

SenQ: An Extensible Query System for Streaming Data in Heterogeneous Interactive Wireless Sensor Networks

Anthony D. Wood, Leo Selavo, John A. Stankovic
Department of Computer Science
University of Virginia
{wood, selavo, stankovic}@cs.virginia.edu

ABSTRACT

Interactive wireless sensor networks (IWSNs) manifest diverse sensors and dynamic user interactions. To support data collection and processing in such heterogeneous systems, we created SenQ: an extensible query system and network protocol for streaming sensor data. We evaluate it using a set of devices we developed for physiological, environmental, and activity monitoring in the context of assisted-living.

Rather than tree-based wide-area sensor aggregation common in environmental sensor networks, data processing patterns in IWSNs are user and event-driven. SenQ's query model and implementation are lightweight enough to allow in-network query issue by wearable interfaces and other resource-constrained devices. It standardizes access to sensor drivers and processing blocks for extensibility to new applications, and to allow complex virtual sensors to be created.

SenQ provides snapshot and streaming access to a dynamically reconfigurable in-network data processing chain, and supports aggregation in the temporal and spatial domains. On-demand buffering, query caching, efficient restart and other optimizations reduce network overhead while minimizing data latency.

We have fully implemented SenQ in TinyOS for MicaZ motes, embedded user interfaces, and in Java for the Stargate gateway and PDA devices in a testbed for smart assisted-living. We present SenQ's architecture, its protocol design and implementation, and evaluate its efficiency and performance.

1. INTRODUCTION

Wireless Sensor Networks (WSN) enable fine-grained collection of sensor data about the real world. Applications are diverse and numerous, including the military, environment, health-care, structural monitoring, and others. A large number of sensing modalities are now available, though integrating them on WSN platforms and efficiently extracting the data from the network remain a challenge.

Sensor devices are constrained by small form-factors and high cost sensitivity. In turn these constraints lead to oth-

ers: low-power, slow processors, and limited memory. These difficulties have attracted significant research effort into integrating sensors with WSN devices and reducing wasted radio transmissions through in-network processing and efficient protocols.

Many WSNs are intended for relatively homogeneous deployments to measure environmental conditions over broad geographic areas [12, 23]. In this context, using hierarchical spatial aggregation as data traverses a tree greatly reduces communication costs while providing data summaries to a sink or base station.

As sensor networks become ubiquitous, many will recede into the background, unobtrusively serving scientific and public purposes. Others will facilitate a rich interaction with the environment and cyber-physical systems. Current data management patterns based on a centralized database model will evolve to also support distributed, user-directed access to ambient sensors via other embedded interfaces.

One growth area for interactive wireless sensor networks (IWSNs), is the health-care domain. It already uses a wide variety of sensing modalities, and can benefit from their dynamic deployment. For example, based on an assisted-living resident's health, a doctor may give a box of sensors to place in the apartment or to be worn by the resident. An emplaced sensor network provides a rich context for resident's environmental conditions—but it must integrate with body area networks, embedded user interfaces, and back-end control and storage. The result is a hybrid and highly interactive network, that strains existing models of centralized query planning, dissemination, and collection [10].

Some systems rely on micro-servers that are tightly coupled with sensor devices [6], or require data processing to be performed exclusively by a tier of more resourceful devices [19]. Interactive sensor networks must support *distributed data access* not just by back-end servers, but by health-care workers, family members, and assisted-living residents. Smart embedded devices require *in-network query issue* to provide reminders, perform actuation, and display real-time *streaming* data from body networks.

SenQ is a query management system and network protocol for IWSNs that is flexible, extensible, and efficient. It provides snapshot and streaming data access, runtime sen-

⁰Technical Report CS-2008-01, Department of Computer Science, University of Virginia, 2008.

sensor discovery, and per-query dynamic rewiring of processing modules. New sensor and processing modules are easily integrated with SenQ, and virtual sensors use hierarchical composition to hide custom logic and processing behind SenQ’s query abstraction. It is compact enough to coexist with an embedded GUI library and streaming data display application on a MicaZ mote.

Contributions of this work include:

- A uniform programmatic abstraction for lightweight embedded applications using a standardized sensor and processor plug-in model for ease of extensibility,
- Support for heterogeneous sensor devices common in health-care applications, and for hierarchical composition of sensors and logic into new virtual sensors,
- A unified architecture for collecting environmental, physiological, and activity data by the system in the background and in response to users and events in-network,
- A runtime sensor and device discovery capability that allows long-term interoperability without “rebooting” the network,
- Efficient spatial and temporal aggregation, on-demand buffering, and query caching to minimize communication overhead and latency for streaming data, and
- A full implementation for TinyOS that is extensible, modular, and small enough to leave room for applications, and an implementation in Java for PC- or PDA-based query issue.

We describe the design and implementation of SenQ for a diverse assortment of sensor devices designed for AlarmNet, a IWSN testbed for assisted-living, including a virtual fall sensor and embedded user interface.

We begin in the next section with the network model of AlarmNet, and then delve into SenQ’s details in Section 3. SenQ’s spatial aggregation and virtual sensor capabilities follow. It is fully implemented and details are briefly described in Section 7, followed by a system evaluation in Section 8, and conclusion.

2. NETWORK MODEL

SenQ was designed to address the needs of AlarmNet, a wireless sensor network for assisted-living and residential monitoring. AlarmNet uses a tiered architecture to unify a diverse set of computing, processing, storage, and sensing elements. Figure 1 shows the components and topology of the architecture: emplaced and mobile sensor networks, gateways on a backbone network, back-end analysis, database storage, and user interfaces. Devices may be dynamically deployed into an existing AlarmNet installation, when, for example, the doctor sends a patient home with new body or emplaced sensors specific to a newly diagnosed condition.

Three types of information flows are supported simultaneously by the network. First, the gateway control software automatically issues long-lived *background queries* for sensor data to maintain the context of users and spaces and to moni-

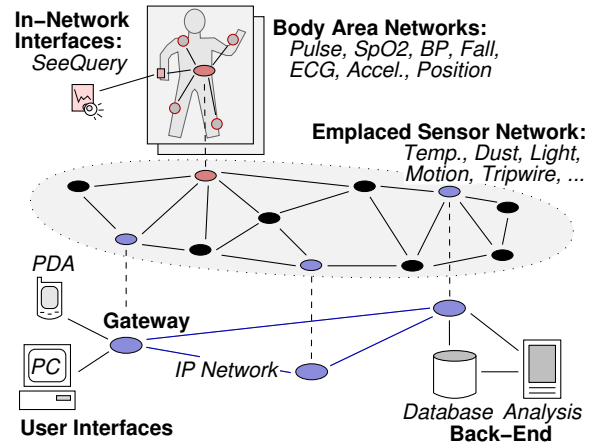


Figure 1: AlarmNet architecture components and logical topology.

tor system health. Sensor data flows back to the gateway for processing and permanent storage in an attached database, where activity inference, longitudinal data-mining, and circadian rhythm detection [22] algorithms execute on back-end other algorithms execute on back-end processors.

Second, *external users*, such as family members or health-care workers, issue queries by connecting to a gateway via its IP connection. The gateway control software manages users’ authentication and authorization for requested data, and binds requests to individual sensors in the network. Data flows to and through the gateway to the attached client.

A significant challenge comes from the third type of flow, in which *embedded user interfaces* or smart sensors query the local body-area network in response to events or user input. These potentially high-rate and unexpected streams within the emplaced sensor network preclude architectures in which all data access is mediated by centralized microservers or gateways.

The limited capabilities (processor, battery, and storage) of embedded sensor and interface devices, and the need for heterogeneity motivate the following goals for our query system:

- Flexibility—SenQ provides runtime sensor discovery, per-query dynamic rewiring of processing modules, snapshot and streaming data access, and local-area spatial aggregation, in a flexible protocol for wireless sensor networks.
- Extensibility—Application-specific sensing and processing components are easily added to SenQ by system designers using standardized interfaces, and virtual sensors use hierarchical composition to hide custom logic and processing behind SenQ’s query abstraction.
- Efficiency—SenQ provides in-network query issue to keep data close to its origin, and uses query, sample, and report caching to reduce communication costs and overhead, leaving room for application resource needs.

