Data-Driven Processing in Sensor Networks*

Adam Silberstein Rebecca Braynard Gregory Filpus Gavino Puggioni
Alan Gelfand Kamesh Munagala Jun Yang
Department of Computer Science Institute of Statistics and Decision Sciences
Duke University, Durham, NC 27708, USA
{adam,rebecca,gef2,kamesh,junyang}@cs.duke.edu {gp10,alan}@stat.duke.edu

ABSTRACT

Wireless sensor networks are poised to enable continuous data collection on unprecedented scales, in terms of area location and size, and frequency. This is a great boon to fields such as ecological modeling. We are collaborating with researchers to build sophisticated temporal and spatial models of forest growth, utilizing a variety of measurements. There exists a crucial challenge in supporting this activity: network nodes have limited battery life, and radio communication is the dominant energy consumer. The straightforward solution of instructing all nodes to report their measurements as they are taken to a base station will quickly consume the network's energy. On the other hand, the solution of building models for node behavior and substituting these in place of the actual measurements is in conflict with the end goal of constructing models. To address this dilemma, we propose *data-driven* processing, the goal of which is to provide continuous data without continuous reporting, but with checks against the actual data. Our primary strategy for this is *suppression*, which uses in-network monitoring to limit the amount of communication to the base station. Suppression employs models for *optimization* of data collection, but not at the risk of correctness. We discuss techniques for designing data-driven collection, such as building suppression schemes and incorporating models into them. We then present and address some of the major challenges to making this approach practical, such as handling failure and avoiding the need to co-design the network application and communication layers.

1 Introduction

Wireless sensor networks have potential to provide a wealth of data about the environments in which they are deployed. This is tempered by sensor nodes' limited battery life. One of the main energy consumers is radio communication, which far outweighs the costs to execute CPU instructions or take simple readings such as temperature. The cost to continuously transmit all readings from within the network to an offline base station is enormous, yet the data must somehow be delivered to the user. Therefore, there exists a great need and opportunity to increase energy efficiency through distributed, in-network processing.

One prominent direction in sensor data management is *model-driven* processing [10, 9, 8]. This approach assumes a probabilistic model of the sensor readings that offers information such as distributions of individual readings, correlations among different types of readings at the same node, and spatial correlations among readings from different nodes. Given a query over the sensor readings, this model is used to minimize the energy spent, both in transmissions and sampling, and provide an acceptably accurate result. For example, to satisfy a query for a particular node's temperature, we may substitute in measuring voltage, because temperature and voltage are highly correlated and measuring voltage is less expensive [10]. As another example, for a query identifying nodes whose values fall within a certain range, we need only request values from nodes for which the model cannot predict with enough certainty their inclusion in or exclusion from the query result. Model-driven captures the intuition that in many deployments, predictable correlations will exist among different types of measurements, and among different nodes. This is a tremendous insight for avoiding total data collection.

A Case for Data-Driven Processing We are currently working in collaboration with a group of ecologists and statisticians at Duke University to deploy and maintain a wireless sensornet in Duke Forest to better understand how forest tree growth, survival, and reproduction are influenced by changes in climate, atmospheric carbon dioxide, disturbances, and other environmental factors. The sensornet measures a variety of readings, including temperature, light, rainfall, soil moisture, sap flow, etc. The monitored environment will undergo controlled experiments involving burning, herbivore exposure, etc. Using high-resolution data collected from the network, our collaborators are developing sophisticated ecological models incorporating these variables.

In this project we have found the model-driven approach not necessarily the best starting point. As is often the case in scientific and exploratory uses of sensornets, we do not always know *a priori* what models best describe the sensor values being monitored. Since data alone is the "ground truth," our collaborators want to collect all data, instead of entrusting rudimentary models to avoid data acquisition. The concern is if these models turn out to be inadequate (and they often do), we might miss important data points that would allow us to construct better models. One of our ecologist collaborators has quipped, "in environmental monitoring the model is never correct." That is, models help us understand the environment, but they are not precise enough to accurately stand in for actual readings on a per-measurement basis. This utmost emphasis on wanting the data leads us to an alternative approach, which we call *data-driven* processing. This approach does not assume we begin

*This work is supported by the NSF CAREER, DDDAS, and REU programs (under awards IIS-0238386, CNS-0540347, and IIS-0625690), and by an IBM Faculty Award.

3^{rd} *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

with a good model of the sensor values. In the extreme, it assumes nothing about the environment in which the sensornet is deployed. Data-driven processing still uses models, but only as optimization guidelines (more on this later), so even inaccurate models do not lead to data loss.

Optimizing `SELECT *` With emphasis on collecting all data, it should not be surprising continuous `SELECT *` is the dominant query for data-driven processing. Much of the research in our project now focuses on optimizing this query, instead of complex, database-like queries. While in a traditional database system, `SELECT *` is uninteresting in terms of optimization possibilities, its continuous sensornet version is quite amenable to interesting *suppression* techniques. The naive alternative, continuous reporting, requires each node transmit its value to the base station in each time step, resulting in heavy message traffic. Suppression, on the other hand, monitors conditions within the sensornet such that data is only transmitted when readings diverge from expected (or simply default) behavior. The simplest example is *value-based temporal suppression*, where each node reports only if its value has changed beyond some threshold since last reported. This technique is quite effective when sensor values change slowly. Still, we can advance suppression further, to handle scenarios where nodes change regularly but predictably, and where changes across nodes exhibit spatial correlations. A number of suppression schemes have been proposed [2, 6, 13, 17, 22, 23, 25]. Suppression is a powerful concept, but many challenging issues remain to be addressed, e.g., how to identify missing reports as suppressions, rather than failures. These problems are addressed throughout this paper.

