SenQ: An Embedded 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 application architectures, hardware capabilities, and user interactions that challenge existing centralized [1], or VM-based [2] query system designs. To support in-network processing of streaming sensor data in such heterogeneous environments, we created SenQ, a multi-layer embedded query system. SenQ enables user-driven and peer-to-peer in-network query issue by wearable interfaces and other resource-constrained devices. Complex virtual sensors and user-created streams can be dynamically discovered and shared, and SenQ is extensible to new sensors and processing algorithms. We evaluated SenQ's efficiency and performance in a testbed for assisted-living, and show that on-demand buffering, query caching, efficient restart and other optimizations reduce network overhead and minimize data latency.

1 Introduction

Wireless sensor networks enable fine-grained collection of sensor data about the real world. Applications are growing in military, environmental, health-care, structural monitoring, and other areas and many sensing modalities are now available. Integrating them on embedded platforms and efficiently managing their data remains a challenge due to application constraints on form-factor and high cost sensitivity.

One growth area for wireless sensor networks is the health-care domain, which already uses a wide variety of sensors, and could 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 residents' environmental conditions—but it must integrate with body area networks, embedded user interfaces, and back-end control and storage. The result is a heterogeneous and highly *interactive* wireless sensor network (IWSN) that presents new challenges for query systems.

This work describes SenQ, an embedded query system for IWSNs that makes several contributions to the state of the art:

* Supported by NSF grants CNS–0435060, CNS–0626616, and CNS–0720640.

– We identify core requirements for emerging interactive wireless sensor networks (IWSNs) and present the design of SenQ, a query system for in-network monitoring that addresses challenges from heterogeneity of: application architectures, user interfaces, device capabilities, and information flows.
– SenQ flexibly supports both hierarchical application architectures and ad hoc decentralized ones, by providing a stack with loosely coupled layers that may be placed independently on devices according to their capabilities, and by enabling in-network peer-to-peer query issue for streaming data. Its standardized software interfaces and protocol mechanisms support extensibility for new sensors, processing algorithms, and context models.
– A novel shared stream abstraction uses virtual sensors to encapsulate complex data processing and to discover and re-use dynamic user-created streams.
– Evaluation on a real implementation shows good performance of the software stack: query caching with restart halves the internal first-data latency to $670\,\mu s$, and we show sampling rates up to $1\,KHz$ with low jitter without ADC DMA or other specialized hardware.

The features of SenQ enable embedded control loops, user reminders, real-time delivery of body network data and other rich interactions within the system that are not supported in any integrated way by existing query systems. We evaluate SenQ in the context of AlarmNet [3], a testbed for assisted-living.

2 Research Challenges in Interactive WSNs

Emerging IWSN systems present research challenges and constraints that are not satisfied by existing query system designs in an integrated way. Here we identify the key challenges that SenQ addresses.

Heterogeneity. Diversity in IWSNs spans device types, capabilities, interface modalities, user-created data flows, and system architectures.

Deployment dynamics. Because IWSNs are human-centric and interactive, network membership and data flow patterns may change continuously.

In-network monitoring. Point-to-point data streams that remain entirely within the network are needed for several situations. 1) *Decentralized control loops* require fast access to sensor data to provide predictable performance. 2) *Embedded user interfaces* allow the ad hoc creation of data streams for personal consumption, and are a vital interaction method for applications.

Localized aggregation. Due to the heterogeneity of sensor types and information flows in IWSNs, most spatial aggregation is within small areas, such as a body area network that combines wearable accelerometer data to detect falls.

Resource constraints. Processor, energy, and memory capabilities of embedded sensor and interface devices are limited, especially for unobtrusive wearable devices with small form factors and low cost.

Data querying and management has received considerable attention in WSN literature, but we are aware of no systems that comprehensively address the additional requirements and challenges of interactive WSNs identified here. After briefly reviewing related work, we present the design of SenQ in Section 3.

2.1 Brief Examples of Related Work

Emerging IWSN systems must support *distributed data access* not just by back-end servers, but by users possessing a wide range of expertise. For traditional back-end interfaces, we want to retain the benefits of declarative query languages like that provided by TinyDB [1]. But they are unsuitable for most *embedded interfaces* with limited capabilities. Virtual machines like Maté [2] and SwissQM [4] provide a flexible programmatic approach useful for sophisticated in-network processing, but their low-level abstractions and compilation and interpretation overhead are not well suited for in-network query issue.

