

TAG: Tiny AGgregate Queries in Ad-Hoc Sensor Networks

Samuel Madden, Michael J. Franklin and Joseph Hellerstein
 {madden, franklin, jmh}@cs.berkeley.edu

UC Berkeley

Wei Hong
 whong@intel-research.net

Intel Berkeley Research Lab

Abstract

We present the Tiny AGgregation (TAG) approach to distributing declarative SQL-style aggregate queries with grouping over networks of low-power, wireless sensors. We discuss various generic properties of aggregates, and show how those properties affect the performance of our in-network approach. We include a performance study demonstrating the advantages of our approach over traditional centralized, out-of-network methods, and discuss a variety of optimizations for improving the performance and fault-tolerance of the basic solution.

1 Introduction

Recent advances in computing technology have led to the production of a new class of computing device: the wireless, battery powered, smart sensor. Traditional sensors deployed throughout buildings, labs, and equipment are passive devices that simply modulate a voltage based on some environmental parameter. In contrast, these new sensors are active, full fledged computers, capable not only of sampling real world phenomena but also filtering, sharing, and combining those sensor readings.

At UC Berkeley, researchers have developed small sensor devices called *motes*, and an operating system that is especially suited to running on them called TinyOS[8]. Motes are equipped with a radio, a processor, and a suite of sensors. TinyOS makes it possible to deploy *ad-hoc* networks of sensors. Ad-hoc and traditional networks differ in that motes in an ad-hoc network can locate each other and route data without any prior knowledge or assumptions about the network topology. It is expected that networks consisting of thousands of such motes will soon be widely deployed.

Database researchers have, thus far, played only a small role in the scientific community that has rapidly developed around the deployment of hardware, operating systems,

and networking software for sensor networks. This paper takes a step towards rectifying this situation by presenting the Tiny AGgregation (TAG) approach for distributing and executing aggregate queries in sensor networks. Aggregate queries are particularly interesting in the context of sensor networks because when thousands of sensors are spread throughout a space, users rarely want to query each of them individually; rather, they often want to know coarse features, such as averages, in different parts of that space.

Aggregation is also important because sensor networks are lossy: individual sensors turn on and off or move, messages are lost, and neighbors and parents change. Aggregation provides a way to cope with such loss: by looking at large scale behavior, sporadic failures become less significant. Furthermore, our algorithms are designed specifically to provide good quality and performance despite loss. Another reason to aggregate is that sensor networks are limited in shared bandwidth at the edge of the network. In many cases, this external bandwidth is a small fraction of the aggregate internal bandwidth. Thus, computing aggregates in-network improves performance and conserves power. To better understand why in-network aggregation is important, we consider a real world example.

1.1 Motivating Example

We are in the process of deploying a several-hundred node ad-hoc network of motes at the Intel lab on the 13th floor of the PowerBar building in downtown Berkeley. This network will be shared by all of the occupants of the lab, both as a testbed for networking and operating systems projects and as a distributed sensing environment. This network is dynamic; researchers add and remove motes, and sensors frequently run out of power, lose their antennas, or simply crash. Lab scientists want to extract a variety of information from this environment, and the most natural interface for such extraction is a simple query language. We anticipate a number of different queries of varying complexity over different subsets of the sensors. Example queries include: 1) *What is the temperature near my workstation?*; or, 2) *What is the quietest area in the building?*; or 3) *Which of the conference rooms are currently in use?*

All three of these examples are aggregate queries; in Query 1, we need to locate the sensors near the user and then compute an average temperature over them. In Query 2, we must find a minimum volume over all sensors in the building. Query 3 requires us to average light or volume readings over the sensors in each conference room and then

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 28th VLDB Conference,
 Hong Kong, China, 2002**

select those aggregates above some threshold.

One way to compute such aggregates would be to bring all sensor readings to the base station and perform the queries centrally; unfortunately, as we will discuss, a centralized scheme has a prohibitive cost in terms of the number of messages sensors must send. Instead, we propose an in-network aggregation scheme that uses an order of magnitude fewer messages and provides significantly better tolerance to network errors than the naive, centralized approach. This benefit is provided transparently to the end user, who still issues queries centrally in a simple, SQL-like language; we describe the basic architecture of this query environment in the next section.

1.2 Overview of the Paper

In this paper, we include a detailed description of the TAG approach, complete with a number of important performance enhancements. We include a taxonomy of aggregates supported by TAG, and discuss how different classes of aggregates vary in performance and tolerance to failure. We also present experimental results demonstrating the effectiveness and robustness of our algorithms in a simulation environment. Because this is an extremely broad topic area, we constrain our discussion in this paper by only focusing on an environment with a single query running at a time. We do not explicitly model sensors whose values change over the life of the query, although we discuss how our algorithms adapt to change. We also assume all sensors are identical; heterogeneous sensor environments will present different challenges and optimization tradeoffs.

In the rest of this paper, we discuss the requirements of an in-network aggregation system, followed by background in sensor networks and the TinyOS operating system. We then show how our solution satisfies the query requirements while providing performance and tolerance to network faults (e.g., dropped messages). We discuss several optimizations for improving the performance of the basic approach. Finally, we discuss related work and conclude.

2 Query Environment

In this section, we discuss the properties of sensor environments that make executing aggregate queries over them difficult. We begin with a discussion of our query model.

2.1 Query Model

In the TAG query architecture users pose queries from a powered, storage-rich base station, which distributes queries over the sensor network. This base station may be a desktop PC, a mobile palmtop computer, or a more powerful intelligent sensor device. We support SQL-style queries over a single virtual table (without joins) called `SENSORS`, whose schema is known at the base station. As is the case in Cougar [14], this table can be thought of as an append-only relational table with one attribute per input of the motes (e.g., temperature, light.) In TAG, we choose not to collect or store individual sensor readings; instead, we only provide aggregate values, storing only the intermediate state required to compute those aggregates. The energy cost of extracting all data over all time from all sensors is so great that, even if it were desired, it would rapidly deplete sensor batteries and render the network useless. We assume all sensors have the same attributes (e.g. temperature or light),

although sensors with different physical inputs can easily be supported by including all known attributes in the `SENSORS` table and excluding sensors without certain attributes from queries requiring those attributes. Thus, queries in TAG have the form:

```
SELECT {aggn(attrn), attrs} FROM sensors
WHERE {selPreds}
GROUP BY {attrs}
HAVING {havingPreds}
EPOCH DURATION i
```

Selection predicates can simply be executed locally at the sensor before readings are communicated, as in [14, 12]; given their simplicity, we will not discuss them further. We reserve a discussion of grouping for Section 6. We will discuss the `EPOCH DURATION` clause below.

As in parallel and object relational databases [2, 22], agg_n is implemented via three functions: a merging function f , an initializer i , and an evaluator, e . In general, f has the following structure:

$$\langle z \rangle = f(\langle x \rangle, \langle y \rangle)$$

where $\langle x \rangle$ and $\langle y \rangle$ are multi-valued *partial state records*, computed over one or more sensor values, representing the intermediate state over those values that will be required to compute an aggregate. $\langle z \rangle$ is the partial-state record resulting from the application of function f to $\langle x \rangle$ and $\langle y \rangle$. For example, if f is the merging function for `AVERAGE`, each partial state record will consist of a pair of values: `SUM` and `COUNT`, and f is specified as follows, given two state records $\langle S_1, C_1 \rangle$ and $\langle S_2, C_2 \rangle$:

$$f(\langle S_1, C_1 \rangle, \langle S_2, C_2 \rangle) = \langle S_1 + S_2, C_1 + C_2 \rangle$$

The initializer i is needed to specify how to instantiate a state record for a single sensor value; for the case of `AVERAGE` with a sensor value of x , the initializer $i_{AVERAGE}(x)$ returns the tuple $\langle x, 1 \rangle$. Finally, the evaluator e takes a partial state record z and computes the actual value of the aggregate. For `AVERAGE`, the evaluator $e(\langle S, C \rangle)$ simply returns S/C .

