DATA AGGREGATION

Yiannis Giannakopoulos
University of Athens, Department of Informatics
Athens, Greece
ygiannak@di.uoa.gr

Christos Koninis
Research Academic Computer Technology Institute
Patras, Greece
koninis@cti.gr

1. Introduction

In Wireless sensor networks, a very common task is that of monitoring a large area in regards to some physical sensed value such as temperature. Usually the end user wants to examine and analyze this information, so they are collect in a central point we call the *sink* node. Due to the large amount of data transmitted as the scale of the network increases, it is important to combine several sensor readings in intermediate nodes along the way towards the requester and therefore to conserve energy. This process is known as data aggregation. However due to the nature of the wireless medium the communication in a WSN is error-prone and the aggregation techniques used must be aware and resilient to errors in message transmission.

2. Algorithm techniques

In this section, we will very briefly describe the fundamental algorithmics, underlying the data aggregation techniques deployed in our sensor network implementation. For a more rigorous and in-depth exposition, the reader is refered to many well-written surveys in the area of in-network data aggregation, e.g. [BCG11a].

Usually, in wireless sensor network scenarios, single individual values are mostly not of great relevance. In fact, users are more interested in the quick extraction of succinct and useful synopses about a large portion of the underlying observation set. Consider, for example, the case of a temperature sensor network. We would like to be able to continuously monitor the entire infrastructure and efficiently answer queries such as “what was the average temperature over the entire terrain during the last 12 hours?”, or “are there any specific clusters that have reached dangerously high temperatures?”.

Trying to collect all data monitored by the sensor nodes would be unrealistic in terms of bandwidth limits, power consumption and communication intensity. So, the canonical approach is to compute statistical *aggregates*, such as max, min, average, quantiles, heavy hitters, etc., that can compactly summarise the distribution of the underlying data. Furthermore, since this information is to be extracted and combined across multiple locations and devices repeatedly and in a dynamic way, *in-network aggregation* schemes [MFHH02] that efficiently merge and quickly update partial information to include new observations, must be developed. Also notice that, computing aggregates instead of reporting exact observations, can leverage the effect of packet losses and, generally, network failures, which are both common and extremely crucial phenomena in wireless networks of tiny artifacts.

It is evident that there are two levels of computation and aggregation in distributed settings. At a low level, each sensor observes a stream of data and needs to efficiently extract and maintain information about it. This is essentially the problem of traditional, centralized streaming which has been extensively studied during the last two decades [AMS99], [FM85], [MP78]. Aggregation is considered with respect to the individual values comprising the data stream, into a concise summary.

At a higher level, all remote sites should coordinate to combine these partial information computed from each device. Here, aggregation is considered with respect to this merging process of creating summaries that describe the entire infrastructure. It should be clear that this in-network aggregation model generalizes traditional streaming, in the way that a single data stream can be seen as values distributed along a linear-chain topology [GK04, section 1.3]. Efficient algorithms for distributed computation, that do not make stringent assumptions about the infrastructure topology, can be readily used for classical streaming problems.

In the setting of massive data stream computations addressed in this book, data are observed or produced at a far higher rate than can be locally stored or, sometimes, even observed. Their delivery to aggregation or elaboration points requires an amount of in-network communication that far exceeds the power capabilities of sensing devices. Furthermore, the computational complexity of exactly evaluating the statistical aggregates of interest is unrealistic [Mut10]. Considering further that we are interested in scenarios where these functions are performed by tiny devices with extremely limited resources, it is natural to ask for algorithmic solutions and data structures that require storage and update times that are sublinear and often (poly)logarithmic with respect to the size of the observed data, further imposing similar constraints on the amount of communication involving each device.

For the purposes of this chapter, we will concentrate mainly on the MAX, AVG and DCOUNT (i.e. number of distinct elements) aggregates, though the underlined algorithmic fundamentals apply to the general theory of data stream aggregation.

2.1 In-sensor data aggregation

Assume that a sensor is observing a continuous stream $S = (a_1, a_2, \ldots, a_n)$ of real valued elements $a_t \in \Re$, indexed by discrete time t (for our needs here, we only assume a *time series model*, see [Mut05]). At any time t, we need to be able to succinctly and efficiently answer statistical aggregate queries regarding the underlying stream up to time t, i.e. $S_t = (a_1, a_2, \ldots, a_t)$. As we've mentioned before, we ideally want to perform that in (poly)logarithmic space and time, respecting the computational and storage capabilities of our tiny devices. We achieve that by maintaining auxiliary data structures called *summaries*[1] which are clever and succinct synopses of the underlying data set observed so far.

