In-Network Aggregation Trade-offs for Data Collection in Wireless Sensor Networks

Ignacio Solis*
isolis@cse.ucsc.edu
Computer Engineering Department
University of California, Santa Cruz

Katia Obraczka
katia@cse.ucsc.edu
Computer Engineering Department
University of California, Santa Cruz

December 9, 2005

Abstract

This paper explores in-network aggregation as a power-efficient mechanism for collecting data in wireless sensor networks. In particular, we focus on sensor network scenarios where a large number of nodes produce data periodically. Such communication model is typical of monitoring applications, an important application domain sensor networks target. The main idea behind in-network aggregation is that, rather than sending individual data items from sensors to sinks, multiple data items are aggregated as they are forwarded by the sensor network.

Through simulations, we evaluate the performance of different in-network aggregation algorithms, including our own *cascading timers*, in terms of the trade-offs between energy efficiency, data accuracy and freshness. Our results show that timing, i.e., how long a node waits to receive data from its children (downstream nodes in respect to the information sink) before forwarding data onto the next hop (toward the sink) plays a crucial role in the performance of aggregation algorithms for applications that generate data periodically. By carefully selecting when to aggregate and forward data, *cascading timers* achieves considerable energy savings while maintaining data freshness and accuracy. We also study in-network aggregation's cost-efficiency using simple mathematical models.

Since wireless sensor networks are prone to transmission errors and losses can have considerable impact when data aggregation is used, we also propose and evaluate a number of techniques for handling packet loss. Simulations show that, when used in conjunction with aggregation protocols, the proposed techniques can effectively mitigate the effects of random transmission losses in a power-efficient way.

1 Introduction

Sensor networks are typically data driven, i.e., the whole network cooperates in communicating data from sensors (*information sources*) to *information sinks*. One of the main challenges raised by sensor networks is the fact that they are usually power constrained since sensing nodes typically exhibit limited capabilities in terms of processing, communication, and especially, power. Sensor networks' power limitation is aggravated by the fact that, often, once deployed, they are left unattended for most of their lifetime. Thus, energy conservation is of prime consideration in sensor network protocols in order to maximize the network's operational lifetime.

In-network aggregation is a well known technique to achieve energy efficiency when propagating data from information sources (e.g., sensors) to sink(s). The main idea behind in-network aggregation is that, rather than sending individual data items from sensors to sinks, multiple data items are aggregated as

*The author is now at the Palo Alto Research Center (PARC) and can also be reached at isolis@parc.com

they are forwarded by the sensor network. Data aggregation is application dependent, i.e., depending on the target application, the appropriate data aggregation operator, or *aggregator*, will be employed. For example, suppose that in a controlled temperature environment, the average temperature needs to be monitored. As sensors generate temperature readings periodically, internal nodes in the data collection tree (rooted at the information sink and spanning relevant data sources [1]) average data received from downstream nodes and forward the result toward the information sink. The net effect is that, by transmitting less data units, considerable energy savings can be achieved. However, how much energy is saved depends on the type of aggregator employed. For instance, in the running average scenario just depicted, a number of packets containing temperature readings from individual sensors are averaged and result in a single packet of the same size as the ones that carry individual temperature readings. However, if the only possible aggregator is *concatenation*, i.e., multiple data items are concatenated and transmitted as a single packet, then the sole source of energy savings is more efficient medium access.

From the information sink's point of view, the benefits of in-network aggregation are that in general (1) it yields more manageable data streams avoiding overwhelming sources with massive amounts of information, and (2) performs some filtering and preprocessing on the data, making the task of further processing the data less time- and resource consuming.

Because of its well-known power efficiency properties, in-network aggregation has been the focus of several recent research efforts on sensor networks. As a result, a number of data aggregation algorithms targeting different sensor network scenarios have been proposed. Directed diffusion [1], TAG [2], eScan [3], and Sensor Protocols for Information via Negotiation (SPIN) [4] are some notable examples. In this paper, we focus on the requirements of an important class of sensor network applications, namely applications that generate data periodically. Monitoring (including monitoring of continuous environmental conditions like temperature, humidity, seismic activity, etc.) is a good example of such applications. One of the constraints imposed by periodic data generation on aggregation algorithms is timing. In other words, how long should a node wait to receive data from its children (downstream nodes in respect to the information sink) before forwarding data already received? Note that there is a tradeoff between data accuracy and freshness, i.e., the longer a node waits, the more readings it is likely to receive and therefore, the more accurate the information it sends out. On the other hand, waiting too long may result in stale data. Furthermore, if a node waits too long, it may interfere with the next "data wave".

Our hypothesis is that timing models play a crucial role in the accuracy and freshness delivered by data aggregation. In this paper, we study how different timing schemes affect the performance of in-network aggregation algorithms. Based on their timing model, we classify existing periodic data aggregation protocols into three categories, namely: *periodic simple*, *periodic per-hop*, and *periodic per-hop adjusted*.

Periodic simple aggregation works by having each node wait a pre-defined period of time (referred to as *timeout*), aggregate all data items received, and send out a single packet containing the result. As discussed in Section 7 below, the directed diffusion [1] sensor network communication paradigm belongs to this category. Aggregation mechanisms in the *periodic per-hop* category have nodes send the aggregated packet as soon as they hear from all their children. A maximum timeout interval equal to the data generation period is used in case the reports get lost. Finally, *periodic per-hop adjusted* uses the same basic principle of *periodic per-hop* but schedules a node's timeout based on its position in the distribution tree (rooted at the information sink and spanning all reporting- as well as appropriate intermediate nodes). Our own *cascading timers* aggregation mechanism falls within this category, and, when compared to other existing *periodic per-hop adjusted*

[1]While in this paper we assume that all nodes produce data, the proposed techniques and the conclusions we draw might apply to the case where only a subset of the nodes are (relevant) information sources.

algorithms, presents benefits such as not requiring clock synchronization among nodes and minimizing timer scheduling overhead. *Cascading timers* schedules a node's timeout based on the time it takes for a packet to travel a single hop, or the *single hop delay* and the number of hops to reach the sink. We also study how the value selected for the *single hop delay* impacts the performance of *cascading timers*.

In summary, the contributions of this paper include: (1) development of *cascading timers* aggregation for periodic data generation applications including a detailed analysis of *cascading timers*' dependence on per-hop delay, (2) trade-off analysis of in-network data aggregation using simple analytical models (3) comparative performance study of different aggregation algorithms using extensive simulations, and (4) development of different loss recovery mechanisms and study of how they impact performance of data aggregation under lossy environments.

For evaluating the performance of the different in-network aggregation mechanisms, energy efficiency, data accuracy and freshness, and communication overhead are used as performance metrics. We should also point out that, unlike previous evaluation studies targeting sensor network protocols, a wide range of network scenarios including different information sink placement strategies are used.

The remainder of the paper is organized as follows. In-network aggregation with *cascading timers* is described in Section 2. Other aggregation types are mentioned in Section 3. Section 4 investigates in-network aggregation's cost-efficiency using simple mathematical models. Section 5 describes the simulation experiments we conduct to compare the performance of different in-network aggregation algorithms, including the experimental setup used, results obtained, as well as the impact of the per-hop delay on the performance of *cascading timers*. Techniques for handling packet losses and their performance are introduced in Section 6. Section 7 discusses related work. Finally, Section 8 presents our concluding remarks and directions for future work.

2 Cascading Timers Aggregation

As previously discussed, our *cascading timers*[5] aggregation algorithm targets periodic data generation applications in which nodes produce data at regular periods. A given node aggregates data received from its children into a single data item, which is then forwarded upstream towards the information sink[2]. Application scenarios that fit well within this communication model include monitoring of continuous environmental conditions like temperature, humidity, seismic activity, etc. While we focus on the single information sink scenario, the proposed technique can be applied to multi-sink scenarios.

