target tracking [17,18] and pursuer-evader applications [8].

In [3], 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 that constructs sparse covers [2].

In [6], using a hierarchy of location servers, a stabilizing location management protocol is presented; this protocol fails to ensure locality of finds. Other tracking protocols such as [9,13], on the other hand, suffer from nonlocal update problems, where updates to a tracking structure may take work dependent on the network size rather than distance moved.

Organization of the paper. After presenting datatypes 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. In Section 9 we present simulation results and then make concluding remarks in Section 10. For space reasons, we relegate code and detailed proofs to the Appendix.

2 Model

We consider a sensor network consisting of multiple sensor locations in a set V. Each sensor location plays host to (possibly) multiple processes with identifiers from a set P. We denote the sensor location of a particular process i using $loc(i)$ and the Euclidean distance between two locations v and w using $dist(v,w)$. For convenience we use $loc(I) = \bigcup_{i\in I}\{loc(i)\}$ for $I \subseteq P$ and $dist(i,j) = dist(loc(i), loc(j))$ for $i,j \in P$.

Hierarchical partitioning. We partition P into $MAX+1$ equivalence classes $\{P_0, P_1, \cdots, P_{MAX}\}$ called *levels*, and partition processes at the same level l, $0 \le l \le MAX$, into groups called *clusters* $\mathcal{C}_l = \{C_{l,1}, C_{l,2}, \cdots, C_{l,k_l}\}$. We call $\mathcal{C}$ the set of all clusters. Each cluster $C \in \mathcal{C}$ has a clusterhead $head(C) \in C$ and a set of cluster neighbors $nbr_l \subseteq \mathcal{C}_l \times \mathcal{C}_l$. We assume this relation is symmetric and constrain the level 0 neighbor relation so that for $i,j \in P_0, (i,j) \in nbr_0 \iff dist(i,j) \le 1$.

For a particular level, a location can host at most one process. We constrain the clustering so that every sensor location hosts one process that belongs to a level 0 cluster and level l clusterhead locations are level $l+1$ process locations for all levels $l < MAX$. At level MAX the hierarchy has a single process.

For each $v \in V, i \in P_k, i \in C$, and $C \in \mathcal{C}_k$ we define some derived notation:
(1) $lvl(i) = k$, the level of process i,
(2) $proc_0(v)$, the level 0 process at location v,
(3) $h(i)$, i's immediate clusterhead in the hierarchy, defined as i if $k = MAX$ and $j \in P_{k+1}$: $loc(j) = loc(head(C))$ otherwise,
(4) $h^n(i)$, the iterated clusterhead, defined as i if $n = 0$, $h(i)$ if $n = 1$ and $h(h^{n-1}(i))$ otherwise,
(5) $nbr(i)$, the set of neighbors of process i, defined as $\{j \in P_k : j \in C' \ \wedge \ (C, C') \in nbr_k\}$,
(6) $radius(C)$, the maximum distance from the clusterhead of C to any process in C,
(7) $children(i)$, the set of processes with i as their parent clusterhead in the hierarchy, defined as $\{i\}$ if $k = 0$ and $\{j : h(j) = i\}$ otherwise.

Geometry assumptions. To reason about algo-

rithms in this paper we fix geometry assumptions about the hierarchical partitioning:
(1) We define a real constant $r \geq 3$, the cluster dilation factor, used to determine cluster size,
(2) We bound the radius of a level l cluster C, using a real maximum cluster radius constant $m \geq 2/\sqrt{3}$, to be $r^l \leq radius(C) \leq mr^l$,
(3) Clusterheads of neighboring level l clusters are at least $2r^l$ and at most $2mr^l$ apart,
(4) The minimum distance between locations contained in two non-neighboring level l clusters is greater than qr^l, where q is a real minimum cluster breadth constant satisfying $\frac{2m+r-1}{r-1} \leq q \leq 2m$,
(5) The maximum distance between two locations in V is bounded by the network diameter D.

The constraints imply a bound, ω, on the number of neighbors at any level $l > 0$. They also imply that the distance is at most $2mr^{l-1}$ between two neighboring level l processes and mr^{l-1} between a level l process and its children in the hierarchy. 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 local clustering protocol, LOCI [15]. LOCI is also locally fault-tolerant and can recover from node failures and state corruptions.