The most important operation of a summary is that of *merging* new elements into it, in order to produce efficiently a new synopsis, describing the new, larger stream. More formally, we need a function F such that if $\mathbf{Sk}(S_t)$ is a summary instance describing an underlying stream S_t, then

$$\mathbf{Sk}(S_{t+1}) = F(\mathbf{Sk}(S_t), a_{t+1}).$$

For example, for the MAX and AVG aggregates it is trivial to see that can be efficiently computed by maintaining (recursively) summaries with $\mathbf{Sk}_{\texttt{MAX}}(S_1) = a_1$, $F_{\texttt{MAX}}(x, y) = \max\{x, y\}$ and $\mathbf{Sk}_{\texttt{AVG}}(S_1) = (a_1, 1)$, $F_{\texttt{AVG}}((x_1, x_2), y) = (x_1 + y, x_2 + 1)$, respectively, where the actual mean value can be extracted by the AVG summary by simply computing $\texttt{AVG}(S) = \frac{x}{y}$, where $\mathbf{Sk}_{\texttt{AVG}}(S) = (x, y)$.

Such a simple structure of an aggregate summary is not always the case. For example, the DCOUNT aggregate, cannot be computed with *exact* precision by using only by a summary of size $O(1)$ (with respect to the size N of the entire observed set). Such "inherently difficult" aggregates are called *holistic*, in contrast to the *algebraic* ones like the MAX and AVG discussed before (for more details in this important distinction, see [GCB$^+$97], [MFHH02]). Fortunately, we can overcome these linear space requirements that would result to some sensor nodes having an unacceptable memory load of maintaining $\Omega(n)$ values, by trying to devise summaries with $O(polylog(n))$ sizes that are able to answer our queries *approximately*, within a sufficient accuracy of ε.

In particular, for the DCOUNT aggregate, we can use the *FM sketch* summary (introduced by Flajolet and Martin, see [FM85]). Each sketch is a vector of m entries, each entry being a bitmap of length $k = O(\log n)$. Considered the s-th bitmap of the sketch. Every observed value is hashed onto the bitmap bits using a (independently chosen) hash function $h_s(\cdot) : \{1, \ldots, n\} \to \{0, \ldots, \log_2 n - 1\}$, such that the probability of hashing onto the h-th bit is 2^{-h}. The bit under consideration is set to 1 if it was 0. After processing the stream, let r_s denote the position of the least significant bit that is still 0 in the s-th bitmap: it is easy to see that r_s is a good estimator for $\log_2 \texttt{DCOUNT}$, the logarithm of the number of distinct pairs observed. To improve accuracy, we consider $\frac{1}{m}\sum_{s=1}^{m} r_s$ as an estimator

[1]The terms *sketch* and *synopsis* are also widely used.

of $\log_2$ DCOUNT, where $m = O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$. This variant on the basic FM sketch algorithm is called *Probabilistic Counting with Stochastic Averaging (PCSA)* ([CMZ06], [AMS99]).

In [DF03]a variant of the *PCSA algorithm* is presented that reduces the size of the accumulation synopsis from $\log n$ to $\log\log N$. However, the standard error is increased from $0.78/\sqrt{k}$ to $1.30/\sqrt{k}$, where k is the number of bitmaps. Which means that the LogLog Counting is less accurate but the space complexity is improved considerably. The algorithm differs from the PCSA in two aspects. The first is that the *maximum bit set to 1* is maintained, in contrast to the PSCA where the position of the *least significant 0-bit* is maintained, and the second is the function used to compute the estimate.

2.2 In-network data aggregation

The predominant assumption in sensor network computation is that our devices form a *tree*, at the root of which is a special device called the *base-station* which is responsible for answering the aggregate queries. This is done in roughly two phases: first, the base station distributes the query to all the nodes and then the needed information is pulled back to the root, in most cases using a fair amount of sophisticated in-network aggregation by the intermediate nodes along the way. Every node merges (aggregates) information about its own stream with information received from its children and transmits this partial information about the distribution of the entire underlying data set to its parent.

Although such single-path, tree-based aggregation models capture the essence of sensor network communications and are straightforward to analyze, they come with a huge disadvantage: in case of a packet loss or a node failure, an entire subtree is disconnected and the aggregating procedure is rendered useless, especially when this node is close to the base station. In general, sensor networks are far from reliable with respect to such events, since usually many inexpensive devices are cast over a wide unattended or even hostile terrain. In addition, wireless networks are prone to environmental interferences, signal strength fading, packet collisions, etc [ZGE03], [CHL$^+$09].We can deal with lossy networks by using multi-path routing aggregation where each node may forward its information to many neighbors, reaching the base station via more than one path. This is a generic idea and can be approached in many ways, for example using directed diffusion or other gossiping techniques.

