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
{demirbas, anish}@cis.ohio-state.edu

Tina Nolte Nancy Lynch

MIT Computer Science &
Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
{tnolte, lynch}@theory.csail.mit.edu

Abstract

In this paper, we introduce the concept of *hierarchy-based fault-local stabilization* and a novel self-healing/fault-containment technique and apply them in STALK. STALK is an algorithm for tracking in sensor networks that maintains a data structure on top of an underlying hierarchical partitioning of the network. Starting from an arbitrarily corrupted state, STALK satisfies its specification within time and communication cost proportional to the size of the faulty region, defined in terms of levels of the hierarchy where faults have occurred. This local stabilization is achieved by slowing propagation of information as the levels of the hierarchy underlying STALK increase, enabling more recent information propagated by lower levels to override misinformation at higher levels before the misinformation is propagated more than a constant number of levels. In addition, this stabilization is achieved without reducing the efficiency or availability of the data structure when faults don't occur: 1) Operations to *find* the mobile object distance d away take $O(d)$ time and communication to complete, 2) Updates to the tracking structure after the object has moved a total of d distance take $O(d * log$ network diameter$)$ amortized time and communication to complete, 3) The tracked object may relocate without waiting for STALK to complete updates resulting from prior moves, and 4) The mobile object can move while a *find* is in progress.

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*

1 Introduction

In a distributed system, faults can occur that might be propagated throughout the system. In some systems, this propagation of errors might be unacceptable. Fault-containment or error confinement is concerned with preventing this propagation of faults beyond a small region. Exactly what is meant by "small" is defined as a polynomial of the *perturbation size*, an error severity measure. Previously, the perturbation size of a failure was defined in terms of the number of errors that occurred. This measure is convenient for expressing the seriousness of a processor fault in the execution of an algorithm as long as the algorithm does not incorporate use of processor hierarchies.

Hierarchies have long been imposed on networks of processors to facilitate design of efficient and scalable protocols. For example, Awerbuch and Peleg's tracking paper [6] described distributed directory servers to store location information for mobile objects. The directory servers were composed of a hierarchy of geographically defined regional directories where directories at higher levels of the hierarchy were responsible for maintaining information for larger regions of a network.

Another example of geographically defined hierarchies used in distributed systems are clusterings based on *hierarchical partitionings*. In such a system, all processes are divided into level zero clusters. Each of these clusters contain members that are close to one another geographically and have a defined clusterhead. These level zero clusterheads are then partitioned into level one clusters, again containing members that are close to one another, and so on.

Using traditional definitions of perturbation size, a fault that occurs at a single level zero process during execution of a hierarchy-based algorithm has the same size as that of a fault of a single level ten process. As a result, a fault-containing algorithm would have to prevent propagation of information beyond an area whose size is a polynomial based on perturbation size one. This kind of level-blind fault-containment is not always possible. Instead, it can be useful to define perturbation size and fault-containment in terms of the hierarchy. Perturbation size would be defined in terms of levels where errors occurred, and a fault-containing algorithm would be required to not propagate faults more than a small number of levels in the hierarchy. In this paper, we define such a notion and use it to evaluate an algorithm for tracking a mobile object.

Because of the recent growth of applications in mobile computing, cellular telephony, and military contexts, tracking of mobile objects has recently received significant attention [6, 8, 12, 20, 22]. 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 (100 node networks) tracking demonstrations: pursuer-evader tracking with one human controlled evader and three autonomous pursuers is showcased in [21], and detection, classification, and tracking of various intruders, such as persons and cars, are demonstrated in [3].

In addition to the opportunities they provide for tracking of objects, wireless sensor networks also impose additional challenges. Sensor nodes are energy-constrained, and algorithms that require excessive communication 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 [17]. On-site maintenance is infeasible and hence sensor networks should be self-healing. Moreover self-healing should achieve fault-containment to prevent a fault in one region of the network from contaminating the entire network and requiring a global correction, wasting the energy of the nodes and reducing the availability of the tracking service.

Contributions. Our novel contribution is to present a hierarchy-based self-healing/fault-containment technique and then demonstrate the concept with an algorithm for tracking in sensor networks, which we call STALK (Stabilizing Tracking viA Layered linKs). To achieve scalability, STALK employs a hierarchical tracking structure. The tracking structure is a path imposed on an underlying hierarchical partitioning of the sensor network into clusters, such as those provided by the self-stabilizing algorithm described in [18]. We implement updates to the tracking structure 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 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 the fact that grow and shrink occur concurrently, we complete the move operation successfully by using suitably-chosen timers to determine when these actions are performed.

STALK is hierarchy-based fault-containing, preventing propagation of faults in the tracking structure beyond a small number of levels in the hierarchy. Starting from an arbitrarily corrupted state, it satisfies its specification in time and work proportional to perturbation size, defined in terms of levels (as defined by the underlying hierarchy) where faults have occurred. 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.

