International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 122–158, January 2015

Near-Optimal Location Tracking Using Sensor Networks

Gokarna Sharma
Louisiana State University, USA
gokarna@csc.lsu.edu

Hari Krishnan
Louisiana State University, USA
hkrish4@tigers.lsu.edu

Costas Busch
Louisiana State University, USA
busch@csc.lsu.edu

and

Steven R. Brandt
Louisiana State University, USA
sbrandt@cct.lsu.edu

Received: August 1, 2014
Revised: November 4, 2014
Accepted: December 3, 2014
Communicated by Akihiro Fujiwara

Abstract

We consider the problem of tracking mobile objects using a sensor network. We present a distributed tracking algorithm, called Mobile Object Tracking using Sensors (MOT), that scales well with the number of sensors and also with the number of mobile objects. MOT maintains a hierarchical structure of detection lists of objects that can efficiently track mobile objects and resolve object queries at any time. MOT guarantees that the cost to update (or maintain) its data structures will be at most $\mathcal{O}(\min\{\log n, \log D\})$ times the optimal update cost and query cost will be within $\mathcal{O}(1)$ of the optimal query cost in the constant-doubling graph model, where n and D, respectively, are the number of nodes and the diameter of the network. MOT achieves polylogarithmic approximations for both costs in the general graph model. MOT balances the object and bookkeeping information load at each node in the expense of only $\mathcal{O}(\log n)$ increase in the update and query costs. The experimentation evaluation in both one by one and concurrent execution situations shows that MOT performs well in practical scenarios. To the best of our knowledge, MOT is the first algorithm for this problem in a distributed setting that is traffic-oblivious, i.e. agnostic to a priori knowledge of objects movement patterns, mobility and query rate, etc., and is load balanced. All previous solutions for this problem assumed traffic-consciousness in constructing the tracking data structure.

Keywords: Sensor networks, Mobile objects, Hierarchical structure, Location tracking, Cost ratio

1 Introduction

We consider the problem of tracking locations of mobile objects using a sensor network so that the presence of particular (mobile) objects (animals, vehicles, etc.) can be detected by nearby sensors [5, 7, 18, 21–23, 36, 37]. This kind of location tracking of mobile objects is a very important problem and has many applications in different areas including military intrusion detection, vehicular networks, and habitat monitoring, e.g. [9, 13, 16, 25, 34]. Our objective in this location tracking problem is to track the mobile objects in such a way that any node in the network can locate any object of interest, among the set of objects available in the network, at any time with the minimum cost possible. This tracking problem needs a data structure to achieve location tracking objective with low cost and the data structure needs to be updated frequently due to mobility of objects. The updates in (i.e., maintenance of) the data structure when objects move from one location to another is done through *maintenance* operations and the objects of interest are located whenever needed through *query* operations in the maintained data structure. By "locating an object of interest" we mean the finding of the sensor node that has (i.e., detects) the object of interest. This problem is similar to the in-network data processing problem studied in several prior papers, e.g. [18, 38].

We model the problem of tracking mobile objects by a weighted graph G, where graph nodes correspond to sensor nodes and graph edges correspond to adjacencies between sensor nodes. Each sensor node has its *detection range*. Two sensors are said to be *adjacent* if objects can move from the detection range of one of the two sensors to that of the other without needing to transit through any third sensor. A sensor node that currently has (or detects) a mobile object is called the *proxy node* for that object. We consider a *nearest-sensor* model in which the sensor that is nearest to the object becomes its proxy [6, 21] breaking ties arbitrarily, i.e., a node that receives the strongest signal from the object becomes the proxy node for that object. The proxy nodes change over time as objects move. Proxy nodes are the subset of the sensor nodes which currently have (at least) a detected object. The objective is to find the proxy nodes in G.

Consider a set of m different mobile objects in a sensor network G. What we mean here by different mobile objects is that they are distinguishable from each other, i.e., they have different IDs. Theoretically, the lowest tracking costs for these mobile objects possible would be as follows: (1) The location maintenance must have a cost equal to the minimum distance the objects traverse if they followed shortest paths in G, and (2) Any query must be answered with a cost that is proportional to the shortest distance from the requesting node to the proxy node. We give a tracking algorithm which is efficient in both these costs.

1.1 Our Results

We measure the costs of maintenance and query operations of any tracking algorithm as follows.

- **Maintenance cost:** We measure the cost of a *maintenance* operation with respect to the *communication cost*, which is the total number of messages sent in the network G by a tracking algorithm. We assume that total number of messages sent in the network are proportional to the total distance between the sender and receiver node in the tracking data structure. Therefore, when an object moves from one proxy node to another proxy node, the optimal communication cost to update the location of that object is at least the distance between these proxy nodes. This is because the distance between the old and new proxy nodes is the actual minimum distance in G the object traverses and hence any tracking algorithm needs to send messages at least this distance to reflect the location change. We compare the communication cost of a tracking algorithm for a set of *maintenance* operations to the optimal communication cost for that set of operations to obtain the *maintenance cost ratio*, which is the approximation of the tracking algorithm for location maintenance of mobile objects.

- **Query cost:** We compare the communication cost of a tracking algorithm for a *query* operation to the optimal communication cost for that operation to obtain the *query cost ratio*, which is the approximation of the tracking algorithm for queries to locate objects of interests.

In this present work, we present a new distributed tracking algorithm, called "Mobile Object Tracking using sensors" (MOT), that scales well with the number of sensors and with the number of mobile objects. MOT essentially maintains a data structure as we mentioned above for the tracking task. When applied to *constant-doubling networks*, which have been widely used as a model of a sensor network in the literature (e.g., [7, 11, 12, 27, 40]), MOT provides a cost ratio of $\mathcal{O}(\min\{\log n, \log D\})$ in maintaining the data structure for any arbitrary set of *maintenance* operations comparing its communication cost to the optimal cost, where n and D, respectively, are the number of nodes and the diameter of the network. This result assumes the case where each *maintenance* operation of an object arrives only after the previous *maintenance* operation for that object is finished. This "one by one case" is the scenario where event inter-arrival times are large compared with the message propagation times [12, 18]. The cost ratio given above considers a sequence of *maintenance* operations because they provide small amortized maintenance cost compared to the analysis of a single *maintenance* operation.

MOT achieves the *query* cost ratio of $\mathcal{O}(1)$ in constant-doubling networks, i.e. if the object of interest is at a proxy node u at distance d from the query node v then MOT finds that object with the total cost of only $\mathcal{O}(d)$. We consider *query* operations individually and they have always small cost. This *query* cost ratio is optimal within a constant factor.

Moreover, we analyze the *load* of MOT through the total number of objects as well as the bookkeeping information it stores in the network nodes of data structure it maintains for the tracking purpose. The average load on each node is $\mathcal{O}(\log D)$ in constant-doubling networks, independent of the number of objects m. This load balancing property of MOT is obtained in the expense of only a $\mathcal{O}(\log n)$ factor increase in the *maintenance* and *query* cost ratios given above.

These results for MOT can also be extended to general networks. For any set of *maintenance* operations, MOT provides a cost ratio of $\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$, and for any *query* operation, it provides a cost ratio of $\mathcal{O}(\log^4 n)$. This *maintenance* cost ratio is within a poly-log factor of $\Omega(\frac{\log n}{\log \log n})$, the lower bound given in Alon *et al.* [2] for a similar problem. These results also assume the one by one case. Moreover, the average load in network nodes is only $\mathcal{O}(\log^2 n \cdot \log D)$, assuming that the maximum number of objects stored at any node is polynomially bounded in the number of nodes n in the network. This load balancing property of MOT increases the *maintenance* and *query* costs given above by a factor of $\mathcal{O}(\log n)$ only.

The *maintenance* cost ratio results given above for the one by one case in both constant-doubling and general network models extend also to "concurrent case" where a *maintenance* operation for an object may arrive before previous *maintenance* operation for that object is finished. The analysis for this concurrent case makes use of the technique presented in [30]. This implies that our tracking algorithm MOT is applicable in any object movement situations. Moreover, the *query* cost ratio results given above for both constant-doubling and general network models assume that *query* operations for an object do not overlap with any *maintenance* operation of that object. Even in the case where a *query* operation overlaps with one or more *maintenance* operations, using the concurrent analysis technique given in [32], we can prove that same cost ratios can be obtained for queries, making our algorithm applicable for queries in any execution situation.

All of our *maintenance* and *query* cost ratio results are deterministic worst-case bounds and outperform existing results presented in [18, 21, 23, 37]. The detailed comparison of MOT results with the prior results is given in Section 1.3 but we provide here highlights of the benefits of our algorithm in brief. Our algorithm is the first solution to this tracking problem in a distributed setting that is traffic-oblivious, i.e., our algorithm is agnostic to a priori knowledge of objects movement patterns, mobility and query rates, etc. All previous solutions [18, 21, 23, 37] to this problem assumed traffic-consciousness in constructing the tracking data structure. The traffic-obliviousness property of our solution makes it suitable for object tracking in environments where a priori knowledge of object movement patterns, mobility and query rates, etc. is not available or difficult to obtain. Moreover, our algorithm balances the object load in the nodes of the tracking data structure whereas previous solutions do not address this issue.

Finally, we supply simulations which demonstrate that MOT performs well compared to previous approaches in real-world scenarios, besides its theoretical guarantees. The simulations are performed considering both one by one and concurrent execution situations of object operations to cover wide

range of practical applications. We performed the simulations in grid networks of sizes ranging from 10 nodes to 1024 nodes using 100 and 1000 mobile objects. We compared the performance of MOT with two previous approaches: STUN algorithm due to Kung and Vlah [18], and Z-DAT and Z-DAT with shortcuts algorithms due to Lin *et al.* [21] (the details on how these algorithms work is given in Section 1.3). To see how theoretical properties translate to practice, the performance metrics that are considered in the comparison are cost ratios of *maintenance* and *query* operations, and object and bookkeeping information load at network nodes. Note that all previous algorithms considered in the experimental evaluation in this paper (i.e., STUN, Z-DAT, and Z-DAT with shortcuts) assume traffic-consciousness in constructing their tracking data structure, whereas our algorithm MOT is traffic-oblivious.

1.2 Techniques

Our technique is to employ a hierarchical structure for the tracking data structure such that this structure can provide efficient tracking despite object mobility. This hierarchical structure is updated appropriately after every time objects move so that they can be queried with low cost whenever needed. We assume, similar to [18], that the maximum distance any object can traverse in a given amount of time is bounded. This allows us to efficiently update the hierarchical structure we maintain.

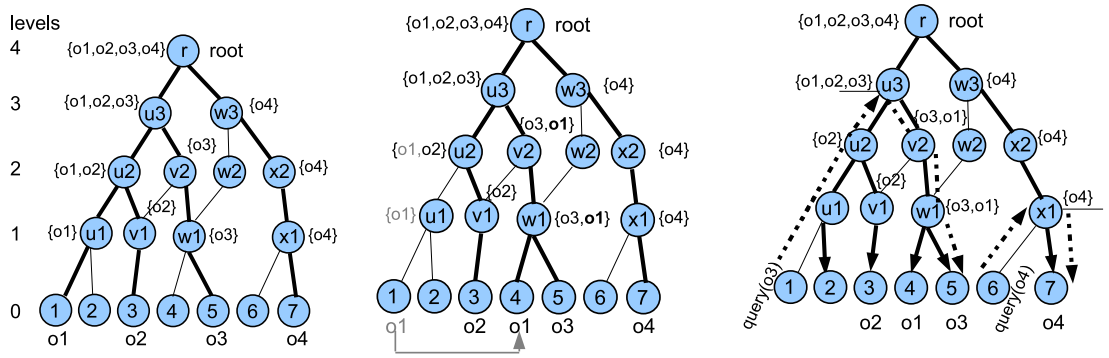
We construct an *overlay structure* $\mathcal{HS}$ on the sensor network G as a tracking data structure. This structure $\mathcal{HS}$ has many levels where the top-most level is called the root level and the lower-most level is called the bottom level. All sensors nodes of G act as the bottom level nodes in $\mathcal{HS}$ and a subset of sensor nodes of G act as nodes in the other levels in $\mathcal{HS}$. Some of the sensor nodes (possibly all) of G that are in the bottom level of $\mathcal{HS}$ are the proxy nodes when they detect at least an object, i.e., proxy nodes are always in the bottom level of $\mathcal{HS}$. A normal sensor node (not proxy) at bottom level of $\mathcal{HS}$ becomes a proxy node when it detects (at least) an object in its detection range and a current proxy node becomes a normal sensor node when all the objects in its detection range are moved from it to other proxy or normal sensor nodes. Moreover, a sensor node of G may act as a node in all the levels of $\mathcal{HS}$. The nodes that are in all the levels up to the root level except the bottom level are called *internal nodes*. These internal nodes are used to to store the presence of the detected objects and to communicate with other nodes. For example, when a proxy node at the bottom level also acts as an internal node in $\mathcal{HS}$, then when it performs operations as an internal node it can only store the detected objects that are in the detection lists of its child nodes in $\mathcal{HS}$.

$\mathcal{HS}$ has a root node, which we denote by r. In a real application, the sink node is often the root of $\mathcal{HS}$. Each internal node stores the set of objects that were detected by its descendants. The set of objects at any node (proxy or internal) is called the *detection set* of that node. Therefore, for a proxy node, the detection set is the set of objects in its detection range, but for the root node, the detection set is the set of objects that are available in the whole region.

$\mathcal{HS}$ is initialized by having the proxy nodes send detection messages towards the root of $\mathcal{HS}$. An internal node that is visited by a detection message adds the object associated with that message in its detection list and passes that message to its ancestor internal node, until that message reaches the root.

Now when an object moves, the new proxy node of that object sends a detection message towards that root. When a detection message is received, there are two possible outcomes. First, if the node does not yet have a reference for the object, it adds it to its list, and forwards the message to its parent. Second, if it already has a reference, it sends a delete message to the old proxy node, and does not forward the message to its parent. The delete message propagates downward until all traces of the old location are erased.

Consider Fig. 1 for the illustration of this approach. Fig. 1a depicts the initialization of $\mathcal{HS}$ in the beginning of tracking. Proxy nodes 1, 3, 5, and 7 send detection messages up to the root r. The internal nodes visited by those messages add the objects to their detection lists. When an object o_1 moves from node 1 to node 4, detection messages are sent towards r from node 4. The internal nodes w_1 and v_2 add o_1 in their detection list. The detection messages are terminated by u_3 as o_1 does not modify its detection list. The object o_1 is also removed from the detection lists of u_1



(a) Initialization of $\mathcal{HS}$ (b) $\mathcal{HS}$ after object o_1 moves to 4 (c) Illustration of queries

Figure 1: Illustration of the mobile object tracking problem.

and u_2 to keep the detection sets up to date. The objects are removed through delete messages by traversing the internal nodes towards the bottom level which have the objects that need to be removed in their detection lists. The upward propagation of detection messages for any object ends at a common ancestor (u_3 in Fig. 1b for o_1 that moved from 1 to 4).

The purpose of maintaining the detection sets is to allow efficient querying. A query that is routed from a requesting node to an object o_i first sends a query message upward towards the root until an internal node which has the object o_i in its detection list is found. After that the query is routed downward from that internal node following child nodes which have o_i in their detection lists until the query reaches the proxy node of o_i. Fig. 1c shows how queries from nodes 1 and 6 for the objects o_3 and o_4, respectively, are served using detection lists of internal nodes of $\mathcal{HS}$. Without the information contained in the detection sets of internal nodes, the queries would need to be flooded to all proxy nodes, which would increase the cost dramatically.

If a query arrives while a move request is being processed, it may arrive at an incorrect proxy node and fail to find the desired object. In this case, the query pauses and waits for the delete message to arrive. The delete message will contain the id of the correct proxy node. In this way, queries can be successful even while a move is in progress. This is one simple approach to solve this problem of a query arriving at an incorrect proxy node. We can have improved algorithm to solve this problem without ever reaching the incorrect proxy node. This improved algorithm will use the fact that queries can be directed to correct proxy nodes without reaching proxy nodes in the bottom level and waiting for the delete message to arrive.

Following the tracking approach based in $\mathcal{HS}$ described above, the internal nodes in higher levels of $\mathcal{HS}$ bear a greater load in the sense that the number of objects and the bookkeeping information that needs to be stored in their detection lists can be $\mathcal{O}(m)$ for m different mobile objects. Alternatively, the load can be evenly shared by letting the internal nodes distribute the object and bookkeeping information to other nodes in their vicinity. To facilitate this, We embed a de Bruijn graph [19] of appropriate size using the neighborhood of every internal node. Using this embedding, we guide the *maintenance* and *query* requests within the neighborhood to relevant objects via hashed pointers. The search process within any neighborhood is efficient due to the properties of the embedded de Bruijn graph. Note that we will consider neighborhood sizes that grow proportionally (2^i for level i) to the level of $\mathcal{HS}$.

We now briefly describe how we construct $\mathcal{HS}$ which plays a vital role in the performance of MOT. We construct $\mathcal{HS}$ organizing nodes of G into levels using a simple distributed maximal independent set algorithm [24] for the constant-doubling network model. According to this construction, $\mathcal{HS}$ has $\mathcal{O}(\log D)$ levels, where at each level there are leader nodes. The leaders of higher levels are subsets or refinements of leaders at lower levels. Leader nodes in each level i cover all the nodes within a 2^i radius, we call these nodes a *node cluster*. All nodes of G are, by default, leaders in the proxy level (level 0) of $\mathcal{HS}$ and there is a single leader node of G at the top level (level $h = \mathcal{O}(\log D)$) which is called the *root*. For the general networks model, we use the $(\mathcal{O}(\log n), \mathcal{O}(\log n))$ partition scheme of

[4, 15, 33] such that there are $\mathcal{O}(\log D)$ levels in $\mathcal{HS}$ and each node of G belongs to exactly $\mathcal{O}(\log n)$ node clusters with the radius of each node cluster $\mathcal{O}(2^i \cdot \log n)$.

1.3 Related Work

One prominent feature of MOT is that it is *traffic-oblivious* while constructing the overlay structure $\mathcal{HS}$, i.e., it is agnostic to any a priori knowledge of object movement patterns, mobility and query rates, etc. All of the prior techniques [18, 21, 23, 37] are traffic-conscious, i.e., these techniques take into consideration the object movement patterns in building their structures to achieve efficiency in tracking. The object movement patterns are measured with respect to how many times objects move between adjacent sensor nodes, which is referred to as the *detection rate*. Each edge of G is assigned the weight that represents the frequency of object movement between a pair of adjacent sensors. Now to yield low communication cost, internal nodes near the root act as the connecting points for the adjacent sensors that have low detection rates, and internal nodes near the bottom level act as the connecting points of the adjacent sensors that have high detection rates. Therefore, prior techniques may not necessarily minimize the communication costs of maintenance and query operations where such information is not available or difficult to compute. Recall that our approach builds the hierarchy without any a priori knowledge of detection rates.

Hierarchical structures developed in prior techniques are mostly *spanning trees* of the sensor nodes in G (also called *message-pruning trees*). Due to the use of spanning trees, cost ratios for maintenance and query operations can be as large as $\mathcal{O}(D)$ in those approaches, e.g. in ring networks, where D is the diameter of the network G. Similarly, prior approaches do not balance the load; the number of objects that need to be stored at any node of the message-pruning tree can be $\mathcal{O}(m)$. Our approach balances this load as well.

