Adaptive Middleware for Distributed Sensor Environments

Xingbo Yu, Koushik Niyogi, Sharad Mehrotra, Nalini Venkatasubramanian
University of California, Irvine
{xyu, kniyogi, sharad, nalini}@ics.uci.edu

Abstract: *Distributed sensor environments are emerging as a feasible solution to a wide range of data collection applications. However, energy constraints on sensor nodes, which are usually powered by batteries, hinder the successful deployment of such networks. We present an adaptive middleware framework for energy aware information collection in sensor networks that ensures overall application quality while reducing communication overhead.*

1. Introduction

Advances in processor, memory and radio technology have made distributed sensor network infrastructures a reality. Cheap sensor nodes are now capable of sensing and processing data, and communicating this information over large-scale distributed environments. Sensors are widely used to monitor real world events and enable variety of applications, such as mobile target tracking and activity monitoring. Traditionally, effective power management to extend battery life is a critical issue in mobile computing. This is further exacerbated in sensor environments where it is usually difficult to replace sensor batteries.

Power in sensors is used for sensing, computing and communication. Modern sensors attempt to shut down components or adjust processing frequency, whenever possible, to conserve power [1]. Although the energy consumption rate for each type of operation is sensor and application specific, it has been shown that the energy consumption of a sensor node for communication activities substantially exceeds that required for sensing and computation [2]. Here, we will focus on reducing frequency of communication, sometimes even at the cost of additional processing, in order to conserve energy.

In contrast to hardware-driven sensor design techniques to conserve power, our approach focuses on exploiting tradeoffs between resource consumption and application quality in sensor environments [3]. Many real-world applications can tolerate application inaccuracy. This provides us with the opportunity to collect approximate data at predetermined accuracy levels, instead of precise data, as long as the application quality constraint is preserved. This in turn reduces the communication overhead between sensors and servers resulting in conservation of energy. In this work, we propose an adaptive middleware framework that exploits resource/quality tradeoffs to reduce energy consumption in the context of information collection in distributed sensor environments. We also show that predictability of sensor readings, which comes from the predictability of the real world phenomena the sensors are monitoring, can be exploited to further reduce communication overhead and thus energy consumption.

2. System Architecture and Middleware Components

A wireless microsensor is typically battery powered and therefore energy constrained. Each sensor node consists of the embedded sensor, an analog-digital converter, a processor with some limited memory, and a radio circuitry [1]. Each component is controlled by the micro operating system that decides which device to turn off and on. For low cost and low energy consumption we assume passive sensors [4]. These sensors only operate on the received acoustic or seismic waveforms from the non-cooperative source. This is in contrast to an active radar or sonar system in which a sophisticated receiver is used to find some information of interest from the reflected target return.

