PRESTO: Feedback-driven Data Management in Sensor Networks

Ming Li, Deepak Ganesan, and Prashant Shenoy
Department of Computer Science,
University of Massachusetts,
Amherst MA 01003.
{mingli,dganesan,shenoy}@cs.umass.edu

Abstract

This paper presents PRESTO, a novel two-tier sensor data management architecture comprising proxies and sensors that cooperate with one another for acquiring data and processing queries. PRESTO proxies construct time-series models of observed trends in the sensor data and transmit the parameters of the model to sensors. Sensors check sensed data with model-predicted values and transmit only deviations from the predictions back to the proxy. Such a model-driven push approach is energy-efficient, while ensuring that anomalous data trends are never missed. In addition to supporting queries on current data, PRESTO also supports queries on historical data using interpolation and local archival at sensors. PRESTO can adapt model and system parameters to data and query dynamics to further extract energy savings. We have implemented PRESTO on a sensor testbed comprising Intel Stargates and Telos Motes. Our experiments show that in a temperature monitoring application, PRESTO yields one to two orders of magnitude reduction in energy requirements over on-demand, proactive or model-driven pull approaches. PRESTO also results in an order of magnitude reduction in query latency in a 1% duty-cycled five hop sensor network over a system that forwards all queries to remote sensor nodes.

1 Introduction

1.1 Motivation

Networked data-centric sensor applications have become popular in recent years. Sensors sample their surrounding physical environment and produce data that is then processed, aggregated, filtered, and queried by the application. Sensors are often untethered, necessitating efficient use of their energy resources to maximize application lifetime. Consequently, energy-efficient data management is a key problem in sensor applications.

Data management approaches in sensor networks have centered around two competing philosophies. Early efforts such as Directed Diffusion [11] and Cougar [23] espoused the notion of the sensor network as a database. The framework assumes that intelligence is placed at the sensors and that queries are pushed deep into the network, possibly all the way to the remote sensors. Direct querying of remote sensors is energy efficient, since query processing is handled at (or close to) the data source, thereby reducing communication needs. However, the approach assumes that remote sensors have sufficient processing resources to handle query processing, an assumption that may not hold in untethered networks of inexpensive sensors (e.g., Berkeley Motes [19]). In contrast, efforts such as TinyDB [13] and acquisitional query processing [3] from the database community have adopted an alternate approach. These efforts assume that intelligence is placed at the edge of the network, while keeping the sensors within the core of the network simple. In this approach, data is pulled from remote sensors by edge elements such as base-stations, which are assumed to be less resource- and energy-constrained than remote sensors. Sensors within the network are assumed to be capable of performing simple processing tasks such as in-network aggregation and filtering, while complex query processing is left to base stations (also referred to as micro-servers or sensor proxies). In acquisitional query processing [3], for instance, the base-station uses a spatio-temporal model of the data to determine when to pull new values from individual sensors; data is refreshed from remote sensors whenever the confidence intervals on the model predictions exceed query error tolerances.

While both of these philosophies inform our present work, existing approaches have several drawbacks:

Need to capture unusual data trends: Sensor applications need to be alerted when unusual trends are observed in the sensor field; for instance, a sudden increase in temperature may indicate a fire or a break-down in air-conditioning equipment. Although rare, it is imperative for applications, particularly those used for monitoring, to detect these unusual patterns with low latency. Both

TinyDB [13] and acquisitional query processing [3] rely on a pull-based approach to acquire data from the sensor field. A pure pull-based approach can never guarantee that all unusual patterns will be always detected, since the anomaly may be confined between two successive pulls. Further, increasing the pull frequency to increase anomaly detection probability has the harmful side-effect of increasing energy consumption at the untethered sensors.

Support for archival queries: Many existing efforts focus on querying and processing of current (live) sensor data, since this is the data of most interest to the application. However, support for querying historical data is also important in many applications such as surveillance, where the ability to retroactively "go back" is necessary, for instance, to determine how an intruder broke into a building. Similarly, archival sensor data is often useful to conduct postmortems of unusual events to better understand them for the future. Architectures and algorithms for efficiently querying archival sensor data have not received much attention in the literature.

Adaptive system design: Long-lived sensor applications need to adapt to data and query dynamics while meeting user performance requirements. As data trends evolve and change over time, the system needs to adapt accordingly to optimize sensor communication overhead. Similarly, as the workload—query characteristics and error tolerance—changes over time, the system needs to adapt by updating the parameters of the models used for data acquisition. Such adaptation is key for enhancing the longevity of the sensor application.

1.2 Research Contributions

This paper presents PRESTO, a two-tier sensor architecture that comprises sensor proxies at the higher tier, each controlling tens of remote sensors at the lower tier. PRESTO[1] proxies and sensors interact and cooperate for acquiring data and processing queries [4]. PRESTO strives to achieve energy efficiency and low query latency by exploiting resource-rich proxies, while respecting constraints at resource-poor sensors. Like TinyDB, PRESTO puts intelligence at the edge proxies while keeping the sensors inside the network simple. A key difference though is that PRESTO endows sensors with the ability to asynchronously push data to proxies rather than solely relying on pulls. Our design of PRESTO has led to the following contributions.

Model-driven Push: Central to PRESTO is the use of a feedback-based model-driven push approach to support queries in an energy-efficient, accurate and low-latency manner. PRESTO proxies construct a model that captures correlations in the data observed at each sensor. The remote sensors check the sensed data against this model and push data only when the observed data deviates from the values predicted by the model, thereby capturing anomalous trends. Such a model-driven push approach reduces communication overhead by only pushing deviations from the observed trends, while guaranteeing that unusual patterns in the data are never missed. An important requirement of our model is that it should be very inexpensive to check at resource-poor sensors, even though it can be expensive to construct at the resource-rich proxies. PRESTO employs seasonal ARIMA-based time series models to satisfy this *asymmetric* requirement.

Support for archival queries: Whereas PRESTO supports queries on current data using model-driven push, it also supports queries on historical data using a novel combination of prediction, interpolation, and local archival. By associating confidence intervals with the model predictions and caching values predicted by the model in the past, a PRESTO proxy can directly respond to such queries using cached data so long as it meets query error tolerances. Further, PRESTO employs interpolation methods to progressively refine past estimates whenever new data is fetched from the sensors. PRESTO sensors also log all observations on relatively inexpensive flash storage; the proxy can fetch data from sensor archives to handle queries whose precision requirements can not be met using the local cache. Thus, PRESTO exploits the proxy cache to handle archival queries locally whenever possible and resorts to communication with the remote sensors only when absolutely necessary.

Adaptation to Data and Query Dynamics: Long-term changes in data trends are handled by periodically refining the parameters of the model at the proxy, which improves prediction accuracy and reduces the number of pushes. Changes in query precision requirements are handled by varying the threshold used at a sensor to trigger a push. If newer queries require higher precision (accuracy), then the threshold is reduced to ensure that small deviations from the model are reported to the proxy, enabling it to respond to queries with higher precision. Overall, PRESTO proxies attempt to balance the cost of pushes and the cost of pulls for each sensor.

We have implemented PRESTO using a Stargate proxy and Telos Mote sensors. We demonstrate the benefits of PRESTO using an extensive experimental evaluation. Our results show that PRESTO can scale up to one hundred Motes per proxy. When used in a temperature monitoring application, PRESTO imposes an energy requirements that is one to two orders of magnitude less than existing techniques that advocate on-demand, proactive, or model-driven pulls. At the same time, the average latency for queries is within six seconds for a 1% duty-cycled five hop sensor network, which is an order of

[1] PRESTO is an acronym for PREdictive STOrage.