Kung and Vlah [18] were the first to study this location tracking problem of mobile objects. They proposed a tracking algorithm, called *Scalable Tracking Using Networked sensors* (STUN), based on a technique called *Drain-And-Balance* (DAB) which constructs a hierarchical tree using detection rate information. A subset of sensor nodes are merged into balanced subtrees until all nodes are included in the hierarchical tree. Subsets are obtained by partitioning the sensors of G using appropriate detection rate thresholds and high detection rate subsets are merged first. However, DAB does not take the *query* cost into account and hence it may not minimize the *query* cost in many cases. Lin *et al.* [21] extended the technique of [18] by proposing two new techniques: *Deviation-Avoidance Tree* (DAT) and *Zone-based Deviation-Avoidance Tree* (Z-DAT). They build DAT and Z-DAT trees using a two stage approach such that they can focus on *maintenance* cost minimization in the first stage and *query* cost minimization in the second stage. In DAT, they arrange edges in decreasing order from the highest weight to the lowest weight and connect the highest weight edges first. In Z-DAT, they divide the sensing region into rectangular zones, and recursively combine those zones into a tree. Later, Lin *et al.* [22] extended the approach of [21] to multi-sink sensor networks.

Liu *et al.* [23] used a hierarchical tree called a *message-pruning tree with shortcuts*, which is a modification of the tree developed in [18, 21], to further minimize the *maintenance* and *query* costs. Later, Yen *et al.* [37] developed a technique to remove the assumptions of [18, 21, 23] on traffic-consciousness. However, they derived a traffic-based knowledge using a mathematical model on topological information. Note that this knowledge may not necessarily reflect the real traffic pattern. Our approach does not need any traffic knowledge, neither the derived traffic knowledge of [37] nor the real traffic knowledge of [18, 21, 23]. Moreover, it is not clear in these works how concurrent situations in the execution of object operations are handled.

Finally, we would like to note here that this present work is different from several problems related to tracking solved in the following papers [1, 3, 6, 8, 10, 17, 20, 26, 35, 39]. The tracking schemes presented in [3, 6, 26, 39] only try to estimate the trajectory path of the object while it is moving. These works [8, 35] try to locate nearest sensor nodes which do not generally change over time. These papers [1, 10, 17, 20] studied the problem of providing a location service for ad hoc networks to allow any source node to know the location of any destination node whose location is unknown.

1.4 Paper Organization

The rest of the paper is organized as follows. We discuss the model and the hierarchical structure construction details in Section 2. We present details of our MOT algorithm in Section 3 and its analysis in Section 4. We then discuss how to extend MOT to achieve load balancing in Section 5. We provide extensions to general graphs in Section 6. After that we discuss how to extend MOT to handle dynamism in networks in Section 7 and provide some experimental results in Section 8. We conclude the paper in Section 9 with a short discussion.

2 Preliminaries

2.1 Model

We consider a static graph $G = (V, E, \mathfrak{w})$ to represent the topology of the sensors deployed in a region, where V indicates the vertices (a set of n sensor nodes), E indicates edges (two sensors are connected by an edge if an object can pass directly from either of them to the other without going through a third sensor), and $\mathfrak{w}$ indicates the weight function (where $\mathfrak{w} : E \to \mathbb{R}^+$ supplies the distances between adjacent vertices of the edges in E). We normalize the edge weights such that the length of the shortest edge in G is 1 (and the weights of other edges are normalized proportionally to that edge). Therefore, even if the network is scaled by some factor δ, the bounds given in this paper hold independent of the scaling factor δ. Note that $\mathfrak{w}$ here is different from previous approaches where $\mathfrak{w}$ represents the detection rates. We assume that each sensor has a unique identifier (ID) and sensors are aware of their geographical locations. We assume that $\mathfrak{w}(u, u) = 0$ for any node $u \in V$. We have m different mobile objects $\mathcal{M}$. Objects are tracked by nearby sensor nodes which are proxy nodes. We assume that G is connected, i.e., there is a path of nodes that connects any pair of nodes in G. Let $\text{dist}_G(u, v)$ be the shortest path length (distance) between nodes u and v with respect to the weight function $\mathfrak{w}$ in G. The *k-neighborhood* of a node v is the set of nodes which are within a distance of at most k from v (including v). The *diameter* is the maximum shortest path distance over all pairs of nodes in G ($D = \max_{u,v \in V} \text{dist}_G(u, v)$). We assume that nodes and edges of G do not crash in most of our results. This assumption of nodes and edges do not crash may be too strong for practical wireless sensor networks as nodes are prone to battery depletion and availability of links vary; we discuss in Section 7 how our algorithm can be adapted to handle the situations of nodes and edges crash to make it suitable for fault-prone and dynamic sensor networks.

2.2 Construction of Overlay Structure

For simplicity, we build here $\mathcal{HS}$ on a constant-doubling network[1]; we describe $\mathcal{HS}$ construction for more general networks in Section 6. Constant-doubling networks have been widely used as an appropriate model of a sensor network metric, e.g. see [7, 11, 12, 27, 40]. We use a distributed maximal independent set algorithm due to Luby [24] to select the nodes of G to include in $\mathcal{HS}$; some recent algorithms [15, 29] can also be used to construct $\mathcal{HS}$. This structure was used before in, e.g. [14, 31], for different problems.

We define a sequence of *connectivity graphs* $\mathcal{I} := \{\mathcal{I}_0 = (V_0, E_0), \mathcal{I}_1 = (V_1, E_1), \ldots \mathcal{I}_h = (V_h, E_h)\}$, where 0 is the lowest (bottom) level and $h \leq \lceil \log D \rceil + 1$ is the highest (top) level. At level 0, we include all nodes of G in $\mathcal{I}_0$, i.e. $V_0 := V$. We define E_ℓ to be the set of all edges in V_ℓ, such that for each pair of nodes (u, v) in V_ℓ, $\text{dist}_G(u, v) < 2^{\ell+1}$. We define V_ℓ (where $1 \leq \ell \leq h$) to be a subset of $V_{\ell-1}$, such that one node from each edge in $E_{\ell-1}$ is excluded from V_ℓ, but all nodes excluded from V_ℓ are connected to nodes in V_ℓ by an edge in $E_{\ell-1}$. Thus, V_ℓ is a maximal independent subset of $V_{\ell-1}$, e.g. $MIS^{\ell-1}$. V_h contains exactly one node, which is the *root* node r, and E_h is the empty set.

[1]Let the space within radius δ of a point be called the ball centered at that point. A point set Γ has doubling dimension ρ if any set of points in Γ that are covered by a ball of radius δ can be covered by 2^ρ balls of radius $\frac{\delta}{2}$. We say that a metric is doubling and has a low dimension if ρ is bounded by a constant and is small.

We define the *default parent* and *parent set* for each node $w \in V_\ell$, which will be useful in forwarding the detection messages towards the root in $\mathcal{HS}$. In particular, default parents are necessary for being able to execute *maintenance* and *query* operations. However, the parent sets are useful in further minimizing the costs of *query* and *maintenance* operations by allowing the detection messages from different nodes to meet at some particular level. We discuss this in detail later in Section 3.1. The default parent of $w \in V_\ell$ is a node in $V_{\ell+1}$ that is closest to w breaking ties arbitrarily, i.e., it is a node at distance at most $2^{\ell+1}$ away from w. The parent set of $w \in V_\ell$ is a subset of nodes in $V_{\ell+1}$ that are within $4 \cdot 2^{\ell+1}$ of w, including the default parent.

$\mathcal{HS} = (V_T, E_T)$ is a layered node structure on $\mathcal{I}$, where V_T are all the nodes in $\mathcal{I}$, and E_T are the edges between parent-child pairs in every two consecutive level connectivity graphs $\mathcal{I}_\ell$ and $\mathcal{I}_{\ell+1}$ (and are not necessarily related to the edges E_ℓ defined above). An $\mathcal{HS}$ edge can be a logical edge over multiple sensor nodes which will be simulated by physical edges in G. Moreover, the nodes at each level $\ell \geq 1$ can be again the logical nodes which are simulated by physical nodes of G. That is, some nodes in G participate as parent nodes for many levels in $\mathcal{HS}$ and some edges participate in connecting many parents in different levels. For a constant-doubling network G, $\mathcal{HS}$ can be constructed with the polynomial communication cost in expectation as the maximal independent set algorithm due to Luby [24] used for the construction of $\mathcal{HS}$ is randomized and outputs a maximal independent set in $\mathcal{O}(\log n)$ rounds in expectation.

The default parent of a level-ℓ node x in $\mathcal{HS}$ is denoted by $home^\ell(x)$. The default parents for a bottom level sensor node x for each level are defined recursively up to the root such that $home^0(x) = x$ and $home^\ell(x)$ is the default parent of $home^{\ell-1}(x)$. These $home^\ell()$ nodes are useful in defining parent sets. We denote by $parentset^\ell(x)$ the parent set of $home^{\ell-1}(x)$ (the nodes at level ℓ within $4 \cdot 2^{\ell+1}$ from $home^{\ell-1}(x)$). According to the $\mathcal{HS}$ construction, we can have the following observation about the number of nodes in a parent set.

Observation 1 *In $\mathcal{HS}$ constructed above, there are at most $2^{3\rho}$ nodes in $parentset^{\ell+1}(x)$ for any node x at level $0 \leq \ell < h$.*

The above observation can be obtained as follows. From the construction of $\mathcal{HS}$, we have that all level $\ell+1$ parents of a level ℓ node can be covered by at most $2^{3\rho}$ radius 2^ℓ-neighborhood balls. Moreover, different level $\ell+1$ parents of a level ℓ node are at least distance $2^{\ell+1}$ from each other since they are maximal independent sets at level ℓ. Therefore any level ℓ node has no more than $2^{3\rho}$ level $\ell+1$ parents.

The nodes in $\mathcal{HS}$ from level $\ell = 1$ to $\ell = h$ are internal nodes. We now specify which internal nodes will by visited by detection messages from any proxy node in the bottom level of $\mathcal{HS}$ on their way up to the root. Any detection message from a node u visits the ascending sequence of its parent sets starting from $parentset^0(u)(= u)$ at level 0 to $parentset^h(u) = r$ at level h (the root level); the nodes in the parent set at each level are visited according to their IDs in increasing order starting from the smallest ID node. Thus, the highest ID node in $parentset^\ell(u)$ is connected to the smallest ID node in $parentset^{\ell+1}(u)$. We now define a notion of *detection path* as follows.

Definition 1 *If we connect the internal nodes visited by a detection message from a proxy node u to reach root level h one after another by shortest paths between them, we obtain a path which we call the* detection path *and denote by* $\mathsf{DPath}(u)$.

For example, in Fig. 1a, $7 \to x_1 \to x_2 \to w_3 \to r$ is the detection path of node 7, i.e., $\mathsf{DPath}(7)$. Any detection message for any object in the detection range of node 7 follow this path while maintaining the detection lists in internal nodes of $\mathcal{HS}$ that are parents of 7. We say that two detection paths from two different nodes *meet* at level i if they visit the same internal node at level i.

Lemma 2.1 *For any two nodes $u, v \in V$, their detection paths $\mathsf{DPath}(u)$ and $\mathsf{DPath}(v)$ meet at level $\lceil \log(\mathrm{dist}_G(u,v)) \rceil + 1$ of $\mathcal{HS}$.*

Proof. Let $\ell = \lceil \log(\mathrm{dist}_G(u,v)) \rceil + 1 \leq h$ and u^ℓ be u's level-ℓ default parent. By definition of default parents,

$$\mathrm{dist}_G(u, u^\ell) < \sum_{i=1}^{i=\ell} 2^i < 2 \cdot 2^\ell.$$

Let $v^{\ell-1}$ be v's level-$(\ell-1)$ default parent, we get

$$\mathrm{dist}_G(v, v^{\ell-1}) < 2^\ell.$$

In the case where the two detection paths $\mathsf{DPath}(u)$ and $\mathsf{DPath}(v)$ do not intersect at level ℓ, then $u^\ell \notin \mathit{parentset}^\ell(v)$. Since $\mathit{parentset}^\ell(v)$ contains nodes at level ℓ within distance $4 \cdot 2^\ell$ of $v^{\ell-1}$, then by definition of parent sets,

$$\mathrm{dist}_G(v^{\ell-1}, u^\ell) \geq 4 \cdot 2^\ell,$$

and by triangle inequality,

$$\begin{aligned} \mathrm{dist}_G(u, v) &\geq \mathrm{dist}_G(v, u^\ell) - \mathrm{dist}_G(u, u^\ell) \\ &\geq \mathrm{dist}_G(v^{\ell-1}, u^\ell) - \mathrm{dist}_G(v, v^{\ell-1}) - \mathrm{dist}_G(u, u^\ell) \\ &> 2^\ell. \end{aligned}$$

Therefore, $\mathsf{DPath}(u)$ and $\mathsf{DPath}(v)$ of nodes $u, v \in V$ must intersect at level $\ell = \lceil \log(\mathrm{dist}_G(u, v)) \rceil + 1$. □

Moreover, we have the following lemma for the length of any detection path.

Lemma 2.2 *The length of the detection path of any bottom level sensor node (proxy or non-proxy) u up to any level j in $\mathcal{HS}$ is* $\mathrm{length}(\mathsf{DPath}_j(u)) \leq 2^{j+3\rho+6}$.

Proof. We have that the distance between a level-ℓ child and its level-$(\ell+1)$ parent is at most $c_b \cdot 2^\ell$ for some constant c_b. This is because the length of an edge (i.e., the shortest path distance) between a level-ℓ child and level-$(\ell+1)$ parent is at most $2 \cdot 2^{\ell+2}$ if the parent node is the default parent of the child node and at most $4 \cdot 2^{\ell+2}$ if the parent node is in the parent set of the child node. Moreover, from Observation 1, we have that there are at most $2^{3\rho}$ nodes in the parent set at level $\ell+1$ for the child node. Now recall that $\mathsf{DPath}(u)$ for any node u visits all the nodes in the parent set $\mathit{parentset}^\ell(u)$ according to their node IDs starting from the smallest ID node and ending in the highest ID node before going to level-$(\ell+1)$. Therefore, the total length of $\mathsf{DPath}(u)$ between the first node in level ℓ and the first node in level $\ell+1$ is at most

$$4 \cdot 2^{\ell+2} + 2^{3\rho} \cdot 2^{\ell+1} \leq 2^{3\rho} \cdot 2^{\ell+5} \leq 2^{\ell+3\rho+5},$$

by adding the length of the paths. Note that the length of $\mathsf{DPath}_j(u)$ is obtained by adding the length of detection path fragments for each level starting from level-0 up to level-j. Therefore, the path length of $\mathsf{DPath}_j(u)$ is

$$\mathrm{length}(\mathsf{DPath}_j(u)) \leq \sum_{i=1}^{j} 2^{i+3\rho+5} \leq 2^{j+3\rho+6}.$$

□

3 MOT Algorithm

Consider a set $\mathcal{M} = \{o_1, o_2, \ldots, o_m\}$ of $|\mathcal{M}| = m$ different mobile objects. The proxy of each object o_i is the sensor node in the bottom level of $\mathcal{HS}$ which currently detects it. The goal of MOT is to maintain the detection lists of objects in $\mathcal{HS}$ (Fig. 1) at all times, even when the objects move. This is done by maintaining object information (for each object o_i) at each internal node of $\mathcal{HS}$ in the detection path of the proxy of that object up to the root r.

We use two kinds of detection messages: *publish*, and *maintenance* for tracking these m mobile objects. There are two types of *maintenance* messages: *insert* and *delete*. We also support a query operation. The *publish* operations are used to form the initial detection lists in $\mathcal{HS}$. Only after this initialization, *maintenance* and *query* operations can be supported. The *maintenance* operation is

used to update the detection list when an object is moved from the current proxy node to some new proxy node. The *query* operation is issued by any sensor node in the bottom level of $\mathcal{HS}$ (proxy or non-proxy) to obtain any object of interest among the m objects.

Any *publish* operation works as follows: A proxy node sends a *publish* message upward following its detection path until it reaches the root. This message stores the object in the detection list of all the internal nodes it visits in its detection path. Note that the *publish* operation is applied for each object only once in the beginning. Fig. 1a depicts the $\mathcal{HS}$ after *publish* operations from the proxy nodes that are at the bottom level of $\mathcal{HS}$. The publish operation from node 1 adds o_1 in the detection lists of internal nodes u_1, u_2, u_3 and r in the detection path of node 1, $\mathsf{DPath}(1)$.

Suppose an object o_i moved from proxy node x to some node y. Now y becomes the proxy node of o_i and begins a *maintenance* operation by sending an *insert* message to update o_i in the detection lists maintained in $\mathcal{HS}$ (see Fig. 1b where node 4 issues an *insert* when o_1 moves from 1 to 4). Let w be the lowest ancestor node of y in $\mathsf{DPath}(y)$ such that o_i is in the detection list of w (u_3 in Fig. 1b). The node w is guaranteed to exist because in the worst-case scenario it may be the root r. The *maintenance* is implemented by first sending *insert* upward from y up to node w and then sending *delete* downward from w to the previous proxy node x of o_i. While going up, *insert* adds o_i to the detection lists of each internal node it visits in $\mathsf{DPath}(y)$, and while going down, *delete* removes o_i from the detection lists of each node that has o_i. In Fig. 1b, when o_1 moves from 1 to 4, the *insert* message travels from 4 towards root r through $\mathsf{DPath}(4)$, and adds o_1 to the detection lists of w_1 and v_2, before it finds o_1 in the detection list of w, after that *delete* messages travel downward visiting nodes u_2, u_1, and 1 which have o_1 in their detection lists and removing o_1 from them. The path from the root r to the object o_i, after the *maintenance* operation is finished, consists of two detection path parts: a part of y's detection path, $\mathsf{DPath}(y)$, from w to y and a part of node x's detection path, $\mathsf{DPath}(x)$, from r to w. This path can be further divided into multiple parts due to subsequent *maintenance* operations.

A *query* operation issued by any node x (not necessarily a proxy) for any object of interest o_i among m available (distinct) objects uses its $\mathsf{DPath}(x)$ to reach to o_i. It works as follows. First *query* goes upward, similar to *maintenance*, following internal nodes in the $\mathsf{DPath}(x)$ and checking whether o_i is in the detection lists of the internal nodes it visits. As soon as *query* reaches the first internal node w which has o_i in its detection list, *query* goes downward following the nodes which have o_i in their detection lists until it reaches the proxy node of o_i. After it reaches the proxy of o_i, it checks to see if o_i is still present. If it is, it sends the result to x. If it is not, *query* waits for the *delete* message to arrive. The *delete* message will contain the new location of o_i, and the query will forward that location to x. In Fig. 1c, the *query* operation issued by 1 for object o_3 follows its $\mathsf{DPath}(1)$ up to u_3 (which is the first internal node in the $\mathsf{DPath}(1)$ that has o_3 in its detection list), after that it goes downward following v_2 and w_1 (since these nodes have o_3 in their detection lists) and reaches the node 5, which is the proxy node of o_3. Note that a *query* operation does not add or delete information from the nodes it visits, hence does not modify the $\mathcal{HS}$.