Some of *cascading timers*' design goals include:

- Simplicity: given that sensor network nodes are typically anemic devices regarding energy, processing, storage, and communication capabilities, designing simple aggregation algorithms is key.

- Efficiency: generate close to minimal control overhead. Again, this is a critical requirement in the resource-constrained environments our algorithms target.

- No clock synchronization: *cascading timers* does not require clock synchronization among nodes. No matter how efficient clock synchronization mechanisms become, they will require additional message exchange among nodes and thus incur additional energy consumption. Efficient synchronization algorithms [6] have emerged recently and might allow other tradeoffs.

- Routing protocol independence: no specific underlying routing protocol is assumed.

Similar to most periodic aggregation mechanisms, *cascading timers* starts by having the sink broadcast the initial request to all nodes. This initial request triggers a simple tree establishment protocol which sets up reverse paths from all nodes back to the sink

[2]As explained in more detail below, data is aggregated over a tree rooted at the information sink

or root of the tree. Upon receiving the request message, nodes send a reply back to their parent in the tree. Each node can then deduce how many children it has. Nodes assume a broadcast medium and forward data using one-hop broadcasts. In order to avoid collisions, transmissions are scheduled using a small staggering delay. The setting of the staggering interval will be discussed in detail in Section 5.3.

Note that tree establishment is essentially the overhead incurred by *cascading timers* and most other in-network aggregation mechanisms. Even if no aggregation is employed, a distribution tree is typically used to collect data from information sources to sinks.

In *cascading timers*, instead of having nodes schedule randomly their timeout, i.e., the time interval they wait to receive data from their children before forwarding the next data aggregate, a node's timeout is set based on the node's position in the data distribution tree. Thus, a node's timeout will happen right before its parent's. This causes the so-called "cascading" effect: data originating at the leaves is clocked out first, reaching nodes in the next tree level in time to be aggregated with data from other leaf nodes and locally generated data, and so on. The net effect is that a "data wave" reaches the sink in one period. This is the main reason behind *cascading timers*' ability to achieve power efficiency and yet deliver fresh data at sink nodes.

Timeout scheduling is part of the distribution tree setup protocol and is triggered by the initial request from the sink. The sink's request contains a "hop count" field which gets incremented as the request travels toward the leaf nodes. Using this hop count information, nodes can estimate their distance, in time, to the sink and schedule their timeout to produce the cascading effect.

Figure 1 shows graphically timeout calculation in *cascading timers*, where t is the data generation period, h is a node's distance to the sink in number of hops, and shd, the *single hop distance*, is the delay to traverse one hop. Once the request packet is received, a node schedules its timeout to be after it's children and before it's parents. This is done by calculating a "mirror" image of the time the packet was received to the end of the period (e), hence the first timer is set for $2e$. Subsequent timers will continue to be scheduled every t interval.

Note that a node's timeout depends on the *single hop distance*, or shd. We investigate this dependence in detail in Section 5.3 and show how it affects the performance of the algorithm. As previously pointed out, *cascading timers*' timing scheme is parallel to the ones employed by both TAG [2] and Convergecasting [7]. According to our taxonomy, all three mechanisms are classified in the *periodic per-hop adjusted* category. In our simulations, we compare the different aggregation techniques. Our *cascading timers* and TAG represent *periodic per-hop adjusted* algorithms.

Our implementation of *TAG* tries to follow their algorithm as closely as possible. The data generation period is equivalent to TAG's epoch. We estimate the maximum number of hops and divide the period in this many slots. To avoid collisions, nodes transmit at a random uniformly distributed time within the slot corresponding to their height on the aggregation tree.

3 Other Periodic Aggregation Mechanisms

Below we describe in detail the other classes of aggregation algorithms we use in our comparative study. As baseline, we employ no in-network aggregation when sending data from information sources to the sink. As previously pointed out, even in the no-aggregation case, we employ a distribution tree rooted at the information sink and spanning all (relevant) data sources. As packets flow from the leaves to the root, nodes simply forward them along the tree.

3.1 Periodic Simple

Nodes in *periodic simple* aggregation wait a predefined amount of time, aggregate all the data received in that period, and send out a single packet. The aggregation period is equal to the data generation period, which, for most our simulation experiments, is set to 1 second.

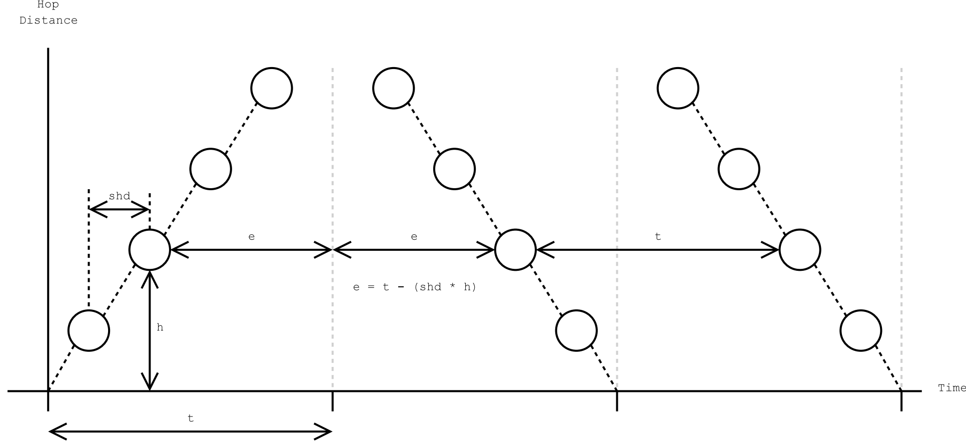

Figure 1: *Cascading timers* timeout calculation

This class of aggregation protocols represents the basic mechanism used by Directed Diffusion [1] considering that all nodes have relevant data to send. Based on feedback (or reinforcements) from the sink, every node uses a specific gradient which determines the rate at which data is sent to the sink. Note that nodes are not necessarily synchronized when "clocking out" data.

3.2 Periodic Per-Hop

According to *per-hop simple* aggregation, once all data items are received from a node's children in the distribution tree, an aggregated packet is produced and sent onto the next hop. Each node uses a timeout for sending out packets in case their children's response is lost. The timeout is equal to the data generation period since once that time is up, we will be expecting and producing new readings.

4 Tradeoff Analysis

In this section, we study the performance tradeoffs raised by in-network aggregation. Using simple mathematical models, we conduct a cost-efficiency analysis of data aggregation.

4.1 Energy Efficiency

Recall that data aggregation's main goal is to achieve energy efficiency. It does so by reducing the number of packets transmitted. Ideally, when aggregation is employed, only a single packet is sent by each node per data collection period, or round. Thus the number of packets sent per round, or $AggPkt/Round$, is given by Equation 1.

$$AggPkt/Round = n \quad (1)$$

$$NoAggPkt/Round = \sum_{i \in N} d_i \quad (2)$$

$$= \sum_{d=1}^{Max(d)} d * N_d \quad (3)$$

Without aggregation, each node will send a packet that will be forwarded to the sink. Each hop traversed by a packet counts as one packet being sent, hence, the cost of getting a reading from a node equals its height on the data distribution tree. The number of packets sent, or $NoAggPkt/Round$ is thus given by Equation 2, where d_i is the depth of node i and N is the set of participating nodes. Alternatively, we can compute the number of packets transmitted per round as a function of the number of nodes at every tree level. This is described by Equation 3, where d is the number of tree levels and N_d is the number of nodes at depth d.