SenQ’s architecture is presented next, followed by a bottom-

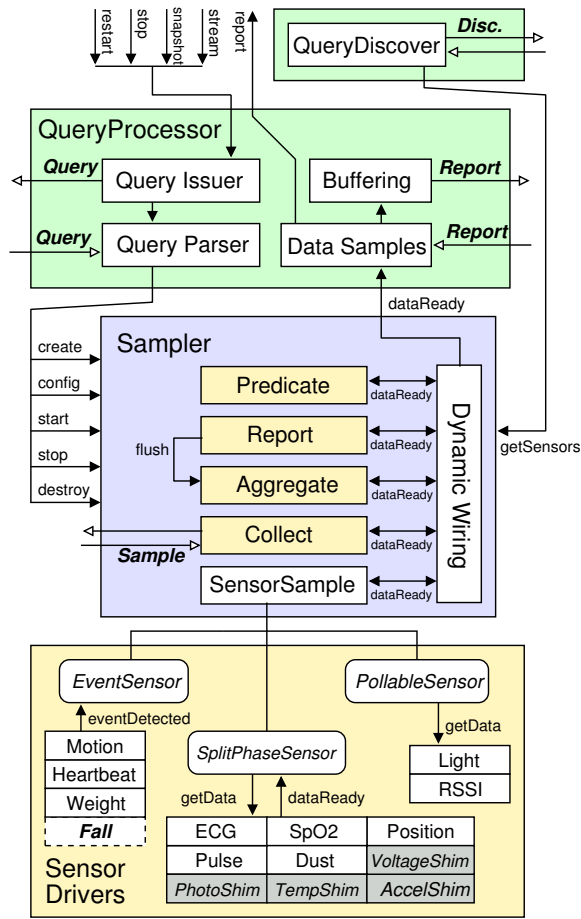


Figure 2: The SenQ query processing stack. QueryProcessor parses queries and Sampler reads data from the sensor drivers on schedule, which flows up the processing chain for reporting. Sensor drivers and processing blocks are extensible; some AlarmNet components are shown.

up description of the sensors, software, and network protocol.

3. SENQ QUERY SYSTEM

SenQ queries are uniquely identified in the network by $\langle source, ID \rangle$ tuples to allow multiple concurrent queries on originators and targets. The originator specifies the 8-bit ID, which is included with each report returned. Three types of queries are supported by the software API and network protocol (see format in Figure 3):

- *Snapshot* queries provide efficient point-in-time samples with no processing and minimal delay. The query consists of the sensor type (pulse, dust, light, etc) and size of data (8–32 bits). The sensor device samples the requested data and returns a single report to the originator, completing the transaction.
- *Streaming* queries are posed with additional parameters that control the sampling rate, processing algo-

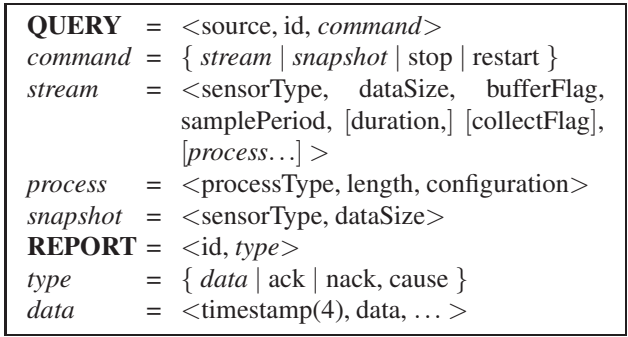


Figure 3: Query and Report message contents.

gorithms, how to buffer results, and whether to aggregate in the local area. As data is collected, reports are streamed back to the requester until a Stop command is received or an optional maximum duration is reached.

- *Spatially aggregated* streams provide in-network fusion for local-area sensor data. Though many queries in AlarmNet are user-centric and request data from a particular device, some benefit from limited-scale spatial aggregation, including the virtual fall sensor we describe later. In SenQ, spatial aggregation is indicated by a collect flag in the query. Upon receipt, the collector rebroadcasts the query to its neighbors, which send back samples from their local processing chain. These samples are combined with the collector’s local data to satisfy the query.

Both stream and snapshot queries may efficiently be *restarted* later with a short command from the originator that minimizes communication, parsing, and startup overhead. If the original query has been flushed or evicted from memory, a negative acknowledgment (NACK) serves to request it again from the originator.

A processing chain on each sensor device transforms sensor data in a query-specified way. Stream queries may include a list of *process* specifiers, consisting of one or more $\langle type, length, configuration \rangle$ tuples, as shown in Figure 3. They may appear in the chain only once, but in any order. SenQ passes the opaque configuration data to the processing block when starting the query. When no processes are specified, raw samples are reported.

Process blocks are specific to the application domain and easily plug-in to the SenQ software using nesC wiring, as described in Section 4. They transform scalar data as it passes up the chain toward the Query Processor for reporting.

3.1 Query Management

Control software residing on the gateways provides query management in AlarmNet for internal and external access to data. Nodes *register* with the nearest gateway upon power-up, providing their device type and hardware ID. They are admitted to the network and receive a network ID. Automatic background queries are issued depending on the de-

vice’s type and sensor capabilities. The data collected are used for internal monitoring and tracking, and the queries are specifically addressed to the registering device.

External user-originated queries require more processing. An authorized user receives a list of active devices that is updated in real-time when registrations are received. However, users of AlarmNet are more likely to request sensor data semantically for a resident or location. This presents a few challenges for query management on the gateway.

First, a request for information about user U must be mapped to a sensor (or group of sensors) S for execution. Some types of physiological data may have static associations, for example, Pulse data for resident Smith is always sensor 53 since this device was assigned to Smith and is worn constantly. Other sensor types require dynamic binding based on the person’s context (location, activity, etc). Environmental and activity data for locations depend on the availability of devices with these sensing capabilities emplaced or temporarily resident in the requested area. Environmental queries for residents are mapped onto sensors in the resident’s current location.

The second challenge is providing privacy for resident data. Privacy policies are expressed in terms of what types of data users acting in various roles may access for particular individuals and in what contexts. In the required tuple, \langle requesting role, sensor type, subject, context \rangle , the mapping from the requesting user to the requesting role is straightforward, and follows from an authentication step. Raw sensor types are mapped onto privacy sensor types, for example, motion data may be classified with respect to privacy policies more generally as activity or tracking data.

If data was requested for a particular resident, that resident is the subject. For specified locations or sensor devices, however, AlarmNet uses a hierarchical association to determine the subject. If a sensor has an assigned owner, that resident’s policy is used. Otherwise, if the sensor has an assigned location (such as a scale in the bathroom) or current location (for mobile sensors), the location’s assigned owner is bound as the subject. Living spaces, for example, may be controlled by particular resident. Common areas or other spaces without an owner use a default subject or policy called Public.

Third, the subject’s and/or system’s context is used to provide dynamism in the privacy policies. Access rights may change to allow greater visibility, for example, if a fall or other medical emergency has been detected. A full treatment of security and privacy in AlarmNet is outside the scope of this paper.

The properly translated and authorized query is sent to the mapped sensor device in the WSN, for further processing as described in the next section.

