

# CountTorrent: Ubiquitous Access to Query Aggregates in Dynamic and Mobile Sensor Networks

Abhinav Kamra  
Department of Computer Science  
Columbia University  
kamra@cs.columbia.edu

Vishal Misra  
Department of Computer Science  
Columbia University  
misra@cs.columbia.edu

Dan Rubenstein  
Department of Electrical  
Engineering  
Columbia University  
danr@ee.columbia.edu

We study the problem of aggregate querying over sensor networks where the network topology is continuously evolving. We develop scalable data aggregation techniques that remain efficient and accurate even as nodes move, join or leave the network. We present a novel distributed algorithm called CountTorrent, that enables fast estimation of certain classes of aggregate queries such as COUNT and SUM. CountTorrent does not require a static routing infrastructure, is easily implemented in a distributed setting, and can be used to inform all network nodes of the aggregate query result, instead of just the query initiator as is done in traditional query aggregation schemes. We evaluate its robustness and accuracy compared to previous aggregation approaches through simulations of dynamic and mobile sensor network environments and experiments on micaz motes. We show that in networks where the nodes are stationary, CountTorrent can provide 100% accurate aggregate results even in the presence of lossy links. In mobile sensor networks where the nodes constantly move and hence the network topology changes continuously, CountTorrent provides a close (within 10 – 20%) estimate of the accurate aggregate query value to all nodes in the network at all times.

## Categories and Subject Descriptors

C.2.2 [COMPUTER-COMMUNICATION NETWORKS]: Network Protocols; C.4 [PERFORMANCE OF SYSTEMS]: [Fault tolerance]

## General Terms

Algorithms, Design, Experimentation

## Keywords

Sensor networks, Aggregate querying, TinyOS, micaz

## 1 Introduction

Sensor nodes can measure specific details about their environment. Nonetheless, resource constraints often require

that these results be summarized en route to sink points that further process the data. One popular form of summarization is the *aggregate* [28,29,34,35]: a single or small set of values that summarize the distributed set measurements. In computing query aggregates instead of individual sensor data, the sensor network becomes a virtual (distributed) database over which SQL-type queries are initiated and distributed across the network [28]. A virtual spanning tree rooted at the query initiator node is formed and acts as a routing tree. Each sensor node then combines its own data with the results received from its child nodes and sends it to its parent, until eventually the desired aggregate reaches the query initiator node. This *in-network* aggregation technique is very effective and energy-efficient for distributive and algebraic aggregates [28] such as MIN, MAX, COUNT, and AVG.

However, the accuracy of the estimate computed via the tree-based approaches degrades rapidly in networks that are highly susceptible to node failures or rapid topology changes. For instance, a single node failure can result in a whole subtree of aggregate data being lost with the error increasing for failed nodes closer to the root. To overcome these shortcomings, solutions based on multi-path routing [5,28] improve estimate accuracy by minimizing the impact of isolated failures.

In mobile sensor networks, the devices may not have prior knowledge of the network topology which itself may change rapidly as nodes move. Additionally, the devices may run out of power or have intermittent connectivity which makes it impractical to aggregate even duplicate-insensitive aggregates, especially since it becomes difficult to maintain the same spanning tree topology for the duration of the query. Aggregation techniques [21,31], such as ones based on the Sketches [25] method provide approximate results in dynamic and faulty sensor networks but incur higher variance in the accuracy of results than the techniques proposed in this paper.

*In this paper, we propose CountTorrent, a robust and scalable mechanism to accurately estimate both duplicate-sensitive and duplicate-insensitive aggregate queries in dynamic and mobile sensor networks.* While our presentation here is geared toward sensor networks, the scheme is generic enough in nature that it can be applied to other environments that need a distributed computation of distributive queries such as COUNT, SUM, or AVG with a few adjustments. We define distributive queries [18] as follows:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
SenSys'07, November 6–9, 2007, Sydney, Australia.  
Copyright 2007 ACM 1-59593-763-6/07/0011 ...\$5.00

Let  $p_1$  and  $p_2$  be two disjoint populations of sensor nodes. Let  $f(p)$  be the aggregate query result over the sensor nodes belonging to population  $p$ . Then a query is distributive if  $f(p_1 \cup p_2) = f'(f(p_1), f(p_2))$ , i.e. if the aggregate over a population can be obtained as a function of the aggregates over two disjoint subsets of the population. One obvious aggregate query which is not distributive is MEDIAN.

CountTorrent is most easily described in two phases. The first phase is a distributed process that assigns unique labels to all nodes in the network. During the second phase, neighboring nodes in the network continually swap information consisting of labels and aggregates. This swapping process can be thought of as a *torrent* since it bears several resemblances to the swapping process in BitTorrent [17]: nodes exchange information with any neighbor that can use its information. Hence, information flows across all existing links in the network, moving simultaneously from every node to all its neighbors as long as the node has useful information to provide to the neighbor. As the information moves in these random directions through the network, it is aggregated, with the labeling scheme from the first phase ensuring that all values are counted once and only once within an aggregate as it moves and grows through the network. Since a node sends information to a neighbor only when it is useful for the neighbor (reminiscent of BitTorrent), CountTorrent avoids excessive bandwidth usage while still being robust to network failures and providing close-to-accurate query aggregates.

Since the aggregation process flows in all directions, it is fault-oblivious in the sense that the estimate is robust in the presence of most link and node failures. In the presence of link failures, CountTorrent computes an accurate query aggregate as long as the network remains connected. In the presence of node failures, CountTorrent’s computed aggregate contains values from all nodes that survive to initiate the second phase and whose value is contained in some aggregate that reaches other surviving nodes. Furthermore, in settings where the aggregate is to be revised continually, we show how the labeling can be adjusted to reflect changes in topology, such that failed nodes’ values become excluded from the aggregate, and late joining nodes’ values can be incorporated into the estimate.

Our contributions can be summarized as follows:

- We propose CountTorrent, a generic distributed query aggregation scheme suitable for faulty as well as mobile sensor networks. In contrast to many previously proposed aggregation methods, CountTorrent can make the aggregate results available at all nodes instead of just the query initiator.
- Through simulations and experiments we show that CountTorrent can compute 100% accurate query aggregates for stationary networks, even with lossy links, as long as the network remains connected. Furthermore, as nodes join and leave the network, the aggregate value can be updated so that it converges to the aggregate matching the current network configuration.
- We show that in a mobile sensor network where the nodes constantly move thereby continually changing

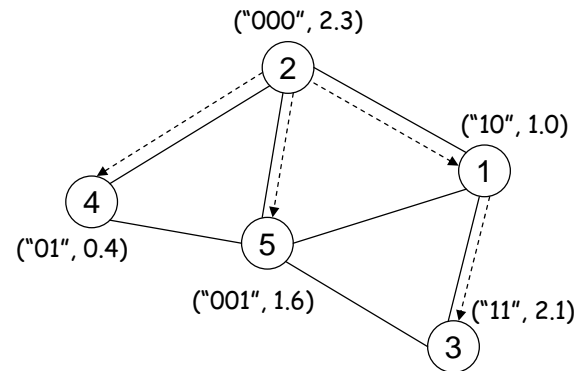
the network topology, CountTorrent can provide a query estimate within 10 – 20% of the accurate aggregate value to all nodes in the network at all times.

- We compare CountTorrent with a plain vanilla, tree-based aggregation scheme, TAG [28], and with an approximate aggregation scheme based on Sketches [21]. We show that CountTorrent can provide exact query aggregates even in networks with lossy links at the expense of slightly more bandwidth utilization whereas TAG and Sketches can have high variance in the computed aggregates.

The remainder of the paper is organized as follows: Sections 2, 3 and 4 describe the CountTorrent algorithm and propose heuristics to optimize it for dynamic and mobile networks. In Sections 5, 6 and 7 we present simulation and experiment results to demonstrate the efficacy of the protocol and compare it with traditional query aggregation techniques. Section 8 describes previous related work. We conclude in Section 9.

## 2 The CountTorrent Protocol

We propose two variants of CountTorrent: *Static CountTorrent* is used for one-shot query aggregation in an unreliable, evolving network. In the case that nodes are joining or leaving, we consider an estimate to be accurate as long as it includes readings from all nodes who were joined to the network prior to the initiation of the query, and were still part of the network when the query is completed, i.e., readings from late joiners and early leavers may or may not be contained. *Dynamic CountTorrent* is used to produce a continually updated estimate as the network evolves. The estimate at time  $t$  is considered accurate when it includes readings from nodes that are alive at time  $t$ . When a measured value changes at a node, the aggregate estimate at all nodes becomes inaccurate instantaneously and then steadily converges to its correct value. In Section 3 we describe these two variants in detail whereas in this section we only discuss the main idea behind CountTorrent common to both variants.



**Figure 1. A sample 5 node sensor network. The solid lines represent that the two nodes are in communication range and can exchange data. The tuples in parenthesis represent the node’s label and measured value respectively. A dashed arrow goes from a node which assigned the label for the node at the other end of the arrow.**

## 2.1 Assumptions and Definitions

For ease of explanation, in this and the following sections we describe the main idea behind CountTorrent assuming that a SUM aggregate is being computed. CountTorrent is trivially adapted to compute other query aggregates as well. Additionally, in this section we make several assumptions about the network within which CountTorrent is implemented. The network consists of  $N$  sensor nodes connected in an arbitrary topology -  $N$  need not be known by any of the nodes (in fact, CountTorrent's initial motivation was to COUNT the value of  $N$ ). Nodes are only aware of their immediate neighbors, to whom they can directly transmit. For now we assume that transmissions, even if broadcast within their local environment, are intended for a particular destination. Later in the paper, we show how CountTorrent can further leverage off of a broadcast medium. We also assume for now that this communication is bi-directional and reliable, and deal with the particulars of more realistic (lossy) transmission mediums later in the paper. Neighbors are said to be connected by an *edge* or *link*. Finally, we number each node with a unique value from 1 to  $N$ . These numbers are only to facilitate our description in the paper. CountTorrent's implementation does not use these numbers, and hence does not require their existence.