These three functions can easily be derived for the basic SQL aggregates. Given this structure of queries and aggregates in TAG, we now discuss the specific classes of aggregates that are important in this work.

2.2 Taxonomy of Aggregates

We classify aggregates according to four attributes that are particularly important to sensor networks: the amount of intermediate state required to compute them, their tolerance to loss, their tolerance to duplicates, and their monotonicity. We begin by considering the amount of state required for each partial state record. Observe that the amount of state required varies with the choice of aggregation function f ; a partial `AVERAGE` record consists of a pair of values, while a partial `COUNT` record constitutes only a single value. Similarly, a partial `MEDIAN` record consists of the entire set of values seen so far. Though the TAG approach correctly computes any aggregate that conforms to the specification of f in Section 2.1 above, its performance scales with the amount of intermediate state required per aggregate. For this reason, it is important to understand the amount of state required by various aggregates; we present a simple taxonomy of aggregates according to their state

	MAX, MIN	COUNT, SUM	AVERAGE	MEDIAN	COUNT DISTINCT	HISTOGRAM ¹	Section
Partial State	Distributive	Distributive	Algebraic	Holistic	Unique	Content-Sensitive	Section 4.3
Duplicate Sensitive	No	Yes	Yes	Yes	No	Yes	Section 7.4
Exemplary (E), Summary (S)	E	S	S	E	S	S	Section 5.2
Monotonic	Yes	Yes	No	No	Yes	No	Section 6

Table 1: Classes of aggregates

requirements below. The first three classifications were initially presented in the original work on data-cubes [5]; the remaining entries are new. As in [5], we are as general as possible in our taxonomy, covering not only the traditional SQL aggregates, but also any user-defined aggregation functions that might arise.

- In *Distributive* aggregates, such as MIN and COUNT, the partial state is simply the aggregate for the partition of data over which they are computed. Hence the size of the partial state records is the same as the size of the final aggregate.
- In *Algebraic* aggregates, such as AVERAGE and VARIANCE, the partial state records are not themselves aggregates for the partitions, but are of constant size.
- In *Holistic* aggregates, such as MEDIAN, the partial state records are proportional in size to the set of data in the partition. In essence, in holistic aggregates no useful partial aggregation can be done, and all the data must be brought together to be aggregated by the evaluator.
- *Unique* aggregates, such as COUNT DISTINCT, are similar to holistic aggregates, except that the amount of state that must be propagated is proportional to the number of distinct values in the partition.
- In *Content-Sensitive* aggregates, the partial state records are proportional in size to some (perhaps statistical) property of the data values in the partition.

The final category represents an important class of aggregates popular in the database literature in recent years: the approximate aggregate. An interesting question for future work is to focus on approximation techniques that minimize intermediate state record size, perhaps through lossy compression or statistical modeling schemes.

The next property of aggregates in our taxonomy has to do with whether the aggregate returns actual sensor values or summaries of many values: *Exemplary* aggregates return one or more representative values over all values; MAX, MIN, and TOP-N are examples. *Summary* aggregates, such as COUNT and AVERAGE, compute some property over all values. This distinction is important because exemplary aggregates behave unpredictably in the face of loss, and, for the same reason, are not amenable to sampling. Consider MIN: if it is applied to only a subset of all values, either because some values are lost or because the set is sampled, nothing can be said about the relationship between the true minimum and the minimum over the subset. Conversely, for summary aggregates, it is usually the case that the aggregate applied to a subset is a bounded approximation of the true aggregate value.

A third partitioning of aggregates has to do with their sensitivity to duplicates: consider a MAX over a set of sensor readings. If one of those readings is reported more than

once, those additional reports will not affect the value of the aggregate. Thus, *Duplicate Insensitive* aggregates, such as MAX, MIN, and variants of DISTINCT are unaffected by duplicate readings from a single sensor while *Duplicate Sensitive* aggregates, which encompass most other functions, will change when a duplicate reading is reported.

The fourth and final classification has to do with whether an aggregate is *monotonic* or *non-monotonic*. Monotonic aggregates include COUNT, MAX, MIN, and SUM (if values are either all greater than or all less than zero.) They have the property that when two partial state records, s_1 and s_2 , are combined via f , the resulting state record s' will have the property that either $\forall s_1, s_2, e(s') \geq \max(e(s_1), e(s_2))$ or $\forall s_1, s_2, e(s') \leq \min(e(s_1), e(s_2))$.

In summary, we have classified aggregates according to their state requirements, their tolerance of loss, and duplicate sensitivity, and their monotonicity. Table 1 relates these properties to specific instances of aggregates and lists the Sections of the paper where the various dimensions of the classification are important. We will refer back to this classification throughout the text, so that specific examples we use can be related to general classes of functions beyond the basic five aggregates supported by SQL. We now discuss the semantics of queries in TAG.

2.3 Query Semantics

Sensors can operate in either a push or pull regime: when pushing, aggregates are computed periodically and when pulling, data is sampled only when the user requests it. Our architecture is amenable to both scenarios; push-based aggregates appear to the user as a stream of values, and are useful when users wish to pose, for example, continuous queries to monitor the status of the network. Pull-based queries, on the other hand, produce a single value or table of values, and are useful when the user seeks a snapshot of the network at a point in time. We focus on streaming, push-based queries in this paper, although we will show how snapshot queries can be implemented as a special case of streaming queries.

When dealing with push-based streaming aggregates, we assume SEQ [15] style semantics. This means that all sensor readings are time-stamped and that readings in any particular aggregate all belong to the same time interval, or *epoch*. The duration of each epoch is the argument of the EPOCH DURATION clause, which specifies the amount of time (in seconds) sensors wait before acquiring and transmitting each successive sample. This value may be as large as the user desires; it must be at least as long as the time it takes for a sensor to process and transmit a single radio message – between 10 and 20 ms for current generation sensors. In general, the data delivered at the root of the sensor network is not from the current epoch but has an age, in epochs, proportional to the number of radio hops

¹The HISTOGRAM aggregate sorts sensor readings into fixed-width buckets and returns the size of each bucket; it is content-sensitive because the number of buckets varies depending on how widely spaced sensor readings are.

from the base station to the most distant sensor; we will see why this is the case in Section 4 below.

Finally, it is important to note that query results in sensor networks are never a perfect portrayal of the actual state of the system. High loss rates mean that some sensor readings will never be reported. Even in the absence of loss, however, the fact that sensors are digitizing noisy analog signals means that the data entering the network is never a perfect representation of the real world – loss is inherent in the sensor network domain.