In TAG [5], 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 [1] distributes queries by flooding, and uses semantic routing trees to prune the sensors involved in query execution. For relatively high-rate streams, however, the delays involved in these approaches may be prohibitive.

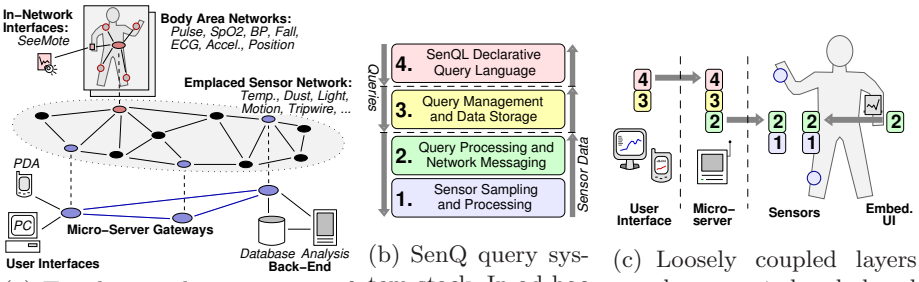
TinyDB and Cougar [6] provide declarative query languages similar to SQL that hide many of the details of network operation from the user and ease construction of queries. However, textual query languages are less useful for embedded user interfaces or sensor devices themselves, and efficiency suffers if the queries must be relayed to a server for parsing and execution. SenQ provides a uniform programmatic abstraction and network protocol that can be used directly by embedded applications.

In the TENET architecture [7], only resourceful nodes are allowed to perform data fusion in a strictly tiered network. VanGo [8] similarly requires the use of micro-servers for adaptive processing. They take advantage of an ADC DMA capability to provide high rate sampling, and have a static processing chain compiled into the motes.

3 SenQ Query System Architecture

The requirement for decentralized, in-network monitoring had a large impact on the design of SenQ, particularly its layered structure shown in Figure 1(b). In a full system deployment, the lowest layers 1 and 2 (sensor sampling and query processing, respectively) reside on embedded sensor devices. Since these are heavily resource-constrained, the software must be efficient with small memory footprint. Layer 3 (query management and storage) resides on micro-server gateways with more abundant resources, such as a connected database and back-end systems. Layer 4 is a high-level declarative language, SenQL, similar to SQL and TinyDB [1] for external user-issued queries. Due to space contraints, we only present the design details of SenQ's bottom two layers.

Arrows in Figure 1(b) show the nominal data flow: from query language to micro-server for authorization, binding, and translation, into the WSN to the sensor device where it is parsed and activated; then streams of data flow back through the micro-server to the user's interface.



(a) Topology and components of AlarmNet, a prototypical IWSN for assisted-living.

(b) SenQ query system stack. In ad hoc mode the upper two layers are optional.

(c) Loosely coupled layers may be separated and placed on heterogeneous devices as resource constraints allow.

Fig. 1. SenQ supports both hierarchical and ad hoc architectures by maximizing layer independence.

However, embedded user interfaces and sensor-initiated queries characteristic of IWSNs are more efficiently supported when the upper layers are bypassed. And not every system will have an architecture with centralized control, even if only temporarily. For example, an elder wearing a body area network disconnects from AlarmNet's infrastructure (shown in Figure 1(a)) while visiting the doctor. It is desirable for the body network to continue to monitor health status and allow the doctor to query vitals in real-time. Other systems may use only sensors and embedded UIs all the time in a low-cost ad hoc network topology.

To satisfy diverse application needs, we designed the layers of SenQ's stack to be *loosely coupled* with well-defined interfaces throughout.

3.1 Query and Data Model

To solve the research challenge of supporting significantly different information flows simultaneously with low delay, SenQ provides snapshots and streams.

Streaming queries specify the sensor, sampling rate, processing chain, and whether to perform local-area spatial aggregation. As data is collected, reports are streamed back to the requester until a *Stop* command is received or an optional maximum duration is reached. *Snapshot* queries provide efficient point-in-time samples of raw sensor values that bypass the entire processing chain for minimal response times.