Formally we have a set of *nodes* (sites, sensors) $I = \{1, 2, \ldots, |I|\}$, indexed by i, each of which is observing a data stream S_i. For our exposition it is enough to think of the individual streams, simply as being multisets drawn from a (finite) universe U, $|U| = N$, i.e. $S_i = \{a_{i\,1}, a_{i\,2}, \ldots, a_{i\,m_i}\}$, $a_{i\,j} \in U$. Following the influential exposition of [MFHH02], for an aggregate query Q to be able to be computed in-network, a summary $\mathbf{Sk}$ must be maintained, supporting three fundamental operations:

- Initialization (I): Used by each node i to create an instance $\mathbf{Sk}_{S_i}$[2] of the summary to describe its own stream, i.e. $I(i) = \mathbf{Sk}_{S_i}$.
- Aggregation (F): Used in intermediate steps of the aggregation procedure to merge two instances $\mathbf{Sk}_{M_1}$, $\mathbf{Sk}_{M_2}$ describing underlying multisets M_1 and M_2, respectively, into a single summary instance $\mathbf{Sk}_{M_1 \cup M_2}$ describing their union, i.e.
$$\mathbf{Sk}_{M_1 \cup M_2} = F(\mathbf{Sk}_{M_1}, \mathbf{Sk}_{M_2}). \tag{1}$$
- Evaluation (E): Used at the final step of the aggregation procedure (e.g. at the base-station) to extract the answer to query Q from an instance $\mathbf{Sk}_{\cup_i S_i}$ describing our entire data set, i.e. $Q(\cup_i S_i) = E(\mathbf{Sk}_{\cup_i S_i})$.

An aggregate query Q is called *duplicate-insensitive* if it is not affected by the insertion of multiple occurrences of the same element in the underlying data set. Formally, if for every multiset M, $Q(M) = Q(\bar{M})$, where $\bar{M}$ is the simple set induced by M (if we delete multiple occurrences). Duplicate-insensitive queries include MIN, MAX and DCOUNT, while notable duplicate-sensitive queries are COUNT, SUM and the ϕ-quantiles. This notion naturally extends to summaries, where a summary will be called duplicate-insensitive if for every multisets M, $M' \subseteq M$, its aggregation function F satisfies the property

$$F(\mathbf{Sk}_M, \mathbf{Sk}_{M'}) = \mathbf{Sk}_M, \tag{2}$$

[2]Sometimes we will use subscript notation $\mathbf{Sk}_M$ if we want to give emphasis on the underlying multiset M our summary (instance) describes.

for all possible instances of the summaries describing M and M'. Duplicate sensitivity plays an important role when we design multi-path aggregation schemes. In such settings, it is possible for the very same data point in the stream of some sensor, to be aggregated many times along different paths from the sensor to the base station. Such events obviously affect duplicate-sensitive queries and so we must try to deploy duplicate-insensitive summaries to answer them correctly.

Due to the decomposability of FM sketches (see, e.g. [CMZ06]) they can be readily used as summaries for in-network computation of `DCOUNT`, using as aggregation operation (1) the bitwise `OR` of the corresponding bitmaps. The evaluation operation is the same as in the centralized setting, i.e. we output $(1/\phi)2^{\sum_s r_s/m}$ where r_s is the position of the leftmost 0 bit in the s-th bitmap of the FM sketch describing our entire data set $\bigcup_i S_i$ and $\phi \approx 0.77351$ is a constant.

3. Implementation details

The purpose of the AG module is to efficiently and reliably extract useful statistical synopses, per end-user request, regarding the entire sensed data infrastructure, i.e. over all the continuous data streams observed by the sensors. There are two key concepts here:

1 *Network topology:* We construct (in real-time) a concrete topology over the sensors' network, so as to be able to systematically push the data summaries from each sensor towards a particular base-station, efficiently aggregating along the way. At the lower level, we build k-hop clusters around designated *cluster-head* nodes and, at a higher level we construct a tree structure among these cluster-heads, rooted at the base-station. The aggregate summaries are routed from the lowest levels of this tree towards the root.

2 *Failures tolerance:* We deploy algorithmic ideas that, combined with the routing procedure of the previous paragraph, allowing us to give aggregate answers within a sufficient approximation, even in the presence of events such as communication and cluster-head node failures, a practically frequent phenomenon in wireless sensor networks. This is achieved by applying *multi-path* routing along the cluster-head tree and duplicate-insensitive data structures for sensitive and insensitive aggregates such as MIN/MAX and DCOUNT.