Clearly, in an average scenario, no-aggregation will send more packets per round. No-aggregation's best case scenario is when the data distribution tree has maximum depth ($Max(d)$) equal to 1. The number of packets sent per round without aggregation is equal to n (Equation 2). This confirms that in "shallow" distribution trees, the benefits of aggregation are not as significant. But it never performs worse than no-aggregation.

On the other hand, in deep trees, aggregation has significant impact in reducing the number of packets transmitted. The worst case scenario for no-aggregation is a "line" topology, a tree where every node has only one child. In the case of a line topology with depth n, the summation in Equation 3 results in $\frac{(n)(n+1)}{2}$ packets per round.

Essentially, these are the lower and upper bounds for the number of packets sent when no-aggregation is used. As demonstrated in Section 5.2, all the aggregation algorithms studied are able to achieve ideal energy efficiency by sending only one packet per node per round. The difference in performance between them lies in the data accuracy and freshness they achieve.

We assume that an aggregated packet will have the same size as a non-aggregated one. This is the case of operations like computing averages, selecting the maximum or minimum value, etc. This implies that, as packets flow toward the sink, they will not grow in size. Hence, our energy efficiency metric computes number of packets-, rather than number of bytes sent.

On the other extreme, if aggregation by concatenation is employed, i.e., data items are concatenated as they traverse the sensor network, there will still be savings on the number of packets transmitted, but the number of bytes sent will not be reduced. Nevertheless, aggregation will still be advantageous in terms of medium acquisition and scheduling. *Cascading timers* will also yield improved data freshness due to its low delay.

4.2 Complexity

Note that tree establishment is essentially the overhead incurred by *cascading timers* and most other in-network aggregation mechanisms. Even if no aggregation is employed, a distribution tree is typically used to propagate data from information sources to sinks.

In terms of communication complexity, tree establishment costs n packets as each node disseminates the original query that triggers formation of the tree. If the protocol also generates a reply from every child, there will be additional $n-1$ packets since every node, except the root, will be a child. Tree establishment's total cost is then $2n-1$ packets.

Nodes reply back to their parents when they receive the original query so that each node knows how many children it has. This information is used to optimize the algorithms considered by allowing a node to know when it has received the readings from all its children. If this optimization is not performed, tree establishment cost is reduced to n packets, which is equal to the cost of tree formation for no-aggregation. Note that nodes can still find out how many children they have as the algorithm runs. In both cases, the algorithm is able to handle new nodes joining the tree as well as existing nodes leaving/failing.

In order to adapt to topology changes, tree re-establishment is performed, which will cost n (or $2n-1$ if nodes reply back to their parents) packets. Of course, localized tree re-establishments can be performed so as to reduce overhead.

All algorithms we present here (including no aggregation) incur these tree formation costs. Hence, under the conditions used in the experiments reported in Section 5, the extra cost of using in-network aggregation over no-aggregation could be $n-1$ additional control packets if we want nodes to know how many children they have right from the start of the algorithm.

In terms of storage complexity, depending on the aggregation operator used, readings from a node's children may need to be stored locally. In our example application, aggregated data can be stored as a single data item which will be sent by the node on timer expiration or when all its children's readings are received. This requires at most one data item size storage unit. In the case of aggregation by concatenation, a node needs as many data item size storage

units as the number of children it has. For instance, a scenario with a single 64-bit data value plus 64-bit for control information, assuming an average number of children of 32, would require a total of 512-byte storage. The computational complexity is also trivial for most cases where simple aggregators (e.g., calculating the minimum, maximum, average, sum, etc.) are employed.

5 Simulations

For our comparative study of the different in-network aggregation algorithms, we ran extensive simulations using the `ns-2` network simulator [8].

5.1 Experimental Setup

In the experiments we conducted, 100 nodes were randomly placed in a $500*500m^2$ area. Nodes' transmission range and data rate are set to 100 meters and 115 Kbps, respectively. 802.11b's broadcast mode is used as the MAC-layer protocol and FLIP [9] as the network protocol so we can take advantage of its optimized headers. Based on values used by commercially available radios, we set transmission and reception power levels to 24.75 and 13.5 milliwatts, respectively. Idle power consumption was set to 0.675 milliwatts to reflect an optimized MAC layer, i.e., MAC protocols that switch to low-power radio mode whenever possible.

In order to avoid collisions, nodes stagger their transmissions using a small random interval. This is important when performing data collection over a tree, especially when nodes try to send at scheduled intervals based on their depth in the tree. The maximum staggering value used was 0.03 seconds. Nodes pick a uniformly distributed random timer between 0 and this value before sending. In Section **??**, we discuss the setting of the staggering interval in more detail.

Nodes are stationary and no transmission errors were simulated for the first set of results. However, packets can still be lost due to collisions. Mechanisms to handle packet loss and their performance are reported in Section 6 below. Simulations were run for 20 seconds with data being generated every second (round). Although establishing the distribution tree can be initiated by the data request from the sink, in our simulations the tree was formed at time 1 second and data collection was triggered by the sink at time 3 seconds. We present steady state results, that is, measurements taken during the second half of the simulation (during the last 10 seconds).

Data points were obtained by averaging over twenty different runs using different seeds to perform random node placement. As will be evident in our results, information sink placement can greatly affect the performance of tree-based aggregation algorithms. For this reason, we ran experiments using three different sink placement strategies: corner, center, and random placement. Placing the sink in corners means that the resulting collection trees will be deeper. Center placement minimizes tree height.

Performance metrics we use include **energy consumed**, **data accuracy**, **data freshness**, and **overhead**. While energy consumed measures the algorithm's energy efficiency, data accuracy and freshness account for its effectiveness in terms of conveying as much information as possible to the sink in a timely manner.

In these experiments, we do not model the actual values being sensed by the nodes, how fast they are changing or in what manner. Therefore, accuracy is measured as the ratio of total number of readings received at the sink to the total number of readings generated. We assume lossless aggregation, that is, no data is discarded. Examples include computing the minimum, maximum, as well as counting. In these scenarios, total accuracy is achieved when the sink can "calculate" an answer that involves one reading from every node per round.

Freshness is computed as the difference between the round a data item is generated and the round it is received at the sink. Overhead measures the communication complexity of the in-network aggregation algorithms.

5.2 Results

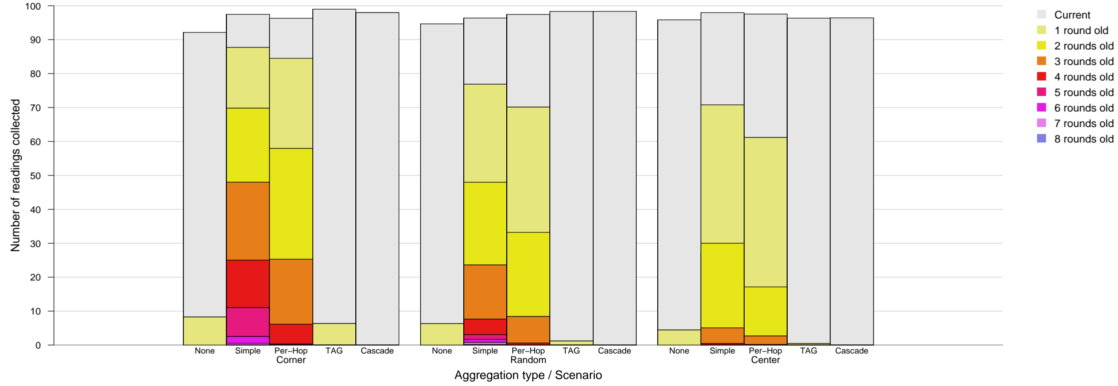

Figure 2: Data accuracy and freshness

Figure 2 shows the freshness and accuracy of the aggregation algorithms. *Periodic simple* is labeled as "Simple", *periodic per-hop* as "Per-hop", *TAG* refers to our implementation of TAG's aggregation mechanism, and "Cascade" represents our *cascading timers*. The sink placements evaluated were corner, random and center. The total number of readings collected (bar height) depicts accuracy and the bar divisions (shades), freshness. For comparison purposes, as baseline we use the no-aggregation strategy (labeled as "None").