For both types of queries, the returned sensor data is composed of $<$*timestamp, value*$>$ tuples. Query caching allows them to be efficiently *stopped and restarted* later with a short command from the originator that minimizes communication, parsing, and startup overhead.

Queries are uniquely identified in the network by $<$*source address, ID*$>$ tuples to allow multiple concurrent queries on sinks and sources—a requirement for the interactive, peer-to-peer traffic flows in IWSNs. The $4\,KB$ of data memory on the MicaZ mote limits SenQ to 21 concurrent queries on each sensor.

3.2 Sensor Sampling and Processing Layer

In contrast to many environmental monitoring networks, IWSNs support a wide variety of sensor types—there are currently twenty in AlarmNet. Sensor data is accessed using internal components, external ADC channels, UART serial links, or interrupts. This heterogeneity complicates the addition of new sensors.

The Sampling and Processing layer (shown in Figure 2) encapsulates access to onboard resources to insulate applications from the complexity. Standard interfaces for the sensor drivers and processing blocks allow them to be easily incorporated and plugged-in (wired) at compile time, and enables the virtual sensor feature described later in Section 3.3.

SenQ treats the sensor and processing types opaquely, so that old devices ignore unknown types. This lets new sensor types be deployed dynamically into the network without having to reloading code on existing devices, and maintains continuity of operation for environments—such as health-care—where it is not practical or safe to download new code and reset the system.

Sensor Drivers. Sensor drivers are categorized by the timing and regularity properties of their access, since these determine the most efficient way for SenQ to sample them. *EventSensors* generate data sporadically as it becomes available, such as from an interrupt, and so need not be sampled periodically. *SplitPhaseSensor* represents sensors which must read and convert data upon request, such as from analog sensors connected to an ADC. Data is provided asynchronously to SenQ. Data that may be quickly read synchronously uses the *PollableSensor* interface. Drivers also provide a *SensorInfo* interface to aid runtime discovery of nodes' capabilities and types.

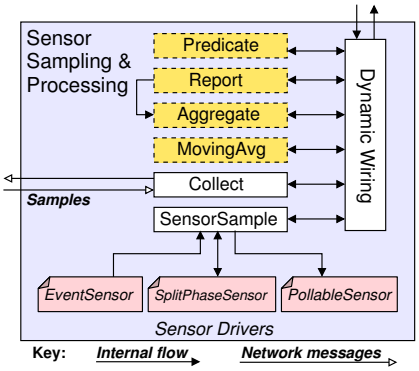


Fig. 2. Layer one in SenQ samples and processes sensor data.

Physiological sensors in AlarmNet include pulse oximetry (Harvard [9] and UVA designs), wearable two-lead electrocardiography (Harvard [9]), and body weight and blood pressure devices by A&D Medical. Long-running background streams monitor residents' environmental conditions, such as air quality, light, temperature, and humidity. Sensors detect motion and activity to inform context-aware back-end algorithms using PIR motion, optical tripwires, magnetic reed switches, and wearable accelerometers for classifying movement-based activities.

Processing Chain. Some sensors require little in-network processing, but for those that provide a high-volume of data, it is essential to reduce both the energy cost and network congestion from sensor streams by filtering at the source.

Above the hardware and sensor drivers is a group of modules comprising a scheduler and data processing chain, collectively called the Sampler. They act in concert to manage sensor sampling for multiple concurrent queries, and filter generated data according to the application query.

Processing modules provide a *ProcessControl* interface with standard control and configuration methods. These allocate structures, pass query configuration to the process, and supervise sampling. As with sensor drivers, the interface eases the extension of SenQ to new, application-specific processing algorithms.

DataProducer interfaces on each block provide incoming and outgoing paths for sensor data. Instead of a static sequence (as in VanGo [8]), blocks are wired to a dynamic data flow coordinator. This provides more flexibility to the application, since each query has its own ordering of the processing chain.

The *SensorSample* module maintains a schedule for sensor sampling to satisfy multiple ongoing queries. A single timer tracks the next sampling operation. Upon expiry, data is requested from the driver according to its category, *Event*, *SplitPhase*, or *Pollable*. When data is available it is propagated up the processing chain to the next consumer, as determined by the query's dynamic wiring.