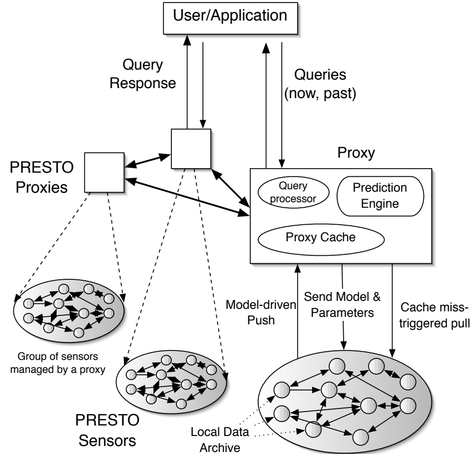


Figure 1: The PRESTO data management architecture.

magnitude less than a system that forwards all queries to remote sensor nodes, while not significantly more than a system where all queries are answered at the proxy.

The rest of this paper is structured as follows. Section 2 provides an overview of PRESTO. Sections 3 and 4 describe the design of the PRESTO proxy and sensors, respectively, while Section 5 presents the adaptation mechanisms in PRESTO. Sections 6 and 7 present our implementation and our experimental evaluation. Finally, Sections 8 and 9 discuss related work and our conclusions.

2 System Architecture

System Model: PRESTO envisions a two-tier data management architecture comprising a number of sensor proxies, each controlling several tens of remote sensors (see Figure 1). Proxies at the upper tier are assumed to be rich in computational, communication, and storage resources and can use them continuously. The task of this tier is to gather data from the lower tier and answer queries posed by users or the application. A typical proxy configuration may be comprised of an Intel Stargate [21] node with multiple radios—an 802.11 radio that connects it to an IP network and a low-power 802.15.4 radio that connects it to sensors in the lower tier. Proxies are assumed to be tethered or powered by a solar cell. A typical deployment will consist of multiple geographically distributed proxies, each managing tens of sensors in its vicinity. In contrast, PRESTO sensors are assumed to be low-power nodes, such as Telos Motes [18], equipped with one or more sensors, a microcontroller, flash storage and a wireless radio. The task of this tier is to sense data, transmit it to proxies when appropriate, while archiving all data locally in flash storage. The primary constraint at this tier is energy—sensor nodes are assumed to be untethered, and hence battery-powered, with a limited lifetime. Sensors are assumed to be deployed in a multi-hop configuration and are aggressively duty-cycled; standard multi-hop routing and duty-cycled MAC protocols can be used for this purpose. Since communication is generally more expensive than processing or storage [5], PRESTO sensors attempt to trade communication for computation or storage, whenever possible.

System Operation: Assuming such an environment, each PRESTO proxy constructs a model of the data observed at each sensor. The model uses correlations in the past observations to predict the value likely to be seen at any future instant t. The model and its parameters are transmitted to each sensor. The sensor then executes the model as follows: at each sampling instant t, the actual sensed value is compared to the value predicted by the model. If the difference between the two exceed a threshold, the model is deemed to have “failed” to accurately predict that value and the sensed value is pushed to the proxy. In contrast, if the difference between the two is smaller than a threshold, then the model is assumed to be accurate for that time instant. In this case, the sensor archives the data locally in flash storage and does not transmit it to the proxy. Since the model is *also known to the proxy*, the proxy can compute the predicted value and use it as an approximation of the actual observation when answering queries. Thus, so long as the model accurately predicts observed values, no communication is necessary between the sensor and the proxy; the proxy continues to use the predictions to respond to queries. Further, any deviations from the model are always reported to the proxy and anomalous trends are quickly detected as a result.

Given such a model-driven push technique, a query arriving at the proxy is processed as follows. PRESTO assumes that each query specifies a tolerance on the error it is willing to accept. Our models are capable of generating a *confidence interval* for each predicted value. The PRESTO proxy compares the query error tolerance with the confidence intervals and uses the model predictions so long at the query error tolerance is not violated. If the query demands a higher precision, the proxy simply pulls the actual sensed values from the remote sensors and uses these values to process the query. Every prediction made by the model is cached at the proxy; the cache also contains all values that were either pushed or pulled from the remote sensors. This cached data is used to respond to historical queries so long as query precision is not violated, otherwise the corresponding data is pulled from the local archive at the sensors.

Since trends in sensed values may change over time, a model constructed using historical data may no longer reflect current trends. A novel aspect of PRESTO is that it updates the model parameters online so that the model

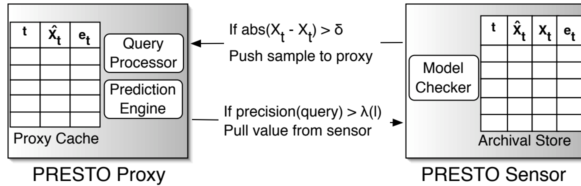


Figure 2: The PRESTO proxy comprises a prediction engine, query processor and a cache of predicted and real sensor values. The PRESTO sensor comprises a model checker and an archive of past samples with the model predictions.

can continue to reflect current observed trends. Upon receiving a certain number of updates from a sensor, the proxy uses these new values to refine the parameters of the model. These parameters are then conveyed back to the corresponding sensor, when then uses them to push subsequent values. Thus, our approach incorporates *active feedback* between the proxy and each sensor—the model parameters are used to determine which data values get pushed to the proxy, and the pushed values are used to compute the new parameters of the model. If the precision demanded by queries also changes over time, the threshold used by sensors to determine which values should be pushed are also adapted accordingly—higher precision results in smaller thresholds. Next, we present the design of the PRESTO proxy and sensor in detail.

3 PRESTO Proxy

The PRESTO proxy consists of four key components (see Figure 2): (i) *modeling and prediction engine*, which is responsible for determining the initial model parameters, periodic refinement of model parameters, and prediction of data values likely to be seen at the various sensors, (ii) *query processor*, which handles queries on both current and historical data, (iii) *local cache*, which is a cache of all data pushed or pulled by sensors as well as all past values predicted by the model, and (iv) a *fault detector*, which detects sensor failures. We describe each component in detail in this section.

3.1 Modeling and Prediction Engine

The goal of the modeling and prediction engine is to determine a model, using a set of past sensor observations, to forecast future values. The key premise is that the physical phenomena observed by sensors exhibit long-term and short-term correlations and past values can be used to predict the future. This is true for weather phenomena such as temperature that exhibit long-term seasonal variations as well as short-term time-of-day and hourly variations. Similarly phenomena such as traffic at an intersection exhibits correlations based on the hour of the day (e.g., traffic peaks during "rush" hours) and day of the week (e.g., there is less traffic on weekends). PRESTO proxies rely on *seasonal ARIMA* models; ARIMA is a popular family of time-series models that are commonly used for studying weather and stock market data. Seasonal ARIMA models (also known as SARIMA) are a class of ARIMA models that are suitable for data exhibiting seasonal trends and are well-suited for sensor data. Further they offer a way to deal with non-stationary data *i.e.* whose statistical properties change over time [1]. Last, as we demonstrate later, while seasonal ARIMA models are computationally expensive to construct, they are inexpensive to check at the remote sensors—an important property we seek from our system. The rest of this section presents the details of our SARIMA model and its use within PRESTO.

Prediction Model: A discrete time series can be represented by a set of time-ordered data $(x_{t_1}, x_{t_2}, ..., x_{t_n})$, resulting from observation of some temporal physical phenomenon such as temperature or humidity. Samples are assumed to be taken at discrete time instants $t_1, t_2, \ldots$. The goal of time-series analysis is to obtain the parameters of the underlying physical process that governs the observed time-series and use this model to forecast future values.

