Software Update Recovery for Wireless Sensor Networks

Stephen Brown[1] and Cormac J. Sreenan[2]

[1] Department of Computer Science, NUI Maynooth, Ireland
stephen.brown@nuim.ie
[2] Mobile and Internet Systems Laboratory, University College Cork, Ireland

Abstract. Updating software over the network is important for Wireless Sensor Networks in support of scale, remote deployment, feature upgrades, and fixes. The risk of a fault in the updated code causing system failure is a serious problem. In this paper, we identify a single, critical, symptom *loss-of-control*, that complements exception-based schemes, and supports failsafe recovery from faults in software updates. We present a new software update recovery mechanism that uses loss-of-control to provide high-reliability, low energy, software updates, including a comparison of optimised-flooding against spanning-tree for determining loss-of-control in a multi-path environment. The solution presented supports a trial phase (with lower latency), and an operational phase (with lower energy). The energy/latency tradeoff of this is shown, and the high-reliability of this update recovery is demonstrated by analysis and simulation. The results presented control the risk in existing WSN software update mechanisms.

1 Introduction

Continuing advances in miniaturization and integration are making wireless sensor network (WSN) technology more realistic for deployment. Over-the-air software updating is an important feature for a number of reasons: high maintenance levels, development, and new software features[1][2][3].

A number of WSN software update mechanisms exist. But, in the field, there can be significant hesitancy to use these to perform software updates. This is due to the risk of system-wide failure, requiring expensive and time consuming manual recovery, or possible leading to complete loss of a sensor network. For example, the software update functionality was not used in [4] due to the risks, as the nodes were inserted deep into a glacier. During the development process, failsafe software updates can remove the need to manually reset the nodes, and replace parallel communication channels for development.

Traditional fault detection mechanisms, such as watchdog timers and exception handlers catch certain classes of software faults, but there are others classes which can prevent the nodes from participating in the WSN's future operation. In this paper we propose and evaluate a novel failsafe recovery mechanism based on detecting this single symptom: *loss-of-control*.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes our automated recovery model. This model uses high reliability broadcasting; Section 4 provides an analysis and simulation results for this. Section 5 describes a protocol to realise the model, and simulation results are provided in Section 6. Conclusions and future work are addressed in Section 7.

2 Related Work

Software update research is a key topic for wireless sensor networks[5]. The need to update the update mechanism itself, to reconfigure parameters, and to provide users with standardized utilities to manage online changes have also been identified, along with the need for overall update management[1].

There are a number of systems for updating wireless sensor network software 'over-the-air', with the focus to date on efficiently propagating the update to the nodes (e.g. ZebraNet[6], DelugeM[7], SensorScheme[8]). If the software update fails, then recovery mechanisms based on exceptions and watchdog timer are used to try and recover. This is the key risk: node loss will occur if management connectivity is lost *without* triggering one of these mechanisms. Given the self-evident high risk of failure immediately after activating a software update, this prompts the need for a fail-safe software update mechanism which includes automated recovery if management connectivity is lost. This work does not address the downloading of software updates, but rather the recovery from downloads which contain faults which would prevent further downloads.

There are two activities required to update software in a WSN: *propagating* the new software to all the nodes, and *activating* this software on each node. The benefits of separating these activities are explored in [2] and [3]. This separation is supported, for example, in Deluge with the 'injection' and 'reprogramming' activities. If the update fails, causing loss of network connectivity, then a third activity is required: *automated local recovery*.

3 Automated Local Recovery

To provide a low-risk environment for updating the software on the nodes in a WSN, every node must locally recover from faults in the updated software. (These faults are those that cause loss of management connectivity. Fault-tolerant data collection is not addressed here: it is an entirely separate matter from ensuring system recovery.) The keystone of failsafe recovery from faulty software updates is to be able to revert to a working software image, on each affected node, following loss-of-control. If a hardware errors occurs, but it allows connectivity to be maintained, then the node can be controlled (perhaps disabled) through network management. If not, the node will fallback to a known good software image, which may under some conditions help to re-establish connectivity. The price to be paid for a fail-safe fallback mechanism is false positives:

where temporary loss of connectivity causes a node to fallback. In this case, when connectivity is restored, the node can revert to the operational software image.

Having a maintenance image to reboot to when all else fails is an important element of node recovery (e.g. in [9]). But, to be effective, this depends on a mechanism to detect *when* it is needed, and to initiate its execution. In this paper, we introduce the term **loss-of-control** to describe the case where a software fault causes a loss of (management) connectivity from the management station to a node. Such loss of connectivity could be caused by various classes of faults in the updated code, but if the fault causes loss of management connectivity without triggering watchdog timeouts or exceptions, then each effected node is lost, and will require physical recovery. There are significant benefits in providing more complex recovery then just invoking the maintenance image[3] - for example, supporting fallback to the previously working version.

3.1 Identifying Loss of Control

We consider the case where a WSN is controlled by a management station via a gateway node. We assert that the *only* way a node can identify loss of control is whether it receives management traffic from the management station(s).

Accurately identifying loss-of-control requires a high-reliability, end-to-end communication path from the management station to every node in the WSN, over unreliable and asymmetric links. Data paths are not suitable: they are mainly in the opposite direction, they may not be periodic to every node, and generally do not have a very high level of guaranteed delivery: this precludes piggybacking on data traffic or acknowledgements. Also, when nodes are running the maintenance image, dedicated connectivity traffic would still be required.

The automated recovery mechanism described here uses periodic traffic from the management station, referred to as *beacons*. This is similar to a keep-alive *heartbeat* used in many embedded systems, but unreliable wireless links, and the constraints of wireless sensor nodes (e.g. energy, memory) make achieving high-reliability operation more difficult. Unlike the Trickle/Drip distribution method used in SNMS [9], designed to eventually update all nodes, it is important that nodes do *not* re-broadcast messages periodically: this would propagate out-of-date connectivity information. Also, protocols like Trickle do not provide the very high per-node reliability needed to ensure high system-wide reliability. Well established mechanisms for distributing network state, such as OSPF, are too heavyweight for infrequent communications in a WSN environment, both in terms of traffic and data storage. For example, every node maintains a full copy of the network-wide database. They also rely on low loss rates to establish stable, bi-directional connections with neighbours. For these reasons, many authors have argued that reactive protocols, such as described in this paper, are more suitable in a low traffic WSN environment than proactive protocols such as OSPF.

