WSN deployments: designing with patterns

Brusey, J. , Gaura, E. and Hazelden, R.

Original citation & hyperlink:
Brusey, J. , Gaura, E. and Hazelden, R. (2012). 'WSN deployments: designing with patterns' In Sensors, 2011 IEEE. (pp.71-76). IEEE.
http://dx.doi.org/10.1109/ICSENS.2011.6127129

WSN Deployments: Designing with Patterns

James Brusey, Elena Gaura
Cogent Computing, Coventry University,
Priory Lane, Coventry, UK CV1 5FB
j.brusey@coventry.ac.uk

Roger Hazelden
TRW Conekt, Solihull, UK
roger.hazelden@trw.com

***Abstract*—Development of application-specific wireless monitoring systems can benefit from concept reuse and design patterns can form the enabling medium for such reuse. This paper proposes a set of five fundamental node-level patterns that resolve common problems when programming low-power embedded wireless sensing devices. The pattern set forms a framework that is aimed at ensuring simple and robust deployed systems. A qualitative evaluation is performed by identifying key design traits from several successfully deployed systems and linking these to elements of the framework.**

***Index Terms*—Design patterns, Wireless Sensor Networks, Programming methodology**

I. Introduction

The idea of automatically and wirelessly acquiring data from a distributed set of sensors is relatively recent—feasible wireless sensors have only been readily available for the last decade or so.

The increasing number of successfully fielded deployments together with the emergence of strong business cases in a variety of industries are currently beginning to move distributed sensing into the mainstream. Application developers are however still faced with a technology that: i) is hard to understand, ii) is difficult to make reliable, iii) necessitates long development cycles including iterative prototyping processes and iv) does not inherently offer an opportunity to grow applications post deployment, over time, as the potential usage for sensory data in a given scenario evolves.

For these reasons, a well-structured, systematic development process and framework are required to ensure that: i) new applications and additional sensors can be easily integrated as the system grows and, ii) the development cycles are reasonably cost effective.

Some support exists for simplifying the development of wireless sensor systems (e.g. TinyOS, Embedded Linux). However, there is little work on guidelines or frameworks that establish best practice in this area. In particular, there appears to be a series of identifiable lessons that are being repeatedly rediscovered by programmers and research groups. Part of the difficulty is that Wireless Sensor Network (WSN) deployments are diverse. There appears to be little carry-over in terms of lessons learnt from one deployment to the next simply because many of the issues do not apply. Component reuse exists (e.g. Collection Tree Protocol (CTP) in TinyOS), however this is mainly occurring where an approach is so well established that it is integrated into the operating system. For application-level reuse to occur widely, a paradigm shift is required.

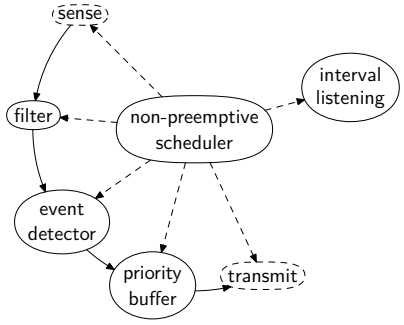

Figure 1. Pattern-based Framework for nodes. A dashed box is drawn for elements that have no explicit pattern described here. Individual patterns are shown in Figures 2 to 5.

Towards the above, the main contribution of this paper is a set of five fundamental node-level patterns that resolve common problems when programming low-power embedded wireless sensing devices. The pattern set forms a framework that is aimed at ensuring simple and robust deployed systems. A qualitative evaluation is performed by identifying key design traits from several successfully deployed systems and linking these to elements of the framework.

The reminder of this paper is structured as follows: Section II describes the proposed framework and the node-level patterns featured in the framework, together with relevant examples of patterns usage in successful deployments. Section III concludes the paper.

II. WSN Design Patterns

A. A Pattern-based Framework

Definitions: a) Framework refers to a form of proto-architecture where elements may be added, removed, or altered to suit an application; it encapsulates the collection of related design patterns. b) A design pattern is a template or guide for solving specific design problems with software development for WSN systems.