A sample network is depicted in Figure 1. The solid edges signify that the two nodes are within the communication range and hence can exchange data with each other. We will refer back to Figure 1 and explain it fully over the next few sections.

Each sensor node  $i$  is assumed to have what we call a *measured value*,  $v_i$ , which represents, for example, the reading that the sensor node has taken of its environment. The objective of SUM is to compute the sum of all measured values in the network, i.e.,  $\sum_{i=1}^N v_i$ . We assume that the measured values change slowly with respect to the time it takes to initiate and complete a query. If the measured value is expected to change rapidly, then there is little motivation for a snapshot reading (and hence little motivation for Static CountTorrent), although in certain cases, for example, if the measured values change in a monotonic fashion, it may be possible to update the aggregates as fast as the values change. While the accuracy of Dynamic CountTorrent's estimate due to rapid time-varying measured values is an interesting problem, it is beyond the scope of this paper.

The CountTorrent Protocol has two mechanisms that each node implements in two different phases. The first phase implements a distributed label assignment mechanism. The second phase implements a data forwarding and combining phase. While the first phase's mechanism is needed to implement the second, we feel that the core idea that distinguishes CountTorrent from previous work lies mostly in the second phase's mechanism, and hence we present this mechanism first.

## 2.2 Data Combining Mechanism

Let us assume that each sensor node  $i$  has a measured value of  $v_i$ . CountTorrent's first phase assigns a label,  $s_i$ , to each node,  $i$ , where the label is a binary string consisting of 0's and 1's, such as 00111010. Let  $\hat{\phi}(s)$  denote

the string  $s$  with its last character removed. For example,  $\hat{\phi}(0110) = 011$  and  $\hat{\phi}(101100) = 10110$ . Furthermore, let  $\phi_i(s)$  be the  $i$  length prefix of  $s$  and  $\phi(s)$  be the set of all (different length) prefixes of  $s$ . For example,  $\phi_3(011010) = 011$  and  $\phi(011010) = \{\epsilon, 0, 01, 011, 0110, 01101, 011010\}$ .

**Merging:** We say that two labels  $x$  and  $y$  can be *merged* if they have the same length  $\ell$  and differ only in their last bit (i.e.,  $x \neq y$  and  $\hat{\phi}(x) = \hat{\phi}(y)$ ). The result of a merge is the common prefix,  $\hat{\phi}(x)$ . For example, 01101 and 01100 can be merged to form 0110. On the other hand, 01101 and 01111 cannot be merged since  $\hat{\phi}(01101) \neq \hat{\phi}(01111)$ .

In the first phase, the labels are assigned to the  $N$  nodes such that the labels satisfy the following properties:

- For any 2 nodes,  $i$  and  $j$ ,  $s_i$  and  $s_j$  are neither the same nor is either one a substring of the other, i.e.  $s_i \notin \phi(s_j)$  and  $s_j \notin \phi(s_i)$ .
- All  $N$  labels can be merged pairwise and recursively to yield  $\epsilon$ , the empty string.

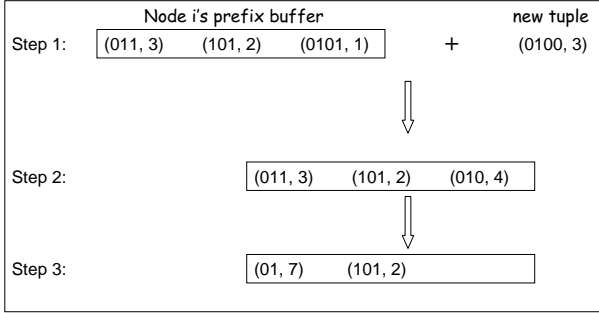
We emphasize that the labels assigned to the various nodes need not be of the same length.

Figure 1 shows a sample sensor network with the labels satisfying the above properties assigned to each node. Note that 001 and 000 merge to 00, which merges with 01 to form 0, while 10 and 11 merge to form 1, which can be merged with 0 to yield  $\epsilon$ . We now discuss how the specific properties of labels assigned to nodes are leveraged to obtain a query aggregate in a distributed manner.

Each node  $i$  is aware of its own label  $s_i$  and the measured value  $v_i$ . Furthermore, in the CountTorrent protocol, each node maintains a buffer we call the *prefix buffer* that stores label-value tuples. Initially node  $i$ 's prefix buffer contains only the tuple  $(s_i, v_i)$  as shown in Figure 1. Thereafter, nodes choose a random tuple from this buffer and send it to a neighbor. Whenever node  $i$  receives a new tuple, it stores and subsequently attempts to consolidate the tuples in its prefix buffer according to the following algorithm:

- When node  $i$  receives a tuple  $(s_{new}, q)$  from another node, this tuple is compared with the existing tuples in its local prefix buffer. Consider a tuple  $(s_{old}, p)$  already present in the prefix buffer:
  - If the binary strings  $s_{old}$  and  $s_{new}$  can be merged, then a new tuple  $(s_{combined}, p + q)$  is added to the node's prefix buffer, and the tuples  $(s_{old}, p)$  and  $(s_{new}, q)$  are discarded.
  - If  $s_{old} \in \phi(s_{new})$ , then  $(s_{new}, q)$  is discarded.
  - If  $s_{new} \in \phi(s_{old})$ , then  $(s_{old}, p)$  is discarded.
- The updated tuple (if any) is now recursively compared to other tuples in the prefix buffer until no two tuples can be merged any further. The consolidation process stops at this point.

To illustrate the above algorithm, consider node  $i$  has a prefix buffer containing three tuples as shown in Figure 2. Node  $i$  receives the tuple (0100, 3) and the consolidation process merges tuples in two steps as illustrated. When the consolidation process completes, two tuples remain in the prefix



**Figure 2. Label Consolidation at node  $i$  after node  $i$  receives the tuple (0100, 3).**

buffer.

It is worth noting that the tuple  $(s, v)$  is intended to represent the sum of values over all nodes where  $s$  is a prefix of the node's labels, i.e.,  $v = \sum_{i: s \in \phi(s_i)} v_i$ . Hence, it can easily be inferred that if all the  $N$  tuples with the initial labels and the respective measured values are available at a single node, the consolidation process will yield a single tuple  $(\epsilon, V)$  where  $\epsilon$  is the empty string and  $V = \sum_{i=1}^N v_i$ , is the desired SUM query aggregate.

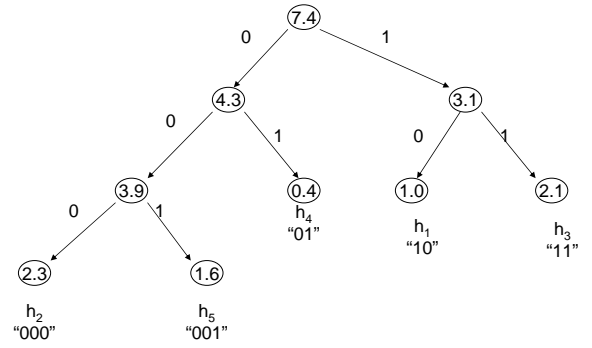
### 2.2.1 Data Forwarding Mechanism

In the above section we have described how a node incorporates an arriving tuple into its prefix buffer. Another important part of the protocol is the decision of how and when nodes exchange tuples. In CountTorrent, a node can receive tuples from any of its neighbors. In the context of the network shown in Figure 1, a node can receive tuples from any node connected to it via a solid line. It is this aspect of CountTorrent that makes the mechanism robust to mobility and to link and node failures. Rather than follow specific routes, tuples are passed to any neighbor who might be able to utilize it. Hence, tuples propagate through the network in a manner reminiscent of how file chunks are exchanged by nodes in the BitTorrent P2P file sharing protocol.

It is also worth noting that even though a tuple can only be merged with exactly one other tuple, the merging does not have to occur at either of the nodes which generated the tuples. Since there can be many copies of these tuples in the network, the merging can happen at some other node that subsequently received these two tuples and has them simultaneously stored in its prefix buffer.

Later, in Section 3.3.1, we discuss that the rate of convergence of the CountTorrent computation can be improved by having nodes select the destination of a tuple based on the labels of its neighbors. However, it is the ability to forward tuples along any link that greatly enhances the robustness of the protocol.

An added benefit of the forwarding and combining process is that all nodes participate in this process. This means that over time, the tuples stored in every network node's prefix buffer will contain shorter and shorter labels, and eventually converge to contain only the label,  $\epsilon$ . At this point, every node will have computed the aggregate.



**Figure 3. An example APT. Each of the 5 nodes is assigned a label corresponding to a leaf node. The SUM aggregate is computed by recursively adding up the tree in a distributed manner**

### 2.3 Label Assignment

The above process gives the idea that if the labels are chosen carefully and the tuples merged as described, an aggregate SUM value for the whole network can be computed in a completely distributed manner. Clearly, these binary labels can be arranged on a (not necessarily balanced) binary tree. For example, consider the binary tree depicted in Figure 3. To ensure that the labels assigned to the  $N$  sensor nodes satisfy the properties mentioned in Section 2.2, we need them to be chosen such that when arranged within a binary tree, the labels satisfy the following additional conditions:

1. For each node  $i$ , the corresponding label  $s_i$  corresponds to a leaf vertex in the binary tree.
2. For any 2 nodes,  $i$  and  $j$ ,  $s_i$  and  $s_j$  are neither the same nor is any one a descendant of the other in the tree.
3. Each leaf vertex in the tree is assigned to exactly one node, i.e. the number of leaf vertices in the tree is  $N$ , the same as the number of nodes in the network.