We assert that there is no way, in an ad-hoc and multi-path environment, for any node (or local group of nodes) to determine whether loss of connectivity is due to a single node or a system-wide failure. Thus every node that loses

connectivity must take action, whether it can communicate with other local nodes or not.

3.2 Recovery Action

There are two principle actions that can be taken to recover a node (and eventually an entire WSN). The first is to fall back to an 'known good software' image. If this fails (or is not available) then the second action is to fall back to a maintenance image (e.g. Golden Image [10]). This needs a high level of reliability, and must provide at a minimum support for downloading and activating software updates.

If the connectivity failure is due to faults in the software update, then recovering to 'known-good' software restores network management connectivity (allowing further software updates to be downloaded/activated). If the connectivity failure is due to other causes (e.g. temporary loss of radio connectivity) then recovery returns the network to a state where, once the external factor is removed, it will be manageable. Data collection may be interrupted during recovery: so long connectivity timeouts are required, and the recovery mechanism should be used a last resort - other mechanisms (such as rerouting around effected areas) should activate on a shorter timescale.

The minimum hardware/memory requirement is for a full-size image and a smaller maintenance image (with software updates performed from the maintenance image). If a node can store two full-sized images, then the software updates can be performed during operation, reducing disturbance to the application. This is reasonable: a MICAz mote has 128KB internal/512KB external flash; the TmoteSky has 48KB/1MB. A maintenance image might take up an estimated 32KB (e.g. Golden Image at c. 24KB). Where the image is stored will depend on the software download protocol and bootstrap loader.

3.3 Two-Phase Approach

It is likely that a WSN will exhibit early software update failures (for reasons explored in [3]). This motivates a two-phase approach to save energy. Software updates are initially run on a *trial* basis with a high frequency of monitoring providing for quick recovery, but using more energy. Following a successful *trial*, the same software version continues to execute, but on an *operational* basis, with a lower frequency of monitoring, providing slower recovery, but lower energy use.

3.4 State Machine

A node can be running in one of four **modes** from a software update and recovery viewpoint - see Figure 1. The TRY, RUN, and RTM commands are issued over the recovery protocol: TRY(x) starts a trial with software version x; RUN(x) starts operational use; and RTM initiates a Return to the Maintenance Image. As shown with the bold arrows in Figure 1, in normal use a node will startup in

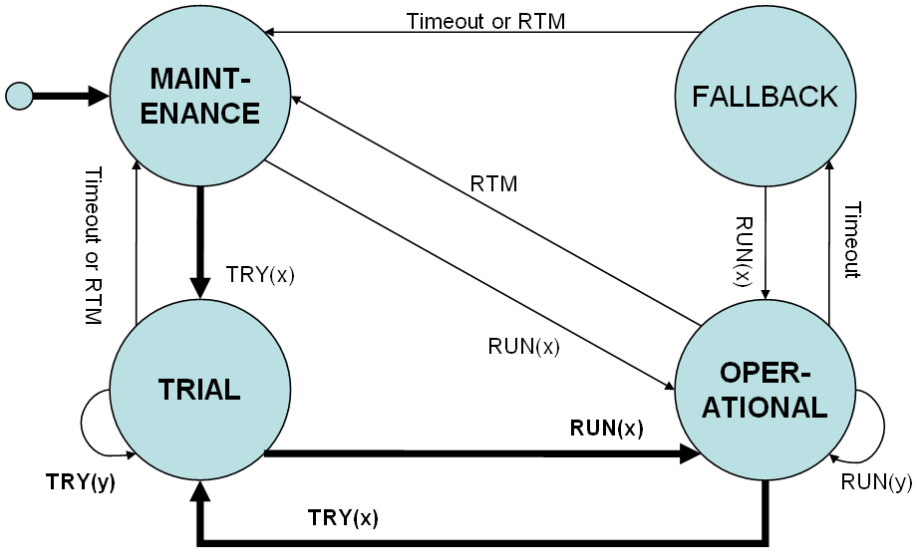


Fig. 1. Recovery State Machine

MAINTENANCE mode; and, for the first software update, will be transitioned to TRIAL mode, and then to OPERATIONAL mode. Subsequent software updates will be run initially in TRIAL mode and then in OPERATIONAL mode. In each state, a node will be running the software image as shown:

- MAINTENANCE: running the maintenance image;
- TRIAL and OPERATIONAL: running the specified software (version "x");
- FALLBACK: the previous software image, or the maintenance image.

The inclusion of a version id causes all (connected) nodes to try and activate the same version. It is regarded as a feature of the software download propagation functionality to ensure that this version is available. Protocols such as DHV[17] address the problem of ensuring that all the nodes are running the same software version following network partitioning, etc.

The "Timeout" events are raised by loss-of-control (i.e. beacon timeouts). The next section presents an investigation into reliably determining this.

4 High-Reliability Polycasting

The term *polycasting* is used in this paper to mean the propagation of traffic from a single node to all other nodes in the WSN (i.e. network-layer broadcasting); this allows clear differentiation from the term *broadcasting* which is used at the MAC layer. For each node to determine connectivity, a beacon must be polycast periodically from the management station (via a gateway). To provide a low level

of unnecessary recovery, this propagation must have a high level of reliability. For example, if the maximum allowed probability of an unnecessary recovery in a 100-node WSN within 1 year is 5%, and connectivity traffic is polycast once a day, then the maximum connectivity error rate for each node is 0.0000014.

In this paper we focus on multi-path physical network topologies; these are common for wireless sensor networks due to the robustness properties that they present. The overlay network may form, for example, a spanning-tree, or linear, or grid, or clustered topology - but in general nodes will be placed so as to allow for alternate paths to cope with link or node failure.

Two algorithms for high-reliability polycasting are Flooding and Spanning Trees. In [11] it is shown that Spanning Trees perform better for some metrics (i.e. count of relay nodes); here we show that, with energy consumption as a metric, optimised flooding can provide higher reliability for lower cost. Flooding has the additional energy benefit of not requiring any setup or maintenance. For generality, link-quality data and node position data is not known *a priori*. In both cases a random transmit delay timer ("jitter") is used to reduce medium contention. Optimizations with high overheads for low traffic levels (e.g. Clustering[12]) are not considered as candidate solutions: see [18] for a comparison of Clustering and Spanning-Tree approaches in WSNs. Also optimisations that would impact application traffic (e.g. power modifications[13]) are not considered.