PRESTO models the time series of observations at a sensor as an *Autoregressive Integrated Moving Average (ARIMA)* process. In particular, the data is assumed to conform to the Box-Jenkins SARIMA model [1]. While a detailed discussion of SARIMA models is outside the scope of this paper, we provide the intuition behind these models for the benefit of the reader. An SARIMA process has four components: *auto-regressive (AR)*, *moving-average (MA)*, *one-step differencing*, and *seasonal differencing*. The AR component estimates the current sample as a linear weighted sum of previous samples; the MA component captures relationship between prediction errors; the one-step differencing component captures relationship between adjacent samples; and the seasonal differencing component captures the diurnal, monthly, or yearly patterns in the data. In SARIMA, the MA component is modeled as a zero-mean, uncorrelated Gaussian random variable (also referred to as white noise). The AR component captures the temporal correlation in the time series by modeling a future value as a function of a number of past values.

In its most general form, the Box-Jenkins seasonal model is said to have an order $(p, d, q) \times (P, D, Q)_S$; the order of the model captures the dependence of the predicted value on prior values. In SARIMA, p and q are the orders of the auto-regressive (AR) and moving average (MA) processes, P and Q are orders of the seasonal AR and MA components, d is the order of differencing, D is

the order of seasonal differencing, and S is the seasonal period of the series. Thus, SARIMA is family of models depending on the integral values of p, q, P, Q, d, D, S. [2]

Model Identification and Parameter Estimation: Given the general SARIMA model, the proxy needs to determine the order of the model, including the order of differential and the order of auto-regression and moving average. That is, the values of p, d, q, P, D and Q need to be determined. This step is called model identification and is typically performed once during system initialization. Model identification is well documented in most time series textbooks [1] and we only provide a high level overview here. Intuitively, since the general model is actually a family of models, depending on the values of p, q, etc., this phase identifies a particular model from the family that best captures the variations exhibited by the underlying data. It is somewhat analogous to fitting a curve on a set of data values. Model identification involves collecting a sample time series from the field and computing its auto-correlation function (ACF) and partial auto-correlation function (PACF). A series of tests are then performed on the ACF and the PACF to determine the order of the model [1].

Our analysis of temperature traces has shown that the best model for temperature data is a Seasonal ARIMA of order $(0,1,1) \times (0,1,1)_S$. The general model in Equation 1 reduces to

$$(1-B)(1-B^S)X_t = (1-\theta B)(1-\Theta B^S)e_t \quad (2)$$

where θ and Θ are parameters of this $(0,1,1) \times (0,1,1)_S$ SARIMA model and capture the variations shown by different temperature traces. B is the backward operator and is short-hand for $B^i X_t = X_{t-i}$. S is the seasonal period of the time series and e_t is the prediction error.

When employed for a temperature monitoring application, PRESTO proxies are seeded with a $(0,1,1) \times (0,1,1)_S$ SARIMA model. The seasonal period S is also seeded. The parameters θ and Θ are then computed by the proxy during the initial training phase before the system becomes operational. The training phase involves gathering a data set from each sensor and using the least squares method to estimate the values of parameters θ and Θ on a per-sensor basis (see [1] for the detailed procedure for estimating these parameters). The order of the model and the values of θ and Θ are then conveyed to each sensor. Section 5 explains how θ and Θ can be periodically refined to adapt to any long-term changes in the sensed data that occurs after the initial training phase.

Model-based Predictions: Once the model order and its parameters have been determined, using it for predicting future values is a simple task. The predicted value X_t for time t is simply given as:

$$\begin{aligned} X_t &= X_{t-1} + X_{t-S} - X_{t-S-1} \\ &+ \theta e_{t-1} - \Theta e_{t-S} + \theta\Theta e_{t-S-1} \end{aligned} \quad (3)$$

where θ and Θ are known parameters of the model, X_{t-1} denotes the previous observation, X_{t-S} and X_{t-S-1} denotes the values seen at this time instant and the previous time instant in the previous season. For temperature monitoring, we use a seasonal period S of one day, and hence, X_{t-S} and X_{t-S-1} represent the values seen *yesterday* at this time instant and the previous time instant, respectively. e_{t-k} denotes the prediction error at time $t-k$ (the prediction error is simply the difference between the predicted and observed value for that instant).

Since PRESTO sensors push a value to the proxy only when it deviates from the prediction by more than a threshold, the actual values of X_{t-1}, X_{t-S} and X_{t-S-1} seen at the sensor may not be known to the proxy. However, since the lack of a push indicates that the model predictions are accurate, the proxy can simply use the corresponding model predictions as an approximation for the actual values in Equation 3. In this case, the corresponding prediction error e_{t-k} is set to zero. In the event X_{t-1}, X_{t-S} or X_{t-S-1} were either pushed by the sensor or pulled by the proxy, the actual values and the actual prediction errors can used in Equation 3.

Both the proxy and the sensors use Equation 3 to predict each sampled value. At the proxy, the predictions serve as a substitute for the actual values seen by the sensor and are used to answer queries that might request the data. At the sensor, the prediction is used to determine whether to push—the sensed value is pushed only if the prediction error exceeds a threshold δ.

Finally, we note the *asymmetric* property of our model. The initial model identification and parameter estimation is a compute-intensive task performed by the proxy. Once determined, predicting a value using the model *involves no more than eight floating point operations* (three multiplications and five additions/subtractions, as shown in Equation 3). This is inexpensive even on resource-poor sensor nodes such as Motes and can be approximated using fixed point arithmetic.

3.2 Query Processing at a Proxy

In addition to forecasting future values, the prediction engine at the proxy also provides a confidence interval for each predicted value. The confidence interval represents a bound on the error in the predicted value and is crucial for query processing at the proxy. Since each

[2] While not essential for our discussion, we present the general Box-Jenkins seasonal model for sake of completeness. The general model of order $(p,d,q) \times (P,D,Q)_S$ is given by the equation

$$\Phi_P(B^S) \cdot \phi_p(B) \cdot (1-B)^d (1-B^S)^D X_t = \theta_q(B)\Theta_Q(B^S)e_t \quad (1)$$

where B is the backward operator such that $B^i X_t = X_{t-i}$, S is the seasonal period, θ, Θ are parameters of the model, and e_t is the prediction error.

query arrives with an error tolerance, the proxy compares the error tolerance of a query with the confidence interval of the predictions, and the current push threshold, δ. If the confidence interval is tighter than the error tolerance, then the predicted values are sufficiently accurate to respond to the query. Otherwise the actual value is fetched from the remote sensor to answer the query. Thus, many queries can be processed locally even if the requested data was never reported by the sensor. As a result, PRESTO can ensure low latencies for such queries without compromising their error tolerance. The processing of queries in this fashion is similar to that proposed in the BBQ data acquisition system [3], although there are significant differences in the techniques.

For a Seasonal ARIMA $(0,1,1) \times (0,1,1)_S$ model, the confidence interval of l step ahead forecast, $\lambda(l)$ is:

$$\lambda(l) = \pm u_{\varepsilon/2}(1 + \sum_{j=1}^{l-1}(1-\theta)^2)^{1/2}\sigma \quad (4)$$

where $u_{\varepsilon/2}$ is value of the unit Normal distribution at $\varepsilon/2$, σ is the variance of 1 step ahead prediction error.

3.3 Proxy Cache

Each proxy maintains a cache of previously fetched or predicted data values for each sensor. Since storage is plentiful at the proxy—microdrives or hard-drives can be used to hold the cache—the cache is assumed to be infinite and all previously predicted or fetched values are assumed to be stored at the proxy. The cache is used to handle queries on historical data—if requested values have already been fetched or if the error bounds of cached predictions are smaller than the query error tolerance, then the query can be handled locally, otherwise the requested data is pulled from the archive at the sensor. After responding to the query, the newly fetched values are inserted into the cache for future use.