3 Hardware and Networking Issues

The problem of computing aggregates in large clusters of nodes has been addressed previously in the context of shared-nothing query processing environments [16]. Those environments, although they are not streaming, utilize a similar query structure and decomposition of aggregate functions as discussed above. However, beyond similarities in query structure, there are a number of substantial differences between parallel clusters and sensor networks:

- *Network Topology*: Clusters typically consist of fully connected peers, whereas sensor networks are hierarchically structured relative to some root node, with each node having a small number of neighbors.
- *Dynamic Membership*: Nodes in a sensor network are not fixed – they can move, break, or run out of battery power. Traditional parallel systems are not built to handle a dynamically changing infrastructure to converge towards correct aggregation of available values despite a changing infrastructure.
- *Resources*: Finally, unlike parallel workstations, motes have very limited storage, processors, and network bandwidth, requiring special measures to limit the amount of state stored, instructions executed, and bytes transmitted on any single sensor.

Given these special requirements of ad-hoc sensor networks, a new architecture for executing aggregate queries over them is clearly needed. Before presenting our solution, we first examine the architecture of motes and the basic algorithms for routing data between them.

3.1 Motes and Ad-Hoc Networks

An example of the current generation of motes is shown in Figure 1. These devices are equipped with a 4Mhz Atmel microprocessor with 4 kB of RAM and 32 kB of code space, a 917 MHz RFM radio running at 50 kb/s, and 128kB of EEPROM. An expansion slot accommodates a variety of sensor boards by exposing a number of analog input lines as well as popular chip-to-chip serial busses. Current sensor options include: light, temperature, magnetic field, acceleration (and vibration), sound, and power.

The single-channel radio is half duplex, meaning motes cannot send and receive at the same time. Currently, the default TinyOS implementation uses a CSMA-like media access protocol with a random backoff scheme. Message delivery is unreliable by default, though applications can build up an acknowledgment layer. Often, a message acknowledgment can be obtained for free (see below in Section 3.2).

Power is supplied via a free-hanging AA battery pack or a coin-cell attached through the expansion slot. The effective lifetime of the sensor is determined by this power supply. In turn, the power consumption of each sensor node is dominated by the cost of transmitting and receiving messages. In terms of power consumption, transmitting a single bit of data is equivalent to 800 instructions[8]. Put another way, a AA battery pack will allow a sensor to send 5.52 million messages (if it does no other computation) which is equivalent to one message per second every day for about two months – not particularly long if the goal is to deploy long lived, zero-maintenance ad-hoc sensor networks. Hence, power-conserving algorithms are particularly important, if only for administrative purposes. In the next section we present details of the messaging and networking aspects of TinyOS and wireless sensors.

To understand how data is routed in our ad-hoc aggregation network, two properties of radio communication need to be emphasized. First, radio is a broadcast medium, such that any sensor within hearing distance hears any message, irrespective of whether or not that sensor is the intended recipient. Second, radio links are typically symmetric: if sensor a can hear sensor b , we assume b can hear a .²

Messages in the current generation of TinyOS are a fixed size – by default, 30 bytes. Each message has a two-byte *message type* that distinguishes it from other varieties of messages. Sensor programmers write type-specific handlers that are invoked by TinyOS when a message of the appropriate type is heard on the radio. Each sensor has a unique *sensor ID* that distinguishes it from others. All messages specify their recipient (or specify *broadcast*, meaning all available recipients), allowing sensors to ignore messages not intended for them, although non-broadcast messages are received by all sensors within range – unintended recipients simply drop messages not addressed to them.

3.2 Ad-Hoc Routing Algorithm

Given this brief primer on wireless sensor communication, we now show how sensors route data. The technique we adopt is to build a routing tree. We appoint one sensor to be the *root*. The root is the point from which the routing tree will be built, and upon which aggregated data will converge. Thus, the root is typically the sensor that interfaces the querying user to the rest of the network. The root broadcasts a message asking sensors to organize into a routing tree; in that message it specifies its own id and its *level*, or distance from the root (in this case, zero.) Any sensor

²Note that this may not be a valid assumption in some cases. However, most ad-hoc routing algorithms in the networking literature also make this assumption [20, 9].

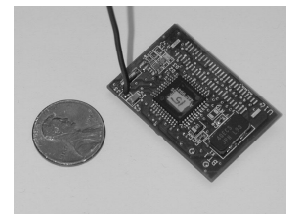


Figure 1: A TinyOS Sensor Mote

without an assigned level that hears this message assigns its own level to be the level in the message plus one. It also chooses the sender of the message as its *parent*, through which it will route messages to the root. Each of these sensors then rebroadcasts the routing message, inserting their own ids and levels. The routing message floods down the tree in this fashion, with each node rebroadcasting the message until all nodes have been assigned a level and a parent. These routing messages are periodically broadcast from the root, so that the process of topology discovery goes on continuously. This constant topology maintenance makes it relatively easy to adapt to network changes caused by mobility of certain nodes, or to the addition or deletion of sensors. To maintain stability in the network, parents are retained unless the child does not hear from them for some long period of time, at which point it selects a new parent using this same process. We will look in more detail at the robustness of this approach with respect to loss and its effect on aggregate values in Section 7.

When a sensor wishes to send a message to the root, it broadcasts a message addressed to its parent, which in turn forwards the message on to its parent, and so on, eventually reaching the root. We will see in the next section how, as data is routed towards the root, it can be combined with data from other sensors to efficiently combine routing and aggregation. The approach we present combines aggregation with the routing and topology maintenance algorithm described above.

4 In-Network Aggregates

A naive implementation of sensor network aggregation would be to use a centralized, *server-based* approach where all sensor readings are sent to the base station, which then computes the aggregates. In TAG, however, we compute aggregates in-network whenever possible, because, if properly implemented, this approach can be lower in both latency and power than the server-based approach.

4.1 Pipelined Aggregation

Given the algorithm for building a tree-based network described in Section 3.2, we now present the *pipelined* approach for computing in-network streaming aggregates. Pipelined aggregation consists of two phases: a *flooding* phase, in which the routing tree is built and aggregate queries are pushed down into sensor networks, and an *aggregation* phase, where the aggregate values are continually routed up from children to parents. As discussed in Section 2.1 above, aggregate queries consist of an aggregation function, a grouping predicate, a having clause, and an epoch duration.³

We divide time into epochs of duration i , as specified in the query. Aggregation proceeds as follows: when a sensor p receives a request to aggregate, r , either from another sensor or from the user, it awakens, synchronizes its clock according to timing information in the message, and prepares to participate in aggregation. At the same time, it chooses its parent according to the tree-building approach

given above. Rather than flooding the message immediately down the tree, however, p waits until the next epoch. It then rebroadcasts r to include any nodes that did not hear the previous round, and include them as children (if it has any.) These nodes continue to forward the request in the following epoch. r floods down the tree in this manner, until a leaf is encountered. A node determines it is a leaf when it does not hear a child forward r in the epoch after it transmitted it; upon this determination, it sends a freshly initialized partial state record a to its parent. In the subsequent epoch, e , each parent will combine a with any other state record it received from its children during epoch $e - 1$ and forward the resulting record on to the next parent in the tree. At the same time, the leaves will forward new partial state records to their parents, to be combined and propagated during epoch $e + 1$. Thus, aggregates flow back up the tree epoch-by-epoch. During the $2d - 1$ st epoch, where d is the depth of tree, the first complete aggregate arrives at the root. Note that if we were to terminate the aggregation at this point, we would have computed a single snapshot query over the network. When the query is allowed to run continuously, however, a new aggregate arrives in each successive epoch. Figure 2 illustrates a simple aggregate running in this pipelined fashion over a small sensor network.