This above described approach provides the minimum communication cost for *maintenance* operations for each object o_i. However, this approach may not provide any guarantee on the cost of *query* operations, i.e., the object of interest of a requesting node may be in its neighboring node but the *query* operation may need to reach up to the root r to find that object. This is due to the possible enormous fragmentation of detection paths after several *maintenance* operations such that the nearby nodes do not have the information about the object that is in its neighboring node.

For the illustration of this scenario, consider the example execution scenario given in Fig. 2. Let the object (denoted by star) be originally at node 4 and the *maintenance* operations are issued by nodes 1, 2, and 3 in a sequence one after another. Here, $\mathsf{DPath}(4) := \{4, x_1, w_2, w_3, u_4, \ldots, r\}$. When the object moved to node 1, $\mathsf{DPath}(1) := \{1, u_1, u_2, u_3, u_4, \ldots, r\}$ in which $u_4, \ldots, r$ is the $\mathsf{DPath}(4)$ fragment and $1, u_1, \ldots, u_4$ is the $\mathsf{DPath}(1)$ fragment. When the object moved to node 2 from node 1, $\mathsf{DPath}(2) := \{2, w_1, v_2, u_3, u_4, \ldots, r\}$, the combination of $\mathsf{DPath}(4)$ fragment from u_4 to r, $\mathsf{DPath}(1)$ fragment from u_3 to u_4, and $\mathsf{DPath}(2)$ fragment from 2 to u_3. Now, when the object finally moved to 3, then the detection path $\mathsf{DPath}(3)$ is the combination of four detection path fragments. Suppose that original $\mathsf{DPath}(3) := \{3, w_1, w_2, x_3, w_4, \ldots, r\}$ and assume that node 5 issues a *query* operation after the *maintenance* operation from node 3 is finished. Then, according to Lemma 2.1, the *query*

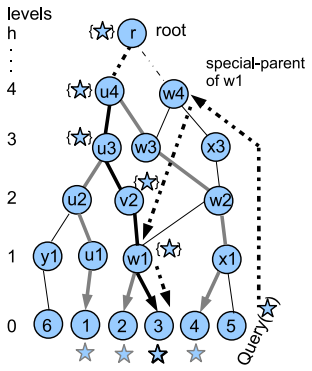


Figure 2: An efficient *query* operation due to a special parent node

should have the information about the object at node w_2 which is the node where $\mathsf{DPath}(5)$ and $\mathsf{DPath}(3)$ intersect. However, w_2 does not have the information about the object because, although the object of interest is at node 3, the information about that object is not recorded in its original $\mathsf{DPath}(3)$ but in the different path $\mathsf{DPath}(3) := \{3, w_1, v_2, u_3, u_4, \ldots, r\}$ due to the fragmentation of detection paths of previous owners of the object. This fragmentation of the detection path for an object can go further when many *maintenance* operations from different nodes are executed for that object.

We reduce the effect of path fragmentation situation and guarantee efficient *query* cost along with low *maintenance* cost using the concept of a *special parent* node, such that whenever an object o_i is added in the detection list of an internal node $t \in \mathsf{DPath}(x)$ of a proxy node x, the special parent node of t is also informed about t holding o_i in its detection list. The special parent node is selected in the $\mathsf{DPath}(x)$ in such a way that any *query* close to t will either reach t or its special parent. As shown in Fig. 2, since the special-parent node have the information about w_1 having the information about the object, the *query* operation from node 5 is served by only reaching w_4 (the special-parent of w_1); otherwise, the *query* operation would have to reach up to the root r to find the information about the object. Details on how special-parents are selected will be provided in the following while giving the formal details of the algorithm.

The formal details of MOT are given in Algorithm 1[2]. Algorithm 1 focuses only on minimizing *maintenance* and *query* cost; we treat load balancing issues later in Section 5. We assume that each node in $\mathcal{HS}$ knows its *parent* and *special-parent*, except the *root* node, whose *parent* and *special-parent* are both $\perp$ (null). Note that special parent nodes for some internal node nearby to root n are undefined and it does not affect our algorithm. For simplicity, assume that there is only one parent internal node $p^\ell(x)$ for a node x at any level ℓ. Therefore, $\mathsf{DPath}(x)$ is simply the concatenation of $p^0(x) = x$, $p^1(x)$, up to $p^h(x) = r$, where $p^k(x)$ is the level k parent internal node of x, $p^{k-1}(x)$ and $p^{k+1}(x)$ are, respectively, its child and parent internal nodes. We have the following definitions for parent and special-parent nodes.

Definition 2 *A parent node of a bottom level node x at level i in $\mathcal{HS}$ is the node $p^i(x)$ that is in $\mathsf{DPath}(x)$ at level i.*

Definition 3 *The special-parent node of a level i parent node $p^i(x)$ of the bottom level node x given in Definition 2 is the level k parent node $p^k(x)$ of x in $\mathsf{DPath}(x)$, where $k = i + 3\rho + 6$ and ρ is a doubling constant. This special-parent node is denoted as $SP(p^i(x))$.*

Definitions 2 and 3 can be extended when considering $parentset^i(x)$ (instead of $p^i(x)$) as follows. The parent node of a node $y \in parentset^i(x)$ is the node $z \in parentset^i(x)$ that has ID greater than

[2]Note that we present *publish*, *maintenance*, and *query* procedures of Algorithm 1 as an iteration over the nodes for the sake of simplicity in understanding the algorithm. This can be immediately converted to a message-passing based distributed algorithm by modifying the procedures from the perspective of what a node in any level of $\mathcal{HS}$ does when it receives a *publish*, *maintenance*, or *query* message from either its child or its parent node.

Algorithm 1: MOT algorithm for each $o_i, 1 \leq i \leq m$

// We describe *publish* and *maintenance* operations assuming only one node $p^k(v_i)$ of $parentset^k(v_i)$ in DPath(v_i) of each node v_i at each level.

1 **Publish object o_i at proxy node v_i:**
2 $k \leftarrow 1$;
3 **Until** node $p^k(v_i)$ is a root node do
4 $p^k(v_i).DL \leftarrow p^k(v_i).DL \cup \{o_i\}$;
5 $k \leftarrow k + 1$;
6 **Maintenance of object o_i by proxy node v_i received from proxy node v_j:**
7 **Insert by v_i:**
8 $k \leftarrow 1$;
9 **Until** $o_i \in p^k(v_i).DL$ **do**
10 $p^k(v_i).DL \leftarrow p^k(v_i).DL \cup \{o_i\}$; $SP(p^k(v_i)).SDL \leftarrow SP(p^k(v_i)).SDL \cup \{o_i\}$;
11 $k \leftarrow k + 1$;
12 **Issue a delete for v_j from level k node $p^k(v_i)$;**
13 **Delete for v_j:**
14 **Until** a proxy node is reached **do**
15 Find a child node x from the current internal node $p^k(v_i)$ such that $o_i \in x.DL$;
16 $x.DL \leftarrow x.DL \backslash \{o_i\}$;
17 $SP(x).SDL \leftarrow SP(x).SDL \backslash \{o_i\}$ for the special parent of x which has o_i in its detection set;
18 $k \leftarrow k - 1$;
19 **Query an object of interest o_i by proxy node v_i:**
20 $k \leftarrow 1$;
21 **Until** $o_i \in t.DL$ or $o_i \in t.SDL$ for any $t \in parentset^k(v_i)$ **do**
22 $k \leftarrow k + 1$;
23 **If** $o_i \in t.DL$ **then** Go to the proxy node following internal nodes downward starting from t which have o_i in their DL and send information to v_i;
24 **Else** Go to the proxy node following internal nodes which have o_i in their DL downward starting from the special-child of t that added o_i in $t.SDL$ and send information to v_i;

the ID of node y but smaller than the IDs of other nodes in $parentset^i(x)$. In case of a special-parent node, the nodes in $parentset^k(x)$ act as $SP(.)$ of the nodes in $parentset^i(x)$. More precisely, for a node $y \in parentset^i(x)$, $SP(y)$ is one of the nodes in $parentset^k(x)$. As we visit the nodes in parent sets according to the order of their IDs, for the smallest ID node in $parentset^i(x)$, we can have its special-parent the smallest ID node in $parentset^k(x)$. If the nodes in $parentset^k(x)$ are not sufficient to provide distinct special-parents for the nodes in $parentset^i(x)$, we can start again from the smallest ID node until all the nodes in $parentset^i(x)$ have their special-parents in $parentset^k(x)$.

We will use the symbols DL and SDL to represent the detection list and special detection list of each internal node in $\mathcal{HS}$. The mobile objects in $\mathcal{M}$ are added in these lists initially by *publish* operations and dynamically updated through *maintenance* operations. The pseudocode for a *publish* operation by a proxy node v_i for an object o_i is given in Lines 1–5 of Algorithm 1. Similarly, the pseudocode for a *maintenance* operation by a node v_i for an object o_i received from node v_j is given in Lines 6–18 of Algorithm 1. While going upward, *maintenance* operation adds o_i to DL and SDL of all the nodes and their special-parent nodes it visits and while going down, removes o_i from the nodes and their special-parent nodes which contained o_i. The pseudocode for a *query* operation is given in Lines 19–24 of Algorithm 1, which is processed similar to a *maintenance* operation but without adding or removing the object of interest at the detection lists of nodes it visits in its detection path.

We will prove in Section 4 that *maintenance* and *query* operations in Algorithm 1 are efficient

but still the nodes in higher levels of $\mathcal{HS}$ bear greater load due to the number of objects and the bookkeeping information they store in their detection lists. For example, the root r needs to store all m objects in its detection list. We extend $\mathcal{HS}$ to have clusters associated with each of its nodes selected through a maximal independent set algorithm using an embedding of a de Bruijn graph [19] to distribute the detection list to other nodes inside the cluster. This technique of extending $\mathcal{HS}$ to have clusters is only needed for the constant-doubling graph model. In general network model, the overlay structure we use has clusters associated with leader nodes already and hence it is not needed. The modifications in Algorithm 1 required to achieve load balancing and other details are given in Section 5 for the constant-doubling graph model; the bound in the general graph model is proven in Section 6.

3.1 Benefits of Having Parent Sets

As discussed in the aforementioned paragraphs, special parents are useful in guaranteeing efficient *query* costs, controlling the severe effect of the fragmentation of detection paths in the *query* cost; without special parents, a *query* operation may need to reach up to the root node to find the information about the object of interest in some situations.

We discuss in this section benefits of having parent sets, $parentset^{\ell}(.)$, in each level ℓ instead of only one default parent, $home^{\ell}(.)$. Firstly, parent sets guarantee that detection paths of distinct nodes that are at distance $\text{dist}_{\text{G}}(.)$ meet at level $\lceil \log \text{dist}_{\text{G}}(.) \rceil + 1$. Secondly, they are useful in finding the information about the object in the lowest level possible. For example, using only the default parents ($home^{\ell}(.)$), when a *maintenance* or a *query* operation for an object o_i reaches level ℓ and there is no information about o_i in $home^{\ell}(.)$, then the operation has to simply go to $home^{\ell+1}(.)$. However, using the $parentset^{\ell}(.)$, a *maintenance* or a *query* operation has the chance of having the information about o_i in the other nodes of $parentset^{\ell}(.)$ even if there is no information about o_i in $home^{\ell}(.)$ and hence the operation does not need to go to level $\ell + 1$ if the object information is found in any of the nodes in $parentset^{\ell}(.)$. This reduces the cost incurred in serving the operations. However, these parent sets in any level need to be visited by an operation in some order to avoid *race conditions* − one operation may miss the detection path set by some other operation while searching in the same level, especially in concurrent situations with two *maintenance* and/or *query* operations from different nodes simultaneously probing the parent sets at some level.

A simple approach will be to visit the nodes in the parent sets in an arbitrary order with only a restriction that default parent will be visited last. That is, when a default parent is visited by an operation at any level, then it guarantees that no other node in the parent set have the information about the object of interest for that operation. Then this operation can jump to the next level and start searching for the information about the object of interest at that level. This approach is sufficient for one by one executions, however it might introduce race conditions as described in Fig. 3 in concurrent executions. Let $parentset^{k}(j)$ and $parentset^{k}(m)$ be the parent sets of nodes j and m at level k, and $p^{k}(j)$ and $p^{k}(m)$ be the default parents. Assume also that $p^{k}(j) \in parentset^{k}(m)$ and $p^{k}(m) \in parentset^{k}(j)$. Let the *query* operation from node j and the *maintenance* operation from node m be probing their parent sets $parentset^{k}(j)$ and $parentset^{k}(m)$ simultaneously at some time step t. In this situation, the *query* operation from j will miss the detection path that will be set by the *maintenance* operation, as the *maintenance* operation updates the information at $p^{k}(m)$ only after all other nodes in $parentset^{k}(m)$ are visited and the *query* operation from j does not see this change as it visits $p^{k}(m)$ before the change happened. This scenario may repeat in the higher levels. One way to resolve this race condition is to serialize the probing of level k parent sets for the operations at level $k-1$. This needs extensive locking of the nodes. Our technique of visiting the nodes of the parent sets in the order of their IDs starting from the smallest ID node and ending at the highest ID node avoids this kind of race conditions at all times. The reason is that all the operations, even from different nodes, visit the nodes in their parent sets in the same order and hence the update at $p^{k}(m)$ by the *maintenance* operation from m is not missed by the *query* operation from j.

Recall that we did not use $parentset^{k}(x)$ in Algorithm 1. If we use $parentset^{k}(v_i)$, then we need to add o_i to the DL of all the nodes of $parentset^{k}(v_i)$ and also to the SDL of the special-parent

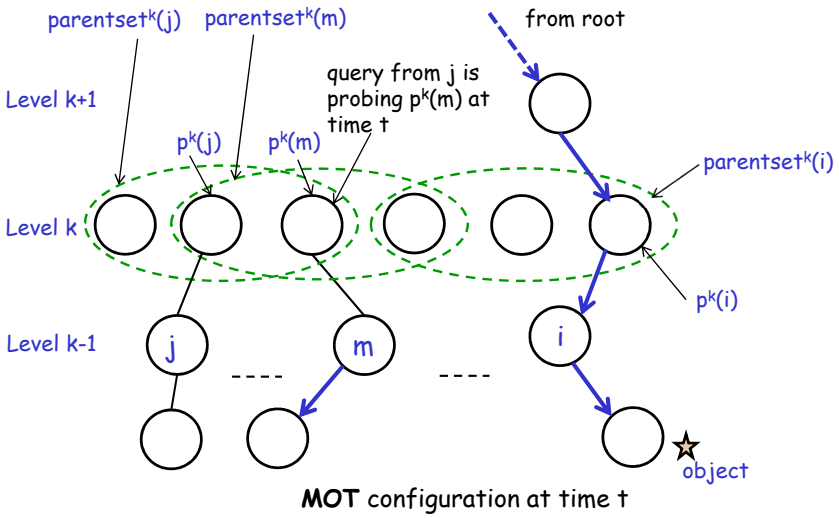


Figure 3: An illustration of a possible race condition in concurrent execution of object operations

nodes of the nodes in $parentset^k(v_i)$. As there are constant number of nodes in the parent sets at any level, this will increase the maintenance and query costs by a constant factor only.

4 Analysis of the MOT Algorithm

We analyze the performance of the MOT algorithm (Algorithm 1) for *maintenance* and *query* operations in both one by one case and concurrent case. We provide the details of the analysis of the one by one case and give an overview of how the same cost ratio bounds can be achieved in the concurrent case. Recall that in the one by one case each *maintenance* operation of an object arrives only after the previous *maintenance* operation for that object is finished whereas in the concurrent case where a *maintenance* operation of an object may arrive before previous *maintenance* operation for that object is finished.

In the analysis, we do not take into account the cost for probing special-parents when the *maintenance* operations are processed at a given level in their cost ratio computation. If we take this cost into account, the cost ratios of *maintenance* operations increase by a constant factor in constant-doubling networks (as the special parents are $\mathcal{O}(1)$ levels higher than the current level where the operation is being processed); this cost increase will be a factor of $\mathcal{O}(\log n)$ in general networks (as special-parents here can be $\mathcal{O}(\log n)$ levels higher than the current level where the operation is being processed).

We proceed by analyzing the cost of *publish* operations. MOT is initialized using *publish* operations from proxy nodes to insert the objects in $\mathcal{HS}$ such that they will be useful for future *maintenance* and *query* operations. The cost of a *publish* operation is the sum of the message costs for inserting the objects o_i in $\mathcal{HS}$. Therefore, the total communication cost for a *publish* operation is obtained immediately from Lemma 2.2, by noticing that the number of levels $h \leq \lceil \log D \rceil + 1$ in $\mathcal{HS}$. This is the one time cost.

Theorem 4.1 *The publish cost of Algorithm 1 is $\mathcal{O}(D)$ for each object in constant-doubling networks.*

4.1 Maintenance Cost

For the sake of analysis, an execution of a set $\mathcal{E}$ of κ *maintenance* operations is represented as $\mathcal{E} = \{r_1 = (u_1, v_1, t_1), r_2 = (u_2, v_2, t_2), \ldots, r_\kappa = (u_\kappa, v_\kappa, t_\kappa)\}$, where $r_i = (u_i, v_i, t_i), 1 \leq i \leq \kappa$, are the subsequent *maintenance* operations for mobile objects $o_j, 1 \leq j \leq m$, with old proxy nodes u_i,

new proxy nodes v_i, and $t_i \leq t_j, j \geq i$, is the arrival time of *maintenance* operation r_i. When t_{i+1} is much greater than t_i such that r_{i+1} is initiated only after r_i is finished execution then it represents the one by one case and captures the widely-considered scenarios where event inter-arrival times are much greater than the message propagation time [12, 18]. This one by one case requires *maintenance* operations for a given object o_i to complete sequentially, but otherwise puts no constraint on when or which object will move. When arrival time is same for all operations in $\mathcal{E}$ then this represents the completely concurrent case otherwise it represents the continuous arrival case of *maintenance* operations in arbitrary moments of time.

For the analysis, in both the one by one case and concurrent case, we divide the execution set $\mathcal{E}$ into m different sets $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m\}$ one for each mobile object such that $\mathcal{E}_j$ contains the *maintenance* operations from $\mathcal{E}$ that are for object $o_j, 1 \leq j \leq m$, only. We then compute, for each object o_j, the total cost of MOT (Algorithm 1) and the corresponding optimal cost, and later combine these costs to the corresponding total and optimal costs of the $m-1$ other sets to obtain the overall *maintenance* cost ratio. The separate analysis for each object is possible because changes in $\mathcal{HS}$ due to operations of one object do not interfere with the changes made by any other object.