Although there have been some recent examples of data-driven processing in the literature [21], including by some of the designers of model-driven [2], we believe this approach has been understudied relative to the model-driven approach. The goal of this paper is to present the key issues in data-driven processing and techniques for applying them to application design. We have encountered the following issues in designing our own deployment, and generalize them for broad applicability.

- Suppression: This is the driving technique behind supporting continuous queries without continuous data streams. We define suppression schemes for supporting continuous queries. Most sensornets likely exhibit temporal and spatial correlations among node readings, which are encoded within models. This behavior can be incorporated into schemes such that with limited messaging within the network, no or few messages are sent to the base station. Suppression, including incorporation of models, is covered in Section 2.
- Message failure: Sensor networks are prone to message drops. This is especially detrimental to suppression schemes, where non-reports are expected, but now may be the result of failures, rather than suppression. In Section 3 we discuss methods for augmenting suppression schemes against failure, and techniques for detecting failure and then inferring the actual readings and process parameters.
- Application/communication layer interaction: Sophisticated suppression schemes involving careful communication among multiple nodes strain the lower communication layer to provide efficient routing between such nodes. While merging these layers would greatly complicate application development, some hooks between them are necessary. In Section 4 we describe the milestone framework, for co-optimization of the layers.
- Data representation: Managing data produced in a sensornet and the inferred at the base station is difficult. Presenting either extreme of a view of only the data known with perfect certainty, or a completely sanitized version of all data are both inadequate in most cases. The data is inherently probabilistic with complex correlations. These correlations, as well as details of suppression and the constraints it enforces on inference must be represented. These open problems are discussed in Section 5.
- Role of models: Models are discussed extensively throughout this paper. In general, models are utilized in data-driven for optimization, but not at the expense of correctness. Further, model-driven and data-driven are not static competitors; the trust and responsibility given models largely defines where along the spectrum between the approaches an application lies. This distinction can become tricky to follow, especially when discussing of failure. We review use of models in Section 6.

2 Suppression

Suppression is the key technique for supporting continuous queries without continuous reporting. The network, on its own volition, chooses when to push data to the base station. The intuition is if the network and base station can agree on an expected behavior, the network need only report when its readings deviate from that. The challenge is encoding expected behavior within the network, such that actual behavior can be efficiently evaluated against it.

The design space for suppression is enormous. Suppression can be utilized in a multitude of ways, even for the same query. Value-based temporal suppression, mentioned in Section 1, leverages the expectation node values are unlikely to change in a given timestep. Each node sends a message only when its value does change beyond some threshold since last reported. This scheme allows computation of `SELECT *`, since the base station maintains the last value reported from each node within the threshold. If values change frequently, this degrades to continuous reporting. Spatial suppression is also possible. *Snapshot* [21] allows individual nodes to report on behalf of a local cluster, as long as nodes in the cluster have values within some threshold of the representative. This approach leverages the expectation that nearby nodes will have similar values, and minimizes the number of messages directed to the base station. If nodes do not exhibit spatial correlation, this degrades to continuous reporting.

The design space extends to more sophisticated schemes that leverage both temporal and spatial correlations. Naturally, the effectiveness of a scheme for a particular deployment depends on how well it captures the correlations existing in the deployment. In order to manage the design process, we now define a general framework for suppression schemes.

2.1 Suppression Scheme

Definition A *suppression scheme* is a set of *suppression links* deployed within the network. A link maps a suppression/reporting relationship between an *updater* node and an *observer* node. Each link synchronously maintains, with some error, a vector of quantities, X, between the updater and observer. X_t denotes the vector at time t as instantiated by the updater. $\hat{X}_t$ denotes the vector as computed by the observer. $\hat{X}_t$ serves as input to the observer for producing its own X vectors, to use on downstream links (observer becomes updater). Eventually, a querying node (such as the root) receives an $\hat{X}_t$ and uses it to produce query results. Variables $x_{t,i}$ and $\hat{x}_{t,i}$ refer to ith quantity in the vector at the updater and observer, respectively. X may contain node readings, process parameters, and any other quantities used by the scheme. Each quantity may be generated locally at the updater (e.g. a reading taken by it) or derived from quantities received from its own upstream updaters.

For each suppression link, the user defines per-quantity precision

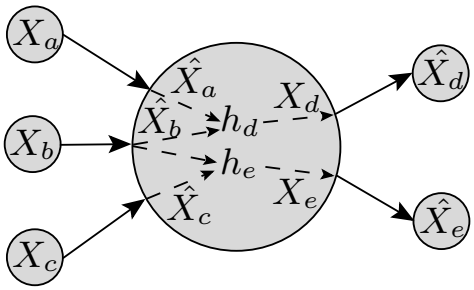

Figure 1: Suppression Scheme Graph.

bounds for each entry in X_t and $\hat{X}_t$. Predicate function $g(X_t, \hat{X}_t)$ returns true if $\hat{X}_t$ is within a prescribed error tolerance of X_t. For example, the function may be component-based, such that each $\hat{x}_{t,i}$ must be within some range of $x_{t,i}$. X_t and $\hat{X}_t$ are considered synchronized if g is true. The synchronization requirement dictates the communication necessary, if any, between the updater and observer in each timestep. The following functions encode the requirement. The updater maintains encoding function:

- $f_{enc}(X_t, X_{t-1}, \ldots)$

The observer maintains decoding function:

- $f_{dec}(r_t, \hat{X}_{t-1}, \hat{X}_{t-2}, \ldots)$

The updater uses f_{enc} to generate report r_t for transmission to the observer so the observer can derive $\hat{X}_t$ within precision bounds. If no message is needed, r_t is denoted $\perp$, for suppression. Note because f_{enc} uses previous settings of X, it implicitly considers previous messages sent to the observer. The observer uses f_{dec} to interpret r_t and derive $\hat{X}_t$. If no message is received, it sets $r_t = \perp$. Assuming no failure, $\hat{X}_t$ necessarily meets the precision requirements, though the observer has no way to verify this.

Note the relationships between f_{enc}, r_t and X, and f_{dec} and $\hat{X}$ are defined by the scheme programmer. For example, X might contain several readings and a single parameter, θ. r_t, when not suppressed, may only consist of updates to θ, from which all other quantities in $\hat{X}$ are derived. We give examples in Section 2.2.

The suppression scheme is a graph of suppression links. An observer node may maintain links with one or more updaters, and may itself then become an updater for one or more observers. The node maintains a function h_j for each downstream observer to transform $\hat{X}_t$'s received from its updaters into its own $X_t^{(j)}$, to then be synchronized with a downstream observer. h functions are defined by the scheme programmer.

Message Dependency Each h_j also establishes a dependency between $\hat{X}_t$'s derived from updater messages to $X_t^{(j)}$. $X_t^{(j)}$ in turn determines what is transmitted downstream for the observer to derive $\hat{X}_t^{(j)}$. These dependencies are *intra-node links*. We distinguish these from suppression links because they have no explicit impact on what is transmitted between nodes, but instead have impact on when that transmission may take place. Figure 1 depicts a portion of a suppression scheme graph, with the middle node, serving as an observer and then updater, enlarged. The suppression links are shown as solid arrows, while the intra-node links are shown as dashed arrows. As observer for three suppression links, the node uses f_{dec} functions to derive $\hat{X}_a$, $\hat{X}_b$ and $\hat{X}_c$. As updater for two suppression links, it uses f_{enc} functions to derive messages to be transmitted downstream for derivation of $\hat{X}_d$ and $\hat{X}_e$. Internally, the node shows dependency on $\hat{X}_a$ and $\hat{X}_b$ to produce X_d, and on $\hat{X}_b$ and $\hat{X}_c$ to produce X_e.

By merging suppression and intra-node links, we derive the *h-graph*, a directed graph whose vertices are the h functions at all network nodes.

Lemma 1. *A suppression scheme is feasible only if its h-graph contains no cycles.*

In a particular timestep, only nodes with h functions not waiting on upstream messages may immediately transmit (X vectors from such functions are populated using input generated completely locally). If a subset of h's at different nodes all wait on one another in a cyclical fashion, the suppression scheme waits indefinitely. The scheme completes when all terminal nodes (serving as observers, but not as updaters) receive their messages. In most query processing applications, this will be a single node, the root. The suppression scheme *supports* a particular query if $\hat{X}$ maintained by the root is sufficient to produce the query result.

2.1.1 Discussion

Why go through the exercise of producing a general definition for suppression schemes, when its absence has not prevented us or others from designing schemes? We want a definition that is flexible, but exposes the important features that characterize and differentiate schemes. The definition provides a number of benefits:

- Compared with low-level programming of network messages, this abstraction makes it easier to reason about suppression algorithms. For example, it is simpler to understand the correlation an algorithm exploits, prove its correctness (both for feasibility and for supporting queries), and evaluate energy cost.
- It allows cost-based optimization in scheme design. While the design space of possible schemes remains enormous, many possibilities can be quickly eliminated due to cost. For example, a scheme with intricate spatial constraints, but higher expected cost than temporal suppression, should not be considered.
- It allows identification of optimizations general to all schemes, in contrast to application-specific optimizations. These can be presented as such and implemented once for all schemes.
- The definition can be adapted to a new application programming abstraction. Existing approaches require programmers to either plan each message by hand, or else provide only a limited set of communication patterns, such as *collection* and *dissemination* [3]. A suppression API will give programmers more control over collection, but while factoring out common tasks. We imagine, for example, setting a *cluster* abstraction by simply selecting a set of cluster members. The choice of cluster leader may be left to the API, which can make the optimal decision. The API is accompanied by cost analysis to assist in refining scheme designs.

2.2 Examples

We now demonstrate how the general definition can be specified to a series of existing suppression schemes.