3 System specification

Here we describe the specification for STALK.

Passage of time. Timers are real numbers advancing at the same rate at all automata. A timer does not advance until all enabled output or internal actions are completed –local processing time is negligible.

Mobile object specification. The mobile object is an IO automaton **Evader** with 2 output actions, $\mathbf{object}_i$ and $\mathbf{no\_object}_i$. It resides at a sensor location v, modeled by having $\mathbf{object}_i$ occur for each process i such that $loc(i) = v$ and no other processes. Other processes instead have **no_object** occur. **object** and **no_object** cannot occur at the same time at the same process.

When the mobile object moves from a location with a level 0 process i it may only move to a location corresponding to one of the neighbors of i. This movement is modeled nondeterministically.

STALK **specification.** STALK is a timed IO automaton that *a*) maintains the tracking structure by propagating mobile object information obtained through **object** and **no_object** inputs and *b*) answers client **find**s by outputting **found** at the mobile object's current location. STALK consists

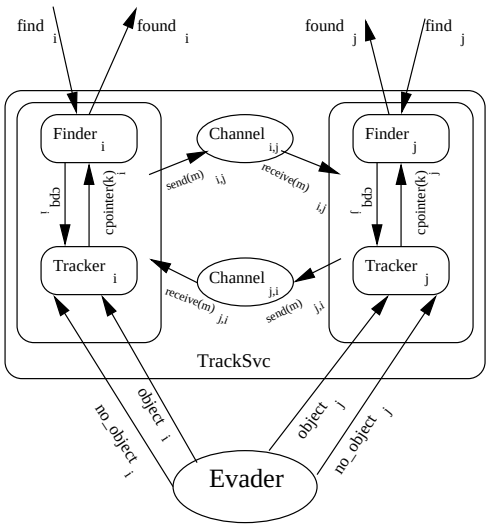


Figure 1. STALK **Subautomata**

of two subautomata as seen in Figure 1. It maintains the tracking structure through **Tracker** and services **find** requests through **Finder**. The automata are implemented on individual processes communicating through channels. **Tracker** takes as input sensor detections of the mobile object, modeled as $\mathbf{object}_i$ and $\mathbf{no\_object}_i$ actions, and answers **Finder cpq**s (request for informations) using **cpointer** actions. **Finder** takes as input client $\mathbf{find}_i$ requests and eventually answers them with $\mathbf{found}_j$ outputs.

Channel specification. We use a communication abstraction of a (possibly) multi-hop channel $\mathbf{Channel}_{i,j}$ between any two processes i and j, where $j \in nbr(i) \cup children(i) \cup \{h(i)\}$. Process i sends message m to j on the channel using the $\mathbf{send(m)}_{i,j}$ input action and process j receives the message with the $\mathbf{receive(m)}_{i,j}$ output action. Formally, the channel is a multiset. A $\mathbf{send(m)}_{i,j}$ adds one copy of message m into the

multiset and a **receive(m)**$_{i,j}$ removes one copy. 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. Automata can suffer from arbitrary state corruption indicated by **corrupt**$_i$. 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 specification. The complete system is the composition of automata described above — **Evader**, STALK$_i$ for each $i \in P$, and **Channel**$_{i,j}$ for each $i, j \in P$.

Consider starting from a consistent state with no outstanding **find** requests and no process or channel corruptions. The system satisfies:
(1) For each **find** event, tag **find**$_i$ in step x of the trace with the pair (i,x). Then, it is possible to tag each **found**$_k$ event at a step z of the trace with a pair (j,y) such that $lvl(k) = 0$, **object**$_k$ occurs at step z, (j,y) is the tag of some **find** event occurring before step z of the execution, and (j,y) is the tag for at most one **found** event,
(2) A **find** is eventually followed by a **found** event,
(3) Consider the last **find** event that occurs before a **found** event. The total time and work, or communication cost, to support that **find** action initiated at a process Euclidean distance d from the mobile object is at most $O(d)$,
(4) If the object moves d distance, amortized time and work performed by STALK is $O(d * log(D))$.