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

SenQ provides an optional *Aggregate* module that is optimized for functions or descriptive statistics that can be calculated while keeping little state. In addition, a "latch" aggregator is provided for polled binary EventSensors. An event that occurs within a report period will be remembered until the next report.

The Aggregate module works in conjunction with the *Report* module, which drops intermediate samples until a 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 4*s*, with intermediate results averaged by the Aggregate module.

An optional *Moving Average* module provides a windowed average, moving average, or exponentially-weighted moving average (EWMA) with specified window or α parameters.

Finally, queries may cull irrelevant or redundant data by specifying relational *Predicate* filters. The query specifies the argument values for comparison.

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

Fig. 3. Memory consumption for processing modules in bytes on MicaZ.

Figure 3 shows the memory tradeoff between the Aggregate and Moving Average modules. With four concurrent queries and a window of ten 32-bit elements, the increase in data memory usage over the Aggregate + Report configuration is 142 bytes, but with 652 fewer bytes of code memory. The variety of such tradeoffs among applications and hardware platforms is the reason SenQ preserves modularity of processing algorithms in the design.

Spatial Aggregation. IWSNs may not need to sample and aggregate sensor data from the entire network, due to their heterogeneity. For example, although

aggregate environmental data is useful in AlarmNet, physiological and activity data must not be mixed among residents.

Spatial aggregation is needed more often for collecting data from other nodes in the local area, such as in a body area network. A flag in the stream query specifies that the recipient is to act as the coordinator of a spatial query. This coordinator node sends the query to its immediate neighbors, including its own network address and sample ID.

Neighboring devices possessing the requested sensor type then process the delegated query. Sensor data is sampled and flows up the processing chain, eventually passing through the *Collect* module. The samples are redirected over the network to the query coordinator's Collect module in a *Sample* message (shown in Figure 2), where they are combined with local samples and inserted into the coordinator's processing chain.

Overall, the Sensor Sampling layer provides flexible mechanisms for access to heterogeneous sensor types and extension to new ones. However, so far the data is available only locally. For application and external access we need a Query Processing layer, which is now described.

3.3 Query Processing and Network Messaging Layer

A Query Processing layer (Figure 4) provides stream and snapshot abstractions to local applications via a software API, and remote ones via a network protocol. Queries for local sensors allocate resources in the Sampling and Processing Layer below, configure the processing chain, and start collection of data. Data is buffered (if the query allows) and reported to the originator, whether local or remote.

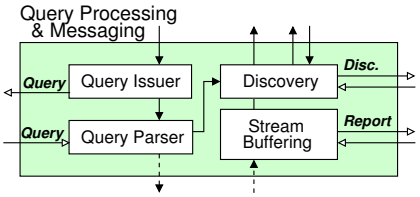


Fig. 4. Layer two in SenQ may stand alone or above the Sampler.

SenQ provides *location transparency* with respect to embedded applications, since they use the same interface to issue queries for local or remote sensors, and the networking aspects are hidden behind a QueryProcessor software interface. Queries for remote sensors are marshalled and routed to the destination, while local queries are managed by the resident Query Processing and Sampling layers.

Embedded query issue capabilities distinguish SenQ from most existing solutions and open many possibilities for smart in-network processing, human interactivity, and embedded control loops. This accrues particular benefits in decentralized, large-scale networks where many data flows exist. Centralized approaches like TinyDB [1] and SwissQM [4] do not cater well to networks in which many point-to-point streams are dynamically created between embedded query issuers and sensors.

Control loops can be embedded in the network so, for example, a light controller sends long-lived queries to a room's door or motion sensors and acts directly and locally on the lights. Control input, decision, and actuation are all close to the data source and suffer lower delays than a centralized approach.

SenQ maximizes *layer independence* to provide flexibility for heterogeneous platforms and architectures. An embedded device with no sensors includes only a Query Processing layer, so it can issue queries. In AlarmNet, the MicaZ-based SeeMote [10] (shown at right) includes SenQ, a graphics library, and a real-time data visualization application in only 4 *KB* of SRAM.

A *Discovery* component allows an embedded application to locate nearby devices, sensor types, and processing modules. Rather than relay queries through the gateway, which incurs extra delay and energy costs, smart sensors can discover each other and issue queries directly.