Temporal Schemes We begin with value-based temporal suppression. The suppression link graph consists of link between each node and the root (i.e. a one-level link tree), where the node is an updater and the root an observer. At each node X has a single component x, its reading. g returns true if two readings are within ϵ_x of one another. Each node maintains a local variable t', the time at which its value was last transmitted. Its encoder function is:

$$f_{enc}(x_t, x_{t'}) = \begin{cases} x_t - x_{t'} & \text{if } |x_t - x_{t'}| > \epsilon_x \\ \perp & \text{otherwise} \end{cases}$$

The root for each link has decoder function:

$$\hat{x}_t = \begin{cases} \hat{x}_{t-1} + r_t & \text{if } r_t \neq \perp \\ \hat{x}_{t-1} & \text{if } r_t = \perp \end{cases}$$

2.2.1 Soil Moisture

We next examine model-encoding temporal suppression schemes for monitoring soil moisture. In our forest modeling efforts, soil

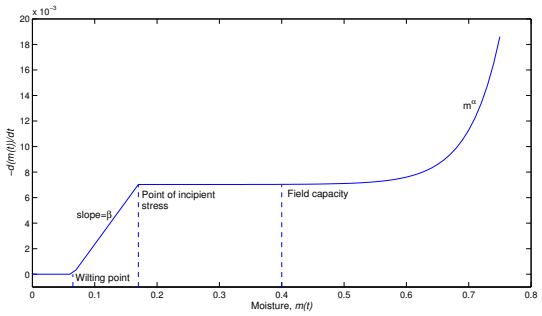

Figure 2: Soil Moisture Model

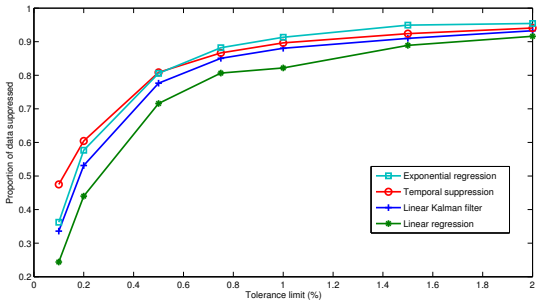

Figure 3: Suppression Results

moisture is an important measurement. Moisture increases dramatically during precipitation, and then falls off either due to subsurface run-off or transpiration by the forest. Precipitation events can be modeled as a Poisson process with amount of precipitation drawn from an exponential distribution. Modeling soil moisture is more involved. The rate of decrease in moisture after a precipitation event depends on the value of the moisture, $m(t)$. Figure 2 depicts the rate of decrease in $m(t)$ as a function of $m(t)$ for a single sensor node. We will focus on the right-most portion of the model, when moisture is above field capacity. We examine two per-node suppression schemes: PAQ [25] and exponential regression.

For both methods, the link graph is identical to that of value-based temporal. Each node maintains a link with the root. PAQ employs linear regression to have a node predict its reading each timestep using the previous three readings and parameter values, $\alpha_t, \beta_t, \gamma_t, \eta_t$. The prediction for x_t is as follows.
$\alpha_t(x_{t-1} - \eta_t) + \beta_t(x_{t-2} - \eta_t) + \gamma_t(x_{t-3} - \eta_t)$
α, β, and γ are derived by regression and η is the mean value of the past readings. These parameters evolve over time, necessitating their subscripts. The root knows the parameters and simultaneously makes the same prediction. If the node's prediction is within ϵ_x of its reading, it suppresses. Otherwise, it has the option of transmitting the reading as an outlier, or revising and transmitting the parameters. The details underlying this decision are in [25] and are local to each node. For our purposes, we abstract these away by assuming the node has boolean functions `modelRefit` and `outlier`. If $|x_t - x_{t-1}| > \epsilon_x$, one and only one of these returns true (otherwise, neither are true). If `modelRefit` returns true, the function has also updated $\alpha_t, \beta_t, \gamma_t$ and η_t. Note `modelRefit` and `outlier` must mimic the prediction made at the root, using the previous three *predictions* $(\hat{x}_{t-1}, \hat{x}_{t-2}, \hat{x}_{t-3})$ rather than the actual readings.

We now place PAQ in our framework. $X_t = [x_t, \alpha_t, \beta_t, \gamma_t, \eta_t]$. g dictates a separate ϵ for each component of X. ϵ_x is user-defined. $\epsilon_\alpha, \epsilon_\beta, \epsilon_\gamma$ and ϵ_η are all set to 0 (i.e. the root is notified of any updates). Each node has encoder function:

$$f_{enc} = \begin{cases} \alpha_t, \beta_t, \gamma_t, \eta_t & \text{if } (\texttt{modelRefit}) \\ x_t & \text{if } (\texttt{outlier}) \\ \perp & \text{otherwise} \end{cases}$$

The root has decoder function:

$$\hat{\alpha}_t, \hat{\beta}_t, \hat{\gamma}_t, \hat{\eta}_t \leftarrow \begin{cases} \alpha_t, \beta_t, \gamma_t, \eta_t & \text{if } r_t = [\alpha_t, \beta_t, \gamma_t, \eta_t] \\ \hat{\alpha}_{t-1}, \hat{\beta}_{t-1}, \hat{\gamma}_{t-1}, \hat{\eta}_{t-1} & \text{otherwise} \end{cases}$$

$$\hat{x}_t \leftarrow \begin{cases} x_t & \text{if } r_t = x_t \\ \hat{\alpha}_t(\hat{x}_{t-1} - \hat{\eta}_t) + \hat{\beta}_t(\hat{x}_{t-2} - \hat{\eta}_t) & \\ \quad + \hat{\gamma}_t(\hat{x}_{t-3} - \hat{\eta}_t) & \text{otherwise} \end{cases}$$