A newly fetched value, upon insertion, is also used to improve the accuracy of the neighboring predictions using interpolation. The intuition for using interpolation is as follows. Upon receiving a new value from the sensor, suppose that the proxy finds a certain prediction error. Then it is very likely that the predictions immediately preceding and following that value incurred a similar error, and interpolation can be used to scale those cached values by the prediction error, thereby improving their estimates. PRESTO proxies currently use two types of interpolation heuristics: forward and backward.

Forward interpolation is simple. The proxy uses Equation 3 to predict the values and Equation 4 to re-estimate the confidence intervals for all samples between the newly inserted value and the next pulled or pushed value. In backward interpolation, the proxy scans backwards from the newly inserted value and modifies all cached predictions between the newly inserted value and the previous pushed or pulled value. To do so, it makes a simplifying assumption that the prediction error grows linearly at each step, and the corresponding prediction error is subtracted from each prediction.

$$X'_t = X_t - \frac{t - T'}{T - T'} e_T \quad (5)$$

where X_t is the original prediction, X'_t is the updated prediction, T denotes the observation instant of the newly inserted value, T' is time of the nearest pushed or pulled value before T.

3.4 Failure Detection

Sensors are notoriously unreliable and can fail due hardware/software glitches, harsh deployment conditions or battery depletion. Our predictive techniques limit message exchange between a proxy and a sensor, thereby reducing communication overhead. However, reducing message frequency also affects the latency to detect sensor failures and to recover from them. In this work, we discuss mechanisms used by the PRESTO proxy to detect sensor failures. Failure recovery can use techniques such as spatial interpolation, which are outside the scope of this paper.

The PRESTO proxy flags a failure if pulls or feedback messages are not acknowledged by a sensor. This use of implicit heartbeats has low communication energy overhead, but provides an interesting benefit. A pull is initiated by the proxy depending on the confidence bounds, which in turn depends on the variability observed in the sensor data. Consequently, failure detection latency will be lower for sensors that exhibit higher data variability (resulting in more pushes or pulls). For sensors that are queried infrequently or exhibit low data variability, the proxy relies on the less-frequent model feedback messages for implicit heartbeats; the lack of an acknowledgment signals a failure. Thus, proxy-initiated control or pull messages can be exploited for failure detection at no additional cost; the failure detection latency depends on the observed variability and confidence requirements of incoming queries. Explicit heartbeats can be employed for applications with more stringent needs.

4 PRESTO Sensor

PRESTO sensors perform three tasks: (i) use the model predictions to determine which observations to push, (ii) maintain a local archive of all observations, and (iii) respond to pull requests from the proxy.

The PRESTO sensor acts as a mirror for the prediction model at the proxy—both the proxy and the sensor execute the model in a completely identical fashion. Consequently, at each sampling instant, the sensor knows the

exact estimate of the sampled value at the proxy and can determine whether the estimate is accurate. Only those samples that deviate significantly from the prediction are pushed. As explained earlier, the proxy transmits all the parameters of the model to each sensor during system initialization. In addition, the proxy also specifies a threshold δ that defines the worst-case deviation in the model prediction that the proxy can tolerate. Let X_t denote the actual observation at time t and let $\hat{X}_t$ denote the predicted value computed using Equation 3. Then,

$$\text{If } |\hat{X}_t - X_t| > \delta, \text{Push } X_t \text{ to Proxy.} \quad (6)$$

As indicated earlier, computation of $\hat{X}_t$ using Equation 3 involves reading of a few past values such as X_{t-S} from the archive in flash storage and a few floating point multiplications and additions, all of which are inexpensive.

PRESTO sensors archive all sensed values into an energy-efficient NAND flash store; the flash archive is an append-only log of tuples of the form: $(t, X_t, \hat{X}_t, e_t)$. A simple index is maintained to permit random access to any entry in the log. A pull request from a proxy involves the use of this index to locate the requested data in the archive, followed by a read and a transmit.

5 Adaptation in PRESTO

PRESTO is designed to adapt to long-term changes in data and query dynamics that occur in any long-lived sensor application. To enable system operation at the most energy-efficient point, PRESTO employs active feedback from proxies to sensors; this feedback takes two forms—adaptation to data and query dynamics.

5.1 Adaptation to Data Dynamics

Since trends in sensor observation may change over time, a model constructed using historical data may no longer reflect current trends—the model parameters become stale and need to be updated to regain energy-efficiency. PRESTO proxies periodically retrain the model in order to refine its parameters. The retraining phase is similar to the initial training—all data since the previous retraining phase is gathered and the least squares method is used to recompute the model parameters θ and Θ [1]. The key difference between the initial training and the retraining lies in the data set used to compute model parameters.

For the initial training, an actual time series of sensor observations is used to compute model parameters. However, once the system is operational, sensors only report observations when they significantly deviate from the predicted values. Consequently, the proxy only has access to a small subset of the observations made at each sensor. Thus, the model must be retrained with *incomplete information*. The time series used during the retraining phase contains all values that were either pushed or pulled from a sensor; all missing values in the time series are substituted by the corresponding model predictions. Note that these prior predictions are readily available in the proxy cache; furthermore, they are guaranteed to be a good approximation of the actual observations (since these are precisely the values for which the sensor did not push the actual observations). This approximate time series is used to retrain the model and recompute the new parameters.

For the temperature monitoring application that we implemented, the models are retrained at the end of each day.[3] The new parameters θ and Θ are then pushed to each sensor for future predictions. In practice, the parameters need to be pushed only if they deviate from the previously computed parameters by a non-trivial amount (i.e., only if the model has actually changed).

5.2 Adaptation to Query Dynamics

Just as sensor data exhibits time-varying behavior, query patterns can also change over time. In particular, the query tolerance demanded by queries may change over time, resulting in more or fewer data pulls. The proxy can adapt the value of the threshold parameter δ in Equation 6 to directly influence the fraction of queries that trigger data pulls from remote sensors. If the threshold δ is large relative to the mean error tolerance of queries, then the number of pushes from the sensor is small and the number of pulls triggered by queries is larger. If δ is small relative to the query error tolerance, then there will be many wasteful pushes and fewer pulls (since the cached data is more precise than is necessary to answer the majority of queries). A careful selection of the threshold parameter δ allows a proxy to balance the number of pushes and the number of pulls for each sensor.

To handle such query dynamics, the PRESTO proxy uses a moving window average to track the mean error tolerance of queries posed on the sensor data. If the error tolerance changes by more than a pre-defined threshold, the proxy computes a the new δ and transmits it to the sensor so that it can adapt to the new query pattern.

6 PRESTO Implementation

We have implemented a prototype of PRESTO on a multi-tier sensor network testbed. The proxy tier employs Crossbow Stargate nodes with a 400MHz Intel XScale processor and 64MB RAM. The Stargate runs the Linux 2.4.19 kernel and EmStar release 2.1 and is equipped with two wireless radios, a Cisco Aironet 340-based 802.11b radio and a hostmote bridge to the Telos mote sensor nodes using the EmStar transceiver. The

[3] Since the seasonal period is set to one day, this amounts to a retraining after each season.

sensor tier uses Telos Mote sensor nodes, each consisting of a MSP430 processor, a 2.4GHz CC2420 radio, and 1MB external flash memory. The sensor nodes run TinyOS 1.1.14. Since sensor nodes may be several hops away from the nearest proxy, the sensor tier employs MultiHopLEPSM multi-hop routing protocol from the TinyOS distribution to communicate with the proxy tier.