Virtual Sensors. User-driven creation of data streams in an open environment is a central characteristic of IWSNs, and a query system should support three classes of users: developers, application domain experts, and the ordinary user. An "ordinary user" of an IWSN could be an assisted-living resident, an elder or soldier recovering at home from an injury, or a student immersed in a campus-wide sensor network.

High-level declarative languages are good for application experts knowledgable about relational database abstractions and the capabilities of the system. However, these languages are a poor choice for system developers who must create specialized processing algorithms, and as a basis for network protocols they are too verbose and require complicated parsing.

SenQ enables a developer to use its embedded query issue capabilities to collect data streams from both local and remote sensors for custom processing, and then export the results as a *virtual sensor* at the bottom of the stack that conforms to the sensor interfaces described in Section 3.2. This encapsulates the complex, hierarchical stream processing as a low-level sensor type that can be discovered, queried, and viewed as any other.

Shared Streams. Declarative and programmatic access methods support domain experts and developers—but they do not consider the ordinary user. This user may face challenges of: 1) unfamiliarity with relational databases and programming, 2) embedded interfaces with poor input capabilities, and 3) uncertainty of domain parameters (e.g., age-appropriate "normal" heart rates).

Shared streams build on the virtual sensor capability to address these challenges. A domain expert crafts a custom stream Q at runtime using an appropriate interface, and enables sharing of the query. The Query Processing layer dynamically allocates a virtual EventSensor VS in the Data Sampling layer, and then the Discovery component advertises VS as a primitive sensor type. When a new query Q' is received for VS, the Sampling layer recursively activates the Query Processing layer for query Q.

Together, virtual sensors and stream sharing enable novel ad hoc user-to-user interactions in the IWSN that are usually outside the scope of other query approaches. Systems using TinyDB or Cougar for declarative data access, or Maté or SwissQM for virtual machine-based access would have to develop additional protocols or user interfaces to provide this capability. By supporting re-use of

custom-crafted sensor streams, SenQ helps to address the challenges of providing open access to ordinary users.

Network Efficiency. The query processing layer uses several techniques to maximize performance for streams in resource-constrained embedded systems.

Combining multiple samples received by the Query Processor into a single report message saves overhead and reduces radio traffic—at the expense of latency. Query originators may specify full buffering or on-demand buffering.

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. It represents a tradeoff between latency and loss.

Reports with data are timestamped 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. Reports also bear status changes, such as positive and negative acknowledgements with cause codes, which are important for meaningful user feedback on embedded UIs.

Compacting reports reduces energy wasted in the transmission of redundant data. Other compression schemes, such as run-length encoding, can be added as processing plug-ins if an application warrants the additional computation.

Memory constraints of WSN devices limit the number of queries that may be simultaneously serviced or stored. Inactive queries are replaced using a least-recently used policy to maximize the ability of applications to restart them later. Restarting a *cached query* that has already been parsed and configured is twice as fast as issuing a new one.

3.4 Query Management and SenQL Layers

Upper layers in Figure 1(b) are described here only briefly due to lack of space.

In hierarchical networks, a *Query Management* layer provides services that are important for usability, context-awareness, connectivity, and data analysis. It manages device registration and client connections, and maps queries for abstract entities (e.g., people and places) onto particular devices.

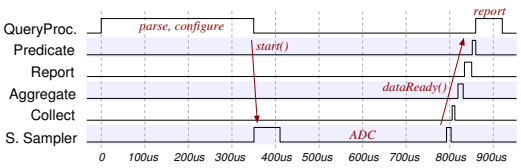
The *SenQL* layer provides a declarative query language to users, allowing them to specify what data is desired independent of how it will be collected. It uses a constrained subset of SQL–99 with extensions for SenQ functionality.

4 Evaluation

We present SenQ's consumption of memory and CPU resources and show the performance limitations of the sampling and processing chain to demonstrate runtime efficiency.

4.1 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,



(a) A 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 μs ADC time. 100 trials.

Fig. 5. SenQ timing measurements on the MicaZ mote.

and the number of maximum concurrent queries supported. Table 1 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 the Base.

We measured SenQ's load on the sensor device using an Intronix LogicPort logic analyzer. This gave accurate profiling of processing times with very little measurement overhead.