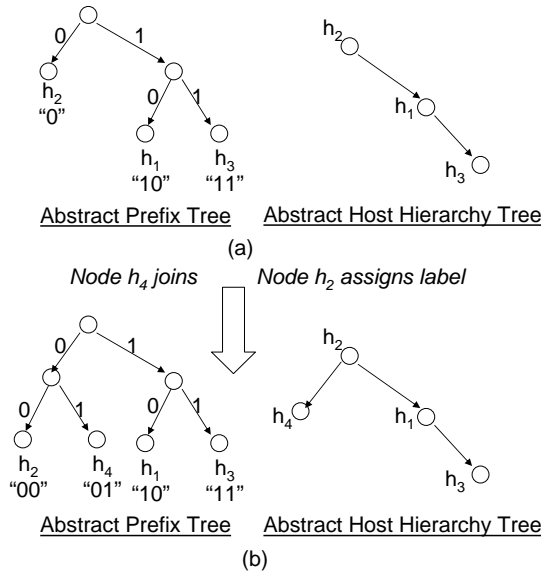
We refer to this tree view that is formed by the labels assigned to the nodes as the *Abstract Prefix Tree (APT)*. Figure 3 depicts the APT representation of the network presented in Figure 1. The values inside the circles are the aggregates of the measured values that correspond to prefix obtained by traversing to that circle from the top of the tree. The above properties ensure that the node labeling scheme provides labels to nodes such that all  $N$  nodes' tuples can be merged recursively to yield a single tuple  $(\epsilon, V)$  where  $V = \sum_{i=1}^N v_i$ , the desired query aggregate.

In a real network, prefix buffers may be finite, such that nodes must sometimes discard tuples. As long as this discard process is done in such a way that no label is "lost" from the network (i.e., if a tuple is dropped, then its label or a tuple containing the prefix of the label must exist elsewhere), and as long as tuples that do not merge continue to visit new nodes, the aggregation process is guaranteed to complete, eventually yielding the tuple  $(\epsilon, V)$  at all nodes.

Note that the sensor nodes need not know the total number of nodes in the network nor any other information about the specific topology of the network. The only information required is local, i.e. about the existence of neighboring nodes

and a way to exchange data amongst them.

**Label Assignment Algorithm:** Labels are assigned to sensor nodes in CountTorrent by a distributed process by having a labeled node communicate with a neighbor who has not yet been labeled as follows: Suppose that two nodes,  $i$  and  $j$  are neighbors, and that  $i$  has been assigned a label  $s_i$  of length  $\ell$ , while  $j$  has not yet been assigned a label. Via a communication exchange between  $i$  and  $j$ ,  $i$  “splits” its label into two different labels of length  $\ell + 1$  where the  $\ell$ -length prefix of these two labels is  $s_i$ .  $i$  reassigns itself one of these two  $\ell + 1$ -length labels, and assigns to  $j$  the other one. For instance, if  $i$ ’s label is initially 0110,  $i$  can reassign itself label 01100, and assign to  $j$  label 01101. We say that  $j$ ’s label is *derived* from  $i$ ’s label.



**Figure 4. An example abstract host hierarchy tree. A new node  $h_4$  joins the network and becomes a child of node  $h_2$ . Node  $h_2$  splits its labels and assigns a new one to node  $h_4$ .  $h_2$  is now the parent of node  $h_4$  in the hierarchy tree**

An alternate way to view the assigned labels as a tree is what we call the *Abstract Host Hierarchy Tree* or *AHHT* for short, which contains a vertex for each of the sensor nodes, and an edge from  $i$  to  $j$  indicates that  $j$ ’s label was derived from  $i$ ’s label. These visual abstractions of the labeling process will be useful in Section 3, where we discuss how Dynamic CountTorrent adapts the node labellings as nodes join, leave, or move within the network. Figure 4 shows how the APT and AHHT representations in the network get updated when a previously unlabeled node  $h_4$  is assigned a label by  $h_2$ .

It can be shown via induction that this assignment process maintains the desired structure of labellings when viewed as an APT. In particular, whenever a label is “split”, the resulting labels can always be merged to reform the label before splitting. Hence, recursively, there exists a sequence of merges across labels assigned to nodes that will result in the prefix  $\epsilon$ .

It is possible that a node  $j$  that has not yet been assigned a label has two or more neighbors with different labels. We note that  $j$  can choose to derive its label from any one of these neighbors. Any decision generates a labeling with the properties required by the second phase. Hence  $j$  could choose to split the label with the first neighbor who offers it a label, or it could wait for several neighbors to offer labels and choose the shortest label in an effort to more evenly balance the length of labels, and hence, the depth of the APT.

We can view all  $N$  sensor nodes to be arranged in a hierarchy tree to facilitate label assignment. Figure 1 shows the AHHT overlaid on a network of sensor nodes. The dashed arrows represent a parent-to-child relationship in the AHHT. Initially the root node is assigned a label of  $\epsilon$ . Any node can assign a label to any of its children nodes. For node  $i$  with a label of  $s_i$  to assign a label to node  $j$ , node  $i$  splits its own label  $s_i$  into  $s_i0$  and  $s_i1$  and assigns one of them to node  $j$ . In this way, all nodes are assigned labels such that they satisfy the properties of Section 2.2 and 2.3.

### 3 CountTorrent Variants

We propose two variants of CountTorrent. Static CountTorrent is suitable for quick one-shot query aggregation. It aims to provide the best possible estimate of the aggregate even if nodes move or fail during the process. On the other hand, Dynamic CountTorrent is meant for continuous query aggregation in the sense that as nodes move, leave or join, they update the data structures such that the query aggregate should reflect the current network configuration at all times. We now describe these two flavors of CountTorrent in detail.

#### 3.1 Initiating CountTorrent

One bootstrapping issue to be resolved is to determine who should initiate the CountTorrent process, i.e., which node should be assigned the label  $\epsilon$ ? The answer is that any node that wishes to compute an aggregate and does not observe an ongoing CountTorrent running can be the initiator. In fact, even if there is an existing CountTorrent in progress, nodes could initiate independent processes (where the root of the process can be at different or the same nodes) to generate multiple simultaneous estimates. All that is required is to convert the prior mentioned tuples stored in the prefix buffer to 3-tuples,  $(x, s, v)$ , where the additional component of the tuple,  $x$ , is a unique identifier assigned by the initiator of that instance of the CountTorrent to distinguish it from other instances. If nodes themselves have unique identifiers, then  $x$  could simply be that identifier (assuming each node initiates a single CountTorrent at a time). If not, it is sufficient to choose  $x$  at random from a large enough uniform distribution such that collisions are extremely unlikely. If  $b$  bits are used for these identifiers and a fraction  $f$  of the  $N$  nodes were to simultaneously initiate CountTorrents, no ID collisions occur at all with probability  $\prod_{i=1}^N (1 - i/2^b)$ . For instance, if  $f = 0.1$ ,  $N = 10^6$ , and  $b = 40$ , no ID collisions occur with probability greater than 0.98.

#### 3.2 Static CountTorrent

Static CountTorrent is used for snapshot query aggregation in sensor networks reminiscent of previous query aggregation approaches such as TAG [28]. In this setting, one of the sensor nodes acts as a query initiator, assigning itself the

empty label,  $\epsilon$  and initiates the first phase by contacting a neighbor, offering to split its  $\epsilon$  label. The remaining Labels are assigned to child nodes by parent nodes as the tree is being constructed as discussed in Section 2.3. Each node then creates a tuple  $(s_i, v_i)$  where  $s_i$  is its label and  $v_i$  the measured value. The second phase is the data combining phase as discussed in Section 2.2 where these tuples are merged in a distributed manner to get the final aggregate value.

Note that once a node and all of its immediate neighbors have been assigned labels, the node's label will no longer change. Hence, it can immediately enter the second phase and start attempting to merge tuples. No merging can take place until the node has a neighbor who is also in the second phase.

If there are no failures during the snapshot query aggregation, all the tuples can be merged to obtain a single tuple  $(\epsilon, V)$  where  $V$  is the accurate query aggregate value. In case of node or link failures during the aggregation process, CountTorrent's estimate is computed using values from all active nodes as well as values from a subset of failed nodes prior to their failure. This may be due to two reasons:

1. *Some nodes died before being assigned labels:* In this case, the remaining alive nodes can still merge the tuples in a distributed manner to get  $(\epsilon, V')$  but  $V'$  will be smaller than the sum of the measured values of the nodes alive at the start of the query.
2. *Some nodes died after being assigned labels and their labels were lost:* In this case, the nodes will not be able to merge all tuples into a tuple with an empty label. Each node will have a number of tuples which cannot be merged further. These could be merged into a single tuple if certain missing tuples were available. In such a situation the best estimate of the aggregate is obtained by assuming some default value for the missing tuples. For example, in case of a COUNT query, the missing tuples can be assumed to have a measured value of 1. This is done after a timeout period in which no tuples in a node's prefix buffer get merged. A final query estimate can thus be computed even if tuples cannot be merged further.

A combination of the above two kinds of failures can also be dealt with similarly. In either case, for most queries, Static CountTorrent computes (assuming non-negative measured values) an estimated aggregate whose value is less than or equal to the actual aggregate (for MAX, COUNT etc.) or always greater than or equal (for MIN) to the original accurate aggregate value. Hence, a simple scheme to obtain a better estimate at the cost of more bandwidth and storage at the nodes is to run multiple Static CountTorrent protocols simultaneously, preferably with different query initiator nodes. When the final estimates are obtained, the maximum (or minimum) of the estimates can be chosen as the best estimate.