The observant reader will have noticed a flaw with this scheme: it allows values from different epochs to be combined together. For example, in Figure 2, the data from sensor 5 reaching sensor 1 in time interval 9 was produced during time interval 6. Combining data from two epochs may not necessarily lead to erroneous behavior, particularly if the rate of change of sensor values is substantially less than the time for data to propagate up the tree, but it does violate our query model laid out above. We can rectify this problem by keeping historical values and only combining data from the same epoch. Notice that parents may have to keep historical values for their children as well – in Figure 2, sensor 3 will have to store sensor 4’s partial state records from the previous epoch for combination with values coming from sensor 5 and 6 during the current epoch. Thus, as an upper bound, each node must store a history up to the depth of the tree for itself and all of its children. Once this modification is made, the reported aggregates at the top of the tree will all be d epochs older than the current time.

We ran a number of simulation experiments to measure the additional state required in this approach, although space precludes including them in detail. These experiments showed that the maximum number of partial state records stored at any node in a 50x50 network of 2500 nodes never exceeds 70 and is about 15 on average – both of these sizes will easily fit into the RAM of a current generation TinyOS mote. Section 4.2 provides more details about our simulation environment.

One of the attractive properties of the pipelined approach is its ability to tolerate loss. In sensor environments, it is very likely that some aggregation requests or partial state records will be garbled, or that sensors will move or run out of power. These losses will invariably result in some sensors becoming *lost*, either without a parent or not

³We assume the aggregation function is drawn from a set of pre-loaded aggregation functions as TinyOS motes do not currently support dynamic loading of code modules.

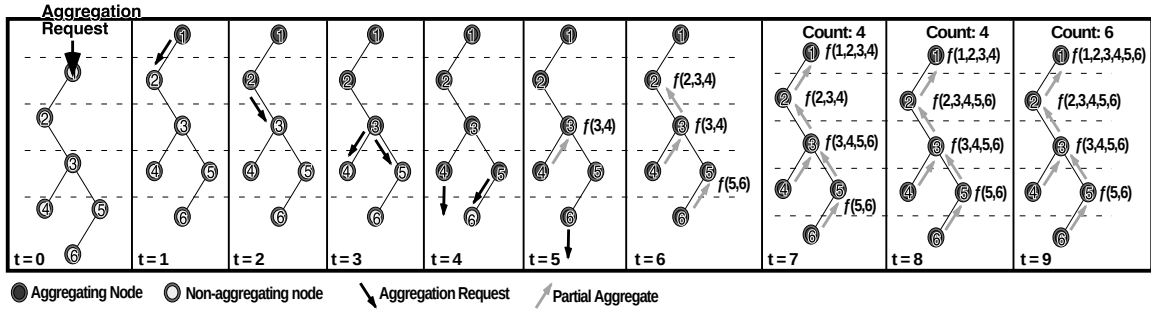


Figure 2: Pipelined computation of aggregates

incorporated into the aggregation network during the initial flooding phase. If we include information about queries in partial state records, lost nodes can reconnect by listening to other sensor’s state records – not necessarily intended for them – as they flow up the tree. We revisit the issue of loss at the end of the paper, in Section 7.

Another benefit of the pipelined approach is that, after aggregates have propagated up from leaves (the “pipeline has filled”), a new aggregate arrives every epoch versus the complete time it takes for a message to propagate down and back up the tree. Assuming that each state record fits into a single radio message, it is easy to see that the cost of each additional aggregate, in terms of total messages sent, is n , where n is the number of sensors. The initial cost to fill the pipeline (or compute a single snapshot query) is $2n$. When computing holistic aggregates, such as MEDIAN, more messages will be required as the size of the partial state grows.

These properties provide users with a stream of aggregate values that changes as sensor readings and the underlying network change. As discussed above, such continuous results are often more useful than a single, isolated aggregate, as they allow users to understand how the network is behaving over time.

4.2 Modeling the TAG Environment

To study the algorithms presented in this paper, we simulated TAG in Java. The simulator models sensor behavior at a coarse level: time is divided into units of epochs, messages are encapsulated into Java objects that are passed directly into sensors without any model of the time to send or decode. Sensors are allowed to compute or transmit arbitrarily within a single epoch, and each sensor executes serially, although messages sent by all sensors during one epoch are delivered in random order during the next epoch to model a parallel execution.

Our simulation includes an interchangeable communication model that defines connectivity based on geographic distance. Figure 3 shows screenshots of a visualization component of our simulation; each square represents a single sensor, and shading (in these images) represents the number of radio hops the sensor is from the root (center); darker is closer. We have run experiments with three communications models; 1) a *simple* model, where sensors have perfect (lossless) communication with their immediate neighbors, which are regularly placed (Figure 3(a)), 2) a *random* placement model (Figure 3(b)), and 3) a *realistic* model that attempts to capture the actual behavior of the

radio on TinyOS motes (Figure 3(c).). In the latter model, notice that the number of hops from a particular node to the root is no longer directly proportional to the distance between the node and the root, although the two values are still related. This model uses results from real world experiments [3] to approximate the actual loss characteristics of the TinyOS radio. Loss rates are high in the realistic model: a pair of adjacent sensors loses more than 20% of the traffic between them. Sensors separated by larger distances lose still more traffic.

The simulator also models the costs of topology maintenance: if a sensor does not transmit a reading for several epochs (which will be the case in some of our optimizations below), that sensor must periodically send a *heartbeat* to advertise that it is still alive, so that its parents and children know to keep routing data through it. The interval between heartbeats can be chosen arbitrarily; choosing a longer interval means fewer messages must be sent, but requires sensors to wait longer before deciding that a parent or child has disconnected, making the network less adaptable to rapid change.

This simulation allows us to measure the the number of bytes, messages, and partial state records sent over the radio by each mote. Since we do not simulate the mote CPU, it does not give us an accurate measurement of the number of instructions executed in each mote. It does, however, allow us to obtain an approximate measure the amount of state required for various algorithms, based on the size of the data structures allocated by each mote.

Sensor readings in our simulation can be drawn from one of several distributions; except for content-sensitive aggregates, the choice of distribution does not effect the performance of our algorithms, so unless otherwise specified, readings are selected randomly from the uniform distribution. Also, unless otherwise specified, our experiments are over the simple radio topology in which there is no loss. We also assume sensor values do not change over the course of a single simulation run.

4.3 Performance of the Pipelined Approach

In the first set of experiments, we compare the performance of the pipelined and centralized approaches on queries for the different classes of aggregates discussed in Section 2.2. Centralized aggregates have the same communications cost irrespective of the aggregate function, since all data must be routed to the root. For this experiment, we compared this cost to the number of bytes required for distributive aggregates (MAX and COUNT), an algebraic aggregate (AVER-

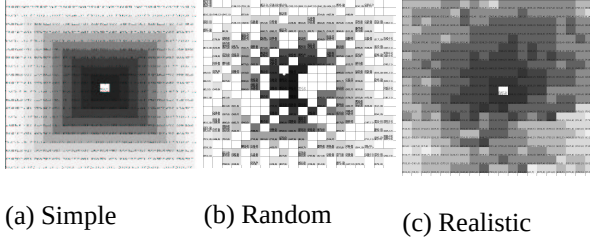


Figure 3: *The TAG Simulator, with Three Different Communications Models, Diameter = 20.*

AGE), a holistic aggregate (MEDIAN), a content-sensitive aggregate (HISTOGRAM), and a unique aggregate (COUNT DISTINCT); the results are shown in Figure 4.