4.1 Minimum Spanning Tree Algorithm

The Minimum Spanning Tree (MST) algorithm used for comparison was based on Bellman-Ford. MAC broadcast messages are used to minimize transmissions for data to multiple children, and overheard data packets provide free acknowledgements using the "wireless broadcast advantage" ([14]) to parent nodes, with onyl leaf nodes sending an explicit acknowledgement. Link weight is expected transmissions to the root node, calculated from collected link-quality data.

4.2 Flooding Algorithm (FDMT)

Flooding with **D**uplicate-suppression, **M**issing packet regeneration, and **T**imeout (FDMT) removes the broadcast storm problem. A *beacon* is sent as a series of packets in a burst, with a burst ID (Burst Index) and a packet-in-burst counter (Burst Size). This allows each packet to be uniquely identified - only the first copy received of any packet is re-broadcast (Duplicate Suppression), using the burst ID and packet-in-burst counter. When missing packets are identified, from gaps in the received packet-in-burst counter, the node regenerates them (Missing Packet Regeneration). A lifetime field allows the burst IDs to be reused. The fundamental novelty of this algorithm is in using controlled duplicate (multi-path) reception in order to provide very high reliability for one-to-many trafic in an ad-hoc wireless network. See Table 3 for the protocol fields.

4.3 Analytical Comparison

To demonstrate the benefits of FDMT (higher reliability for lower energy cost) we compare the results for a 4x4 grid of 16 nodes (see Figure 2) with the FDMT and MST algorithms described above. The root in both cases is at the top left hand corner, and the link quality/packet-receive-rate (PRR) p_j^i is shown (reflecting noise, distance, and interference-related packet loss). The probability

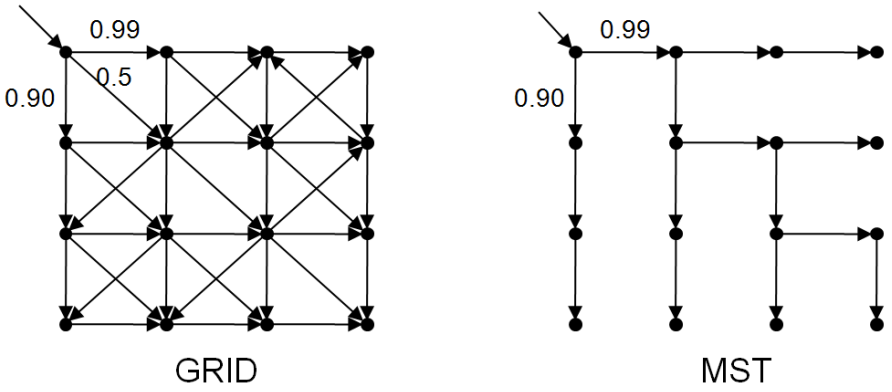


Fig. 2. 4x4 Grid Data Flows

of system failure is determined by the equation shown, where $Pr^i(success)$ is the probability that node i receives at least one packet from P packets injected in a network of N nodes:

$$Pr(systemfailure) = 1 - \prod_{n=1}^{N} Pr_i(success) \tag{1}$$

For the spanning tree, the probability of success is determined as follows:

$$Pr_i(success) = Pr_{parent}(success) * (1 - (1 - p_i^{parent})^P) \tag{2}$$

And for FDMT:

$$Pr_i(success) = 1 - \prod_{n=1}^{n=N} (1 - (Pr_n(success) * (1 - (1 - p_i^n)^P))) \tag{3}$$

The results of applying these equations to the 4x4 grid are shown in Table 1 for increasing P, showing the probability of system failure and energy cost (Tx count) in each case.

FDMT provides higher reliability for lower cost. For P=1, using FDMT each node only transmits once (reliability is attained through multiple receptions);

Table 1. MST vs FDMT

Packets	MST		FDMT	
P	Pr(Fail)	Tx Count	Pr(Fail)	Tx Count
1	0.316934	18.4	0.293000	16
2	0.100447	36.8	0.085849	32
3	0.031835	55.1	0.000003	48
4	0.010090	73.5	6.73E-12	64
5	0.003198	91.9	1.08E-14	80
6	0.001013	110.3	1.75E-17	96

but using MST, each node must reliabily transmit both an *ack packet* to its parent, and a *data packet* to each of its children in the tree. So, for example, to achieve a failure rate $<$ 1-in-1000, costs 110 transmissions for MST, and 48 for FDMT. Or, for an energy cost of 48-55 transmissions, MST provides a failure rate of $32,000$ ppm, and FDMT a significantly lower failure rate of 3 ppm.

4.4 Experimental Comparison

To validate these results with a more realistic radio model, and a larger network, the two algorithms were simulated using TOSSIM [15], with the default TinyOS MAC (details in Section 6). This radio model introduces significant traffic losses due to both noise and interference. Similar results (not included for space reasons) are seen for random topologies with the same average node densities as for the three topologies shown.

A random transmit delay after reception of a new packet is used to reduce medium contention - the maximum value is specified in the "Jitter" field. A minimum inter-packet transmission interval of 4*Jitter is used. A Jitter value of 675mS was used: 225 slots of 3mS each (large anough for an average transmit). Experiments have shown that this value achieves a reception rate over 99.9% for all densities with 225 nodes.

The results of the simulation (measured over 100 runs for FDMT, 20 runs for MST) are shown in Table 2, showing the system reliability (Reliability) and MTTF (Mean Time To Failure). The cost is the average per-node transmit count, and the latency is the time from the first beacon injection, to full coverage (reception by every node). Note that, as for the analytical case, FDMT provides a higher reliability/cost ratio than MST.

Notes:

- if latency is used as an indication of cost, representing required receiver on time, FDMT shows an even greater advantage;
- the Spanning Tree Tx figures show the transmit count required to build the spanning tree;
- achieving this reliability for the spanning tree required, on average, up to 23 re-transmissions - this is a significant contributor to the large latencies.