Sensor Implementation: Our PRESTO implementation on the Telos Mote involves three major tasks: (i) model checking, (ii) flash archival, and (ii) data pull. A simple data gathering task periodically obtains sensor readings and sends the sample to the model checker. The model checking task uses the most recent model parameters (θ and Θ) and push delta (δ) obtained from the proxy to determine if a sample should be pushed to the proxy as per Equation 6. Each push message to the proxy contains the id of the mote, the sampled data, and a timestamp recording the time of the sampling. Upon a pull from the proxy, the model checking task performs the forward and backward updates to ensure consistency between the proxy and sensor view. For each sample, the archival task stores a record to the local flash that has three fields: (i) the timestamp when the data was sampled, (ii) the sample itself, and (iii) the predicted value from the model checker. The final component of our sensor implementation is a pull task that, upon receiving a pull request, reads the corresponding data from the flash using a temporal index-based search, and responds to the proxy.

Proxy Implementation: At the core of the proxy implementation is the prediction engine. The prediction engine includes a full implementation of ARIMA parameter estimation, prediction and update. The engine uses two components, a cache of real and predicted samples, and a protocol suite that enables interactions with each sensor. The proxy cache is a time-series stream of records, each of which includes a timestamp, the predicted sensor value, and the prediction error. The proxy uses one stream per node that it is responsible for, and models each node's data separately. The prediction engine communicates with each sensor using a protocol suite that enables it to provide feedback and change the operating parameters at each sensor.

Queries on our system are assumed to be posed at the appropriate proxy using either indexing [5] or routing [12] techniques. A query processing task at the proxy accepts queries from users, checks whether it can be answered by the prediction engine based on the local cache. If not, a pull message is send to the corresponding sensor.

Our proxy implementation includes two enhancements to the hostmote transceiver that comes with the EmStar distribution [6]. First, we implemented a priority-based 64-length FIFO outgoing message queue in the transceiver to buffer pull requests to the sensors. There are two priority levels — the highest priority corresponds to parameter feedback messages to the sensor nodes, and the lower priority corresponds to data pull messages. Prioritizing messages ensures that parameter messages are not dropped even if the queue is full as a result of excess pulls. Our second enhancement involves emulating the latency characteristics of a duty-cycling MAC layer. Many MAC-layer protocols have been proposed for sensor networks such as BMAC [17] and SMAC [24]. However, not all these MAC layers are supported on all platforms — for instance, no duty-cycling scheme is currently supported on the Telos Motes that we use. We address this issue by benchmarking the latency introduced by BMAC on Mica2 sensor nodes, and using these measurements to drive our experiments. Thus, the proxy implementation includes a MAC-layer emulator that adds duty-cycling latency corresponding to the chosen MAC duty-cycling parameters.

7 Experimental Evaluation

In this section, we evaluate the performance of PRESTO using our prototype and simulations. The testbed for our experiments comprises one Stargate proxy and twenty Telos Mote sensor nodes. One of the Telos motes is connected to a Stargate node running the EmStar sensor network emulator [8]. The EmStar emulator enables us to introduce additional virtual sensor nodes in our large-scale experiments that share a single Telos mote radio as the transceiver to send and receive messages. In addition to the testbed, we use numerical simulations in Matlab to evaluate the performance of the data processing algorithms in PRESTO.

PRESTO sensors in our testbed are seeded with two temperature traces—a seven day temperature dataset from James reserve [22] (also used in our simulations) and a four day outdoor data from a live PRESTO deployment at UMass. The first two days are used to train the model for the James reserve dataset, and the first day is used for training in the UMass deployment. In our experiments, sensors use the values from the remainder of these traces—which are stored in flash memory—as a substitute for live data gathering. This setup ensures repeatable experiments and comparison of results across experiments (which were conducted over a period of several weeks). We also experiment with a live, outdoor deployment of PRESTO to demonstrate that our results are representative of the "real world".

In order to evaluate the query processing performance of PRESTO, we generate queries as a Poisson arrival process. Each query requests the value of the temperature at a particular time that is picked in a uniform random manner from the start of the experiment to the current time. The confidence interval requested by the query is chosen from a normal distribution.

7.1 Microbenchmarks

Our first experiment involves a series of microbenchmarks of the energy consumption of communication, processing and storage to evaluate individual components of the PRESTO proxy and sensors. These microbenchmarks are based on measurements of two sensor platforms — a Telos mote, and a Mica2 mote augmented with a NAND flash storage board fabricated at UMass. The board is attached to the Mica2 mote through the standard 51-pin connector, and provides a considerably more energy-efficient storage option than the AT45DB041B NOR flash that is loaded by default on the Mica2 mote [15]. The NAND flash board enables the PRESTO sensor to archive a large amount of historical data at extremely low energy cost.

Module	Component	Operation	Energy
NAND flash-enabled Mica2	NAND Flash	Read + Write + Erase 1 sample	21nJ
	ATmega128L Processor	1 Prediction	240nJ
	CC1000 Radio	Transmit 1 sample + Receive 1 ACK	20.3μJ
Telos Mote	ST M25P80 Flash	Read + Write + Erase 1 sample	2.14μJ
	MSP430 Processor	1 Prediction	27nJ
	CC2420 Radio	Transmit 1 sample + Receive 1 ACK	3.3μJ

Table 1: Energy micro-benchmarks for sensor nodes.

	Round Trip Latency(ms)		
Routing Hops	1%	7.53%	35.5%
1-hop	2252	350	119
2-hop	4501	695	235
3-hop	6750	1040	347
4-hop	8999	1388	465
5-hop	11249	1733	580

Table 2: Round trip latencies using B-MAC

Energy Consumption: We measure the energy consumption of three components—computation per sample at the sensor, communication for a push or pull, and storage for reads, writes and erases. Table 1 shows that the results depend significantly on the choice of platform. On the Mica2 mote with external NAND flash, storage of a sample in flash is an order of magnitude more efficient than the ARIMA prediction computation, and three orders of magnitude more efficient than communicating a sample over the CC1000 radio. The Telos mote uses a more energy-efficient radio (CC2420) and processor (TI MSP 430), but a less efficient flash than the modified Mica2 mote. On the Telos mote, the prediction computation is the most energy-efficient operation, and is 80 times more efficient than storage, and 122 times more efficient than communication. The high cost of storage on the Telos mote makes it a bad fit for a storage-centric architecture such as PRESTO.

In order to fully exploit state-of-art in computation, communication and storage, a new platform is required that combines the best features of the two platforms that we have measured. This platform would use the TI MSP 430 microcontroller and CC2420 radio on the Telos mote together with NAND flash storage. Assuming that the component-level microbenchmarks in Table 1 hold for the new platform, storage and computation would be roughly equal cost, whereas communication would be two to three orders of magnitude more expensive than both storage and communication. We note that the energy requirements for communication in all the above benchmarks would be even greater if one were to include the overhead due to duty-cycling, packet headers and multi-hop routing. These comparisons validate our key premise that in future platforms, storage will offer a more energy-efficient option than communication and should be exploited to achieve energy-efficiency.