### 3.3 Dynamic CountTorrent

Dynamic CountTorrent is suitable for sensor networks where nodes move, leave or join frequently. As the network topology changes, CountTorrent updates the labels for the alive nodes so that they satisfy the properties described in Section 2.2. In addition to a prefix buffer and keeping

track of one's parent and children, another data structure which we call the *tuple cache* is maintained at every node in this dynamic version. An estimate of the query aggregate is available at every node at all times and hence it can be acquired just by querying the nearest sensor node. As the network topology changes, Dynamic CountTorrent updates the labellings in the background to always have a good estimate of the aggregate available at all nodes. If the network topology eventually stabilizes, the estimate converges to the accurate aggregate value.

We now describe how Dynamic CountTorrent handles node joins, node leaves and mobile nodes.

#### 3.3.1 Node Join and Label Assignment

As described in Section 2.3, the sensor nodes are arranged in a virtual hierarchy by means of an AHHT. When a new node  $i$  joins the network, it finds a node  $j$  which is already part of the network and asks to be assigned a new label. Node  $j$  then becomes the parent of node  $i$  in the AHHT and assigns it a label. In this way the AHHT representation in the network is updated locally for a node join. Typically, node  $i$  picks  $j$  as the neighbor which can assign it the smallest length label. This simple heuristic helps keep the AHHT balanced. There can be certain unavoidable scenarios though, for example when nodes are placed in a collinear fashion and each node can communicate only with its immediate neighbors, that the labels can be  $O(n)$  in length and the AHHT is highly unbalanced.

When a new node joins the network, apart from assigning it a label, the query aggregate has to be updated to account for the measured value of the new node. In order to consistently propagate this information throughout the network and accurately update the data structures at each node, the tuple cache is used. This cache contains tuples associated with the node's children nodes in the AHHT as well as for its own label. For example, if node  $i$  has 3 child nodes, it maintains a cache containing 4 tuples, one each for the 3 child nodes and one for its own label. The tuples for the child nodes have labels which were initially assigned to the child nodes, even though the children nodes' labels may change later when they themselves assign labels to new nodes.

Using the cache at each node, the propagation of information when a new node joins works as follows:

1. When node  $i$  gets a new label  $s_i$  and creates the tuple  $(s_i, v_i)$ , it updates its cache and sends this tuple to its parent node.
2. When node  $j$ (parent) receives a tuple  $(s_i, v_i)$ , where  $s_i$  is the label associated with one of its child nodes, it checks its cache to see if the measured value for the label  $s_i$  as stored in its cache has changed. If so, the node updates the cache and deletes all tuples from its local prefix buffer which are in conflict with this new information (i.e. whose labels are substrings of  $s_i$ ). After updating its cache and the local prefix buffer, the node then computes the tuple associated with the label  $s_j^{orig}$ , the label assigned to it by its parent when node  $j$  joined the network. This can be easily computed by merging together the tuples present in its cache. This is so because even when node  $j$  becomes the parent of a new

node, it splits its label and the tuples associated with both the new labels are stored in its cache in lieu of the tuple containing node  $j$ 's previous label. Hence at any given time, the tuples in node  $j$ 's cache can be merged to obtain the tuple corresponding to the label originally assigned to node  $j$ . Node  $j$  now sends this tuple to its parent and this update process continues recursively.

### 3.3.2 Node Leaving the Network

When a node  $i$  leaves the network, the CountTorrent protocol again needs to adapt to this changed circumstance and adjust the network view of APT and AHHT in order to update the aggregate value and let it propagate to all nodes. There are two sets of nodes in the network which are immediately affected, the parent of  $i$  and the children of  $i$ . The parent of  $i$  simply removes  $i$  from its list of children and updates the cache and the local prefix buffer. This involves updating the cache by setting the value associated with node  $i$ 's label to 0. Update notifications are then sent recursively up the tree just as is done in case of a node joining the network. The label associated with the departed node is kept at the parent as part of a floating tuple and can be assigned when new nodes join the network.

The children of the departed node probe the remaining nodes in the neighborhood and ask for new labels to be assigned. A node  $j$  whose parent leaves the network has to find a new parent node and possibly get a new label. This new label  $s'_j$  then replaces the old label  $s_j$  in the cache and the prefix buffer. In fact, any labels in the cache or the prefix buffer which start with  $s_j$  get  $s'_j$  substituted for  $s_j$ . The labels of any nodes which are descendants of this node in the AHHT have also to be updated. Therefore, node  $j$  initiates a label substitution and asks its children to modify all labels which begin with  $s_j$  by substituting  $s'_j$  for  $s_j$ . This substitution is done recursively until all descendant nodes of  $j$  have the updated tuples.

### 3.3.3 CountTorrent for Mobile Networks

In a wireless mobile sensor network, nodes dynamically move in and out of range of other nodes. If there are no node failures and no new nodes joining the network, the labels assigned to the mobile nodes can remain the same and the distributed data combining protocol still gets the query aggregate by merging tuples. On the other hand, if there are node leaves and joins in the mobile network, the labels for the alive nodes need to be updated so that they satisfy the properties listed in Section 2.2. This is accomplished as follows. If node  $i$  is the parent of node  $j$  and nodes  $i$  and  $j$  move out of communication range:

- Node  $i$  assumes that node  $j$  left the network and updates its cache and prefix buffer accordingly.
- Node  $j$  on the other hand, assumes that the node  $i$  has left the network and finds another node as parent and updates its cache and prefix buffer accordingly.

In both cases, local data structures at the nodes are updated as was discussed in the previous subsections. In this way, CountTorrent can compute and continuously provide query aggregates in a mobile sensor network, albeit with a more frequent update of data structures at the sensor nodes.

## 4 CountTorrent Heuristics

In the CountTorrent protocol, individual nodes maintain a number of tuples along with their associated count values in a prefix buffer of limited size. Each node chooses a random neighbor and picks a random tuple from its buffer to send to this neighbor. For very large networks and when the buffer at each node is small, the consolidation of tuples can take a very long time. This is because with tuples flowing in random directions, the arrival of duplicate tuples and tuples that cannot immediately be merged occurs with greater frequency.

We propose two heuristics to expedite the CountTorrent consolidation process. The first heuristic called *Intelligent Selection* is a way for a node to avoid sending redundant tuples to a neighbor. The second heuristic which we call *Preferred Diffusion* allows for the CountTorrent protocol to converge very fast if there aren't any failures in the network and allows the convergence rate to degrade gracefully as failures occur.

### 4.1 Intelligent Selection

To limit transmission of duplicate tuples, a node can intelligently guess which tuples from its pool of tuples does a neighbor already have. This can be determined from the tuples sent and received from that neighbor in the past. Each node thus keeps track of a set of tuples that each of its neighbors is supposed to have. There are two simple rules to update this set:

- When node  $i$  receives a tuple containing label  $s$  from neighbor  $j$ , node  $i$  can deduce that in future,  $j$  is unlikely to require any tuple with label including  $s$  and all its descendants (any label starting with  $s$ ). So  $i$  can include all these tuples in the set which we call  $X_j^i$ .
- When node  $i$  sends a tuple with label  $s'$  to neighbor  $j$ ,  $i$  can again include  $s'$  and all its descendants in the set  $X_j^i$  since node  $j$  now already has the tuple with label  $s'$ .

The set  $X_j^i$  need not be maintained as a whole but can be efficiently stored by keeping track of only the labels which do not have any ancestors in the set, i.e. if label  $s$  is stored in the set, there is no need to store any labels of which  $s$  is a prefix. By maintaining this set, when node  $i$  needs to send a tuple to node  $j$ , it can check this set to determine which tuples it should not send. Node  $i$  then picks a random label  $s''$  from its buffer such that it is not contained in  $X_j^i$ . Of course, node  $j$  might already have received  $s''$  from another of its neighbors but this strategy minimizes the possibility of node  $i$  sending duplicate or redundant tuples to its neighbors.

Apart from maintaining a set representing a node's view of which tuples a neighbor might already have, the tuples themselves can be prioritized. Hence, tuples belonging to the subtree corresponding to a node's own label are sent earlier than other tuples. As far as storage is concerned, set  $X_j^i$  can be stored in the same manner as a prefix buffer since  $X_j^i$  also contains tuples which are merged the same way as in a prefix buffer. The unused memory at a node can be used for storing these sets. Even if the storage of  $X_j^i$  is limited, it does not affect the accuracy of the protocol but might decrease the convergence rate.

## 4.2 Preferred Diffusion

Using *Intelligent Selection*, a node can minimize the possibility of sending duplicate and redundant tuples to its neighbors. A node sends tuples to one of its random neighbors at time intervals according to a predefined distribution. When there are many failures in the network or when nodes are mobile, this randomization helps the CountTorrent protocol to converge despite the failures and every surviving node gets the final query aggregate. On the other hand, when there are negligible failures or when the nodes are relatively stationary, the randomization unnecessarily slows down the convergence. It is much faster to compute the aggregate along links forming the AHHT, as labels can be merged immediately. The best of both worlds would be to have a protocol which seamlessly moves from one that propagates information along the links of the AHHT (when available) and using the other links as “backups” to increase resilience to a completely distributed protocol as the network becomes more and more dynamic. This is the objective of the *Preferred Diffusion* heuristic.