Lastly, the system is fault-local stabilizing.

4 Move operation

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 and climbing to increasingly higher 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 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 structure. 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.

Each process maintains a child pointer c, a parent pointer p, a grow timer *gtime*, and a shrink timer *stime*. In the initial states, for all $i \in P$, $i.c = \perp$, $i.p = \perp$, $i.gtime = \infty$, and $i.stime = \infty$, where $i.x$ indicates the value of the state variable x at process i. We use g and s, the grow and shrink timer constants, to calculate $gtime = g * r^{lvl(i)}$ and $stime = s * r^{lvl(i)}$ durations at each process i:

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

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

Tracker$_i$'s signature includes four input actions **object**$_i$, **no_object**$_i$, **cpq**$_i$, and **receive**$(msg)_{j,i}$ and two output actions **cpointer**$(j)_i$ and **send**$(msg)_{i,j}$ ($msg \in$ {gquery, ack_gquery, grow, shrink}). Action **cpointer**$(j)_i$ causes **Tracker**$_i$ to update **Finder**$_i$ with the current value of the c pointer in response to **Finder**$_i$'s **cpq**$_i$ input action. The **send** and **receive** actions are related to propagation of grows and shrinks and are explained in detail below.

4.1 Grow action

The grow action constructs a path to the new location of the object. If **object**$_i$ is fired at $i \in P_0$ and $i.c \neq i$, meaning process i is not already a leaf, then i becomes the leaf by setting $i.c := i$ and sets the grow timer, $gtime$. If a process receives a **grow** message, it sets its c pointer to the sender, sets $gtime$, and sends a **gquery** message to its neighbors to check if the original path is reachable through any of its neighbors. A neighbor j that receives the **gquery** sends an **ack_gquery** back if it is on the tracking path and has a parent pointer to its clusterhead, $h(j)$. This corresponds to j not having a lateral link, since only then it is possible to insert a lateral link from i to j.

When the grow timer expires, if $i.c$ is still non-$\perp$, meaning that the path has not shrunk while i's grow timer was counting down, the **send (grow)** action is enabled at i. If i had heard an **ack_gquery** from a neighbor j and had consequently set $i.p = j$ then the grow message is sent to j, inserting a lateral link. Else, i sets its parent to be its clusterhead, $h(i)$, and sends a **grow** message to its clusterhead. In either case $i.gtime$ is set to ∞ as i does not need to send more messages.

When a **grow** message is received at a process i, i need not propagate the grow further if i already has a parent in the tracking path or i is the MAX level process. If i is the MAX level process, then it sets its parent pointer to be itself. Otherwise, the above procedure is repeated: a grow timer is set, and a **gquery** message is sent to neighbors.

4.2 Shrink action

When a new path connects to the original tracking path at some level l, all levels below l in the original path become deadwood, an old branch of the tracking path deserted by the mobile object. The shrink action cleans deadwood.

If **no_object**$_i$ occurs, i removes itself from the path by setting $i.c$ to $\perp$ and setting its shrink timer. When the timer expires, if $i.c$ is still $\perp$, meaning no newer path has connected at i while i's shrink timer was counting down, i sends a **shrink** message to its parent in the path and sets $p := \perp$.

When i receives a **shrink** message from j, i checks to see whether $i.c = j$. Even though $j.p = i$, $i.c$ might not be pointing to j; $i.c$ may have been updated to point to a process on a newer path. If $i.c = j$ then i disassociates itself from the path and sets its shrink timer, scheduling a **shrink** message to be sent to $i.p$. Otherwise, if $i.c \neq j$, i simply ignores the message, ensuring that shrink actions clean only deadwood and not the entire path.

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.