Embedded user interfaces and smart sensors operate most efficiently if their interaction is localized. Rather than proxy queries through the gateway control software, they issue them directly. First, they use the *Query Discover* module to locate nearby devices of interest. Discovery request messages con-

DISCOVER	= \langle <i>device, sensor, process, exclude</i> \rangle
<i>device</i>	= \langle deviceType, ... \rangle
<i>sensor</i>	= \langle sensorType, ... \rangle
<i>process</i>	= \langle processType, ... \rangle
<i>exclude</i>	= \langle network ID, ... \rangle

Figure 4: Discover request and reply contents.

tain lists of device types, sensor types, and process types that the node is seeking (see Figure 4). Neighboring devices which satisfy the request (and which are not listed in the discovery message’s exclude list) send a reply containing their capabilities. The exclude list prevents known neighbors from responding again on retries.

The SeeQuery interface we developed uses discovery to present a list of nearby devices and their capabilities to the user, who can select among pre-defined queries based on the sensor types present.

3.2 Query Processing

Local (via API) or remote queries (via network protocol) are parsed and managed by a Query Processor, shown at the top of Figure 2. It instructs the Sampler module to allocate a free data structure for the new query, configures each process block with the data provided, and starts collection of data. The SensorSampler module at the bottom of the figure uses standard interfaces to request data from sensor drivers, which passes through the configured processing chain on its way up to the Query Processor, where it is buffered (if requested) and reported.

A useful property of SenQ’s query processor is that it provides a certain degree of *location transparency* to query issuers. Embedded nesC applications use the same interface to issue queries for local or remote sensors, and the networking aspects are hidden behind the QueryProcessor interface, shown in Figure 5. Queries for remote sensors use the underlying routing protocol, while queries for local sensors are managed by the resident Query Processor and Sampler modules.

Embedded query issue opens many possibilities for smart in-network processing. For example, in AlarmNet the SeeQuery provides an in-situ wearable display on a MicaZ platform (described in Section 6). It may be used to query the battery supply of nearby sensor devices to determine when maintenance is required, or it may show a graph of pulse or blood pressure reported from body sensors. Or, a smart light-actuator may send a long-lived query to nearby door contacts or motion sensors so that it acts directly and locally to control the lights, without back-end intervention and delays.

The combination of symmetric interfaces and embedded query issue enables another important feature: sensor drivers themselves may issue local and/or remote queries hierarchically, using the same interface to SenQ as client applications.

These hierarchically composed sensors we call *virtual sensors*, and we describe one example, a sensor for detecting falls, in Section 6. System designers can use virtual sensors to create sophisticated, multi-tier sensing, processing, and logic, to minimize unnecessary network traffic beyond the location of an event or user.

The Query Processor has a number of other capabilities to maximize performance for resource-constrained embedded systems we target.

An embedded application with no local sensors, such as the SeeQuery, may be built with only the QueryProcessor module shown at the top of Figure 2. A stub Sampler and sensor drivers are wired into the application to avoid unnecessary code or data memory waste.

Further efficiency is gained by combining multiple samples received by the Query Processor in a single report message to save overhead and reduce radio traffic—at the expense of latency. Stream query originators may specify full buffering, in which the Query Processor aggregates as many individual reports as possible in a single TinyOS message. Otherwise, each sampled and processed value is sent in a separate report—subject to on-demand buffering during congestion.

On-demand buffering is used when a sample has been received in the Query Processor and is ready to be transmitted, but the outgoing message buffer is busy due to channel congestion. This incurs less average latency than full-buffering (though has more overhead) and avoids dropping high-rate samples.

Reports with data are timestamped (see Figure 3) to allow the receiver to properly sequence the data and detect drop-outs, in case the underlying routing provides out-of-order delivery or messages are lost in-transit. For full or on-demand buffering, the timestamp included is that of the last sample. Reports also bear status changes, such as positive and negative acknowledgements with cause codes. Local applications receive them through the `dataReceived` and `statusReceived` events in programmatic interfaces shown in Figure 5.

Each sensor type may generate individual sample readings of different sizes. The default data-size may be overridden by the user since domain knowledge may allow greater packing efficiency, and because some of the aggregation functions (like `sum` and `count`) may overflow the default data-size of their sensor types. Compacting reports reduces energy costs due to the transmission of redundant data. Other compression schemes, such as run-length encoding, can be added as processing plug-ins if an application warrants it.

Memory constraints of WSN devices limit the number of queries that may be simultaneously serviced or stored. Stopped queries are replaced using a least-recently used policy to maximize the ability of applications to restart them later.

Next, we describe the sensor data generation and processing chain that provides samples to the Query Processor.

4. SENSOR AND PROCESSING PLUG-INS

In contrast to many environmental monitoring networks [23, 12], AlarmNet must support a wide variety of sensor types—currently twenty. Some use internal components (e.g., supply voltage), connect to external ADC channels (e.g., dust, temperature), read from UART serial links (e.g., pulse, weight, BP), or are triggered by interrupt (e.g., motion, switch). Each may require different processing or conditioning locally before reporting.

To provide flexibility for the addition of new sensor types, SenQ provides common interfaces for the sensor drivers and processing blocks so they may be plugged-in (wired) at compile time.

Applications must provide three pieces of information to integrate SenQ. First, active message types for queries, reports, discovery and sample messages must be defined, as is typical for TinyOS applications.

Second, sensor and processing enumerations, by which the Sampler module references the application-specific components. The structure of the standardized enumerations is shown below. Pairs of `_MIN` and `_MAX` values are used to iterate over the Sampler’s parametric interfaces without explicit knowledge of the sensor or processor plug-ins used.

```
typedef enum {
    S_MIN,
    S_EVENT_MIN = S_MIN,
    S_MOTION = S_EVENT_MIN,
    S_HEARTBEAT, ...
    S_EVENT_MAX,
    S_SPLIT_MIN = S_EVENT_MAX,
    S_PULSE = S_SPLIT_MIN,
    S_DUST, ...
    S_SPLIT_MAX,
    S_POLL_MIN = S_SPLIT_MAX,
    S_MOTIONLIGHT = S_POLL_MIN,
    S_RSSI, ...
    S_POLL_MAX,
    S_MAX = S_POLL_MAX
} samplerSensor_t;

typedef enum {
    P_MIN,
    P_SAMPLE = P_MIN,
    P_COLLECT, ...
    P_OUTPUT,
    P_MAX = P_OUTPUT
} samplerProcess_t;

typedef enum {
    D_MIN,
    DEVICE_ANY = D_MIN,
    DEVICE_MTS310, ...
    D_MAX
} samplerDevice_t;
```

Third, the Sampler module’s interfaces must be wired to the application’s plug-ins. The AlarmNet wiring for the Aggregate processing block and Fall sensor is shown here:

```
Sampler.ProcessStdControl [P_AGGREGATE] → Aggregate;
Sampler.ProcessControl [P_AGGREGATE] → Aggregate;
Sampler.Input [P_AGGREGATE] → Aggregate.Output;
Sampler.Output [P_AGGREGATE] ← Aggregate.Input;

Sampler.SensorStdControl [S_FALL - S_MIN] → Fall;
Sampler.SensorInfo [S_FALL - S_MIN] → Fall;
Sampler.EventSensor [S_FALL - S_EVENT_MIN] → Fall;
```

As sensor hardware or processing algorithms become available, new devices incorporate their components and are de-

```

interface QueryProcessor {
  command queryCause_t stream (uint16_t dest, uint8_t qid, samplerSensor_t sensor, uint32_t samplePeriod, uint16_t duration,
                               uint8_t dataSize, bool cache, bool collect, samplerProcessConfig_t process [], uint8_t processLen);
  command queryCause_t snapshot (uint16_t dest, uint8_t qid, samplerSensor_t sensor, uint8_t dataSize);
  command queryCause_t stop (uint16_t dest, uint8_t qid);
  command queryCause_t restart (uint16_t dest, uint8_t queryId);
  event result_t statusReceived (uint8_t qid, queryStatus_t status, queryCause_t code);
  event result_t dataReceived (uint8_t qid, uint32_t timestamp, uint8_t *data, uint8_t len);
}

```

Figure 5: Query Processor API interface for in-network issue of queries using nesC for TinyOS.

ployed without reloading code on other devices. This maintains continuity of operation for environments, such as health-care, where it is not practical or safe to reset the system. SenQ’s internal processing and network protocol treat the sensor and processing types opaquely, so that old devices ignore new codepoints.