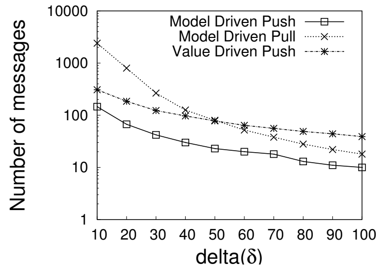
Component	Operation	Latency	Energy
Stargate (PXA255)	Model Estimation	21.75ms	11mJ
Telos Mote (MSP430)	Predict One Sample	18μs	27nJ

Table 3: Asymmetry: Model estimation vs Model checking

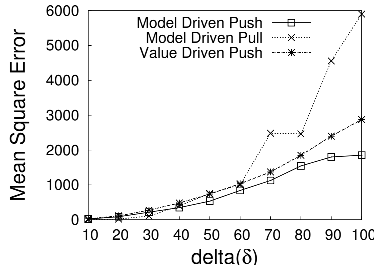
Communication Latency: Our second microbenchmark evaluates the latency of directly querying a sensor node. Sensor nodes are often highly duty-cycled to save energy, *i.e.* their radios are turned off to reduce energy use. However, as shown in Table 2, better duty-cycling corresponds to increased duration between successive wakeups and worse latency for the CC1000 radio on the Mica2 node. For typical sensor network duty-cycles of 1% or less, the latency is of the order of many seconds even under ideal 100% packet delivery conditions. Under greater packet-loss rates that is typical of wireless sensor networks [25], this latency would increase even further. We are unable to provide numbers for the CC2420 radio on the Telos mote since there is no available TinyOS implementation of an energy-efficient MAC layer with duty-cycling support for this radio.

Our measurements validate our claim that directly querying a sensor network incurs high latency, and this approach may be unsuitable for interactive querying. To reduce querying latency, the proxy should handle as many of the queries as possible.

Asymmetric Resource Usage: Table 3 demonstrates how PRESTO exploits computational resources at the proxy and the sensor. Determining the parameters of the ARIMA model at the proxy is feasible for a Stargate-class device, and requires only 21.75 ms per sensor. This operation would be very expensive, if not infeasible, on



(a) Communication Cost



(b) Mean Square Error

Figure 3: Comparison of PRESTO SARIMA models with model-driven pull and value-driven push.

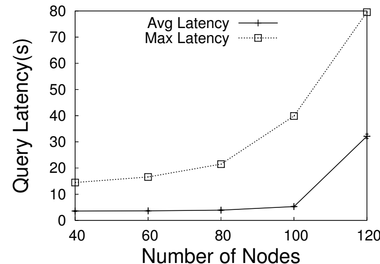


Figure 4: Scalability of PRESTO: Impact of network size.

a Telos Mote due to resource limitations. In contrast, checking if the model is correct at the Mote consumes considerably less energy since it consists of only three floating point multiplications (approximated using fixed point arithmetic) and five additions/subtractions corresponding to Equation 3. This validates the design choice in PRESTO to separate model-building from model-checking and to exploit proxy resources for the former and resources at the sensor for the latter.

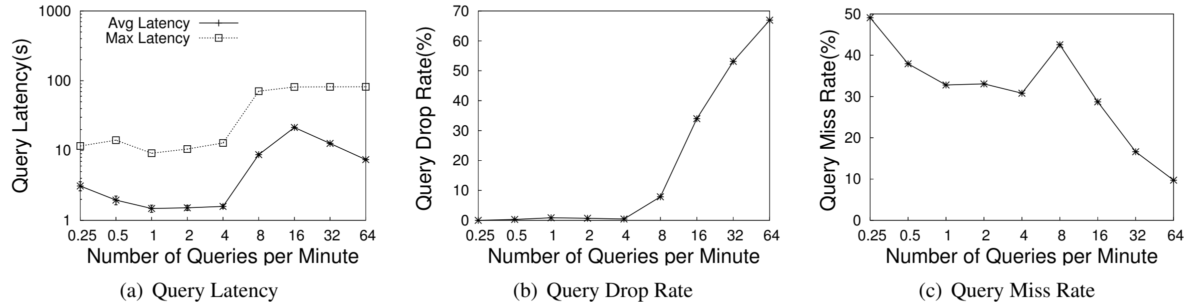
Summary: Our microbenchmarks validate three design choices made by PRESTO—the need for a storage-centric architecture that exploits energy-efficient NAND flash storage, the need for proxy-centric querying to deal with high latency of duty-cycled radios, and exploiting proxy resources to construct models while performing only simple model-checking at the sensors.

7.2 Performance of Model-Driven Push

In this section, we validate our claim that intelligently exploiting both proxy and sensor resources offers greater energy benefit than placing intelligence only at the proxy or only at the sensor. We compare the performance of model-driven push used in PRESTO against two other data-acquisition algorithms. The first algorithm, model-driven pull, is representative of the class of techniques where intelligence is placed solely at the proxy. This algorithm is motivated by the approach proposed in BBQ [3]. In this algorithm, the proxy uses a model of sensor data to predict future data and estimate the confidence interval in the prediction. If the confidence interval exceeds a pre-defined threshold (δ), the proxy will pull data from the sensor nodes, thus keeping the confidence interval bounded. The sensor node is simple in this case, and performs neither local storage nor model processing. While BBQ uses multi-variate Gaussians and dynamic Kalman Filters in their model-driven pull, our model-driven pull uses ARIMA predictions to ensure that the results capture the essential difference between the techniques and not the difference between the models used. The second algorithm that we compare against is a relatively naive value-driven push. Here, the sensor node pushes the data to the proxy when the difference between current data and last pushed data is larger than a threshold (δ). The proxy assumes that the sensor value does not change until the next push from the sensor. In general, a pull requires two messages, a request from the proxy to the sensor and a response, whereas push requires only a single message from the sensor to the proxy.

We compared the three techniques using Matlab simulations that use real data traces from James Reserve. Each experiment uses 5 days worth of data and each data point is the average of 10 runs. Figure 3 compares these three techniques in terms of the number of messages transmitted and mean-square error of predictions. In communication cost, PRESTO out-performs both the other schemes irrespective of the choice of δ. When δ is 100, the communication cost of PRESTO is half that of model-driven pull, and 25% that of value-driven push. At the same time, the mean square error in PRESTO is 30% that of model-driven pull, and 60% that of value driven push. As δ decreases, the communication cost increases for all three algorithms. However, the increase in communication cost for model-driven pull is higher than that for the other two algorithms. When δ is 50, value driven push begins to out perform model-driven pull. When δ reaches 10, the number of messages in model-driven pull is 20 times more than that of PRESTO, and 8 times more than that of value driven push. This is because in the case of model-driven pull, the proxy pulls samples from the sensor whenever the prediction error exceeds δ. However, since the prediction error is often an overestimate and since each pull is twice as expensive as a push, this results in a larger number of pull messages compared to PRESTO and value-driven push. The accuracies of the three algorithms become close to each other when δ decreases. When δ is smaller than 40, model-driven pull has slightly lower mean square error than PRESTO but incurs 4 times the number of messages.

Summary: These performance numbers demonstrate that model-driven push combines the benefits of both



(a) Query Latency (b) Query Drop Rate (c) Query Miss Rate

Figure 5: Scalability of PRESTO: Impact of query rates.

proxy-centric as well as sensor-centric approaches. It is 2-20 times more energy-efficient and upto 3 times more accurate than proxy-centric model-driven pull. In addition, PRESTO is upto 4 times more energy-efficient than a sensor-centric value-driven push approach.

7.3 PRESTO Scalability

Scalability is an important criteria for sensor algorithm design. In this section, we evaluate scalability along two axes — network size and the number of queries posed on a sensor network. Network size can vary depending on the application (e.g: the Extreme Scaling deployment [7] used 10,000 nodes, whereas the Great Duck Island deployment [14] used 100 nodes). The querying rate depends on the popularity of sensor data, for instance, during an event such as an earthquake, seismic sensors might be heavily queried while under normal circumstances, the query load can be expected to be light.