Our approach for the above issues is based on standard in-network aggregation routing and algorithmic techniques, see e.g. [BCG11b], [NGSA08]and the resulting algorithm that can give the Max sensor values, for specific time windows in a environment with constrained memory. We use the *Partition Greedy* algorithm of [BK09]at a lower, sensor node level, in order to compute a Max aggregate for the values that are reported form the sensor in a periodic, time-windowed manner. The Partition Greedy algorithm has two important parameters, (i) the window size in elements (denoted as w) (ii) the memory size in elements (denoted as k). The main operation involves partitioning the input stream in blocks of size $\frac{w}{k}$, the s-th block thus being the one between time indices $(s-1) \times \frac{w}{k} + 1$ and $s \times \frac{w}{k}$. In each memory slot we store the sensor's value along with a time-stamp that shows exactly when the value was read. Throughout each block s only one memory slot is active (can be updated), in particular $i = 1 + (s \mod k)$. This slot always accepts the first element of the current block and while its active is only updated greedily if a larger value is observed or if the time-stamp of the stored value is older than the window size.

Using the above schemes the algorithm keeps a good estimate of the Max observed value of the last w (window size) value samples, while using a vector containing only k samples. Then the algorithm sends the Max values using a in-network aggregation mechanism to a sink node, this can be triggered from a response to a user query or from a periodic timeout-event. The in-network aggregation utilizes the CL, End-to-End and a Tree Routing module to forward the aggregate Max value to the sink node. As a first step the AG module performs an aggregation at the cluster level, this is initiated by nodes sending the values towards the cluster-head, using routing information that is provided by the CL module. The cluster-head nodes upon receiving the first value from a cluster member calculate and maintain an cluster aggregate Max value, and start a timer waiting for the rest of the cluster-members to report their values. The timeout for the timer is set at $2 \times k \times messagePropagationDelay$, where k is the maximum hop distance for any cluster-member from the cluster-head. When the cluster-head receives messages from all the cluster-members or a timeout event occurs the cluster-head sends the cluster's aggregate Max value to the sink node. The second step of the in-network aggregation employs a tree-routing protocol that is running only on the cluster-heads and it is using the Highways to facilitate the communication between them. The cluster-head that the sink node belongs, starts building a tree-structure by broadcasting search messages through the Highways. This internal structure provides the cluster-heads with the routing path to the sink node. When the aggregate messages(containing the clusters Max values) are delivered through the Highways, the sink node aggregates them to get the entire network's Max value.

4. Interfaces

The Aggregation Module follows the general Wiselib interface.

```
template<...> class AggregationModule {
 int enable(void);
 int disable(void);

 init(RadioT, ClockT, TimerT , DebugT, NeighborDiscoveryT, ClusteringT, HighwaysT);

 set_sink(node_id_t);
 set_k(int);
 set_w(int);

 put(value_t);
 value_t get();
 send(value_t);

 receive(node_id_t form, value_t value);
}
```

Interfaces with other components:

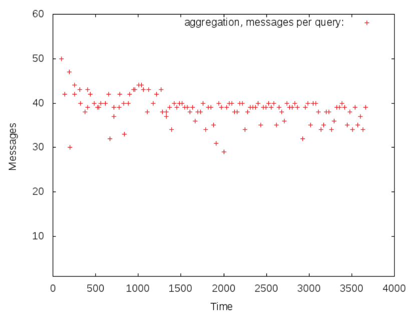
Neighborhood Discovery – This module provides a list with the stable neighbors to the AG module. It is used for piggybacking the aggregate value in the broadcast beacons for the second step of the in-network aggregation.

Clustering – The clustering module is used to structure the network in clusters of nodes, for the first step of the in-network aggregation. The AG module will assign different roles in each node depending on the CL module status(i.e., if the node is a cluster-head or not).

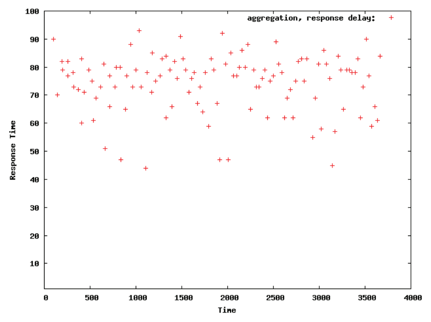
End-to-End – The module is used for routing the aggregate data from the cluster-head nodes to the sink node.