4.1 Sensor Devices

Sensor drivers must provide standard interfaces to be used with SenQ, according to the most appropriate category. An *EventSensor* generates data sporadically as it becomes available, such as from a hardware interrupt. *SplitPhaseSensor* represents sensors which must read and convert data upon request, such as from analog sensors connected to the ADC. Data is provided asynchronously to the caller. Data that may be quickly read synchronously uses the *PollableSensor* interface.

Drivers also provide a *SensorInfo* interface, which includes *isPresent*, *getCategory* and other functions to aid runtime discovery of nodes’ capabilities and types. Interfaces to sensors not present on a node are bound at compile time to non-functional stubs. Third-party and TinyOS sensor drivers are wrapped in a conforming lightweight shim.

Figure 6 shows many of the devices supported by SenQ and used in the AlarmNet system for monitoring the health of residents. We can broadly classify them into physiological, environmental, and activity sensors, and briefly describe sensors of each type.



Figure 6: Sensor hardware supported by SenQ in AlarmNet.

Pulse oximetry is a noninvasive technology to determine a person’s oxygen saturation in blood (SpO₂) and heart rate (HR) [21]. We use one hardware design created by Harvard for CodeBlue [13], and a second that was designed at the University of Virginia for the AID-N project developed at APL at Johns Hopkins University [4, 14]. Both use a sensor

clip worn on the finger.

Typically, an **electrocardiography** (ECG) sensor is a stationary clinical device preventing the mobility of the person under observation. Our ECG sensor monitors basic cardiac rhythms with only two-leads, and was adapted for AlarmNet from the CodeBlue system at Harvard [13]. This sensor generates high-rate streaming data that is managed by SenQ.

Residents’ **body weight** provides AlarmNet with important long-term trend information. We adapted the LifeSource ProFit Body Scale UC-321 consumer device manufactured by A&D Medical, which reports measurements to SenQ over a digital serial link.

Blood pressure meter UA-767PC manufactured by LifeSource/A&D Co has been clinically validated [15], and indicates three vital signs for a person: systolic and diastolic blood pressure, and heart rate. These readings are expected to be in a certain range that is determined by the age, weight, gender, medical history, and current activity of the user. Divergence from the expected range can reveal a potentially harmful medical condition.

AlarmNet uses long-running SenQ stream queries to monitor residents’ living environments. If conditions become unfavorable the system alerts care-givers.

To monitor and prevent allergenic conditions we developed a **dust sensor** to report the air quality on demand. SHARP detector module GP2Y1010AU [18] has an infrared LED, photodetector, and amplifier enclosed in a package that allows air through but blocks any light in order to prevent false alarms. The LED illuminates dust particles, which reflect light to the photosensor for detection and hand-off to SenQ.

Temperature and light measurements are provided by the MTS300 and MTS310 sensor boards from Crossbow [2]. The drivers provided in TinyOS are wrapped by small configurations, or “shims”, that conform to the Sampler interfaces already described.

Sensors placed in and between rooms and on appliances detect resident motion and activity. Changes in activity patterns may indicate a decline in health status.

AlarmNet’s **motion sensor** is based on a MS13A device that was originally designed for X-10 home automation applications [26]. It was adapted by removing the controller and X10 radio circuit, and interfacing with the MicaZ mote. Movement is reported to SenQ as an interrupt-driven event. Also, a light sensor generates a pollable binary signal indi-

cating light and dark.

The **tripwire sensor** detects persons that move through doorways. It uses two laser beams and photo sensors to create a barrier that is broken when a person moves through the doorway. The driver reports a direction code to the Sampler as an event.

A magnetic **switch** attaches to doors and cabinets. Open/close events can be used as a hardware interrupt to wake the device from sleep and to classify activities.

In addition to some of the physiological sensors, **accelerometers** are worn by residents to provide activity recognition and fall detection. Though relatively large for wearable sensors, the MTS310 sensor boards by Crossbow are supported by SenQ and serve as a prototype body area network. Later, we describe their use for fall detection.

4.2 Processing Chain

Some sensors require little in-network processing, but for many of the types that provide high-volume data, it is essential to reduce both the energy cost and network congestion from sensor streams.

Above the hardware and sensor drivers is a group of modules comprising a scheduler and data processing chain, collectively called the Sampler, shown in Figure 2. They act in concert to manage sensor sampling for multiple concurrent queries, and to reduce generated data to only that which is useful for the application.

For example, pulse-rate samples may be collected every $200ms$ but reported only every $1s$, each an average of five samples. These may further be filtered by discarding them unless the average pulse-rate is above $130 bpm$, which may be a dangerous zone based on a resident's age.

Processing modules provide a *ProcessControl* interface with `create`, `config`, `start`, `stop`, and `destroy` methods. These allocate sample structures, pass user-specified configuration to the process, and supervise sampling activity.

DataProducer interfaces on each block provide incoming and outgoing paths for sensor data. It contains a single `dataReady` event. Instead of wiring each block in a static sequence (as in VANGO [6]), they are wired to Input and Output interfaces of WiringM, a dynamic data flow coordinator.

The chain always begins with `P_SAMPLE` (see enumerations in Section 4), where sensor drivers provide data, and ends with `P_OUTPUT`, which provides data to the Query Processor. In between, each query has its own ordering of the processing chain, specified in the API or network message.

Our processing modules operate on scalar values, rather than blocks. Nothing prevents a module from accumulating a block of data on which to operate (as does our Moving Average process), but the interfaces between modules are limited to passing a single scalar value. Extending this to pass blocks is straightforward, and we leave it for future work.

Next, we describe the operation of the pre-defined processing modules in AlarmNet, to give some idea of extensibility of SenQ.

SensorSample

The *SensorSample* module is required and built-in, and maintains a schedule for sensor sampling to satisfy multiple ongoing queries. A single timer is used to track the next scheduled sampling operation. Upon expiry, sensor data is requested from the driver according to the sensor's category, *Event*, *SplitPhase*, or *Pollable*. In general, when data is available it is propagated up the processing chain to the next consumer, as determined by the dynamic wiring.

PollableSensors return a value immediately. For *SplitPhaseSensors*, data is requested in the timer handler. The data becomes available asynchronously, usually after an ADC conversion. Normally, when a sample is initiated for an *EventSensor*, the *SensorSampler* only enables the event in the sensor driver. Events are signaled asynchronously, not on a schedule, and then the data is propagated up the chain. If polled, the *EventSensor*'s `getDefault` function is consulted for the appropriate value to return.

All queries that have concurrently requested the same type of sensor data will be notified upon its availability. This sample caching is necessary to promptly service queries despite the limited bandwidth of the analog to digital converter.

Aggregate and Report

Many aggregation functions or descriptive statistics can be calculated while keeping little state. The *Aggregate* module is optimized for such lightweight functions: count, sum, difference, maximum, minimum, average, and delta (absolute value of change from previous). In addition, a "latch" aggregator is provided for polled *EventSensors*. An event that occurs within a report period will latch its value and the data will be stored until the next report, unaffected by subsequent "readings" of the *EventSensor* if the default value is zero.

The *Aggregate* module works in conjunction with the *Report* module, which takes a report period as its configuration. The *Report* module drops intermediate samples until the specified period has passed. Then the data is passed to the next block in the chain, and the module flushes or resets the intermediate results stored in the associated *Aggregate* module.

Specifying a report period longer than the sample period provides temporal aggregation. For example, light may be sampled every second but only reported every $4s$, with intermediate results averaged by the *Aggregate* module.

The *Report* also provides rate limitation for events, which are not usually sampled at all, but provide data sporadically. In AlarmNet, a background query is issued to every registering motion sensor with no sampling, but a report period of $100ms$. Motion events will be reported no more frequently than that, even though the hardware often generates a rapid (on the scale of a few μs) train of interrupts when it detects motion.