Assumptions: a) There are benefits to processing at the node in a wide variety of applications [10], [5]; b) Deployed WSNs eschew complexity. Benefits from sharing information between (leaf) nodes within a network are rare in practice and even less frequently worth the associated design complexity and deployment risks [17], [18]. Thus, single-hop and multi-hop node-to-sink configurations are considered here. This approach is aligned with Raman and Chebrolu [17].

The proposed framework, with its five node-level patterns, is depicted in Figure 1. The approach is general as the sensed

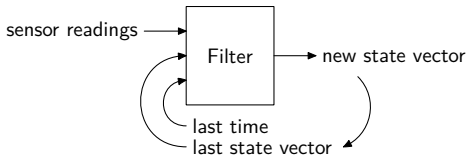

Figure 2. Overview of the Filter Pattern.

data and the subsequent inferred state are described by simple vectors; thus, the framework lends itself to a wide variety of possible sensors, phenomena, and applications.

The central task for the node begins with the "sense" operation. Noise is *filtered* and the original data is transformed into meaningful information. *Event detection* occurs next potentially leading to a message that must be transmitted. The message is *buffered* according to *priority* before being transmitted. These tasks must be interleaved along with sleeping and *listening at intervals* and this is the job of the *scheduler*. In the following subsections each design pattern is described in more detail. A general description of each pattern is given, together with its *Aims* (what is the intended function achieved by the pattern), *Triggers* (the reasons why the pattern might be used), *Collaborators* (which other patterns are often used with it), *Possible Extensions* (additional functionality) where appropriate, and *Examples* (some examples from the literature of successful use of the pattern). A summary of *Aims*, *Triggers* and *Examples* for each pattern is also given in Table I.

B. Filter Pattern

The Filter pattern (see Figure 2) could also be termed the Model-based Smoothing pattern. Its aims are to *smooth* the raw, sensed data and/or *infer the state* of the phenomena at the node.

Whilst Filtering is well understood and generally applied as a post-processing step on sensed data, at the sink, in some applications there are specific reasons (*Triggers*, in Table I) for performing filtering at the node. Primarily, filtering at node ensures that other processing, such as Event Detection (an issue distinct from Filtering), is affected minimally by noise and that the scarce (by and large) resources within a WSN are optimally used to fulfil the application. High data rate sensing applications would particularly benefit from this node-level pattern. Information or knowledge driven WSN system designs [6] would also have Filtering as a key pattern.

Filters are often explicitly model-based and their outputs are state vectors. Commonly a filter attempts to derive an estimate of the state of the system based on past sensor readings. Often it is possible to assume that the system has the Markov property, which means that the most recent sensor reading and the last state estimate are all that are required to estimate the current state and that no better can be done by knowing the complete history of states.

Exponentially Exponentially Weighted Moving Average (EWMA) and Kalman Filters are common choices for implementing this pattern.

1) Collaborators: The Filter pattern is often used in collaboration with the Event Detection pattern, given that:

- The process of filtering removes noise thus reducing spurious event detection.
- Transforming the raw sensor data into a state vector simplifies the task of identifying whether the state has changed in a way that can be considered a meaningful event.
- It supports avoiding a "slippery slope" problem where the event detection mechanism cannot detect a change if the change occurs slowly enough.

Further, combining the Filter pattern with the Interval Listening pattern can avoid the possibility that the energy saved from reducing transmissions is then subsequently lost due to excessive listening time.

2) Possible Extensions:

- The framework begins with the assumption that individual nodes do not share information and are not required to communicate directly with one another. For some applications, however, it may be useful to allow such communication. In this case, the state estimate produced by the Filter can take into account measurements from neighbouring nodes. For example, an animal call detection system might consider a possible animal sighting more likely if a number of neighbouring nodes are sensing a call (e.g. VoxNet [2]).
- The Lance architecture [18] suggests a useful extension that involves locally storing the original (unfiltered) data and providing it on-demand while normally sending only summaries. The summaries, plus perhaps information from neighbouring nodes, can be used at the sink to work out if the original data is likely to be interesting. This approach is particularly useful when local information is not sufficient to fully make a decision about how useful the data is.
- The state vector need not be just about the phenomena. It is often useful to expand the state vector to include management information or, in other words, information about the state of the sensors or the wireless node. For example, this could include local timestamps, battery voltages, estimates of uncertainty in measurement readings, link reliability statistics, and so forth.