We next look at exponential regression. Each node has a prediction model for suppression: $x_t = \alpha_t x_{t-1} + \beta_t$. $X_t = [x_t, \alpha_t, \beta_t]$. Like PAQ, ϵ_x is user-defined, while ϵ_α and ϵ_β are 0. Each node has encoder function:

$$f_{enc} = \begin{cases} \alpha_t, \beta_t & \text{if } (\texttt{modelRefit}) \\ x_t & \text{if } (\texttt{outlier}) \\ \perp & \text{otherwise} \end{cases}$$

The root has decoder function:

$$\hat{\alpha}_t, \hat{\beta}_t \leftarrow \begin{cases} \alpha_t, \beta_t & \text{if } r_t = [\alpha_t, \beta_t] \\ \hat{\alpha}_{t-1}, \hat{\beta}_{t-1} & \text{otherwise} \end{cases}$$

$$\hat{x}_t \leftarrow \begin{cases} x_t & \text{if } r_t = x_t \\ \hat{\alpha}_t(\hat{x}_{t-1}) + \hat{\beta}_t & \text{otherwise} \end{cases}$$

Framing both of these approaches as suppression schemes makes for an elegant comparison. First, they both use local suppression at each node. This commonality is exposed in their suppression link graphs, which are identical. The differences are the encoding and decoding functions. Given the probability of how often each component of X is suppressed, it is straightforward to predict the cost of each scheme.

We now return to Figure 2. Moisture drops exponentially when moisture is high (on the right side of the graph). We expect a suppression scheme aware of this to have an advantage over those that do not. This is a good example of the role of models in data-driven acquisition. All of the discussed schemes produce readings within ϵ_x of the actual. But because exponential regression best models the data, it should have the most accurate predictions and the most suppression. In Figure 3, we compare suppression rates for value-based, linear regression (PAQ), exponential regression, and a Kalman filter variation [17] using simulated moisture data. Error tolerance is varied on the x-axis and corresponding suppression rate is plotted on the y-axis. We see exponential regression indeed achieves the best suppression rate, though the improvement over the others is not dramatic. We are investigating these types of comparisons further.

2.2.2 *Spatio-temporal Suppression*

We next look at a suppression scheme that exploits not just temporal correlations at individual nodes, but spatio-temporal correlations between multiple nodes. This requires monitoring spatial conditions in-network, so the suppression scheme will not consist solely of links between each node and the root.

Conch The scheme is one we have previously designed, *Conch* [23]. We examine a value-based version of it, making it a spatio-temporal equivalent of value-based temporal suppression. Note we employ a different version of error tolerance than published in [23] that better mirrors the prior temporal schemes (we discuss both shortly). The main idea is to monitor network edges connecting neighboring nodes; specifically, we monitor the difference in value between each edge's endpoint nodes. The root maintains the current difference across each such edge. The intuition is to choose edges whose nodes are correlated; when their values change, they will change synchronously by similar amounts, and the difference will not change and be suppressed. We next define monitoring of edges in our suppression framework, and then describe how to build

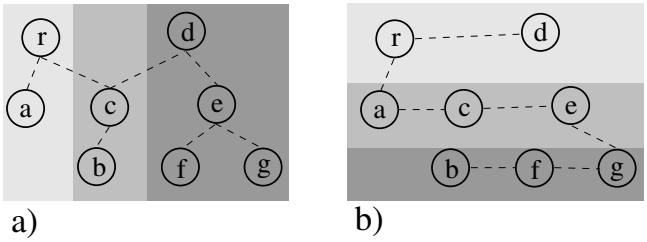

Figure 4: Conch Example

a suppression scheme to support `SELECT *` from these.

Monitoring an edge requires communication, and thus a suppression link, between its endpoint nodes u_1 and u_2, and then another link between u_2 and the root. g at u_1 returns true if values are within ϵ_x of one another. X at u_1 contains only its reading, x. u_1 maintains a local variable, t', the time in which it last transmitted to u_2. u_1 has encoder function:

$$f_{enc} = \begin{cases} x_t - x_{t'} & \text{if } |x_t - x_{t'}| > \epsilon \\ \perp & \text{otherwise} \end{cases}$$

u_2 has decoder function:

$$\hat{x}_t = \begin{cases} \hat{x}_{t-1} + r_t & \text{if } r_t \neq \perp \\ \hat{x}_{t-1} & \text{if } r_t = \perp \end{cases}$$

Note these functions are the same as used by the node-root link for temporal suppression.

u_2 measures its own reading, y. X at u_2 contains only the difference in readings across the edge, Δ. To produce this, u_2 has function $h_\Delta = y_t - \hat{x}_t$. g returns true if two Δ's are within ϵ_Δ of one another. u_2 maintains a local variable, t'', the time in which it last transmitted to the root. u_2 has encoder function:

$$f_{enc} = \begin{cases} \Delta_t - \Delta_{t''} & \text{if } |\Delta_t - \Delta_{t''}| > \epsilon \\ \perp & \text{otherwise} \end{cases}$$

The root has decoder function:

$$\hat{\Delta}_t = \begin{cases} \hat{\Delta}_{t-1} + r_t & \text{if } r_t \neq \perp \\ \hat{\Delta}_{t-1} & \text{if } r_t = \perp \end{cases}$$

Though more details are available in [23], we now briefly explain how to support `SELECT *` with edge monitoring. In one special case of Conch, we monitor the root with temporal suppression and monitor a set of edges such that they form a spanning tree over the network. If a single node serves as u_2 for multiple edges, its X contains a Δ for each. Its encoder function suppresses Δ's that have not changed by more than ϵ_Δ since last transmitted, and reports those that have.