We define  $parent(i)$  as the node which is the parent of node  $i$  in the AHHT. and  $children(i)$  as the set of nodes which have  $i$  as their parent. When node  $i$  has to send a tuple to one of its neighbors, it does so as follows:

1. If node  $j = parent(i)$  is not alive, goto step 2. Else select a tuple to send to  $j$  (possibly using *Intelligent Selection*). If there is such a tuple, send to  $j$  and stop. If no such tuple is there in the buffer, goto step 2.
2. Randomize the list  $children(i)$ . For each node  $j$  in  $children(i)$ , select a tuple to send to  $j$ . If there is such a tuple, send it to  $j$  and stop. If no such tuple is there for any of the nodes in  $children(i)$ , goto step 3.
3. Randomize the list of remaining neighbors. Again, for each node in the list, select a tuple to send. If there is such a tuple, send it to the corresponding node and stop.

The above algorithm orders the neighbors of a node such that the node always sends a tuple to its parent in the hierarchy tree if it has a useful tuple to send. If not, the node tries to send to one of its child nodes in the hierarchy tree and finally to the remaining neighbors.

## 4.3 How Intelligent Selection and Preferred Diffusion Work Together

In case there are no failures, *Preferred Diffusion* results in all nodes sending their tuples first to their parents so that the root node of the AHHT gets all the tuples and is able to quickly calculate the final count value which then percolates down to the children nodes as nodes start sending this final value to neighbors which are their children nodes in the hierarchy tree.

Another way to look at it is that *Preferred Diffusion* coupled with *Intelligent Selection* results in each node preferentially sending the tuples from its own subtree towards the root node. This results in the root node being able to compute the final aggregate value which is then sent back down the tree towards the rest of the nodes. Hence, the two heuristics coupled together result in all nodes acquiring the final query aggregate value in  $O(h)$  time where  $h$  is the height of

the AHHT. This is as good as previous spanning tree based aggregation techniques.

## 5 Experimental Evaluation of CountTorrent

We simulate CountTorrent in various sensor network scenarios and assess its performance in terms of efficiency, accuracy and cost (in terms of bandwidth utilized and time taken). We also compare CountTorrent with traditional query aggregation schemes in sensor networks using a homegrown C simulator and the TAG simulator of [28]. In Section 5.1 we describe our simulation setup and in Section 6 and 7 we present simulation and experimental results for various scenarios.

### 5.1 Simulation Setup

We compare CountTorrent with previous query aggregation schemes for sensor networks using our homegrown C simulator. The simulator can simulate various network topologies in a sensor network. By default, each sensor node has a fixed range and can communicate with any other node in that range, the time taken for a packet sent by a node to reach another node in its range is chosen uniformly randomly from the interval  $(\tau_{min}, \tau_{max})$ . For the simulations and experiments in the following sections, we use  $\tau_{min} = 0.9s$  and  $\tau_{max} = 1.1s$  unless otherwise stated.

We use the TAG simulator [28] modified in [21] for simulating previous methods of query aggregation in sensor networks. In all these strategies, each node has one or more parents which are closer to the root and each node aggregates results received from its children nodes and sends it to one or more of its parent nodes. Ultimately, the overall query aggregate can be computed at the root (query initiator node) of the spanning tree.

We simulate 3 previous approaches to query aggregation using the TAG simulator. **TAG1** is the plain vanilla aggregation strategy where each node sends its aggregate to a single parent. In **TAG2** on the other hand, each sensor node sends a fraction of its aggregate to each of its multiple parents such that the fractions sum to the whole aggregate. Any node within the communication range and which is closer to the root node is considered a parent for this purpose. **SKETCH** [21] uses the Sketches [25] approach to query aggregation.

In all simulations, we show the average over 100 runs. Furthermore, we use a fixed prefix buffer size of 10 at all nodes. We found that increasing the buffer size beyond this does not affect the performance significantly, although we do not show the results for that due to lack of space.

#### 5.1.1 Stationary nodes

For simulating stationary nodes, we position nodes randomly in an area. Each node can communicate with all others within a fixed distance of it and with packet delivery times chosen randomly from  $(\tau_{min}, \tau_{max})$ . Nodes leave and join the network randomly so for a particular sensor node, the number of nodes in range increases or decreases as more nodes join or leave respectively.

#### 5.1.2 Mobile nodes

For simulating mobile nodes, we use one of the most widely used mobility models, the Random Waypoint Border model [26]. In this model, the sensor nodes move in a rectangular area. There are multiple “waypoints” along the

border of the grid. Each node picks a waypoint and moves in a straight line towards it at a certain speed. We pick the speed randomly from  $v_{max}/2, v_{max}$ . Once a node reaches the waypoint, it randomly picks another waypoint and a new speed and moves towards the newly chosen waypoint. In this way the nodes move in and out of range of each other and hence the network topology changes continuously. Random Waypoint Border Model has certain well studied deficiencies, such as the initial-placement problem, but we think it is sufficient for the purposes of evaluating CountTorrent.

## 6 Experimental Results

In this section, we evaluate the performance of CountTorrent and compare it with previous approaches to query aggregation in sensor networks such as TAG [28] and SKETCH [21]. All the previous aggregation approaches focus on query aggregation at a particular snapshot in time with the accuracy of the results deteriorating if there are node or link losses in the network during the execution of the query. Static CountTorrent provides snapshot aggregation whereas Dynamic CountTorrent aims to provide ubiquitous access to query aggregates at any point in time with the aggregate updated continuously with minimal bandwidth as nodes move, leave or join the network.

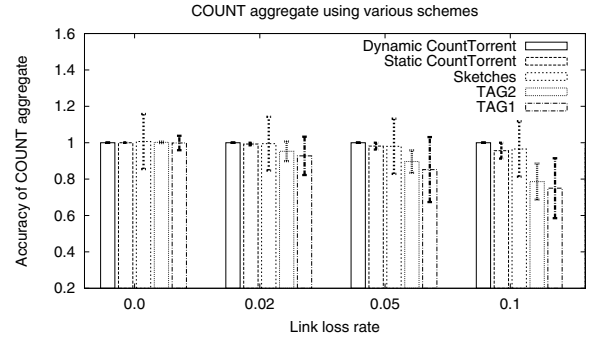
In mobile sensor networks, where sensor nodes move around and the network topology changes continuously, it is difficult to take a snapshot reading of the sensor network in the form of a query aggregate and hence the traditional approaches which use a spanning tree to get query results at an initiator sensor node are not practical. Dynamic CountTorrent avoids this problem by continuously updating the aggregate results over the network with minimal bandwidth use and hence can get a snapshot reading of the query aggregate at any point in time.

### 6.1 Comparison with Previous Aggregation Schemes

We compare CountTorrent with some popular query aggregation schemes for sensor networks. In a typical scheme, a query initiator node floods the network with the query and the result is received at the initiator node via in-network aggregation as is the case with Static CountTorrent as well. Dynamic CountTorrent is different in spirit because it aims to continuously update the query aggregate in the network as the network topology changes. Hence, Dynamic CountTorrent is not directly comparable with previous aggregation schemes in most scenarios although many previous schemes such as [28, 31] provide methods for periodic and event-driven continuous queries although they tend to employ approximation of aggregates to maintain robustness.

To compare Dynamic CountTorrent with snapshot schemes, we choose a query initiator node and measure how long it takes for Dynamic CountTorrent to provide the first aggregate result at this node. We compare it with Static CountTorrent as well as TAG1, TAG2 and SKETCH. This comparison is possible since in this particular case, Dynamic CountTorrent will resemble one-shot query aggregation because it will have to setup the data structures at every node from scratch.

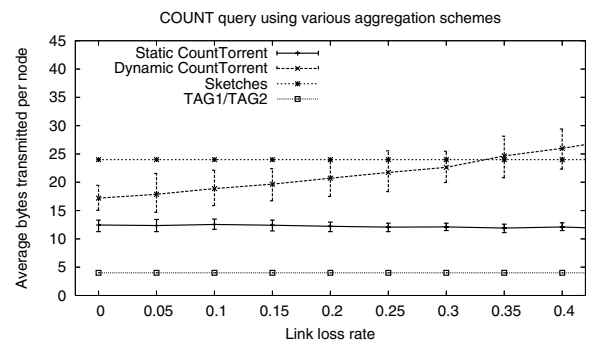
We assume a 100 node sensor network, with nodes placed



**Figure 5. Accuracy of COUNT query aggregation with increasing link failures**

randomly in a 100 x 100 area with each node’s communication range being 15. Hence, each node has about 7 neighbors on average. The links are assumed to be lossy and any packet sent on the link is lost with the particular link loss probability. In Figure 5 we show the accuracy of the different protocols with varying link loss probability in computing the COUNT aggregate. Each bar is an average of 100 simulation runs. A value of 1 suggests complete accuracy while a value of say 0.8 implies the mean computed aggregate is off 20% from the correct value. We also show 1 standard deviation error from the computed mean. Hence, the plot for SKETCH shows that there is high variance in the computed aggregates even though the mean aggregate is close to the correct value.

In a network with no lossy links, all schemes can compute almost accurate aggregates but as the link loss rate increases, TAG1 and TAG2 deteriorate rapidly while SKETCH and Static CountTorrent deteriorate slowly. Dynamic CountTorrent on the other hand is able to compute accurate COUNT aggregate values even with high link loss rates. This is essentially because in Dynamic CountTorrent, even when packets are lost, the sensor nodes continue sending packets until the CountTorrent data structures stabilize and hence every node gets the accurate aggregate value.