The testbed used in the scalability experiments comprises one Stargate proxy, twenty Telos mote sensor nodes, and an EmStar emulator that enables us to introduce additional virtual sensor nodes and perform larger scale experiments. Messages are exchanged between each sensor and the proxy through a multihop routing tree rooted at the proxy. Each sensor node is assumed to be operating at 1% duty-cycling. Since MAC layers that have been developed for the Telos mote do not currently support duty-cycling, we emulate a duty-cycling enabled MAC-layer. This emulator adds appropriate duty-cycling latency to each packet based on the microbenchmarks that we presented in Table 2.

7.3.1 Impact of Network Size

A good data management architecture should achieve energy-efficiency and low-latency performance even in large scale networks. Our first set of scalability experiments test PRESTO at different system scales on five days of data collected from the James Reserve deployment. Queries arrive at the proxy as as a Poisson process at the rate of one query/minute per sensor. The confidence interval of queries is chosen from a normal distribution, whose expectation is equal to the push threshold, $\delta = 100$.

Figure 4 shows the query latency and query drop rate at system sizes ranging from 40 to 120. For system sizes of less than 100, the average latency is always below five seconds and has little variation. When the system size reaches 120, the average latency increases five-fold to 30 seconds. This is because the radio transceiver on the proxy gets congested and the queue overflows.

The effect of duty-cycling on latency is seen in Figure 4, which shows that the maximum latency increases with system scale. The maximum latency corresponds to the worst case of PRESTO when a sequence of query misses occur and result in pulls from sensors. This results in queuing of queries at the proxy, and hence greater latency. An in-network querying mechanism such as Directed Diffusion [11] that forwards every query into the network would incur even greater latency than the worst case in PRESTO since every query would result in a pull. These experiments demonstrate the benefits of model-driven pushes for user queries. By the use of caching and models, PRESTO results in low average-case latency by providing quick responses at the proxy for a majority of queries. We note that the use of a tiered architecture makes it easy to expand system scale to many hundreds of nodes by adding more PRESTO proxies.

7.3.2 Impact of Query Rate

Our second scalability experiment stresses the query handling ability of PRESTO. We test PRESTO in a network comprising one Stargate proxy and twenty Telos mote sensor nodes under different query rates ranging from one query every four minutes to 64 queries/minute for each sensor. Each experiment is averaged over one hour. We measure scalability using three metrics: the query latency, query miss rate, and query drop rate. A query miss corresponds to the case when it cannot be answered at the proxy and results in a pull, and a query drop

results from an overflow at the proxy queue.

Figure 5 shows the result of the interplay between model accuracy, network congestion, and queuing at the proxy. To better understand this interplay, we analyze the graphs in three parts, *i.e.*, 0.25-4 queries/minute, 4-16 queries/minute and beyond 16 queries/minute.

Region 1: Between 0.25 and 4 queries/minute, the query rate is low, and neither queuing at the proxy nor network congestion is a bottleneck. As the query rate increases, greater number of queries are posed on the system and result in a few more pulls from the sensors. As a consequence, the accuracy of the model at the proxy improves to the point where it is able to answer most queries. This results in a reduction in the average latency. This behavior is also reflected in Figure 5(c), where the query miss rate reduces as the rate of queries grows.

Region 2: Between 4 and 16 queries/minute, the query rate is higher than the rate at which queries can be transmitted into the network. The queue at the proxy starts building, thereby increasing latency for query responses. This results in a sharp increase in average latency and maximum latency, as shown in Figure 5(a). This increase is also accompanied by an increase in query drop rate beyond eight queries/minute, as more queries are dropped due to queue overflow. We estimate that eight queries/minute is the breakdown threshold for our system for the parameters chosen.

Region 3: Beyond sixteen queries/minute, the system drops a significant fraction of queries due to queue overflow as shown in Figure 5(b). Strangely, for the queries that do not get dropped, both the average latency (Figure 5(a)), and the query miss rate (Figure 5(c)) drop! This is because with each pull, the model precision improves and it is able to answer a greater fraction of the queries accurately.

The performance of PRESTO under high query rate demonstrates one of its key benefits — the ability to use the model to alleviate network congestion and queuing delays. This feature is particularly important since sensor networks can only sustain a much lower query rate than tethered systems due to limited wireless bandwidth.

Summary: We show that PRESTO scales to around hundred nodes per proxy, and can handle eight queries per minute with query drop rates of less than 5% and average latency of 3-4 seconds per query.

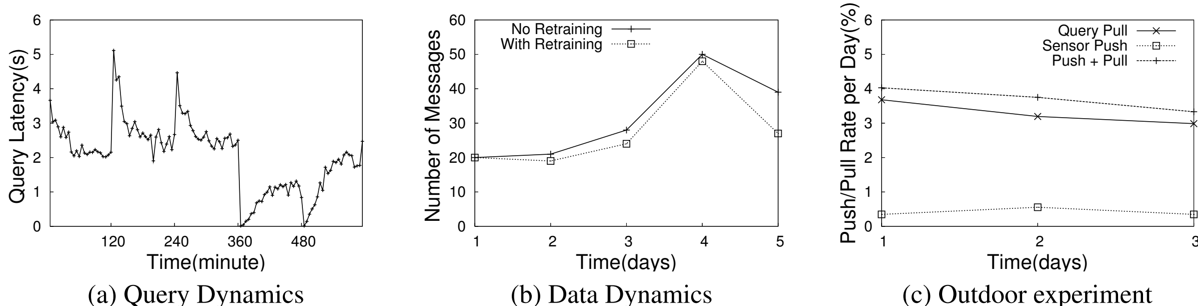
7.4 PRESTO Adaptation

Having demonstrated the scalability and energy efficiency of PRESTO, we next evaluate its adaptation to query and data dynamics. In general, adaptation only changes what the sensor does for future data and not for past data. Our experiments evaluate adaptation for queries that request data from the recent past (one hour).

In our first experiment, we run PRESTO for 12 hours. Every two hours, we vary the mean of the distribution of query precision requirements thereby varying the query error tolerance. The proxy tracks the mean of the query distribution and notifies the sensor if the mean changes by more than a pre-defined threshold, in our case, 10. Figure 6(a) shows the adaptation to the query distribution changes. Explicit feedback from the proxy to each sensor enables the system to vary the δ corresponding to the changes in query precision requirements. From the figure, we can see that there is a spike in average query latency and the energy cost every time the query confidence requirements become tighter. This results in greater query miss rate and hence more pulls as shown in Figure 6(a). However, after a short period, the proxy provides feedback to the sensor to change the pushing threshold, which decreases the query miss rate and consequently, the average latency. The opposite effect is seen when the query precision requirements reduce, such as at the 360 minute mark in Figure 6(a). As can be seen, the query miss rate reduces dramatically since the model at the proxy is too precise. After a while, the proxy provides feedback to the sensors to increase the push threshold and to lower the push rate. A few queries result in pulls as a consequence, but the overall energy requirements of the system remains low. In comparison with a non-adaptive version of PRESTO that kept a fixed δ, our adaptive version reduces latency by more than 50%.

In our second experiment, we demonstrate the benefits of adaptation to data dynamics. PRESTO adapts to data dynamics by model retraining, as described in Section 5. We use a four day dataset, and at the end of each day, the proxy retrains the model based on the pushes from the sensor for the previous day, and provides feedback of the new model parameters to the sensor. Our result is shown in Figure 6(b). For instance, on day three, the data pattern changes considerably and the communication cost increases since the model does not follow the old patterns. However, at the end of the third day, the PRESTO proxy retrains the model and send the new parameters to the sensors. As a result, the model accuracy improves on the second day and reduces communication. The figure also shows that the model retraining reduces pushes by as much as 30% as compared to no retraining.