Formally, let $C^*(\mathcal{E}_j)$ denote the total communication cost of performing all the operations in $\mathcal{E}_j$ using the optimal algorithm in G. Let $C(\mathcal{E}_j)$ denote the total communication cost of performing all the operations in $\mathcal{E}_j$ using Algorithm 1. The total communication cost is measured through the total distance traversed by all the messages. Then then cost ratio of Algorithm 1 to perform all the *maintenance* operations in $\mathcal{E}_j$ will be $\frac{C(\mathcal{E}_j)}{C^*(\mathcal{E}_j)}$. Combining this cost with the cost ratios for other objects, the overall cost ratio of Algorithm 1 will be

$$\frac{C(\mathcal{E})}{C^*(\mathcal{E})} = \frac{\sum_{j=1}^{m} C(\mathcal{E}_j)}{\sum_{j=1}^{m} C^*(\mathcal{E}_j)},$$

where $C(\mathcal{E})$ is the total cost and $C^*(\mathcal{E})$ is the optimal cost due to all $E_j, 1 \leq j \leq m$. The optimal costs $C^*(\mathcal{E}_j)$ for $1 \leq j \leq m$ need to be summed as these optimal bounds are for different objects. We will also show that the cost ratio of other object $o_k, k \neq j$, is the same as the cost ratio for object o_j. Since $\mathcal{E}$ that we considered is arbitrary, the bound $\frac{C(\mathcal{E})}{C^*(\mathcal{E})}$ translates to the cost ratio of Algorithm 1 for any set $\mathcal{E}$ of *maintenance* operations, i.e., the bound for $\mathcal{E}$ translates to the bound $\max_{\mathcal{E}} \frac{C(\mathcal{E})}{C^*(\mathcal{E})}$ for any arbitrary set $\mathcal{E}$ of *maintenance* operations. We prove the *maintenance* cost ratio for one by one case in Section 4.1.1 and for concurrent case in Section 4.1.2.

4.1.1 One by One Case

In the one by one case, time component in the representation of *maintenance* operation r_i does not play any role and hence it can be dropped from consideration and simply write $r_i = (u_i, v_i)$. We analyze the total and optimal costs for the *maintenance* operations in $\mathcal{E}_j$ for the object o_j. We denote by $s_{k,j}, 1 \leq k \leq h$ the number of *maintenance* operations in $\mathcal{E}_j$ that reached or traversed level k while updating $\mathcal{HS}$ after o_j moves. Thus, $s_{k,j} \leq s_{k',j}$ for any $k > k'$. The *peak level* for an operation $r_i \in \mathcal{E}_j$ is the maximum level reached by r_i. The peak level reached by a *publish* operation is h (the maximum level in $\mathcal{HS}$). The peak level for a *maintenance* is the level it reaches before going downward. We prove the following lemmas for the total and optimal communication costs for the operations in $\mathcal{E}_j$.

Lemma 4.2 *$C(\mathcal{E}_j) \leq \sum_{k=1}^{h} s_{k,j} \cdot 2^{k+3\rho+7}$, where $h = \lceil \log D \rceil + 1$ and ρ is a doubling constant.*

Proof. Let $C_k(\mathcal{E}_j)$ be the total communication cost of Algorithm 1 for the *maintenance* operations in $\mathcal{E}_j$ that reach level k in $\mathcal{HS}$. From Lemma 2.2, $C_k(\mathcal{E}_j) \leq s_{k,j} \cdot 2^{k+3\rho+7}$. By combining the costs for each level, the total communication cost of Algorithm 1 is

$$C(\mathcal{E}_j) = \sum_{k=1}^{h} C_k(\mathcal{E}_j) \leq \sum_{k=1}^{h} s_{k,j} \cdot 2^{k+3\rho+7}.$$

□

Lemma 4.3 *$C^*(\mathcal{E}_j) \geq \max_{1\leq k\leq h} s_{k,j} \cdot 2^{k-1}$, where $h = \lceil \log D \rceil + 1$ and ρ is a doubling constant.*

Proof. Let $C^*_k(\mathcal{E}_j)$ be the optimal communication cost for the *maintenance* operations in $\mathcal{HS}$. Let the old proxy node be u, the new proxy node be v, and let the peak node be found at level k. The distance $\text{dist}_G(u, v) \geq 2^{k-1}$, because otherwise the detection paths of u and v would intersect at level $k-1$ or lower (Lemma 2.1).

Therefore, all the operations $s_{k,j}$ have destination nodes at least at a distance $\text{dist}_G(u, v) \geq 2^{k-1}$. Hence,

$$C^*_k(\mathcal{E}_j) \geq s_{k,j} \cdot 2^{k-1},$$

the total shortest path distance between the destination nodes of $s_{k,j}$ operations that reached level k in sensor graph G. Considering all the levels $1 \leq k \leq h$,

$$C^*(\mathcal{E}_j) \geq \max_{1\leq k\leq h} C^*_k(\mathcal{E}_j) \geq \max_{1\leq k\leq h} s_{k,j} \cdot 2^{k-1}.$$

□

Theorem 4.4 *$C(\mathcal{E}_j) \leq h \cdot 2^{3\rho+8} C^*(\mathcal{E}_j)$, where $h = \lceil \log D \rceil + 1$ and ρ is a doubling constant.*

Proof. Since

$$\begin{aligned} C(\mathcal{E}_j) &\leq \sum_{k=1}^{h} s_{k,j} \cdot 2^{k+3\rho+7} \\ &\leq h \cdot \max_{1\leq k\leq h} s_{k,j} \cdot 2^{k+3\rho+7}, \end{aligned}$$

then

$$C(\mathcal{E}_j) \leq h \cdot 2^{3\rho+8} C^*(\mathcal{E}_j)$$

using Lemma 4.3. □

The total communication cost $C(\mathcal{E}_j)$ of Algorithm 1 for the operations in $\mathcal{E}_j$ given in Theorem 4.4 is $\mathcal{O}(h) = \mathcal{O}(\log D)$ factor away from the optimal communication cost $C^*(\mathcal{E}_j)$. $C(\mathcal{E}_j)$ is already small compared to $C^*(\mathcal{E}_j)$ if $D < n^{\mathcal{O}(1)}$. We now argue that $C(\mathcal{E}_j)$ is at most $\mathcal{O}(\log n)$ factor away compared to $C^*(\mathcal{E}_j)$ even when $D > n^{\mathcal{O}(1)}$, where n is the number of nodes in the network. This analysis is more involved in the sense that we need to separate the operations in $\mathcal{E}_j$ into ranges according to the distance between the source and destination proxies. Then we will argue that when D is asymptotically greater than n then the *maintenance* operations in $\mathcal{E}_j$ that reach to the higher levels in $\mathcal{HS}$ need to use high weight edges of G in both the optimal algorithm and MOT, and hence the optimal cost for the *maintenance* operations that reach to higher levels can not be combined to the optimal cost for the *maintenance* operations that visit only lower levels.

We start with describing a setup that is useful in the formal proof of $C(\mathcal{E}_j) \leq \mathcal{O}(\log n) \cdot C^*(\mathcal{E}_j)$. We partition the number of levels $h \leq \lceil \log D \rceil + 1$ of $\mathcal{HS}$ into two types of groups, $\mathcal{G}_1$ and $\mathcal{G}_2$, as described below.

- $\mathcal{G}_1$ groups: There are $\vartheta = \lceil \frac{h}{\log n} \rceil + 1$ groups in $\mathcal{G}_1$ such that $\mathcal{G}_1 = \{G^1_1, G^2_1, \ldots, G^\vartheta_1\}$, where group G^i_1 contains levels in the range $[h-(i-1)\cdot\log n, \max\{h-i\cdot\log n-1, 0\}]$ for $1 \leq i \leq \vartheta$. In other words, each group $G^i_1, 1 \leq i \leq \vartheta$, except G^ϑ_1 contains exactly $\log n$ number of consecutive levels starting from the root level h at any time. The last group G^ϑ_1 contains all the levels from level $h-(\vartheta-1)\cdot\log n$ to level 0 such that the number of levels in G^ϑ_1 are at most $\log n$. According to this division, we have that $\sum_{i=1}^{\vartheta} |G^i_1| = h$.

- $\mathcal{G}_2$ groups: There are at most $\vartheta + 1$ groups in $\mathcal{G}_2$ such that $\mathcal{G}_2 = \{G^1_2, G^2_2, \ldots, G^{\vartheta+1}_2\}$, where group G^i_2 contains levels in the range $[h - \frac{(i-1)}{2} \cdot \log n, \max\{h - (\frac{(2*i-1)}{2} \cdot \log n) - 1, 0\}]$ for

$1 \leq i \leq \vartheta + 1$. In other words, there are exactly $\frac{\log n}{2}$ levels in group G_2^1, exactly $\log n$ levels in each group from G_2^2 to G_2^{ϑ}, and at most $\frac{\log n}{2}$ levels in group $G_2^{\vartheta+1}$. We can see $\mathcal{G}_2$ groups as the shifted version of $\mathcal{G}_1$ groups by $\frac{\log n}{2}$ levels starting from the highest level h. We also have that $\sum_{i=1}^{\vartheta+1} |G_2^i| = h$.

The *maintenance* operations in $\mathcal{E}_j$ are assigned to $\mathcal{G}_1$ and $\mathcal{G}_2$ as follows. The assignment is done according to the peak level reached by each *maintenance* operation in $\mathcal{HS}$. Recall that the peak level of a *maintenance* operation is the level in $\mathcal{HS}$ where *insert* of that *maintenance* meets its corresponding *delete* operation. Formally, we assign the *maintenance* operations in $\mathcal{E}_j$ that have peak level in the range $[h-(i-1)\cdot\log n, \max\{h-i\cdot\log n-1, 0\}]$ to group G_1^i for $1 \leq i \leq \vartheta$. Similarly, the *maintenance* operations with peak level in the range $[h-\frac{(i-1)}{2}\cdot\log n, \max\{h-(\frac{(2*i-1)}{2}\cdot\log n)-1, 0\}]$ are assigned to group G_2^i for $1 \leq i \leq \vartheta+1$. We denote by $p_{k,j}$ the number of operations of $\mathcal{E}_j$ whose peak level is k for $1 \leq k \leq h$. Here $p_{k,j}$ is in fact $s_{k,j}$ after removing the count of the number of operations from $s_{k,j}$ for which the peak level is not k. The total and optimal cost due to the operations in $p_{k,j}$ is considered only in the group where k falls ignoring its impact on the cost in the lower levels, therefore this assignment has the impact of a constant factor increase (at most 2 times) in the total communication cost due to properties that $\mathcal{HS}$ satisfies.

For the sake of analysis, we proceed by dividing $\mathcal{G}_1$ and $\mathcal{G}_2$ into two subgroup sets, which we denote by $\mathcal{G}_{1,A}$, $\mathcal{G}_{1,B}$, and $\mathcal{G}_{2,A}$, $\mathcal{G}_{2,B}$, respectively, such that $\mathcal{G}_{1,A} = \{G_1^1, G_1^3, \ldots\}$, $\mathcal{G}_{1,B} = \{G_1^2, G_1^4, \ldots\}$, $\mathcal{G}_{2,A} = \{G_2^1, G_2^3, \ldots\}$, and $\mathcal{G}_{2,B} = \{G_2^2, G_2^4, \ldots\}$. That is, for each $\mathcal{G}_1$ and $\mathcal{G}_2$, the first subgroup A contains odd numbered groups and the second subgroup B contains even numbered groups. This subgroup set division is helpful later in bounding the optimal communication cost to serve all the operations in $\mathcal{E}_j$. For convenience, we analyze the costs for these subgroup sets separately. Hence, the *maintenance* cost of the MOT algorithm for the operations in $\mathcal{E}_j$ is at most $C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) + C_{\mathcal{G}_{1,B}}(\mathcal{E}_j) + C_{\mathcal{G}_{2,A}}(\mathcal{E}_j) + C_{\mathcal{G}_{2,B}}(\mathcal{E}_j)$, where $C_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$ is the total communication cost of the MOT algorithm for serving all the operations of $\mathcal{E}_j$ inside $G_{1,A}$; other costs are defined similarly. Moreover, $C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$ is the optimal cost for serving all the operations of $\mathcal{E}_j$ inside $G_{1,A}$; optimal costs for other groups are defined similarly. In the lemma below, we analyze the total communication cost due to only one subgroup set $\mathcal{G}_{1,A}$.

Lemma 4.5 $C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) \leq 2 \cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \sum_{k=h-(l-1)\cdot\log n}^{\max\{h-l\cdot\log n-1,0\}} p_{k,j} \cdot 2^{k+3\rho+7}$, *where* $h = \lceil \log D \rceil + 1$ *and* ρ *is a doubling constant.*

Proof. We first focus on a single group $G_1^l \in \mathcal{G}_{1,A}$. When we use l in the following we mean any odd l from $1 \leq l \leq \vartheta$. For the operations in $\mathcal{E}_j$ inside $G_1^l \in \mathcal{G}_{1,A}$, analyzing the total communication cost $C_{G_1^l}(\mathcal{E}_j)$ similar to Lemma 4.2, we have that

$$C_{G_1^l}(\mathcal{E}_j) \leq 2 \cdot \sum_{k=h-(l-1)\cdot\log n}^{\max\{h-l\cdot\log n-1,0\}} p_{k,j} \cdot 2^{k+3\rho+7}.$$

Combining the total communication cost of each group G_1^l, we have that

$$\begin{aligned} C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) &\leq \sum_{l=1,\ \text{odd}\ l}^{\vartheta} C_{G_1^l}(\mathcal{E}_j) \\ &\leq 2 \cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \sum_{k=h-(l-1)\cdot\log n}^{\max\{h-l\cdot\log n-1,0\}} p_{k,j} \cdot 2^{k+3\rho+7}. \end{aligned}$$

□

Lemma 4.6 $C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j) \geq \frac{1}{2} \cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \max_{h-(l-1)\cdot\log n \geq k \geq \max\{h-l\cdot\log n-1,0\}} (p_{k,j}-1) \cdot 2^{k-1}$, *where* $h = \lceil \log D \rceil + 1$ *and* ρ *is a doubling constant.*

Proof. Analyzing along the lines of Lemma 4.3 for the optimal communication cost $C^*_{G_1^l}(\mathcal{E}_j)$ for the operations inside $G_1^l \in \mathcal{G}_{1,A}$, we have that

$$C^*_{G_1^l}(\mathcal{E}_j) \geq \frac{1}{2} \cdot \max_{h-(l-1)\cdot\log n \geq k \geq \max\{h-l\cdot\log n-1,0\}} (p_{k,j}-1)\cdot 2^{k-1}.$$

We now show that $C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$ is at least the sum of the optimal cost $C^*_{G_1^l}(\mathcal{E}_j)$ of all the groups G_1^l in the subgroup set $\mathcal{G}_{1,A}$, i.e.,

$$\begin{aligned} C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j) &\geq \sum_{l=1,\ \text{odd}\ l}^{\vartheta} C^*_{G_1^l}(\mathcal{E}_j) \\ &\geq \frac{1}{2} \cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \max_{h-(l-1)\cdot\log n \geq k \geq \max\{h-l\cdot\log n-1,0\}} (p_{k,j}-1)\cdot 2^{k-1}. \end{aligned}$$

The argument behind this is as follows. Consider any two consecutive groups G_1^l, $G_1^{l+2} \in \mathcal{G}_{1,A}$. For any two operations $r_1 \in G_1^l$ and $r_2 \in G_1^{l+2}$, according to the division of groups, their peak level difference is at least $\log n$ levels. We now show that the edges in the graph G that are used by the operations in G_1^l are different than the edges in G that are used by the operations in G_1^{l+2}.

As there are n nodes in the graph and the diameter D is asymptotically greater than n, the average length of the edges used by the *maintenance* operation $r_1 \in G_1^l$ to connect its source proxy node with its destination proxy node is at least $2^{h-l\cdot\log n-1}$. Similarly, the average length of the edges is at most $2^{h-(l+1)\cdot\log n}$ for the operation $r_2 \in G_1^{l+2}$. In other words, if the average length is d for the edges used by r_1, then the average length is at most d/n for the edges used by r_2 in G. Recall that the total communication cost and the optimal communication cost for the operations that are inside each group G_1^l is considered only in the group where they fall. This will increase the total communication cost $C_{G_1^l}(\mathcal{E}_j)$ for each group G_1^l by the factor of 2 only; moreover, the optimal communication cost $C^*_{G_1^l}(\mathcal{E}_j)$ for each group G_1^l will decrease by at most the factor of 2. Therefore, if any *maintenance* operation $r_1 \in G_1^l$ would have used the edges used by $r_2 \in G_1^{l+2}$ for any odd l in $1 \leq l \leq \vartheta$, then it would not have reached to the level inside G_1^l. Therefore, the set of edges used by any of the operations inside two different groups in $\mathcal{G}_{1,A}$ is different and hence the optimal cost for different groups in $\mathcal{G}_{1,A}$ needs to be added. □

Theorem 4.7 *$C(\mathcal{E}_j) \leq 2^{3\rho+13}\cdot\log n\cdot C^*(\mathcal{E}_j)$, where ρ is a doubling constant.*

Proof. First we compare cost $C_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$ with the optimal cost $C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$. Similar to Theorem 4.4, since

$$\begin{aligned} C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) &\leq 2\cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \sum_{k=h-(l-1)\cdot\log n}^{\max\{h-l\cdot\log n-1,0\}} p_{k,j}\cdot 2^{k+3\rho+7} \\ &\leq 2^{3\rho+11}\cdot\log n\cdot \sum_{l=1,\ \text{odd}\ l}^{\vartheta} \max_{h-(l-1)\cdot\log n \geq k \geq \max\{h-l\cdot\log n-1,0\}} p_{k,j}\cdot 2^k, \end{aligned}$$

then

$$C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) \leq 2^{3\rho+11}\cdot C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j).$$

Now the total and optimal communication costs for other subgroup sets can be analyzed similarly. Combining the cost bounds of all four subgroup sets, we obtain that $C^*(\mathcal{E}_j)$ is at least

$$C^*(\mathcal{E}_j) \geq \max\left\{C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j), C^*_{\mathcal{G}_{1,B}}(\mathcal{E}_j), C^*_{\mathcal{G}_{2,A}}(\mathcal{E}_j), C^*_{\mathcal{G}_{2,B}}(\mathcal{E}_j)\right\},$$

and $C(\mathcal{E}_j)$ is at most

$$C(\mathcal{E}_j) \leq 4\cdot\max\left\{C_{\mathcal{G}_{1,A}}(\mathcal{E}_j), C_{\mathcal{G}_{1,B}}(\mathcal{E}_j), C_{\mathcal{G}_{2,A}}(\mathcal{E}_j), C_{\mathcal{G}_{2,B}}(\mathcal{E}_j)\right\}.$$

We obtain the same bounds as

$$C_{\mathcal{G}_{1,A}}(\mathcal{E}_j) \leq 2^{3\rho+11} \cdot C^*_{\mathcal{G}_{1,A}}(\mathcal{E}_j)$$

for all the remaining subgroup sets $\mathcal{G}_{1,B}, \mathcal{G}_{2,A}$, and $\mathcal{G}_{2,B}$. Therefore,

$$C(\mathcal{E}_j) \leq 2^{3\rho+13} \cdot \log n \cdot C^*(\mathcal{E}_j).$$

□

We are now ready to provide the overall cost ratio $\max_{\mathcal{E}} \frac{C(\mathcal{E})}{C^*(\mathcal{E})}$ of Algorithm 1.