Software Configuration	Code	Data
Base (4 query, 70B msg)	19010	1751
Process: Aggregate	+ 1016	+ 36
Process: Aggregate + Report	1620	98
Process: Moving Average	968	240
Process: Predicate	602	44

Table 1: Memory consumption for processing modules in bytes on MicaZ. Moving Average uses a maximum window of ten 32-bit samples.

Moving Average

A *Moving Average* module provides a windowed average, windowed moving average, or exponentially-weighted moving average (EWMA). The configuration passed in the query specifies which function is desired and the window size of α parameter.

To minimize error on hardware with no floating point support, the EWMA function uses a fixed-point representation for intermediate results. Parameter α is specified as an integer from 0 to ALPHA_MAX. Another compile-time constant defines the maximum window size for the other two functions.

Figure 1 shows the tradeoff between the Aggregate and Moving Average modules in data memory. In the Moving Average module, each concurrent query has a max-window-size buffer of 32-bit elements. With four queries and a ten-element window, the increases in data memory usage over the Aggregate + Report configuration is 142 bytes, but with 652 fewer bytes of code memory.

Predicate

Queries may specify a relational *Predicate* filter to cull uninteresting or redundant data. If the incoming sample satisfies the predicate, it continues up the chain, otherwise it is dropped with no output.

Relational predicates supported are: greater than, less than, within a range, equal to, and changed from previous. The configuration data contained in the query specifies the constants for comparison.

5. SPATIAL AGGREGATION

AlarmNet does not typically need to sample and aggregate sensor data from the entire network into the back-end, due to its heterogeneity. Although aggregate environmental data (temperature, light) may be useful in a health-care context, physiological and activity data certainly should not be mixed among residents, nor does every sensor type need to be sampled, but only those which are useful for the resident’s own health needs.

Spatial aggregation is needed more often for collecting data from other nodes in the local area, such as from a body area network. We give a concrete example in the Fall sensor in the next section.

A flag in the stream query specifies that the recipient is

to act as the coordinator of a spatial query. This node clears the flag and retransmits the query to its immediate neighbors, including an additional information element with the coordinator’s own network address and sample ID.

Neighbors which possess the requested sensor type then process the delegated query and act as sample slaves. Sensor data is sampled and flows up the processing chain, and passes through the *Collect* module according to the query originator’s specified process ordering.

In the slaves, samples are redirected over the network to the query coordinator’s *Collect* module in a *Sample* message (indicated in Figure 2). Incoming samples from local sensors and from query slaves are inserted into the coordinator’s processing chain.

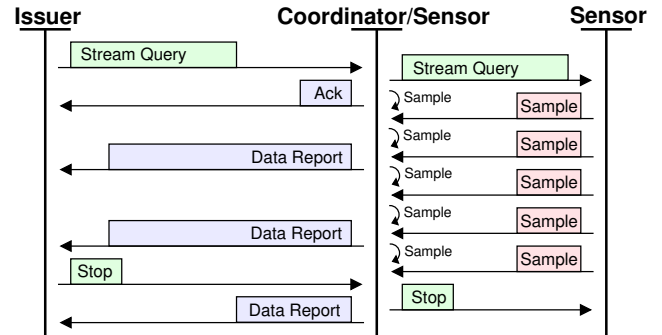


Figure 7: Example spatially aggregated query interaction. A sink issues a stream query which is re-broadcast by the coordinator. Samples are collected and reported to the sink until a Stop command is received.

Figure 7 shows an example interaction between a query originator and sensor devices. A Stream query is issued specifying temporal and spatial aggregation and report buffering. The coordinator rebroadcasts to its neighbors, each of which sends the samples back to the coordinator. As samples are taken, they are aggregated, packed into the buffer, and eventually reported when a message buffer is full. A Stop command halts the query, and partial buffer contents are returned to the originator. A Restart command may be used later to quickly and efficiently continue sampling.

Sample messages are very concise, containing only the 32-bit sample and the coordinator’s sample ID, to avoid causing network congestion. Placing a predicate filter prior to the *Collect* module will further reduce the number of Sample messages sent by nodes participating in a spatially aggregated query in the local neighborhood.

6. VIRTUAL SENSORS: FALL DETECTION

Falls account for a majority of accidental deaths in the U.S. of those age 65 or older, the population for which AlarmNet was designed. Using SenQ’s standardized sensor interfaces and existing supported sensors, we created a virtual fall sensor to detect and assess the severity of falls. The fall

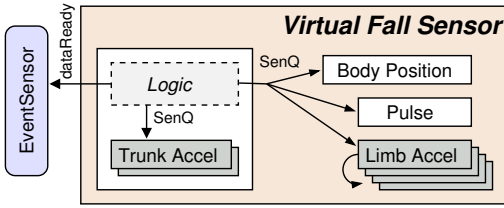


Figure 8: Virtual Fall EventSensor composed of other SenQ sensors and detection and assessment logic.

sensor provides an EventSensor interface to SenQ, as shown in Figure 8.

In the envisioned scenario, a health-care worker wears an embedded interface device, such as the SeeQuery described in Section 7.2, in an assisted-living facility to receive alerts and interact with the network. When arriving at work for the day, she uses the SeeQuery to issue a stream query to fall sensors worn by at-risk residents. The body area network worn by these residents consists of a pulse sensor, a body position sensor, and accelerometers on the trunk and limbs.

The fall detection sensor is *virtual sensor*, in that it does not directly interface to any sensor hardware, even though it connects to SenQ as a sensor driver. Rather, it uses SenQ itself to sample and filter data and processes the results to generate a fall condition as a new type of sensor. Thus, the health-care worker’s SeeQuery only receives fall condition reports of: $\{none, possible, confirmed, no\ movement, changed\ vitals\}$.

After receiving the query, the fall sensor first uses SenQ to issue a moderately high-rate stream query for the trunk accelerometer’s X and Y axes, using the Aggregate module’s delta function, and the Predicate module’s greater-than relation. These eliminate reports until the sensor experiences a sufficient shock to indicate a *possible* fall.

When a possible fall is detected, the fall sensor issues a short duration stream query to the resident’s body position sensor. If a majority of reports indicate a horizontal body position, the fall is *confirmed*. A fall event is generated up to SenQ, which reports to the fall query originator, i.e., the worker’s SeeQuery.

The fall sensor further *assesses* the potential severity of the fall by issuing a stream query to the pulse sensor, which reports if the five-second moving average changes. A significantly altered pulse rate may indicate either pain or unconsciousness. At the same time, a spatially aggregating query is issued to all the limb accelerometers. This query also specifies Aggregate delta and Predicate greater-than filters, to detect whether the fallen resident is moving or is still (perhaps unconscious or unable to move).

The current estimate of severity is periodically reported to the SeeQuery application until a majority of body position readings indicate that the resident has *recovered*.

The virtual fall sensor is both a component of SenQ, and a client of it, using hierarchical in-network query issuing to provide data fusion for the body area network sensors.

7. IMPLEMENTATION

SenQ has been fully implemented in the AlarmNet testbed. The Java gateway software runs on the Crossbow stargate [2]. It has a 400 MHz Intel PXA255 Xscale processor, 64 MB RAM, and 32 MB of built-in flash. There a QueryManager module interfaces with IP clients that originate queries, and generates background queries based on registering devices’ types.

Sensor devices used are primarily Crossbow MicaZ motes. These have 8MHz Atmel ATmega128L 8-bit processors, 4KB SRAM, and 128KB program memory and run on two AA batteries. The ECG sensor uses the TelosB mote, which is similar in capability but with double the SRAM. The SenQ modules for these motes is written in nesC for TinyOS.

In the fall sensor implementation, we used two-axis accelerometers for the trunk shock and body position sensors, and a single-axis to detect limb movement. The body position sensor compares X and Y levels to determine its orientation.