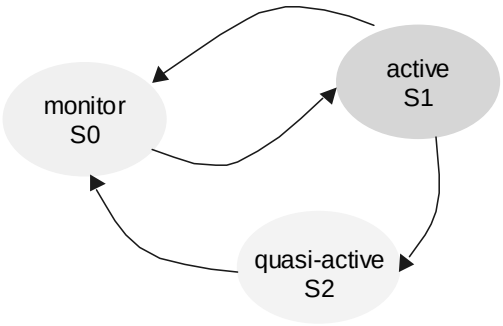

Figure 1. Sensor States

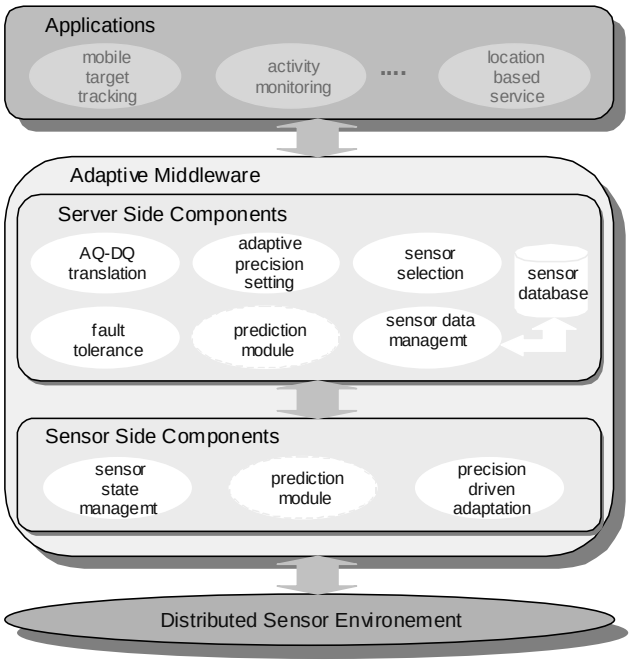

Figure 2. Middleware Architecture

A power aware sensor node has three different operation states (figure 1). In the active state *S1*, a sensor node receives and transmits at every time instant. In the quasi-active state *S2*, a sensor node receives and transmits only when a new measurement exceeds its previous value by an amount larger than certain error bound. This results in its operation at low frequency. The most energy-efficient state *S0* is a monitor state, in which a sensor only monitors (senses) the environment. We assume that sensors perform sensing and communicating at the same time instants with same frequency.

Figure 2 shows our proposed middleware framework, using which applications that trade quality requirements for energy consumption can be built. The framework supports two forms of adaptation (a) precision-based and (b) prediction-based. The middleware services bridge the application layer with the underlying sensor network infrastructure and has components at both the sensor and server side.

The server module includes the following components:

- AQ-DQ Translation: This component translates a given application quality requirement to corresponding quality requirements on data collected by sensors. Note that the translation may be application-specific and is often complicated. Each application built on the middleware framework has a corresponding AQ-DQ translation function built into this component.
- Adaptive Precision Setting: The data quality obtained from AQ-DQ translation will be expressed as a tolerable measurement error and communicated to sensors by this component.
- Sensor Selection: This component determines the state of sensors in the sensor environment and communicates this information to other sensors when necessary.
- Sensor Data Management: This component manages approximate sensor data within a sensor database. It also maintains other relevant information, such as states of sensors.
- Fault Tolerance: This component handles fault tolerance issues by implementing fault tolerance algorithms, such as those exploiting redundancy checking and heart-beat signals.

On the sensor side, the components include:

- Sensor State Management: It manages state transitions of sensors based on either sensing input, or instructions from server.
- Precision-Driven Adaptation: This component implements a filter on data generated by the sensor based on the error tolerance from adaptive precision setting process at server side.

In addition to the components described above, we envision a Prediction Module that resides on both the sensor and server sides that executes in parallel with the precision-based module. In precision-based adaptation, we compare the current sensor reading with its previous reading to decide if an update is necessary. If both the sensor and server side agree that the current sensor reading is the same as previous reading, they follow a "static model" and there is no need for a message. In general, sensor reading predictability can be exploited through more complicated prediction models [6] to further reduce communication cost. The prediction module implements prediction models that both sensor and server agree on to enable the adaptation process.

3. Designing Sensor and Server Side Protocols

At the sensor side, a sensor that is not engaged in any tasks will stay in its monitor state *S0*. When an external event happens, such as a temperature increase in temperature monitoring application, the sensor measurements may exceed a threshold. The sensor then transitions into active state *S1*. In the active state, a sensor will send updates to the server at every time instant. The update message consists of the new measurement, an associated timestamp, and possibly sensor state parameters such as its battery power level. A sensor may also go back to monitor state from the active state if the condition rendering the sensor active disappears (e.g. when sensor measurement goes down below certain threshold) or Alternately, the server may explicitly cause a sensor to transition to the monitor or quasi-active state S2. In the latter case, the server communicates a measurement error bound that is computed using application quality requirement to the corresponding sensor. In this energy-efficient mode, the sensor needs to send an update to the server only when the actual measurement value exceeds the previous value or predicted value by at least the given error bound.

t: time; *sid*: sensor id; r_t: reading at time t; r_{pre}: previous or predicted reading; δr: measurement error tolerance
1. if((external event) and (sensor_state=*S1* or sensor_state= *S2*)
2. send message to server to remove node from "active list";
3. sensor_state=*S0*;
4. else if(external event)
5. if(sensor_state=*S0*)
6. state=*S1*;
7. send_update_packet(*sid*, *t,* r_t, sensor_state);
8. if(sensor_state=*S2*)
9. if($\lvert r_t - r_{pre} \rvert \le \delta r$)
10. send_update_packet(*sid, t,* r_t, sensor_state);
11. $r_{pre} = r_t$; // if r_{pre} is previous reading
12. else do nothing;

Table 1: Sensor Side Protocol

At the server side, a server with a group of applications running on it contains a component that translates application qualities to sensor data qualities. When there exist multiple applications over same set of sensors, setting error tolerance for sensors becomes an optimization problem [5].