Two example Conch spanning trees are shown in Figure 4. For each edge, the root knows the current difference between its vertices within ϵ_Δ. To derive any node's value, the root finds a path of edges in the spanning tree from itself to the node (by definition of spanning tree, exactly one path must exist). The root then starts with its own value and modifies it by each edge difference in the path, a process called *chaining*. Many different spanning trees can be built over a single network. As we see from the scheme declaration, the cost of monitoring a particular edge is tied to how often x changes by more than ϵ_x and the cost to communicate from u_1 to u_2, and how often Δ changes by more than ϵ_Δ, and the cost to communicate from u_2 to the root. If two nodes' values demonstrate high correlation, their edge is a good candidate for monitoring; it will seldom need to report to the root. Suppose in Figure 4, each shaded region exhibits a particular behavior, such that all nodes in a single region are correlated. Any edge between nodes in the same region will never report to the root, while edges between nodes in different regions will frequently report. The goal in building the spanning tree is to select as few edges that cross regions as possible. In both examples, each with three regions, we are forced to choose two such edges. Imagine we trade the spanning trees, such that the tree in 4a is assigned to the environment in 4b, and vice-versa. It is easy to see reporting costs will increase in both cases.

Error We see interesting implications for suppression parameters ϵ_x and ϵ_Δ. The suppression link graph has hierarchy of depth two. At u_2, $\hat{x}_t$ can diverge from x_t by as much as ϵ_x. Thus, Δ may diverge from $y_t - x_t$ by ϵ_x. This error accumulates at the root: $\hat{\Delta}_t$ may diverge from $y_t - x_t$ by $\epsilon_x + \epsilon_\Delta$. The same problem carries over to chaining. For each edge involved in chaining to a particular node, error accumulates. For a chain of length l, the computed node's value has error up to $l(\epsilon_x + \epsilon_\Delta)$. To achieve the same accuracy as temporal suppression, it is necessary to use lower ϵ values, and likely different ϵ's for each edge. It is important to understand the error implications of a suppression scheme. The suppression link graph view of the scheme exposes this.

Original Conch suppression works as follows. Nodes monitor not x, but the discrete quantity, $x' = \lfloor x/\epsilon \rfloor$, with error bound of 0. Differences are computed on discrete quantities, also suppressed with error bound 0. We estimate $\hat{x}_t$ as $x'_t\epsilon$, and can guarantee $\hat{x}_t \leq x_t < \hat{x}_t + \epsilon$. The advantage of this approach is it makes chaining less susceptible to error accumulation. The disadvantage is this discretization is somewhat artificial; small changes in actual value that happen to cross a discretization step must be reported.

2.3 Toward an Optimization Framework

We are building toward a framework for suppression optimization. The first component of this is the suppression scheme definition. As mentioned, this allows us to manage the large design space of suppression schemes, and to evaluate cost for each. We foresee two levels of optimization. The higher level is the compile-time decision of what scheme should be deployed in the network, such as temporal suppression and, as we have seen with soil moisture, what models can be monitored within that scheme.

The lower level is at run-time. The network itself makes dynamic adjustments to suppression. For PAQ, this involves deciding when a sequence of outliers actually signifies a change in process parameters (on which suppression will now be based). The moisture model contains three distinct components. An effective suppression scheme must encode all of them and know when to shift between. One possibility is to set choice of model as a state parameter in X, reported only when the node shifts, and suppressed otherwise. In these cases, the ability to modify suppression is precompiled within the network, and need only be invoked by nodes at run-time.

A final question is how and when to make externally initiated changes to the suppression scheme. For example, the base station may contain a rain sensor, making it better qualified than individual nodes to decide whether increased moisture readings are outliers or a shift to the high-moisture component of the model. Further, the base station collects a great deal more information than any particular node. Any sweeping changes to the scheme, which likely come with high energy cost, should be made by the base station.

End Goal We have extensively discussed the use of suppression, design of suppression scheme, and integration of models into these. Beside our omnipresent goal of energy-efficiency, it is easy to lose sight of what we hope to achieve. To that end, we have the following succinct vision for suppression:

1. No messages reach the base station reporting expected behavior.
2. For each unexpected phenomenon affecting the network, only one message reaches the base station.

If all nodes are affected similarly by the same event, they should detect this in-network and not independently report it to the base station. Nevertheless, this goal must be tempered by the in-network

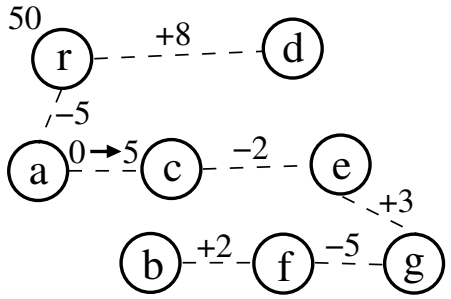

Figure 5: Conch Failure

cost to coordinate the single report. In the end, this vision is a guideline to motivate suppression scheme design rather than something to be achieved in practice.