**Figure 6. Bandwidth usage of COUNT query aggregation with increasing link failures**

To achieve the accuracy even with high link loss rates, Dynamic CountTorrent needs to send many more packets than other aggregation schemes but each packet is just 6

bytes, consisting of the prefix, the aggregate value for that prefix and a timestamp with each being 2 bytes in size. A 2 byte prefix limits the network diameter to 16 although this can be easily increased while the CountTorrent structure still fits in a TinyOS packet. In TAG1 and TAG2, the packet size is just 2 bytes whereas for the SKETCH scheme, we use 20 bitmaps of size 16 bits each as suggested by the authors in [21]. The SKETCH packets are compressed using the compression techniques of [9] which result in approximately 70% compression. In Figure 6 we plot the number of bytes transmitted over the network for each of the aggregation schemes for the query initiator node to obtain the final aggregate value. TAG1, TAG2 and SKETCH have a fixed bandwidth consumption for a given topology. Dynamic CountTorrent on the other hand, needs to send more packets (including some acknowledgment packets) to obtain the accurate count as the link loss rate mounts to compensate for the lost packets. We also plot the 95% confidence intervals for the average bytes transmitted. In terms of number of packets, TAG1 and SKETCH send 1 packet per non-root node while TAG2 and CountTorrent need to send more packets per node.

For the following simulations and experiments, we use Dynamic CountTorrent instead of Static CountTorrent unless otherwise stated. In dynamic or mobile networks where nodes move, leave or join frequently, only Dynamic CountTorrent can be used.

## 6.2 Effect of Intelligent Selection and Preferred Diffusion

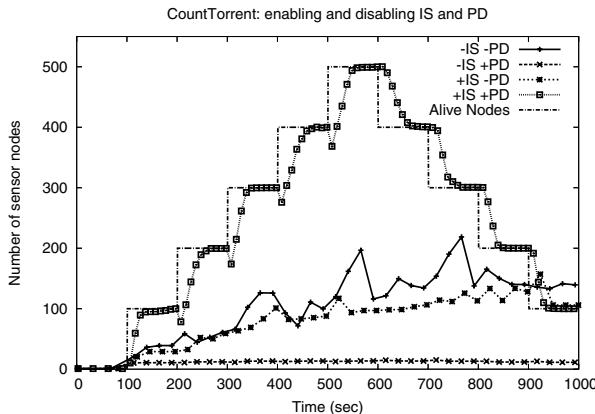


Figure 7. Effect of IS and PD on COUNT query aggregate

In Section 4, we described two heuristics to enhance the performance of CountTorrent. We now evaluate the efficacy of these heuristics when used separately as well as when used together. The simulation setup comprises of a maximum of 500 stationary sensor nodes placed randomly in a 100 x 100 grid. Each sensor node has a communication range of 15. The number of sensor nodes alive at any given time varies as shown in Figure 7. With CountTorrent being used to compute the COUNT aggregate query, we plot the aggregate value computed over time with or without the 2 heuristics. The + and - signs represent the presence or absence of

the corresponding heuristic respectively. For example, +IS -PD is the CountTorrent protocol which uses *Intelligent Selection* but not *Preferred Diffusion*.

As explained in Section 4.3, *Intelligent Selection* and *Preferred Diffusion* reinforce each other hence allowing CountTorrent to quickly update the aggregate metric when the network changes and we observe from Figure 7, the importance of the two heuristics enabled in tandem for CountTorrent to be efficient. We also observe that PD without IS performs very badly. This is because it is a broken protocol as every node will keep sending random prefixes to its parent and hence only the root node will have the correct estimate.

We also observe in Figure 7 that at each point in time when more sensor nodes become alive, the CountTorrent aggregate value suddenly drops before converging to the accurate value. This is because of the way we depict the aggregate value, as a mean of the aggregate estimates at each sensor node. Since the new set of nodes join with an initial aggregate value of 0, it suddenly lowers the mean aggregate value.

In the remaining experiments, we always use the CountTorrent protocol with both *Intelligent Selection* and *Preferred Diffusion* enabled by default.

## 6.3 Unicast or Broadcast

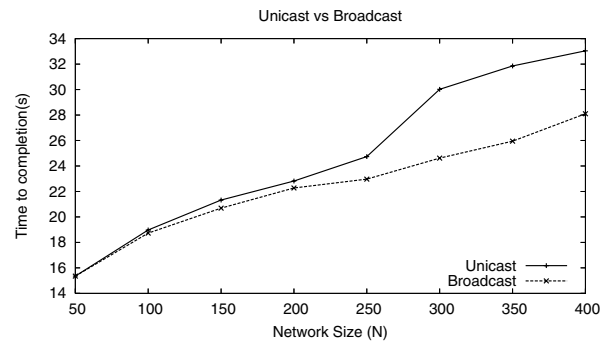
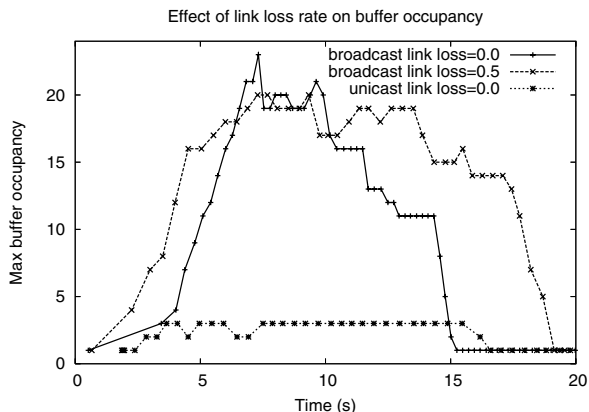


Figure 8. Taking advantage of broadcast in CountTorrent

CountTorrent assumes unicast data communication between a sensor node and one of its neighbors. But in a wireless setting CountTorrent can take advantage of the broadcast channel to send data to multiple nodes simultaneously. In Figure 8, we plot the time it takes for CountTorrent to get the accurate COUNT aggregate to all nodes in the network. We vary the number of nodes in the network but maintain the average degree of the nodes in the network at 4. When a node has to send a tuple, it chooses the data according to the 2 heuristics described in Section 4 and broadcasts instead of unicasting to a single neighbor and all nodes within its communication range receive the packet. It can be observed from Figure 8 that if available, CountTorrent can leverage the broadcast channel to expedite the query aggregation especially in case of large networks.

## 6.4 Prefix buffer occupancy at the nodes

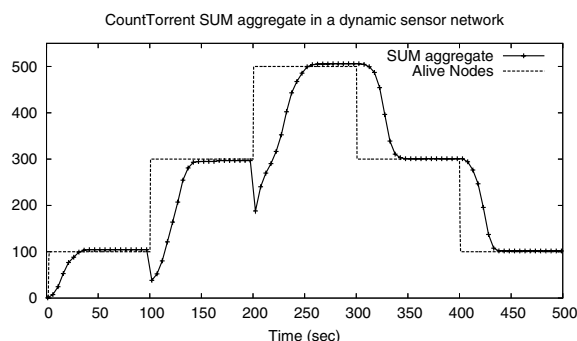
Each sensor node in the CountTorrent protocol has a prefix buffer to store tuples which cannot be combined at that



**Figure 9. Maximum of prefix buffer occupancy at all nodes**

time. In Figure 9, we show the maximum of the buffer utilization at all nodes in a 100 node network. With the *Preferred Diffusion* heuristic enabled, nodes prioritize sending tuples to their parents, followed by other neighbors. Hence in the case of unicast, the buffer occupancy is low since the tuples received from a node’s children are more likely to be merged with existing tuples in the buffer. For the unicast case with lossy links, we do not show the curve on the plot as the maximum buffer occupancy is similar to the non-lossy case. For broadcast with high link losses, nodes need to send tuples to many more nodes before the tuples all merge together and the aggregate obtained. This results in more buffer occupancy and for a longer time than when there are no link losses.

### 6.5 CountTorrent in Networks with Failures



**Figure 10. CountTorrent in Dynamic On-Off Networks**

We can observe in Figure 7 that as the number of alive nodes in the network changes, the CountTorrent COUNT aggregate protocol adapts to the changed scenario and updates the query aggregate at all nodes. This is especially true when both the heuristics *Intelligent Selection* and *Preferred Diffusion* are enabled. From now on, in all following experiments, we assume the heuristics are always enabled unless stated otherwise.

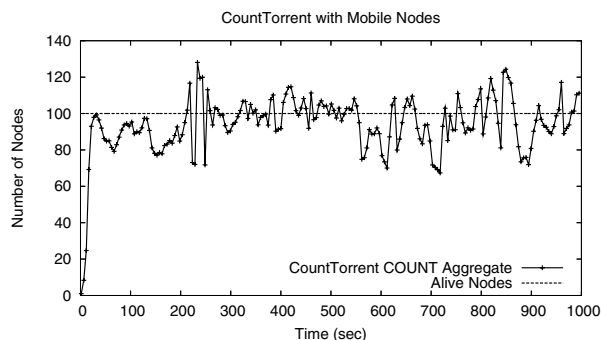
CountTorrent can be used for any distributive aggregate queries. We now show how CountTorrent SUM aggregate

adapts to a changing network scenario in a sensor network. The network consists of 100 sensor nodes and once again, we place nodes randomly in a 100 x 100 area with a communication range of 15. More and more sensor nodes join the network until the number is 500 and then nodes leave until the number is back to 100. At join time, each sensor node picks a random number between 0 and 2.0 as its measured value. Hence at any point, the SUM aggregate over the network will approximately be equal to the number of alive nodes in the network since the average data value is 1.0.

Figure 10 shows that as the number of nodes in the network change, hence changing the SUM aggregate value, CountTorrent adapts and the aggregate converges to the correct value quickly. Again, we observe a dip in the SUM aggregate when new nodes join. As already explained, this is due the way we depict the aggregate value, as a mean of the estimates at each sensor node. Since nodes join with an initial aggregate estimate of 0, it suddenly lowers the mean aggregate value.

Note that it takes about 40 seconds for the estimate to stabilize after a sudden change in the number of nodes. A simple calculation shows why this is so: Since the communication range is 15, the network diameter is approximately 10. Hence it can take up to 20 link traversals for new information to traverse up and down the network. Since an exchange of information between 2 neighbors takes 2 seconds (for tuple transmission and acknowledgment), it can take up to 40 seconds for the aggregate estimate to stabilize.