7.1 Network Query Protocol

Messages contain a fixed header identifying the query or report, plus a list of variably-sized Information Elements (IEs). Each IE consists of a single-byte code (if the high-bit is set), or a code, length, and payload. This structure allows network nodes to parse messages even after additional protocol codepoints have been defined, and it simplifies parsing when many fields are optional.

An optional maximum duration for the query may be specified as a separate IE included with a stream query. When the given number of reports have been generated, the query expires automatically as if a Stop command had been received.

Snapshot, Stop, and Restart commands are designed to be as small as possible for maximum efficiency. The latter two commands use the single-byte IE format. The Restart command removes the need to send the longer stream command for repetitive queries. If the target has flushed the old query, it will respond with a NACK message. The originator can then re-send the full Stream query.

7.2 Query Interfaces

SenQ allows queries to be issued from clients external to the WSN and by the system itself, including other sensor devices. The Java-language QueryManager module in the gateway issues background queries to devices based on their type as they power-on and register with the network. Long-lived queries for environmental conditions, activities, and health data are issued using SenQ and the reports are fed to internal modules for updating context and saving in the database.

We also developed a Java client for accessing AlarmNet from a PDA, such as a health-care worker would use. It allows queries to be issued for users, locations, or devices and graphs the results returned in real-time. One view of the GUI is shown in Figure 9. A library of template-based queries are

provided for ease of use by doctors and other busy professionals. Users must be authenticated and authorized to view the data requested.

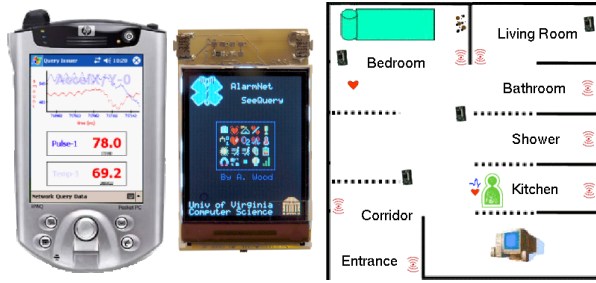


Figure 9: User interfaces on the PDA, SeeQuery, and PC.

SeeQuery is a TinyOS program for the SeeMote [16], a hardware module compatible with MicaZ and Mica2 Motes, with a color LCD screen (128x160 pixels, 64K colors, 1.8 inch diagonal, backlight), two navigation buttons, and an MMC/SD removable storage card. It provides in-situ display for reminders and data presentation, and is used as a diagnostic aid for system deployment.

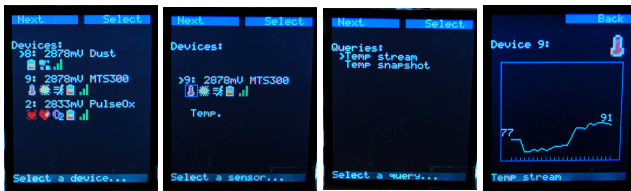


Figure 10: SeeQuery interface: discovered devices, sensor selection, query list, and real-time data graphing.

On power-up, SeeQuery uses SenQ’s discovery functionality to build a table of nearby devices and their supported sensors. Each is issued a snapshot voltage query, which indicates to the user whether fresh batteries are needed. Using the momentary buttons on the SeeMote, the user selects a device and one of its sensors. SeeQuery presents a list of available queries. When one is selected, it issues the query to the chosen device, and graphs the reported data in real-time. Screenshots of the SeeQuery showing these steps are in Figure 10.

Other user interfaces may be developed by building on the Java and nesC libraries available for query generation.

8. EVALUATION

A critical design goal for SenQ was to be compact enough to support embedment with in-network user interfaces, which are already resource challenged. First, we present SenQ’s consumption of memory and CPU resources to evaluate how well this goal was met. Then we determine performance limitations of the sampling and processing chain to demonstrate runtime efficiency. Finally, we evaluate the network performance of the query protocol in a multi-hop testbed.

Software Configuration	Code	Data
Minimum (no sensors, 1 query, 29B payload)	16756	711
Base (voltage, 4 queries, 70B payload)	18830	1576
Sensor: Blood Pressure	+ 698	+ 29
Sensor: Pulse, SpO2, Heartbeat	1140	32
Sensor: ECG (Tmote Sky)	138	4
Sensor: Scale	1366	27
Sensor: Dust	414	25
Sensor: Motion, Light	594	0
Sensor: MTS300 (temp, photo)	1524	33
Sensor: MTS310 (+accel, mag)	2032	45
Sensor: Switch	502	4
Sensor: Tripwire	2834	76
Sensor: Fall	4654	105
Sensor: Body position	592	12

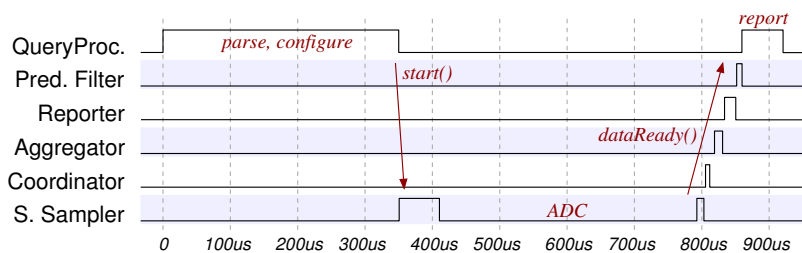
Table 2: Memory consumption for sensor drivers in bytes. Includes required TinyOS components (radio, timer, ADC, etc). Values for sensors are relative to the Base configuration.

8.1 Resource Consumption and Efficiency

The program and data memory required for SenQ depends on three application-specific parameters: the size of TinyOS messages, the sensor drivers linked in, and the number of maximum concurrent queries supported. Table 2 shows the size in bytes of TinyOS applications with different parameters. In the minimum configuration, all non-query related modules (such as localization, configuration, etc) are removed from the mote application that runs in AlarmNet. No sensor drivers are included, and the default TinyOS message length is used. The code takes about 16KB, with 711B data memory. A configuration more typical for use in AlarmNet is also shown: “Base” provides access to the internal mote voltage, supports four concurrent queries (per-node), and uses 70 byte payloads. Each included sensor driver requires code and data memory in addition to that of the Base.

The SeeMote GUI drivers, SeeQuery application, and SenQ query-only stack all reside comfortably in the MicaZ’s 4K data memory. Without icons, the code is 29KB and the application uses 2037 bytes of RAM. Adding all the color icons shown in Figures 9–10 takes an additional 1KB code memory, and fills up the data memory (1864 bytes additional), but these are not necessary to present streaming data. SenQ’s compact implementation leaves up to 2KB of RAM for application-specific functionality, such as custom sensor drivers, security, or routing protocols.

IWSN devices are often power-constrained, and the applications and protocols that execute on them must be efficient. We instrumented the entry and exit points of the QueryProcessor and Sampler modules to measure SenQ’s



(a) One empirical timing diagram showing receipt of a stream query, message parsing, Sampler configuration, and propagation of first datum up the processing chain until a report is ready to send.

Query Operation	mean	stddev
Stream start (initial)	654.18	176.352
(subseq.)	217.62	0.003
Stream stop	36.93	2.639
Stream restart	267.10	10.904
Snapshot	601.05	1.745
Snapshot restart	301.28	1.684

(b) Time in μs from start of operation until first report is ready (if applicable). Does not include $366.42 \mu s$ ADC time. 100 trials.

Figure 11: SenQ timing measurements using an Intronix LogicPort logic analyzer connected to a MicaZ.

load on the sensor device. Unused pins on the microprocessor were connected to an Intronix LogicPort logic analyzer sampling at 100MHz. This gave very accurate timing of the signals from the code modules with very little measurement overhead (three instructions to drive the pin high or low).