Theorem 4.8 *The maintenance cost ratio of Algorithm 1 is* $\mathcal{O}(\min\{\log n, \log D\})$ *in constant-doubling networks.*

Proof. We have that $\mathcal{E}$ is arbitrary. Moreover, each subset $\mathcal{E}_j$ is arbitrary. Therefore, using the values of $C(\mathcal{E}_j)$ (Theorems 4.4 and 4.7) we can prove that

$$\begin{aligned} C(\mathcal{E}) &= \sum_{j=1}^{m} C(\mathcal{E}_j) \\ &\leq c \cdot \min\{\log n, \log D\} \cdot \sum_{j=1}^{m} C^*(\mathcal{E}_j) \\ &= c \cdot \min\{\log n, \log D\} \cdot C^*(\mathcal{E}), \end{aligned}$$

for some positive constant c. Moreover, lower bound costs $C^*(\mathcal{E}_j)$ for different objects need to be summed up for the total lower bound cost for the operations in $\mathcal{E}$ because lower bound costs for different objects can not be combined. Hence, the theorem follows. □

4.1.2 Concurrent Case

In the concurrent case, whether the current *maintenance* operation overlaps with any previous *maintenance* operation is determined through the arrival time of the *maintenance* operation. We can drop the destination proxy from the definition of the *maintenance* operations in $\mathcal{E}$ in the concurrent case as destination proxies depend on the ordering of requests under concurrent execution. Therefore, time of arrival plays the vital role in the analysis of cost ratios of concurrent *maintenance* operations. We already saw in Section 4.1.1 that the analysis of Algorithm 1 is intriguing. The concurrent case further complicates the analysis. However, assuming a synchronous execution and using the analysis technique recently presented in Sharma and Busch [30], we can prove that the same cost ratio bound given in Theorem 4.8 holds for *maintenance* operations in concurrent executions.

We provide here the overview of the analysis approach presented in [30]. We start with some definitions. Lets assume that a *time unit* is of duration a message requires to reach a destination node that is unit distance far from the source node which sent that message. Now we can define a *period* of time for each level i which is of duration $\Phi(i) = 2^{i+3\rho+6}$ (this period is proportional to $\mathsf{DPath}_i(.)$ length up to level i). In other words, the period at each level i denotes the time a *maintenance* operation needs to reach and modify the information of all the parent nodes at level i following the detection path of the node which issued that *maintenance* operation. This division of time into periods is helpful in obtaining the upper bounds on the costs of *maintenance* operations. We assume that the execution of *maintenance* operations starts from 0 and increases continuously unit by unit. In every level i, a period of duration $\Phi(i)$ starts immediately after the current period of $\Phi(i)$ expires. Moreover, since the neighborhood in the $\mathcal{HS}$ construction increases/decreses by the factor of 2 between two consecutive levels, each period at level $i-1$ is exactly of half duration compared to that of level i and each period at level i is exactly of double duration compared to that of level $i-1$. We call by a *round* a duration of $\Phi(h)$, i.e., the period for the root level $h = \lceil \log D \rceil + 1$. Therefore, in a round, there will be only one period of duration $\Phi(h)$ at level h, 2 periods of duration

$\Phi(h-1)$ each at level $h-1$, 4 periods of duration $\Phi(h-2)$ each at level $h-2$, and so on. If we take any level $0 \leq k \leq h$, we have 2^{h-k} periods of duration $\Phi(k)$ each. Lets denote the consecutive periods at some level k by $\Phi_0(k), \Phi_1(k), \ldots$, such that $\Phi_0(k)$ starts at time 0, $\Phi_1(k)$ starts as soon as $\Phi_0(k)$ expires, and so on.

Now when an operation is processed at level k during some period $\Phi_j(k)$, it will be sent to level $k+1$ at the end of the period $\Phi_j(k)$. Similarly, the operation is forwarded to level $k-1$ again at the end of the period $\Phi_j(k)$. That is, when an operation is processed and ready to be forwarded before the current period expires, the operation waits until the period expires. After that the operation is forwarded to its parent/child nodes. This controls the way operations are forwarded between two consecutive levels of $\mathcal{HS}$ but it does not affect the lower bound analysis (that is, lower bound cost computation does not depend on this approach) and increases the upper bound cost by only a constant factor.

If all the *maintenance* operations are issued simultaneously (one *maintenance* operation per node) at time zero, we can have as a lower bound for the *maintenance* operations that reach any level k the cost that is given by the *Steiner tree* of the nodes in the network that issued those *maintenance* operations; this cost is actually the cost that is similar to the one given in Lemma 4.3 for one by one executions. For the upper bound on the cost for MOT, we have the cost that is similar to the cost given in Lemma 4.2 for one by one executions. Summing the upper bound costs for all the levels and dividing it by the lower bound cost similarly as we did in Theorem 4.8, we obtain the $\mathcal{O}(\log D)$ bound in cost ratio for *maintenance* operations when they are issued simultaneously at time 0 and no further operations are issued. This can again be made $\mathcal{O}(\log n)$ using the analysis technique similar to the one given in Lemmas 4.5 and 4.6, and Theorem 4.7 for the $\mathcal{O}(\log n)$ analysis for one by one *maintenance* operations.

If all the operations are not issued simultaneously but are issued over time, we can divide the periods of each level into *dense* and *sparse* based on how many *maintenance* operations reach to a particular level within the duration of a particular period, as it is done in [30]. In this case, the lower and upper bound analysis depends on both time and distance and provides similar bounds as in the case of one by one executions. Sharma and Busch [30] called a period of a level dense when there are three or more *maintenance* operations that reach to that level during that period; otherwise the period is called sparse. Moreover, they showed that when a *maintenance* operation r_1 reaches a level at period $\Phi_p(.)$ and another *maintenance* operation r_2 reaches that level at period $\Phi_q(.)$ such that $q-p \geq 3$, then r_1 is always ordered before r_2 in the sense that r_1 finishes execution before r_2 starts execution. That is, the analysis for this case captures both the time and distance restrictions in ordering the *maintenance* operations that are issued over time. Therefore, we can prove the following corollary for the *maintenance* operations in concurrent situations using the analysis technique and results of [30]; more details can be found in [30].

Corollary 4.9 *The maintenance cost ratio of Algorithm 1 is* $\mathcal{O}(\min\{\log n, \log D\})$ *in constant-doubling networks even in concurrent executions.*

4.2 Query Cost

We bound here the *query* cost ratio $C(\wp(o_j))/C^*(\wp(o_j))$, where $C(\wp(o_j))$ is the total communication cost of serving a *query* operation $\wp(o_j)$ for some object $o_j, 1 \leq j \leq m$, through MOT (Algorithm 1) and $C^*(\wp(o_j))$ is the optimal cost of serving $\wp(o_j)$. If a *query* operation is issued before any *maintenance* operation arrives in the system (that is, after all *publish* operations are finished), due to Lemma 2.1, it is trivial to see that a *query* operation $\wp(o_j)$ from a sensor node x finds o_j in the detection list of an internal node in the detection path $\mathsf{DPath}(x)$ of x at level $\lceil \log(\mathrm{dist}_G(x,v)) \rceil + 1$, where $\mathrm{dist}_G(x,v)$ is the distance of the proxy node v of o_j from x in G.

If a *query* operation is issued after some *maintenance* operations are executed in the system, there are two possible scenarios. In the first scenario, the *maintenance* operation inter-arrival time is such that the *query* does not overlap with any *maintenance* operation. In the second scenario, the *query* operation overlaps with other (possibly many) *maintenance* operations. We first analyze the cost ratio of a *query* operation in the first scenario in Section 4.2.1 and deal with the cost ratio in the second scenario in Section 4.2.2.

4.2.1 When Queries do not Overlap with Maintenance Operations

In this case, following the internal nodes which have o_j in their detection lists downward from level $\lceil \log(\text{dist}_G(x,v)) \rceil + 1$, any *query* operation $\wp(o_j)$ can reach to the proxy v of o_j at any time, according to our assumption of event inter-arrival times. Therefore, the question is how many levels the *query* $\wp(o_j)$ from x needs to visit to find o_j in the detection list of any internal node in $\mathsf{DPath}(x)$ when *query* is issued after at least a *maintenance* operation is executed in $\mathcal{HS}$. We prove the following lemma.

Lemma 4.10 *A query operation $\wp(o_j)$ from any sensor node x for the object o_j at proxy node v at distance $\text{dist}_G(x,v) \leq 2^i$ is guaranteed to find o_j in the detection list of an internal node in $\mathsf{DPath}(x)$ at level $k \leq i+3\rho+6$ in the scenario where queries do not overlap with any maintenance operations.*

Proof. Assume that $w = p^k(x)$ be a internal node of x at level $k = i+3\rho+6$ in $\mathsf{DPath}(x)$, which is the special parent of a level i node v_i which has object o_j in its detection list. All the nodes between v_i to v also have o_j in their detection lists. For a *query* operation $\wp(o_j)$ to find the special parent of v_i at w, x must be within the 2^{k+1}-neighborhood of w. It suffices to show that the neighborhood 2^{k+1} of w is at least the distance $\text{dist}_G(v_i,x)$ between v_i and x to guarantee that w is a internal node in $\mathsf{DPath}(x)$. As the length of a detection path from a proxy level node to any level i node is bounded by $2^{i+3\rho+6}$ (Lemma 2.1) and $\text{dist}_G(x,v) = 2^i$, $\text{dist}_G(v_i,x) \leq 2^{i+3\rho+6} + \text{dist}_G(x,v) \leq 2^{i+3\rho+6} + 2^i \leq 2^{i+3\rho+7} \leq 2^{k+1}$ for $k = i+3\rho+6$. That is, some node should be within at most $2^{i+3\rho+6}$ to have such information, and thus, that node will be at level $k = i+3\rho+6$. The lemma follows. □

Theorem 4.11 *The cost ratio of Algorithm 1 for any query operation is $O(1)$ in constant-doubling networks.*

Proof. An object o_j can be found at level $k \leq i+3\rho+6$ through either the detection list DL or the special detection list SDL. Therefore, $C(\wp(o_j))$ is at most the sum of the distances $\text{length}(\mathsf{DPath}_k(x))$ (the length of $\mathsf{DPath}(x)$ up to level k), $\text{dist}_G(w,v_i)$ (the path length between the level k internal node w in $\mathsf{DPath}(x)$ and the level i internal node v_i which is the special-child of w from special parent and child relation), and the length of the path the query operation $\wp(o_j)$ follows downward from v_i to v. We have that $\text{length}(\mathsf{DPath}_k(x)) \leq 2^{k+3\rho+6}$, $\text{dist}_G(x,v_i) \leq 2^k$, and $\text{length}(\mathsf{DPath}_i(v)) \leq 2^{i+3\rho+6}$. Therefore,

$$C(\wp(o_j)) \leq 2^{i+6\rho+13}$$

substituting k by $i+3\rho+6$, where ρ is a doubling constant. As x and v are $\text{dist}_G(x,v) = 2^i$ apart, the optimal cost for $\wp(o_j)$

$$C^*(\wp(o_j)) \geq 2^i.$$

Therefore, $C(\wp(o_j)) \leq \mathcal{O}(1) \cdot C^*(\wp(o_j))$. The theorem follows. □

4.2.2 When Queries Overlap with Maintenance Operations

In this case, a *query* operation chases a moving object as many *maintenance* operations may change the destination that the *query* operation was supposed to reach. We need a distance notion which we define as follows. Assume that a *query* operation $\wp(o_j)$ from some node w starts at $start(\wp(o_j))$ (when w sends it to its parent node in $\mathsf{DPath}(w)$ for the very first time) and ends at $end(\wp(o_j))$ when the read-only copy of o_j arrives at w. A *maintenance* operation p issued by some node v is said to be overlapping with $\wp(o_j)$ if v has o_j at any time during the interval $\Delta(\wp(o_j)) = [start(\wp(o_j)), end(\wp(o_j))]$ or if p is outstanding at any time during the interval $\Delta(\wp(o_j))$; the operation is outstanding after it is issued and before it is served. Therefore, we need to redefine the distance $\text{dist}_G(.,.)$ of such *query* operation $\wp(o_j)$ from any node $w \in G$ to the maximum shortest path distance from w to the source node of any overlapping *maintenance* operation in G. Now adapting the analysis technique used in [32, Section 5.4.2], we can prove the following theorem.

Theorem 4.12 *The cost ratio of Algorithm 1 for any query operation is $O(1)$ in constant-doubling networks even when the query operation overlaps with one or more maintenance operations.*

5 Load Balancing

We now show how to modify MOT to solve the tracking problem in a way that balances the load among the network nodes by distributing the detection list of the internal nodes to other nodes in their neighborhood. We prove the load result for *growth-restricted networks*, which are a slightly restricted variations of constant-doubling networks. Let $N(x, r)$ be the radius r-neighborhood of x. In growth-restricted networks, there exists a constant which bounds the ratio between $N(x, 2r)$ and $N(x, r)$ for arbitrary node x and arbitrary radius r.

We give a simple variation of MOT to achieve load balancing. In $\mathcal{HS}$, we form a cluster around each internal node by including all the nodes that are within the specified radius of the central node. The radius of a cluster is defined according to the level of the central node. According to this cluster construction, if a central node is at level i, then all the nodes in G that are within radius 2^i of that central node are included in the cluster. We now allow the central node to distribute its detection list to other nodes within the cluster so such that the object load at each node is minimized. This can be done through a hash function. We assign a unique key to each object such that $key(o_i) \in [1...m]$. Likewise, we assign identifiers to each node in a cluster X such that $ident(X) \in [0...|X| - 1]$, where $|X|$ is the number of nodes in the cluster. Each object o_i is now stored in the detection list of a node with label $key(o_i) \mod |X|$ where X is the relevant cluster. The purpose of the hash function is to distribute the objects evenly among the nodes in the cluster, so that the load at any node is minimized.

The load of the MOT algorithm is the total number of objects and the bookkeeping information that it needs to store in each physical (proxy) node to track m mobile objects. We assume that each node may act as a proxy for up to m_1 objects (maximum number of objects proxied by any one node). We compare the total information stored by MOT in any node, including its detection list and proxy list, to m_1 to obtain the *load ratio*.

Theorem 5.1 *MOT achieves load ratio of $\mathcal{O}(\log D)$ on average in growth-restricted networks.*

Proof. According to the construction of $\mathcal{HS}$, each physical node $x \in G$ has constant number of parents (at most $2^{3\rho}$ number of parent in the $parentset^j(x)$ for $1 \leq j \leq h$) in each level. Moreover, summing over x's role up to h levels, the total number of parents is $\mathcal{O}(h) = \mathcal{O}(\log D)$. Taking the expectation over uniform object key distribution through a universal hash function, following the analysis approach of [14], the expected object storage load at x at level ℓ is $\mathcal{O}(m_1)$ for fixed ℓ, where m_1 is the maximum number of objects proxied at any one physical node of G. Assuming that nodes generate uniform load and this load condition holds at all times, the expected object load at x is $\mathcal{O}(m_1 \cdot \log D)$ when summing over all different levels. □

To retrieve the object information that is distributed to other nodes inside the cluster efficiently, we embed a de Bruijn graph in each cluster. If a de Bruijn graph is not embedded, then large tables are needed to translate the virtual node levels provided to the nodes within the cluster in the distribution process into physical addresses. However, the embedding of a de Bruijn graph makes objects distributed in the cluster nodes searchable in $\mathcal{O}(\log n)$ time in every cluster by allowing every node of the cluster to store only a constant number of IDs of the neighboring nodes inside the cluster. Therefore, we obtained the desired bound for load in Theorem 5.1. Without this embedding, every node may need to store about as much as $\mathcal{O}(n)$ IDs of the neighboring nodes. The expense of a de Bruijn graph embedding is an increase of a $\mathcal{O}(\log n)$ factor in *maintenance* and *query* costs.

The embedding is done as follows using ideas from Rajaraman *et al.* [28]. We embed a $d = \lceil \log |X| \rceil$-dimensional de Bruijn graph into each cluster X, where $|X|$ is the number of nodes in that cluster. The d-dimensional de Bruijn graph consists of 2^d vertices whose labels are binary strings of length d. In other words, the nodes in the cluster X are assigned integers from $[|X|]$. Any de Bruijn graph vertex with label $\ell \in [|X|]$ is hosted by the cluster node u with level ℓ. Any de Bruijn graph vertex with label $\ell > |X|$ is hosted by the cluster node u with level l except the most significant bit. Note that the labels of the nodes are all integers. In the de Bruijn graph, there is a directed edge from each vertex with label $u_1 u_2 \dots u_d$ to the vertices with labels $u_2 \dots u_d 0$ and $u_2 \dots u_d 1$. The diameter of this directed graph is $\log |X|$ and there is a unique shortest path between every

pair of vertices which can be computed easily. If a node v in cluster X needs to send a message to node $\ell(X)$ (the leader of the cluster X), for some object o_j, then this message follows the edges on the shortest path from v to $\ell(X)$ in the de Bruijn network. To follow the shortest path, every node in this shortest path only needs to know the physical address of the next node on the path. This information is obtained when every node in the path stores the physical address of the nodes incident on its outgoing edges. Since the in-degree and out-degree of each vertex in the de Bruijn graph is at most 2, the neighborhood table at each node is of constant size.

The cost of routing messages along shortest paths in the de Bruijn network is $\mathcal{O}(D_X \cdot \log |X|)$ instead of $\mathcal{O}(D_X)$ because a shortest path visits up to $\mathcal{O}(\log |X|)$ intermediate nodes, where D_X is the diameter of the cluster X. Therefore, we have the following corollary.

Corollary 5.2 *Due to de Bruijn graph embedding in constant-doubling networks, the maintenance cost ratio of Algorithm 1 becomes* $\mathcal{O}(\min\{\log n, \log D\} \cdot \log n)$ *and the query cost ratio becomes* $\mathcal{O}(\log n)$*. However, the increase of a* $\mathcal{O}(\log n)$ *factor in maintenance and query cost ratios allows to balance the load in the nodes of* $\mathcal{HS}$ *such that the load ratio is* $\mathcal{O}(\log D)$ *on average per node.*

6 Extensions to General Networks

We now show the scalability of our tracking algorithm in general network topologies. We use a $(\mathcal{O}(\log n), \mathcal{O}(\log n))$ *partition scheme* of [4, 15, 33] to maintain an overlay structure $\mathcal{HS}$. A similar scheme is used by [27] to solve the problem of distributing information from a collection of sources to mobile users in a wireless mesh network. This scheme can also be computed through a distributed algorithm in a message efficient manner following the technique presented in Gao *et al.* [12].

There are $h \leq \lceil \log D \rceil + 1$ levels in this $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-partition scheme. At the bottom level (level 0), each node in V belongs to exactly one cluster which consists only of the node itself. At the top level (level h), there is only one cluster that contains all the nodes in V. In any level $\ell, 1 \leq \ell \leq h-1$, each node $u \in V$ belongs to exactly $\mathcal{O}(\log n)$ clusters. The $\mathcal{O}(\log n)$ clusters at each level are provided with different integer labels. A leader node is selected arbitrarily for each cluster from the nodes that are in that cluster so that $\mathcal{HS}$ can have a hierarchical structure of leader nodes of all clusters at all the levels. These leader nodes act similar to the internal nodes of $\mathcal{HS}$ we developed in Section 2.