3) Examples:

- Lance [18] is an architecture built originally for volcano monitoring that made use of seismoacoustic sensors. Transmitting all of the audio over a multi-hop network led to much contention and low yield. By sending summaries of the data instead, the bandwidth requirement was significantly reduced and the yield of useful data improved.
- The Cane Toad monitoring project [10] is another excellent example of successfully filtering complex audio data on the node. Frog calls were collected in the wild and analysed in real-time using spectrograms and C4.5 decision trees to classify the frog species. Whilst the initial deployments required sophisticated processors on nodes, efficient filtering allowed successful designs based on Mica2 motes. Filtering was used to convert raw audio data to identified frog species on-node, thus reducing the bandwidth requirement for this application

Table I
NODE-LEVEL PATTERN DESCRIPTIONS.

Pattern	Aims	Triggers	Example
Filter	Reduce noise, summarise a sensory "chunk", and infer state (possibly from sensors of differing modality). It is important to keep distinct the two issues of: 1) filtering, which transforms data into an estimate or summary of the state, and 2) event detection, which detects whether the change in the state is significant.	Available bandwidth is low relative to the amount of data sensed. The relative cost of transmitting data is higher than processing it on the node Actual sensor readings are not necessarily required (or only contingently required).	Lance [18]
Event Detector	Reduce the transmission of unnecessary data. Allow for increased rate of transmission of needed data.	The system being measured has a steady or easily predicted state for extended periods. Transmission cost (say, in terms of energy or bandwidth use) is high.	Posture monitoring [3], VoxNet [1]
Priority Buffer	Increase likelihood that important packets are transmitted. Reduce transmission medium contention. gracefully handle extended periods without the ability to transmit.	Messages priority varies. Successful transmission likelihood varies over time.	Glacier monitoring [13]
Non-preemptive scheduling	Provide efficient interleaving of sleeping, sensing, listening, and transmitting cycles. Allows for timed communication for listening and transmitting. Support long running or slow external sensors with minimal CPU.	Multi-tasking operating system avoided or not feasible. Need for more complex task interleaving than possible with a simple sense-process-send-sleep cycle. Not possible to use an off-the-shelf non-preemptive OS (TinyOS or Contiki), due to limits of microprocessor, or in an attempt to reduce the power budget.	TinyOS
Interval listening	Support mesh-networking. Allow nodes to spend most of their time asleep but still not miss (most) messages. Reduce the amount of time spent "idle listening".	Multi-hop networks. Relatively low-frequency sensing.	LPL (B-MAC) [14], TSMP [15]

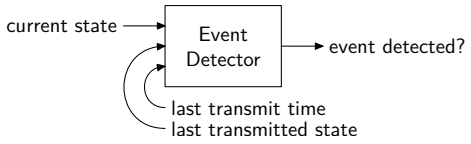

Figure 3. Overview of the Event Detector Pattern.

and allowing it to be implemented with inexpensive motes.

C. Event Detection Pattern

The concepts of "events" and "state" are more formally defined by Cassandras and Lafortune [4]. Within the framework presented here, event detectors compare the current state with the last transmitted state. If the difference exceeds some threshold, then an event is detected (see Figure 3 and corresponding entries in Table I). Comparing with the last *transmitted* state avoids the possibility of sending duplicate event messages. Furthermore, using the last *transmitted* state for comparison (as opposed to the last *sensed* state) avoids a "slippery slope" effect where a slowly changing phenomena may appear to be uneventful (the gradient at any point is low) but the long term change is still significant.

The event detection pattern is most suited to mature designs, where the phenomena is well understood and the informational outputs required to be delivered by the application are clear.

In the authors view, the pattern needs to be defined in terms of "state", rather than "sensor reading" since it is typically the case that the raw, unsmoothed, uncalibrated reading will first be processed into an application-specific state vector by a Filter prior to event detection. For example, it is simpler to design event detection based on a state vector that includes, say, an estimate of the residual life rather than one that gives wear sensor measurement readings.