### 6.6 CountTorrent in Mobile Sensor Networks

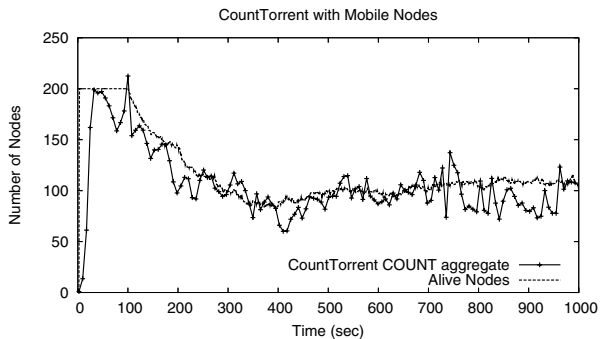


**Figure 11. COUNT aggregate with CountTorrent in a mobile sensor network**

In a mobile sensor network, with sensor nodes constantly moving around and node to node links breaking and forming in an ad-hoc manner, one-shot query aggregation will invariably give inaccurate results. CountTorrent aims to continuously adapt to the changing network topology and converge to the accurate aggregate value over the network. If the topology eventually stabilizes, CountTorrent aggregate value will quickly converge to the correct value.

We simulate a mobile sensor network using the Random Waypoint Border Model as explained in Section 5.1.2. There are 100 sensor nodes initially placed randomly in a 100 x 100 area with 20 border waypoints. The speed of the nodes is chosen randomly from  $v_{max}/2, v_{max}$  with  $v_{max}$  set to 5.0. The communication range is 15. Nodes move out of range of

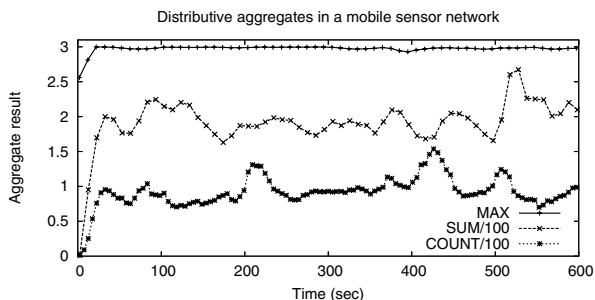
each other and in the range of other nodes on an ad-hoc basis as they move inside the area. Figure 11 shows the COUNT aggregate value as computed by CountTorrent over time. The aggregate value depicted is the average of the aggregate values estimated by each sensor node. After 100 seconds, the average absolute error in the estimate is about 10 with a standard deviation of about 7.



**Figure 12. COUNT aggregate with CountTorrent in a mobile sensor network with Node Arrivals and Departures**

In a mobile sensor where sensor nodes also have an arrival and departure process, it becomes even more complicated to keep track of the accurate query aggregate value. In this simulation we initially have 200 mobile nodes moving according to the RWPB model. At time  $t = 100$ , the arrival-departure process is triggered. The alive nodes depart at the average rate of  $u$  and the dead nodes rejoin the network at the rate of  $d$  where  $u = d = \frac{1}{N}/node/second$ . Hence in the stable state the number of alive nodes is approximately 100. In Figure 12, we observe that before  $t = 100$ , CountTorrent COUNT aggregate value oscillates around the value 200, the number of alive nodes. After  $t = 100$ , the number of alive nodes starts decreasing until it gets to around 100. After 100 seconds, the average absolute error in the estimate is about 14 with a standard deviation of about 10. CountTorrent adapts to this and we observe the aggregate COUNT value following the number of alive sensor nodes in the network.

## 6.7 Other distributive query aggregates



**Figure 13. Computing various distributive query aggregates with CountTorrent**

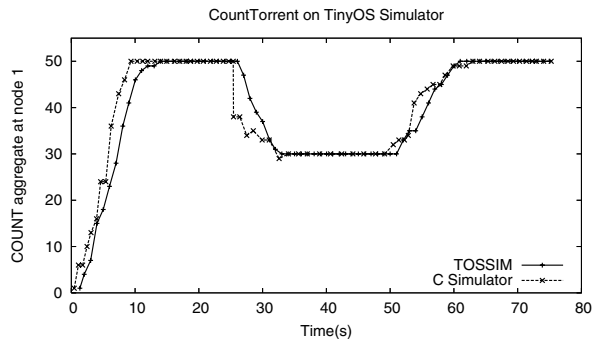
Apart from COUNT and SUM, CountTorrent can also compute other distributive query aggregates such as MAX,

MIN, AVG etc. In Figure 13, we illustrate how well CountTorrent can aggregate distributive queries in mobile sensor networks. There are 100 sensor nodes in a  $100 \times 100$  grid. The communication range of each node is 15 and  $v_{max}$  is set to 5.0. Each node has a measured value between 1.0 and 3.0 chosen uniformly randomly. We show the aggregates COUNT, SUM and MAX as computed using CountTorrent in this mobile network. For SUM and COUNT, we show the result value per sensor node. As can be observed from Figure 13, in a mobile network with a continuously changing topology, CountTorrent can provide a near accurate estimate for various distributive aggregates.

## 7 TinyOS Experiments

In this section, we demonstrate the practicality of CountTorrent through experiments on TinyOS [2] Simulator (TOSSIM). Later, we use micaz motes from Crossbow [1] to test the protocol in a real setting. Since TinyOS does not yet support a fully tested mobility model, we only simulate network scenarios with stationary nodes.

### 7.1 Experiments on TOSSIM



**Figure 14. Simulations on TinyOS Simulator**

We use TOSSIM to simulate 50 sensor nodes arranged in a  $5 \times 10$  grid and Dynamic CountTorrent to obtain a COUNT query aggregate. Each node can communicate with only the nodes directly up, down, left and right of it, if any. We number the nodes randomly with distinct ids chosen from 1 through 50. At time  $t = 0$ , all 50 nodes are alive. At time  $t = 25$ , nodes numbered 31 through 50 turn off reducing the network size to 30. At time  $t = 50$ , all the dead nodes come back up again and the network size is back to 50 again. We show the aggregate estimated at the node with the id 1. Figure 14 shows how the query aggregate computed with CountTorrent adapts with the changing network size. We use our C simulator for the same simulation to compare with the implementation on TOSSIM.

### 7.2 Experiments on micaz motes

Crossbow micaz motes are wireless sensor device prototypes which run TinyOS [2] and can be programmed in a high level language. We conduct an experiment on micaz motes to demonstrate that CountTorrent can estimate query aggregates accurately in dynamic networks. We arrange 15 micaz motes numbered 1 through 15 in a  $3 \times 5$  grid with the node at the center (node 8) acting as a base station via a Crossbow program controller and hence can be accessed

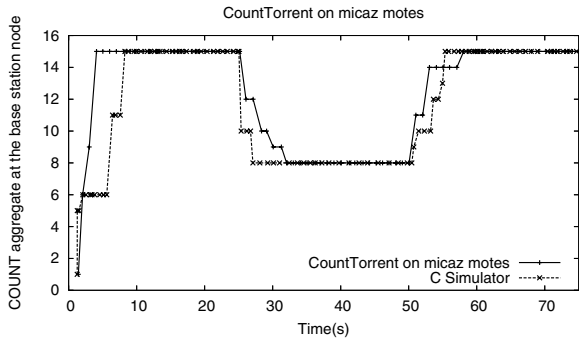


Figure 15. Experiments on Crossbow micaz motes

for data through a PC. As in the previous experiment, each node can only communicate with at most four nodes directly up, down, left and right of it. At time  $t = 0$ , all nodes are alive and start the Dynamic CountTorrent protocol. At time  $t = 25$ , nodes numbered 1 through 7 turn off hence reducing the network size to 8. At time  $t = 50$ , all the nodes come up again and the network size is 15 once again. Figure 15 shows the COUNT aggregate at the base station node as computed using CountTorrent with time. We also simulate the scenario on our C simulator to validate the CountTorrent performance on micaz motes.

## 8 Related Work

Previous approaches to distributed aggregate querying can be divided into two broad categories: those used for a broad range of distributive queries and those focused in particular on counting the number of nodes in the network.

### 8.1 General Aggregate Querying

In the realm of sensor networks, traditional querying approaches [6] compute aggregates in two phases: the *distribution* phase and the *collection* phase via *in-network* aggregation. Similar approaches have been used in many other systems [8, 33, 35]. However, in-network aggregation along a tree is susceptible to node and transmission failures, which are common in sensor networks [20, 28, 33]. Because each of these failures loses an entire subtree of readings, a large fraction of the readings are typically unaccounted for which can introduce significant errors in the query results [16, 28]. In-network query aggregation schemes have the advantage of being fast and low overhead but are vulnerable to network failures. CountTorrent on the other hand, can provide accurate aggregates in lieu of some extra bandwidth even in failure-prone networks.

Various works have proposed multipath based alternatives to the tree-based approaches for query aggregation. For example, TAG [28] proposes dividing the partial aggregates and sending them toward the root via multiple paths. This solves to some extent the problem for losing multiple readings when a node or link fails but can result in large transmission overheads and overcounting. Nath et. al. [31] propose techniques to decouple aggregation from message routing so that the aggregates are not affected by the order and the number of times partial results are received from the child nodes. They also propose an Adaptive Rings technique for aggrega-

tion in dynamically changing topologies. Manjhi et. al. [5] propose an adaptive scheme which uses both tree-based and multipath aggregation mechanisms simultaneously in different regions of the networks depending on the network conditions. The scheme is robust due to its adaptive nature although the aggregates may not be exact in highly dynamic conditions.