In terms of data accuracy, we observe that there is not a big difference in performance when comparing the different aggregation mechanisms. No aggregation and *TAG* exhibit a high percentage of fresh data. *Cascading timers* practically eliminates old data. *Periodic simple* exhibits the largest range of data ages; this is because nodes simply send data periodically, thus it can take up to D periods for the readings to arrive in the worst case, where D is the diameter of the network.

Our implementation of *TAG* is a simple one. We do not calculate the maximum number of hops at runtime and so in some paths the real hop count is higher than our estimation of the network diameter. This leads to TAG having some 1 round old readings. For more accurate calculation we would have required extra message exchanges which we chose not to implement for this experiment.

Even though most data aggregation studies often do not account for sink placement, we observe from Figure 2 that sink placement has an impact on data freshness. Even for the no-aggregation case, where packets are forwarded immediately after they are received, placing the sink in the center yields fresher data.

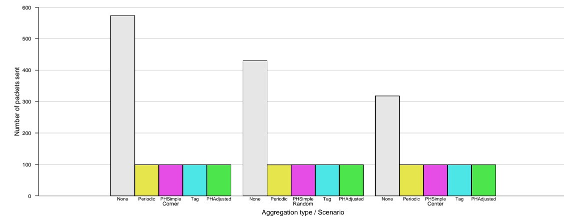

Figure 3: Number of data packets transmitted per round

From Table 1, which shows the energy consumed by the different algorithms, we observe that, for our experimental setup, energy consumption can be reduced to a third when data aggregation is used. Note that all aggregation schemes exhibit similar energy efficiency. These values will be affected by the choices of radio and MAC layers.

As an another way to compare the performance of the aggregation algorithms with respect to freshness, we introduce a metric that accounts for a data item's age. We call this metric *weighted accuracy*. The motivation behind measuring weighed accuracy lies in

	None	Simple	Per-Hop	TAG	Cascade
Corner sink	0.1418	0.0485	0.0483	0.0467	0.0421
Random sink	0.1302	0.0486	0.0484	0.0464	0.0419
Center sink	0.1134	0.0487	0.0487	0.0454	0.0404

Table 1: Energy consumed by the different in-network aggregation algorithms

the fact that while some applications are interested in historical data, others may only want the most up-to-date information. This is the case of real-time monitoring, where information sinks are only interested in the latest data sensed. For the latter type of applications, aggregation algorithms should not delay data delivery beyond a certain threshold.

To compute weighted accuracy, readings received in the same period they were produced have a weight of 1. Older readings are assigned an exponentially decaying weight: the older the reading, the less weight we assign to it. The expression for weighted accuracy is thus given by:

$$weighted_accuracy = \sum_{i \in I} r_i w^i$$

Where I is the set of ages of the readings, r_i is the number of readings of age i per period and w is the weight. Readings from the current period have an age of 0 and therefore a weight of 1.

The graphs in Figure 4 show the performance of in-network aggregation according to the weighted accuracy metric. As expected, no aggregation, *TAG* and *cascading timers* exhibit the best weighted accuracy. *Cascading timers* has an edge, specially when weights of old readings are low. *Periodic simple* and *periodic per-hop* perform poorly if old data has low weight. Their performance increases considerably as we assign higher weight to older information. Under corner sink placement, *periodic simple* and *periodic per-hop* start lower since it takes more periods for the data to arrive. The opposite is true when the sink is in the center.

The delay of a reading provides an alternate way to measure data freshness. We measure the average delay (in seconds) for a given reading as the time interval between when the reading is originally produced by a node until the sink processes all readings generated. Since we are generating results based on all readings we have to wait until all of them are gathered, hence the delay has to factor this in. Table 2 presents the average delay per reading for our experiments. *Cascading timers* performs the best since this is the very metric it tries to optimize.

	None	Simple	Per-Hop	TAG	Cascade
Corner sink	0.590	2.843	2.080	0.593	0.367
Random sink	0.565	2.047	1.544	0.418	0.286
Center sink	0.545	1.523	1.247	0.298	0.201

Table 2: Average reading delay

Cascading timers exhibits good performance on longer data collection periods as well. For example, in the case of a 10-second collection period using random sink placement, *cascading timers* achieves average delays similar to the ones in table 2, $0.28s$. No-aggregation's delays range around the 5.0 second mark, while *TAG* has an average of $4.23s$ delays. These results are due to the fact that no aggregation just sends at uniformly distributed times and *TAG* staggers transmissions as spread out as possible due to its method of dividing the period into slots. In both of these cases the delay will increase as the period increases.

In summary, as expected, our results show that in-network data aggregation can achieve considerable energy savings. *Periodic per-hop adjusted* aggregation (specifically our *cascading timers* algorithm) is able to maintain the same *freshness* and *accuracy* as when no aggregation is used. This is an impressive result considering the constraints imposed by periodically generated data. Furthermore, while *TAG* exhibits reasonable performance for shorter collection periods, *cascading timers* performs consistently well across a wide range of collection intervals. We study the behavior of the different aggregation timing models in more detail in Section 5.4.

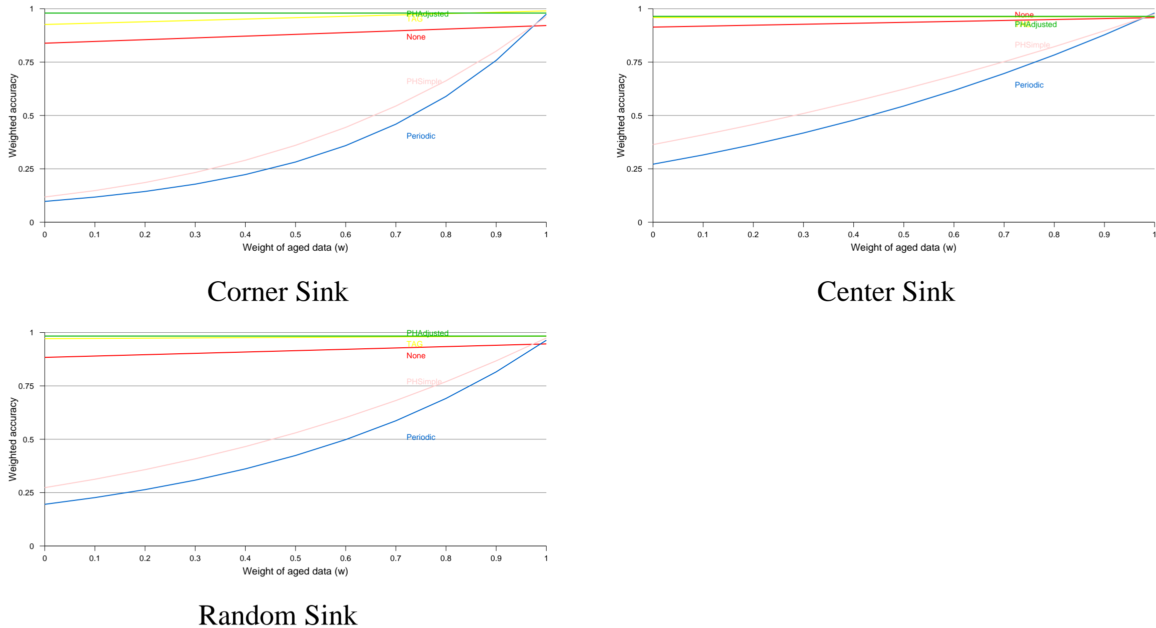

Figure 4: Weighted accuracy

5.3 Estimating the Single Hop Delay

As discussed in Section 2, *periodic per-hop adjusted* aggregation algorithms schedule a node's timeout, i.e., the time a node is due to "clock out" the current data aggregate, as a function of the node's position in the distribution tree. More specifically, in our *cascading timers* aggregation, timeout is a function of the *single hop delay* or shd, an estimate of the time a packet is expected to take to traverse one hop in the data collection network, including processing time. Given shd, we can then estimate the time it takes for a packet to traverse the path from the source to the sink by multiplying shd by the number of hops traversed.