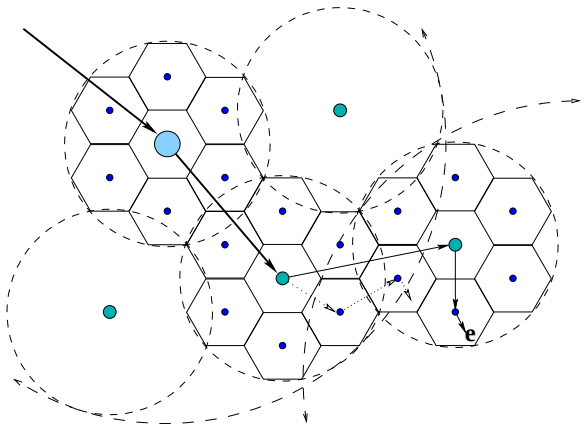


Figure 2. Tracking path example

4.3 Correctness

Here we present system invariants and define consistent states of the system. We show consistent states are closed under move operations and the system eventually reaches a consistent state.

In the absence of faults, we have shown (Theorem 4.5 in appendix) that 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's child pointer is not $\perp$, then one of the following holds: *(a)* $i.c = i$ and the object is at i, *(b)* $i.c$ points to a clusterhead contained in its cluster at the next lower level [i.e., $i.c \in children(i)$], or *(c)* $i.c$ points to a neighboring clusterhead at the same level and $i.p$ points to i's clusterhead at the next higher level,
(I2) If i's parent pointer is not $\perp$, then either i's child pointer is not $\perp$ or i is executing a shrink action and will send a **shrink** message to its parent,
(I3) This is the dual: if the child pointer of i is not $\perp$, then i's parent pointer is also not $\perp$ or i is executing a grow action and is about to send a **grow** message to its prospective parent,
(I4) If $i.c \neq \perp$ and $i.c \neq i$ then $(i.c).p$, the parent pointer of the process $i.c$ points to, is i or $\perp$ and a **shrink** message from $i.c$ is in transit to i. □

A *tracking path* is a sequence of processes $\{i_x, \ldots, i_1\}$ 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.

Lemma 4.6 *Consider execution α that starts from an initial state and contains $\mathbf{object}_i$ actions at one process i. α reaches a consistent state.* □

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

4.4 Work

In order to prove our work claims, we first show that the timing of changes to the tracking path satisfy certain relationships between the newer and older portions of the tracking path and ensure that original path is reused to the extent possible. More specifically, using our assumptions on s and g we prove in Lemma 4.8 that a shrink propagated from level 0 does not erase a level l pointer before a grow action can use it.

Building on this, we prove in Lemma 4.9 that lateral links are used as often as possible, and in Theorem 4.10 that a move to a distance d away results in $O(d * \log(D))$ time and work.

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 vertical growth to l,* [1]
4. *Path $\{i_x, \ldots, i_1\}$ 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.* □

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. Lemma 4.9 implies a level l pointer in the path is updated as often as every $q\sum_{j=1}^{l-2} r^j$ distance because of full extension of lateral links. A $O(mr^{l-1})$ work cost 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 up to MAX. □

5 Fault-containment

After state corruption of a region of (potentially all) processes, our tracking path heals itself in a

[1] A tracking path has *vertical growth* up to l if at every level $k, 0 < k < l$, the tracking path has one process at level k. A tracking path has *full extension* below l if at every level $k, 0 < k < l$, the tracking path contains 2 processes at level k.

fault-local manner, within work proportional to perturbation size. Here we present 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 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 actions 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.

Next we present the correction actions for reestablishing the tracking path invariant I, and then prove that they are fault-containing. The signature for the correction actions are in Figure 3. To correct $I4$ we use heartbeat messages and two timers: *next* for periodically sending heartbeats to the parent and *timeout* for dissociating a child if no heartbeat is heard. The correction actions (code is in the appendix) 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:

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

Signature:
Input: **receive(heartbeat)**$_{j,i}, j \in P$,
Output: **send(heartbeat)**$_{i,j}, j \in P$,
Internal: **heartbeat_set**$_i$
timeout_set$_i$
timeout_expire$_i$
start-shrink$_i$
start-grow$_i$

Figure 3. Fault containment signature at i

Correction actions for $I0$ and $I1$. $I0$ is established trivially by **object** and **no_object** actions. For correction of $I1$, we make domain assumptions 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 actions for $I2$ and $I3$. $I2$ is corrected by **start-shrink**$_i$. If i has a valid parent but no valid child, then this action schedules a **shrink** message to be sent to $i.p$. Similarly, **start-grow**$_i$ is responsible for correcting $I3$. If i has a valid child but no parent, then this action sends a **gquery** message to i's neighbors and schedules a **grow** message to be sent to the future parent of i.