Similar to Section 2, the parent set $parentset^{\ell}(x)$ of node x in $\mathcal{HS}$ consists of all the parent nodes within distance $\mathcal{O}(2^{\ell} \cdot \log n)$ of x. According to this construction, the total number of nodes in the parent set is $\mathcal{O}(\log n)$. Parent child pairs here are also connected similarly to Section 2. The $\mathsf{DPath}(u)$ for each bottom level sensor node (proxy or non-proxy) $u \in V$ visits the leader nodes in $parentset^{\ell}(u)$ at level ℓ that u belongs to according to the label of the clusters starting from the smallest label cluster and ending at the largest label cluster. The cluster labeling is derived from labeling of the $\mathcal{O}(\log n)$ partition hierarchies of $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-partition scheme. The details can be found in [33]. We have these following results in general networks.

Lemma 6.1 *For any two nodes* $u, v \in V$ *in general networks, their detection paths* $\mathsf{DPath}(u)$ *and* $\mathsf{DPath}(v)$ *intersect at level* $\min\{h, \lceil \log(\mathrm{dist}_G(u,v)) \rceil + 1\}$*. Moreover, the length of the detection path of any node* $u \in V$ *up to any level* j *is* $\mathrm{length}(\mathsf{DPath}_j(u)) \leq \mathcal{O}(2^j \cdot \log^2 n)$*.*

Proof. The first part follows arguing similar to Lemma 2.1. The proof of the second part is also similar to Lemma 2.2, but now $\mathrm{length}(\mathsf{DPath}_j(u))$ increases by the factor of $\mathcal{O}(\log^2 n)$ in general networks. This is because, in contrast to $\mathcal{HS}$ for constant-doubling networks, there are $\mathcal{O}(\log n)$ nodes in the parent set at level $\ell+1$ for any leader node at level ℓ in $\mathcal{HS}$ for general networks. Moreover, any node in the parent set at level $\ell+1$ can be as much as $\mathcal{O}(2^{\ell+2} \cdot \log n)$ distance away from a child node at level ℓ. Therefore, summing up the lengths of the root path segments for each level, $\mathrm{length}(\mathsf{DPath}_j(u))$ for any node u up to level j increases by the factor of $\mathcal{O}(\log^2 n)$ in comparison to the detection path length of Lemma 2.2. □

Theorem 6.2 *The maintenance cost ratio of Algorithm 1 is* $\mathcal{O}(\log^2 n \cdot \min\{\log n, \log D\})$ *in general networks in any (one by one or concurrent) execution.*

Proof. For the upper bound, similar to Lemma 4.2, we have that for all the operations in $\mathcal{E}_j$ that reach level k following their respective detection paths,

$$C_k(\mathcal{E}_j) \leq s_{k,j} \cdot \mathcal{O}(2^k \cdot \log^2 n).$$

This is because $\mathsf{DPath}_k(u)$ for any node u is $\mathcal{O}(2^k \cdot \log^2 n)$ (Lemma 6.1). Therefore, the total communication cost of Algorithm 1 in a general network is

$$C(\mathcal{E}_j) \leq \sum_{k=1}^{h} C_k(\mathcal{E}_j) \leq \sum_{k=1}^{h} s_{k,j} \cdot \mathcal{O}(2^k \cdot \log^2 n).$$

For the lower bound, arguing similarly as in Lemma 4.3 using intersection property of detection paths given in Lemma 6.1, we have that for the *maintenance* operations that reach level k in $\mathcal{HS}$ it holds that

$$C_k^*(\mathcal{E}_j) \geq (s_{k,j} - 1) \cdot 2^{k-1}.$$

Therefore, considering all the levels $1 \leq k \leq h$,

$$C^*(\mathcal{E}_j) \geq \max_{1 \leq k \leq h} (s_{k,j} - 1) \cdot 2^{k-1}.$$

Comparing these two bounds, we have that

$$\begin{aligned} C(\mathcal{E}_j) &\leq \leq \sum_{k=1}^{h} C_k(\mathcal{E}_j) \leq \sum_{k=1}^{h} s_{k,j} \cdot \mathcal{O}(2^k \cdot \log^2 n) \\ &\leq c_1 \cdot h \cdot \log^2 n \cdot \max_{1 \leq k \leq h} s_{k,j} \cdot 2^k \\ &\leq c_2 \cdot h \cdot \log^2 n \cdot C^*(\mathcal{E}_j) \end{aligned}$$

with a reduction similar to Theorem 4.4, where $h \leq \lceil \log D \rceil + 1$, and c_1 and c_2 are some positive constants.

The total communication cost $C(\mathcal{E}_j)$ of Algorithm 1 for the operations in $\mathcal{E}_j$ calculated above is already small if $D < n^{\mathcal{O}(1)}$. We now argue that $\mathcal{O}(h) = \mathcal{O}(\log D)$ factor in $C(\mathcal{E}_j)$ can be replaced with $\mathcal{O}(\log n)$ even when $D > n^{\mathcal{O}(1)}$. This can be proved from a fine-tuned analysis similar to the one given in Theorem 4.7 (Section 4.1.1) for constant-doubling networks. Therefore, combining these two different cost bounds for $C(\mathcal{E}_j)$ and using the argument of Theorem 4.8, we obtain that the *maintenance* cost ratio of Algorithm 1 is $\mathcal{O}(\min\{\log^3 n, \log^2 n \cdot \log D\})$ in general networks. Note that this analysis is for the one by one case. For the concurrent case, adapting the proof technique outlined in Section 4.1.2, we can prove the same cost ratio of $\mathcal{O}(\min\{\log^3 n, \log^2 n \cdot \log D\})$ for Algorithm 1 in general networks. □

Lemma 6.3 *A query operation $\wp(o_j)$ from any sensor node x for the object o_j at proxy node v at distance $\mathrm{dist}_G(x, v) = 2^i$ is guaranteed to find o_j in the detection list of an internal node in $\mathsf{DPath}(x)$ at level $k \leq i + 2 + 2 \log\log n + \log c$, for some positive constant c in the scenario where query operations do not overlap with any maintenance operations.*

Proof. We argue similar to Lemma 4.10. Lets assume that w is a leader node of cluster X at level $k = i + 2 + \log\log n + \log c$, which has special-child the level i leader node v_i (set by some previous *insert* operation). For the *query* operation $\wp(o_j)$ to find the special-child to v_i at w, w must include x since the root path $\mathsf{DPath}(x)$ of x visits the leaders of all clusters that contain it. It suffices to show that the neighborhood 2^{k-1} of X is at least the distance $\mathrm{dist}_G(v_i, w)$ between v_i and w to guarantee that w is a parent node in $\mathsf{DPath}(x)$. As the length of the downward path towards v from level i node v_i is also at most $c \cdot 2^i \cdot \log^2 n$ (the length of the detection path given in Lemma 6.1)

and $\text{dist}_G(x, v) = 2^i$, we get

$$\begin{aligned}
\text{dist}_G(v_i, w) &\leq c \cdot 2^i \cdot \log^2 n + \text{dist}_G(x, v) \\
&\leq c \cdot 2^i \cdot \log^2 n + 2^i \\
&\leq c \cdot 2^{i+1} \log^2 n \\
&\leq 2^{i+1+2 \log\log n + \log c} \\
&\leq 2^{k-1},
\end{aligned}$$

for $k = i + 2 + 2\log\log n + \log c$. Hence, some node should be within at most $2^{i+2+2\log\log n+\log c}$ distance to have that information, i.e., such node will be at level $k = i + 2 + 2\log\log n + \log c$. □

Theorem 6.4 *The cost ratio of Algorithm 1 for any query operation is $\mathcal{O}(\log^4 n)$ in general networks in any (non-overlapping or overlapping) execution of query operations with maintenance operations.*

Proof. First we consider the non-overlapping execution of the *query* operation $\wp(o_j)$ with *maintenance* operations. Similar to Lemma 4.12, $C(\wp(o_j)) \leq \text{length}(\mathsf{DPath}_k(x)) + \text{dist}_G(w, v_i) + \text{length}(\mathsf{DPath}_i(v))$ since the length of the downward path to the object from the level i node is also at most the length of the detection path from proxy node to a level i internal node. As $k \leq i + 2 + 2\log\log n + \log c$, $\text{length}(\mathsf{DPath}_k(x)) \leq \mathcal{O}(2^k \cdot \log^2 n)$, $\text{dist}_G(w, v_i) \leq \mathcal{O}(2^k \cdot \log n)$, and $\text{dist}_G(v, v_i) \leq \mathcal{O}(2^i \cdot \log^2 n)$, we have that

$$\begin{aligned}
C(\wp(o_j)) &\leq \mathcal{O}(2^k \cdot \log^2 n) + \mathcal{O}(2^k \cdot \log n) + \mathcal{O}(2^i \cdot \log^2 n) \\
&\leq \mathcal{O}(2^k \cdot \log^2 n) \\
&\leq \mathcal{O}(2^{i+2+2\log\log n+\log c} \cdot \log^2 n) \\
&\leq \mathcal{O}(2^i \cdot \log^4 n).
\end{aligned}$$

Moreover, as x and v are 2^i apart, $C^*(\wp(o_j)) \geq 2^{i-1}$. Therefore $C(\wp(o_j))/C^*(\wp(o_j)) \leq \mathcal{O}(2^i \cdot \log^4 n)/2^{i-1} = \mathcal{O}(\log^4 n)$, as needed. When the *query* operation $\wp(o_j)$ overlaps with one or more *maintenance* operations, following the technique outlined in 4.2.2, we can prove the same cost ratio of $\mathcal{O}(\log^4 n)$ for $C(\wp(o_j))/C^*(\wp(o_j))$ using the result of [32]. The theorem follows. □

Theorem 6.5 *Algorithm 1 achieves load ratio of $\mathcal{O}(\log^2 n \cdot \log D)$ in general networks.*

Proof. Suppose that the maximum number of objects that can be stored in the memory of any sensor node is m_1. According to our construction of $\mathcal{HS}$ for general networks, any sensor node of G is contained in $\mathcal{O}(\log n \cdot \log D)$ clusters. From the result of [28, Lemma 8], using an $\mathcal{O}(\log(n \cdot m_1))$-wise independent hash function to distribute the objects to the nodes of the clusters, it can be shown that, for every cluster X, each node $v \in X$ holds information (both object and bookkeeping) about at most $\mathcal{O}(\frac{(m_1+\log n)\cdot\log(n\cdot m_1)}{\log n})$ objects with high probability. Moreover, the representation of this hash function can be done using $\mathcal{O}(\log(n \cdot m_1))$ words and the de Bruijn neighborhood can be store using only two words; note that a de Bruijn graph can be embedded to each cluster through a procedure similar to the one described in Section 5. Therefore, total object load at any node is

$$\mathcal{O}\left(\log n \cdot \log D \cdot \left(\frac{(m_1 + \log n) \cdot \log(n \cdot m_1)}{\log n} + 2\right) + \log(n \cdot m_1)\right).$$

Assuming that $m_1 \leq n^{\mathcal{O}(1)}$, i.e., polynomially bounded on n, we have that the load ratio is $\mathcal{O}(\log^2 n \cdot \log D)$. Hence, the theorem follows. □

Due to the embedding of the de Bruijn graph in each cluster, the cost of routing messages along the shortest paths increases by a factor of $\mathcal{O}(\log n)$ when compared to routing messages without the de Bruijn graph embedding (Section 5). Therefore, we have the following corollary for the *maintenance* and *query* cost ratios in general networks.

Corollary 6.6 *Due to the de Bruijn graph embedding in general networks, the maintenance cost ratio of Algorithm 1 becomes* $\mathcal{O}(\log^3 n \cdot \min\{\log n, \log D\})$ *and the query cost ratio becomes* $\mathcal{O}(\log^5 n)$. *However, the increase of* $\mathcal{O}(\log n)$ *factor in maintenance and query ratio allows to balance the load in the nodes of* $\mathcal{HS}$ *such that the load ratio is* $\mathcal{O}(\log^2 n \cdot \log D)$ *on average per node.*

7 Handling Node and Edge Failures

We assumed so far that nodes and links do not crash; this assumption may be too strong for practical wireless sensor networks as nodes are prone to battery depletion and availability of links vary. In this section, we outline how to extend MOT for dynamic networks where nodes join and leave over time. Joining and leaving of sensor nodes change the availability of links as well. We argue below that MOT can be made adaptable to these network changes. The adaptability can be defined as the number of nodes of $\mathcal{HS}$ that have to be updated when a node joins or leaves the sensor network.

We need to adjust the hierarchical structure $\mathcal{HS}$ and transfer the object states appropriately. If the node that is leaving the system is a leader, the leadership information should be transferred to some other node of that cluster by electing that node as a leader for that cluster. This new leader information should be propagated to all the nodes in the cluster. For this to work, we have to assume that the nodes announce their failures (or departures) before they actually fail. Otherwise, the object and pointer information at that node will be lost. This assumption has widely been used in the literature and we argue that it is not much of a restriction. Moreover, a sequence of nodes joining the system may make the diameter of the cluster too big. Similarly, the leaving of a sequence of nodes may make the cluster disjoint. These issues can be tackled by putting some threshold on how much clusters can grow or become disjoint. After the threshold, the hierarchy can be rebuilt from scratch.

The hierarchical structures $\mathcal{HS}$ built in Sections 2 and 6 respectively for constant-doubling and general networks are adaptable to changes due to the leaving and joining of a few nodes without the need of rebuilding of them from scratch up to some threshold. We assume that a de Bruijn graph is already embedded in each cluster of the network as described in Section 5 for constant-doubling networks and Theorem 6.5 for general networks.

We now provide some details on how to adapt to the changes when a node, say p, leaves the network, and argue that for a sequence of node additions, the amortized adaptability of our technique is $\mathcal{O}(1)$ within a cluster for both constructions. The description below is for one cluster X in which p belongs; this can be extended to other clusters of $\mathcal{HS}$ where p belongs accordingly. Remember that the adaptability discussion heavily depending on the de Bruijn graph embedding and we borrow the techniques of [28] here. Let $\ell(p)$ be the label of p in cluster X after embedding a de Bruijn graph. If $\ell(p) = |X|-1$ and $|X|-1$ is not a power of 2, then we emulate the label $\ell(p)$ by the node $p' \in X$ that has a label $\ell(p')$ which is identical to p except the most significant bit. If p was the leader of X then p' now becomes the new leader and this information is propagated to the leaders at the parent and child clusters of X, and also to the other nodes inside X. The neighboring nodes list that currently holds by p' in the embedded de Bruijn graph is updated to reflect also the list of neighbors of p; the objects and bookkeeping information is also transferred to p' from p and updated accordingly. This implies that only $\mathcal{O}(1)$ nodes are updated in this process. If $|X|-1$ is a power of 2, then from the embedding we have that all the nodes in X except p has two labels. Therefore, we decrease the dimension of the embedded de Bruijn graph by 1 such that each node in X now maintains exactly one of its labels and merge the bookkeeping information associated with the two labels. This implies that all $|X|$ nodes are updated. We can choose any node in X to become a new leader and the information about new leader is propagated to other nodes as described above. For the case where $\ell(p)$ is less than $|X|-1$, we can set $\ell(p)$ the label of the node with current label $|X|-1$ and proceed similarly as of the removal of the node with label $|X|-1$. That node also works as the leader of X. Therefore, arguing through amortization, we have that the amortized adaptability for a sequence of node departures is $\mathcal{O}(1)$ with in a cluster.

We now provide some details on how to adapt to the change when a node, say p, joins the network, and argue that the adaptability for a sequence of node arrivals is $\mathcal{O}(1)$ within a cluster as

well for both constant-doubling and general network $\mathcal{HS}$ constructions. Note that leader election is not required here and hence we omit the discussion related to it. The node p is assigned a new label $|X|$ as soon as it joins a cluster X. If $|X|+1$ is not a power of 2 after the addition of p, the node p' that has label $|X|$ (before the addition of p), and the nodes that have de Bruijn edges to p' are updated to reflect the changes. It is immediate that the number of nodes that are updated are $\mathcal{O}(1)$. If $|X|+1$ is a power of 2, then previously all the nodes in $|X|$ had only one label, but now they need to emulate 2 labels which is done by increasing the dimension of the embedded de Bruijn graph by 1. Therefore, the number of nodes updated inside a cluster are $|X|$ and the amortized adaptability is again $\mathcal{O}(1)$ for a cluster, even for a sequence of node arrivals.

Therefore, the amortized adaptability within a cluster for a sequence that contains nodes both joining and leaving the system is $\mathcal{O}(1)$ for both constant-doubling and general networks. As each node belongs to $\mathcal{O}(\log D)$ levels in constant-doubling networks (Section 2) and $\mathcal{O}(\log n \cdot \log D)$ levels in general networks (Section 6), we have the amortized adaptability of $\mathcal{O}(\log D)$ for constant-doubling networks and $\mathcal{O}(\log n \cdot \log D)$ for general networks. Note that this amortized adaptability works until some threshold since the arbitrary number of addition and deletion of nodes may either make the diameter of a cluster too big or the cluster becomes disjoint, and rebuilding of the hierarchy is preferred after some threshold. The appropriate threshold can be determined based on the application requirement by maintaining some profiling parameter while the network is in operation. The complete treatment of these issues is currently outside the scope of this paper and therefore we leave them for future work.

8 Experiments

We have implemented our MOT algorithm and compared its performance with STUN algorithm due to Kung and Vlah [18] and Z-DAT algorithm due to Lin *et al.* [21]. For Z-DAT, we consider two variations of it: One is the original Z-DAT algorithm and the other is modified Z-DAT algorithm enriched with shortcuts which we denote as Z-DAT + shortcuts in the figures; the details of Z-DAT variations can be found in [21]. The difference among STUN, Z-DAT, and MOT is in the tracking structure construction and the properties they hold. We note here that STUN and Z-DAT do not necessarily minimize *maintenance* and/or *query* costs in some special networks, e.g. ring networks. Moreover, they do not address the load balancing issue and also assume the availability of traffic knowledge. We performed the experiments on grid networks of sizes ranging from 10 nodes to 1024 nodes with 100 and 1000 mobile objects. We used the static grid networks and leave the case of dynamic networks for future work. The plots are the average of 5 experiments. All the algorithms are implemented in Java and results are obtained from the simulation. We provide *maintenance* and *query* cost ratio results as well as load results for all the algorithms considered in simulation in both one by one and concurrent execution of the operations to cover wide range of practical applications.

We first provide experimental evaluation results obtained from one by one execution of operations. Note that, in one by one executions, there is only one operation executing at any time for any single object. We start with *maintenance* cost ratio results. Fig. 4 compares the cost ratio of the algorithms in performing 1000 *maintenance* operations per object in random order for 100 mobile objects. Similarly, Fig. 5 compares the *maintenance* cost ratios in the same setting of Fig. 4 for 1000 mobile objects. The cost ratio of MOT is significantly better compared to that of STUN for both 100 and 1000 objects. This is because of the fact that STUN relies on the spanning tree based tracking structure which does not necessarily minimize the tracking cost even in the case where the traffic knowledge is available. The cost ratio of MOT matches the performance of Z-DAT and Z-DAT with shortcuts. Although MOT has a small overhead compared to Z-DAT variations, we will see later that the load will be significantly reduced in network nodes while using MOT.