Many techniques [8, 10, 14] try to maintain a robust routing layer so that any aggregation protocol can utilize it to route successfully to the root node. There are overheads with maintaining routing in the face of node and link failures and they can suffer from the same problems of overcounting if the aggregation protocol is not carefully designed. Prefix routing schemes [7, 13] on the other hand, maintain a routing structure over the network by assigning hierarchical prefixes to nodes and at the same time provide a natural spanning tree structure which can be used for in-network aggregation.

Chen et. al. [19] propose computing local aggregates by overlapping subsets of neighboring nodes to arrive at a global consensus for an aggregate value. This works well for most duplicate-insensitive queries such as MIN, MAX and some duplicate-sensitive queries such as AVG (although the convergence time can be large) but not as well for COUNT and SUM. Kempe et. al. [12] use gossip-style communication to compute aggregates. Such approaches provide a high degree of robustness but suffer from high communication costs.

It should be noted that almost all the previous aggregation approaches rely on a root node to initiate the query and collect the results. This introduces a central point of failure as well as making it highly probable that any node or link failure in the midst of a query will disrupt the ongoing query aggregation. In CountTorrent, there is no unique query initiator node, the query aggregate is available at all times at every node and in case of topology change/disruption, the aggregate is updated at neighboring nodes which then diffuses to all other nodes.

### 8.2 Node Counting

Various studies have focused solely on counting the number of nodes in a mobile ad-hoc or sensor network, or in other words, computing the COUNT aggregate.

The problem of estimating the session size, i.e., estimating the number of nodes in a multicast network is well studied. The naive approach is to flood the multicast tree with requests and have each node respond directly to the root. This results in the so called “feedback-implosion” problem which results in too many acknowledgment packets vying to get to the root and congesting all the paths towards that node. Various studies have tried to address this problem primarily by restricting the number of responses so that each node replies with a certain probability [3, 32]. The total number can be estimated by the number of replies and the associated probability. This can be accomplished in multiple rounds by starting with a very low reply probability and slowly increasing it. An alternative approach [22] does the estimation in a single step by setting timers at each node for the reply and canceling the timer after hearing another node sending the reply. For estimating the sizes of multicast trees in the Internet, many works [11, 30] propose using models of the underlying topology. All these schemes provide an approxi-

mate value for the COUNT query. CountTorrent on the other hand can always provide an accurate query aggregate at all nodes, network conditions permitting.

A completely different way is to use linear [23] or logarithmic counting [16, 21] techniques also called *Sketches* method. This involves using an array of bits and each node using multiple hash functions to mark one of the bits. The number of bits marked gives an estimate of  $\log(N)$  in case of logarithmic counting or of  $N$  in case of linear counting. Other frequency counting techniques based on sketches [15, 27, 36] have also been proposed. Unlike CountTorrent, these approaches provide only approximate aggregate results with a large variance, though it is possible to reduce the error and variance by doing multiple Sketches simultaneously and taking the average.

Counting in mobile ad-hoc networks has been studied by Hatzis et. al. [4] using a flooding based technique which takes care of overcounting provided no nodes leave or join during the query process. Malpani et. al. [24] describe a number of token circulation based algorithms for distinct node counting but assume that the topology of the network is known at any given time. Token based counting approaches are susceptible to lost tokens in case of node and link failures and hence are not suitable for sensor networks prone to such failures.

## 9 Conclusions

We present CountTorrent, a novel approach for fast estimation of distributive query aggregates in sensor networks. We propose two flavors of CountTorrent, a Static version suitable for one-shot query aggregation and a Dynamic version more suited to query aggregation in networks where nodes move, leave and join frequently. We show that in networks with stationary nodes, CountTorrent's main advantage is its *robustness*: in comparison to previous approaches, its estimates remain up to 30% more accurate in networks with lossy links. In networks where the topology varies due to node movement, joins, and leaves, CountTorrent can provide estimates within 10 – 20% of the accurate value. We demonstrate the practicality of CountTorrent through simulations on TinyOS simulator and through experiments on Crossbow micaz motes.

## 10 References

- [1] Crossbow micaz motes. <http://www.xbow.com/Products/productsdetails.aspx?sid=105>.
- [2] TinyOS Homepage. <http://www.tinyos.net/>.
- [3] Jean-Chrysostome Bolot, Thierry Turletti, Ian Wakeman. Scalable Feedback Control for Multicast Video Distribution in the Internet. In *ACM Computer Communication Review*, volume 24, pages 58–67, 1994.
- [4] K. Hatzis, G. Pentaris, P. Spirakis and B. Tampakas. Counting in Mobile Networks: Theory and Experimentation. In *International Workshop on Algorithm Engineering*, 1999.
- [5] Amit Manjhi, Suman Nath and Phillip B. Gibbons. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. In *Proceedings of SIGMOD*, 2005.
- [6] Bhaskar Krishnamachari, Deborah Estrin and Stephen Wicker. The Impact of Data Aggregation in Wireless Sensor Networks. In *International Workshop on Distributed Event-Based Systems*, 2002.
- [7] Brian Levine and J. J. Garcia-Luna-Aceves. Improving Internet Multicast with Routing Labels. In *International Conference on Network Protocols*, 1997.
- [8] C. Intanagonwivat, R. Govindan and D. Estrin. Directed diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *ACM Conference on Mobile Computing and Networking*, 2000.
- [9] C. Palmer, P. Gibbons and C. Faloutsos. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. In *ACM Conference on Knowledge Discovery and Data Mining*, 2002.
- [10] D. Ganesan, R. Govindan, S. Shenker and D. Estrin. Highly-Resilient, Energy-Efficient Multipath Routing in Wireless Sensor Networks. In *ACM Mobile Computing and Communications Review*, volume 5, 2001.
- [11] Danny Dolev, Osnat Mokryn, Yuval Shavitt. On Multicast Trees: Structure and Size Estimation. In *Proceedings of INFOCOM*, 2003.
- [12] David Kempe, Alin Dobra and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Symposium on Foundations of Computer Science*, 2003.
- [13] E. Bakker, J. Leeuwen and R. Tan. Prefix Routing Schemes in Dynamic Networks. In *Computer Networks and ISDN Systems*, volume 26, 1993.
- [14] F. Ye, G. Zhong, S. Lu and L. Zhang. GRADient Broadcast: A Robust Data Delivery Protocol for Large Scale Sensor Networks. In *ACM Wireless Networks (WINET)*, volume 11, 2005.
- [15] G. Cormode and S. Muthukrishnan. Improved data stream summaries: The count-min sketch and its applications. In *Journal of Algorithms*, 2004.
- [16] George Kollios, John Byers, Jeffrey Considine, Marios Hadjieleftheriou and Feifei Li. Robust Aggregation in Sensor Networks. In *IEEE Data Engineering Bulletin*, volume 28.
- [17] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, H. J. Sips. The Bittorrent P2P File-sharing System: Measurements and Analysis. In *International Workshop on Peer-to-Peer Systems*, February 2005.
- [18] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals. In *Data Mining and Knowledge Discovery*, volume 1, 1997.
- [19] J. Y. Chen, G. Pandurangan, D. Xu. Robust Computation of Aggregates in Wireless Sensor Networks: Distributed Randomized Algorithms and Analysis. In *IEEE Transactions on Parallel and Distributed Systems*, volume 17, 2006.
- [20] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems*, 2003.
- [21] Jeffrey Considine, Feifei Li, George Kollios and John Byers. Approximate Aggregation Techniques for Sensor Databases. In *International Conference on Data Engineering*, 2004.
- [22] Jorg Nonnenmacher, Ernst W. Biersack. Optimal Multicast Feedback. In *Proceedings of INFOCOM*, 1998.
- [23] B. Vander-Zanden K. Whang and H. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications.

- In *ACM Transactions on Database Systems*, volume 15, pages 208–229, 1990.
- [24] Navneet Malpani, Yu Chen, Jennifer L. Welch and Nitin H. Vaidya. Distributed Token Circulation in Mobile Ad Hoc Networks. In *IEEE Transactions on Mobile Computing*, 2005.
  - [25] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Database Applications. In *Journal of Computer and System Sciences*, volume 31, 1985.
  - [26] Pasi Lassila, Esa Hyytia and Jorma Virtamo. Spatial Node Distribution of the Random Waypoint Mobility Model with Applications. In *IEEE Transactions on Mobile Computing*, volume 5, page 680.
  - [27] S. Ganguly, M. Garofalakis and R. Rastogi. Processing set expressions over continuous update streams. In *Proceedings of SIGMOD*, 2003.
  - [28] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad hoc sensor networks. In *ACM/USENIX Symposium on Operating Systems Design and Implementation*, 2002.
  - [29] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of SIGMOD*, 2003.
  - [30] Sara Alouf, Eitan Altman, Philippe Nain. Optimal On-Line Estimation of the Size of a Dynamic Multicast Group. In *IEEE Infocom*, 2002.
  - [31] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *ACM Conference on Embedded Networked Sensor Systems*, 2004.
  - [32] Timur Friedman and Don Towsley. Multicast Session Membership Size Estimation. Technical Report TR 98-32, 1998.
  - [33] Y. J. Zhao, R. Govindan, D. Estrin. Residual Energy Scans for Monitoring Wireless Sensor Networks. In *IEEE Wireless Communications and Networking Conference (WCNC 02)*, 2002.
  - [34] Y. J. Zhao, R. Govindan, D. Estrin. Computing Aggregates for Monitoring Wireless Sensor Networks. In *International Workshop on Sensor Network Protocols and Applications*, 2003.
  - [35] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. In *ACM SIGMOD Record*, volume 31, 2002.
  - [36] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of RANDOM*, 2002.