Values for in this experiment represent the steady-state cost to extract an additional aggregate from the network once the pipeline is full; the cost to flood a request down the tree and then fill the pipeline is twice the steady-state cost. For the HISTOGRAM aggregate, we set the size of the fixed-width buckets to be 10; sensor values range over the interval [0..1000]. MAX and COUNT have the same cost in-network, since they both send just a single integer per partial state record; similarly AVERAGE requires just two integers, and thus always has double the cost of the distributive aggregates. MEDIAN costs the same as a centralized aggregate, which is significantly more expensive than other aggregates, especially for larger networks, as parents have to forward all of their children's values to the root. The COUNT DISTINCT is only slightly less expensive, as there are few duplicate sensor values; a smaller sensor-value interval would reduce the cost of this aggregate. Finally, the HISTOGRAM is an efficient technique for extracting a density summary of readings from the network.

In this experiment, in-network aggregation requires an order of magnitude less communication than the centralized approaches for non-holistic aggregates, and provides performance equal to the centralized approach for worst-case holistic aggregates like MEDIAN. Given these basic experiments about the performance of TAG, we now present a number of optimizations to further improve its performance.

5 Optimizations

In this section, we present several techniques to improve the performance and/or accuracy of the basic approach described above. These techniques can be broadly classified into two categories: First, techniques that improve perfor-

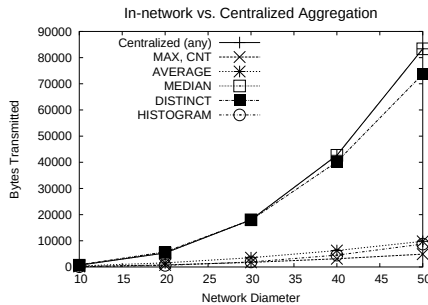


Figure 4: *In-network Vs. Centralized Aggregates, log(messages) vs. Network Diameter*

mance by *collapsing the network stack* to allow applications to peer into parts of network packets they normally would not have access to. Second, optimizations that save messages by *presuming* children or parents are in a specific state when no data is received.

5.1 Taking Advantage of A Shared Channel

In our discussion of aggregation algorithms up to this point, we have largely ignored the fact that sensors communicate over a shared radio channel. The fact that every message is effectively broadcast to all other sensors within range enables a number of optimizations that can significantly reduce the number of messages transmitted and increase the accuracy of aggregates in the face of transmission failures.

We saw an example of how a shared channel can be used to increase message efficiency when a sensor misses an initial request to begin aggregation: it can initiate aggregation even after missing the start request by *snooping* on the network traffic of nearby sensors. When it sees another sensor reporting an aggregate, it can assume it too should be aggregating. This is an instance of collapsing the network stack: by allowing sensors to examine messages not directly addressed to them, sensors are automatically integrated into the aggregation.

Snooping can also be used to reduce the number of messages sent for certain classes of aggregates. Consider computing a MAX over a group of sensors; if a sensor hears a peer reporting a maximum value greater than its local maximum, it can elect to not send its own value and be assured of not affecting the value of the final aggregate. This technique combines a collapsed network stack with presumption: when parents do not hear from children, they presume their children cannot affect the value of the aggregate, thus significantly reducing the number of messages that must be sent. We will discuss variants of this technique in more detail in the next section.

5.2 Hypothesis Testing

The previous example showed that we only need to hear from a particular sensor if that sensor's value will affect the end value of the aggregate. For some aggregates, this can significantly reduce the number of nodes that need to report. This technique can be generalized to an approach we call *hypothesis testing*. If a node is presented with a guess as to the proper value of an aggregate, it can decide locally whether contributing its reading and the readings of its children will affect the value of the aggregate.

For MAX, MIN and other monotonic, exemplary aggregates, this technique is directly applicable. There are a number of ways it can be applied – the snooping approach, where sensors suppress their local aggregates if they hear other aggregates greater than their own, is one. Alternatively, the root of the network (or any subtree of the network) seeking an exemplary sensor value, such as a MIN, might compute the minimum sensor value m over the high-level levels of the subtree, and then abort the aggregate and issue a new request asking for values less than m over the whole tree. In this approach, leaf nodes need not send a message if their value is greater than the minimum observed over the top k levels; intermediate nodes, however, may be called upon to forward partial state records, so even

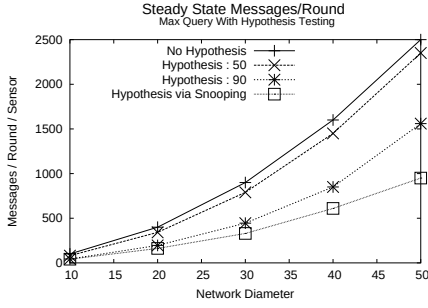


Figure 5: *Benefit of Hypothesis Testing for MAX Aggregate*

if their value is suppressed, they may still have to transmit. Assuming for a moment that sensor values are independent and randomly distributed, then a particular leaf node must transmit with probability $1/b^k$ (where b is the branching factor, so $1/b^k$ is the number of sensors in the top k levels), which is quite low for even small values of k . Since, in a balanced tree, at least half the nodes are in the bottommost level, this technique cuts the total number of messages required by 50% or more. Of course, the performance benefit may not be as substantial for other, non-uniform, sensor value distributions; for instance, a distribution in which all sensor readings are clustered around the minimum will not allow many messages to be saved by hypothesis testing.

We conducted a very simple experiment to measure the benefit of this approach for a MAX aggregate. The results are shown in Figure 5. In this experiment, sensor values were uniformly distributed over the range [0..100], and a hypothesis was made at the root. Sensors whose values were less than the hypothesis did not report their own values, although they continued to forward aggregation requests on to children and periodically rebroadcast this request to reintegrate nodes that had lost or never found their parents, as discussed in Section 5.3 above. Notice that the performance savings are nearly two-fold for a hypothesis of 90. We compared the hypothesis approach with the snooping approach (which will be effective even in a non-uniform distribution); surprisingly, it beat the other approaches by offering a nearly three-fold performance increase over the no-hypothesis case. This is because in the densely packed simple sensor distribution, most sensors have three or more neighbors to snoop on, suggesting that only about one in four sensors will have to transmit. With topology maintenance and forwarding of child values by parents, the total savings is reduced to a factor of three. We have not explored approaches beyond those discussed above for generating an initial hypothesis – we assume the distribution is known beforehand or can be easily approximated via examination of the top few levels of the tree; other more sophisticated techniques, such as gathering statistics about historical behavior, remain an area for future work. We now turn our attention to the applicability of hypothesis testing to other aggregates.