In this section, we study how shd impacts the performance of *cascading timers*. Note that shd depends on current network conditions such as load, channel quality, etc. If the network is heavily loaded, packets might take longer to traverse one hop. This may result in nodes timing out and readings getting lost. The performance of *periodic simple* and *periodic per-hop* aggregation schemes will also be impacted when under heavy load.

Recall that, in order to avoid collisions, we stagger node transmission in relation to one another using a small random interval. Thus, shd has basically a deterministic- as well as a non-deterministic component. While propagation- and transmission delay make up shd's deterministic component, the staggering interval and queuing delay are responsible for shd's non-determinism. shd can then be computed using the expression in Equation 4,

$$shd = sd + td + pd + qpd \quad (4)$$

where, sd is the staggering delay of the packet, td and pd correspond to the transmission and propagation delays respectively and qpd accounts for both queuing and processing delays.

Cascading timers uses max_shd as given by Equation 5, which is obtained by using the maximum possible staggering delay and an upper bound (Up) of the rest of the components. The maximum sd is chosen by us. The propagation and transmission delays are a function of the network architecture and the maximum size of the packets. The processing and queuing in this scenario are almost negligible since as soon as packets arrive they are aggregated and the processing for this is minimal.

$$max_shd = Max(sd) + Up(qpd + td + pd) \quad (5)$$

Figure 5 presents *cascading timers*' weighted accuracy for different values of shd ranging from 0.01 to 0.3. The weight used was 0.5. For good performance we chose the maximum staggering interval to be 0.05. This is a good enough compromise between large enough sd to avoid collisions and small enough to allow the last readings to get to the sink within the collection interval. Our original experiments used an sd of 0.03. When studying the values of the shd and the sd later we found the value of 0.05 to be slightly more optimal for our scenario and density.

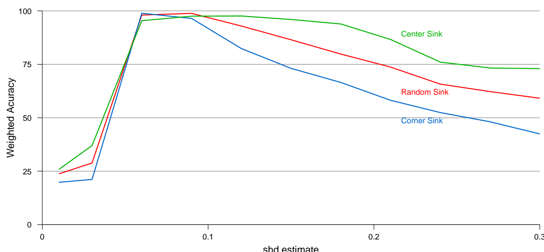

Figure 5: Weighted accuracy for different shd values

For shd smaller than the maximum random staggering interval (left-hand portion of the graph), we observe that the algorithm's performance in terms of accuracy deteriorates considerably. This is expected: data is aggregated and sent even before children nodes try to send since their stagger timer hasn't expired. When shd is slightly higher than the staggering interval, performance is maximized. This is because child nodes will have enough time to send in their data items to the corresponding parent.

For large values of shd the algorithm's weighted accuracy starts to drop. This is expected since the time it takes for readings from farther nodes to reach the sink using large shd ends up being longer than the collection period, and hence they are worth less according to weighted accuracy.

Setting the staggering interval is an important tradeoff. While large staggering intervals are effective in avoiding collisions, they yield higher delays. As previously pointed out, the value of sd also depends on network density. In dense networks larger staggering intervals are needed to avoid collisions. Another important consideration is the type of medium access control (MAC) protocol employed. Contention-based MAC protocols, such as CSMA (which is what we use in our simulations), are prone to collisions and node transmissions need to be staggered in time. However, other types of MAC, such as scheduled access protocols, are collision free and thus do not require staggering of node transmissions, probably at the expense of delay.

We also measured the values of the real shd over the course of our simulations and observed very small variations. This is expected, since traffic flows over the same data collection tree and the offered load is essentially constant.

5.4 Data Collection Interval

This section discusses the impact of the data collection interval's length on the performance of periodic aggregation timing models. Intuitively, the smaller the collection period, the more critical the timing model used by in-network aggregation.

Periodic simple and *periodic per-hop* have no timer organization, and send data randomly during the data collection interval. This basically creates a data path where readings from nodes will take approximately as many collection intervals as hops to reach the sink. The longer the collection interval, the longer delays these data paths will incur.

In the case of *no-aggregation*, data is sampled at random times within the interval and then sent. Data reaches the sink as quickly as possible since it is forwarded with no waiting. Following our *cascading timers* model, data is transmitted near the end of the collection period depending on the position in the aggregation tree. This is independent of the period's length. All the data arrives with a small delay at the sink right before the period ends.

We define two different performance metrics associated with the delay a reading takes to reach the

sink. The first definition focuses on how long the reading takes to reach the sink from the time it is sampled. We call this metric the *sample-to-sink* delay. The second metric looks at the time it takes from when a reading is sampled until **all** readings are received by the sink (*sample-to-all*). While the former metric targets applications that are mostly interested in the latest readings, the latter is useful when the application performs some computation which requires all readings.

As previously discussed, according to the *sample-to-sink* metric, *periodic simple* and *periodic per-hop* aggregation will not perform well. Furthermore, in larger networks, the extra hops packets take to reach the sink will add to the delay. *No-aggregation* will incur the smallest delay possible, having packets hop their way to the sink. Assuming the staggering delay is uniformly distributed from 0 to sd, packets under *no-aggregation* will take on average $(\frac{sd}{2} + C) * hops$, where $hops$ is the number of hops data has to traverse and C accounts for transmission, propagation, and queuing delays.

Cascading timers will exhibit a *sample-to-sink* performance of almost double that of *no-aggregation* since the shd used is based on the maximum, and not the average staggering delay (sd) like *no-aggregation*. This means that the delay will be $(sd + C) * hops$. Double the *sample-to-sink* delay might seem like a big disadvantage; however, given that the staggering interval is small compared to the data generation period, the *sample-to-sink* difference between *no aggregation* and *cascading timers* is relatively small.

The *sample-to-all*($StA()$) delay provides a metric for the overall freshness of the data the sink receives. In *no aggregation* data gets to the sink throughout the generation period. Thus, the average time a reading will have to sit idle at the sink is $\frac{period_length}{2}$, that is, from when it arrives at the sink, until it is tallied at the period's end. *Cascading timers*, on the other hand, sets nodes to transmit at the end of the period, so the time the readings have to wait at the sink is always close to 0 (see Figure 1).

$$StA(NoAgg) = n((\frac{sd}{2} + C)D + \frac{pl}{2}) \quad (6)$$

$$StA(Cascading) = n((sd + C)D) \quad (7)$$

Equation 6 and 7 give us the average *sample-to-all* delay over all nodes (n) for *no-aggregation* and *cascading timers* respectively. D represents the average node depth and pl denotes the period length. As shown in Equation 8, for *cascading timers* to have a lower overall *sample-to-all* delay than *no-aggregation* the period length (pl) has to be larger than the maximum staggering delay (sd) times the average node depth (D).

$$\begin{aligned} StA(NoAgg) &> StA(Cascading) \quad (8) \\ n((\frac{sd}{2} + C)D + \frac{pl}{2}) &> n((sd + C)D) \\ (\frac{sd}{2})D + \frac{pl}{2} &> (sd)D \\ pl &> (sd)D \end{aligned}$$

In the worst case scenario, i.e. a line topology, the average node depth will be $\frac{n}{2}$. For any large sample period, pl will be larger that $\frac{sd*n}{2}$. Using our simulation parameters, 100 nodes and a 0.03 sd, $StA(Cascading)$ would be smaller than $StA(NoAgg)$ for any period length greater than 1.5 seconds. Again we note that this is the worst case scenario. Using *no aggregation* under that scenario will probably incur collision problems due to the amount of traffic at the nodes close to the sink.