Correction actions for $I4$. Heartbeats detect violations of $I4$. **send(heartbeat)**$_{i,p}$ ensures that every process i with a non-$\perp$ valued parent sends a **heartbeat** message to its parent every $b * r^{lvl(i)}$ time by setting *next*. **Receive (heartbeat)**$_{j,i}$ resets i's *timeout* variable to $(b + 2\delta m/r) * r^{lvl(i)}$ every time i receives a **heartbeat** message from its child, $i.c$. 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.

Timeout_expire$_i$ is enabled 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. It establishes $I4$ by setting $i.c$ to $\perp$.

Finally, **heartbeat_set**$_i$ and **timeout_set**$_i$ actions ensure stabilization of the *next* and *timeout* variables of the corrector by ensuring that their values are within their respective domains. In addition, in the **receive (grow)** action we reset "*timeout* to $(b + 2\delta m/r) * r^{lvl(i)}$," 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. In this case, receiving a **grow** message from a child is as good as receiving a **heartbeat**.

Stabilization proofs. We prove in Theorem 5.2, using Lemma 5.1, that our program is self-stabilizing to a consistent state, where a complete tracking path exists. Then, in Theorem 5.5 we prove that our program is fault-local stabilizing.

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

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

To prove fault-local stabilization we first give a bound on arresting distance of grow/shrink waves in Lemmas 5.3 and 5.4. For these lemmas, we assume 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.

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*

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

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*

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

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.

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 sketch. Uses the containment arguments in Lemmas 5.3 and 5.4. □

6 Find operation

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. The signature and code are in the appendix.

A find consists of two phases: *searching* and *tracing*. Searching follows pointers to 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 find is initiated by a **find** input. The level 0 process at that location starts servicing the request. A **find**$_i$ or **receive(find)**$_{j,i}$ is serviced at a process i by first querying the **Tracker**$_i$ automata, using **cpq**$_i$. When **Tracker**$_i$ returns a pointer c' through the **cpointer**$(j)_i$ action, **Finder**$_i$ automata considers the following cases: If $c' = i$, the object is found at i and the tracing phase is over, so i outputs **found**$_i$. If $c' \neq i$ and $c' \neq \perp$, the tracing phase is continuing, and i forwards the find request to the process pointed to by c'.

If $c' = \perp$ it is still the search phase, and i sends a **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 a **fqack** message and start processing the find if they are on the tracking path and ignore it otherwise. If a **fqack** is received from a neighbor j before the timeout period expires, the tracking path is found and tracing starts; i has effectively forwarded the find request to j. If

the timeout period expires with no reply from a neighbor, the search phase is continuing and i outputs $\mathbf{send(find)}_{i,h(i)}$, propagating the find to its clusterhead.

Work. Finds are local: a find initiated at process i distance d from the mobile object requires $O(d)$ work to complete. 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.* □

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 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 $i_{y,1}, y \neq 1$ 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 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. □

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

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

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

Theorem 7.5 *The tracking service with concurrent moves is self-stabilizing to a good state.* □

Theorem 7.6 (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.* □

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

In this section we discuss the execution of find operations while several concurrent move operations are still in progress on the tracking structure.

In the searching phase we show (Theorem 8.1 in appendix) that 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 does not move fast enough to result in propagation of a shrink to level $\lceil log_r d \rceil + 1$ before a find operation reaches it.

In a tracing phase 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 (Lemmas 8.2 and 8.3 in appendix). Together they show that 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 Simulation results

In the preceding sections we presented analytical worst-case performance of STALK. Here we consider a random movement model for the object and investigate the average-case performance through simulation. For simulations we use Prowler [19], an event-driven simulator for wireless sensor networks. 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/`.

Figure 11 (Appendix) 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.

Figure 12 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.

10 Concluding remarks

We presented STALK, a fault-local stabilizing tracking service for sensor networks. STALK is scalable: A *find* operation invoked within distance d of the mobile object requires $O(d)$ work to intercept the mobile object, and a *move* of the evader to distance d away requires $O(d * log(D))$ work to update the tracking structure.