Algorithm	FDMT			MST		
Density	Tight	Medium	Sparse	Tight	Medium	Sparse
System Reliability	100.0%	99.999%	99.55%	99.97%	99.92%	99.84%
System MTTF	9.75×10^{12}	1.93×10^{5}	2.42×10^{2}	3.93×10^{3}	1.33×10^{3}	6.28×10^{2}
Avg. Node Tx	1	2	4	5.17	4.06	4.54
Latency [S]	0.051	0.444	1.633	9.1	19.05	36.8
Spanning Tree Tx	0	0	0	29	29	29

Table 2. MST vs FDMT

For FDMT the key parameter is the burst size (number of packets sent in a burst). This is set by monitoring the minimum-duplicates feedback field (see Section V). In our experiments, modifying the burst size to produce a minimum value of 5 to 6 duplicates provides the high reliability shown.

For the Spanning-Tree, the key parameter is the per-link reliability: only one packet is injected as retransmissions are used to guarantee reliability. The per-link reliability is used to calculate the maximum retransmit count per link, based on the measured round-trip link quality (used as the weight for building the minimum cost spanning tree). A per-link reliability figure of 0.999999 (i.e. 1 failure per million) was used to achieve the above results - this produces an average system reliability of $0.999999^{225} = 99.978\%$ (MTTF=225 bursts).

As in the analytical case, the results demonstrate that FDMT provides higher reliability with lower cost. FDMT introduces no setup overhead, and has reduced RAM requirements: 64 bytes vs approx 1KB in our implementations.

5 Protocol Overview

The protocol packet definition used to evaluate recovery is shown in Table 3. This implementation contains 23 bytes, fitting into a standard TinyOS packet. The management station periodically polycasts "Activation and Recovery" PDU's into the network. The (wrapping) sequence number uniquely identifies the packet, and is timed out by the lifetime (to allow for safe wrapping) which is decremented on every hop. Burst Size and Burst Index are used to regenerate missing packets. Mode-ID is the mode sequence number; it is changed for each new Mode command. Mode specifies what state the node should take (reference Figure 1). A burst is used, instead of single packets, to provide better reliability within a power-cycled environment (higher reliability can be achieved with one cycle); it also allows separation of the required recovery time and the beacon injection period (by allowing multiple packets to be injected per period). A burst also provides quicker feedback to the management station (later packets in the burst will piggyback feedback to the management station from more remote nodes in the network).

Table 3. Protocol Field Summary

Field	Description
Sequence Number	unique ID for each burst (wrapping)
Lifetime	timeout the wrapping Sequence No.
Jitter	random delay to reduce media contention
Burst Size	packets in the burst
Burst Index	packet burst position - for regeneration
Mode-ID	unique ID for the mode field (wrapping)
Mode	the mode to run the software in
Software-ID	the software version to run
LoffOfControl Timeout	timeout for "loss of control"
Recovery Timeout	from MAINT to FALLBACK
Flags	Flag0=Failure Seen Bit, set after recovery
Max Hops	maximum hops seen
Min life	minimum lifetime seen, used to set Lifetime
Min Dups	minimum duplicates seen

6 Simulation

The protocol is simulated, both to demonstrate its correct operation, and also to measure the reliability and cost in an environment that allows direct access to each node to collect data. The protocol and state machine were simulated using the TOSSIM 2 simulator. Results for 225 node, 15x15 grid configurations ("tight", "medium", and "sparse") are shown here representing networks with the characteristics shown in Table 4 using the simulator's default parameters[1]. The radio model uses the log-normal shadowing path loss model, and provides a realistic model of real-world conditions[16] with significant variability in packet reception rates as shown in Figure 3. The topologies span a wide range of network densities (from very tightly connected to very weakly connected), and provide for validation of the protocol across this wide range, and also show the impact that the network density has on the protocol performance characteristics.

6.1 Behavioral Results

The correct operation of the protocol is demonstrated here, showing 100% completion, and the latency times. Figure 4 shows the behavior of a successful software trial followed by operational use of the new software version for a medium density network. Initially the network is running the maintenance image. At

[1] See www.tinyos.net for details

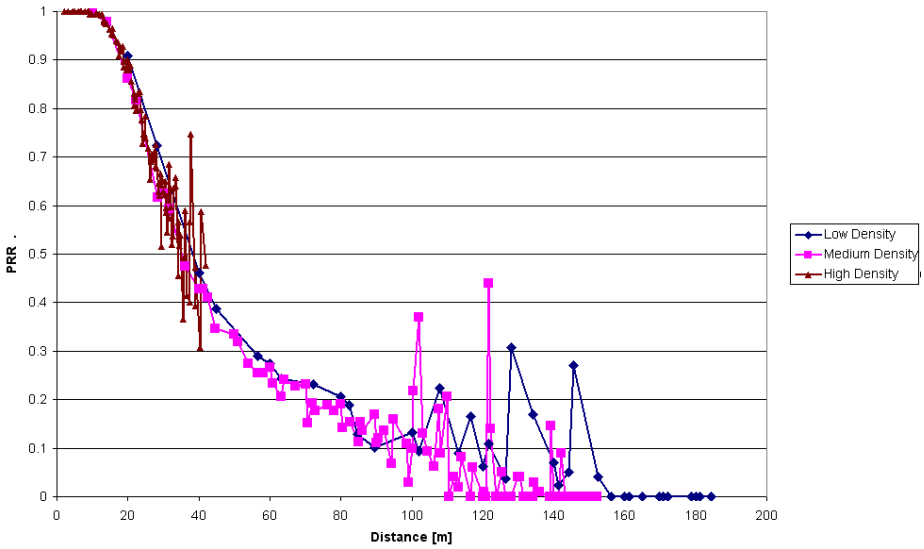


Fig. 3. PRR vs Distance

Table 4. Network Parameters

	Sparse	Medium	Tight
Spacing	20m	10m	2.2m
Nbrs.	9.17	30.01	191.37
Gain	-108±9dB	-110±9dB	-90±9dB
Noise	-105±1.8dB		
Power	0dBm		

time=20 a "Try" command is inserted into the WSN via the gateway, and at time=30 a "Run" command is inserted. In practice, it is likely that a trial would last for multiple cycles of the WSN operation, possibly in the order of days rather than seconds. Figure 5 shows the behavior of a recovery during a trial following a software failure (with quick fallback, using the failure-seen flag) for a medium density network. The center node (112) detects loss of connectivity at time 180, and propagates the failure-seen flag quick, network-wide, automatic recovery.