While most of our experiments involved the use of temperature traces as a substitute of live temperature sampling, we conducted a number of experiments with a live outdoor deployment of PRESTO using one proxy and four sensors. These experiments corroborate our findings from the trace-driven testbed experiments. The result of one such experiment is shown in Figure 6(c). The figure shows that, over a period of three days, as the model adapts via retraining, the frequency of pulls as well as the total frequency of pushes and pulls falls.



(a) Query Dynamics (b) Data Dynamics (c) Outdoor experiment

Figure 6: Adaptation in PRESTO to data and query dynamics as well as adaptation in an outdoor deployment.

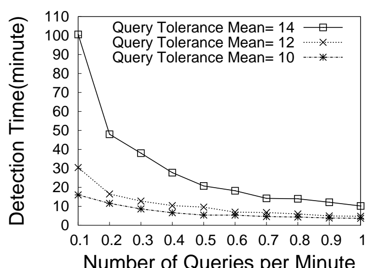


Figure 7: Evaluation of failure detection

Summary: Feedback from the proxy enables PRESTO to adapt to both data as well as query dynamics. We demonstrate that the query-adaptive version of PRESTO reduces latency by 50%, and the data-adaptive version reduces the number of messages by as much as 30% compared to their non-adaptive counterparts.

7.5 Failure Detection

Detecting sensor failure is critical in PRESTO since the absence of pushes is assumed to indicate an accurate model. Thus, failures are detected only when the proxy sends a pull request or a feedback message to the sensor, and obtains no response or acknowledgment.

Figure 7 shows the detection latency using implicit heartbeats and random node failures. The detection latency depends on the query rate, the model precision and the precision requirements of queries. The dependence on query rate is straightforward—an increased query rate increases the number of queries triggering a pull and reduces failure detection latency. The relationship between failure detection and the model accuracy is more subtle. Model accuracy depends on two factors—the time since the last push from the sensor, and model uncertainty that captures inaccuracies in the model. As the time period between pushes grows longer, the model can only provide progressively looser confidence bounds to queries. In addition, for highly dynamic data, model precision degrades more rapidly over time triggering a pull sooner. Hence, even queries with low precision needs may trigger a pull from the sensor. The failure detection time also reduces with increase in precision requirements of queries. For instance, for a query rate of 0.1 queries/minute, the detection latency increases from 15 minutes when queries require high precision to 100 minutes when the queries only require loose confidence bounds.

The worst-case time taken for failure detection is one day since this is the frequency with which a feedback message is transmitted from the proxy to each sensor. However, this worst-case detection time occurs only if a sensor is very rarely queried.

Summary: Our results show that sensor failure detection in PRESTO is adaptive to data dynamics and query precision needs. The PRESTO proxy can detect sensor failures within two hours in the typical case, and within a day in the worst case.

8 Related Work

In this section, we review prior work on distributed sensor data management and time-series prediction.

Sensor data management has received considerable attention in recent years. As we described in Section 1, approaches include in-network querying techniques such as Directed Diffusion [11] and Cougar [23], stream-based querying in TinyDB [13], acquisitional query processing in BBQ [3], and distributed indexing techniques such as DCS [20]. Our work differs from all these in that we intelligently split the complexity of data management between the sensor and proxy, thereby achieving longer lifetime together with low-latency query responses.

The problem of sensor data archival has also been considered in prior work. ELF [2] is a log-structured file system for local storage on flash memory that provides load leveling and Matchbox is a simple file system that is packaged with the TinyOS distribution [10]. Our prior work, TSAR [5] addressed the problem of constructing a two-tier hierarchical storage architecture. Any of these

techniques can be employed as the archival framework for the techniques that we propose in this paper.

A key component of our work is the use of ARIMA prediction models. Most relevant to our work on prediction models are the approaches proposed in BBQ [3], in which multi-variate Gaussian models were used for addressing spatial correlations, and dynamic Kalman filters for addressing temporal correlations. Our work differs in that we propose model-driven push instead of pull, and we split modeling complexity between proxy and sensor tiers rather than using only the proxy tier. ARIMA models for time-series analysis has also been studied extensively in other contexts such as Internet workloads, for instance in [9].

9 Conclusions and Future Work

This paper described PRESTO, a model-driven predictive data management architecture for hierarchical sensor networks. In contrast to existing techniques, our work makes intelligent use of proxy and sensor resources to balance the needs for low-latency, interactive querying from users with the energy optimization needs of the resource-constrained sensors. A novel aspect of our work is the extensive use of an asymmetric prediction technique, Seasonal ARIMA [1], that uses proxy resources for complex model parameter estimation, but requires only limited resources at the sensor for model checking. Our experiments showed that PRESTO yields an order of magnitude improvement in the energy required for data and query management, simultaneously building a more accurate model than other existing techniques. Also, PRESTO keeps the query latency within 3-5 seconds, even at high query rates, by intelligently exploiting the use of anticipatory pushes from sensors to build models, and explicit pulls from sensors. Finally, PRESTO adapts to changing query and data requirements by modeling query and data parameters, and providing periodic feedback to sensors. As part of future work, we plan to (i) extend our current models to other weather phenomena beyond temperature and to other domains such as traffic and activity monitoring, and (ii) design spatio-temporal models that exploit both spatial and temporal correlations between sensors to further reduce communication costs.

10 Acknowledgments

This research was supported, in part, by NSF grants EEC-0313747, CNS-0546177, CNS-052072, and EIA-0080119. We wish to thank Ning Xu at University of Southern California for providing the James Reserve Data. Thanks also to our shepherd, Matt Welsh, as well as the anonymous reviewers for their helpful comments on this paper.

References

[1] G. E. P. Box and G. M. Jenkins. *Time Series Analysis*. Prentice Hall, 1991.

[2] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proc. ACM SenSys*, 2004.

[3] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. VLDB*, 2004.

[4] P. Desnoyers, D. Ganesan, H. Li, and P. Shenoy. PRESTO: A predictive storage architecture for sensor networks. In *Proc. HotOS X*, 2005.

[5] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: A two tier storage architecture using interval skip graphs. In *Proc. ACM SenSys.*, 2005.

[6] Emstar: Software for wireless sensor networks. `http://cvs.cens.ucla.edu/emstar/`.

[7] A. Arora, et al. ExScal: Elements of an Extreme Scale Wireless Sensor Network. In *The 11th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005

[8] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proc. ACM SenSys*, 2004.

[9] J. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. In *Proc. the IEEE Intl. Conf. on Systems and Network Management*, 1999.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, 2000.

[11] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. Mobicom*, 2000.

[12] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proc. Mobicom*, 2000.

[13] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. In *ACM Transactions on Database Systems*, 2005.

[14] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[15] G. Mathur, P. Desnoyers, D. Ganesan and P. Shenoy. Ultra-low Power Data Storage for Sensor Networks. In *Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS)*, 2006.

[16] M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson1, D. Hahnel, D. Fox, and H. Kautz. Inferring adls from interactions with objects. *IEEE Pervasive Computing*, 3(4):50–56, 2003.

[17] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. ACM SenSys*, 2004.

[18] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS)*, 2005.

[19] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In *Proc. Hot Chips 16: A Symposium on High Performance Chips*, 2004.

[20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT - a geographic hash-table for data-centric storage. In *First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[21] Stargate platform. `http://platformx.sourceforge.net/`.

[22] Center for Embedded Networked Sensing (CENS) - James Reserve Data Management System. `http://dms.jamesreserve.edu/`.

[23] Y. Yao and J. E. Gehrke. The cougar approach to in-network query processing in sensor networks. In *Sigmod Record*, 31(3), 2002.

[24] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proc. IEEE Infocom*, 2002.

[25] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. ACM SenSys*, 2003.