We use two concepts to achieve fault locality: hierarchical partitioning and arresting waves. The key idea is to wait longer before updating a wider region's view by employing larger timeouts when propagating an update at higher levels of the hierarchy. This way, more recent updates from lower levels can catch-up to updates at higher levels.

STALK has applications in message routing to mobile units (tanks, platoons, or mobile agents) 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 [10]. For future work, we consider adapting STALK in the context of multiple pursuer - multiple evader problems where the objective is to coordinate the pursuers to collaborate in minimizing the *cumulative* cost of catching all the evaders. In addition, we are examining other problems that could benefit from our hierarchy-based local stabilization technique.

References

[1] A. Arora, P. Dutta, S. Bapat, and et. al. Line in the sand: A wireless sensor network for target detection, classification, and tracking. Technical Report OSU-CISRC-12/03-TR71, The Ohio State University, 2003.

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

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

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

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

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

[7] L. Guibas, J.-C. Latombe, S.M. LaValle, D. Lin, and R. Motwani. A visibility-based pursuit-evasion problem. *International Journal of Computational Geometry and Applications*, 9(4/5):471–481, 1999.

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

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

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

[11] 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.

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

[13] A. Kumar and T. Camp. A new matching algorithm for managing location information in mobile computing. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 231–239, 2000.

[14] 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.

[15] 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.

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

[17] D. M. Reid. An algorithm for tracking multiple targets. *IEEE Transactions on Automatic Control*, 1979.

[18] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target identities in wireless ad-hoc sensor networks. *Proceedings of 2nd International Workshop on Information Processing in Sensor Networks (IPSN)*, April 2003.

[19] G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *IEEE Aerospace Conference*, March 2003.

[20] 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.

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

[22] 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 4. 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 5. 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 6. 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 7. 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

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 8. Heartbeat actions at process i

process actions as discussed in Sections 5. $I0$ 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.

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:

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

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

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

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 $nbrtimeout = \infty$.

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*
- $\exists i \in nbr(h^{\lceil log_r d \rceil+1}(j))$ *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

Signature:

Input: $\mathbf{find}_i$
$\mathbf{receive}(msg)_{j,i}, j \in P$,
$msg \in$ {find, fqack, fquery}
$\mathbf{cpointer}(j)_i, j \in P \cup \{\perp\}$

Output: $\mathbf{found}_i$
$\mathbf{cpq}_i$
$\mathbf{send}(msg)_{i,j}, j \in P$
$msg \in$ {find, fqack, fquery}

Internal: $\mathbf{fqset}_i$

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 9. Finder signature/state at process i

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.

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

□

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

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
$nbrtimeout := now + 4\delta mr^{lvl(i)}$

Figure 10. Finder actions at process i

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 $2\delta m + \sum_{j=1}^{\lceil log_r d\rceil+1}(2\delta mr^j + \delta mr^{j-1})$ for a total time of $O(d * \delta rm)$. □

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: $s + \delta m + \sum_{j=1}^{l-1}[2sr^j + \delta mr^j + 2\delta mr^{j-1}] + sr^l + 2\delta mr^{l-1}$. 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 $\sum_{j=0}^{MAX-1}[gr^j + \delta mr^j]$ 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 $\sum_{j=0}^{l-1}[gr^j + \delta mr^j]$ 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: $ed > gr^l + \sum_{j=0}^{l-1} \delta mr^j$, 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. □

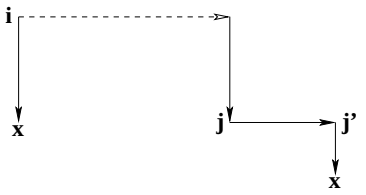
8 Concurrent find and move operations

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 $\sum_{j=0}^{\lceil log_r d \rceil} [\delta mr^j + 4\delta mr^{j-1}]$ time, there should still be a pointer at a neighbor (it takes up to $2\delta mr^{\lceil log_r d \rceil}$ 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:
$\sum_{j=0}^{\lceil log_r d \rceil} [\delta mr^j + 4\delta mr^{j-1}] + 2\delta mr^{\lceil log_r d \rceil} < e[qr^{\lceil log_r d \rceil} - d] + s + 2\sum_{j=1}^{\lceil log_r d \rceil} sr^j$.
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.