A timing trace from one experiment is shown in Figure 11(a) to illustrate the relative magnitudes of processing times. Here, a stream query was issued to the instrumented node for sampling and reporting its internal battery voltage every $73 ms$, with no data processing or buffering in the QueryProcessor. The figure’s timeline starts when the QueryProcessor received the query from the network, and shows the time to parse it, allocate data structures, and configure and start the Sampler ($358 \mu s$).

SensorSampler started the sampling timer and requested data from the VoltageShim driver ($50 \mu s$). ADC conversion takes 25 ADC clock cycles, or $366.42 \mu s$ (found empirically) from the request until the data is available. Then the SensorSampler propagates it up the processing chain. Since no coordination, aggregation, or filtering was specified, it reaches the QueryProcessor $68 \mu s$ later.

The QueryProcessor generates a report immediately since buffering is not enabled, and is ready to send it to the radio layer $54 \mu s$ later. The total time spent on the first sample is $918 \mu s$, of which $366 \mu s$ is waiting for an ADC interrupt. Subsequent samples begin at the SensorSampler module when the sample timer fires.

The mean and standard deviation of worst-case execution times from 100 trials are shown in Figure 11(b) for each query operation. Queries sampled battery voltage, used *mean* aggregation, a *range* predicate filter, no buffering, and four-bytes of reported data. These parameters together give the largest possible execution overhead of a non-coordinated (spatially distributed) query.

Even on the 8MHz MicaZ, these worst-case execution times leave most CPU resources for application demands. At a sampling rate of $100 Hz$, the steady-state overhead of SenQ is only 2.18%.

SenQ gains even more efficiency by caching query definitions so they may be restarted later. Processing time for a restart is half that of the initial query. Also, propagation

and processing of individual samples in a stream query takes only $218 \mu s$ —including time to prepare it for transmission. If aggregation, buffering, or filtering is used, the time is much less, as shown graphically in Figure 11(a).

8.2 Sampling Performance

The *maximum effective sampling* rate of SenQ depends on the execution overhead (described above) and the query’s sampling rate. As the sampling rate increases beyond a point, we expect to see worse performance in sampling jitter and dropped messages or samples. This is especially true due to the limited processing capability of the MicaZ and the non-real-time design of TinyOS.

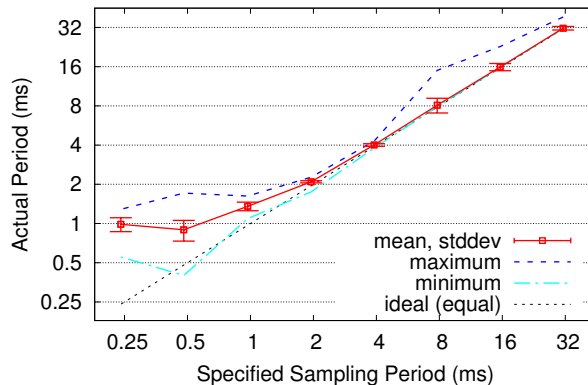


Figure 12: Specified versus actual (measured) sample period for over 300 trials. The ideal linear plot, when specified equals actual, is also shown. Timer performance degrades for $S \leq 1 ms$.

To find SenQ’s limits on sampling, we use a stream query with relatively costly parameters: mean aggregate, range predicate filter, four-byte data size, and no buffering. We varied the sampling and report periods identically, to result in a message being transmitted for *every* sample. The timing was captured precisely by the Intronix logic analyzer connected to a MicaZ mote.¹

¹The default TinyOS timer component allows a minimum period

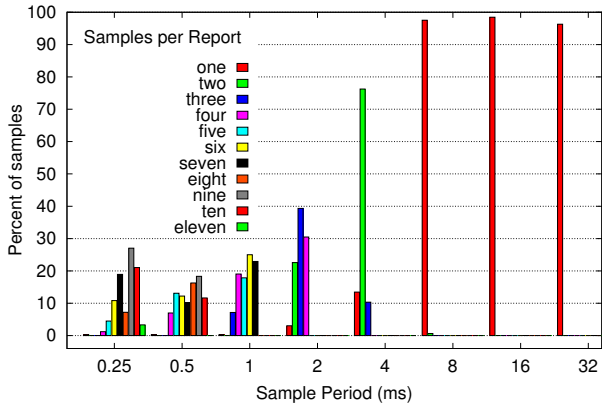


Figure 13: Distribution of report sizes resulting from on-demand buffering during congestion. The percentage of variable-sized reports bearing 1–11 samples is shown.

Figure 12 shows sampling jitter (difference from the requested rate) as the sampling rate varies from 4 KHz to 32 Hz . We focus on small sampling periods here, where difficulty is more likely to occur. The actual sampling rate tracks the specified rate closely down to 2 ms , with low variance and only occasional aberrations as shown by the plotted maximum. Below 1 ms , the microprocessor fails to service the timer as fast as requested, due to frequent radio and ADC interrupts and high CPU utilization from SenQ and other tasks. Without DMA to lighten the load on the microprocessor, it saturates.

Congested conditions benefit from SenQ’s on-demand data buffering. The effects are shown in Figure 13. When message transmission became a bottleneck at around a 4 ms sampling period, reports included more samples. At 0.25 ms , half of the reports included nine or more samples. A maximum sample loss of 11% was recorded at the sender for these trials.

8.3 Multi-hop Network Performance

SenQ was designed to layer atop any reasonable routing or network layers that provide point-to-point connectivity in the IWSN. AlarmNet uses a hybrid of geographic and tree-based routing due to its multi-tiered structure. Here we focus on SenQ’s composition with a related routing protocol, Implicit Geographic Forwarding (IGF) [1], to show SenQ’s performance in a multi-hop network. IGF uses late-binding of a packet to the next-hop forwarding node, and has small overhead with no requirement for maintaining neighborhood tables. Nodes compete for forwarding by setting timers proportional to their desirability. When a node responds, the other candidates abort.

We placed twelve MicaZ nodes in a six by two grid, and artificially limited their connectivity to create a five-hop network. A SeeMote device running the embedded SeeQuery

of 3 ms by default, so we modified the clock scaler to provide finer resolution for this test.

application was placed in one corner of the grid, to issue a stream query to the mote in the opposite corner for more than 1300 samples at varying rates.

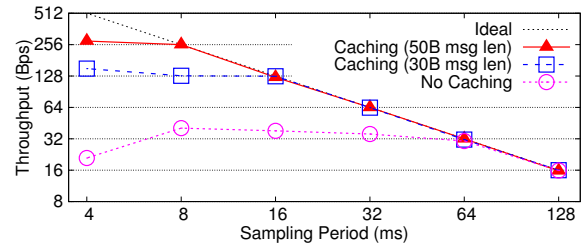


Figure 14: Application-data throughput across a five-hop network, using SenQ over IGF.

Figure 14 shows the network-wide application throughput (removing SenQ and IGF overhead). The ideal throughput is bounded in this experiment by the offered load, which is determined by the sampling period and size of each sample (two bytes). It is shown as a diagonal in the figure.

The degradation of throughput for small sampling periods is due to congestion and dropped packets. Although SenQ can sustain higher sampling rates (shown in Figure 12), IGF’s throughput is exceeded. The maximum forwarding-candidate selection time (64 ms in this experiment) governs how quickly nodes are forwarded. Interference further reduced throughput, since most nodes were within radio range despite the artificial forwarding constraints imposed. SenQ’s data buffering improves the throughput, matching the ideal down to 64 Hz or 128 Hz , for 30B or 50B message lengths, respectively.

8.4 Discussion

SenQ provides its support for streaming queries over a wide variety of sensor types at a low cost in code and data memory. It supports sampling rates with low loss up to nearly 1 KHz without requiring ADC DMA (as in [6]) due to its lightweight scalar processing chain model, efficient implementation, and on-demand buffering.