For systems that are predictable over time, a predictive model is needed to correctly detect events. For example, if the last transmitted state was taken 5 minutes ago and indicated that the state was at 1 unit and rising linearly by half a unit every minute, then the predicted state is 3.5 units. If the new state estimate is within some threshold of the predicted state estimate, then it is considered uneventful (i.e. it would not be interesting to the sink, which can already do the prediction). In principle, arbitrarily complex models could be used here. In practice, however, simple linear regression is sufficient for most cases.

A further advantage of event detection is that it may save sufficient transmission energy and bandwidth to allow an increase in sensing frequency. This potentially allows detection of short-lived phenomena that might be missed otherwise.

Steady state systems are reasonably common and, for these systems, the use and benefit of the Event Detector pattern is more obvious. Less obvious is the application of event detection to systems that follow diurnal, periodic, or short term linear trends. Some examples include: temperatures within a building, water pressure within the water supply pipe network, wear on machine bearings, and so forth.

1) Collaborators: The Event Detector pattern is often used in conjunction with the Filter pattern. In fact, they are so often used jointly that it is easy to confuse them or not to know when to use one without the other.

Filters are used *without* event detection when the decision about whether or not an event has occurred must be deferred until more information is known. Perhaps the decision can only be made at the sink, when summaries from other nodes have been collected.

Event detection is used *without* filtering when the sensor already provides a sufficiently clean signal. For example, an RFID tag reader provides tag-read messages that are free

from noise. Event detection is needed to identify when tagged items appear or disappear. Even in this case, it may still be useful to have a "filter" to organise the incoming tag-read messages into an estimate of which items are present (i.e. a representation of state).

2) Possible Extensions: There are several ways in which to extend the basic Event Detector pattern:

- Incorporating a "heartbeat" message can ensure that the sink will eventually detect node failure. Without this, the node might not send any data for an indefinite period, if the phenomena is in a steady state. A simple method to incorporate a heartbeat is to signal an event if the last transmission time was long ago, even if the state is unchanged. (The exact definition of how long to wait before sending a heartbeat will depend on the application.)
- Model-based event detection (based on predicting from linear or other trends) can be further enhanced by assuming that the sink can also apply the same model-based prediction. The Spanish Inquisition Protocol [8] describes an event detector that makes use of dual prediction (on both node and sink).
- A useful assumption is that the state vector (used as input) has the Markov property. This is a helpful consideration when deciding what features to include in the state vector. For example, rate of change is needed if one wants to predict based on a linear extrapolation of the trend.

3) Examples: The use of the Event Detector pattern is commonplace in the literature. Two interesting examples are given below, both dealing with high data rate sensors.

- VoxNet is a deployed WSN that localises animal calls using a set of four microphones at each node [2], [1]. Full trilateration of incoming audio signals could only be performed at the sink, however sending all of the audio signals tended to overload the 802.11 network used. To reduce the network load, an event detector was used to detect and transmit only start times and end times of animal calls. The sink would then decide, based on these, which nodes and which time periods to query for full audio data.
- Event detection for human activity monitoring systems can substantially reduce transmissions. In work elsewhere [3], a postural activity monitoring system was developed that classified posture based on two or more body worn accelerometers. A combination of on-node posture classification, an exponentially weighted voting filter and event detection reduced the transmission rate from the original 10 Hz to about 0.3 Hz with event detection but without filtering, and to 0.06 Hz with filtering included.

D. Priority Buffer Pattern

A synopsis of the Priority Buffer pattern is given in Figure 4 and Table I. The first part of the pattern consists on ordering the buffer according to priority. (The priority of any message is determined by the contents of the message.) The second part consists of controlling the timing of transmission and, in particular, controlling when transmissions should be

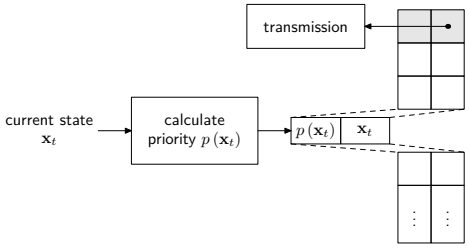

Figure 4. Overview of the Priority Buffer Pattern.

retried. This simple pattern can be critical in ensuring that high priority messages are communicated successfully and that the deployed system gracefully handles extended periods without the ability to transmit.