For summary aggregates, such as AVERAGE or VARIANCE, hypothesis testing via a guess from the root can still be applied, although the message savings are not as dramatic. Note that the snooping approach cannot be used: it only applies to monotonic, exemplary aggregates where

values can be suppressed locally without any information from a central coordinator. To obtain any benefit with summary aggregates, the user must define a fixed-size error bound that he or she is willing to tolerate over the value of the aggregate; this error is transmitted into the network along with the hypothesis value. Consider the case of an AVERAGE: any sensor that is within the error bound of the hypothesis value need not answer – its parent will then assume its value is the same as the approximate answer and count it accordingly (for AVERAGE parents must know how many children they have.) It can be shown that the total computed average will not be off from the actual average by more than the error bound, and leaf sensors with values close to the average will not be required to report.⁴ Obviously, the value of this scheme depends on the distribution of sensor values. If values are uniformly distributed, the fraction of leaves that need not report approximates the size of the error bound divided by the size of the sensor value distribution interval. If values are normally distributed, a much larger fraction of leaves do not report.

5.3 Decreasing Communication Frequency

Hypothesis testing provides one way to limit the number of sensors sending during any particular epoch; another presumption based technique is to require sensors to send their state up the pipeline only when their partial state changes, either as a result of their own sensor reading changing or due to the receipt of a new partial state record from one of their children. In other words, parents *presume* their children are online, have not moved, and still have the same partial state.

We have not attempted to realistically model the rate of change of sensors values in our simulation; understanding how real sensor values change remains an area for future work. We can lower-bound the number of messages sent in this scheme by noting that, when sensors suppress the transmission of values, they must periodically transmit heartbeats to advertise that they are still alive. In our current simulation, the interval between heartbeats is set to eight epochs. Thus heartbeats alone, without any changing values, require 1/8 the messages of the unsuppressed pipelined scheme.

This completes our discussion of optimizations to the pipelined approach; in the next section, we augment the algorithms presented thus far to support grouping.

6 Grouping

The basic technique for grouping is to push down an expression that specifies the attribute to be used for grouping, ask sensors to choose the group they belong to, and then, as answers flow back, update the aggregate values in the appropriate groups.

Partial state records are aggregated just as in the pipelined approach described above, except that those records are now tagged with a group number. When a sensor is a leaf, it applies the grouping expression to compute

⁴Note that the sensitivity to error varies based on the aggregate. To meet the user's error bound for the final aggregate value, the per-sensor error for a SUM, for example, must be much less than the per sensor error for an AVERAGE, since in an AVERAGE, each sensor contributes only $1/n$ of its value to the final aggregate.

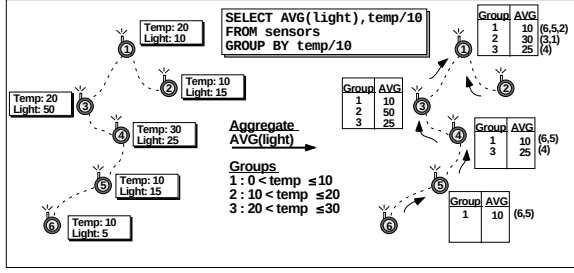


Figure 6: A sensor network (left) with an in-network, grouped aggregate applied to it (right). Parenthesized numbers represent sensors that contribute to the average

a group number. It then tags its partial state record with the group and forwards it on to its parent. When a sensor receives an aggregate from a child, it checks the group number. If the child is in the same group as the sensor, it combines the two values using the combining function f . If it is in a different group, it stores the value of the child's group along with its own value for forwarding in the next interval. If another child message arrives with a value in either group, the sensor updates the appropriate aggregate. During the next interval, the sensor will send out the value of all the groups it collected information about during the previous interval, combining information about multiple groups into a single message as long as the message size permits. Figure 6 shows an example of computing a query grouped by temperature that selects average light readings. In this snapshot, data has filled the pipeline, such that results from the bottom of the tree have reached the root.

Recall that SQL queries also contain a **HAVING** clause, which constrains the set of groups in the final query result. We sometimes pass this predicate into the network along with the grouping expression. The predicate is only sent into the network if it can potentially be used to reduce the number of messages that must be sent: for, example, if the predicate is of the form $\text{MAX}(\text{attr}) < x$, then information about groups with $\text{MAX}(\text{attr}) \geq x$ need not be transmitted up the tree, and so the predicate is sent down into the network. When a node detects that a group does not satisfy a **HAVING** clause, it can notify other nodes in the network of this information to suppress transmission and storage of values from that group. Note that **HAVING** clauses can be pushed down only for monotonic aggregates; non-monotonic aggregates are not amenable to this technique. However, not all **HAVING** predicates on monotonic aggregates can be pushed down; for example, $\text{MAX}(\text{attr}) > x$, cannot be applied in the network because a node cannot know that, just because its local value of attr is less than x , the MAX over the entire group will be less than x .

Because the number of groups can exceed available storage on any one (non-leaf) sensor, a way to evict groups is needed. Once an eviction victim is selected, it is forwarded to the sensor's parent, which may choose to hold on to the group or continue to forward it up the tree. Notice that a single sensor may evict the same group multiple times in a single epoch. Performing multiple evictions requires multiple radio messages per epoch. Because groups can

be evicted, the base station at the top of the network may be called upon to combine partial groups to form an accurate aggregate value. Evicting partially computed groups is known as *partial preaggregation*, as described in [11].

6.1 Grouping Experiments

We ran a number of experiments to compare various eviction techniques in different sensor environments. Sensors have two attributes, *light* and *temperature*; we are selecting the light attribute grouped by temperature, where temperature can take on 100 distinct values (or, alternatively, can fall within 1 of 100 temperature bins). We measured the number of partial state records sent collectively by all the sensors. We varied three parameters: the eviction policy, the distribution of sensor values, and the amount of storage available to sensors for group data (in terms of state records). We tried a number of simple eviction policies, such as evicting the group with the fewest members, evicting a random group, and evicting the group with the most members. We found that the choice of policy made little difference for any of the sensor-value distributions we tested. This is largely due to the tree topology: near the leaves of the tree, most sensors will see only a few groups, and the eviction policy will matter very little. At the top levels of the tree, the eviction policy becomes important, but the cost of forwarding messages from the leaves of the tree tends to dominate the savings obtained at the top. In the most extreme case, the difference between the best and worst case eviction policy accounted for less than 10% of the total messages.

While eviction policy did not matter, we did, however, observe that evicting multiple groups simultaneously is beneficial, as long as all of those groups can fit within a single message. Figure 7 shows this effect in an experiment with an **AVERAGE** aggregate, using a random eviction policy and a random sensor-value distribution in a simple topology with diameter 50; each column represents an experiment with a different amount of group storage (G), and number of simultaneous evictions (E). In the best case, five evictions with a storage of five groups, about 40% fewer messages are sent than in the worst approaches, where only one eviction is done at a time. Evicting more than five groups at a time is not beneficial because above that, two network messages are required per eviction, since each partial average requires four bytes of storage, so ten groups occupy 40 bytes, which exceeds the 30 byte message size. This also explains the slight upward trend in Figure 7 for group storage above five groups: even when there is no eviction, larger numbers of groups require more than one network message when sent to the sensor's parent.