5. Performance analysis

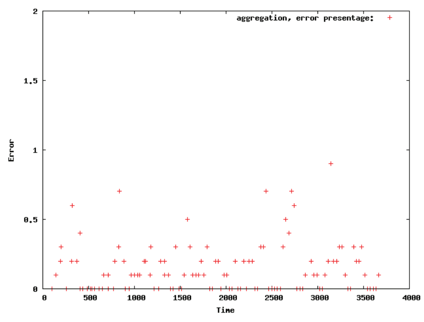
We conduct a series of experiments in order to determine the performance of the AG module. We measure the: (i) *response time per query* (i.e., we calculate time from the beginning of the request until the final value is delivered in the SINK node.), (ii) *value error percentage per query* (i.e., the difference between the value what the AG module return and the actual maximum value in the network at the time of the request), (iii) *number of messages per query* (i.e., the total number of messages the algorithm needs in order to complete one query). For the purposes of the experiment, each node is generating a random number for 0 to 1000 every second, that is used as input to the AG module. The AG is using a window of 60 seconds and we use a limited memory that can hold 10 values. Each experiment has a total duration of 3600 seconds and we execute a query every 30 seconds. The results for the 120 queries are depicted in Figure 1. In the Figure 1(a) we can observe that the messages that the aggregation algorithm sends per query are strongly concentrated around the value 40, with a minimum observed value of 30 and a maximum value of 50. Due to the use of the clustering as a first step in our protocol we have reduced the total number of messages that need to be transmitted to the SINK node. As a result the duplicated transmitted messages generated as part of the multi-path routing brings the total number of transmitted messages at a maximum of 1.8 times the number of nodes. In Figure 1(b) we present the response time of each query that is executed. More than 50% of the values are between 70 and 80 milliseconds, with a maximum response time of 98 milliseconds and minimum value of 43 milliseconds. In Figure 1(c) we show the results of the error percentage in the return value per query. There are two main sources that affect the error: (i) the error of the estimation of the streaming aggregation per node and (ii) the error introduced from messages that are lost when



(a) Messages per query

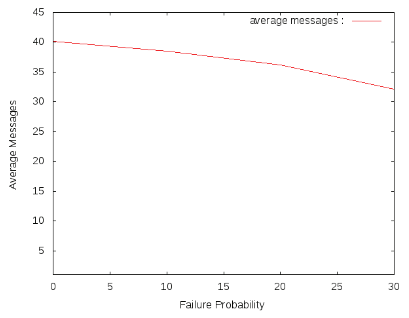


(b) Response time per query

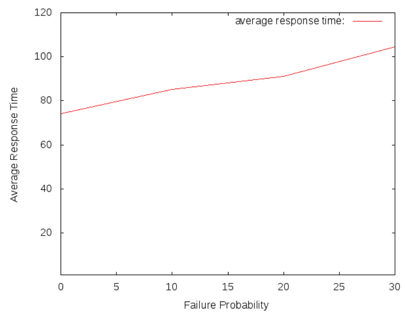


(c) Error percentage per query

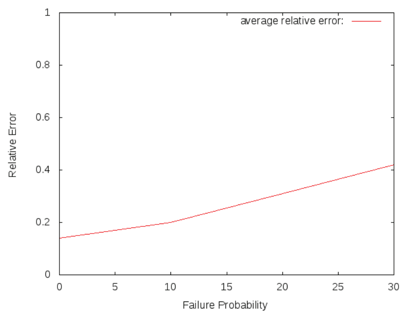
Figure 1: Experimental results per query without node failures



(a) Average Messages



(b) Average Response time



(c) Average Error percentage

Figure 2: Average results from experiments with different node failure probability

a query is executed. As we see from the error results most of estimated values are very close to the actual values as expected from the theoretical results.

We now discuss the effect of node failures on the performance of the AG module. In order to determine what are the effects of node failures on the protocol's performance, we conducted a number of experiments involving node failures. The experiments were conducted using a window of 60 seconds and we use a limited memory that can hold 10 values. Each experiment has a total duration of 3600 seconds, and we execute a query every 30 seconds. For every query a percentage of nodes are disabled according to a failure probability. More specifically we have conducted 4 sets of experiments with 0% 10% 20% 30% probability of node failure. The results are shown in Figure 2. More specifically in Figure 2(a) we observe that the average number of messages generated are declining as the failure probability increases, this is to be expected since failing nodes do not participate in the protocol, they no not send their aggregate value neither propagate other in-network aggregated values. In Figure 2(b) we depict the average response time over all the queries. Finally in Figure 2(c) we can see how the average error evolves as the node probability failure increases. The average error is increasing as more nodes fail. This is caused from messages that fail to reach the SINK node since some node in the route path has failed. The effect is greater as the failure rate increases because a lot of the cluster-heads are disabled, leading to a higher loss of information in the network.