3 Message Failure

Failure is a serious problem in sensornets, with message loss common. Experimental results have shown transmission loss rates to be at times 50% and higher, with congestion blamed for the majority of these [16]. Congestion causes message interference, where multiple messages are sent in the same air space, causing all of them to be corrupted. When message buffers at nodes fill to capacity, nodes must either drop messages as they arrive, or delete messages from their buffers; in either case, messages are lost. These findings were from a high-traffic network running a continuous reporting application. To some degree, suppression should naturally mitigate these problems. Other work blames failure on functioning, but unreliable, links, for which multiple transmissions are likely needed to achieve successful delivery [26, 27]. In outdoor deployments, environmental interference can certainly cause message loss. In ours, for example, connectivity degrades during the summer, when there is more foliage. Failure not only occurs at the message level, but at the bit level within each message. For now we assume corrupted bits are corrected (message is successful) or cannot be (message fails) at a lower level.

Failures are detrimental to suppression schemes. In the absence of failure, observers assume non-reporting suppression links have not transmitted a message, but suppressed. Failure creates ambiguity: now an observer must consider the possibility its updater did transmit a report, but that it was lost. For value-based temporal suppression, a single lost report from a node results in the node's value being mis-set by the base station. For more complex schemes, the impact may be more widespread. Figure 5 depicts a Conch example, labeled with the latest reported value for each spanning tree edge, with r as the temporally monitored root. In the current timestep, the edge $a \rightarrow c$ reports its difference rising from 0 to 5. Suppose this report is lost. It is easy to see the base station will mis-calculate the values of nodes c, e, g, f and b in chaining. The single failure affects all node values computed using $a \rightarrow c$, and continues to affect them at least until $a \rightarrow c$ changes again and attempts to report. Without further effort, the base station has no way to differentiate suppressions and failures and remedy this.

Coping at the Network Layer The most straightforward strategy for coping with failure is to eliminate it. At the medium-access (MAC) layer, we can require each message along a single hop from sender node to receiver be followed with an acknowledgment message from the receiver. If no acknowledgment is received after some time, the sender sends the original message again. After some number of attempts, however, the sender must give up. At that point, the MAC layer may return the transmission back to the communication layer, and request an alternative path be used. Eventually, the sender may run out of paths and attempts at each and give up entirely. Fundamentally, there is no way to fully eliminate the chance of failure.

Coping Implicitly Some applications assume there is enough naturally occurring redundancy from temporal and spatial correlation in data to compensate for failure. In continuous reporting applications, if values are received from most nodes, missing ones can be filled in with neighbors' values. If a node's value is missing in one timestep, it can be filled in with the average of its values in adjacent timesteps. The suppression algorithm Ken [2], which tries to maintain an accurate model of the network at all times, assumes any mistakes due to failure will eventually be corrected when reports are received. They use heartbeat messages to ensure mistakes do not last indefinitely.

Our Approach We have advocated heavily for using suppression specifically because it removes redundancy in network reporting, saving energy. This raises important questions. If redundancy is necessary to compensate for failure, is there any point to using suppression? Or in coping with failure, will we revert suppression schemes back toward continuous reporting?

We will not revert. Suppression lets us remove redundancy so we can add it back in a controlled fashion. There exists a fundamental trade-off between redundancy cost and benefit. If we rely on natural redundancy, we have no control over this trade-off. By explicitly adding redundancy we have very flexible control. If certain suppression links are very reliable, for example, we need not add much redundancy to them or on their behalf. In general, redundancy is a component of suppression scheme design.

3.1 Application-Level Redundancy

The goal in application-level redundancy is not to reduce failure (as with MAC layer re-transmission), but to make applications robust to it. The base station's ability to produce a correct (within user-defined error) query result does not hinge on any particular reports successfully transmitting. This has a number of advantages over re-transmission (though the two can be applied in concert). First, if many re-transmissions are needed on average, that strategy may become quite expensive. The application redundancy we propose can often be added onto existing messages, saving on overhead costs. Second, programming nodes at the application-level is arguably easier than at the lower MAC and communication layers.

We rely on both redundancy and models of network behavior to produce a query result. This is clearly a blend of data-driven (redundancy) and model-driven (model) techniques; we discuss how to tune between these.

3.2 Examples

Temporal Suppression We begin with value-based temporal suppression. A node only transmits when its value differs by more than ϵ since its last transmission. We make a subtle tweak to minimize the impact of failure. Until now we have had the node transmit the difference from the last transmission, potentially saving a small number of bits versus transmitting the new value itself. The difference is used to update the node's value stored at the base station. With this approach, however, failures accumulate on top of one another and, even when interrupted by successful transmissions, perpetuate indefinitely. There is never a re-calibration that tells the base station the correct value. To prevent this, on each transmission we report the new value itself. This means we at least learn the correct value on each successful transmission, eliminating the effect of previous failures on future derivations. The sacrifice of a few extra bits is likely not significant. This illustrates the types of modifications we must make to deal with failure.

We now examine four versions of value-based temporal suppression. The first is the standard version, while the latter three add redundancy to their payloads at increasing cost.

- `Standard`
- `Counter`
- `Timestamp`
- `History`

Regardless of version, as before, a node only transmits if its value differs by more than ϵ since last transmitted. With each transmission, however, the application payload differs. This means the number of reports, and thus the total overhead paid on behalf of the lower network layers (MAC and communication), is identical. We prefer this approach to sending messages at more timesteps and paying more overhead. The root uses the received messages to fill in the set of the node's readings over time, V (v_t denotes the reading at time t). If it does not know the exact reading for a particular time, it tries to fill in one of two special symbols: s for suppression or f for failure. Combined with ϵ, these place constraints on the possible actual values to be used later in producing results. If the base station cannot distinguish between s and f, it fills in `na`, for "not available."