STALK provides good locality guarantees; a *move* of the object being tracked to distance d away requires $O(d * logD)$ time and communication (work) to update the tracking structure, where D is the network diameter. In the full version of our paper [11] we also describe a *find* operation using the tracking structure. A find operation invoked at a process queries neighboring processes at increasingly higher levels of the clustering hierarchy until it encounters a process on the tracking path. Once the path is found, the find operation follows it to its leaf to reach the mobile object. In the full version we also show that a *find* invoked within distance d of the mobile object requires $O(d)$ work to reach the object and that when no faults occur, our scheme for achieving fault-containment does not increase the complexity of tracking or finding. Furthermore, we show that STALK achieves seamless tracking of a continuously moving object by allowing concurrent tracking and finding operations. For space reasons, we refer the reader to the full version [11] for these results and instead concentrate here on the tracking program actions of STALK and fault-containment.

Related work. The idea of employing a hierarchical structure for achieving scalability of tracking has been extensively researched. The idea of using a partial information strategy to optimize both finds and moves in a relatively static point-to-point network was investigated in [6]. In [6], 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). However, a topology change, such as a node failure, necessitates a global reset of the system since the regional directories depend on a non-local clustering program [5] that constructs a sparse cover of a graph.

In [9], the tracking problem is considered for a geometric network model similar to ours, and cost complexity similar to ours is achieved. However, the tracking structure maintained is not available during moves of mobile objects and the program for finding a mobile object is only implicitly defined. This algorithm is also not fault-tolerant. Papers such as [2,23] are concerned with non-stabilizing solutions for personal communication systems and the mobile Internet Protocol, not sensor networks. A location service for ad hoc networks is described in [1] and provides attractive worst case and average case costs and provides some fault-tolerance, though it is not fault-containing.

There has been work on self-stabilizing, though not fault-containing, tracking algorithms [10, 12, 15]. The distributed arrow protocol [15] is one such algorithm but 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 [10] do not exploit the hierarchy idea and are not scalable for large networks. In [12], using a hierarchy of location servers, a stabilizing location management protocol is presented. However, the protocol in [12] does not ensure locality of finds. In [14] another self-stabilizing algorithm using hierarchies to solve a problem close to tracking is presented, though it too is not fault-containing.

Fault-containment of self-stabilizing algorithms in general has received growing interest [4, 7, 13, 19], though none of these algorithms use a hierarchy-based concept of fault-containment. The notion of fault containment within the context of stabilization was first formalized in [13]; algorithms were proposed to contain state-corruption of a single node in a stabilizing spanning tree protocol. In [19] fault-containment of Byzantine nodes was studied in dining philosophers and graph coloring algorithms; this work required the range of contamination to be constant and is too limiting for problems such as tracking and routing where locality is not constant. In [7], a broadcast protocol was proposed to contain observable variables in the presence of state corruptions, but the protocol allowed for global propagation of internal protocol variables. Another protocol that achieved fault-local stabilization in shortest path routing was presented in [4]. To achieve fault-containment the protocol used privileged containment actions that were a constant time faster than the fault-intolerant program actions.

Organization of the paper. After presenting the model in the next section, we present the specifications of STALK and a definition for hierarchy-based fault-localization in Section 3. In Section 4, we present the move operation. Fault local stabilization actions for the tracking path are discussed in

Section 5. Finally we conclude our paper in Section 6. For space reasons, we relegate detailed proofs to the Technical Report [11].

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 the system.

Mobile object. The mobile object **Evader** resides at exactly one sensor location. We model the **Evader** using **object** and **no_object** inputs at processes: An $\mathbf{object}_i$ occurs at all processes residing at the object's current location and $\mathbf{no\_object}_j$ 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** would query **Tracker** for location information through **cpq** requests and **Tracker** would answer with **cpointer** responses.

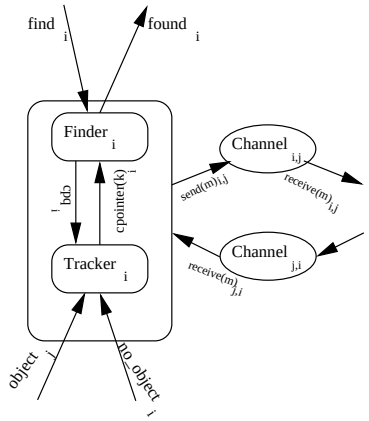

Figure 1. STALK **architecture at process** i

STALK is implemented distributively 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 $\mathbf{Channel}_{i,j}$ between any two processes i and j. Such channels are accessed using $\mathbf{send(m)}_{i,j}$ to send from i and $\mathbf{receive(m)}_{i,j}$ to receive at j. The cost of sending a message through $\mathbf{Channel}_{i,j}$ 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 the *perturbation size* of a system to be a weighted sum of the sizes 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 we then require that if the object moves d distance, the amortized time and work to update the tracking structure is $O(d * log(D))$. (Other guarantees and requirements relating to *find*s and concurrent tracking and finding are discussed in the Tech Report [11].)