In the process of testing our group-eviction policies, we also experimented with several sensor-value distributions. Although the choice of distribution had little effect on the effectiveness of the eviction policies, it did, as expected, affect the total number of messages sent. Widely spaced, uniform distributions, with lots of groups in many parts of the network, tended to incur many more evictions, and hence send more messages, than more clustered, non-uniform distributions with fewer or more spatially collocated groups. We omit a detailed summary of the effect of distribution

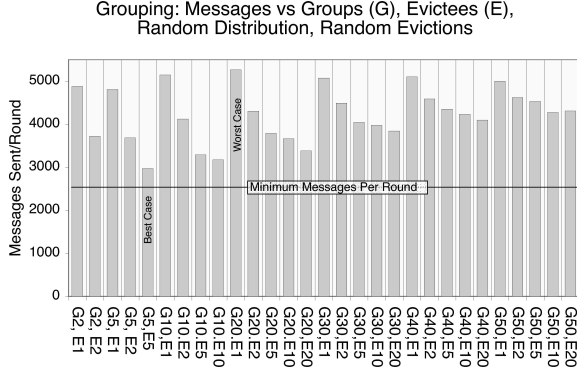


Figure 7: Effect of Group Size and Number of Evictions in A Diameter 50 Network

since it does not affect the relative performance of our techniques and space is at a premium.

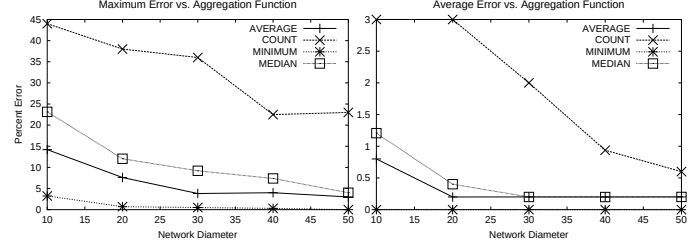
Thus, we have shown how to partition sensor readings into a number of groups and properly compute aggregates over those groups, even when the amount of group information exceeds available storage in any one sensor. In the next section, we discuss how TAG deals with loss and present a simple extension for improving its reliability.

7 Improving Tolerance to Loss

In this section, we address the problem of loss and its effect on the algorithms presented thus far, since previous experiments used a reliable environment where no messages were dropped and no sensors were lost. Throughout this section, the reader should recall that loss is a matter of life in the sensor domain; our techniques do improve aggregate quality, but the real-time, lossy nature of sensor networks means we can simply never provide the data integrity guarantees of traditional databases.

7.1 Effects of A Single Loss

In the first experiment, we study the effect a single sensor going offline has on the value of the aggregate; we believe this is an important measurement as it gives some intuition about the magnitude of error that a single loss can generate. Note that, because we are doing hierarchical aggregation, a single sensor going offline causes the entire subtree rooted at the sensor to be (at least temporarily) disconnected. We ran this experiment with the simple topology, with sensor readings chosen from the uniform distribution over [1..1000]. After running the simulation until the pipeline was full, we selected, uniformly and at a random, a sensor to disable. In this environment, children of the disabled node were temporarily disconnected but eventually their values were reintegrated into the aggregate once they rediscovered their parents. Note that the amount of time taken for lost nodes to reintegrate is directly proportional to the depth of the lost sensor, so we did not measure it experimentally. Instead, we measured the maximum temporary deviation from the true value of the aggregate that the loss caused in the perceived aggregate value at the root during any epoch. This maximum was computed over 100 runs at each data point; we also show the average percent error over all of the epochs of all of the same 100 runs.



(a) Maximum Error

(b) Average Error

Figure 8: Effect of a Single Loss on Various Aggregate Functions. Computed over a total of 100 runs at each point.

Figure 8 shows the results of this experiment. Note that the maximum loss is highly variable and that some aggregates are considerably more sensitive to loss than others. COUNT, for instance, has a very large error in the worst case: if a node that connects the root to a large portion of the network is lost, the temporary error will be very high. The variability in maximum error is because a well connected subtree is not always selected as the victim; indeed, as the network size increases, the chances of selecting such a node go down. In the average case, the error associated with a COUNT is not as high: most losses do not result in a large number of disconnections. Note that MIN is insensitive to loss when in this uniform distribution, since several nodes are at or near the true minimum. The error for both MEDIAN and AVERAGE is less than COUNT and more than MIN: both are sensitive to the variations in the number of sensors involved, but not as dramatically as COUNT.

7.2 Effect of the Realistic Communication Model

In the second experiment, we examine how well TAG performs in the realistic environment (discussed in Section 4.2 above). In such an environment, without some technique to counteract loss, a large number of partial state records will invariably be dropped and not reach the root of the tree. We ran a simple experiment to measure the effect of this loss in the realistic environment. The simulation was run until the pipeline filled (the first aggregate arrived at the root), and then the average number of sensors involved in the aggregate over the next several epochs was measured. The "No Cache" line of Figure 9 shows the performance of this approach; at diameter 10, about 40% of the partial state records are reflected in the aggregate at the root; by diameter 50, this percentage has fallen to less than 10%. Performance falls off as the number of hops between the average sensor and the root increases, since the probability of loss is compounded by each additional hop. Thus, the basic TAG approach presented so far, running on current prototype hardware (with its very high loss rates), is not highly tolerant to loss, especially for large networks. Note that any centralized approach would suffer from the same loss problems. In the next section, we will show two simple modifications to TAG that significantly improve its loss tolerance.

7.3 Child Cache

To improve the quality of aggregates, we propose a simple caching scheme: parents remember the partial state records their children reported for some number of rounds, and use

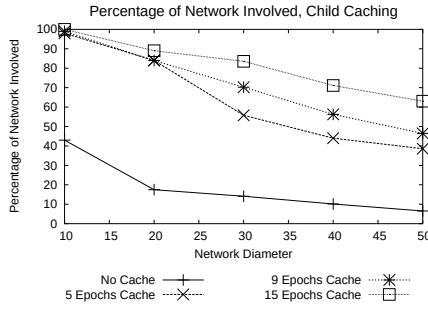


Figure 9: *Percentage of Network Participating in Aggregate With Realistic Loss Model For Varying Amounts of Child Cache Per Node*

those previous values when new values are unavailable due to lost child messages. As long as the duration of this memory is shorter than the interval at which children select new parents, this technique will increase the number of nodes included in the aggregate without over-counting any nodes.

We conducted some experiments to show the improvement this technique offers over the basic pipelining approach; we allocate a fixed size buffer at each node and measure the average number of sensors involved in the aggregation as in Section 7.2 above. The results are shown in Figure 9 – notice that even five epochs of cached state offer a significant increase in the number of nodes counted in any aggregate, and that 15 rounds increases the number of sensors involved in the diameter 50 network to 70% (versus less than 10% without a cache). There are two drawbacks to caching; First, it uses memory that could be used for group storage. Second, it sets a minimum bound on the time sensors must wait before determining their parent has gone offline; given the benefit it provides in terms of accuracy, however, we believe it to be useful despite these disadvantages. The substantial benefit of this technique also points out an interesting tradeoff for the designers of mote hardware and low-level networking protocols: by providing more RAM for caching, the perceived quality of radio transmission can be dramatically increased.