We now provide *query* cost ratio results. Fig. 6 shows the comparison of *query* cost ratios of the algorithms in performing a *query* operation for 100 mobile objects and Fig. 7 shows this comparison for 1000 mobile objects, in the networks of sizes ranging from 10 nodes to 1024 nodes. The comparison results show that the performance of MOT is significantly better compared to STUN. Moreover, MOT performs more or less in synch with Z-DAT and Z-DAT with shortcuts. On

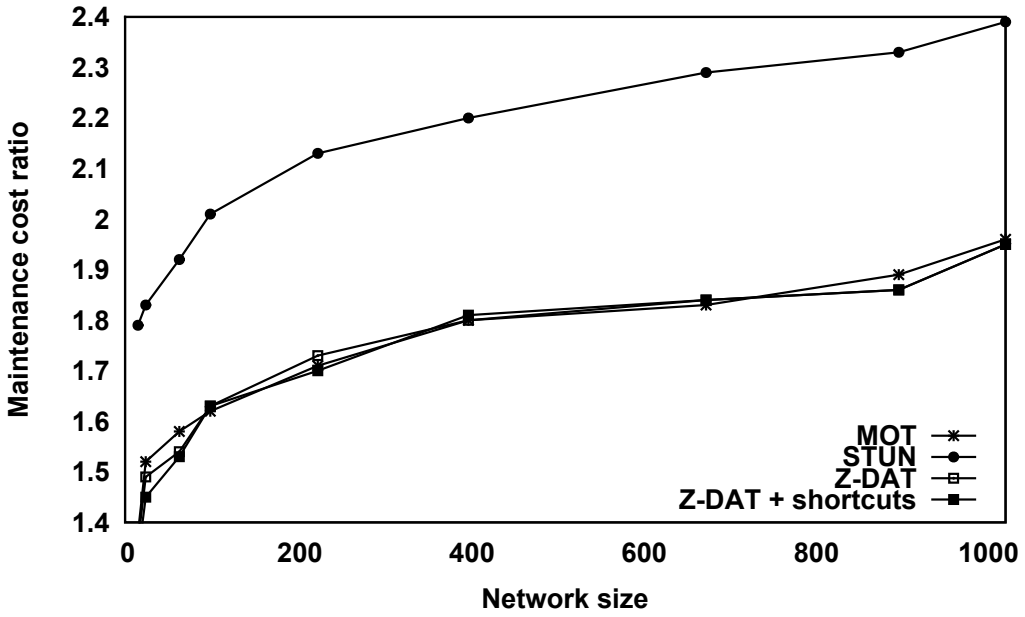


Figure 4: Comparison of *maintenance* cost ratio results in performing 1,000 operations per object in the networks of size 10 to 1024 nodes with 100 objects in one by one execution scenario. Lower is better.

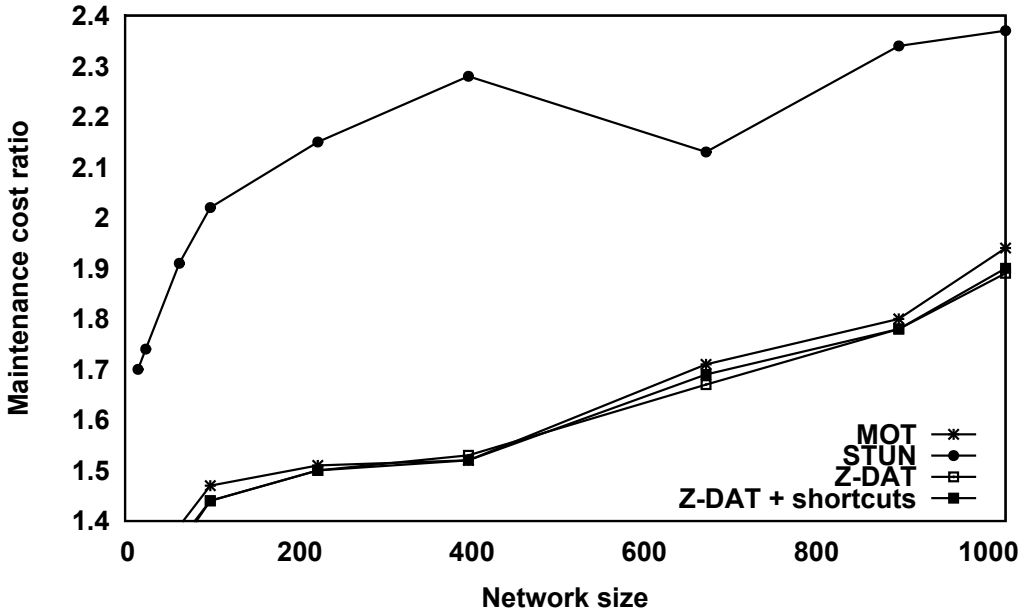


Figure 5: Comparison of *maintenance* cost ratio results in performing 1,000 operations per object in the networks of size 10 to 1024 nodes with 1000 objects in one by one execution scenario. Lower is better.

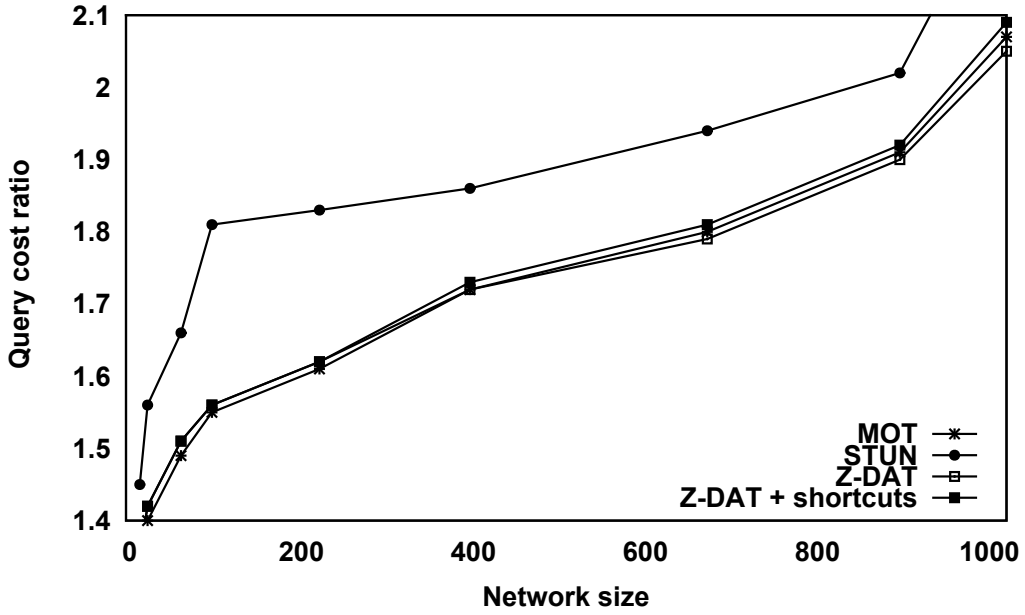


Figure 6: Comparison of *query* cost ratio results in performing a query operation in the networks of size 10 to 1024 nodes with 100 objects in one by one execution scenario. Lower is better.

closer observation, it is evident that MOT can only do as good as Z-DAT with shortcuts. This is because the Z-DAT with shortcuts algorithm not only chooses the shortest paths at all times but also saves the relevant details at parent nodes.

We now compare the load in the corresponding tracking data structures just after they are initialized and after some *maintenance* operations are performed. Fig. 8 shows the load/node results comparison of MOT and STUN in tracking 100 mobile objects in the network of 1024 nodes just after their tracking data structures are initialized. As STUN does not address the load balancing issue, it has uneven load among the nodes; some of the nodes have load proportional to the number of objects (actually, the nodes at higher levels of the hierarchical structure used in STUN), which might be a significant bottleneck in sensor networks as sensor nodes are generally memory-constrained. MOT utilizes the available network nodes to distribute the load, avoiding possible bottlenecks due to uneven utilization of the computing resources. Fig. 9 shows the load/node results comparison of MOT and STUN in the same setting of Fig. 8 after 10 *maintenance* operations per object are finished execution; the *maintenance* operations are performed in a random order, that is, subsequent *maintenance* operations are not necessarily for the same object. Figs. 10 and 11 show the load/node results comparison of MOT and Z-DAT in the same settings of Figs. 8 and 9, respectively. We omit the load results of Z-DAT with shortcuts algorithm as the results are similar to Z-DAT algorithm.

It is evident from Figs. 8-11 that load is well balanced in MOT compared to other existing algorithms. As the load balancing procedure of MOT kicks in when a *maintenance* operation floods the detection list of an internal node of $\mathcal{HS}$ with more objects, the *maintenance* operation takes more time to find the information that is distributed in the other nodes inside the cluster while it is executing. Therefore, the *maintenance* cost ratio of MOT is slightly worse compared to that of Z-DAT and Z-DAT with shortcuts, however, it in turn allows us to achieve load balancing.

We now provide experimental evaluation results obtained from concurrent execution of operations. For the concurrent execution of operations, we follow the same setting as of one by one execution of operations except that there are more than one operation running for any single object during execution. We fix the maximum number of concurrent operations for an object at any time to 10. Note that in one by one execution of operations, there were at most one operation for any object at any time. We again perform 1000 *maintenance* operations and a *query* operation per ob-

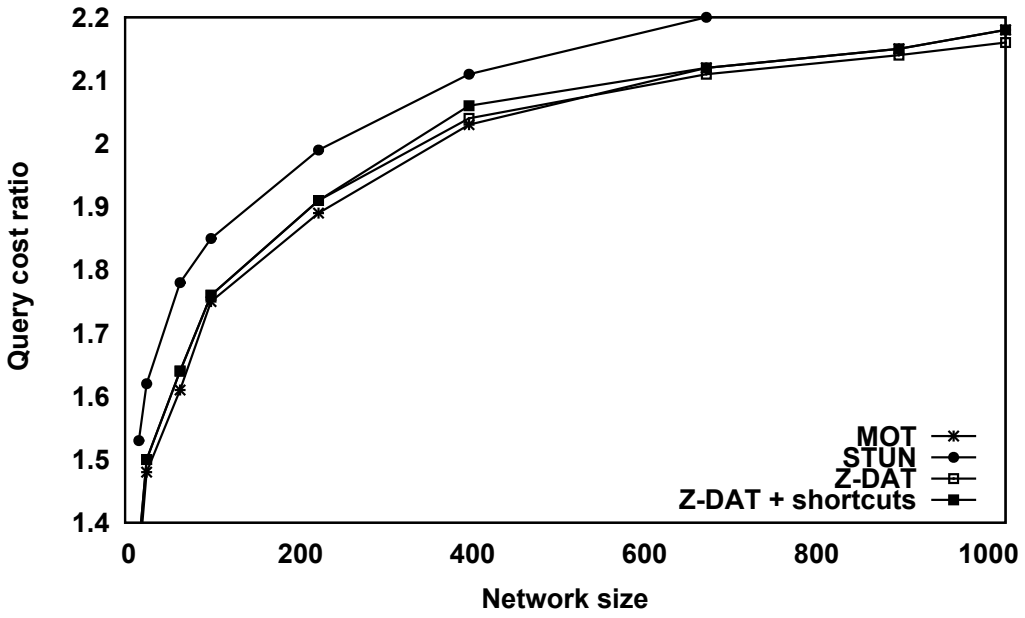


Figure 7: Comparison of *query* cost ratio results in performing a query operation in the networks of size 10 to 1024 nodes with 1000 objects in one by one execution scenario. Lower is better.

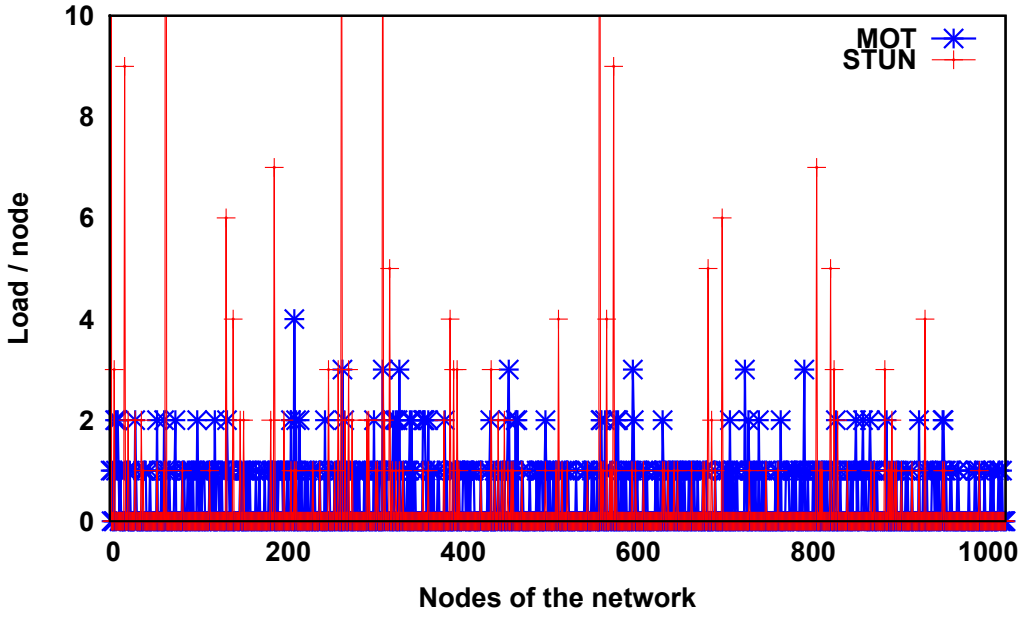


Figure 8: Comparison of load/node results of MOT and STUN in tracking 100 mobile objects in the network of 1024 nodes just after the initialization of the corresponding tracking structures in one by one execution scenario. There are 5 nodes in STUN with load > 10 and no node in MOT with load > 10. Lower is better.

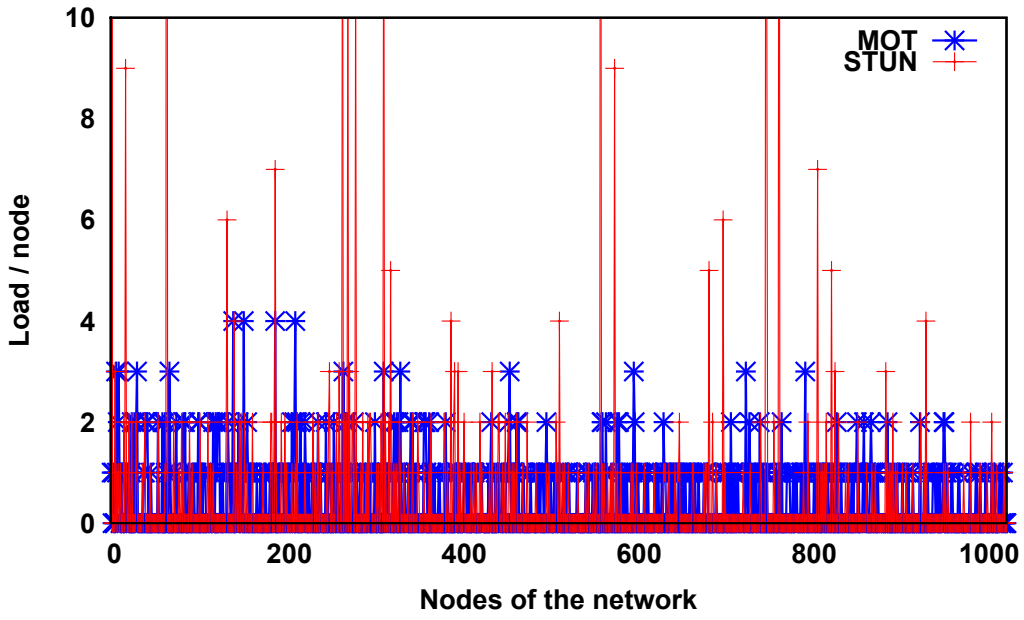


Figure 9: Comparison of load/node results of MOT and STUN in tracking 100 mobile objects in the network of 1024 nodes after 10 *maintenance* operations per object are finished execution in one by one execution scenario. There are 7 nodes in STUN with load > 10 and no node in MOT with load > 10. Lower is better.

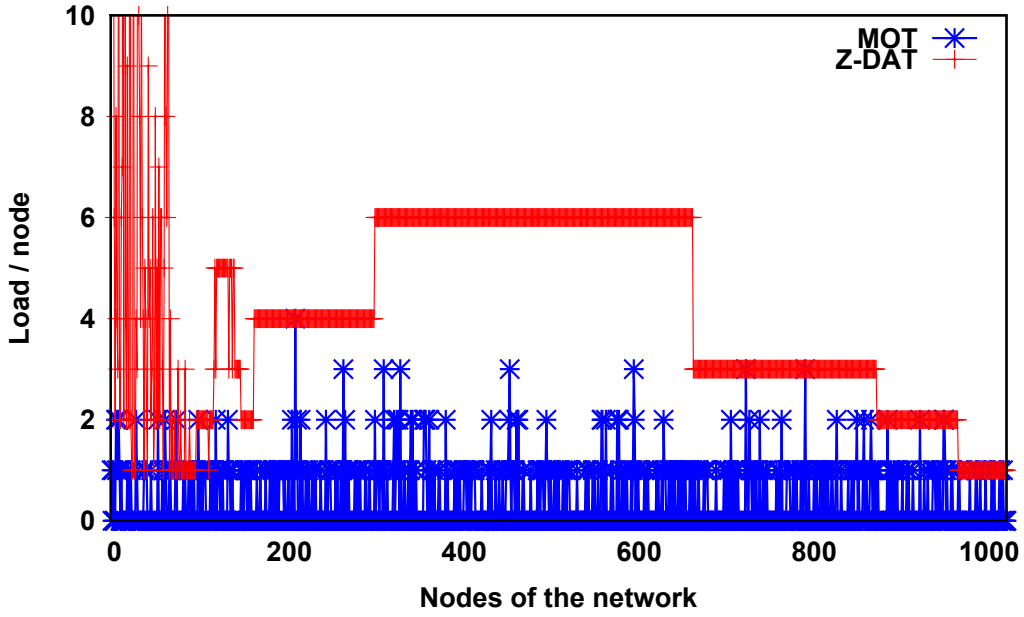


Figure 10: Comparison of load/node results of MOT and Z-DAT in tracking 100 mobile objects in the network of 1024 nodes just after the initialization of the corresponding tracking structures in one by one execution scenario. There are 14 nodes in Z-DAT with load > 10 and no node in MOT with load > 10. Lower is better.