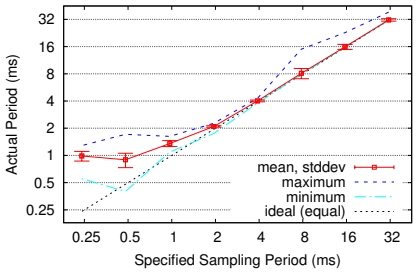
A timing trace from one experiment shows the relative magnitudes of processing times (shown in Figure 5(a)). A stream query samples the node's internal battery voltage every 73 ms, with no processing or buffering in the QueryProcessor. The timeline starts when the query is received from the network, and shows the time to parse it, allocate data structures, and configure and start the Sampler (358 μs).

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

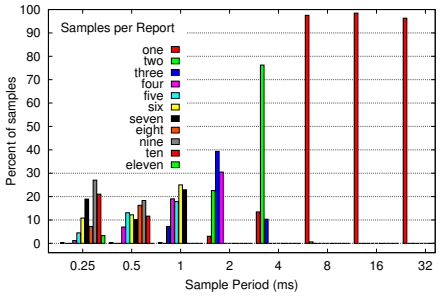
Table 1. Memory consumption for sensor drivers in bytes. Includes required TinyOS components (radio, timer, ADC, etc). Values for sensors are relative to the Base configuration.

SensorSampler started the sampling timer and requested data from the Voltage driver (50 μs). ADC conversion takes 25 ADC clock cycles, or 366.42 μs from the request until the data is available. Then the SensorSampler propagates it up the processing chain. Since no processing was specified, it reaches the QueryProcessor 68 μs later.

QueryProcessor generates a report immediately since buffering is not enabled. The total time spent on the first sample is 918 μs, of which 366 μs is waiting for



(a) 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 $S <= 1\,ms$.



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

Fig. 6. SenQ sampling jitter for a processing- and transmit-intensive query, and impact on on-demand buffering.

the ADC. 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 5(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 $8\,MHz$ 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%.

4.2 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.

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 so a message is transmitted for *every* sample. The timing was captured precisely by the logic analyzer.

Figure 6(a) 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, shown in Figure 6(b). 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. By applying this aggregation at the data source, network overhead is reduced while low latency and loss are preserved.

5 Conclusion

SenQ supports heterogeneous device types, user interfaces, data flows, processing algorithms, and application architectures. Network load and energy consumption is reduced by using temporal and spatial aggregation, filtering at data sources, data compaction, and on-demand report buffering. Virtual sensors and stream sharing enable rich user interactions, and embedded interfaces and sensor devices can issue queries without the aid of powerful back-end servers. SenQ enables distributed smart networking for the kind of interactive systems we expect to see in the near future.

References

1. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An acqusitional query processing system for sensor networks. ACM TODS **30**(1) (2005) 122–173
2. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. In: Proc. of ASPLOS. (2002) 85–95
3. Wood, A., Virone, G., Doan, T., Cao, Q., Selavo, L., Wu, Y., Fang, L., He, Z., Lin, S., Stankovic, J.: ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring. Technical Report CS-2006-11, Department of Computer Science, University of Virginia (2006)
4. Müller, R., Alonso, G., Kossmann, D.: A virtual machine for sensor networks. In: Proc. of EuroSys. (2007) 145–158
5. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: a Tiny AGgregation service for ad-hoc sensor networks. In: Proc. of OSDI. (2002) 131–146
6. Yao, Y., Gehrke, J.E.: The Cougar approach to in-network query processing in sensor networks. SIGMOD Record **31**(3) (September 2002) 9–18
7. Gnawali, O., Greenstein, B., Jang, K.Y., Joki, A., Paek, J., Vieira, M., Estrin, D., Govindan, R., Kohler, E.: The TENET architecture for tiered sensor networks. In: Proc. of SenSys. (2006) 153–166
8. Greenstein, B., Mar, C., Pesterev, A., Farshchi, S., Kohler, E., Judy, J., , Estrin, D.: Capturing high-frequency phenomena using a bandwidth-limited sensor network. In: Proc. of SenSys. (2006) 279–292
9. Malan, D., Fulford-Jones, T., Welsh, M., Moulton, S.: Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In: Proc. of BSN. (2004)
10. Selavo, L., Zhou, G., Stankovic, J.A.: SeeMote: In-situ visualization and logging device for wireless sensor networks. In: Proc. of BASENETS. (2006) 1–9