6.2 Estimating Long-Term Reliability

Long-term reliability is measured as the probability that all the nodes receive a beacon before timing out. The probability that no packets would have been received by the node is calculated as follows, where c is the number of packets received by node n from node i:

$$Reliability = Pr(success) = 1 - \prod_{n=1}^{n=N} \prod_{i=1}^{i=N} (1 - p_i^n)^{c_i^n} \quad (4)$$

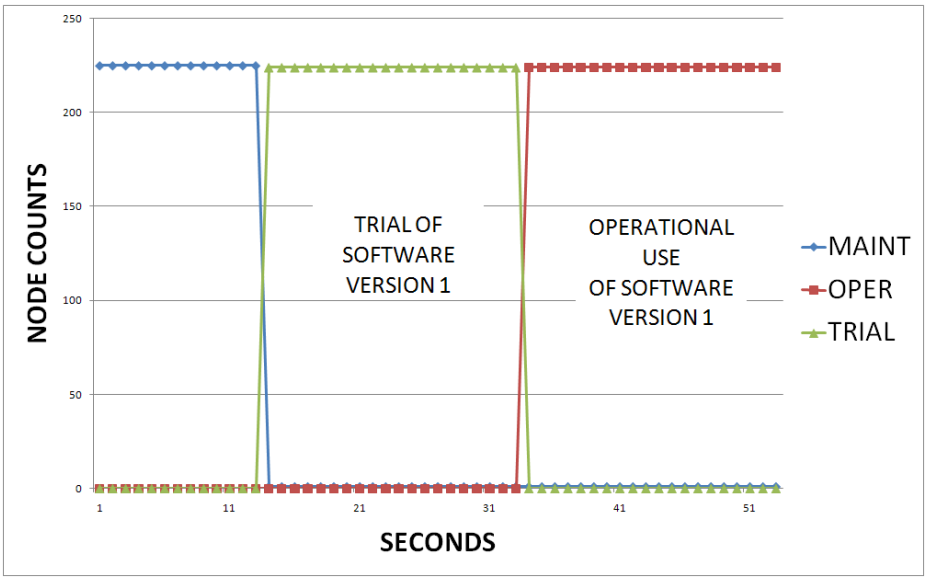
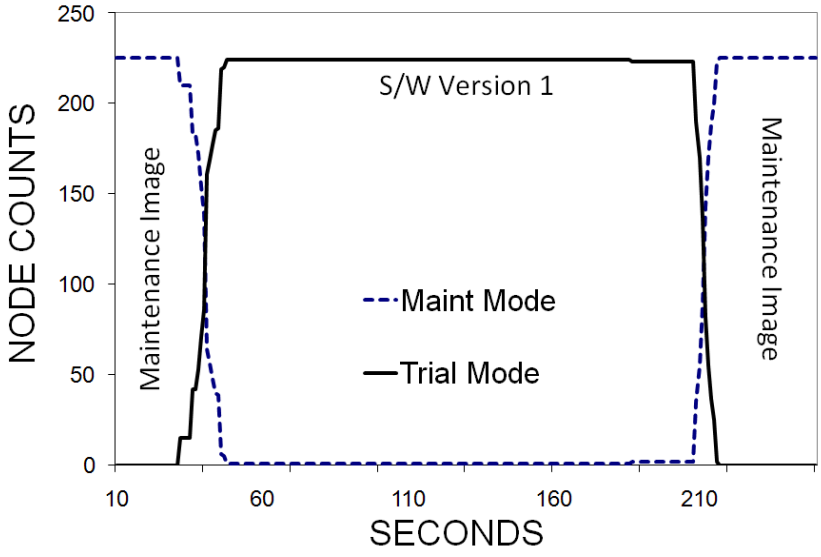


Fig. 4. Successful Trial

Fig. 5. Automatic Fallback



The PRR p_i^n is measured during each simulation. This method allows the reliability of the system to be estimated from a relatively small number of runs. The

Table 5. System Reliability

	Tight	**Medium**	**Sparse**
Reliability	1.0	1.0	0.999994
99.9% CI	<1.0E-9	1.81E-9	5.44E-6
MTTF	>1000000	>1000000	166666.7

results in Table 5, averaged over 450 runs, show the high reliability achieved. The 99.9% Confidence Interval (CI) was calculated using the Student-t distribution. The number of runs is based on producing a largest CI in the order of 10^{-6}.

6.3 Propagation Delay

The propagation delay results (measured as the time from initial injection to 100% reception) are shown in Table 6 averaged over 93 runs, showing the min, max, and average in seconds (with 99.9% Confidence Intervals calculated using the Student-t distribution). The results show that higher network densities have both lower and less variable propagation times, and that reasonable propagation times can be achieved. Figure 6 shows a sample propagation delay pattern - the

Table 6. Average Propagation Delays [Seconds]

	Density		
Delay	**Tight**	**Medium**	**Sparse**
Min.	0.035	0.369	1.5
Avg.	0.285	0.743	2.5
99.9% CI	0.015	0.023	0.16
Max.	0.6	1.2	9.3

X and Y axes are spatial dimensions (150m from side to side), and the gateway node is in the bottom left corner. Note that the unreliable wireless links lead to both peaks (nodes that receive the beacon much later than their neighbours) and troughs (beacons that receive an early beacon over long-distance, low-reliability links).

6.4 Recovery Latency

The "failure-seen" flag provides feedback to the management station that a connectivity failure has been seen and recovered from by some nodes. For a "Quick-Trial", no reaction is required: the network will automatically fallback.