Traditional wired networks assume that the probability of any given transmission failing is always the same. Wireless networks, however, suffer from variations in failure probability. For example, mobile wireless devices may be in range and able to communicate for some long period and then subsequently out of range or RF occluded from communicating for a period. Fixed devices can have similar variations in failure probability due to environmental factors such as rain or snow, the movement of occluding objects, and so forth. For this reason, when a transmission fails, particularly if it has already failed several times, it may not be best to retry immediately. Where communication is failing because of contention for the transmission medium, reducing the number of attempts to transmit will help to reduce contention and this is an important consideration for the designer of a Priority Buffer.

Communication may also be failing due to a transient environmental effect (such as rain or snow) that will continue to prevent successful transmission for some time. An application-level strategy can balance the importance of timely transmission against the cost of many retries.

The Priority Buffer pattern responds to this problem by:

1) raising the communication buffer to an application level (rather than an operating system one),
2) allowing re-ordering of transmissions by priority even when a transmission has failed, and,
3) allowing control of when retries should occur.

Dealing with the communication buffer at an application level means that it is possible to support much larger buffers than usual, perhaps making use of flash memory. Furthermore, since message headers have not been added yet, the individual messages will be smaller. When communication is cut for an extended period, this application buffer may be sufficient to ensure that no information is lost. The pattern is essential in safety critical applications [11] in particular.

1) Collaborators: A critical question when devising a Priority Buffer is how to determine which messages are more important. In particular, the state vector should contain enough information to enable a decision about its priority to be made. This implies an interaction with the associated Filter. The Filter helps the Priority Buffer by placing sufficient context into the state vector.

2) Examples: During the development of a glacier monitoring application, Martinez et al. [13] had the problem

of wirelessly transmitting from the glacier to an Internet café several miles away. To save power, communication was reduced to a few transmissions per day. However, snow storms would severely disrupt communication. If the transmission was continuously retried throughout the storm, it would just drain the batteries. Therefore, a series of three failures caused the node to give up transmitting for several hours before retrying. During the intervening period while communication is down, a series of events might be detected. When communication is re-established, the Priority Buffer plays a key role in ensuring that most recent or most important events are sent first.

The above example illustrates how it is important to consider the application and its environment. It also shows how useful it is to elevate the question of when to retry to an application-level, rather than leaving this to the operating system, to avoid wasting battery power and to allow consideration of the priority of the message being communicated.

E. Non-preemptive Scheduler Pattern

Non-preemptive scheduling is a central component of simple, embedded operating systems such as TinyOS. There are two reasons for declaring this as a pattern. The first reason is that an understanding of the implications of this pattern will enable developers to best use TinyOS and similar systems. The second reason is that there are still many specialised applications that call for simpler hardware or more stripped down software than TinyOS or a similar operating system would allow but where task interleaving and timed operations are still required.

While preemptive scheduling is the norm in modern computers, low-power microprocessors or generic PIC microcontrollers, which are widely used for WSN applications, tend to be limited in their support for fundamental multitasking building blocks such as task switching and memory protection. Nonetheless, hardware interrupts, due to timers and I/O, will interrupt the main processing loop and care is needed to ensure that there are no race conditions for memory areas shared between the main process and the different interrupt routines.

Correctly dealing with hardware interrupts is a key issue for this pattern. As pointed out by Pont [16], high priority interrupt service routines may mask lower priority interrupts from being serviced. Therefore, interrupt service routines must be minimalist—perhaps even just waking and setting a flag to note their occurrence. The Non-preemptive Scheduler pattern provides a structure in which to keep interrupt service routines minimal and move application logic to sequentially executed "steps".

In most programming idioms, each subroutine or module, once started, will run to completion. A program that calculates π to one million decimal places, e.g., will hold the CPU captive for as long as the task requires. Multitasking operating systems avoid this problem by preemption. That is, they interrupt the task, save its state, and switch to a new task transparently. This allows other tasks to carry on working while the calculation is ongoing. Preemptive multitasking, however, is expensive (in terms of memory and CPU overhead) and may be difficult to support on low-power processors.