For the other topology extreme, i.e. a single hop tree with an average depth of 1, it is easy to see that *cascading timers* will have lower delays for all values of the period length (greater than the maximum staggering delay). The main drawback would be that, if there are many nodes, the traffic at the end of the period may cause too many collisions. This could be potentially solved by increasing the staggering delay. If the staggering delay is increased to match the period length then $StA(NoAgg) = StA(Cascading)$.

6 Packet Losses

As previously discussed, timing models are critical for the performance of in-network aggregation. In-

deed, efficient aggregation may result in packets carrying several readings. Packet losses can, thus, considerably degrade the accuracy and timeliness of data aggregation. In this section, we study the effect of packet losses when collecting and aggregating periodic data. We also propose three different mechanisms to handle loss and evaluate their performance.

Since we target applications that generate data periodically, we avoid recovery mechanisms that introduce delay. If data recovery and retransmission take too long, nodes would already be producing the readings for the following round, error recovery would interfere with the propagation of new data. Take for example a traditional error recovery scheme where negative acknowledgments (NACKs) are used. Under the proposed aggregation protocols, every node knows how many children it has. Therefore, if the node times out waiting to hear from some of its children, it could then send a NACK to request them to retransmit their data for the current round. However, this would interfere with the next round of data. Since target application scenarios don't require perfect data delivery, we are willing to sacrifice accuracy in favor of freshness.

We use a proactive approach to error recovery along the lines of error correcting schemes such as Forward Error Correction (FEC) [10]. Since the typical data items produced in the target application scenarios are generally small (a few bytes), we decided to use a very simple form of error correction mechanism, namely send a packet multiple times. In future work, we plan to investigate other, more sophisticated error correcting codes that can pay off in the case of more complex data.

6.1 Handling Losses

Our loss model is simple: drops are independent and the loss probability p is fixed for all links and is kept constant throughout the simulation. A packet is successfully transmitted with probability $(1-p)$. While this is a simple scheme, it can be used to get a general picture. We will look at spatial and temporal correlated losses in future work.

As mentioned earlier, we deal with losses by proactively sending data multiple times. This strategy of course creates a tradeoff between consuming energy to send redundant packets and increasing the probability of delivery. Redundant transmission mechanism might not be suitable for all scenarios; in particular, when data is generated sporadically (rather than periodically) normal error recovery using acknowledgements and retransmissions will likely be more cost-effective.

We use three different redundant transmission schemes, namely *double-send*, *max-send*, and *adaptive-send*. While *double-send*, as its name implies, sends every packet twice, *max-send* sends every packet as many times as readings are aggregated in the packet. This requires the protocol to include an "aggregation counter" in the data packet. The reasoning behind this strategy is that packets carrying more readings are more valuable and we therefore want to increase their chances of getting through. *Adaptive-send* is a more complex algorithm, whose goal is to achieve certain delivery guarantees (expressed by number of acceptable losses) for a given loss rate. Equation 9 describes the relationship between the number of acceptable losses l and the drop probability p, where a and t are the number of aggregated readings in a packet and the number of transmissions, respectively. The number of transmissions to achieve l under loss probability p is then given by Equation 10.

$$l = a \cdot p^t \quad (9)$$

$$t = \log_p \left(\frac{l}{a} \right) \quad (10)$$

Note that, since we aim for hop-by-hop guarantees, these equations apply to a single link. In other words, if $l = 0.05$, *adaptive send* will perform the necessary number of retransmissions to guarantee a 95% delivery guarantee over a particular link given that link's loss rate. The overall network's delivery guarantee may be slightly different (higher or lower) depending on the interdependencies between the aggregates in the packets as they flow towards the sink. As our simulation results show, the overall reliability achieved by *adaptive-send* (shown in Figure 6 as data accuracy) is typically very close to $1 - l$.

In a real network, the loss probability can be either known a priori (from previous experience operating the network) or estimated over time. Underestimating the loss rate yields delivery rates lower than required; conversely, loss rate overestimation results in more redundant packets.

6.2 Results

We repeated the simulation experiments described in Section 5 to study the performance of the three proposed loss handling strategies when employed by *cascading timers* aggregation. Loss probabilities range from 0.05 to 0.5 in steps of 0.05. We kept all other parameters the same. In the case of *adaptive-send*, the acceptable loss rate was set to 0.05 (which implies relatively high delivery guarantees). We present the results for different sink placement scenarios in Figures 6 and 7. For comparison purposes, we plot results for *cascading timers* aggregation and no aggregation when they employ no error recovery.

The results obtained for the different algorithms bring out some interesting points. Figure 6 shows data accuracy (percentage of readings received by the sink) achieved by the different mechanisms under different loss conditions. These results confirm our expectations: *adaptive-send* is the best performer with close to perfect accuracy even under high loss. *Max-send* delivers more than 50% of the readings, while *double-send* starts with a very good delivery but quickly degrades under high loss conditions.

At low loss rates we notice that *double-send* performs slightly better than *max-send* and *adaptive-send*. This is attributed to the fact that it sends every packet twice, including packets with one reading, irrespective of loss rate. *Adaptive-send* and *max-send* send packets with one reading only once. When packets with one reading get dropped, they are not recovered. Packets that would have included the dropped reading will now be transmitted fewer times, decreasing their chance to get to their next hop, which decreases the next packet and so on. After a certain drop probability , *adaptive-send* starts sending packets with single readings twice. This seems to indicate that taking good care of packets, even when they only have one reading collected is important.

It is interesting to note that under the loss model and scenarios used, drops have similar impact on data collection with and without aggregation. We observe that the curves for *None* and *cascading timers* (without error recovery) look very similar. Data accuracy decays very quickly in both cases. We plan to study this phenomenon further using different topologies and network diameters.

To evaluate the overhead incurred by these error recovery schemes, we plot total number of packets sent in Figure 7. *Double-send* sends approximately 200 packets per round for all loss rates as it always sends a packet twice. *Cascading timers* (without error recovery) also sends a constant number of packets (around 100 in this case) since it does not adjust its behavior for different loss conditions. At the higher error levels we notice that 100 packets are not sent per round. This is due to the fact that because of packet drops, at tree construction time some nodes don't become part of the tree.

It is somehow expected that *none* and *max-send* would exhibit similar behavior since *max-send* should transmit approximately the same number of packets as *none*. However, this is not the case: for *none*, the loss of packet means that nodes in the path to the sink will now transmit one less packet. In *max-send*, if all the transmissions of a packet are lost, subsequent packets will be transmitted fewer times since they will have less readings. The decline of *max-send* is slower than *none* because all of the transmissions of a packet have to be lost for subsequent nodes to transmit less. *Adaptive-send* increases the number of packets sent as loss probability increases since the number of transmissions is a function of the number of aggregated values and the drop rate.