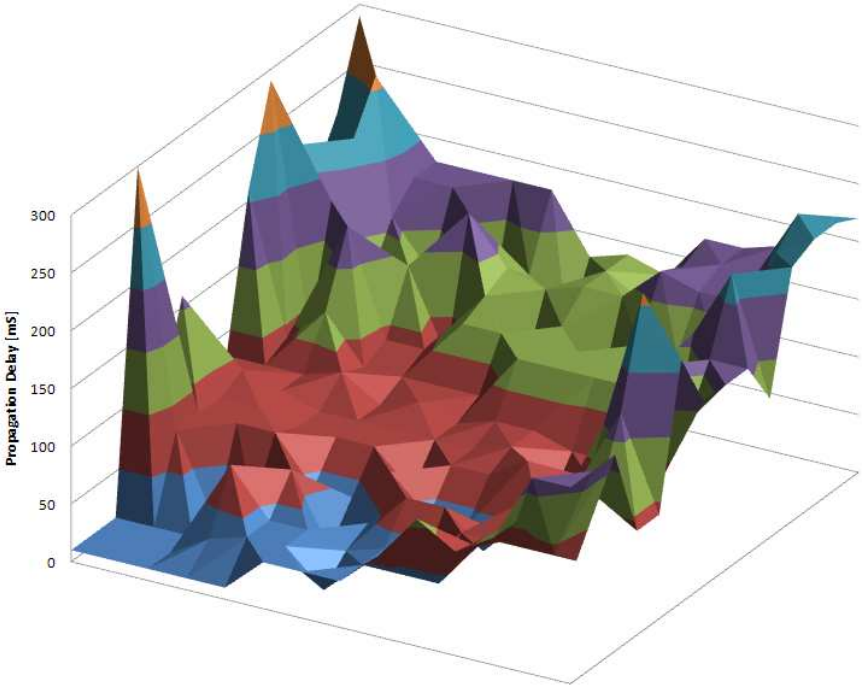


Fig. 6. Propagation Pattern

In "Operational" use, the management station may query the nodes (using a management protocol such as SNMS [9]), or issue further Software Activation commands to recover the network.

Table 7 shows the recovery latency for a failed "Quick" software trial, measured between a (simulated) software failure occurring and: that node identifying loss of connectivity (T_1), all the nodes automatically recovering (T_2), and the fail flag seen at the management station (T_3). The failure was simulated on the center node (112); the burst frequency was 20 seconds. The results are averaged over 20 runs, and are shown in seconds (with ± 99.9% Confidence Intervals shown in brackets). Table 8 shows the performance for an Operational software

Table 7. Recovery Latency [Seconds]

	Tight	**Medium**	**Sparse**
$\mathbf{T_1}$	19.0 (±0.0)	19.3 (±0.13)	19.4 (±0.67)
$\mathbf{T_2}$	23.6 (±0.43)	25.4 (±0.34)	31.2 (±0.92)
$\mathbf{T_3}$	24.0 (±0.0)	25.0 (±0.33)	33 (±1.01)

failure in terms of the times between the simulated software failure occurring

Table 8. Reporting Latency [Seconds]

	Tight	Medium	Sparse
$\mathbf{T}_4$	19.0 (±0.00)	19.3 (±0.13)	19.4 (±0.67)
$\mathbf{T}_5$	24.0 (±0.00)	25.0 (±0.33)	33.0 (±1.01)

and (a) the failed node identifying loss of connectivity (T_4), and (b) the fail flag seen at the management station (T_5). The failure was simulated on the center node (112), with a burst frequency of 20 seconds, with results averaged over 20 runs, and shown ± the 99.9% Confidence Interval in seconds. Note the close correlation of results with the separate set of experiments shown in Table 7.

6.5 Energy Use

This activation and recovery protocol runs during powered-up cycles of a power-cycled WSN, and thus works with the synchronisation mechanism in use (for downloading software updates, and network management, as well as for the sensornet application). This implies that no additional energy is required to receive packets (typically transceiver power use is the same whether receiving or not), so the incremental power used by the protocol is measured in transmits. The low latency requires little extension to the power-up phase of the duty cycle. If an alternative power saving approach, such as wake-on-wireless or preamble sampling, is being used, the energy cost would be directly proportional to the number of transmits. Figure 7 shows the energy use under normal operation (in terms of packets transmitted): the average is one packet transmitted per node per packet injected. This graph is extracted from a long run (10,000 beacons), and shows 22 beacon cycles, with a burst-size of 2. This could represent, for example, 22 days with one sequence per day (providing for a connectivity timeout value of 2 days), or 22 hours with one sequence an hour (providing for a 2-hour connectivity timeout). Energy use is linear with time, and with network size. If transceiver on time is used as the energy metric, then the power requirement would be in the order of 1 second per day for a medium network (based on a measured maximum of 0.8S).

Figure 8 shows the energy use of recovery following a failed software trial. There is higher energy use during the software trial operation, due to the higher frequency of beacon sequences, and a lower latency for identifying failure (and recovering). The energy use per beacon sequence is the same as for normal operation, but additional energy is expended during the automated recovery (steeper slope of the energy line) reflecting the propagation of the fail flag.

6.6 Energy vs Latency Tradeoff

The energy/latency tradeoff is shown in Figure 9 for a burst size of 2 beacons, and using the Mica2 transmit power level of 0dBm=1mW with an average broadcast transmitter on time of 2.1mS per packet (as measured in the simulator). The