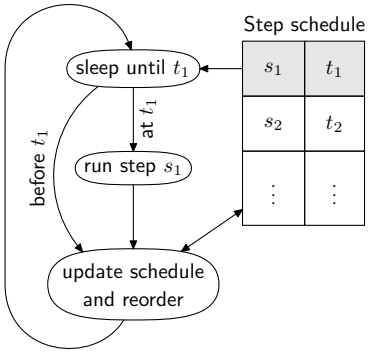

Figure 5. Overview of the Non-preemptive Scheduler Pattern, adapted from Cassandras and Lafortune [4]

A synopsis of the Non-preemptive Scheduling pattern is given in Figure 5 and Table I. The scheduler maintains a "step schedule" or list of active "steps" and their associated start times along with a register of all possible steps and their feasibility conditions. A *step* is a short-lived task. For example, beginning to send a message is a step, whereas the whole process of sending a message is a series of steps inter-linked by time delays or response to I/O requests. The scheduling process begins by taking the first item from the list of steps and sleeping until its start time. If the start time has arrived, the step is executed. If the start time has not yet arrived, it may be due to waking for another reason such as an interrupt (and thus the schedule may need updating). The last part of the scheduling loop is to update the schedule and sort it according to each step's start time. The schedule is updated based on asking every registered step if it is currently feasible based on the current state of the node. Scheduled but unfeasible steps are deleted, while unscheduled but feasible steps are added. The overall approach substantially reduces the need for application-level code within interrupt service routines. Race conditions are still possible but more easily avoided.

1) Extensions: A natural extension to the Non-preemptive Scheduler pattern is to (automatically or manually) recode procedural logic as a finite state machine allowing each procedural statement to be executed as a "step".

2) Examples: The best known example of this type of scheduling pattern in the WSN domain is TinyOS. There are a number of other systems that use a similar approach (such as the JACK agent programming environment [9] and the COLBERT robot programming language [12]). A common approach is to automatically rewrite the programmed code as a state machine (this is true for both the JACK and COLBERT languages).

The Nonpreemptive Scheduler pattern derives much of its design from Pont's patterns for time-triggered architectures [16] and Cassandras and Lafortune's description of timed automata [4].

F. Interval Listening Pattern

To function as a mesh network, individual nodes must be capable of acting as routers. In principle, this means

that they must be ready to receive messages at any time. In practice, such high alertness is generally only required when nodes are initially deployed or subsequently moved. For most installations, communication quickly stabilises into a predictable pattern based on regular sensing cycles and well established routing paths. Therefore, despite the need for nodes to act as routers, they can predict when the next message will arrive and revert to an ultra-low power mode until then.

The aims and triggers of the Interval Listening pattern are summarised in Table I.

To TinyOS developers, this may seem less like a pattern and more like a product that is taken off the shelf (i.e. Low Power Listening (LPL)). For other operating systems (or where no operating system is used), this mechanism is more likely to be handcrafted to suit the application. Furthermore, even when using TinyOS, it is worth considering other ways of performing interval listening that may give better performance than LPL.

1) Collaborators: While the Filter and Event Detection patterns can dramatically reduce the number of transmissions required, the real benefit in terms of energy savings is not made until the node can revert to low-power mode between transmissions. Therefore, if the Event Detection pattern is required, in all likelihood, the Interval Listening pattern will also be required.

2) Examples: One form of the Interval Listening pattern is implemented as LPL [14]. This protocol is a simple extension of standard TinyOS message transmission. It works by repeating the message continuously for X seconds or until acknowledgement is received. The receiver then only needs to wake up once every X seconds to listen for any transmissions. This simple modification substantially extends the life of each node.

The Time Synchronised Mesh Protocol (TSMP) [15] developed by Dust Networks is another approach to Interval Listening that is based on a combination of Time Division Multiplexing (TDM), where each node has a specific slot when it can transmit, and integrated time synchronisation that works by replying back to any sender how late or early their packet was. Note that the integrated time synchronisation is needed for two reasons: (a) to ensure that nodes wake at the right time to listen to neighbours, and (b) to avoid the need for top-down time synchronisation.

TSMP is potentially much more efficient than LPL since transmissions can be short and the node does not necessarily need to wake up as frequently as every second. TSMP is implemented in WirelessHART and is part of the ISA100 standard.