The server maintains a list of active sensors ("*active list*"), with a list of history values for each sensor over a time period. These history values are maintained for application-related activities

as well as for prediction-based adaptation. When the server receives an update message, it first checks if the corresponding sensor is in active sate *S1*. If so, it adds the sensor to its "*active list*" and sends the sensor a tolerable error bound. This allows the sensor to transit into quasi-active state. However, if the sensor is already in the quasi-active state *S2*, the server determines if the sensor is in its "*active list*", indicating that the sensor is currently in use by an application. If this is true, the corresponding sensor reading history list will be updated and the data is further processed to produce output for the corresponding applications. Otherwise, the server causes the sensor to switch to monitor state. If the server receives a message indicating that a sensor has transitioned into monitor state *S0*, the sensor is removed from the "*active list*" if an entry for the sensor exists.

t: time; *sid*: sensor id; r_t: reading at time *t*; *al*: "active list"; δ_{app}: application quality
1. if(sensor_state becomes *S1*) 2. al.add(*sid,* r_t*, t*); 3. compute δr from δ_{app} and *al*; 4. send_error_update(*sid,* δ*r*); 5. else 6. look up sid in al; 7. if(*al*.contains(*sid*)) 8. add r_t to the corresponding entry in al;

Table 2: Server Side Protocol

To support prediction based adaptations, a sensor and the server need to store a set of prediction models {$M_1, M_2, M_3, \ldots, M_n$}. A model is initiated based on sensor history readings either at the sensor(server) side and subsequently communicated to the server(sensor). The initial default model uses a precision-based "*static model*". This is followed by a model-fitting approach that matches history values of a sensor to one among a set of predetermined models. Whether or not model switching is actually done depends on a relative priority of the new model in comparison to the priority assigned to the original model. Such a priority can be assigned based on the number of updates required if a model is to be followed. A careful choice of model fitting parameters can reduce communication overhead significantly without sacrificing on application quality.

4. Issues and Challenges

In this article, we briefly touched upon the issue of exploiting prediction models to reduce communication cost. While it is difficult to discover an "*optimal prediction model*" for any application, different models yield different resource consumption characteristics. Building appropriate models, as well as dynamic model fitting and switching remain interesting research topics. We are also exploring additional operations that exploit location characteristics of sensors to achieve further energy savings. For example, sensors that are geometrically closed to each other may cooperate among themselves and aggregate readings before forward them to the server.

Several fundamental challenges arise in the implementation of the distributed sensor environments, such as fault tolerance and timeliness. Given the large number of sensors and the ad hoc nature of sensor environments, failures of sensor nodes and communication failures among sensors are very likely. Fault tolerance algorithms are critical to deal with the failures. Such algorithms will consist of components to detect failures in a timely fashion, as well as components to recover from failures. Several sensor applications, e.g. real time environmental monitoring, have time constraints on the delivery of sensor values to applications. Both precision-based adaptation and prediction model implementation may cause delays in data acquisition; these delays must be accounted for when determining if the required timeliness constraints can be met. The QUASAR [7] and AutoSeC projects at the University of California, Irvine are pursuing research that addresses tradeoffs in accuracy, timeliness, performance and flexibility in sensor networks. The development of an adaptive middleware sensor framework is a first step towards this goal.

References

[1] Amit Sinha, Anantha Chandrakasan. Dynamic Power Management in Wireless Sensor networks. *IEEE Design and Test of Computers, 2001*

[2] G.J. Pottie and W.J. Kaiser. Wireless Integrated network sensors. *Communications of the ACM, 2000*

[3] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. *22nd International Conference on Distributed Computing Systems, July 2002*

[4] J. C. Chen, K. Yao, and R. E. Hudson, Source Localization and Beamforming. *IEEE Signal Processing Magazine, 2002*

[5] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. *To appear:SIGMOD 2003*

[6] L.Gao and X.S Wang. Continually Evaluating Similarity-Based Pattern Queries On a Streaming Time Series *SIGMOD 2002*

[7] http://www-db.ics.uci.edu/pages/research/quasar/index.shtml