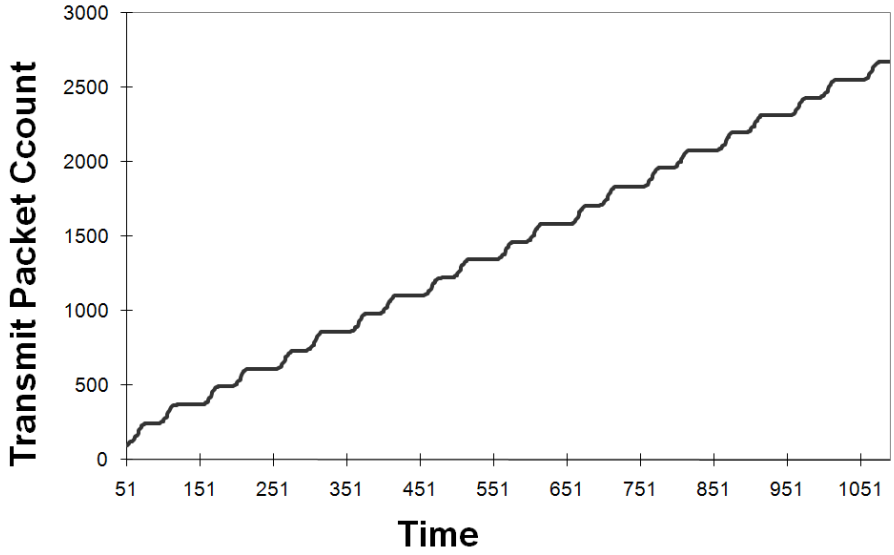


Fig. 7. Energy Use, Normal Operation

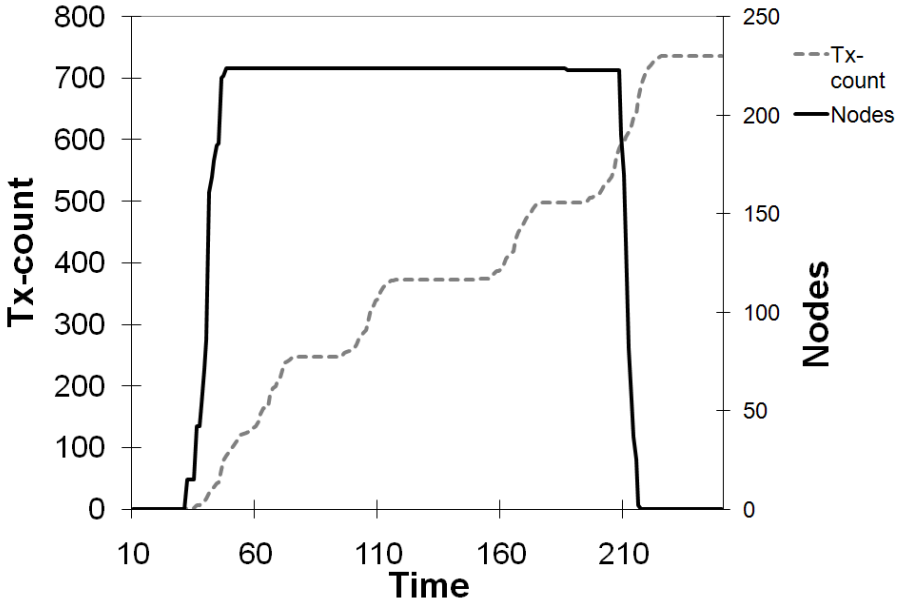


Fig. 8. Energy Use, Quick-Trial Recovery

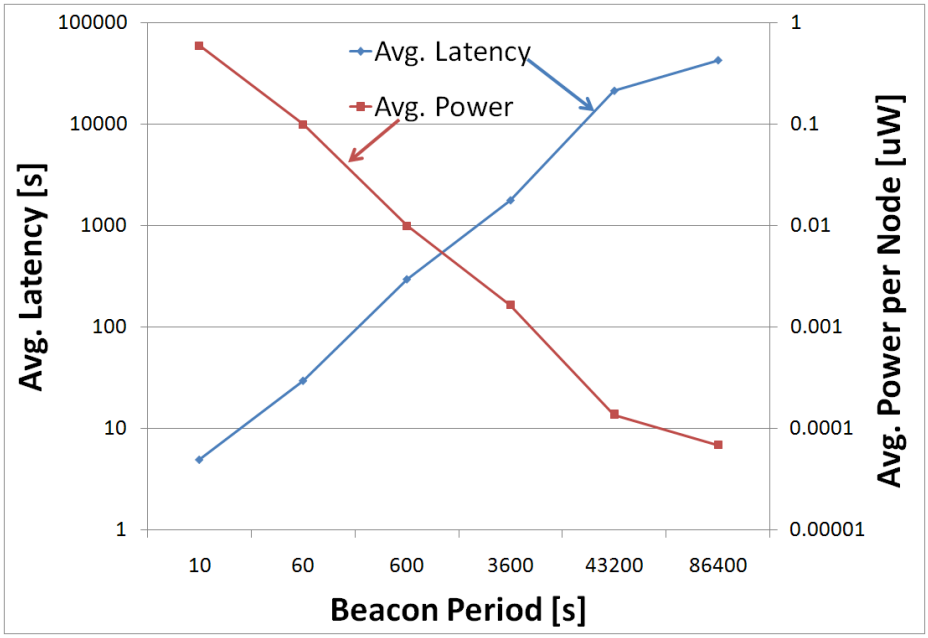


Fig. 9. Energy/Latency Tradeoff

latency is for a node to identify loss-of-connectivity. This shows the benefit of two-phase operation: low latency/high power usage software trials (left), and high latency/low power operational use (right).

6.7 Feedback

Feedback is propagated for free by piggy-backing data (Table 3) on the beacons. An example, using a sparse network, is shown in Figures 10 and 11 over two runs. The figures show that the actual number of hops (hmax-actual) is reported more accurately to the management station (hmax) as the burst size increases, and that the actual number of duplicates (dmin-actual) is also reported more accurately (dmin) with the burst size.

The burst size is modified until maximum hops (hmax) value stabilizes, and then the minimum duplicates value (dmin) used to refine the burst size. Actual values as well as reported values are shown for comparison. Minimum duplicates between 5 and 10 works well for the wide range of network densities simulated.

7 Conclusions and Future Work

In this paper we present a software update safety net that reduces the risk of WSN loss during software updates, and we demonstrate its effectiveness through