In some scenarios, rather than maximizing data

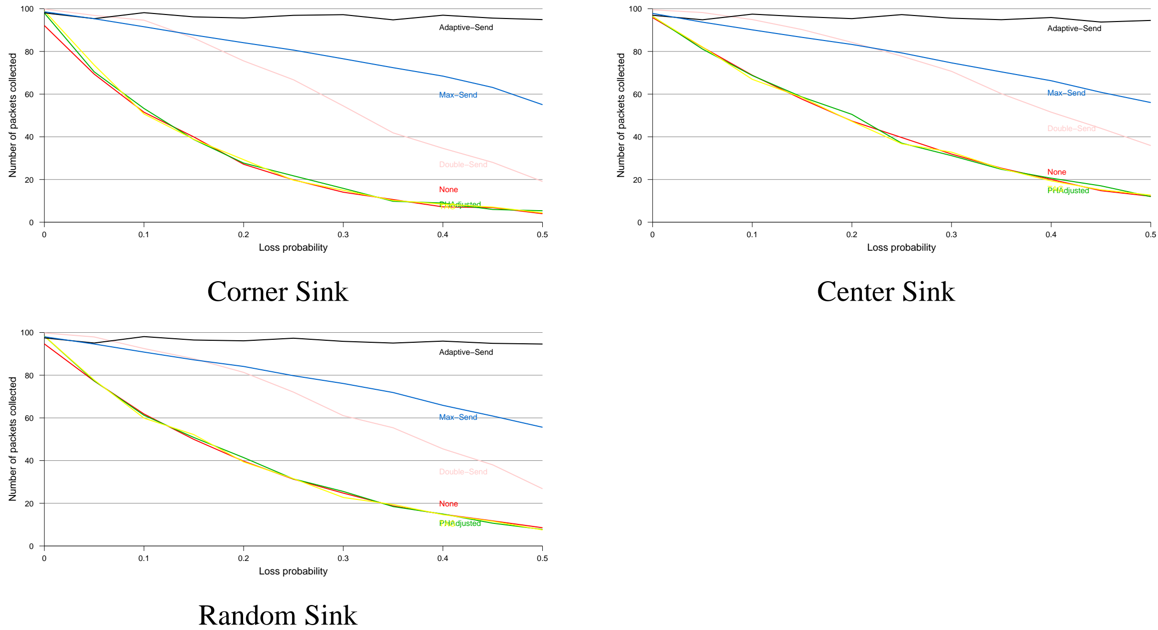

Figure 6: Data accuracy under loss

accuracy, it may be more attractive to strike a balance between number of packets sent (or energy consumed) and accuracy. Figure 8 plots the number of packets sent per reading collected. The optimal case of course is one packet sent per reading. *Cascading timers* lets us achieve that goal when there are no losses. *Adaptive-send*, which is implemented atop *cascading timers*, yields good performance under all conditions due to its adaptive nature.

A corner sink means routes are longer, *none* and *max-send* due to this, specially at low drop rates. It is also the case that for the longer paths the single reading packets of *none* have a greater probability of getting dropped, hence the poor performance in packets per reading.

7 Related Work

Protocols for sensor networks have sparked considerable interest in the network research community. In this context, data aggregation rose as a technique for improving sensor network protocols' energy efficiency. We briefly describe some previous and ongoing research efforts in order to put our work in perspective.

Directed diffusion [1] has been proposed as a data gathering protocol for sensor networks. It targets the monitoring of events which are typically sensed only by a few nodes. An example scenario is tracking animal herds in a given geographic region. Diffusion's communication paradigm is based on information sinks broadcasting requests, or *interests*, for relevant data. Nodes producing relevant information respond and *data paths* are formed. Data is aggregated when a node is part of various data paths.

Diffusion's aggregation is based on a report gradient, which defines how many reports to send per time unit. Therefore, according to our classification, diffusion falls in the *periodic simple* category. Every node can potentially perform aggregation; however, nodes in the shortest path from information sources to the sinks do most of it. Diffusion adapts well to node failures by keeping state of the interest throughout the network. When a path fails the neighboring nodes remember alternate paths. Some of their more recent work addresses the impact of node density in this type of data collection scenario [11].

eScan [3] is an energy monitoring scheme that col-

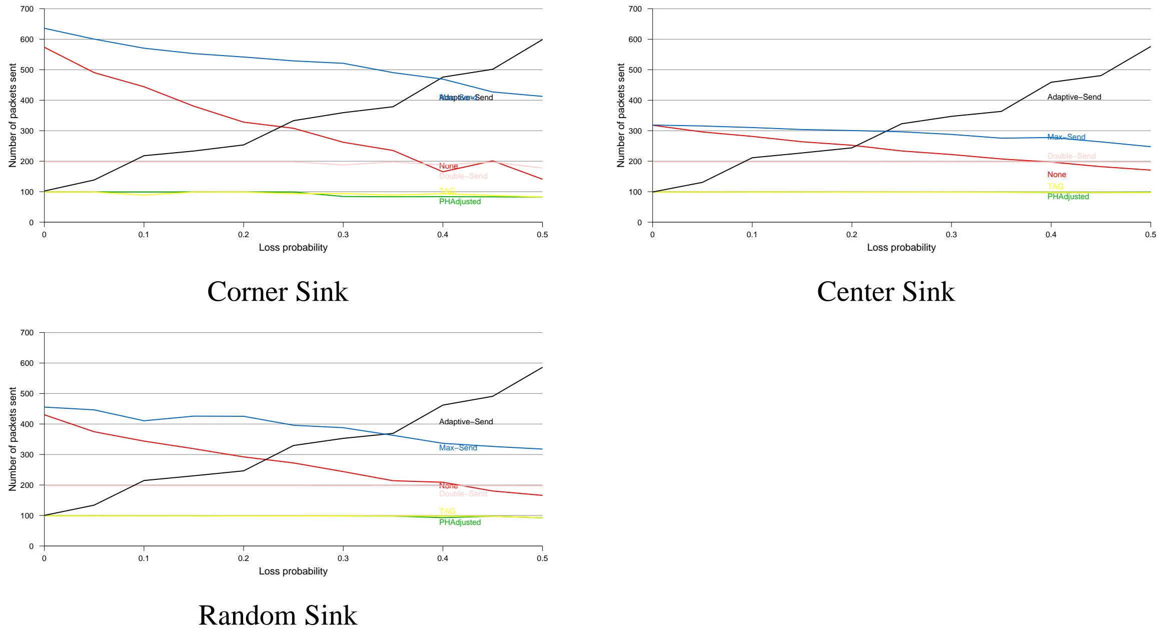

Figure 7: Packets sent under loss

lects energy readings from every participating node. Their scenario is somewhat similar to the ones we target, i.e., every node maintains an energy value that is reported to a collection sink at the edge of the network. However, rather than generate periodic reports, new data is reported only when the energy of a node changes beyond a certain threshold. Aggregation is performed as data flows to the sink by merging reports of similar energy values into *energy range polygons*.

The initial eScan work does not handle latency in propagating data; they also assume a perfect MAC layer and instead of using time they use data generation events to drive their simulations. Rather than being an alternative to eScan, our aggregation techniques can be incorporated by eScan to, for example, improve data freshness.

SPIN [4], Sensor Protocols for Information via Negotiation, is a protocol for data collection and dissemination. In SPIN, all nodes have pieces of named information that they want to send to the rest of the nodes. Data transfers are first negotiated based on the names of items. Only requested items are exchanged. This avoids the cost of sending the data needlessly but incurs the overhead of engaging in the negotiation phase. Note that SPIN's communication model is based on a gossip-style approach. The resulting protocol is very similar to NNTP [12] for propagation of news over the Internet. Essentially, it uses point-to-point communication among pairs of nodes to eventually convey data to all interested participants. SPIN does not really use an explicit aggregation mechanism; aggregation is performed implicitly during initial negotiation between two nodes using the meta-data to decide whether actual data will be exchanged.