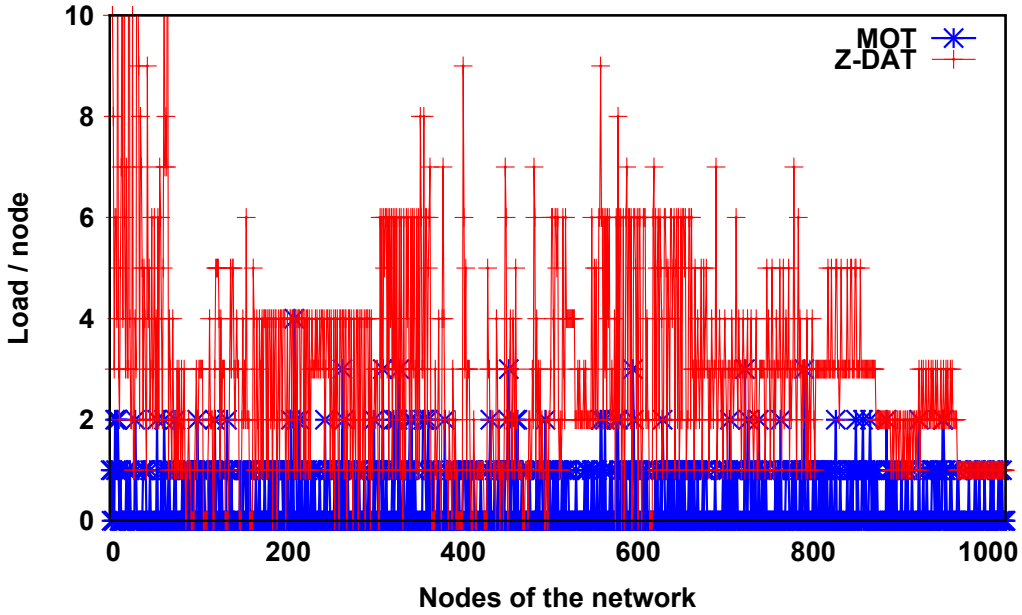


Figure 11: Comparison of load/node results of MOT and Z-DAT in tracking 100 mobile objects in the network of 1024 nodes after 10 *maintenance* operations per object are finished execution in one by one execution scenario. There are 11 nodes in Z-DAT with load > 10 and no node in MOT with load > 10. Lower is better.

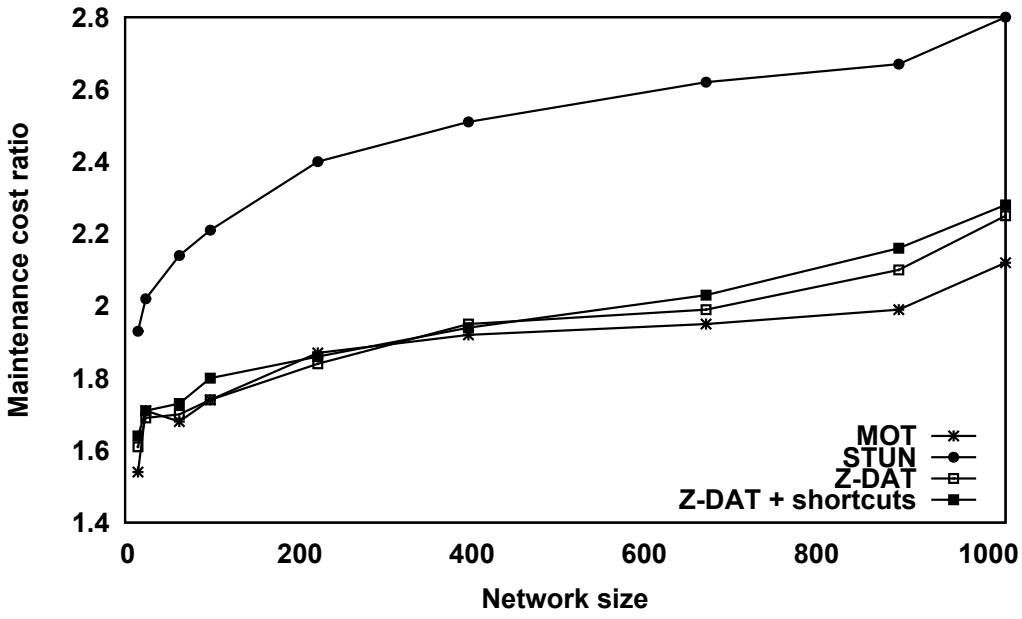


Figure 12: Comparison of *maintenance* cost ratio results in performing 1,000 operations per object in the networks of size 10 to 1024 nodes with 100 objects in concurrent execution scenario. Lower is better.

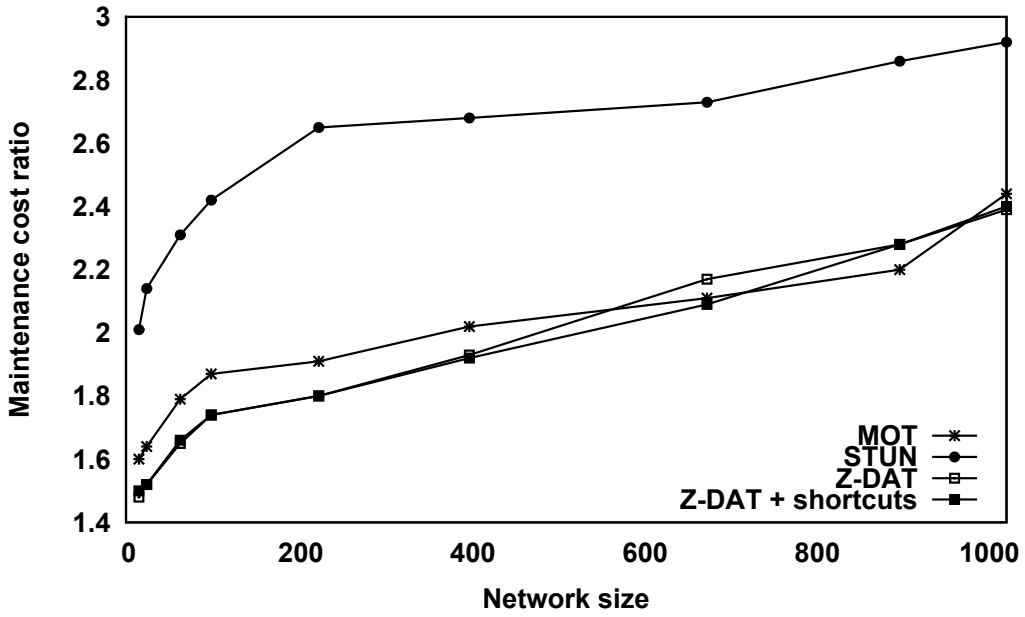


Figure 13: Comparison of *maintenance* cost ratio results in performing 1,000 operations per object in the networks of size 10 to 1024 nodes with 1000 objects in concurrent execution scenario. Lower is better.

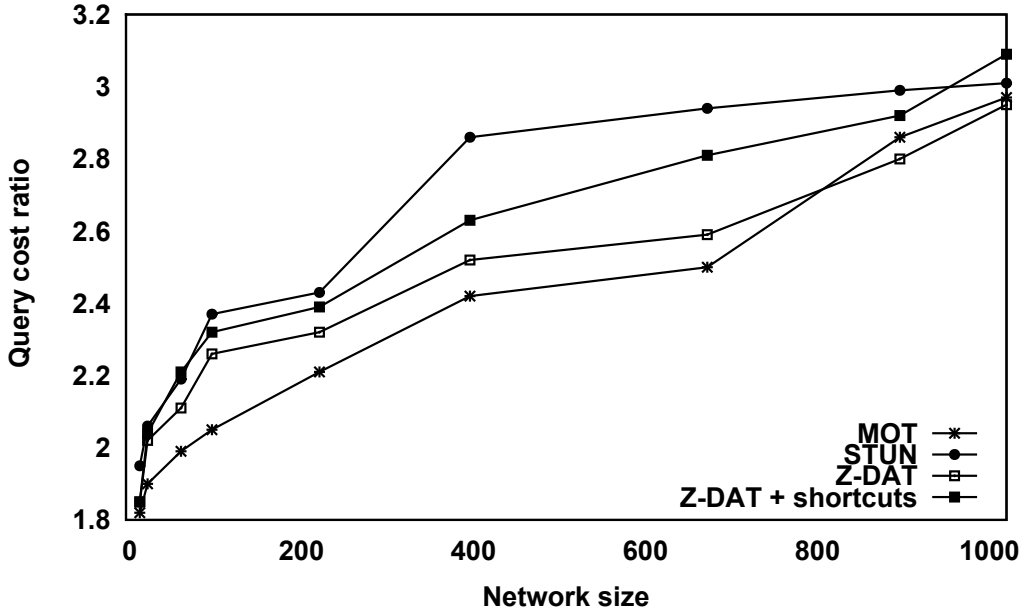


Figure 14: Comparison of *query* cost ratio results in performing a query operation in the networks of size 10 to 1024 nodes with 100 objects in concurrent execution scenario. Lower is better.

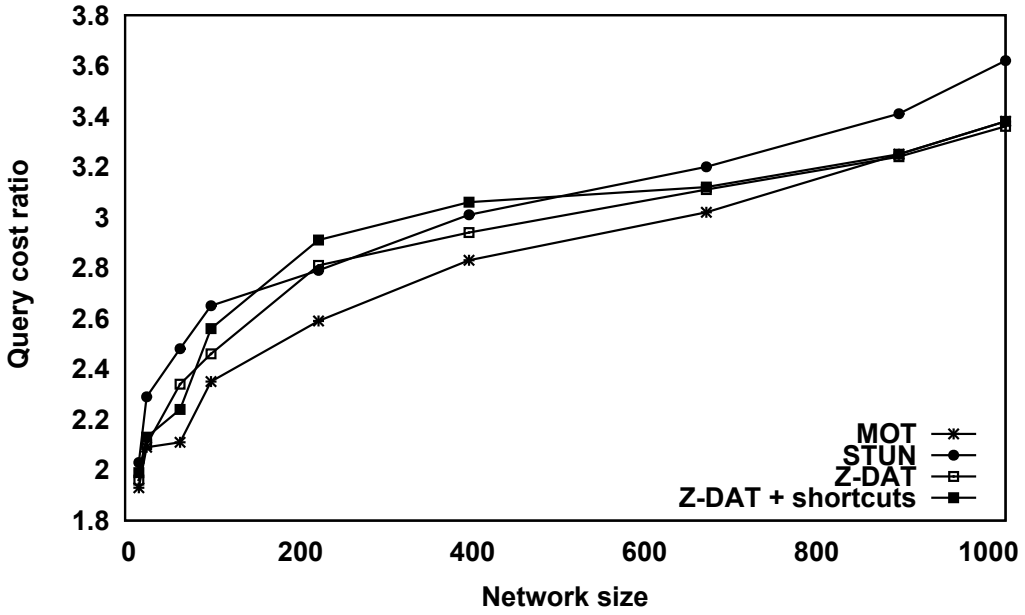


Figure 15: Comparison of *query* cost ratio results in performing a query operation in the networks of size 10 to 1024 nodes with 1000 objects in concurrent execution scenario. Lower is better.

ject and they are executed concurrently as mentioned above. For simplicity, we start 10 concurrent operations for some other object (chosen in random order) after 10 concurrent operations for one object finished execution. As changes in $\mathcal{HS}$ due to operations for one object do not interfere with the changes in $\mathcal{HS}$ due to operations for any other object, our setting for concurrent execution is not much of a restriction. For the fair comparison of results, each node generates equal number of operations for the same object as in the one by one execution.

Fig. 12 compares the cost ratio of the algorithms in performing 1000 *maintenance* operations per object in random order for 100 mobile objects. Similarly, Fig. 13 compares the *maintenance* cost ratios in the same setting of Fig. 12 for 1000 mobile objects. In comparison to *maintenance* cost ratio results in one by one executions, we observed a very small factor increase in the *maintenance* cost ratio results of all the algorithms. However, MOT still performs significantly better compared to that of STUN for both 100 and 1000 objects. The gap on performance slightly increases between these algorithms in concurrent situations compared to their performance gap in one by one executions. The cost ratio of MOT matches again the performance of Z-DAT and Z-DAT with shortcuts similar to one by one execution scenario. Fig. 14 shows the comparison of *query* cost ratios of the algorithms in performing a *query* operation for 100 mobile objects and Fig. 15 shows this comparison for 1000 mobile objects, in the networks of sizes ranging from 10 nodes to 1024 nodes. The comparison results again show that the performance of MOT is significantly better compared to STUN and it performs similar to Z-DAT and Z-DAT with shortcuts. However, in comparison to the one by one execution results, the *query* cost ratio of all the algorithms increases by a small factor in concurrent scenarios. Besides *maintenance* and *query* cost performance, the load is well-balanced in MOT compared to other algorithms even in concurrent executions. Actually, the load results in the concurrent scenario are similar to the load results given in Figs. 8-11 for one by one executions and hence we omit them here.

9 Conclusions

We presented a scalable distributed algorithm, MOT, for tracking mobile objects using a static sensor network. It is traffic-oblivious and balances the object load among network nodes; previous

approaches were traffic-conscious and were not load balanced. In constant-doubling networks, MOT has low data structure maintenance costs and resolves object queries at any time with near optimal cost. In more general topologies, it maintains the data structure and resolves queries with costs that are polylogarithmic factors away from the best possible costs. MOT also balanced the load of object information to be stored for tracking in the expense of a logarithmic factor increase in the *query* and *maintenance* operation costs. The experimental evaluation showed that MOT performs well in practical scenarios besides its near-optimal theoretical guarantees.

References

[1] Ittai Abraham, Danny Dolev, and Dahlia Malkhi. Lls: a locality aware location service for mobile ad hoc networks. In *DIALM-POMC*, pages 75–84, 2004.

[2] Noga Alon, Gil Kalai, Moty Ricklin, and Larry Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. *Theor. Comput. Sci.*, 130(1):175–201, 1994.

[3] Javed Aslam, Zack Butler, Florin Constantin, Valentino Crespi, George Cybenko, and Daniela Rus. Tracking a moving object with a binary sensor network. In *SenSys*, pages 150–161, 2003.

[4] Baruch Awerbuch and David Peleg. Sparse partitions. In *FOCS*, volume 2, pages 503–513, 1990.

[5] Zuhal Can and Murat Demirbas. A survey on in-network querying and tracking services for wireless sensor networks. *Ad Hoc Networks*, 11(1):596 – 610, 2013.

[6] Wei-Peng Chen, J.C. Hou, and Lui Sha. Dynamic clustering for acoustic target tracking in wireless sensor networks. *IEEE Trans. Mob. Comput.*, 3(3):258–271, 2004.

[7] Murat Demirbas, Anish Arora, Tina Nolte, and Nancy A. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *OPODIS*, pages 299–315, 2004.

[8] Murat Demirbas and Hakan Ferhatosmanoglu. Peer-to-peer spatial queries in sensor networks. In *P2P*, pages 32–39, 2003.

[9] Karim Emara, Wolfgang Woerndl, and Johann H. Schlichter. Vehicle tracking using vehicular network beacons. In *WOWMOM*, pages 1–6, 2013.

[10] Roland Flury and Roger Wattenhofer. Mls: an efficient location service for mobile ad hoc networks. In *MobiHoc*, pages 226–237, 2006.

[11] Stefan Funke, Leonidas J. Guibas, An Nguyen, and Yusu Wang. Distance-sensitive information brokerage in sensor networks. In *DCOSS*, pages 234–251, 2006.

[12] Jie Gao, Leonidas Guibas, Nikola Milosavljevic, and Dengpan Zhou. Distributed resource management and matching in sensor networks. In *IPSN*, pages 97–108, 2009.

[13] Tran Hoang Hai, Faraz Khan, and Eui-Nam Huh. Hybrid intrusion detection system for wireless sensor networks. In *ICCSA - Volume Part II*, pages 383–396, 2007.

[14] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distrib. Comput.*, 20(3):195–208, 2007.

[15] Lujun Jia, Guolong Lin, Guevara Noubir, Rajmohan Rajaraman, and Ravi Sundaram. Universal approximations for tsp, steiner tree, and set cover. In *STOC*, pages 386–395, 2005.

[16] Oleg Kachirski and Ratan Guha. Effective intrusion detection using multiple sensors in wireless ad hoc networks. In *HICSS - Track 2 - Volume 2*, page 57.1, 2003.

[17] Vinodkrishnan Kulathumani, Anish Arora, Mukundan Sridharan, and Murat Demirbas. Trail: A distance-sensitive sensor network service for distributed object tracking. *ACM Trans. Sen. Netw.*, 5(2):15:1–15:40, 2009.

[18] H. T. Kung and D. Vlah. Efficient location tracking using sensor networks. In *WCNC*, volume 3, pages 1954–1961, 2003.

[19] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[20] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *MobiCom*, pages 120–130, 2000.

[21] Chih-Yu Lin, Wen-Chih Peng, and Yu-Chee Tseng. Efficient in-network moving object tracking in wireless sensor networks. *IEEE Trans. Mob. Comput.*, 5(8):1044–1056, 2006.

[22] Chih-Yu Lin, Yu-Chee Tseng, and Ten H. Lai. Message-efficient in-network location management in a multi-sink wireless sensor network. In *SUTC*, volume 1, pages 496–505, 2006.

[23] Bing-Hong Liu, Wei-Chieh Ke, Chin-Hsien Tsai, and Ming-Jer Tsai. Constructing a message-pruning tree with minimum cost for tracking moving objects in wireless sensor networks is np-complete and an enhanced data aggregation structure. *IEEE Trans. Comput.*, 57(6):849–863, 2008.

[24] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC*, pages 1–10, 1985.

[25] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, pages 88–97, 2002.

[26] Kirill Mechitov, Sameer Sundresh, Youngmin Kwon, and Gul Agha. Poster abstract: cooperative tracking with binary-detection sensor networks. In *SenSys*, pages 332–333, 2003.

[27] Arik Motskin, Ian Downes, Branislav Kusy, Omprakash Gnawali, and Leonidas J. Guibas. Network warehouses: Efficient information distribution to mobile users. In *INFOCOM*, pages 2069–2077, 2011.

[28] Rajmohan Rajaraman, Andreáa W. Richa, Berthold Vöcking, and Gayathri Vuppuluri. A data tracking scheme for general networks. In *SPAA*, pages 247–254, 2001.

[29] Johannes Schneider and Roger Wattenhofer. An optimal maximal independent set algorithm for bounded-independence graphs. *Distrib. Comput.*, 22(5-6):349–361, 2010.

[30] Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, pages 1–32, 2013.

[31] Gokarna Sharma and Costas Busch. Optimal nearest neighbor queries in sensor networks. In *ALGOSENSORS*, pages 260–277, 2013.

[32] Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. *Distrib. Comput.*, 27(5):329–362, 2014.

[33] Gokarna Sharma, Costas Busch, and Srivathsan Srinivasagopalan. Distributed transactional memory for general networks. In *IPDPS*, pages 1045–1056, 2012.

[34] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *SenSys*, pages 214–226, 2004.

[35] Julian Winter and Wang-Chien Lee. KPT: a dynamic knn query processing algorithm for location-aware sensor networks. In *DMSN*, pages 119–124, 2004.

[36] Yuxia Yao, Xueyan Tang, and Ee-Peng Lim. In-network processing of nearest neighbor queries for wireless sensor networks. In *DASFAA*, pages 35–49, 2006.

[37] Li-Hsing Yen, Bang Ye Wu, and Chia-Cheng Yang. Tree-based object tracking without mobility statistics in wireless sensor networks. *Wirel. Netw.*, 16(5):1263–1276, 2010.

[38] Wensheng Zhang and Guohong Cao. DCTC: Dynamic convoy tree-based collaboration for target tracking in sensor networks. *IEEE Transactions on Wireless Communications*, 3(5):1689–1701, 2004.

[39] Wensheng Zhang and Guohong Cao. Optimizing tree reconfiguration for mobile target tracking in sensor networks. In *INFOCOM*, volume 4, pages 2434–2445, 2004.

[40] Dengpan Zhou and Jie Gao. Maintaining approximate minimum steiner tree and k-center for mobile agents in a sensor network. In *INFOCOM*, pages 511–515, 2010.