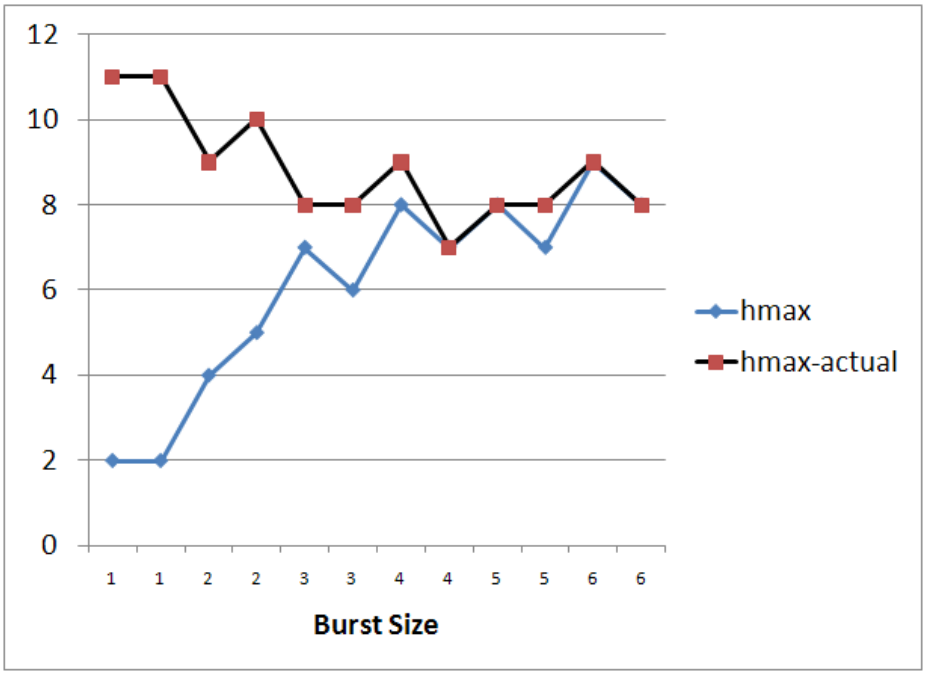


Fig. 10. Maximum Hops vs Burst Size

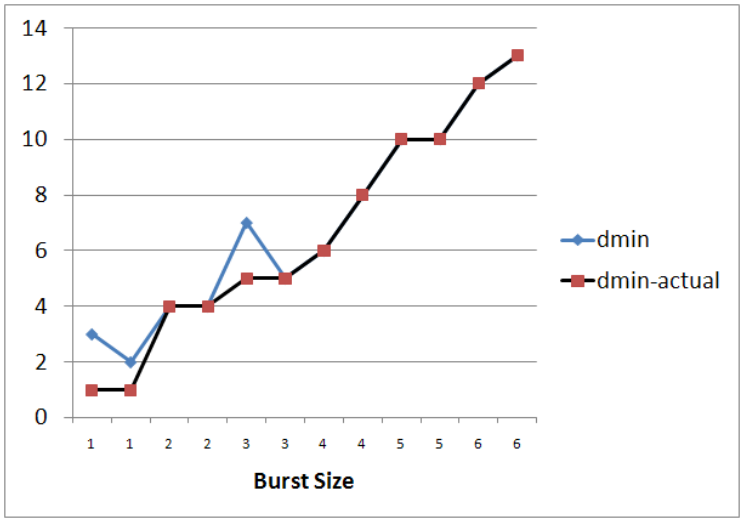


Fig. 11. Minimum Duplicates vs Burst Size

simulation. We show a new result: that an optimised flooding approach provides better efficiency and responsiveness than a spanning-tree for very high reliability polycasting. Our results show the energy costs of using the software update and recovery mechanism, and evaluate the energy/latency tradeoff. Using the same mechanism for both propagating update commands, and also measuring management connectivity, is shown to be effective. The protocol provides the update recovery mechanism that we have argued is an essential part of real-world wireless sensor networks.

The two phases of operation of a software update, *Trial* and *Operational*, allow the conflicting requirements for a quick, network-wide recovery in the one case, and a low-energy, node-specific recovery in the other case, to be resolved. This allows for quick recovery of the network when a software update fails soon after its deployment, at the cost of higher energy use. And for better continuation of operation of the set of working nodes, while still guaranteeing eventual recovery, with lower energy use in normal operational use of the sensor network.

Future work includes simulation and real-world assessment on a range of network densities and topologies, and integration with existing software update mechanisms. This recovery might also be used for *communication* nodes in a hierarchical networks, with a simplified version for the *sensing* nodes (e.g. IEEE802.15.4 full and reduced function devices).

In summary, by providing for fail-safe software update recovery, which significantly reduces the risk of losing nodes due to faulty updates, this work provides the basis for high confidence in using over-the-air software updates. This is an important element of real-world deployment, and will make WSNs more flexible and supportable in practice.

References

1. C.-C. Han, R. Kumar, S. R., and M. Srivastavam. Sensor network software update management: a survey. Intl. Journal of Network Management, 15, pp. 283–294 (2005)
2. Q. Wang, Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: challenges and approaches. IEEE Network, 20(3), pp. 48–55 (2006)
3. S. Brown and C. Sreenan. A new model for updating software in wireless sensor networks. IEEE Network, 20(6), pp. 42–47 (2006)
4. P. Padhy, K. Martinez, A. Riddoch, H. L. R. Ong, and J. K. Hart. Glacial environment monitoring using sensor networks. In Proc. RealWSN'05, pp. 10–14 (2005)
5. N. Kothari, K. Nagaraja, V. Raghunathan, F. Sultan, and S. Chakradhar. Hermes: A software architecture for visibility and control in wireless sensor network deployments. in Proc. IPSN'08, pp. 395–406 (2008)
6. T. Liu, C. Sadler, and M. Zhang, P. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet. In Proc. MobiSys'04, pp. 256–269, ACM (2004)
7. Z. Xiao and B. Sarikaya. Code dissemination in sensor networks with mdeluge. In Proc. SECON'06 (2), pp. 661–666, IEEE (2006)
8. L. Evers, P. Havinga, and J. Kuper. Flexible sensor network reprogramming for logistics. In Proc. MASS 2007, pp. 1–4. IEEE (2007)

9. G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In Proc. EWSN'05, pp. 121–132. IEEE (2005)
10. P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In Proc. IPSN 2006, pp. 407–415, IEEE (2006)
11. B. Williams and T. Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In MobiHoc '02, pp. 194–205, ACM (2002)
12. C.-K. Liang, Y.-J. Huang, and J.-D. Lin. An energy efficient routing scheme in wireless sensor networks. In Proc. AINAW 2008, pp. 916–921 (2008)
13. N. Rahnavard, B. Vellambi, and F. Fekri. Distributed protocols for finding low-cost broadcast and multicast trees in wireless networks. In Proc. SECON'08, pp. 551–559, IEEE (2008)
14. J. Wieselthier, G. Nguyen, and A. Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In Proc. INFOCOM 2000 (2), pp. 585–594, IEEE (2000)
15. P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proc. SenSys '03*, pp. 126–137, ACM (2003)
16. M. Zuniga and B. Krishnamachari. Analyzing the transitional region in low power wireless links. in Proc. SECON'04, pp. 517–526, IEEE(2004)
17. T. Dang, N. Bulusu, W. Feng, and S. Park. DHV: A Code Consistency Maintenance Protocol for Multi-hop Wireless Sensor Networks. in Proc. EWSN 2009, pp. 327–342, Springer-Verlag (2009)
18. H.O. Tan and I Korpeoglu. Power Efficient Data Gathering and Aggregation in Wireless Sensor Networks. SIGMOD Record, Vol. 32, No. 4, Dec., pp. 66–71, ACM (2003)