Reusability is enhanced by preserving traditional layering over network and routing layers, and minimizing modifications to the TinyOS kernel. This sometimes comes at the expense of performance, as shown in the sampling and multi-hop performance. TinyOS’s timer subsystem saturates before SenQ’s sampling mechanism, and IGF’s throughput degrades more quickly than SenQ at high data rates.

We have also deployed SenQ in an environmental monitoring network on Hog Island, in the Luster system [17]. Luster’s architecture is quite different from AlarmNet, being cluster-based with a TDMA MAC and strictly periodic sampling. Nevertheless, SenQ was chosen to extract the data for several reasons. First, it is quite easy to extend SenQ to new sensor types, such as the seven-lead light sensor “Medusa” in Luster. Second, Luster uses a deployment-time validation tool to allow in-the-field interaction with sensors. A

SeeQuery-like application uses SenQ's in-network query issue capability to validate the actual path—through SenQ—that sensor data will follow in the running system. Last, the extensible processing chain allows environmental scientists to easily change how the system provides data in response to their increasing experience with it.

9. RELATED WORK

Data is of preeminent value in sensor networks, and data querying and management has attracted much attention in the research community. An overview of query processing issues and approaches unique to WSN is available in [5, 24].

In Directed Diffusion [8], sinks publish interest tuples that induce sensor-value gradients in the network, along which data is routed from producers back to the sink. This sparked other designs for data-centric routing in WSN. In TAG [11], Cougar [27, 28], and TinyDB [10, 7, 9], data is filtered and aggregated as it transits the network. These approaches are oriented primarily to centralized collection of sensor data. In AlarmNet, transparent processing of health-related data as it transits the network may be prohibited due to security and privacy requirements that mandate encryption of payloads. Sensor and device heterogeneity also limit hierarchical tree-based structures.

In TAG, sampling periods or epochs are sub-divided among nodes in a path from the source to the sink. Data flows up the tree in a synchronized fashion to ensure parents can receive and process the data before relaying it themselves. TinyDB [10, 7, 9] distributes queries by flooding, and uses semantic routing trees to prune the sensors involved in query execution.

SenQ supports sensors like the ECG that demand high-rate streams to properly capture the signal for expert analysis. Distributing the query and streaming results must be efficient, with smaller delays than may be possible when flooding, pruning, and propagating data in lock-step.

TinyDB and Cougar [27, 28] provide declarative query languages similar to SQL. This abstracts many of the details of network operation from the user, and is a popular and useful approach for extracting data from the WSN, especially where wide-area or hierarchical aggregation is needed. However, it is not clear whether embedded user interfaces or even sensor devices themselves can autonomously issue queries using the same facility without involving a more powerful node to perform query planning and optimization. SenQ provides a uniform programmatic abstraction and network protocol that is accessible by users and embedded applications alike. Mapping higher-level declarative languages onto SenQ remains for future work.

In the TENET architecture [19], only resourceful nodes are allowed to perform data fusion in a strictly tiered network. VanGo [6] similarly requires the use of micro-servers for adaptive processing. They take advantage of an ADC/DMA capability to provide very high rate sampling, and have a static processing chain compiled into the motes. SenQ

deliberately does not require either powerful micro-servers or specialized hardware, and supports dynamic rewiring of processing chains.

None of the above systems provides all the capabilities that we needed for the highly interactive health-care testbed we designed.

Systems for medical applications are increasing as devices become smaller and less obtrusive. CodeBlue [13] is one example, and uses wearable devices for pre-hospital, emergency, and disaster-site care. SATIRE [3] is another application which collects data from body area networks in residential environments.

10. CONCLUSION

We have presented SenQ, a streaming query system and network protocol for IWSNs that supports diverse sensor types common in health-care applications, such as assisted living. It provides lightweight in-network processing that can be reconfigured to suit an application's needs, and extended by sensor and processing plug-ins. Network load and energy consumption is reduced by using temporal and spatial aggregation, predicate filtering, data compaction, and on-demand report buffering. Embedded interfaces and sensor devices can issue queries without the aid of powerful back-end servers, to enable distributed smart networking in the kind of interactive systems we expect to develop in the near future. We evaluated SenQ's resource consumption, sampling limits, and multi-hop performance over IGF to show that SenQ meets its design goals for flexibility, extensibility, and efficiency.

11. REFERENCES

- [1] B. Blum, T. He, S. Son, and J. Stankovic. IGF: A state-free robust communication protocol for wireless sensor networks. Technical Report CS-2003-11, Univ. of Virginia, Charlottesville, VA, 2003.
- [2] Crossbow Inc., Wireless Sensor Network Hardware, URL: http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.
- [3] R. K. Ganti, P. Jayachandran, T. F. Abdelzaher, and J. A. Stankovic. SATIRE: A software architecture for Smart AtTIRE. In *Proc. of Mobisys*, pages 110–123, 2006.
- [4] T. Gao, T. Massey, W. Bishop, D. Bernstein, L. Selavo, A. Alm, D. White, and M. Sarrafzadeh. Integration of triage and biomedical devices for continuous, real-time, automated patient monitoring. In *Proc. of ISSS-MDBS*, 2006.
- [5] J. Gehrke and S. Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004.
- [6] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *Proc. of SenSys*, 2006.

- [7] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proc. of IPSN*, 2003.
- [8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of MobiCom*, pages 56–67, 2000.
- [9] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of ICDE*, 2002.
- [10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM TODS*, 30(1):122–173, 2005.
- [11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *Proc. of OSDI*, Dec. 2002.
- [12] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of WSNA*, pages 88–97, Sept. 2002.
- [13] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *Proc. of the International Workshop on Wearable and Implantable Body Sensor Networks (BSN)*, 2004.
- [14] T. Massey, T. Gao, D. Bernstein, A. Husain, D. Crawford, D. White, L. Selavo, and M. Sarrafzadeh. Pervasive triage: Towards ubiquitous, real-time monitoring of vital signs for pre-hospital applications. In *Proc. of UbiHealth*, 2006.
- [15] A. N. Rogoza, T. S. Pavlova, and M. V. Sergeeva. Validation of A&D UA-767 device for the self-measurement of blood pressure. *Blood Pressure Monitoring*, 5(4):227–231, Aug. 2000.
- [16] L. Selavo, G. Zhou, and J. A. Stankovic. SeeMote: In-situ visualization and logging device for wireless sensor networks. In *Proc. of BASENETS*, 2006.
- [17] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. LUSTER: Wireless Sensor Network for Environmental Research In *Proc. of SenSys*, 2007.
- [18] SHARP Microelectronics. GP2Y1010AU: Compact Dust Sensor for Air Conditioners. URL: http://document.sharpsma.com/files/GP2Y1010AU_DS.pdf.
- [19] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET Architecture for Tiered Sensor Networks. In *Proc. of Sensys*, Nov. 2006.
- [20] Smiths Medical PM, Inc. BCI Micro Power Oximeter Board. URL: http://www.smiths-bci.com/html/Products/oem_products.htm.
- [21] K. K. Tremper and S. J. Barker. Pulse oximetry. *Anesthesiology*, 70(1):98–108, 1989.
- [22] G. Virone, N. Noury, and J. Demongeot. A system for automatic measurement of circadian activity deviation in telemedicine. *IEEE Transactions on Biomedical Engineering*, 49(12):1463–1469, Dec. 2002.
- [23] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of OSDI*, Nov. 2006.
- [24] A. Woo, S. Madden, and R. Govindan. Networking support for query processing in sensor networks. *CACM Special Issue: Wireless Sensor Networks*, 47(6):47–52, 2004.
- [25] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring. Technical Report CS-2006-11, Department of Computer Science, University of Virginia, 2006.
- [26] X10 HawkEye Wireless Motion Detector. URL: http://www.x10.com/products/x10_ms14a.htm.
- [27] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, Sept. 2002.
- [28] Y. Yao and J. E. Gehrke. Query processing in sensor networks. In *Proc. of CIDR*, 2003.