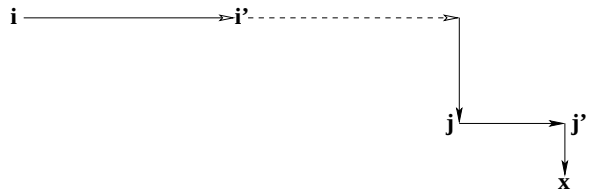


The maximum time a find takes in following the downward pointer at i to the dead end process, querying neighbors there, returning to i, querying neighbors of i, and following the newer path to j is: $5\delta mr^{l-1} + 4\delta mr^{l-1} + 4\delta mr^{l-2}$.

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: $sr^{l-1} > 9\delta mr^{l-1} + 4\delta mr^{l-2}$, 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: $5\delta mr^{l-1} + 4\delta mr^{l-1}$.



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: $sr^{l-1} > 9\delta m r^{l-1}$, 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 $\sum_{j=0}^{\lceil log_r d\rceil+1}(4mr^{j-1} + 4\omega mr^{j-1} + 5mr^{j-1} + 4\omega mr^{j-1} + 4\omega mr^{j-2})$, 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

In our experiments, we use a grid topology and employ a static, two-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) neighboring nodes.

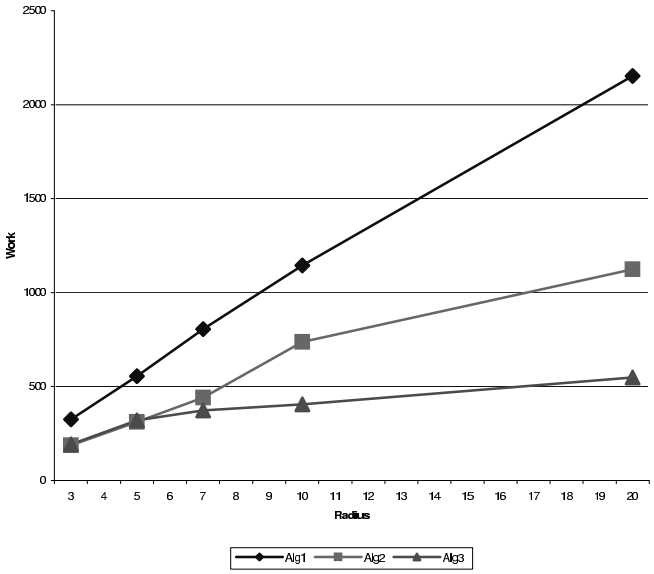


Figure 11. Work done with respect to r

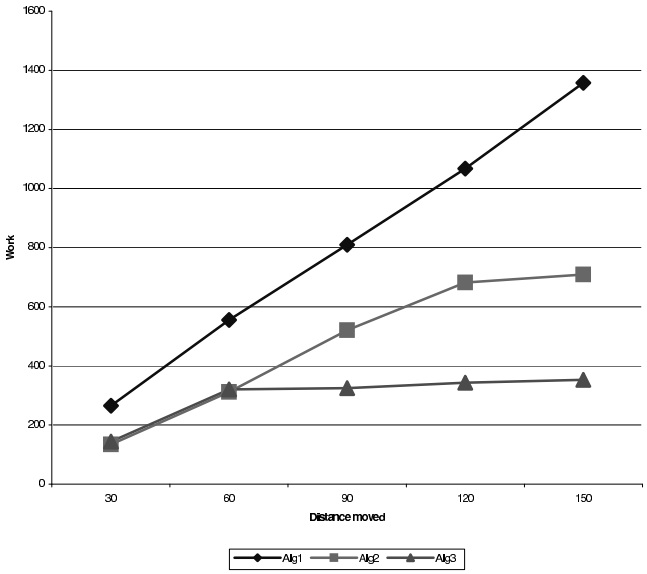


Figure 12. Work done with respect to d