Differentiating the Approaches `Standard` simply transmits the node's reading. If the base station receives a report at time t, it sets v_t to the received reading. If it does not, it must set v_t to `na`.

A node running `Counter` increments a local counter on each report sent to the root and adds it to the standard message. Suppose the root receives reports with none between at t and $t + z + 1$, but with non-consecutive counter values c and $c + f + 1$. The root detects f failures and $z - f$ suppressions have occurred in z timesteps. It can estimate a recent failure rate of f/z from the node, and can enumerate $\binom{z}{f}$ possible suppression/failure scenarios to fill in the gap between reports. Using knowledge of a model and reports from other nodes, some permutations may be more likely than others, letting us express the actual values in V with some likelihood, or generate samples of them.

A node running `Timestamp` includes in its reports a list of timestamps of the last n times in which it transmitted, ordered from most recent going backward. The root applies this list to V as follows:

- Loop through each time k from current time t back to the earliest time listed in the timestamp list.
- If k is listed in the timestamps and v_k is currently set to `na`, set v_k to f.
- Else if k is not listed in the timestamps, set v_k to s.

This playback fills in timestamps as s or f, but only backward to the earliest listed timestamp. Prior to that, there is no way to distinguish failures from suppressions. The greater n, the more consecutive failures must occur for timestamps to be left as `na`. We expect a small constant n to be effective, and not add much cost compared to `Counter`. Knowing the exact positions of s's and f's lets us place bounded constraints on the actual values in V. For example, if we know v_t and identify v_{t+1} as a suppression, v_{t+1} must be within $\{v_t - \epsilon, v_t + \epsilon\}$.

`History`, finally, transmits along with the last n timestamps, the readings taken at them. Given the same n, for each v_t `Timestamp` identifies as f, `History` fills in the actual reading. If n is high enough to eliminate all `na`'s in V, this is sufficient to bound all readings within ϵ.

It is easy to see as the versions increase in payload size and therefore consume more energy in transmission, we also increase our knowledge, or certainty, about the values in V. We illustrate this trade-off by comparing `Standard` and `Timestamp` experimentally. Both schemes have ϵ set to 0.3 and are aware raw data is generated according to an auto-regressive(1) process. We examine a series of four consecutive timesteps, where only the endpoints are reported at the base station. The series produced by each scheme are:

- `Actual`: $\{-2.5, -3.5, -3.7, -2.7\}$
- `Standard`: $\{-2.5, \texttt{na}, \texttt{na}, -2.7\}$
- `Timestamp`: $\{-2.5, f, s, -2.7\}$

We have constructed 2-D joint scatter plots to show possible reconstruction combinations of the two missing values. Figure 6 shows a joint scatter plot of samples inferred using `Standard`, while Figure 7 shows samples from `Timestamp`. We observe `Timestamp` establishes stronger bounds, eliminating many possible reconstructions. Qualitatively, we see two scenarios: one where the second value rises above -2.2 and one where it falls beneath -2.8 (the latter scenario is the correct one). In either case, because the third value is a suppression, each of its samples must be within 0.3 of the sample generated for the second value.

Conch We have discussed a variety of ways to add redundancy to temporal suppression. For Conch, we again report not the changes in difference for each monitored edge, but the actual difference, to prevent failures from compounding over time. We sketch out one idea for adding redundancy to *Conch* that adds additional monitored edges. For example, in Figure 5, we can add edge $r \rightarrow c$ to provide another path from r to each of c, e, g, f and b. The edges are used to construct linear constraints on the true edge differences. For this example, $(r \rightarrow a) + (a \rightarrow c) = (r \rightarrow c)$. If this constraint does not hold, at least one of the three edges must have failed to report a change.

While detecting failure is important, it still leaves the problem of inferring the true node values. The more redundant edges added, the more evidence there is with which to produce the correct values, but of course at an added cost. It is not clear how to best add redundancy. This likely involves a combination of extra edges and the per-report additions used for temporal suppression. What edges should be added? Ideally, there should be multiple independent paths to each node. Further, edges meant to "cover" for one another must not have correlated failure, or else they provide no benefit and only extra cost.

3.3 Inference

The inference problem is to determine the actual sequence of values, or a sample of them, as in the examples depicted in Figures 6 and 7. This is quite complex and here we simply sketch the input, output, and strategies. First, the temporal suppression examples show how the suppression scheme, notably ϵ, and redundancy provide constraints on possible reconstructions of the raw values. Using constraints based on on evidence drawn from the network is a data-driven concept. We can also utilize models of node behavior, such as the AR(1) used in the examples. Even when values cannot be hard-bounded, such as with `Standard`, AR(1) dictates the actual values are unlikely to deviate far from timestep to timestep. This is visible in Figure 6, where the distributions for $x2$ and $x3$ are conditioned on the known values $x1$ and $x4$. Using a model in this way is clearly a model-driven concept; the samples are not derived purely using evidence drawn from the network. It is possible to control model reliance. We can diminish its importance relative to data-driven evidence by, for example, lowering ϵ or increasing redundancy. The samples are then influenced more by constraints than the model, but at increased cost. We see another fundamental trade-off, then, between energy cost and reliance on model. Aside from reconstructing the actual readings, we can also try to learn the model process parameters. In our deployment, where the end goal