7.4 Using Available Redundancy

Because there may be situations where the RAM or latency costs of the child cache are not desirable, it is worthwhile to look at alternative approaches for improving loss tolerance. In this section, we show how the network topology can be leverage to increase the quality of aggregates. Consider a sensor with two parents: instead of sending its aggregate value to just one parent, it can send it to both parents. It is easy for a node to discover that it has multiple parents, since it can simply build a list of nodes it has heard that are one step closer to the root. Of course, for duplicate-sensitive aggregates (see Section 2.2), sending results to multiple parents has the undesirable effect of causing the node to be counted multiple times. The solution to this is to send part of the aggregate to one parent and the rest to the other. Consider a COUNT; a sensor with $c - 1$ children and two parents can send a COUNT of $c/2$ to both parents instead of a count of c to a single parent. A simple statistical analysis reveals the advantage of doing this: assume that a message is transmitted with probability p , and that

losses are independent, so that if a message m from sensor s is lost in transition to parent P_1 , it is no more likely to be lost in transit to P_2 .⁵ First, consider the case where s sends c to a single parent; the expected value of the transmitted count is $p \times c$ (0 with probability $(1 - p)$ and c with probability p), and the variance is $c^2 \times p \times (1 - p)$, since these are standard Bernoulli trials with a probability of success p multiplied by a constant c . For the case where s sends $c/2$ to both parents, linearity of expectation tells us the expected value is the sum of the expected value through each parent, or $2 \times p \times c/2 = p \times c$. Similarly, we can sum the variances through each parent to get:

$$\text{var} = 2 \times (c/2)^2 \times p \times (1 - p) = c^2/2 \times p \times (1 - p)$$

Thus, the variance of the multiple parent COUNT is much less than with just a single parent, although its expected value is the same. This is because it is much less likely (assuming independence) for the message to both parents to be lost, and a single loss will less dramatically effect the computed value.

We ran an experiment to measure the benefit of this approach in the realistic topology for COUNT with a network diameter of 50. We waited until the pipeline was full, then measured average number of sensors involved in the aggregation over each of the next 50 epochs. When sending to multiple parents, the mean COUNT was 974 ($\sigma = 330$), while when sending to only one parent, the mean COUNT was 94 ($\sigma = 41$). This approach is particularly effective a dense topology because most sensors have many parents, and so, while the chance of loss of one parent is high, the probability that many will be lost is low.

This technique applies equally well to any distributive or algebraic aggregate. For holistic aggregates, like MEDIAN, this technique cannot be applied, since partial state records cannot be easily decomposed.

This completes our discussion of algorithms for TAG. We now summarize the extensive related work in the networking and database communities.

8 Related Work

The database community has proposed a number of distributed and push-down based approaches for aggregates in database systems [16, 21], but as discussed in see Section 3, most of these assume a well-connected, low-loss topology that is unavailable in sensor networks. The partial preaggregation techniques [11] used to enable group eviction were proposed as a technique to deal with very large numbers of groups to improve the efficiency of hash joins and other bucket-based database operators. The classifications of aggregates presented in Section 2.2 were originally developed as a part of the research on data-cubes [5]. [6] discusses generalized aggregate predicates and their placement in query plans. [18] discusses online aggregation in the context of nested-queries; it proposes optimizations to reduce the flow of tuples between outer and inner queries that bear some similarities to our technique for pushing HAVING clauses into the network. With respect to query language, we rely on the SEQ model of streams

⁵ Although failure independence is not always a valid assumption, it will occur when a hidden-node garbles communication to P_1 but not P_2 , or when one parent is forwarding a message and another is not.

for the semantics of our aggregate queries [15]. There are many other relevant languages in models from the Temporal Database literature; see [17] for a survey of relevant work. The Cougar project at Cornell [14] discusses queries over sensor networks, as does our own work on Fjords [12], although the former only considers moving selections (not aggregates) onto sensors and neither presents specific, power-sensitive algorithms for use in sensor networks.

Literature on active networks [19] first identified the idea that the network could simultaneously route and transform data, rather than simply serving as an end-to-end data conduit. Within the sensor network community, work on networks that perform data analysis has been largely confined to the USC/ISI and UCLA communities. Their work on directed diffusion [9] discusses techniques for moving specific pieces of information from one place in a network to another, and proposes aggregation-like operations that nodes may perform as data flows through them. [7] proposes a scheme for imposing names onto related groups of sensors in a network, in much the way that our scheme partitions sensor networks into groups. These papers recognize that aggregation dramatically reduces the amount of data routed through the network but are focused on application specific solutions that, unlike the approach in TAG, are not obviously applicable or efficient in a wide range of environments. We initially noted the advantages of a database style approach to aggregation in a workshop publication [13].

Networking protocols for routing data in wireless networks are very popular within the literature [10, 1, 4], however, none of them address higher level issues of data processing, merely techniques for data routing. Our tree-based routing approach is clearly inferior to these approaches for peer to peer routing, but works well for the aggregation scenarios we are focusing on.

9 Conclusions

As sensor networks become more widely deployed, especially in remote, difficult to administer locations, power sensitive methods to query and extract data from those networks will become increasingly important. TAG satisfies this demand by translating declarative aggregate queries with grouping into in-network execution plans where aggregate processing is tightly coupled with routing and topology discovery protocols. This in-network approach offers an order of magnitude reduction in energy consumption over approaches where data is aggregated and processed centrally. Tight integration with the network enables optimizations that further decrease message costs and improve tolerance to failure. The grouping facility, which includes the ability to push in message saving HAVING clauses, yields correct results and good performance even when the space required to store information about all groups far exceeds the memory of any one sensor. The applicability of the different optimizations, as well as the cost of computing various classes of aggregates, is tied into a taxonomy of properties especially relevant to aggregation in sensor networks.

Looking forward, we see the TAG approach as an essential starting point for an important investigation into the re-

lationships between databases and networked systems. By demonstrating an error-resilient, in-network approach that lends itself to both snapshot and continuous aggregates, along with a taxonomy of aggregation functions and intuition about the distributability, performance, and applicability of optimizations to those functions, the groundwork is laid for moving entire data-processing systems down into the network.

Acknowledgments

Robert Szweczyk, David Culler, and Ramesh Govindan contributed to the design of the networking protocols discussed in this paper. Per Åke Larson suggested the use of partial preaggregation for group eviction.

References

- [1] W. Adjue-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *ACM SOSP*, December 1999.
- [2] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *VLDB*, 1987.
- [3] D. Ganesan. Network dynamics in rene motes. PowerPoint Presentation, January 2002.
- [4] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu. Preemptive routing in ad hoc networks. In *ACM MobiCom*, July 2001.
- [5] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. February 1996.
- [6] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehousing environments. In *VLDB*, 1995.
- [7] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *SOSP*, October 2001.
- [8] J. Hill, R. Szweczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.
- [9] C. Intanagonwiwat, R. Govindan, , and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCOM*, Boston, MA, August 2000.
- [10] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCOM*, Seattle, WA, 1999.
- [11] P.-Å. Larson. Data reduction by partial preaggregation. In *ICDE*, 2002. (to appear).
- [12] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002. (to appear).
- [13] S. Madden, R. Szweczyk, M. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. Submitted, Workshop on Mobile Computing and Systems Applications, 2002.
- [14] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management*, Hong Kong, January 2001.
- [15] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database systems. In *VLDB*, 1996.
- [16] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD*, 1995.
- [17] R. T. Snodgrass, editor. *The SQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.
- [18] K.-L. Tan, C. H. Goh, and B. C. Ooi. Online feedback for nested aggregate queries with multi-threading, 1999.
- [19] D. Tennenhouse. Active networks. In *OSDI*, October 1996.
- [20] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *ACM Mobicom*, July 2001.
- [21] W. P. Yan and P. Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.
- [22] A. Yu and J. Chen. *The POSTGRES95 User Manual*. UC Berkeley, 1995. <http://www.uoi.gr/services/noc/documentation/postgress/pg95user.html>.