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 [11], 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.

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:

$$s \geq 10.5\delta m \tag{1}$$

$$\frac{s + \delta m}{r} < g \leq s - \delta m \tag{2}$$

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.

Signature:	State:
Input: $\mathbf{object}_i$	$c \in P \cup \{\perp\}$, initially $\perp$
$\mathbf{no\_object}_i$	$p \in P \cup \{\perp\}$, initially $\perp$
$\mathbf{cpq}_i$	$gqack \in P \cup \{\perp\}$, initially $\perp$
$\mathbf{receive}(msg)_{j,i}, j \in P$,	$gnbrquery \subseteq P$, initially $\emptyset$
$msg \in$ {gquery, ack_gquery, grow, shrink}	$update$, a Boolean, initially $false$
Output: $\mathbf{send}(msg)_{i,j}, j \in P$,	$gtime \in \Re$, a timer, initially ∞
$msg \in$ {gquery, ack_gquery, grow, shrink}	$stime \in \Re$, a timer, initially ∞
$\mathbf{cpointer}(j)_i, j \in P \cup \{\perp\}$	$now \in \Re$, a timer indicating current time at i

Figure 2. Signature and state of $\mathbf{Tracker}_i$

$\mathbf{Tracker}_i$ 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

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 3. Grow actions at process i

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

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 4. Shrink actions at process i

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 5 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.

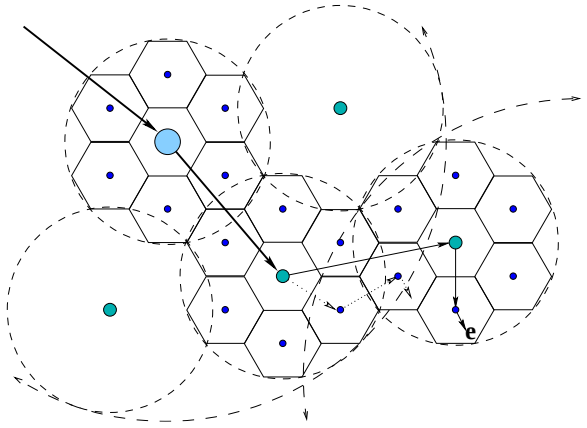

Figure 5. Tracking path example

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 and that consistent states are closed under moves of the object.

In the case where the evader can relocate before updates have been completed it is necessary to relax the definition of a tracking path and instead define a more general *tracking structure* describing path segments that satisfy certain reachability conditions. Details can be found in the Technical Report.

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; hierarchy-based 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.

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

Figure 6. Starting grow/shrink at process i

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

Correction actions for $I0$ 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.[1]

Theorem 5.2 STALK *is self-stabilizing.* □

[1] In the case where the evader can relocate before updates are completed, the algorithm self-stabilizes to a state where a more general tracking structure exists, as mentioned in Section 4.3.

Fault-local stabilization. To prove hierarchy-based 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 proofs for both lemmas are done by comparing the maximum time the propagation of a lower wave takes to reach level 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*

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

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 [11] we provide values that satisfy these requirements, as well as a number of others ($g = 5\delta m, s = 11\delta m, b = 11\delta mr$).

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

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

Proof sketch. 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 a constant number of levels in the hierarchy, or $O(r^{lvl(i)})$ time and work.

The work for fault-containment is additive: summation of the work for all perturbed processes gives the work for the system. However, since fault-containment takes place concurrently for all perturbed processes, the fault-containment time $O(r^L)$ for the highest level perturbed process (at level L) dominates, giving at most $O(r^L)$ time. □

6 Concluding remarks

We presented STALK, a hierarchy-based fault-local stabilizing tracking service for sensor networks. We use two concepts to achieve hierarchy-based 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 within a constant number of levels above the fault. 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. This last point is described more fully in our Technical Report [11].

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 [16]. For future work, we are examining other problems that could benefit from our hierarchy-based local stabilization technique.

References

[1] I. Abraham, D. Dolev, and D. Malkhi. LLS: a locality aware location service for mobile ad hoc networks. *Manuscript*, 2004.

[2] 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.

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

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

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

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

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

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

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

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

[11] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. STALK: A self-stabilizing hierarchical tracking service for sensor networks. Technical Report OSU-CISRC-4/03-TR19, The Ohio State University, April 2003.

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

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

[14] M. Herlihy and Y. Sun. A location-aware concurrent mobile object directory for ad-hoc networks. *Manuscript*, 2004.

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

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

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

[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] 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.

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

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