TAG [2], or Tiny AGreggation, is a sensor network querying system. It employs a SQL-like syntax and uses aggregation as the query is processed within the network. When a query involves an *epoch*, requiring readings to be collected periodically, TAG uses the *periodic adjusted* aggregation approach. It subdivides the epoch into slots. The length of a slot is given by the epoch length divided by n, the maximum number of hops separating data generation nodes from the sink. Following *periodic adjusted* aggregation operation, slots are assigned to nodes in decreasing order, $n, n-1, n-2, \ldots$, as the query prop-

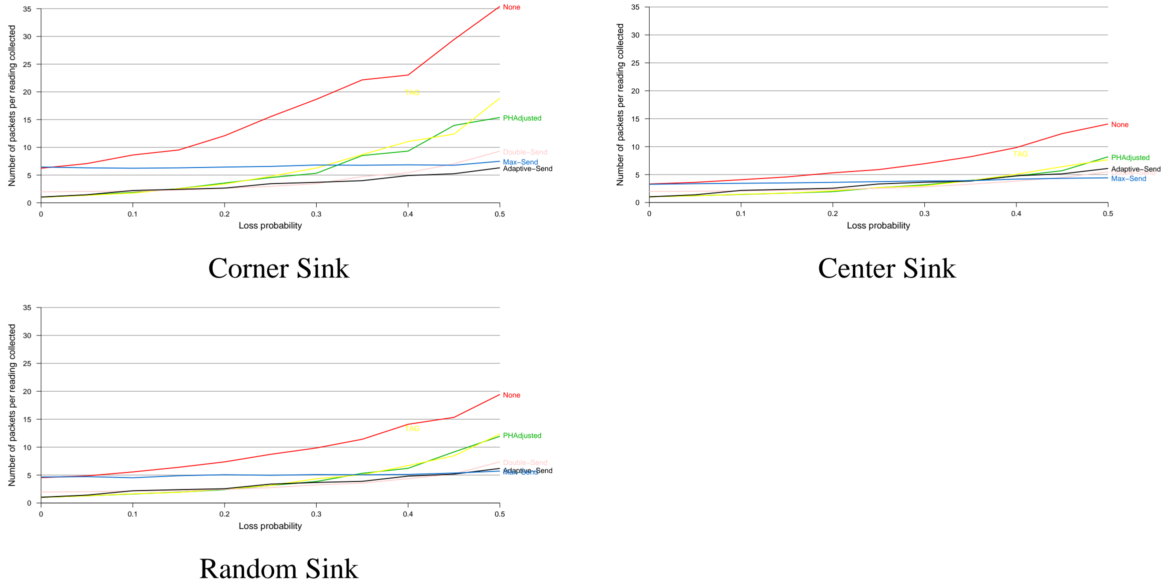

Figure 8: Packets sent per reading collected

agates through the network. Nodes transmit in their slot, hence, the out-most nodes will transmit first and nodes closest to the sink, last. As in any time-slotted mechanism, clock synchronization among nodes is required so that nodes transmit in their designated slots. TAG also takes advantage of time slotting to switch idle nodes' radios off.

Convergecasting [7] performs aggregation as it collects data periodically from all nodes to a single sink. Like TAG, its data aggregation mechanism also falls in the *periodic adjusted* category. It assigns aggregation slots as the query percolates the sensor network, trying to assign nodes to different slots in order to avoid collisions. Once the algorithm finishes assigning slots, that is, when the query setup reaches the edge of the network, the order of the slots is inverted to reflect a data collection tree. A similar concept is used by the work reported in [13].

8 Conclusions and Future Work

This paper explored in-network aggregation as a power-efficient mechanism for collecting data in wireless sensor networks. Our focus was on applications where a large number of nodes produce data periodically which is consumed by fewer sink nodes. Such communication model is typical of monitoring scenarios, one key application of sensor networks.

Using simple analytical models, we present a trade-off analysis of in-network data aggregation. Through simulations, we evaluate the performance of different in-network aggregation algorithms, including our own *cascading timers*, and characterize the tradeoffs between energy efficiency, data accuracy and freshness. Our results show that timing, i.e., how long a node waits to receive data from its children (downstream nodes in respect to the information sink) before forwarding data onto the next hop (toward the sink) plays a crucial role in the performance of aggregation algorithms in the context of periodic data generation. By carefully selecting when to aggregate and forward data, we achieved considerable savings (as much as 5 times less traffic) while maintaining data freshness and accuracy.

Finally, in order to perform well under packet loss conditions, we developed three different proactive error recovery techniques which are suitable to periodic data generation (as they incur no additional delay). They are essentially based on pro-actively transmitting packets multiple times. Simulations showed that the proposed techniques were able to

maintain high accuracy even under high loss conditions. Among the proposed mechanisms, *adaptive-send*, which adjusts the number of times a packet is sent based on the number of readings aggregated in the packet and an estimate of the loss probability, yielded the best performance.

As future work, we plan to use more sophisticated loss models to evaluate our recovery mechanisms under different loss conditions. We will also develop techniques to handle node failures. We also plan to investigate aggregation algorithms that target different scenarios. For example, instead of periodic data generation, explore scenarios in which data is reported only when it changes significantly. Another direction is to explore cluster-based aggregation, where clusters are formed based on a set of constraints and then data is aggregated within clusters.

References

[1] C. Intanagonwiwat, R. Govindan and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*. ACM, August 2000.

[2] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TAG: a tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the Symposium on Operating Systems Design and Implementation, OSDI*, December 2002.

[3] J. Zhao, R. Govindan and D. Estrin, "Residual energy scans for monitoring wireless sensor networks," in *Proceedings of the IEEE Wilress Communications and Networking Conference (WCNC'02)*, March 2002, pp. 17–21.

[4] W. Heinzelman, J. Kulik and H. Balakrishnan, "Adaptive protocols for information dissemination in wireless sensor networks," in *Proceedins of the International Conference on Mobile Computing and Networking (MobiCom)*, August 1999.

[5] I. Solis and K. Obraczka, "The impact of timing in data aggregation for sensor networks," *Proceedings of the IEEE International Conference on Communications (ICC)*, June 2004.

[6] J. Elson, L. Girod and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[7] L. Schwiebert V. Annamalai, S.K.S Gupta, "On tree-based convergecasting in wireless sensor networks," in *Proceedings of the IEEE Wilress Communications and Networking Conference (WCNC'03)*, March 2003.

[8] VINT, "The Network Simulator NS-2," http://www.isi.edu/nsnam, November 2001.

[9] I. Solis and K. Obraczka, "FLIP: A flexible interconnection protocol for heterogeneous internetworking," *ACM/Kluwer Mobile Networking and Applications (MONET) Special on Integration of Heterogeneous Wireless Technologies*, August 2004.

[10] R.E. Blahut, *Theory and Practice of Error Control Codes*, Addison Wesley, 1984.

[11] C. Intanagonwiwat, D. Estrin, R. Govindan and J. Heidemann, "Impact of Network Density on Data Aggregation in Wireless Sensor Networks," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, July 2002.

[12] B. Kantor and P. Lapsley, "Network News Transfer Protocol," IETF RFC-977, February 1986.

[13] C. Florens and R. McEliece, "Packet distribution algorithms for sensor networks," in *Proceedings of IEEE INFOCOM 2003*, April 2003.