III. Conclusions

With wider adoption, the WSN domain is presently transitioning in much the same way as Computer Science transitioned towards Software Engineering in the past: from being a research-only domain that focused on optimising algorithms to being one that included a greater focus on the problem of developing reliable, functionally correct, useful and applicable systems. This naturally leads to greater consideration of the task facing WSN developers. Design patterns have revolutionised the way software is engineered. A similar revolution is needed in WSN engineering.

Examples throughout have shown that the patterns described in this paper appear repeatedly in reports on functioning deployed systems and are representative of key design ideas. If design patterns are taken up and further evolved by the WSN community, they will: i) lead to better concept reuse across platforms and operating systems; ii) guide systematic development across a broad range of applications; iii) shorten development cycles, and iv) ensure design flexibility, constraint mitigation, and application tailoring.

References

[1] M. Allen, L. Girod, R. Newton, S. Madden, D. T. Blumstein, and D. Estrin. Voxnet: An interactive, rapidly-deployable acoustic monitoring platform. In *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pages 371–382, 2008.

[2] Mike Allen. VoxNet: Reducing latency in high data-rate applications. In Gaura et al. [7].

[3] James Brusey, Elena Gaura, and Ramona Rednic. Classifying transition behaviour in postural activity monitoring. *Sensors & Transducers journal*, 7:213–223, October 2009. Online: http://www.sensorsportal.com/HTML/DIGEST/P_SI_98.htm.

[4] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[5] Geoffrey Challen and Matt Welsh. Volcano monitoring: Addressing data quality through iterative development. In Gaura et al. [7].

[6] Elena I. Gaura, James Brusey, and Ross Wilkins. Bare necessities—knowledge-driven wsn design. In *Proc. IEEE Sensors 2011*. IEEE, October 2011.

[7] Elena I. Gaura, Lewis Girod, James Brusey, Mike Allen, and Geoff Werner Challen, editors. *Wireless Sensor Networks: Deployments And Design Frameworks (Designing and Deploying Embedded Sensing Systems)*. Springer, 2010.

[8] Daniel Goldsmith and James Brusey. The Spanish Inquisition Protocol: Model-based transmission reduction for wireless sensor networks. In *Proc. IEEE Sensors*. IEEE, 2010.

[9] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents - summary of an agent infrastructure. In *Proceedings of the 5th International Conference on Autonomous Agents (Agents '01)*, 2001.

[10] Wen Hu, Nirupama Bulusu, Thanh Dang, Andrew Taylor, Chun Tung Chou, Sanjay Jha, and Van Nghia Tran. Cane toad monitoring: Data reduction in a high rate application. In Gaura et al. [7].

[11] John Kemp, Elena I. Gaura, James Brusey, and C. Douglas Thake. Using body sensor networks for increased safety in bomb disposal missions. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC2008)*, pages 81–89, Taichung, Taiwan, June 11-13 2008. IEEE Computer Society.

[12] Kurt Konolige. Colbert: A language for reactive control in sapphira. In *Proc. 21st Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pages 31–52, London, UK, 1997. Springer-Verlag.

[13] Kirk Martinez and Jane K. Hart. Glacier monitoring: Deploying custom hardware in harsh environments. In Gaura et al. [7].

[14] David Moss, Jonathan Hui, and Kevin Klues. Low power listening. Technical Report TEP 105, TinyOS Core Working Group, 2007.

[15] Kristofer S. J. Pister and Lance Doherty. TSMP: Time synchronized mesh protocol. In *Proc. IASTED Intl. Symp. Distributed Sensor Networks (DSN 2008)*, pages 391–398, 2008.

[16] Michael J. Pont. *Patterns for time-triggered embedded systems: building reliable applications with the 8051 family of microcontrollers*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2001.

[17] Bhaskaran Raman and Kameswari Chebrolu. Censor networks: A critique of "sensor networks" from a systems perspective. *ACM SIGCOMM Computer Communication Review*, 38(3):75–78, 2008.

[18] Geoff Werner Allen, Stephen Dawson-Haggerty, and Matt Welsh. Lance: Optimizing high-resolution signal collection in wireless sensor networks. In *Proc. 6th ACM conference on Embedded Network Sensor Systems (SenSys '08)*, pages 169–182, New York, NY, USA, 2008. ACM.